Limited-memory Common-directions Method for Distributed Optimization and its Application on Empirical Risk Minimization

Ching-pei Lee* Po-Wei Wang† Weizhu Chen‡ Chih-Jen Lin§

Abstract

Distributed optimization has become an important research topic for dealing with extremely large volume of data available in the Internet companies nowadays. Additional machines make computation less expensive, but inter-machine communication becomes prominent in the optimization process, and efficient optimization methods should reduce the amount of the communication in order to achieve shorter overall running time. In this work, we utilize the advantages of the recently proposed, theoretically fast-convergent common-directions method, but tackle its main drawback of excessive spatial and computational costs to propose a limited-memory algorithm. The result is an efficient, linear-convergent optimization method for parallel/distributed optimization. We further discuss how our method can exploit the problem structure to efficiently train regularized empirical risk minimization (ERM) models. Experimental results show that our method outperforms state-of-the-art distributed optimization methods for ERM problems.

1 Introduction

Distributed optimization is now an active research topic because of the rapid growth of data volume. By using multiple machines in distributed optimization, each machine shares a lower computational burden, but as the expensive inter-machine communication cost emerges, the overall running time may not be shortened. However, it is a must to use multiple machines to store and operate on these high-volume data. Therefore the question to ask is not "Can we accelerate the optimization process by multiple machines," but "How do we make the optimization procedure in distributed environments more efficient." In order to conduct distributed optimization efficiently, we should carefully consider methods that address the issue of the communication.

Recently, [17] proposed the common-directions method, which has both the optimal global-linear convergence rate for first-order methods and local-quadratic convergence. For empirical risk minimization (ERM) problems, experiments in [17] show that among state-of-the-art batch optimization methods, their algorithm has the fewest number of data passes (i.e., pass through the whole training data), which is proportional to the rounds of communication. These properties of fast convergence and few data passes are desirable for distributed optimization. However, [17] combines all previous gradients to form the update direction, so the computational and spatial costs of involving past gradients from the first iteration on can grow unlimitedly. This high spatial cost and the expense of some non-parallelizable operations make it less ideal for commonly seen distributed environments such that the machines being used have only limited memory and computational capability.

In this work, to remedy the unbounded growing spatial and computational costs of the common-directions method, we utilize the idea of how limited-memory BFGS (L-BFGS) [9] is derived from BFGS to propose a limited-memory common-directions framework for unconstrained smooth optimization. The proposed algorithm has an upper-bounded per-iteration computational cost, and consumes a controllable amount of memory just like L-BFGS, meanwhile possesses faster theoretical/empirical convergence from the common-directions method because of using the Hessian information. Our method is highly parallelizable, and the fast convergence inherited from the common-directions method makes our method communication-efficient. These properties make our method suitable for large-scale distributed or parallel optimization.

Besides past gradients used in the common-directions method in [17], we investigate combinations of other vectors to form the update direction. We utilize the ideas from the heavy-ball method [14] and L-BFGS [9]. Results show our choices of vectors for combination have better convergence in the limited-memory setting. We provide convergence analysis for a general framework that includes our method. Empirical studies show our method pushes forward state of the art of distributed batch optimization for ERM.

This work is organized as follows. Section 2 describes motivations and details of our method. The theoretical convergence is in Section 3. We illustrate efficient implementations for the ERM problems in Section 4. Section 5 discusses related works. Empirical comparisons are conducted in Section 6. Section 7 concludes this work. The program used in this paper is available at `http://www.csie.ntu.edu.tw/~cjlin/papers/l-commdir/`.

*University of Wisconsin-Madison. `ching-pei@cs.wisc.edu`. Parts of this work were done when this author was at Microsoft.

†Carnegie Mellon University. `poweiw@cs.cmu.edu`

‡Microsoft. `wzchen@microsoft.com`

§National Taiwan University. `cjlin@csie.ntu.edu.tw`

2 Our Method

We consider optimizing the following problem.

$$\min_{\boldsymbol{w}\in\mathbf{R}^n} \quad f(\boldsymbol{w}), \tag{2.1}$$

where f is strictly convex, lower-bounded, and differentiable with its gradient being ρ-Lipschitz continuous. That is, there exists $\rho > 0$ such that for any $\boldsymbol{w}_1, \boldsymbol{w}_2$, we have

$$\|\nabla f(\boldsymbol{w}_1) - \nabla f(\boldsymbol{w}_2)\| \leq \rho\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|.$$

Lipschitzness of the gradient indicates that f is twice-differentiable almost everywhere. Therefore, if the Hessian does not exist, we can still use the generalized Hessian [5] in our algorithm. However, for the ease of description, we will always use the notation $\nabla^2 f(\boldsymbol{w})$ to represent either the real or the generalized Hessian.

We will first briefly describe some motivating works, and then propose our algorithm.

2.1 The Common-directions Method. Recently, to improve the convergence of sub-sampled Hessian Newton method for ERM problems [2], the work [16] proposed to incorporate the stochastic Newton step with the update direction at the previous iteration by minimizing the second-order Taylor expansion.

$$\begin{aligned} \min_{\beta_1,\beta_2} \quad & \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k^{\text{SN}} + \frac{1}{2}(\boldsymbol{p}_k^{\text{SN}})^T \nabla^2 f(\boldsymbol{w}_k)\boldsymbol{p}_k^{\text{SN}} \\ \text{subject to} \quad & \boldsymbol{p}_k^{\text{SN}} = \beta_1 \boldsymbol{d}_k + \beta_2 \boldsymbol{p}_{k-1}^{\text{SN}}, \end{aligned} \tag{2.2}$$

where $\boldsymbol{d}_k$ is the stochastic Newton step at the iterate $\boldsymbol{w}_k$, and $\boldsymbol{p}_{k-1}^{\text{SN}}$ is the update direction at the previous iteration. The optimum of this quadratic problem can easily be obtained by solving a 2×2 linear system. Then $\boldsymbol{p}_k^{\text{SN}}$ is taken as the final update direction, and the step size along this direction is chosen by a backtracking line search procedure.

Extending this idea of combining two directions to multiple directions, [17] proposed the common-directions method that at each iteration, uses the gradients from the first iteration on to decide the update direction by solving a $k \times k$ linear system, where k is the iteration counter. This approach is shown to possess the optimal global linear convergence rate of first-order methods as well as local quadratic convergence. Moreover, they showed that the computational cost for ERM using their method can be significantly reduced by exploiting the problem structure. However, a drawback of the common-directions method is that the memory consumption grows linearly with the number of iterations, and the computational cost at the k-th iteration has a factor of $O(k^3)$ for solving the linear system. Moreover, when being applied in distributed environments, it requires a communication of an $O(k^2)$ matrix. These quadratic and cubic factors of k can be prohibitively expensive for large k, which depends on the number of iterations required to solve a problem.

2.2 From BFGS to L-BFGS. BFGS is a quasi-Newton method that uses the curvature information obtained from the gradient differences and the iterate differences to approximate the Newton step. Given an initial positive definite B_0, at the current iterate $\boldsymbol{w}_k$, BFGS constructs a symmetric positive definite matrix B_k to approximate $(\nabla^2 f(\boldsymbol{w}_k))^{-1}$, and the update direction $\boldsymbol{p}_k^{\text{BFGS}}$ is obtained by

$$\boldsymbol{p}_k^{\text{BFGS}} = -B_k \nabla f(\boldsymbol{w}_k). \tag{2.3}$$

The matrix B_k is constructed by

$$\begin{aligned} B_k \equiv & V_{k-1}^T \cdots V_1^T B_0 V_1 \cdots V_{k-1} \\ & + \sum_{j=1}^{k-1} \rho_j V_{k-1}^T \cdots V_{j+1}^T s_j s_j^T V_{j+1} \cdots V_{k-1}, \end{aligned} \tag{2.4}$$

where

$$\begin{aligned} & V_j \equiv I - \rho_j \boldsymbol{s}_j \boldsymbol{u}_j^T, \quad \rho_j \equiv 1/(\boldsymbol{u}_j^T \boldsymbol{s}_j), \quad \boldsymbol{u}_j \equiv \boldsymbol{w}_{j+1} - \boldsymbol{w}_j, \\ & \boldsymbol{s}_j \equiv \nabla f(\boldsymbol{w}_{j+1}) - \nabla f(\boldsymbol{w}_j). \end{aligned} \tag{2.5}$$

A widely used choice of B_0 is $B_0 = \tau I$ for some $\tau > 0$. To avoid explicitly computing and storing B_k, by (2.4), one can use a sequence of vector operations to compute (2.3). After deciding the update direction, a line search procedure is conducted to ensure the convergence.

It is known that BFGS has local superlinear convergence [12, Theorem 6.6]. However, the major drawback of BFGS is that the computational and spatial costs of obtaining the update direction via (2.4) grows linearly with the number of iterations passed, so for large-scale or difficult problems, BFGS may be impractical. Therefore, [9] proposed its limited-memory version, L-BFGS. The idea is that given a user-specified integer $m > 0$, only $\boldsymbol{u}_j, \boldsymbol{s}_j$ for $j = k - m, \ldots, k-1$ are kept. The definition of B_k then changes from (2.4) to

$$\begin{aligned} B_k \equiv & V_{k-1}^T \cdots V_{k-m}^T B_0^k V_{k-m} \cdots V_{k-1} \\ & + \sum_{j=k-m}^{k-1} \rho_j V_{k-1}^T \cdots V_{j+1}^T s_j s_j^T V_{j+1} \cdots V_{k-1}, \end{aligned} \tag{2.6}$$

where B_0^k are positive definite matrices that can be changed with k. The most common choice is

$$B_0^k = (\boldsymbol{u}_{k-1}^T \boldsymbol{s}_{k-1}/(\boldsymbol{s}_{k-1}^T \boldsymbol{s}_{k-1}))I, \tag{2.7}$$

where I is the identity matrix. Because earlier history is discarded, L-BFGS no longer possesses local superlinear convergence, but as shown in [9], it is linear-convergent. Experiments in [17] showed that L-BFGS is competitive with BFGS for linear classification under a single-core setting.

It is observed in [4] that if B_0^k in (2.6) is a multiple of the identity matrix like (2.7), the L-BFGS step $\boldsymbol{p}_k^{\text{BFGS}}$ is a linear combination of $\boldsymbol{u}_j, \boldsymbol{s}_j$, $j = k-m, \ldots, k-1$, and the current gradient $\nabla f(\boldsymbol{w}_k)$. They utilized this property to efficiently parallelize the L-BFGS method.

2.3 The Proposed Method. Since the common-directions method shares the same major drawback with BFGS, the success of the L-BFGS algorithm derived from BFGS motivates us to consider a limited-memory modification of the common-directions method. Our method absorbs the advantages of the common-directions method and ideas for using only recent historical vectors to have the following settings.

Consider a matrix P_k so that its columns include m selected vectors. We consider a linear combination of these vectors to form the search direction at the k-th iteration. The coefficients for the linear combination are decided by solving the following quadratic program that involves the Hessian $\nabla^2 f(\boldsymbol{w}_k)$ at $\boldsymbol{w}_k$.

$$\begin{aligned} \min_{\boldsymbol{t}_k} \quad & \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k + \frac{1}{2} \boldsymbol{p}_k^T \nabla^2 f(\boldsymbol{w}_k) \boldsymbol{p}_k \\ \text{subject to} \quad & \boldsymbol{p}_k = P_k \boldsymbol{t}_k. \end{aligned} \tag{2.8}$$

Our settings differs from that in [17] in two aspects.

1. A fixed number of vectors rather than k vectors, where k is the iteration index, are used.
2. Instead of using past gradients to form the matrix P_k, we will investigate other choices.

Regarding our setting and L-BFGS, they differ in the following aspects.

1. We solve (2.8) to decide the coefficients for combining vectors, while [4] uses (2.6) to derives coefficients for the vector combination in L-BFGS.
2. For the candidate vectors to be combined, L-BFGS considers $\boldsymbol{u}_i$, $\boldsymbol{v}_j$ in (2.5) derived from approximating the Hessian inverse while ours allows flexible choices.

We note that (2.8) is related to the second-order approximation of $f(\boldsymbol{w})$ in obtaining the Newton direction:

$$\min_{\boldsymbol{p}_k \in \mathbf{R}^n} \quad \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k + \frac{1}{2} \boldsymbol{p}_k^T \nabla^2 f(\boldsymbol{w}_k) \boldsymbol{p}_k \tag{2.9}$$

They both consider the Hessian matrix. However, (2.8) is a simplified form of (2.9) by converting from $\boldsymbol{p}_k \in \mathbf{R}^n$ to $\boldsymbol{p}_k$ in the column space of P_k.

2.3.1 Choices of P_k. In general we let P_k have $m+1$ columns, in which m vectors are obtained from the previous m iterations, and the current gradient $\nabla f(\boldsymbol{w}_k)$ is included to ensure convergence. The most natural choice of P_k is to discard older gradients in the common-directions method, resulting in

$$P_k = [\nabla f(\boldsymbol{w}_{k-m}), \ldots, \nabla f(\boldsymbol{w}_k)] \in \mathbf{R}^{n \times (m+1)}. \tag{2.10}$$

However, in a limited-memory setting, empirically we find that the gradients may not be sufficient to capture the curvature information to ensure good convergence. We thus also consider other choices.

The first one extends the idea from the heavy-ball method [14] to use the updates at the previous iterations. The heavy-ball method updates the iterate by

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \alpha \nabla f(\boldsymbol{w}_k) + \beta \boldsymbol{u}_{k-1}$$

with some pre-specified α and β. We thus consider $\boldsymbol{u}_j = \boldsymbol{w}_{j+1} - \boldsymbol{w}_j$ for $j = k-m, \ldots, k-1$, so

$$P_k \equiv [\boldsymbol{u}_{k-m}, \ldots, \boldsymbol{u}_{k-1}, \nabla f(\boldsymbol{w}_k)] \in \mathbf{R}^{n \times (m+1)}. \tag{2.11}$$

For the second choice, we mentioned in Section 2.2 that the L-BFGS step is a linear combination of $\boldsymbol{u}_j$, $\boldsymbol{s}_j$, $j = k-m, \ldots, k-1$, and $\nabla f(\boldsymbol{w}_k)$. This property suggests that we can also consider these directions to form P_k.

$$[\boldsymbol{u}_{k-m}, \boldsymbol{s}_{k-m}, \ldots, \boldsymbol{u}_{k-1}, \boldsymbol{s}_{k-1}, \nabla f(\boldsymbol{w}_k)] \in \mathbf{R}^{n \times (2m+1)}. \tag{2.12}$$

Note that under the same m, this setting uses m more vectors in P_k than (2.10) and (2.11).

In Section 6, we empirically confirm the advantages of considering the choices of (2.11) or (2.12) over (2.10) in solving logistic regression problems. As our method uses the real Hessian information to combine these vectors more wisely, one can expect that it converges at least as good as, if not better than, the heavy-ball method and L-BFGS. Finally, we note that the choice of P_k is not limited to (2.10)-(2.12).

2.3.2 Solving (2.8). We can get the solution of (2.8) by solving the following.

$$P_k^T \nabla^2 f(\boldsymbol{w}_k) P_k \boldsymbol{t} = -P_k^T \nabla f(\boldsymbol{w}_k). \tag{2.13}$$

Obtaining $P_k^T \nabla^2 f(\boldsymbol{w}_k) P_k$ and then solving (2.13) may be expensive. However, for problems with special structures (e.g., linear risk minimization in Section 4), we are able to compute it cheaply. Because the columns of P_k may be linearly dependent, $P_k^T \nabla^2 f(\boldsymbol{w}_k) P_k$ may be positive semi-definite rather than positive definite. We can use the pseudo inverse (denoted as A^+ of a matrix A) to obtain a solution for (2.13). Thus, if $P_k^T \nabla^2 f(\boldsymbol{w}_k) P_k$ and $P_k^T \nabla f(\boldsymbol{w}_k)$ have been calculated, then the computational cost for solving (2.13) is $O(m^3)$, which is negligible when m is small.

2.3.3 Overall Procedure After solving (2.13), we take $\boldsymbol{p}_k = P_k \boldsymbol{t}_k$, and conduct a backtracking line search to ensure global convergence. Specifically, we find the minimum nonnegative integer i such that for a given $\beta \in (0,1)$, $\theta_k = \beta^i$ satisfies

$$f(\boldsymbol{w}_k + \theta_k \boldsymbol{p}_k) \leq f(\boldsymbol{w}_k) + c_1 \theta_k \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k, \tag{2.14}$$

for some pre-specified $0 < c_1 < 1$. The next iterate is then obtained by $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \theta_k \boldsymbol{p}_k$.

To update from P_k to P_{k+1}, new vectors are needed. We discuss the most complex case (2.12), and the other two

Algorithm 1: Limited-memory common-directions method

Given $\boldsymbol{w}_0, m > 0$, compute $\nabla f(\boldsymbol{w}_0)$ and an initial P that includes $\nabla f(\boldsymbol{w}_0)$

for *k=0,1,...* **do**

- Compute $P^T \nabla^2 f(\boldsymbol{w}_k) P$ and $P^T \nabla f(\boldsymbol{w}_k)$
- Let $\boldsymbol{t}_k = -(P^T \nabla^2 f(\boldsymbol{w}_k) P)^+ P^T \nabla f(\boldsymbol{w}_k)$
- $\Delta \boldsymbol{w} = P \boldsymbol{t}_k$
- Backtracking line search on $f(\boldsymbol{w}_k + \theta \Delta \boldsymbol{w})$ to obtain θ_k that satisfies (2.14)
- $\boldsymbol{u}_k = \theta_k \Delta \boldsymbol{w}$
- $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \boldsymbol{u}_k$
- Compute $\nabla f(\boldsymbol{w}_{k+1})$
- Update P by some choice that includes $\nabla f(\boldsymbol{w}_{k+1})$

end

choices (2.10) and (2.11) follow the same technique. For (2.12), three vectors $\boldsymbol{u}_k, \boldsymbol{s}_k$, and $\nabla f(\boldsymbol{w}_{k+1})$ are needed. After line search, we have

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \theta_k P \boldsymbol{t}_k = \boldsymbol{w}_k + \boldsymbol{u}_k, \tag{2.15}$$

so $\boldsymbol{u}_k$ is obtained. We then calculate $\nabla f(\boldsymbol{w}_{k+1})$ that is needed in the next iteration, and $\boldsymbol{s}_k$ is obtained by (2.5).

Algorithm 1 summarizes our framework, which allows a flexible choice of P_k and requires only that the current gradient $\nabla f(\boldsymbol{w}_k)$ is included in P_k. Although we describe the framework in a sequential manner, with suitable choices of P_k, it is possible that most computation-heavy steps are parallelizable. We will demonstrate an example in Section 4.

Regarding the number of vectors included in P_k, one can expect that larger m leads to faster empirical convergence, but has larger cost per iteration because the linear system (2.13) becomes larger.

3 Convergence Analysis

We now discuss the convergence of our method to understand the overall cost. To have a more general analysis we relax the condition on (2.1) so that $f(\boldsymbol{w})$ may not even be convex.

ASSUMPTION 1. *The objective is bounded below, differentiable, and has ρ-Lipschitz continuous gradient with $\rho > 0$.*

By replacing the real Hessian $\nabla^2 f(\boldsymbol{w}_k)$ in (2.8) with a matrix, we consider a more general setting in finding coefficients for combining vectors.

$$\begin{aligned} \min_{\boldsymbol{t}_k} \quad & \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k + \frac{1}{2} \boldsymbol{p}_k^T H_k \boldsymbol{p}_k \\ \text{subject to} \quad & \boldsymbol{p}_k = P_k \boldsymbol{t}_k, \end{aligned} \tag{3.16}$$

where vectors to be combined are included in

$$P_k \equiv [\boldsymbol{q}_1, \ldots, \boldsymbol{q}_m] \in \mathbf{R}^{n \times m}.$$

Note that for easy description we let m be the number of columns in P_k though in (2.10)-(2.12) we have $m+1$ or $2m+1$ vectors. In (3.16), we require that H_k is positive definite and there exist $M_1 \geq M_2 > 0$ satisfying

$$M_1 I \succeq H_k \succeq M_2 I, \forall k. \tag{3.17}$$

If f is strongly convex, then the choice of the real Hessian in Algorithm 1 satisfies (3.17), as we discuss below in Corollary 3.1.

The following theorems show the finite termination of line search and the iteration complexity of our method. All the proofs are in the supplementary material.

THEOREM 3.1. *Consider* (3.16) *for $f(\boldsymbol{w})$ satisfying Assumption 1. If at iteration k,*

$$\frac{\left|\nabla f(\boldsymbol{w}_k)^T \boldsymbol{q}_j\right|}{\|\nabla f(\boldsymbol{w}_k)\| \|\boldsymbol{q}_j\|} \geq \delta > 0, \text{ for some } q_j \text{ in } P_k, \tag{3.18}$$

then backtracking line search for (2.14) *with any given $\beta, c_1 \in (0,1)$ terminates in finite steps.*

THEOREM 3.2. *Assume $f(\boldsymbol{w})$ satisfies Assumption 1. In an algorithm of sequentially solving* (3.16) *that satisfies the conditions* (3.17) *and* (3.18)*, and using the solution as the update direction with backtracking line search, the minimum of the norm of gradients of the iterates vanishes at an $O(1/\epsilon^2)$ rate:*

$$\min_{0 \leq j \leq k} \|\nabla f(\boldsymbol{w}_j)\| = O(1/\sqrt{k+1}), \tag{3.19}$$

and $\|\nabla f(\boldsymbol{w}_k)\|$ also converges to zero as k approaches infinity. If in addition $f(\boldsymbol{w})$ satisfies the Polyak-Łojasiewicz condition [10, 13, 6] for some $\sigma > 0$, i.e.,

$$\|\nabla f(\boldsymbol{w})\|^2 \geq 2\sigma(f(\boldsymbol{w}) - f^*), \ \ \forall \boldsymbol{w}, \tag{3.20}$$

where f^* is the optimum of $f(\boldsymbol{w})$, then the function values linearly converge. That is, it takes $O(\log(1/\epsilon))$ iterations to get an ϵ-accurate solution satisfying

$$f(\boldsymbol{w}_k) - f^* \leq \epsilon.$$

Note that if $\{\boldsymbol{w}_k\}$ is in a compact set, then it has convergent sub-sequences, and every limit point of $\{\boldsymbol{w}_k\}$ is stationary because the gradient vanishes.

Our theorems provide convergence not only for our algorithm, but also many others. For example, the algorithm in [16] for convex L2-regularized ERM, whose analysis proved (3.18) and (3.20) but only showed asymptotic convergence, is linearly convergent by using our results. Our result also provides a convergence-rate result of (3.19) for their algorithm on the nonconvex neural network problem; see Section VI in the supplementary material. Similarly, the algorithm of

combining past gradients in [17] can also be treated as a special case of our framework, and our analysis coincides with their linear convergence result. However, we notice that the convergence rate of their another algorithm that uses multiple inner iterations before updating the possible directions is better than the result one can obtain using the analysis flow here, as that algorithm does not fit in our framework.

If f is strongly convex, for Algorithm 1, we show (3.20) holds and establish the linear convergence.

COROLLARY 3.1. *Consider using Algorithm 1 to solve* (2.1) *satisfying Assumption 1 with f being σ-strongly convex for some $\sigma \in (0, \rho]$. If the backtracking line search procedure picks the step size as the largest $\beta^i, i = 0, 1, \ldots$ that satisfies* (2.14) *for some given $\beta, c_1 \in (0, 1)$, then at each iteration, the line search procedure terminates within $\lceil \log_\beta(\beta(1 - c_1)(\sigma/\rho)) \rceil$ steps, and the number of iterations needed to attain an ϵ-accurate solution is $O(\log(1/\epsilon)/\log(1/(1-2\beta c_1(1-c_1)\sigma^3/\rho^3)))$ for any $\epsilon > 0$.*

When f is convex but not strongly convex, (3.17) may not hold and we cannot apply Corollary 3.1. In this case, we can use $H_k = \nabla^2 f(\boldsymbol{w}_k) + \tau I$ with some $\tau > 0$ so that

$$(\rho + \tau)I \succeq H_k \succeq \tau I.$$

Then from Theorem 3.2 we have the result in (3.19) by letting $(M_1, M_2) = (\rho + \tau, \tau)$ in (3.17).

4 Application on Distributed Optimization for ERM

We apply Algorithm 1 to distributedly solve the L2-regularized ERM problem.

$$\min_{\boldsymbol{w} \in \mathbf{R}^n} \quad f(\boldsymbol{w}) \equiv \boldsymbol{w}^T\boldsymbol{w}/2 + C\sum_{i=1}^{l} \xi(y_i; \boldsymbol{w}^T\boldsymbol{x}_i), \tag{4.21}$$

where $C > 0$ is a parameter specified by users, $(y_i, \boldsymbol{x}_i)$, $i = 1, \ldots, l$ are the training instances, such that $\boldsymbol{x}_i \in \mathbf{R}^n$ are the features while y_i are the labels, and the loss function ξ is convex with respect to the second argument. We simplify our notation by $\xi_i(\boldsymbol{w}^T\boldsymbol{x}_i) \equiv \xi(y_i; \boldsymbol{w}^T\boldsymbol{x}_i)$ and consider an ξ function that makes $f(\boldsymbol{w})$ in (4.21) satisfy Assumption 1. A typical example is the logistic regression. For distributed optimization, we assume the data set is split across K machines in a instance-wise manner: $J_r, r = 1, \ldots, K$ form a disjoint partition of $\{1, \ldots, l\}$, and the r-th machine stores the instances $(y_i, \boldsymbol{x}_i)$ for $i \in J_r$.[1] We then split the P_k matrix to K parts by their rows. We define $\tilde{J}_r, r = 1, \ldots, K$ as a disjoint partition of $\{1, \ldots, n\}$, and the i-th row of P_k is stored on machine r if $i \in \tilde{J}_r$. Similar to the way P_k is stored, the model vector $\boldsymbol{w}$ is also maintained distributedly under the $\tilde{J}_r, \forall r$ partitions. The partitions can be defined to best parallelize the computation, but for simplicity we assume all $\tilde{J}_r$ are of the same size n/K and the data set is partitioned such that each machine has $\#\text{nnz}/K$ entries, where $\#\text{nnz}$ is the number of nonzero entries in the data. In notation, for a vector $\boldsymbol{\psi}$, we denote $\boldsymbol{\psi}_{J_r}$ as the sub-vector of $\boldsymbol{\psi}$ with the indices in J_r, and for a matrix A, $A_{J_{r_1}, J_{r_2}}$ is the sub-matrix consists of $A_{i,j}, (i, j) \in J_{r_1} \times J_{r_2}$. For simplicity, we assume a master-master framework such that for any non-parallelizable parts, each machine conducts the same task individually. Adapting the algorithm for a master-slave setting should be straightforward.

Under Assumption 1, f in (4.21) is twice-differentiable almost everywhere. Its gradient and (generalized) Hessian [11] are respectively

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + CX^T\boldsymbol{v}_{\boldsymbol{w}}, \tag{4.22}$$

$$\nabla^2 f(\boldsymbol{w}) \equiv I + CX^T D_{\boldsymbol{w}} X, \tag{4.23}$$

where $X^T = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l]$, $D_{\boldsymbol{w}}$ is a diagonal matrix, and

$$(D_{\boldsymbol{w}})_{i,i} \equiv \xi_i''(z) \mid_{z=(X\boldsymbol{w})_i}, i = 1, \ldots, l, \tag{4.24}$$

$$(v_{X\boldsymbol{w}})_i \equiv \xi_i'(z) \mid_{z=(X\boldsymbol{w})_i}, i = 1, \ldots, l. \tag{4.25}$$

The $\boldsymbol{w}^T\boldsymbol{w}/2$ term guarantees that f is 1-strongly convex, and therefore Corollary 3.1 applies here.

Now we propose techniques to make our algorithm more efficient for (4.21). We focus the description on using (2.12) for choosing P_k, as the other two choices follow the same techniques. Because we aim to handle large-scale data, throughout the discussion we assume

$$m \ll l \quad \text{and} \quad m \ll n. \tag{4.26}$$

4.1 Reducing Data-related Costs.

The main computation at each iteration of Algorithm 1 is to construct and solve the linear system (2.13). For ERM, the matrix in (2.13) is

$$P_k^T\nabla^2 f(\boldsymbol{w}_k)P_k = P_k^T P_k + C(XP_k)^T D_{\boldsymbol{w}_k} XP_k. \tag{4.27}$$

We can compute it without explicitly forming the Hessian matrix. We begin with discussing the efficient calculation of the second term, which is related to the data matrix X, while leave details of the first term in Section 4.2.

Following [17], we maintain XP_k and $X\boldsymbol{w}_k$ so that the second term of (4.27) can be cheaply computed in $O(m^2 l)$. Otherwise, $O(\#\text{nnz} \cdot m)$ for directly calculating XP_k is generally more expensive. We begin with the case of a single machine. For obtaining XP_k, we use XP_{k-1} and some cheap operations. From (2.12), three new vectors

$$X\boldsymbol{u}_{k-1}, \quad X\boldsymbol{s}_{k-1}, \quad X\nabla f(\boldsymbol{w}_k) \tag{4.28}$$

must be calculated. From (2.5) and (2.15),

$$X\boldsymbol{s}_{k-1} = X\nabla f(\boldsymbol{w}_k) - X\nabla f(\boldsymbol{w}_{k-1}), \tag{4.29}$$

$$X\boldsymbol{u}_{k-1} = \theta_{k-1}(XP_{k-1})\boldsymbol{t}_{k-1}. \tag{4.30}$$

[1] Implementing our method with splitting the data feature-wisely is also possible, in which case the split of P_k remains the same, but details are omitted because of the space limit.

Because $X\nabla f(\boldsymbol{w}_{k-1})$ is maintained in XP_{k-1}, the main operation for (4.28) is $X\nabla f(\boldsymbol{w}_k)$ that costs $O(\#\text{nnz})$. For obtaining $X\boldsymbol{w}_k$, from (2.15),

$$X\boldsymbol{w}_k = X\boldsymbol{w}_{k-1} + X\boldsymbol{u}_{k-1}, \tag{4.31}$$

where $X\boldsymbol{u}_{k-1}$ is calculated in (4.30).

In our distributed setting, (4.29)-(4.31) become

$$\begin{aligned}(X\boldsymbol{s}_{k-1})_{J_r} &= (X\nabla f(\boldsymbol{w}_k))_{J_r} - (X\nabla f(\boldsymbol{w}_{k-1}))_{J_r},\\ (X\boldsymbol{u}_{k-1})_{J_r} &= \theta_{k-1}(XP_{k-1})_{J_r,:}\boldsymbol{t}_{k-1},\\ (X\boldsymbol{w}_k)_{J_r} &= (X\boldsymbol{w}_{k-1})_{J_r} + (X\boldsymbol{u}_{k-1})_{J_r}.\end{aligned} \tag{4.32}$$

That is, we aim to maintain $(XP_k)_{J_r,:}$ and $(X\boldsymbol{w}_k)_{J_r}$ at the r-th node. If these two terms and $\boldsymbol{t}_{k-1}$ (see Section 4.3) have been available, $(X\boldsymbol{u}_{k-1})_{J_r}$ and $(X\boldsymbol{w}_k)_{J_r}$ can be locally computed. For $(X\boldsymbol{s}_{k-1})_{J_r}$, we need an *allreduce* operation to compute $\nabla f(\boldsymbol{w}_k)$ and make it available to all nodes.

$$\nabla f(\boldsymbol{w}_k) = \bigoplus_{r=1}^{K}\left(CX_{J_r,:}^T(\boldsymbol{v}_{\boldsymbol{w}_k})_{J_r} + \begin{bmatrix}\mathbf{0}\\ (\boldsymbol{w}_k)_{\tilde{J}_r}\\ \mathbf{0}\end{bmatrix}\right), \tag{4.33}$$

where $\oplus$ denotes the *allreduce* operation that gathers results from each node, sums them up, and then broadcasts the result to all nodes. Afterwards, the computation of $(X\nabla f(\boldsymbol{w}_k))_{J_r}$ is conducted locally using only instances on the r-th node. Therefore, the computation of the second term of (4.27) is fully parallelized and thus costs $O((\#\text{nnz} + m^2 l)/K)$ computation for each machine, with one round of $O(n)$ communication to generate the gradient.

Before the *allreduce* operation (4.33), we need $(\boldsymbol{v}_{\boldsymbol{w}_k})_{J_r}$. It together with $(D_{\boldsymbol{w}_k})_{J_r,J_r}$ can be calculated using the locally maintained $(X\boldsymbol{w}_k)_{J_r}$. Then $X_{J_r,:}^T(\boldsymbol{v}_{\boldsymbol{w}_k})_{J_r}$ can also be computed locally. After the *allreduce* operation, $(X\boldsymbol{s}_{k-1})_{J_r}$ can be obtained because $(X\nabla f(\boldsymbol{w}_k))_{J_r}$ involves only local instances and $(X\nabla f(\boldsymbol{w}_{k-1}))_{J_r}$ is maintained from the previous iteration.

Finally, we need an *allreduce* operation on local matrices of size $O(m^2)$ to get the second term in (4.27).

$$\bigoplus_{r=1}^{K}(XP_k)_{J_r,:}^T(D_{\boldsymbol{w}_k})_{J_r,J_r}(XP_k)_{J_r,:}. \tag{4.34}$$

In summary, the computation of the second term of (4.27) is fully parallelized, with $O(n + m^2)$ communication cost.

4.2 Reducing Cost for $P_k^T P_k$. [2] Let $M^k \equiv P_k^T P_k$ be the first term in (4.27). We mentioned in Section 2.2 that in [4] for distributed L-BFGS, their method, called VL-BFGS, combines $\boldsymbol{u}_j, \boldsymbol{s}_j$ vectors in (2.5). Their method also requires calculating M^k (i.e., inner products between P_k's columns). They pointed out that M^{k-1} and M^k share most elements, so only the following new entries must be calculated.

$$P_k^T[\boldsymbol{u}_{k-1}, \boldsymbol{s}_{k-1}, \nabla f(\boldsymbol{w}_k)], \tag{4.35}$$

where $P_k \in \mathbf{R}^{n\times(2m+1)}$. Clearly $6m+6$ inner products are needed. Besides, their method involves $2m$ vector additions (details omitted). The cost is thus higher than $2m$ inner products and $2m$ vector additions in the standard L-BFGS.[3] However, the advantage of [4]'s setting is that the $6m+6$ inner products can be parallelized when P_k is distributedly stored. In this section, we will show that by a careful design,

Algorithm 2: A distributed implementation of Algorithm 1 for (4.21); (2.12) is used for constructing P_k

Given $m > 0$, $\boldsymbol{w}_0$, and partitions J_r, $\tilde{J}_r$
Compute $\boldsymbol{z}_{J_r} = X_{J_r}\boldsymbol{w}_0$
Use $\boldsymbol{z}_{J_r}$ to calculate $(\boldsymbol{v}_{\boldsymbol{w}_0})_{J_r}$ and $(D_{\boldsymbol{w}_0})_{J_r,J_r}$ in (4.25) and (4.24)
Compute $\nabla f(\boldsymbol{w}_0)$ by (4.33)
$P_{\tilde{J}_r,:} = [(\nabla f(\boldsymbol{w}_0))_{\tilde{J}_r}]$, $M^0 = \bigoplus_{r=1}^{K}\|\nabla f(\boldsymbol{w}_0)_{\tilde{J}_r}\|^2$
$(\hat{\boldsymbol{u}}_0)_{J_r} = X_{J_r,:}\nabla f(\boldsymbol{w}_0)$, $U_{J_r} = (\hat{\boldsymbol{u}}_0)_{J_r}$, $\boldsymbol{g} = M^0$
for *k=0,1,...* **do**
  Compute $U^T D_{\boldsymbol{w}_k} U$ by (4.34)
    ▷ $O(m^2 l/K)$; $O(m^2)$ communication
  Obtain $\boldsymbol{t}$ by solving (2.13) with $\boldsymbol{t} = [M^k + CU^T D_{\boldsymbol{w}_k} U]^+[-\boldsymbol{g}]$ ▷ $O(m^3)$
  Compute $\Delta\boldsymbol{z}_{J_r} = U_{J_r,:}\boldsymbol{t}$ ▷ $O(ml/K)$
  Backtracking line search on $f(\boldsymbol{w}_k + \theta\Delta\boldsymbol{w})$ to obtain θ that satisfies (2.14) using $\boldsymbol{z}_{J_r}$ and $\Delta\boldsymbol{z}_{J_r}$ for (4.40)
  $(X\boldsymbol{u}_k)_{J_r} = \theta\Delta\boldsymbol{z}_{J_r}$
  $(\boldsymbol{w}_{k+1})_{\tilde{J}_r} = (\boldsymbol{w}_k)_{\tilde{J}_r} + \theta_k P_{\tilde{J}_r,:}\boldsymbol{t}$ ▷ $O(mn/K)$
  $\boldsymbol{z}_{J_r} = \boldsymbol{z}_{J_r} + (X\boldsymbol{u}_k)_{J_r}$
  Use $\boldsymbol{z}_{J_r}$ to calculate $(\boldsymbol{v}_{\boldsymbol{w}_{k+1}})_{J_r}$ and $(D_{\boldsymbol{w}_{k+1}})_{J_r,J_r}$
  Calculate $\nabla f(\boldsymbol{w}_{k+1})$ by (4.33)
    ▷ $O(\#\text{nnz}/K)$; $O(n)$ communication
  $(\hat{\boldsymbol{u}}_{k+1})_{J_r} = X_{J_r,:}\nabla f(\boldsymbol{w}_{k+1})$ ▷ $O(\#\text{nnz}/K)$
  $(\boldsymbol{s}_k)_{\tilde{J}_r} = \nabla f(\boldsymbol{w}_{k+1})_{\tilde{J}_r} - \nabla f(\boldsymbol{w}_k)_{\tilde{J}_r}$
  $(X\boldsymbol{s}_k)_{J_r} = (\hat{\boldsymbol{u}}_{k+1})_{J_r} - (\hat{\boldsymbol{u}}_k)_{J_r}$
  Update $P_{\tilde{J}_r,:}$ according to (2.12)
  Update $U_{J_r,:} = (XP)_{J_r,:}$ by $(X\boldsymbol{s}_k)_{J_r}$, $(X\boldsymbol{u}_k)_{J_r}$, $(\hat{\boldsymbol{u}}_{k+1})_{J_r}$
  Calculate $\boldsymbol{g} = P^T\nabla f(\boldsymbol{w}_{k+1})$ and $\boldsymbol{s}_k^T\boldsymbol{s}_k$ by (4.38) and (4.39)
    ▷ $O(mn/K)$; $O(m)$ communication
  Update M^{k+1} using (4.37) ▷ $O(m^2)$
end

[2] After this work was published in the proceedings of SIAM 2017 Conference on Data Mining, Wei-Lin Chiang pointed out to us that the approach discussed in this section has already been proposed in [3].

[3] An iteration of standard L-BFGS involves two loops, each of which includes m inner products and m vector additions. See details in [9].

(4.35), or M^k, can be done by $2m+2$ parallelizable inner products in (4.36). Hence, this technique is useful not only for our method but also for improving upon VL-BFGS.

To obtain (4.35), we first calculate

$$P_k^T \nabla f(\boldsymbol{w}_k), \quad \text{and} \quad \boldsymbol{s}_k^T \boldsymbol{s}_k \tag{4.36}$$

by $2m+2$ inner products of vectors in $\mathbf{R}^n$. For $P_k^T \boldsymbol{s}_{k-1}$, from (2.5) and $M^k_{2m+1,i} = (P_k^T \nabla f(\boldsymbol{w}_k))_i$,

$$\left(P_k^T \boldsymbol{s}_{k-1}\right)_i = \left(P_k^T \nabla f\left(\boldsymbol{w}_k\right)\right)_i - \left(P_k^T \nabla f\left(\boldsymbol{w}_{k-1}\right)\right)_i =$$
$$\begin{cases} M^k_{2m+1,i} - M^{k-1}_{2m+1,i+2} & \text{if } i < 2m-1 \\ M^k_{2m+1,i} - \theta_{k-1} M^{k-1}_{2m+1,:} \boldsymbol{t}_{k-1} & \text{if } i = 2m-1 \\ \boldsymbol{s}_{k-1}^T \boldsymbol{s}_{k-1} & \text{if } i = 2m \\ \nabla f\left(\boldsymbol{w}_k\right)^T \boldsymbol{s}_{k-1} = \left(P_k^T \nabla f\left(\boldsymbol{w}_k\right)\right)_{2m} & \text{if } i = 2m+1 \end{cases},$$

where for $i = 2m-1$, we used

$$\left(P_k^T \nabla f\left(\boldsymbol{w}_{k-1}\right)\right)_{2m-1} = \nabla f(\boldsymbol{w}_{k-1})^T \boldsymbol{u}_{k-1}$$
$$= \theta_{k-1} \nabla f(\boldsymbol{w}_{k-1})^T P_{k-1} \boldsymbol{t}_{k-1} = \theta_{k-1} M^{k-1}_{2m+1,:} \boldsymbol{t}_{k-1}.$$

Next, for $P_k^T \boldsymbol{u}_{k-1}$, we have

$$\left(P_k^T \boldsymbol{u}_{k-1}\right)_i = \begin{cases} \theta_{k-1} M^{k-1}_{i+2,:} \boldsymbol{t}_{k-1} & \text{if } i < 2m-1 \\ \theta_{k-1}^2 \boldsymbol{t}_{k-1}^T M^{k-1} \boldsymbol{t}_{k-1} & \text{if } i = 2m-1 \\ \left(P_k^T \boldsymbol{s}_{k-1}\right)_{2m-1} & \text{if } i = 2m \\ (P_k^T \nabla f(\boldsymbol{w}_k))_{2m-1} & \text{if } i = 2m+1 \end{cases},$$

where detailed derivations are in the supplementary material. The rest terms are available from M^{k-1} because they are inner products between vectors in P_{k-1}. In summary,

$$M^k_{i,j} = \begin{cases} M^{k-1}_{i+2,j+2}, & \text{if } i,j < 2m-1 \\ \left(P_k^T \boldsymbol{u}_{k-1}\right)_i, & \text{if } j = 2m-1 \\ \left(P_k^T \boldsymbol{s}_{k-1}\right)_i, & \text{if } j = 2m \\ \left(P_k^T \nabla f\left(\boldsymbol{w}_k\right)\right)_i, & \text{if } j = 2m+1 \\ M^k_{j,i}, & \text{if } i \geq 2m-1 > j \end{cases}. \tag{4.37}$$

In the above calculation, the most expensive operation takes $O(m^2)$ cost for $M^{k-1}\boldsymbol{t}_{k-1}$ in obtaining $P_k^T \boldsymbol{u}_{k-1}$. This is smaller than the inner products in (4.36).

In our distributed setting, we compute and make $P_k^T \nabla f(\boldsymbol{w}_k)$ and $\boldsymbol{s}_{k-1}^T \boldsymbol{s}_{k-1}$ available at all nodes by the following *allreduce* operations with $O(mn/K)$ cost and $O(m)$ communication.

$$P_k^T \nabla f\left(\boldsymbol{w}_k\right) = \bigoplus_{r=1}^K \left(P_k\right)^T_{\tilde{J}_r,:} \nabla f\left(\boldsymbol{w}_k\right)_{\tilde{J}_r}, \tag{4.38}$$

$$\boldsymbol{s}_{k-1}^T \boldsymbol{s}_{k-1} = \bigoplus_{r=1}^K \left(\boldsymbol{s}_{k-1}\right)^T_{\tilde{J}_r} \left(\boldsymbol{s}_{k-1}\right)_{\tilde{J}_r}, \tag{4.39}$$

By maintaining M^k and $\boldsymbol{t}_k$ on all nodes (see Section 4.3), we have that all other operations can be conducted locally without any communication.

4.3 Solving the Linear System (2.13)**.** Once the matrix in (4.27) is generated, we must solve a linear system of m variables. Because m is small from (4.26), it may not be cost-effective to solve (2.13) distributedly. We thus make the information of the linear system available at all computing nodes, and each node takes $O(m^3)$ to obtain the same $\boldsymbol{t}_k$. To have the matrix (4.27) of the linear system, we mentioned in Section 4.2 that all nodes maintain the same M^k while the second term is from the *allreduce* operation in (4.34). For the right-hand side vector $P_k^T \nabla f(\boldsymbol{w}_k)$, we have seen from (4.38) that it is available at all nodes.

4.4 Line Search. Similar to how we evaluate the new $D_{\boldsymbol{w}}$ and $\boldsymbol{v}_{\boldsymbol{w}}$ efficiently in Section 4.1, one can compute the loss function in the line search procedure by

$$\bigoplus_{r=1}^K \left(\sum\nolimits_{i \in J_r} \xi_i((X\boldsymbol{w}_k)_i + \theta(XP_k)_{i,:}\boldsymbol{t}_k)\right). \tag{4.40}$$

With the availability of $(X\boldsymbol{w}_k)_{J_r}$ and $(XP_k)_{J_r,:}$, this cheaply costs $O(l/K)$ in computation, and $O(1)$ in communication for *allreduce*. Note that $\nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k$ needed in (2.14) for line search can be obtained in $O(m)$ by $M^k_{2m+1,:}\boldsymbol{t}_k$ rather than an inner product between two $O(n)$ vectors.

4.5 Cost Analysis. We list details of applying Algorithm 1 to (4.21) in a distributed environment with instance-wise data split in Algorithm 2. The computational cost per iteration per machine is

$$O\left(\frac{\#\text{nnz} + m^2 l + l \times \#\left(\text{line search}\right) + mn}{K} + m^3\right),$$

where from (4.26), #nnz is in general the dominant term. For the communication, the cost per iteration is

$$O\left(n + m^2 + \#\left(\text{line search}\right)\right).$$

From Theorem 3.1, #(line search) is bounded by a constant, and in practice (2.14) is often satisfied at $\theta_k = 1$.

5 Related Works

Besides L-BFGS, another effective batch distributed optimization method is the truncated Newton method. At each iteration, the second-order approximation in (3.16) is considered. The update direction $\boldsymbol{p}_k^{\text{N}}$ is obtained by approximately solving the following linear system.

$$\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{p}_k^{\text{N}} = -\nabla f(\boldsymbol{w}_k). \tag{5.41}$$

Then either line search or trust region methods are applied to decide the update from $\boldsymbol{w}_k$ to $\boldsymbol{w}_{k+1}$. To solve (5.41), a Hessian-free approach is considered by using the conjugate gradient (CG) method, where a sequence of Hessian-vector products is needed. Take (4.21) as an example. From (4.23),

$$\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{d} = \boldsymbol{d} + X^T\left(D_{\boldsymbol{w}_k}\left(X\boldsymbol{d}\right)\right), \tag{5.42}$$

Table 1: Data statistics.

Data set	#instances	#features	#nonzeros
criteo	45,840,617	1,000,000	1,787,773,969
kdd2012.tr	119,705,032	54,686,452	1,316,755,352
url	2,396,130	3.231,961	277,058,644
KDD2010-b	19,264,097	29,890,096	566,345,888
epsilon	400,000	2,000	800,000,000
webspam	350,000	16,609,143	1,304,697,446
news20	19,996	1,355,191	9,097,916
rcv1t	677,399	47.226	49,556,258

so we need only to store X rather than $\nabla^2 f(\boldsymbol{w}_k)$. Our method also utilizes the Hessian matrix to decide the update direction. However, for the Newton-type approaches, all CG iterations within one Newton iteration use the same Hessian matrix, but our method updates the Hessian more frequently.

In a distributed setting, X is stored across machines, so each operation of (5.42) requires the communication. For example, if an instance-wise split is used, we need

$$X^T D_{\boldsymbol{w}} X \boldsymbol{d} = \bigoplus_{r=1}^{K} \sum\nolimits_{i \in J_r} \boldsymbol{x}_i (D_{\boldsymbol{w}})_{i,i} (\boldsymbol{x}_i^T \boldsymbol{d})$$

with one *allreduce* operation. This communication cost is similar to that in our one iteration. The works [19, 8] extended the single-machine trust region Newton method (TRON) [15] for logistic regression in [7] to distributed environments. Experiments in [19] show that TRON is faster than another distributed optimization method ADMM [18, 1] when both are implemented in MPI.

6 Experiments

We present results on solving (4.21) distributedly. We consider logistic regression, whose loss term $\xi_i(\cdot)$ in (4.21) is $\xi_i(z) \equiv \log(1 + \exp(-y_i z))$, with $y_i \in \{-1, 1\}$. The distributed environment is a cluster with 16 nodes. In Table 1, we give statistics of problems obtained from the LIBSVM data sets.[4]

We compare the following methods for logistic regression by checking the relative difference to the optimal objective value: $|(f(\boldsymbol{w}) - f(\boldsymbol{w}^*))/f(\boldsymbol{w}^*)|$, where $\boldsymbol{w}^*$ is the optimum obtained by running our algorithm long enough.

- TRON [19, 8]: the distributed trust region Newton method. We use the solver in MPI-LIBLINEAR.[5]
- VL-BFGS [4]: distributed L-BFGS. We use the techniques in Section 4.2 to reduce the cost from the original algorithm. We set $m = 10$, so P_k has 21 columns.
- L-CommDir: our method with different selections of P_k, including GRAD (2.10), STEP (2.11), and BFGS (2.12). We fix the number of columns of P_k to be 11 (current gradient plus 10 historical vectors). Specifically, for BFGS we take the past five pairs of $(\boldsymbol{u}_j, \boldsymbol{s}_j)$, and for the rest two, we take the past 10 gradients/steps.

The reason of using fewer columns in P_k of L-CommDir than that of VL-BFGS is that our method takes more memory to additionally store XP_k. We ensure a fair comparison such that both approaches consume a similar amount of memory. All methods are implemented in C/C++ and MPI with $\boldsymbol{w}_0 = \boldsymbol{0}$. Results using $C = 1$ are shown in Figure 1, while more experiments are in the supplementary material.

We observe that L-CommDir-Step and L-CommDir-BFGS are significantly faster than state-of-the-art methods on most data sets, and are competitive on the rest. The reasons for this efficiency are two-fold. First, our method converges faster and requires fewer rounds of communication. Second, our method has lower cost after parallelization (provided m is not too large) because most parts of the computation of our method are fully parallelized.

Among GRAD (2.10), STEP (2.11), or BFGS (2.12) for choosing P_k under the limited-memory setting, we see that the natural modification from the common-directions method [17] of using (2.10) may be slower than the other two choices in some data sets. Thus there is a necessity of using different information from past gradients.

7 Conclusions

In this work, we present an efficient distributed optimization algorithm that is inspired by the common-directions method, but avoids the excessive memory consumption. Theoretical results show that our method is linear convergent, and empirical comparisons indicate that our method is more efficient than state-of-the-art distributed optimization methods. Because of the high parallelizability, we expect the proposed algorithm also works well in multi-core environments.

Based on this work, we have expanded the package MPI-LIBLINEAR to include the proposed method.

Acknowledgement

This work was supported in part by MOST of Taiwan grant 104-2221-E-002-047-MY3. We thank Wei-Lin Chiang for helpful discussion and pointing out to us that the approach in Section 4.2 has already been proposed by [3].

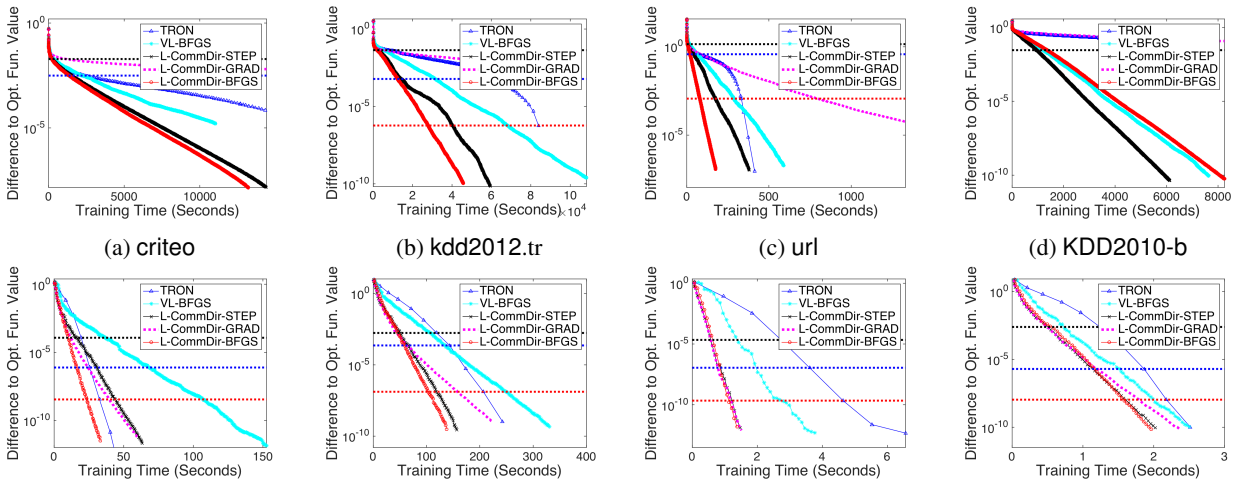
References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3:1–122, 2011.

[2] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal. On

[4] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets. For criteo, our version is slightly different because we did not instance-wise scale the feature vectors.

[5] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/distributed-liblinear/.



(a) criteo (b) kdd2012.tr (c) url (d) KDD2010-b

(e) epsilon (f) webspam (g) news20 (h) rcv1t

Figure 1: Comparison of different algorithms with $C = 1$. We show training time v.s. relative difference to the optimal function value. The horizontal lines indicate when the algorithm is terminated in practice by marking the stopping condition of TRON in MPI-LIBLINEAR: $\|\nabla f(\boldsymbol{w})\| \leq \epsilon \frac{\min(\#y_i=1, \#y_i=-1)}{l} \|\nabla f(\mathbf{0})\|$, with $\epsilon = 10^{-2}$ (default), 10^{-3}, and 10^{-4}.

the use of stochastic Hessian information in optimization methods for machine learning. *SIAM J. Optim.*, 21(3):977–995, 2011.

[3] R. H. Byrd, J. Nocedal, and R. B. Schnabel. Representations of quasi-Newton matrices and their use in limited memory methods. *Mathematical Programming*, 63(1):129–156, 1994.

[4] W. Chen, Z. Wang, and J. Zhou. Large-scale L-BFGS using MapReduce. In *NIPS*, 2014.

[5] J.-B. Hiriart-Urruty, J.-J. Strodiot, and V. H. Nguyen. Generalized Hessian matrix and second-order optimality conditions for problems with c^{L1} data. *Appl. Math. Optim.*, 11(1):43–56, 1984.

[6] H. Karimi, J. Nutini, and M. Schmidt. Linear convergence of gradient and proximal-gradient methods under Polyak-łojasiewicz condition. In *ECML/PKDD*, 2016.

[7] C.-J. Lin, R. C. Weng, and S. S. Keerthi. Trust region Newton method for large-scale logistic regression. *JMLR*, 9:627–650, 2008.

[8] C.-Y. Lin, C.-H. Tsai, C.-P. Lee, and C.-J. Lin. Large-scale logistic regression and linear support vector machines using Spark. In *IEEE BigData*, 2014.

[9] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Math. Program.*, 45(1):503–528, 1989.

[10] S. Łojasiewicz. Une propriété topologique des sous-ensembles analytiques réels. In *Les Équations aus Dérivées Partielles*. Éditions du centre National de la Recherche Scientifique, 1963.

[11] O. L. Mangasarian. A finite Newton method for classification. *Optim. Methods Softw.*, 17(5):913–929, 2002.

[12] J. Nocedal and S. Wright. *Numerical Optimization*. Springer, second edition, 2006.

[13] B. T. Polyak. Gradient methods for minimizing functionals. *Zhurnal Vychislitel'noi Matematiki i Matematicheskoi Fiziki*, 3(4):643–653, 1963.

[14] B. T. Polyak. Some methods of speeding up the convergence of iteration methods. *Comput. Math. Math. Phys.*, 4(5):1–17, 1964.

[15] T. Steihaug. The conjugate gradient method and trust regions in large scale optimization. *SIAM J. Numer. Anal.*, 20:626–637, 1983.

[16] C.-C. Wang, C.-H. Huang, and C.-J. Lin. Subsampled Hessian Newton methods for supervised learning. *Neural Comput.*, 27:1766–1795, 2015.

[17] P.-W. Wang, C.-P. Lee, and C.-J. Lin. The common directions method for regularized empirical loss minimization. Technical report, National Taiwan University, 2016.

[18] C. Zhang, H. Lee, and K. G. Shin. Efficient distributed linear classification algorithms via the alternating direction method of multipliers. In *AISTATS*, 2012.

[19] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. Distributed Newton method for regularized logistic regression. In *PAKDD*, 2015.