Scaling up Stochastic Gradient Descent for Non-convex Optimisation

Saad Mohamad
WMG, University of Warwick
Coventry, UK

Abdelhamid Bouchachia
epartment of Computing,
Bournemouth University
Poole, UK

Giovanni Montana
WMG, University of Warwick
Coventry, UK

ABSTRACT

Stochastic gradient descent (SGD) is a widely adopted iterative method for optimizing differentiable objective functions. In this paper, we propose an approach for scaling up SGD in applications involving non-convex functions and large datasets. We address the bottleneck problem arising when using both shared and distributed memory. Typically, the former is bounded by limited computation resources and bandwidth whereas the latter suffers from communication overheads. We propose a distributed and parallel implementation of SGD (DPSGD) that relies on both asynchronous distribution and lock-free parallelism. By combining two strategies into a unified framework, DPSGD is able to strike an optimal trade-off between local computation and communication. The convergence properties of DPSGD are studied for non-convex problems such as those arising in statistical modelling and machine learning. Our analysis shows that DPSGD leads to speed-up with respect to the number of cores and number of workers while guaranteeing an asymptotic convergence rate of $O(1/\sqrt{T})$ given that the number of cores is bounded by $T^{1/4}$ and the number of workers is bounded by $T^{1/2}$ where T is the number of iterations. The potential gains that can be achieved by DPSGD are demonstrated empirically on a classification task with convolutional neural networks.

KEYWORDS

Stochastic Gradient Descent, Large Scale Non-convex Optimisation, Distributed and Parallel Computation, Deep Learning

1 INTRODUCTION

Stochastic gradient descent (SGD) is a general iterative algorithm for solving large-scale optimisation problems such as minimising a differentiable objective function $f(\boldsymbol{v})$ parameterised in $\boldsymbol{v} \in \mathcal{V}$ and can be written as $f(\boldsymbol{v}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{v})$ i.e.

$$\min_{\boldsymbol{v}} f(\boldsymbol{v}) \tag{1}$$

In several statistical models and machine learning (ML) algorithms, $f(\boldsymbol{v}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\boldsymbol{v})$ is the empirical average loss where each $f_i(\boldsymbol{v})$ indicates that the function has been evaluated at a data instance $\boldsymbol{x}_i$. SGD updates $\boldsymbol{v}$ using the gradients computed on some (or single) data points. In this work, we are interested in problems involving non-convex objective functions such as artificial neural networks [4, 5, 13, 20, 21] amongst many others. Non-convex problems abound in ML, and are often characterised by a very large number of parameters (e.g. deep neural nets), which hinders their optimisation. These challenges are often compounded by the sheer size of the available datasets, which can be in order of millions of data points. As the size of the available data increases, SGD becomes computationally inefficient because the data samples are processed sequentially. The need for scalable optimisation algorithms is shared across application domains.

Many studies have been proposed to scale-up SGD by distributing the computation over different computing units benefiting from the advances in hardware technology. The main existing paradigms exploit either *shared memory* or *distributed memory* architectures. While shared memory is usually used to run the algorithm in parallel on a single multi-core machine [9, 14, 18, 23], distributed memory is used to distribute the algorithm on multiple machines [2, 12, 14, 24]. Distributed SGD (DSGD) is appropriate for very large-scale problems where data can be distributed over massive (theoretically unlimited) number of machines each with its own computational resources and I/O bandwidth, however its efficiency is bounded by the communication latency across machines. On the other hand, parallel SGD (PSGD) takes advantage of the multiple and fast processing units within a single machine with higher bandwidth communication, however, the computational resources and I/O bandwidth are limited,

In this paper, we set out to explore the potential gains that can be achieved by leveraging the advantages of both the distributed and parallel paradigms in a unfied approach. The proposed algorithm, DPSGD, curbs the communication cost by updating a local copy of the parameter vector being optimised, $\boldsymbol{v}$, multiple times during which each machine performs parallel computation. The distributed computation of DPSGD among multiple machines is carried out in an *asynchronous* fashion whereby workers compute their local updates independently [14]. A master aggregates these updates to amend the global parameters. The parallel implementation of DPSGD on each local machine is *lock-free* whereby multiple cores are allowed equal access to the shared memory to read and update the variables without locking [23] (i.e. they can read and write the shared memory simultaneously). We provide a theoretical analysis of the convergence rate of DPSGD for non-convex optimisation problems and prove that linear speed-up with respect to the number of cores and workers is achievable while they are bounded by $T^{1/4}$ and $T^{1/2}$, respectively, where T is the total number of iterations. Furthermore, we empirically validate these results by applying DPSGD to train convolution neural networks for a simple image classification task.

The rest of the paper is organised as follows. Section 2 presents the related work. Section 3 presents the proposed algorithm and the theoretical study. In Section 4, we carry out experiments and discuss the empirical results. Finally, Section 5 draws some conclusions and suggests future work.

2 RELATED WORK

We divide this section into two parts. The first part discusses some of the related literature on distributed and parallel SGD. Due to space limitation, only SGD-based methods for non-convex problems are briefly covered. The second part covers selected distributed and parallel algorithms proposed for deep learning, respectively. A handful of SGD-based methods have been proposed recently for large-scale non-convex optimisation problems [7, 10, 14, 23], which embrace either a distributed or parallel paradigm. [14] provide a theoretical analysis for two different non-convex asynchronous distributed and lock-free parallel SGD algorithms showing that linear speed-up with respect to the number of workers, T, is achievable when T is bounded by $O(\sqrt{T})$. Improved versions using variance reduction techniques have recently been proposed in [7, 10] to accelerate the convergence rate with a linear rate being achieved instead of the sub-linear one of SGD. Although many algorithmic implementations are lock-free [7, 10, 14], the theoretical analysis of the convergence is based on an assumption that no over-writing happens. Hence, write-lock or atomic operation for the memory are needed to prove the convergence. In contrast, [23] propose a completely parallel lock-free implementation and analysis. A summary of related work is presented in Table 1.

ASYSG [14], an implementation of SGD that distributes the training over multiple workers, has been adoped by DistBelief [5] (a parameter server-based algorithm for training neural networks) and Project Adam [4] (another DL framework for training neural networks). [16] showed that ASYSG can achieve noticeable speedups on small GPU clusters. Other similar work [13, 19] have also employed ASYSG to scale deep neural networks. The two most popular and recent DL framework TensorFlow [1] and Pytorch [17] have embraced the Hogwild [18, 23] and ASYSG [14] implementations to scale-up DL problems[1].

[1]See, for instance, https://github.com/pytorch/examples and https://github.com/tmulc18/Distributed-TensorFlow-Guide

Algorithm 1 DPSGD-Master: Updates performed at the master machine.

```
1:  initialise: number of iteration $T$, global variable $\boldsymbol{v}$, global
    learning rate $\{\rho_t\}_{t=0,\dots,T-1}$
2:  for $t = 0, 1, 2, \dots T-1$ do
3:      Collect $M$ updating vectors $\boldsymbol{w}_1, \dots, \boldsymbol{w}_M$ from the workers.
4:      Update the current estimate of the global parameter
        $\boldsymbol{v} \leftarrow \boldsymbol{v} + \rho_t \sum_m \boldsymbol{w}_m$
5:      $t \leftarrow t + 1$
6:  end for
```

Algorithm 2 DPSGD-Worker: Updates performed at each worker machine.

```
1:  initialise: number of iterations of per-worker loop $B$,
    learning rate $\eta$, number of threads $p$
2:  while (MasterIsRun) do
3:      Pull a global parameter $\boldsymbol{v}$ from the master and put it in
        the shared memory.
4:      Fork p threads
5:      for b=0 to B-1 do
6:          Read current values of $\boldsymbol{u}$, denoted as $\hat{\boldsymbol{u}}$, from the
            shared memory.
7:          Randomly pick i from $\{1, \dots n\}$ and compute the
            gradient $\nabla f_i(\hat{\boldsymbol{u}})$.
8:          $\boldsymbol{u} \leftarrow \boldsymbol{u} - \eta \nabla f_i(\hat{\boldsymbol{u}})$
9:      end for
10:     Push the update vector $\boldsymbol{u} - \boldsymbol{v}$ from the shared memory
        to the master
11: end while
```

3 THE DPSGD ALGORITHM AND ITS PROPERTIES

Overview of the algorithm

The proposed DPSGD algorithms assumes a *star-shaped* computer network architecture: a master maintains the global parameter $\boldsymbol{v}$ (Alg. 1) and the other machines act as workers which independently and simultaneously compute local parameters $\boldsymbol{u}$ (Alg. 2). The workers communicate only with the master in order to access the state of the global parameter (line 3 in Alg. 2) and provide the master with their local updates (computed based on local parameters) (line 10 in Alg. 2). Each worker is assumed to be a multi-core machine, and the local parameter are obtained by running a lock-free parallel SGD; see Alg. 2. This is achieved by allowing all cores equal access to the shared memory to read and update at any time with no restriction at all [23]. The master aggregates predefined amounts of local updates coming from the workers (line 3 in Alg. 1), and then computes its global parameter. The

Table 1: Relevant work on parallel and distributed SGD.

Algorithms	Lock-free parallel	Asynchronous distributed	Both
ASYSG-CON [14]		✓	
ASYSG-INCON [14]	✓		
AsySVRG [23]	✓		
ASVRG-atom [7]		✓	
ASVRG-wild [7]	✓		
Shared-AsySVRG [10]	✓		
DistributedAsySVRG [10]		✓	
DPSGD (our algorithm)			✓

update step is performed as an atomic operation such that the workers are locked out and cannot read the global parameter during this step (see Alg. 1). Note that the local distributed computations are done in an asynchronous style, i.e. DPSGD does not lock the workers until the master starts updating the global parameter. That is, the workers might compute some of the stochastic gradients based on early values of the global parameter.

Convergence analysis

First, we introduce the notation, then we provide details of the DPSGD updates. Next, we list the needed assumptions that are used to develop the theoretical analysis of DPSGD.

Notation: In the following, $\boldsymbol{v}$ denotes the parameter being optimised, which we call the *global parameter* as it is maintained and updated by the master machine; $\boldsymbol{u}$ denotes a copy of the global parameter maintained and is updated by the workers; $\hat{u}$ denotes a copy of the local parameter $\boldsymbol{u}$ and is stored in the worker's shared memory; $||\boldsymbol{x}||$ denotes the Euclidean norm of vector $\boldsymbol{x}$; $f_i(.)$ is the objective function defined on the i^{th} instance; $\nabla f(\boldsymbol{v})$ is the gradient vector of $f(\boldsymbol{v})$.

As in [22], we define a synthetic sequence $\{\boldsymbol{u}_{t,m,b}\}$ equivalent to the updates for the b^{th} per-worker loop of the m^{th} update vector associated with the t^{th} master loop.

Algorithm 2, line 3 refers to:

$\boldsymbol{u}_{t,m,0} = \boldsymbol{v}_{t-1}$

Algorithm 2, line 8 refers to:

$\boldsymbol{u}_{t,m,b+1} = \boldsymbol{u}_{t,m,b} - \eta S_{t+\tau_{t,m},m,b} \nabla f_{i_{t+\tau_{t,m},m,b}}(\hat{\boldsymbol{u}}_{t,m,b})$

Algorithm 1, line 4 refers to: (2)

$$\boldsymbol{v}_t = \boldsymbol{v}_{t-1} + \rho_{t-1}\Big(\sum_{m=1}^{M} \boldsymbol{u}_{t-\tau_{t,m},m,\tilde{B}} - \boldsymbol{v}_{t-1-\tau_{t,m}}\Big) \qquad (3)$$

where $S_{t,m,b}$ is a diagonal matrix whose diagonal entries are 0 or 1, $\tilde{B}$ is the total number of per-worker loop's iterations done by p threads ($\tilde{B} = pB$) and $\tau_{t,m}$ is the delay of the m^{th} global update for the t^{th} iteration caused by the asynchronous distribution. $S_{t,m,b}$ is used to denote whether over-writing happens. If $S_{t,m,b}(k,k) = 0$, then the kth element of the gradient vector $\nabla f_{i_{t,m,b}}(\hat{\boldsymbol{u}}_{t,m,b})$ is overwritten by other threads. Otherwise, that element successfully updates $\boldsymbol{u}_{t,m,b}$. To compute $\nabla f_{i_{t+\tau_{t,m},m,b}}(\hat{\boldsymbol{u}}_{t,m,b})$, $\hat{\boldsymbol{u}}_{t,m,b}$ is read from the shared memory by a thread.

Algorithm 2, line 6 refers to:

$$\hat{\boldsymbol{u}}_{t,m,b} = \boldsymbol{u}_{t,m,a(b)} - \eta \sum_{j=a(b)}^{b-1} P_{b,j-a(b)}^{t+\tau_{t,m},m} \nabla f_{i_{t+\tau_{t,m},m,j}}(\hat{\boldsymbol{u}}_{t,m,j}) \qquad (4)$$

where $a(b)$ is the step in the inner-loop whose updates have been completely written in the shared memory. The partial updates of the rest steps up to $b-1$ are defined by $\{P_{b,j-a(b)}^{t,m}\}_{a(b)}^{b-1}$. The following assumptions are needed to obtain the convergence results of DPSGD:

Assumption 1: The function $f(.)$ is smooth, that is to say, the gradient of $f(.)$ is *Lipschitzian*: there exists a constant $L > 0$, $\forall \boldsymbol{x}, \boldsymbol{y}$,

$$||\nabla f(\boldsymbol{x}) - \nabla f(\boldsymbol{y})|| \leq L||\boldsymbol{x} - \boldsymbol{y}||$$

or equivalently,

$$f(\boldsymbol{y}) \leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T(\boldsymbol{y} - \boldsymbol{x}) + \frac{L}{2}||\boldsymbol{y} - \boldsymbol{x}||^2.$$

Assumption 2: the per-dimension over-writing defined by $S_{t,m,b}$ is a random variate, independent of $i_{t,m,j}$[2]. As mentioned in [22], this assumption is reasonable since $S_{t,m,b}$ is affected by the hardware while $i_{t,m,j}$ is independent of the hardware.

Assumption 3: The conditional expectation of the random matrix $S_{t,m,b}$ on $\boldsymbol{u}_{t,m,b}$ and $\hat{\boldsymbol{u}}_{t,m,b}$ is a strictly positive definite matrix, i.e. $\mathbb{E}[S_{t,m,b}|\boldsymbol{u}_{t,m,b}, \hat{\boldsymbol{u}}_{t,m,b}] = S \succ 0$ with the minimum eigenvalue $\alpha > 0$.

Assumption 4: The gradients are unbiased and bounded: $\nabla f(\boldsymbol{x}) = \mathbb{E}_i[\nabla f_i(\boldsymbol{x})]$ and $||\nabla f_i(\boldsymbol{x})|| \leq V$, $\forall i \in \{1, ...n\}$.

[2]Note that i can be a set of indices for a per-worker mini-batch. In this paper, i refers to a single index for simplicity

Then, it follows that the variance of the stochastic gradient is bounded. $\mathbb{E}_i[||\nabla f_i(\boldsymbol{x}) - \nabla f(\boldsymbol{x})||^2] \leq \sigma^2$, $\forall \boldsymbol{x}$, where $\sigma^2 = V^2 - ||\nabla f(x)||$

Assumption 5: Delays between old local stochastic gradients and the new ones in the shared memory are bounded: $0 \leq b - a(b) \leq D$ and the delays between stale distributed update vectors and the current ones are bounded $0 \leq \max_{t,m} \tau_{t,m} \leq D'$

Assumption 6: All random variables in $\{i_{t,m,j}\}_{\forall t, \forall m, \forall j}$ are independent of each other.

Let $q(\boldsymbol{x}) = \frac{1}{n}\sum_{i=1}^{n} ||\nabla f_i(\boldsymbol{x})||^2$. We have $\mathbb{E}_i[||\nabla f_i(\boldsymbol{x})||^2] = q(\boldsymbol{x})$. Hence, $\mathbb{E}_i[q(\boldsymbol{x})] = \mathbb{E}_i[||\nabla f_i(\boldsymbol{x})||^2]$. Taking the full expectation on both sides, we get $\mathbb{E}[q(\boldsymbol{x})] = \mathbb{E}[||\nabla f_i(\boldsymbol{x})||^2]$. It can be proven that:

Lemma 1:

$$\mathbb{E}_t[q(\hat{\boldsymbol{u}}_{t,m,j})] < \mu \mathbb{E}_t[q(\hat{\boldsymbol{u}}_{t,m,j+1})] \tag{5}$$

given μ and η satisfying

$$\frac{1}{1 - \eta - \frac{9\eta(D+1)L^2(\mu^{D+1}-1)}{\mu-1}} \leq \mu \tag{6}$$

where $\mathbb{E}_t[.]$ denotes $\mathbb{E}_{i_{t,*,*}, s_{t,*,*}}[.]$. The proof can be derived from that in [23] using Assumptions (1) and (5).

We are now ready to state the following convergence rate for any non-convex objective:

Theorem 2: If Assumptions (1) to (6) hold, and the following inequalities are true:

$$M^2 \tilde{B}^2 \eta^2 L^2 \rho_{t-1} D' \sum_{n=1}^{D'} \rho_{t+n} \leq 1 \tag{7}$$

$$\frac{1}{1 - \eta - \frac{9\eta(D+1)L^2(\mu^{D+1}-1)}{\mu-1}} \leq \mu \tag{8}$$

then, we can obtain the following results:

$$\begin{aligned} &\frac{1}{\sum_{t=1}^{T}\rho_{t-1}} \sum_{t=1}^{T} \rho_{t-1}\, \mathbb{E}[||\nabla f(\boldsymbol{v}_{t-1})||^2] \\ &\leq \frac{2(f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*))}{M\tilde{B}\eta\alpha\sum_{t=1}^{T}\rho_{t-1}} + \frac{\eta^2 L^2}{\tilde{B}\sum_{t=1}^{T}\rho_{t-1}} \sum_{t=1}^{T} \rho_{t-1} \Bigg[V^2 \Bigg(\\ &\sum_{b=0}^{\tilde{B}-1} \frac{\mu(\mu^b - 1)}{\mu - 1} + \tilde{B}\frac{\mu(\mu^D - 1)}{\mu - 1} \Bigg) + M\tilde{B}^2\sigma^2 \sum_{j=t-1-D'}^{t-2} \rho_{j-1}^2 \Bigg] \\ &+ \frac{L\eta V^2}{\alpha\sum_{t=1}^{T}\rho_{t-1}} \sum_{t=1}^{T} \rho_{t-1}^2 \end{aligned}$$

where $\tilde{B} = pB$, $\boldsymbol{v}_*$ denotes the global optimum of the objective function in Eq. 1. We denote the expectation of all random variables in Alg. 2 by $\mathbb{E}[.]$. Theorem 2 shows that the weighted average of the l_2 norm of all gradients $||\nabla f(\boldsymbol{v}_{t-1})||^2$ can be bounded, which indicates an ergodic convergence rate. It can be seen that it is possible to achieve speed-up by increasing the number of cores and workers. Nevertheless to reach such speed-up, the learning rates η and ρ_t have to be set properly (see Corollary 3).

Corollary 3: By setting the learning rates to be equal and constant:

$$\rho^2 = \eta^2 = \frac{\sqrt{(f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*))}}{A\alpha\sqrt{TM\tilde{B}}} \tag{9}$$

such that $A = LV^2\left(\frac{1}{\alpha} + \frac{1}{\alpha^2} + \frac{2L\mu}{(1-\mu)\alpha}\right)$, μ is a constant where $0 < \mu < 1$, then the bound in Eq. 7 and Eq. 8 can lead to the following bound:

$$T \geq max\left\{ \frac{M\tilde{B}L^2D'^2(f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*))}{A^2\alpha^2}, \frac{\big(f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*)\big)\big(\mu(\mu-1) + 9L^2\mu(D+1)(\mu^{D+1}-1)\big)^4}{M\tilde{B}A^2\alpha^2(\mu-1)^8} \right\} \tag{10}$$

and Theorem 2 gives the following convergence rate:

$$\frac{1}{T}\sum_{t=1}^{T} \mathbb{E}[||\nabla f(\boldsymbol{v}_{t-1})||^2] \leq 3A\sqrt{\frac{f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*)}{TM\tilde{B}}} \tag{11}$$

This corollary shows that by setting the learning rates to certain values and setting the number of iterations T to be greater than a bound depending on the maximum delay allowed, a convergence rate of $O(1/\sqrt{TMpB})$ can be achieved, and this is delay-independent. The negative effects of using old parameters (asynchronous distribution) and over-writing the shared memory (lock-free parallel) vanish asymptotically. Hence, to achieve speed-up, the number of iterations has to exceed a bound controlled by the maximum delay parameters, the number of iterations B (line 5 in Alg. 2), the number of updating vector M (line 3 in Alg. 1) and the number of parallel threads (cores) p.

Since D' and D are related to the number of workers and cores respectively, bounding the latter allows speed-up with respect to the number of workers and cores with no loss of accuracy. The satisfaction of Eq. 10 is guaranteed if:

$$T \geq \frac{M\tilde{B}L^2D'^2(f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*))}{A^2\alpha^2}$$

and

$$T \geq \frac{\big(f(\boldsymbol{v}_0) - f(\boldsymbol{v}_*)\big)\big(\mu(\mu-1) + 9L^2\mu(D+1)(\mu^{D+1}-1)\big)^4}{M\tilde{B}A^2\alpha^2(\mu-1)^8}$$

The fist inequality leads to $O(T^{1/2}) > D'$. Thus, the upper bound on the number of workers is $O(T^{1/2})$. Since $0 < \mu < 1$, the second inequality can be written as follows: $O(T^{1/4}) \geq \big(\mu(1-\mu) + 9L^2\mu(D+1)(1-\mu^{D+1})\big)$. Hence, $O(T^{1/4}) \geq D$. Thus, the upper bound on the number of number of cores (threads) is $O(T^{1/4})$. The convergence rate for serial and synchronous parallel stochastic gradient (SG) is consistent with $O(1/\sqrt{T})$ [6, 8, 15]. While the work load for each worker running DPSGD is almost the same as the work load of the serial or synchronous parallel SG, the progress done by DPSVG is Mp times faster than that of serial SG.

In addition to the speed-up, DPSGD allows one to steer the trade off between multi-core local computation and multi-node communication within the cluster. This can be done by controlling the B parameter. Traditional methods reduce the communication cost by increasing the batch size which decreases the convergence rate, increase local memory load and decrease local input bandwidth. On the contrary, increasing B for DPSGD can increase the speed-up if some assumptions are met (see Theorem 2 and Corollary 3). This ability makes DPSGD easily adaptable to diverse spectrum of large-scale computing systems with no loss of speed-up.

$$T * Tc \leq T * Tc \leq M * T * Tc \tag{12}$$

where Tc is the communication time need for each master - worker exchange. For simplification, we assume that the Tc is fixed and the same for all nodes. If the time needed for computing one update $Tu \leq Tc$, the total time needed by the distributed algorithm DTT could be higher than that of the sequential SGD STT:

$$T * Tc + STT/M \leq DTT \leq M * T * Tc + STT/nw \tag{13}$$

where $STT \leq T * Tc$, hence

$$STT < STT(1 + 1/M) \leq DTT \tag{14}$$

In such cases, exiting distributed algorithms increases the local batch size so that Tu increases resulting in lower stochastic gradient variance allowing higher learning rate to be used, hence better convergence rate. This introduces a trade-off between computational efficiency and sample efficiency. Increasing the batch size by a factor of k increase the time need for local computation by $O(k)$ and reduces the variance proportionally to $1/k$ [3]. Thus, higher learning rate can be used. However, there is a limit on the size of the learning rate. In another word, maximising the learning speed with respect to the learning rate and the batch size has a global solution. This maximum learning speed can be improved on using DPSGD which performs B times less communication steps. For mini-batch SGD with minibatch size G, the convergence rate can written as $O(1/\sqrt{GT})$. Since the total number of examples examined is GT while there is only a $\sqrt{G}$ times improvement, the convergence speed degrades with increasing mini-batch size. The convergence rate of DPSGD with mini-batch G can be easily deduced from Theorem 2 as $O(1/\sqrt{BMGT})$. Hence, $\sqrt{BM}$ better convergence rate than mini-batch SGD and $\sqrt{BM}$ better convergence rate than standard asynchronous SGD with B times less communication.

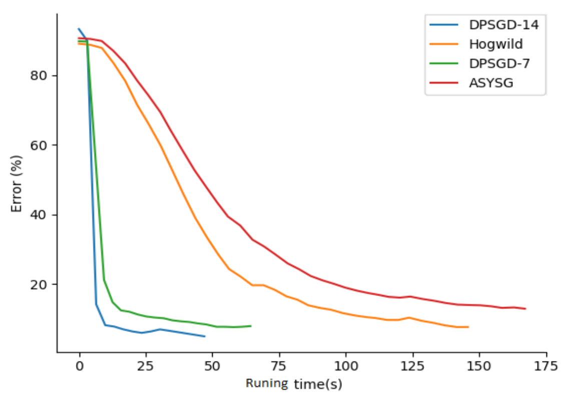

Figure 1: MNIST image classification task using a CNN: accuracy error with respect to running time in second.

4 EXPERIMENTS

In this section we empirically verify the potential speed-up gains expected from the theoretical analysis. We use DPSGD to train a convolutional neural network on an image classification task and compare against a parallel lock-free SGD algorithm, Hogwild [18, 23], and the asynchronous distributed SGD algorithm, ASYSG [14]. The experiments are run on the same data sets used by Hogwild and ASYSG [14, 23], i.e. CIFAR10-FULL [11] and MNIST[3].

In these experiments DPSGD was used to train convolutional neural networks (CNNs) using the Pytorch package on both the MNIST and CIFAR10-FULL datasets. All experiments were performed on a high-performance computing (HPC) environment using message passing interface (MPI) for Python (MPI4py). The *cluster* consists of 14 nodes, excluding the head node, with each node is a 2-sockets-6-cores-2-thread processor. As our interest here is in the speed ups achievable with DPSGD, we used only shallow networks consisting of 2 convolution layers and 2 fully connected for MNIST, and 2 convolution layers and 3 fully connected layers for CIFAR10-FULL case. The classification rate as a function of running time for MNIST and CIFAR10-FULL can be found in Figure 1 and Figure 2, respectively. We report on a comparison of DPSGD running on both 7 and 15 nodes (threads fixed to 24) against Hogwild (24 threads) and ASYSG (12 nodes). These comparisons indicate that significant speed-ups can be achieved improving over Hogwild and ASYSG when training CNNs for image classification. Lowering the error rate further would require larger network architectures.

[3]http://yann.lecun.com/exdb/mnist/

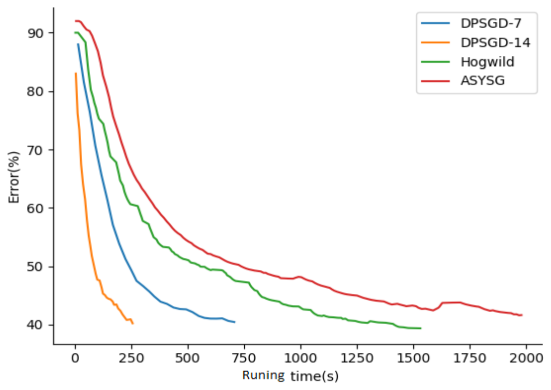

Figure 2: CIFAR10-FULL image classification task using a CNN: accuracy error with respect to running time in second.

5 CONCLUSION AND DISCUSSION

We have proposed a novel asynchronous distributed and lock-free parallel optimisation algorithm. The algorithm is implemented on a computer cluster with multi-core nodes. Both theoretical and empirical results show that DPSGD leads to speed-up on non-convex problems. An implementation of DPSGD has been assessed to train convolutional neural networks with empirical results validating the theoretical findings. We also provide some comparison to similar state-of-the-art methods.

Going forward, further improvements and validations could be achieved by pursuing research along four directions: (1) employing variance reduction techniques to improve the convergence rate (from sub-linear to linear) while guaranteeing multi-node and multi-core speed-up; (2) proposing a framework enabling dynamic trade-offs between local computation and communication; (3) proposing techniques to improve the local optimum of the distributed parallel algorithms; (4) applying DPSGD to large-scale deep learning problems.

Acknowledgements. A. Bouchachia was supported by the European Commission under the Horizon 2020 Grant 687691 related to the project. We also thank Warwick Manufacturing Group for providing the computational resources required to carry out the experiments.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: a system for large-scale machine learning.. In *OSDI*, Vol. 16. 265–283.
[2] Alekh Agarwal and John C Duchi. 2011. Distributed delayed stochastic optimization. In *Neural Information Processing Systems*.
[3] Léon Bottou, Frank E Curtis, and Jorge Nocedal. 2018. Optimization methods for large-scale machine learning. *SIAM Rev.* 60, 2 (2018), 223–311.
[4] Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project Adam: Building an Efficient and Scalable Deep Learning Training System.. In *OSDI*.
[5] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*. 1223–1231.
[6] Ofer Dekel, Ran Gilad-Bachrach, Ohad Shamir, and Lin Xiao. 2012. Optimal distributed online prediction using mini-batches. *Journal of Machine Learning Research* (2012).
[7] Cong Fang and Zhouchen Lin. 2017. Parallel Asynchronous Stochastic Variance Reduction for Nonconvex Optimization.. In *AAAI*.
[8] Saeed Ghadimi and Guanghui Lan. 2013. Stochastic first-and zeroth-order methods for nonconvex stochastic programming. *SIAM Journal on Optimization* (2013).
[9] Zhouyuan Huo and Heng Huang. 2016. Asynchronous stochastic gradient descent with variance reduction for non-convex optimization. *arXiv preprint arXiv:1604.03584* (2016).
[10] Zhouyuan Huo and Heng Huang. 2017. Asynchronous Mini-Batch Gradient Descent with Variance Reduction for Non-Convex Optimization.. In *AAAI*.
[11] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.
[12] John Langford, Alex J Smola, and Martin Zinkevich. 2009. Slow learners are fast. *Neural Information Processing Systems* (2009).
[13] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. 2014. Scaling Distributed Machine Learning with the Parameter Server.. In *OSDI*.
[14] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Neural Information Processing Systems*.
[15] Arkadi Nemirovski, Anatoli Juditsky, Guanghui Lan, and Alexander Shapiro. 2009. Robust stochastic approximation approach to stochastic programming. *SIAM Journal on optimization* 19, 4 (2009), 1574–1609.
[16] Thomas Paine, Hailin Jin, Jianchao Yang, Zhe Lin, and Thomas Huang. 2013. GPU asynchronous stochastic gradient descent to speed up neural network training. *arXiv preprint arXiv:1312.6186* (2013).
[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[18] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Neural Information Processing Systems*.
[19] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
[20] Eric P Xing, Qirong Ho, Wei Dai, Jin Kyu Kim, Jinliang Wei, Seunghak Lee, Xun Zheng, Pengtao Xie, Abhimanu Kumar, and Yaoliang Yu. 2015. Petuum: A new platform for distributed machine learning on big data. *IEEE Transactions on Big Data* (2015).
[21] Sixin Zhang, Anna E Choromanska, and Yann LeCun. 2015. Deep learning with elastic averaging SGD. In *Neural Information Processing Systems*.
[22] Shen-Yi Zhao and Wu-Jun Li. 2016. Fast Asynchronous Parallel Stochastic Gradient Descent: A Lock-Free Approach with Convergence Guarantee. In *AAAI*.
[23] Shen-Yi Zhao, Gong-Duo Zhang, and Wu-Jun Li. 2017. Lock-Free Optimization for Non-Convex Problems.. In *AAAI*.
[24] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. 2010. Parallelized stochastic gradient descent. In *Neural Information Processing Systems*.