FTSGD: An Adaptive Stochastic Gradient Descent Algorithm for Spark MLlib

Hong Zhang*, Zixia Liu†, Hai Huang‡, Liqiang Wang§
*†§Department of Computer Science, University of Central Florida, Orlando, FL, USA
‡IBM T.J. Watson Research Center, Yorktown Heights, NY, USA
Email: *hzhang1982@knights.ucf.edu, †zixia@knights.ucf.edu, ‡haih@us.ibm.com, §lwang@cs.ucf.edu

***Abstract*—The proliferation of massive datasets and the surge of interests in big data analytics have popularized a number of novel distributed data processing platforms such as Hadoop and Spark. Their large and growing ecosystems of libraries enable even novice to take advantage of the latest data analytics and machine learning algorithms. However, time-consuming data synchronization and communications in iterative algorithms on large-scale distributed platforms can lead to significant performance inefficiency. MLlib is Spark's scalable library consisting of common machine learning algorithms, many of which employ Stochastic Gradient Descent (SGD) to find minima or maxima by iterations. However, the convergence can be very slow if gradient data are synchronized on each iteration.**

In this work, we optimize the current implementation of SGD in Spark's MLlib by reusing data partition for multiple times within a single iteration to find better candidate weights in a more efficient way. Whether using multiple local iterations within each partition is dynamically decided by the 68-95-99.7 rule. We also design a variant of momentum algorithm to optimize step size in every iteration. This method uses a new adaptive rule that decreases the step size whenever neighboring gradients show differing directions of significance. Experiments show that our adaptive algorithm is more efficient and can be 7 times faster compared to the original MLlib's SGD.

***Index Terms*—Spark; MLlib; Asynchronous Stochastic Gradient Decent; Adaptive Iterative Learning**

I. Introduction

The executions of machine learning and deep learning algorithms often span multiple machines as data or model usually cannot fit in a single machine. Apache Hadoop [1] [2] is an open-source framework that distributes data across a cluster of machines and parallelizes their execution using MapReduce programming model. Apache Spark [3] improves upon Hadoop by keeping data in-memory in the format of resilient distributed dataset (RDD) between Map and Reduce iterations. It has been rapidly adopted due to its 10x to 100x speedup over Hadoop, especially in machine learning related applications such as classification, regression, and clustering. Using Spark, programmers can efficiently design, deploy and execute streaming, machine learning, and graph processing workloads. Spark MLlib is a scalable machine learning library, which provides RDD-based APIs to support scalable machine learning. It consists of common machine learning algorithms, including classification, regression, clustering, etc.

In the area of distributed machine learning, to handle very large datasets and/or complex models, parameter server frameworks [4–6] are proposed to store, share and update parameters for solving large scale machine learning problems. The framework uses *synchronous* and/or *asynchronous* data communication between compute nodes to support parallel computing of SGD. In the context of Spark, the driver of a Spark application can be regarded as a specialized parameter server that updates and distributes parameters in a synchronized way. However, unlike parameter server frameworks that are usually implemented on highly efficient MPI, Spark's *synchronous* iterative communication pattern is based on MapReduce between the driver and workers, which makes it an inefficient platform for machine learning algorithms using SGD.

Gradient descent, also known as steepest descent, is one of the most popular methods in the field of machine learning since it can minimize error rate by following the fastest direction. But it is difficult to decide the correct learning rate and choose a fast convergent algorithm. Robert *et al.* [7] proposed four heuristics for achieving faster rates of convergence: (1) each parameter should have its own learning rate; (2) each parameter learning rate should be allowed to change in each iteration; (3) if the derivative of a parameter has the same direction for consecutive iterations, its learning rate should be increased, (4) otherwise, the learning rate should be decreased. There are several implementations [8, 9] that are widely used by the deep learning community to increase the convergence rate through adaptive learning, but none of them considers the root cause of oscillation when the current step overshoots the optimum.

In this paper, we propose an adaptive fast-turn stochastic gradient descent algorithm, called FTSGD, which works more efficiently on the platforms that rely on iterative communication such as Spark and Hadoop to speed up the convergence of machine learning algorithms, *e.g.*, linear regression, logistic regression, and SVM. Our contributions are summarized as follows:

- We design an asynchronous parallel SGD algorithm with iterative local search to reuse data partition multiple times within a single global iteration. By analyzing the derivatives of parameters, our algorithm gives an optimized number of local iterations.
- We design a new adaptive learning rate algorithm on momentum that can adjust the learning rate and momentum coefficients adaptively based on similarities between the current and previous gradients.

- We use the 68-95-99.7 rule to check whether the parameters calculated by all nodes have a normal distribution, to guard against performance degradation due to skewed data.
- We terminate local iterations when an oscillation is detected in order to reduce the execution time and avoid amplification effect.

In our experiments, we demonstrate that FTSGD gains an improvement of 7 times compared with the algorithms implemented by MLlib when the input data size is 100 GB.

This paper is organized as follows. Section II gives the background about gradient descent and the current parallel implementation in Spark. Then we describe the details of our design in Section III. Section IV shows experimental results. A review of related work is presented in Section V. Section VI gives our conclusions and future work.

II. Background

A. Gradient Descent

Gradient descent is an iterative optimization algorithm that minimizes an error function defined by a set of parameters. We use the terms weight and parameter interchangeably in this paper. There are several steps in finding a local minimum of a function using gradient descent: (1) Initialize weights with random values. (2) Compute the gradient, which is the first derivative of the error function $err(w)$, since it is the fastest decreasing direction for the current weights. (3) Update the weights with the negative gradients, as shown in Equation 1, where γ is the learning rate. (4) Repeat steps 2 and 3 until the error cannot be reduced notably or already reach the maximum iterations.

$$W_t = W_{t-1} - \gamma \nabla err(W_{t-1}) \tag{1}$$

B. Stochastic Gradient Descent

However, calculating gradients over the entire training set is often too expensive, and may have a high probability of obtaining a partial optimal solution. Moreover, if the entire dataset is too large to be cached in memory, performance can degrade significantly. Stochastic gradient descent (SGD), on the other hand, is a stochastic approximation of the gradient descent algorithm by using a few training examples or a minibatch from the training set to update parameters in every iteration. It avoids the high cost of calculating gradients over the whole training set, but is sensitive to feature scaling. It can be denoted as follows.

$$W_t = W_{t-1} - \gamma \sum_{i=1}^{n} \nabla err_i(W_{t-1})/n \tag{2}$$

where n is the number of samples in a minibatch.

C. Parallel Stochastic Gradient Descent

Stochastic gradient descent is one of the most important optimizers in Spark MLlib. Algorithm 1 shows the process of calculating stochastic gradient descent in Spark MLlib. At first, it broadcasts the initial weights or the weights calculated by the previous iteration to every compute node, which may host one or more partitions of datasets. Then the input RDD is sampled according to the minibatch rate. After that, it calculates gradient for each sample in every partition, and aggregates all gradients by the driver using a multi-pass tree-like reduce (which is called $TreeAggregate$). At the end of each iteration, the weights are updated according to Equation 2. This process terminates when all iterations are executed or when it has converged.

There are three major problems of SGD in Spark: (1) the data uploaded are only used once in each iteration. If the memory reserved for this application is not large enough to cache all data, it must read data from disk or even worse from other compute nodes. (2) *treeAggregate* operation reduces all gradients by multiple stages, which is inefficient. (3) The original learning rate updating algorithm is too simple, which makes convergence too slow.

Algorithm 1 Parallel SGD of Original MLlib

Input: $input\_rdd, init\_weights, num\_iterations, minibatch\_rate$
Output: $weights$
$weights = init\_weights$
for $step = 1$ to $num\_iterations$ **do**
    $broadcast(weights)$
    for all $partitions$ in $input\_rdd$ **parallel do**
        $samples = partition.sample(minibatch\_rate)$
        for each $sample$ in $samples$ **do**
            $grad = computeGradient(weights, sample)$
        end for
    end for
    $grads$ = tree aggregate gradients from all compute nodes
    $weights = updateWeights(weights, grads)$
end for
return $weights$

III. Design and Implementation

To overcome the problems mentioned in Section II, we design a novel algorithm that has inner iterations within each global iteration. An inner iteration updates local weights multiple times without sending back the gradients before the accumulated weights are reduced on the driver at the end of each global iteration. This approach dramatically reduces the amount of communication and avoid aggregating gradients in multiple stages.

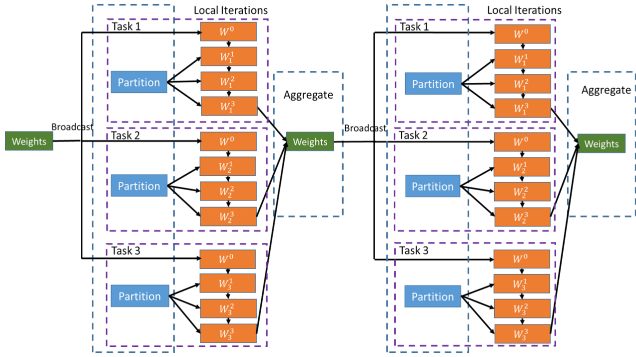


Fig. 1. Gradient Descent with Iterative Local Search.

Algorithm 2 Parallel SGD with Iterated Local Search

Input: $input\_rdd, init\_weights, global\_iters, minibatch\_rate, local\_iters$
Output: g_ws

$g\_ws = init\_weights$
for $g = 1$ to $global\_iters$ **do**
  $broadcast(g\_ws)$
  for all $partitions$ in $input\_rdd$ **parallel do**
    $l\_ws = g\_ws$
    for $l = 1$ to $local\_iters$ **do**
      $samples = partition.sample(minibatch\_rate)$
      $l\_ws = computeWeights(l\_ws, samples)$
    end for
  end for
  $weights$ = aggregate l_ws from all compute nodes
  // Test if weights satisfy 68-95-99.7 rule
  if $(g == 1$ && $!satisfyGaussianDist(weights))$ **then**
    $local\_iters = 1$
  end if
  $g\_ws = averageWeights(weights)$
end for
return g_ws

A. Parallel SGD with iterative local search

Algorithm 2 shows the parallel SGD algorithm with local iterations. Within each global iteration, we asynchronously update local weights multiple times within local iterations before updating the global weights using the new average weights at the end of each global iteration on the driver. In each local iteration, the calculated weights are used to compute subsequent weights using the same partition. Let $local\_iters$ denote the number of local iterations, which is determined by the features of input data, computational complexity of algorithm, and capability of cluster. Roughly, we notice that the $local\_iters$ is inversely proportional to the standard deviation of the weights calculated from all computers to guarantee convergence. $local\_iters$ is a hyper-parameter in our current system. It will be our future work to investigate how to find the optimal $local\_iters$ adaptively.

In the first global iteration, we check whether the weights calculated by all partitions fit a Gaussian distribution. If so, our optimizations are applied; otherwise, we fallback to the original algorithm in Spark's MLlib due to having too significant difference among the calculated weights on all partitions. Note that in Algorithm 2, weights are aggregated by the driver. However, in Algorithm 1, gradients are aggregated by the driver. Such an optimization is based on Spark's intrinsic features. In the original MLlib implementation shown in Algorithm 1, a partition is sampled once and each sample generates a vector of gradients. All these gradients within one global iteration are reduced to the driver based on tree-aggregation. In our Algorithm 2, all gradients are applied to the local weights, and only the weights are transfered at the end of each global iteration. This approach dramatically reduces the data to be transferred. In addition, tree-aggregation consists of multiple stages, which may be efficient for large-scale Spark systems, but could degrade the performance of small-scale systems.

Because of the limit of pages, we have no space to give the proof of convergence which depends on the distribution of the weights.

B. 68-95-99.7 Rule

As mentioned above, we need to know the distribution of our data to determine whether to employ aggressive local iterative search to speedup. However, to assert the input data as normal is more complex and time consuming, like Kolmogorov-Smirnov test (K-S test) [10]. From the analysis of the convergence, we found that 99.7% of vectors of weights calculated by all compute nodes lie within 3σ, and that σ is small enough. So we use the 68-95-99.7 rule instead of hypothesis testing to check whether the sets of weights collected from all compute nodes have a normal distribution.

$$similarity = \frac{A \bullet B}{||A|| \times ||B||} \tag{3}$$

This empirical rule is the facts that 68.27%, 95.45% and 99.73% of the values in a normal distribution fall within one, two and three standard deviations of the mean, respectively. Since one set of weights is a vector, to simplify the process, we employ cosine similarity [11] between each set of weights and the mean weights to check whether the weights satisfy the 68-95-99.7 rule. Equation 3 shows the measure of similarity between two non-zero vectors.

C. Parallel SGD with Adaptive Momentum

Spark MLlib, by default, uses a simple adaptive updater that adjusts the learning rate by the inverse of the square root of the number of iterations executed, as shown in Equation 4. This algorithm does not include any heuristics proposed by [7]. It only reduces the step size gradually to ensure the convergence for non-convex optimization problems.

$$w_t = w_{t-1} + \frac{current\_step}{\sqrt{num\_steps}} w_{t-1} \tag{4}$$

As one of the heuristics indicated by [7], Momentum adds a fraction of the previous updating vector to the current gradients, as shown in Equation 5. If the previous updating vector is in the same direction with the current gradient, it increases the step size towards the target; otherwise the step size is reduced. But this algorithm causes an overshooting problem, *i.e.*, the search step strides over the minima but the next search direction does not turn round since the momentum term is too large.

$$\begin{aligned} v_t &= \alpha v_{t-1} + \beta \nabla err(w) \\ w_t &= w_{t-1} - v_t \end{aligned} \tag{5}$$

Swanston [7] proposes a simple adaptive momentum (SAM) algorithm that still uses constant terms for both learning rate and momentum, but adds a coefficient to adaptively adjust the momentum according to the similarities between the last gradient and the current gradient. If two adjacent gradients have similar directions ($\cos(\theta) > 0$), it increases the influence of the previous iteration; otherwise, it reduces the influence and changes direction quickly. But this algorithm neither analyzes the root cause of the oscillation nor gives a guideline for adjusting the learning rate.

$$\begin{aligned} v_t &= \alpha(1 + \cos(\theta))v_{t-1} + (\beta/2)\nabla err(w) \\ w_t &= w_{t-1} - v_t \end{aligned} \tag{6}$$

Algorithm 3 shows the steps of calculating SGD in parallel with adaptive momentum. At the beginning of each global iteration, we broadcast the current average weights g_ws to all compute nodes. Then for each partition, we calculate a vector of weights by local iterations. In each local iteration, we sample the partition randomly with the $minibatch$ rate. Then we calculate the gradients for each sample with the present local weights. After that, we check cosine similarity between the previous gradient and the current gradient. If the cosine value is less than 0, which means the current direction is totally different from the previous direction, there must be overshooting for some weights. There are two potential reasons causing this phenomenon: (1) the momentum term is too large, so even if the learning rate is small, the current step still strides over the target; (2) the learning rate is too large, and the weights cross over and are on the other side. Between the two causes, we must decide which is the dominating factor for overshooting. Firstly, we adjust the momentum coefficient α to 0 to check whether the momentum is the root cause. Then we update the local weights, and start the next local iteration. If there is no oscillation in the next iteration, then it indicates that momentum being too large caused this oscillation. Otherwise, the oscillation must be caused by the learning rate being too fast.

We do not adjust the learning rate between local iterations, but simply terminate local iterations if oscillation is detected. Here we use $vibrate\_last$ to store whether or not there is an oscillation in the previous iteration. To summarize, if oscillation occurs, we first adjust the momentum coefficient to 0; and if two consecutive oscillations occur, we reduce the learning rate by half.

Algorithm 3 Parallel SGD with Adaptive Momentum

Input: $input\_rdd, l\_ws, old\_grad, local\_iters, \alpha, \beta, minibatch\_rate$
Output: g_ws

```
g_ws = init_weights;
for g = 1 to global_iters do
    broadcast(g_ws)
    for all parts in input_rdd parallel do
        oscillation = 0
        vibrate_last = false
        for l = 1 to local_iters do
            samples = part.sample(minibatch_rate)
            new_grad = computeGrad(l_ws, samples);
            similarity = cosθ(old_grad, new_grad)
            if (similarity < 0) then
                if (vibrate_last) then
                    oscillation = 1
                    break
                else
                    vibrate_last = true
                    l_ws = momentum(0, β, old_grad, new_grad)
                end if
            else
                vibrate_last = false
                l_ws = momentum(α, β, old_grad, new_grad)
            end if
        end for
    end for
    num_oscils = aggregate oscillation from all nodes
    if (num_oscils/num_parts >= 0.5) then
        β = β/2.0
    end if
    weights = aggregate l_ws from all compute nodes
    if (g == 1 && !satisfyGaussianDist(weights)) then
        local_iters = 1
    end if
    g_ws = averageWeights(weights)
end for
return g_ws
```

IV. Experiments

A. Experimental Setup

Our experiments were performed on a cluster consisting of 1 NameNode and 6 DataNodes. Each node has an Intel(R) Xeon(R) CPU E5-2620 v3 with 6 cores, and 32 GB memory. Our Spark cluster is based on CentOS Linux Server 7, JDK version 1.8, Apache Hadoop version 2.7 and Apache Spark 2.1. We use a representative gradient descent method, Linear Regression, as benchmark to test the performance of FTSGD.

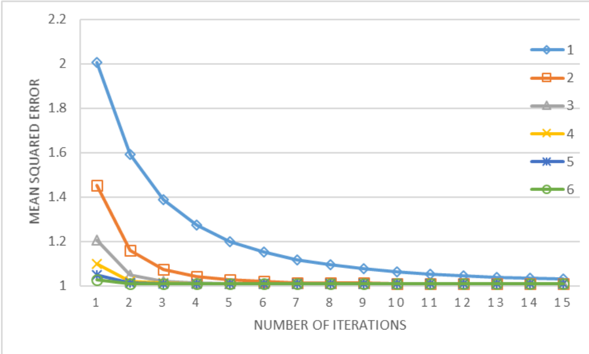


Fig. 2. Accuracy of different numbers of local iterations in every global iteration.

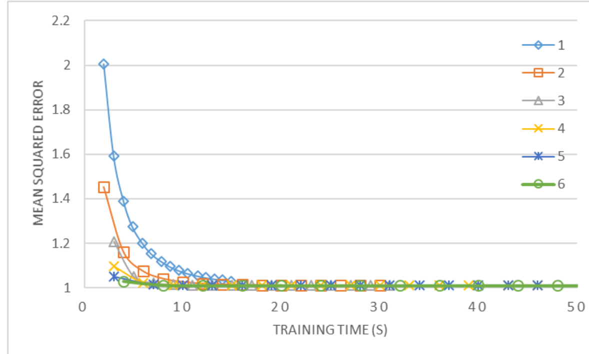
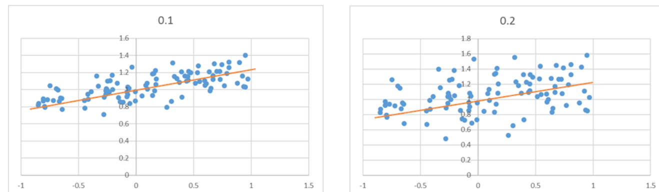


Fig. 3. Performance of different numbers of local iterations.



(a) Scaling factor: 0.1 (b) Scaling factor: 0.2

Fig. 4. Data distributions with different scaling factor.

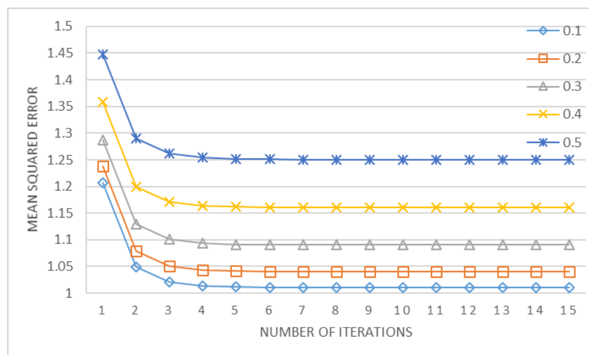


Fig. 5. Accuracy of different scaling factor

B. Experiments without Adaptive Learning Rate

We first compare the performance between the original SGD in MLlib and Algorithm 2 without considering the effect of the optimization of adaptive learning rate.

Figure 2 shows the accuracy of different number of local iterations, which is evaluated by mean squared error (MSE). In this experiment, the total input data size is 50 GB, which are generated by `LinearDataGenerator` class in MLlib package. Each sample has 100 features, and the scaling factor ϵ is 0.1. The initial learning rate is 1.0. When the local iteration is 1, our algorithm is the same as MLlib's SGD. When the number of local iterations is 6, we find that the accuracy is the best. It only use two global iterations to converge MSE to a very small value (1.029), which is even better than the accuracy of MLlib's SGD with 15 iterations.

Figure 3 indicates the performance of different number of local iterations with the same configuration in Figure 2. Although each global iteration in our algorithm is slower than MLlib's SGD, the performance of our algorithm is still better. This is because the convergence of our algorithm is fast, and we avoid the communication time of tree aggregation. When the number of local iterations is 5, the performance of our algorithm is best, which is 6.5 times faster than MLlib SGD with the same MSE. Although FTSGD with 6 local iterations is more accurate than with 5 iterations using the same global iteration, the longer execution time for each global iteration undermines the performance improvement.

Figure 4 demonstrates the input datasets with different ϵ scaling factor. The ϵ scaling factor is used to add white Gaussian noise to a full linear dataset. Figure 4(b) is the dataset with scaling factor 0.2, which is wider than the dataset in Figure 4(a) with scaling factor 0.1. Figure 5 shows the convergence with datasets of different scaling factor. We find that even with a large scaling factor 0.5, our algorithm with 3 local iterations is still convergent. This means if the distribution of data is an uniform normal distribution, our algorithm can converge, even the standard deviation is a little bigger. It is reasonable that the larger scaling factor dataset has larger MSE.

C. Experiments with Adaptive Learning Rate

In this experiment part, we discuss the performance of our algorithm with adaptive learning rate (*i.e.*, Algorithm 3 or FTSGD).

Figure 6 compares the performance between Algorithm 2 and Algorithm 3. We find that when the number of global iterations is 5, the accuracy of FTSGD is better than Algorithm 2 that uses the learning rate updating algorithm in Equation 6. Another is that the MSE difference between two adjacent iterations is very small after 10 global iterations (less than 0.00001), which means that FTSGD is convergent when the global iteration is 10, and FTSGD can terminate earlier than Algorithm 2 without learning rate optimization.

Figure 7 shows the performance of FTSGD with different input data sizes. We vary the input data size from 25 GB to 100 GB. When the input data size is 25 GB and 50 GB, FTSGD can outperform the original SGD by about 4.3 times and 3.6

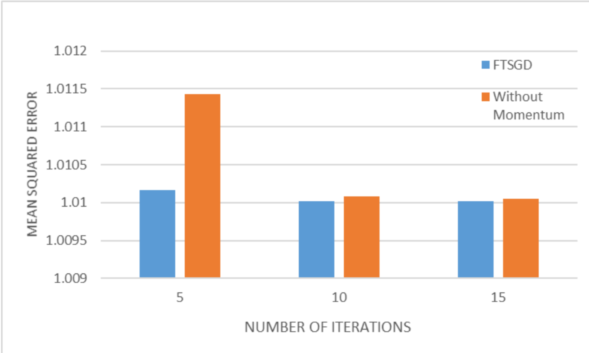


Fig. 6. Performance with and without adaptive learning rate.

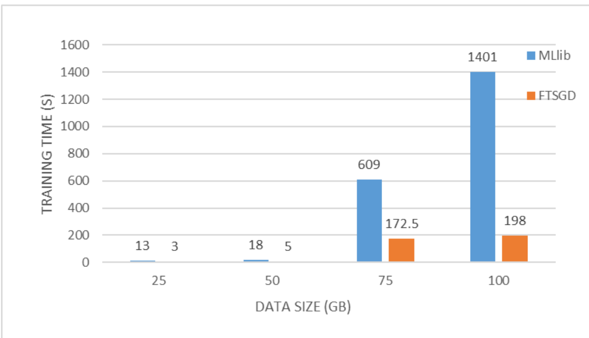


Fig. 7. Performance of different input data sizes.

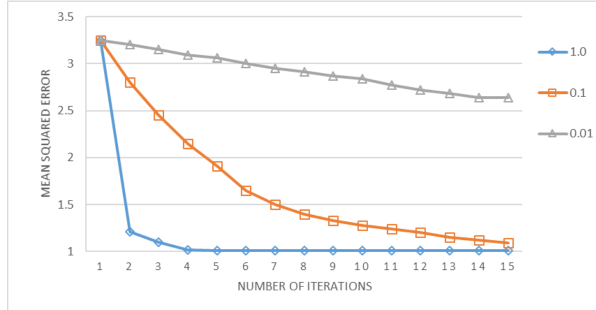


Fig. 8. Performance of different initial learning rate

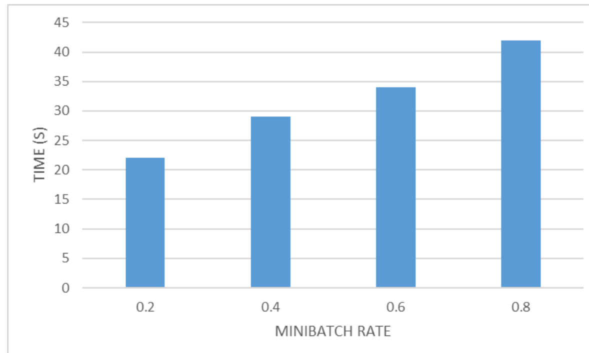


Fig. 9. Performance of different minibatch rate.

times, respectively. When the input data size is larger than 50 GB, the cost increases exponentially. This is due to data size exceeding memory size, which forces disk I/Os because data cannot be cached in memory. There are three kinds of tasks: (1) process local, where the data are cached in local memory; (2) node local, where the data are stored in local disk; (3) rack local, where the data must be fetched from a remote node in the same rack. We have 100 GB memory in our cluster, and 50% is reserved for OS and Spark system. If input is larger than 50 GB, Spark has to read data from disk (node local) or remotely (rack local). It is very common that input data are too large to be cached entirely into memory. That is why we reuse the data loaded in memory to do an asynchronous updating to save execution time. Figure 7 demonstrates that when the input data size is 100 GB, FTSGD only spends 198 seconds to converge and gives a better accuracy compared to MLlib's SGD that takes 1401 seconds.

Figure 8 shows the performance of FTSGD with the initial learning rate varied from 0.01 to 1.0. We notice that FTSGD with initial learning rate 1.0 gains more performance improvement. From Figure 8, the initial learning rate cannot be too small since a small rate may cause too many steps towards the optimal target. However, the initial learning rate cannot be too large since a large value will cause overshooting problem, even non-convergence. Because FTSGD can reduce the learning rate quickly if there exists oscillation, we can set the initial learning rate slightly larger, which also can detect more area to avoid stepping into local optimum.

Figure 9 demonstrates the effect of the minibatch rate. It is obvious that the larger the minibatch rate is, the longer it takes for one global iteration. For 50 GB input data size, it takes 22 seconds to calculate one global iteration when setting minibatch 0.2, but 42 seconds for minibatch 0.8, which almost doubles cost. But the accuracy is similar since the input data is very uniform.

In order to test the performance when the input data is non-uniform, we generate a data set which contains two data distributions, as shown in Figure 10. We vary the ratio of the number of samples in two datasets, and the results are shown in Table I for 15 iterations. In Table I, we calculate mean and standard deviation, and check if the parameters satisfy the 68-95-99.7 rule. We compare the performance of MLlib's SGD and FTSGD with 5 local iterations. The results shows that if the data is not a normal distribution, only the dataset satisfying the 68-95-99.7 rule can outperform the original SGD algorithm in MLlib. Even the data size ratio between two datasets is large such as 20:1, MLlib's SGD still shows better performance than FTSGD if the 68-95-99.7 rule is not satisfied. However, when the data size ratio between two datasets reaches 100:0.3, the 68-95-99.7 rule is satisfied and our algorithm finally outperforms MLlib's SGD.

D. Experiments with Different Size of Partitions

In this experiment, we measure the performance of input datasets with different number of partitions. We vary the number of partitions of 50 GB data from 500 to 2000, then the

TABLE I
PERFORMANCE OF TWO DATA SET WITH DIFFERENT RATIOS

	1:1	2:1	3:1	4:1	5:1	10:1	20:1	100:0.3
Mean	0	0.004329004	0.00654233	0.007792208	0.008658009	0.010625738	0.011750155	0.012909324
STDEV	0.012987013	0.012244273	0.011218753	0.01038961	0.009679948	0.007467007	0.005531399	0.001418401
68-95-99.7 Rule	No	No	No	No	No	No	No	yes
MSE (MLlib)	1.763563134	1.593385182	1.507113571	1.451839211	1.414957992	1.344010868	1.293488292	1.258641468
MSE (5 Local Iters)	1.829216647	1.615403542	1.518092382	1.458397027	1.418999539	1.345111833	1.293792029	1.258606695

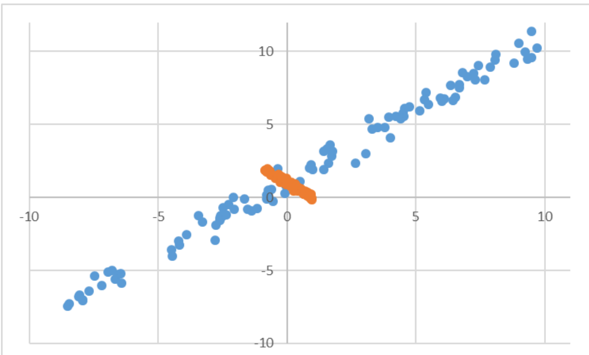


Fig. 10. Two datasets with different distributions.

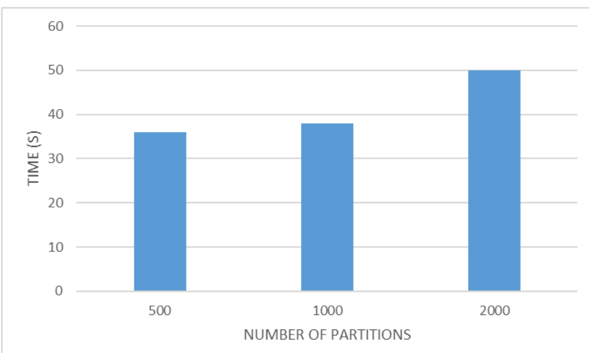


Fig. 11. Performance of different number of partitions with the same data size.

partition size is changed from 120 MB to 30 MB. As shown by Figure 11, we notice that the larger the partition size is, the better the performance is, because the large partition size can reduce the total number of tasks, and then decrease the overhead of task setup and cleanup. The maximum partition size is bounded by the block size in HDFS (128 MB in our cluster).

V. RELATED WORK

Designing asynchronous parallel stochastic gradient descent algorithms with or without lock is an active area in recent years. Liu *et al.* [12] introduce an asynchronous parallel stochastic descent algorithm that achieves a linear convergence rate and almost linear speedup on a multicore system. But it requires the cost function that satisfies an essential strong convexity property. AsySVRG [13] is an asynchronous SGD variant (SVRG) that adopts a lock-free strategy and convergent with a linear convergence rate. But this algorithm is only designed for multicore systems, which has some limitations to deploy on clusters of multiple machines. HOGWILD! [14] implements SGD in parallel that allows processors to overwrite each other's work without locking, but the gradient updates only modify small parts of the weights to avoid conflicts. Zinkevich *et al.* [15] present a novel data-parallel stochastic gradient descent algorithm to reduce I/O overhead but must place every sample on every machine. However, these algorithms cannot be applied to Spark because Spark collects parameters from all compute nodes using low-frequent synchronous methods like $reduce$ and $aggregate$ rather than the high-frequent pushing and pulling mechanism used by the aforementioned algorithms.

A few approaches have been proposed to improve the performance of calculating SGD based on model parallelism and data parallelism. Zhang *et al.* [16] design an asynchronous SGD system on multiple GPUs working together to calculate gradients and update the global model parameters. However, when extending it to a multi-server multi-GPU architecture, the performance becomes poor due to the network bottleneck. DistBelief [4] is a software framework that introduces two algorithms, Downpour SGD and Sandblaster, for large-scale distributed training using tens of thousands of CPU cores. GPU A-SGD [17] is a new system that makes use of both model parallelism and data parallelism which is similar to DistBelief [4] but with GPUs to speed up training of convolutional neural networks. However, none of these system is compatible with Spark framework because all of them need intensive communications through a centralized parameter server, which is hard to be implemented efficiently on Spark.

To improve the training efficiency of gradient descent, there are several projects to develop adaptive learning rate techniques. Jacobs [7] analyzes why the steepest descent is slow to converge and propose four heuristics to speed up the convergence. Simple adaptive momentum (SAM) [18] dynamically adjusts the momentum-coefficient by the similarities between the current weights and previous weights to reduce the negative effect of overshooting the target. [19] introduces a fast convergent algorithm based on Fletcher-Reeves update by adaptively changing the gradient search direction. Unfortunately, none of them is implemented distributively on a cluster, and have no guarantee of the convergence.

There are also some papers to discuss the optimizations based on platforms like Hadoop and Spark or some specific hardwares [20, 21]. Zhang *et al.* [22] employ a caching technique to avoid disk I/O for short jobs. HogWild++ [23] is a novel decentralized asynchronous SGD algorithm which replaces the global model vector with a set of local model vectors on top of multi-socket NUMA systems. [24] and [25] discuss how to build a computing framework to support Large-scale applications like Logistic Regression and Linear Support Vector Machines.

VI. Conclusions and Future Work

In this work, we design an asynchronous parallel SGD algorithm with iterative local search, called FTSGD, by reusing data partition multiple times within a single global iteration. We also design a variant of momentum algorithm to find the optimal step size on every iteration, which uses a new adaptive rule to decrease the step size whenever the neighboring gradients have been shown in different directions. The experiments show that our algorithm is more efficient and reaches convergence faster than the MLlib library.

For the future work, we plan to compare the performance of FTSGD with some state-of-the-art SGD optimization algorithms like Adadelta and Adam [8, 9]. We also plan to do more experiments on other benchmarks such as Logistic Regression and Linear SVM.

VII. Acknowledgement

This work was supported in part by NSF-CAREER-1622292 and NSF-1741431.

References

[1] H. Zhang, L. Wang, and H. Huang. Smarth: Enabling multi-pipeline data transfer in hdfs. In *ICPP*, 2014.

[2] http://hadoop.apache.org/. Apache Hadoop website.

[3] https://spark.apache.org/. Apache Spark website.

[4] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[5] M. Li, D. G. Andersen, J. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[6] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola. Parameter server for distributed machine learning. In *NIPS*, 2013.

[7] R. A. Jacobs. Increased rates of convergence through learning rate adaptation. In *Neural Networks:Volume 1, Issue 4*, 1988.

[8] M. D. Zeiler. Adadelta: An adaptive learning rate method. In *arXiv:1212.5701*, 2012.

[9] D. P. Kingma and J. Ba. Adam: a method for stochastic optimization. In *3rd International Conference for Learning Representations*, 2015.

[10] Wikipedia for kolmogorov-smirnov test. https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test.

[11] Wikipedia website for cosine similarity. https://en.wikipedia.org/wiki/Cosine_similarity.

[12] J. Liu, S. J. Wright, C. R, V. Bittorf, and S. Sridhar. An asynchronous parallel stochastic coordinate descent algorithm. In *Machine Learning Research 16*, 1988.

[13] S. Zhao and W. Li. Fast asynchronous parallel stochastic gradient descent: A lock-free approach with convergence guarantee. In *AAAI-16*, 2016.

[14] F. Niu, B. Recht, C. Re, and S. Wright. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *NIPS*, 2011.

[15] M. A. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, 2010.

[16] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *ICASSP*, 2013.

[17] T. Paine, H. Jin, J. Yang, Z. Lin, and T. Huang. Gpu asynchronous stochastic gradient descent to speed up neural network training. In *CVPR*, 2013.

[18] D. J. Swanston, J. M. Bishop, and R. J. Mitchell. Simple adaptive momentum: new algorithm for training multilayer perceptrons. In *Electronics Letters*, 1994.

[19] N. M. Nawi, R. S. Ransing, and M. R. Ransing. An improved conjugate gradient based learning algorithm for back propagation neural networks. In *International Journal of Computational Intelligence 4*, 2008.

[20] P. Guo, H. Huang, Q. Chen, L. Wang, E. Lee, and P. Chen. A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 2011.

[21] H. Huang, J. Dennis, L. Wang, and P. Chen. A scalable parallel lsqr algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography. *Procedia Computer Science*, 18, 2013.

[22] H. Zhang, H. Huang, and L. Wang. Mrapid: An efficient short job optimizer on hadoop. In *IPDPS*, 2017.

[23] H. Zhang, C. Hsieh, and V. Akella. Hogwild++: A new mechanism for decentralized asynchronous stochastic gradient descent. In *ICDM*, 2016.

[24] C. Lin, C. Tsai, C. Lee, and C. Lin. Large-scale logistic regression and linear support vector machines using spark. In *IEEE Big Data*, 2014.

[25] Z. Liu, H. Zhang, , and L. Wang. Hierarchical spark: A multi-cluster big data computing framework. In *IEEE Cloud*, 2017.

[26] H. Zhang, Z. Sun, Z. Liu, C. Xu, and L. Wang. Dart: A geographic information system on hadoop. In *IEEE: Cloud*, 2015.

[27] Wikipedia for the 68-95-99.7_rule. https://en.wikipedia.org/wiki/68-95-99.7_rule.

[28] S. Gupta, W. Zhang, and F. Wang. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *ICDM*, 2016.

[29] Z. Huo and H. Huang. Asynchronous mini-batch gradient descent with variance reduction for non-convex optimization. In *AAAI*, 2017.