# 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. In particular, the communication cost is usually proportional to the number of iterations, and thus methods that converge faster are more desirable in distributed optimization.

Among nonlinear optimization methods, on one end lies methods that have low cost per iteration but with slow convergence, like gradient descent, while methods on the other end, such as Newton-type methods, have fast convergence, in the cost of expensive computation per iteration. For high-dimensional problems, one iteration of Newton-type methods actually involves many inner iterations of a sub-routine for solving a linear system to obtain the update direction, where each inner iteration involves one round of communication. Therefore, the communication cost is still high. In addition, Newton-type methods are known to have only fast local convergence, but the number of iterations may still be large. 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 methods, this method 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, since it stores and uses all previous gradients obtained from the first iteration on, the computational and spatial costs may be high as they grow unlimitedly with the number of iterations. 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 for distributed optimization, we utilize the idea of how limited-memory BFGS (L-BFGS) [9] is derived from BFGS to propose a limited-memory common-directions method framework for unconstrained smooth optimization. The proposed algorithm has limited per-iteration computational cost, and consumes a controllable amount of memory just like L-BFGS, meanwhile still possesses fast theoretical as well as empirical convergence from the common-directions method. Our method is highly parallelizable, and the computational as well as spatial burdens on each node are low. Moreover, the fast convergence inherited from the common-directions method makes our method communication-efficient. These properties our method suitable for large-scale distributed or parallel optimization. By exploiting the problem structure, the communication and computation costs per iteration of

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

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

‡Microsoft. wzchen@microsoft.com

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

our method are as low as that of L-BFGS when applied to distributedly solve regularized ERM problems, but our method has faster convergence because the Hessian information is incorporated. Our contributions are as follows.

1. We solve the issue of excessively growing memory consumption and per-iteration cost of the common-directions method by a limited-memory setting, and therefore make it more practical especially for low-end machines used in distributed optimization.
2. To exploit more different possibilities, we utilize the idea from both the heavy-ball method [14] and L-BFGS [9] to consider vectors other than the gradients, which are the only vectors used in the common-directions method. Results show our choices have better convergence in the limited-memory setting.
3. We provide convergence analysis for a general framework that includes our method. This framework fits many different algorithms and thus provides more theoretical understanding for them.
4. Empirical study shows our method pushes forward state of the art of distributed batch optimization for ERM.

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

## 2 Our Method

In the description of our algorithm, we consider the unconstrained optimization of 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 $\rho$-Lipschitz continuous. That is, there exists $\rho > 0$ such that for any $\boldsymbol{w}_1, \boldsymbol{w}_2$, we have

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

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 [11] in our algorithm. However, for the ease of description, we will always use the notation $\nabla^2 f(\boldsymbol{w})$, and this will represent either the real Hessian or the generalized Hessian, depending on the twice-differentiability of the objective function.

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 the stochastic Newton step obtained from sub-sampled Hessian matrices for ERM problems [2], the work [16] proposed to incorporate it with the update direction at the previous iteration by minimizing the second-order Taylor expansion.

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

where $\boldsymbol{d}_k$ is the stochastic Newton step at the iterate $\boldsymbol{w}_k$, and $\boldsymbol{p}_{k-1}$ is the update direction at the previous iteration. The optimum of this two-dimensional quadratic problem can easily be obtained by solving a $2 \times 2$ linear system. After obtaining the solution, $\boldsymbol{p}_k$ 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 tries to use the curvature information obtained from the gradient differences and the iterate differences to approximate the Newton step. Given an initial positive definite estimate $B_0$ of the inverse Hessian matrix, at the $k$-th iteration with 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 $\boldsymbol{p}_k$ is obtained by

$$\boldsymbol{p}_k = -B_k\nabla f(\boldsymbol{w}_k). \tag{2.2}$$

The matrix $B_k$ is constructed by

$$\begin{aligned} B_k \equiv & V_{k-1}^T\cdots V_1^T B_0 V_1\cdots V_{k-1} \\ & + \sum_{j=1}^{k-1}\rho_j V_{k-1}^T\cdots V_{j+1}^T s_j s_j^T V_{j+1}\cdots V_{k-1}, \end{aligned} \tag{2.3}$$

where

$$V_j \equiv I - \rho_j \boldsymbol{s}_j\boldsymbol{u}_j^T, \quad \rho_j \equiv \frac{1}{\boldsymbol{u}_j^T\boldsymbol{s}_j}, \quad \boldsymbol{u}_j \equiv \boldsymbol{w}_{j+1} - \boldsymbol{w}_j,$$

$$\boldsymbol{s}_j \equiv \nabla f(\boldsymbol{w}_{j+1}) - \nabla f(\boldsymbol{w}_j). \tag{2.4}$$

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 (2.3), one can use a sequence of vector operations to compute (2.2). 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] and global linear convergence [9, Theorem 6.1]. However, the major drawback of BFGS is that the computational and spatial costs of obtaining the update direction via (2.3) 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. Their idea is that given a user-specified integer $m > 0$, only $\boldsymbol{u}_j, \boldsymbol{s}_j$ for $j = k-1, \cdots, k-m$ are kept. The definition of $B_k$ then changes from (2.3) to the following.

$$\begin{aligned} B_k \equiv & V_{k-1}^T \cdots V_{k-m}^T B_0^k V_{k-m} \cdots V_{k-1} \\ & + \sum_{j=k-m}^{k-1} \rho_j V_{k-1}^T \cdots V_{j+1}^T s_j s_j^T V_{j+1} \cdots V_{k-1}, \end{aligned} \tag{2.5}$$

where $B_0^k$ are positive definite matrices that can be changed with $k$, and the most common choice is

$$B_0^k = \frac{\boldsymbol{u}_{k-1}^T \boldsymbol{s}_{k-1}}{\boldsymbol{s}_{k-1}^T \boldsymbol{s}_{k-1}} I. \tag{2.6}$$

Because earlier history is discarded, L-BFGS no longer possesses local superlinear convergence, but as shown in [9], it is still linear-convergent. Experimental results in [17] showed that L-BFGS is competitive with BFGS for linear classification in single-core environments.

**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 the recent historical information to have the following settings.

1. We follow the idea in [17, 16] to combine directions by solving a linear system involving the Hessian.
2. The method in [17] maintains all past gradients for the combination, but as mentioned earlier, in distributed environments, the excessively growing memory consumption and local computational burden of their method are not ideal. We follow the modification from BFGS to L-BFGS to maintain only the most recent history vectors.
3. To have better convergence in a limited-memory setting, we exploit different choices of historical information, including previous update steps used in the heavy-ball method [14], and the vectors used by L-BFGS.

Following L-BFGS, at the $k$-th iteration, we choose and combine $m$ vectors by solving

$$\begin{aligned} \min_{\boldsymbol{t}_k} \quad & \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k + \frac{1}{2} \boldsymbol{p}_k^T \nabla^2 f(\boldsymbol{w}_k) \boldsymbol{p}_k \\ \text{subject to} \quad & \boldsymbol{p}_k = P_k \boldsymbol{t}_k. \end{aligned} \tag{2.7}$$

Columns of $P_k$ include these $m$ vectors, which in general are obtained from the previous $m$ iterations. Therefore, in (2.7), the problem is now converted from minimizing the function in $\mathbf{R}^n$ to finding the minimizer of the second-order approximation of (2.1) in the column space of $P_k$. The most natural choice of $P_k$ is to discard older gradients in the common-directions method, resulting in

$$P_k = [\nabla f(\boldsymbol{w}_{k-m}), \nabla f(\boldsymbol{w}_{k-m+1}), \ldots, \nabla f(\boldsymbol{w}_k)]. \tag{2.8}$$

However, in a limited-memory setting, these gradient directions may not be sufficient to capture the curvature information to ensure good convergence. We therefore also consider other possible settings.

The first one we consider is the idea of the heavy-ball method [14] to use the update at the previous iteration in addition to the gradient. 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 $\alpha$ and $\beta$. We can extend this idea to use $\boldsymbol{u}_j = \boldsymbol{w}_{j+1} - \boldsymbol{w}_j$ for $j = k-1, k-2, \ldots, k-m$, so

$$P_k \equiv [\boldsymbol{u}_{k-m}, \boldsymbol{u}_{k-m+1}, \ldots, \boldsymbol{u}_{k-1}, \nabla f(\boldsymbol{w}_k)]. \tag{2.9}$$

For the second choice, [3] observed that if $B_0^k$ in (2.5) is a multiple of the identity matrix, like in the case of (2.6), the L-BFGS step $\boldsymbol{p}_k$ is a linear combination of $\boldsymbol{u}_j$, $\boldsymbol{s}_j$, $j = k-m, \ldots, k-1$, and $\nabla f(\boldsymbol{w}_k)$, while their combination coefficients are decided by the inner products between these vectors. This property suggests that we can also consider these directions to form the $P_k$ matrix.

$$P_k \equiv [\boldsymbol{u}_{k-m}, \boldsymbol{s}_{k-m}, \ldots, \boldsymbol{u}_{k-1}, \boldsymbol{s}_{k-1}, \nabla f(\boldsymbol{w}_k)]. \tag{2.10}$$

Although (2.9) and (2.10) both use some vectors generated in the same way as in L-BFGS, our algorithm is different from L-BFGS for using (2.7) to combine vectors.

We can get the solution of (2.7) by solving the following.

$$P_k^T \nabla^2 f(\boldsymbol{w}_k) P_k \boldsymbol{t} = -P_k^T \nabla f(\boldsymbol{w}_k). \tag{2.11}$$

Because the columns of $P_k$ may be linear dependent, $P_k^T \nabla^2 f(\boldsymbol{w}_k) P_k$ may be positive semi-definite rather than positive definite. Thus if we know in advance certain columns are redundant, we can remove them from (2.11) to save some computations. However, the matrix may still be singular after these directions are removed. To solve such

an underdetermined linear system, we use the pseudo inverse (denoted as $A^+$ of a matrix $A$) to obtain an optimal solution for (2.7). 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 (2.11) is $O(m^3)$, which is negligible when $m$ is small.

**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 $\theta_k$ that satisfies (2.12)
  $\boldsymbol{u}_k = \theta_k \Delta \boldsymbol{w}$
  $\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \boldsymbol{u}_k$
  Compute $\nabla f(\boldsymbol{w}_{k+1})$
  Update $P$ by some choice that includes $\nabla f(\boldsymbol{w}_{k+1})$
**end**

After solving (2.11), we take the update direction $\boldsymbol{p}_k = P_k \boldsymbol{t}_k$, 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

$$f(\boldsymbol{w}_k + \theta_k \boldsymbol{p}_k) \leq f(\boldsymbol{w}_k) + c_1 \theta_k \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k, \tag{2.12}$$

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 (2.10), and the other two choices (2.8) and (2.9) follow the same technique. For (2.10), three vectors $\boldsymbol{u}_k, \boldsymbol{s}_k$, and $\nabla f(\boldsymbol{w}_{k+1})$ are needed. After line search, we have

$$\boldsymbol{w}_{k+1} = \boldsymbol{w}_k + \theta_k P \boldsymbol{t}_k = \boldsymbol{w}_k + \boldsymbol{u}_k, \tag{2.13}$$

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 (2.4). Algorithm 1 summarizes our framework which only requires that the current gradient $\nabla f(\boldsymbol{w}_k)$ is included in $P_k$. Note that the choice of $P_k$ is not limited to (2.8)–(2.10) discussed above. 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.

In Section 6, we will empirically confirm the advantages of considering the choices of (2.9) or (2.10) over (2.8) in solving logistic regression problems. As our method uses the real Hessian information to combine these vectors more wisely, it is expected that it converges at least as good as, if not better than the heavy-ball method and L-BFGS.

## 3 Convergence Analysis

We now discuss the convergence of our method to understand the overall cost. The following theorems show the number of backtracking steps required for line search and the iteration complexity of our method. As we suggested several choices of the $P_k$ matrix in (2.8)–(2.10), we consider a more general algorithm such that each choice is a special case. Further, 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 $\rho$-Lipschitz continuous gradient with $\rho > 0$.*

Given the current iterate $\boldsymbol{w}_k$ we obtain the update direction by the following optimization problem.

$$\begin{aligned} \min_{\boldsymbol{t}} \quad & \nabla f(\boldsymbol{w}_k)^T \boldsymbol{p}_k + \frac{1}{2} \boldsymbol{p}_k^T H_k \boldsymbol{p}_k \\ \text{subject to} \quad & \boldsymbol{p}_k = \sum\nolimits_{j=1}^{s} t_j \boldsymbol{q}_j, \end{aligned} \tag{3.14}$$

for some given directions $\boldsymbol{q}_j \in \mathbf{R}^n, j = 1, \ldots, s$, together with positive definite $H_k$ such that there exists $M_1 \geq M_2 > 0$ satisfying

$$M_1 I \succeq H_k \succeq M_2 I, \forall k. \tag{3.15}$$

Now we start our analysis for the convergence. All the proofs are in the supplementary material.

THEOREM 3.1. *Consider* (3.14) *for $f(\boldsymbol{w})$ satisfying Assumption 1. If at iteration $k$, there is one $\boldsymbol{q}_j$ in* (3.14) *satisfying*

$$\frac{\left|\nabla f(\boldsymbol{w}_k)^T \boldsymbol{q}_j\right|}{\|\nabla f(\boldsymbol{w}_k)\| \|\boldsymbol{q}_j\|} \geq \delta > 0, \tag{3.16}$$

*then backtracking line search for* (2.12) *with some given $\beta, c_1 \in (0, 1)$ terminates in finite steps.*

THEOREM 3.2. *Consider minimizing $f(\boldsymbol{w})$ satisfying Assumption 1. If at each iteration, there is a $\boldsymbol{q}_j$ in* (3.14) *satisfying* (3.16) *for some constant $\delta > 0$ fixed over iterations, then the iterates generated by using the direction $\boldsymbol{p}_k$ at each iteration with backtracking line search converges to a stationary point in an $O(1/\epsilon^2)$ rate, a.k.a. $\min_{0 \leq j \leq k} \|\nabla f(\boldsymbol{w}_j)\| = O(1/\sqrt{k+1})$. If in addition $f(\boldsymbol{w})$ satisfies the Polyak-Łojasiewicz condition [10, 13, 6] for some $\sigma > 0$, i.e.,*

$$\|\nabla f(\boldsymbol{w})\|^2 \geq 2\sigma (f(\boldsymbol{w}) - f^*), \ \ \forall \boldsymbol{w}, \tag{3.17}$$

*where $f^*$ is the optimum of $f(\boldsymbol{w})$, then we have global linear convergence to $f^*$.*

Our theorems here can be applied to a general framework of combining multiple directions or even only when

one direction is used, and thus we provide convergence not only for our algorithm, but also many others. For example, the algorithm in [16] for regularized ERM, whose analysis proved (3.16) and (3.17) but only showed asymptotic convergence to the optimum, is linearly convergent by using our results here. Our result here also presents a sublinear convergence rate for their algorithm on the nonconvex neural network problem. 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.

We now apply the above results to Algorithm 1 to further examine the convergence behavior of our method.

COROLLARY 3.1. *Consider using Algorithm 1 to solve* (2.1) *satisfying Assumption 1 with $f$ being $\sigma$-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* (2.12) *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 $\epsilon$-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 not strongly convex, we cannot guarantee that the Hessian matrix used in Algorithm 1 is positive definite. In this case, we can use a modified Hessian $H_k$ instead of $\nabla^2 f(\boldsymbol{w}_k)$ satisfying $(\rho + \tau)I \succeq H_k \succeq \tau I$ for some $\tau > 0$. This way, Theorems 3.1 and 3.2 follow by letting $(M_1, M_2) = (\rho + \tau, \tau)$ in (3.15), and we still have the $O(1/\epsilon^2)$ convergence to a stationary point.

## 4 Application on Distributed Optimization for ERM

We consider applying our method to distributedly solve the L2-regularized ERM problem.

$$\min_{\boldsymbol{w} \in \mathbf{R}^n} \quad f(\boldsymbol{w}) \equiv \boldsymbol{w}^T\boldsymbol{w}/2 + C\sum_{i=1}^{l} \xi(y_i; \boldsymbol{w}^T\boldsymbol{x}_i), \tag{4.18}$$

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 $\xi$ is a convex function with respect to the second argument. We simplify our notation by $\xi_i(\boldsymbol{w}^T\boldsymbol{x}_i) \equiv \xi(y_i; \boldsymbol{w}^T\boldsymbol{x}_i)$. We consider the $\xi$ function that makes $f(\boldsymbol{w})$ in (4.18) 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 $i \in J_r$. 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 $i \in \tilde{J}_r$.[1] 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 $\boldsymbol{v}$, we denote $\boldsymbol{v}_{J_r}$ as the sub-vector of $\boldsymbol{v}$ 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 (4.18) is twice-differentiable almost everywhere, and its gradient and (generalized) Hessian are respectively

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + CX^T\boldsymbol{v}_{\boldsymbol{w}}, \tag{4.19}$$

$$\nabla^2 f(\boldsymbol{w}) \equiv I + CX^T D_{\boldsymbol{w}} X, \tag{4.20}$$

where $X^T = [\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_l]$, $D_{\boldsymbol{w}}$ is a diagonal matrix with

$$(D_{\boldsymbol{w}})_{i,i} \equiv \xi_i''(z) \mid_{z=(X\boldsymbol{w})_i}, i = 1, \ldots, l, \tag{4.21}$$

and

$$(v_{\boldsymbol{w}})_i = \xi_i'(z) \mid_{z=(X\boldsymbol{w})_i}, i = 1, \ldots, l. \tag{4.22}$$

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 (4.18). We focus the description on using (2.10) 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

$$m \ll n \quad \text{and} \quad m \ll n. \tag{4.23}$$

### 4.1 Reducing Data-related Costs.

The main computation at each iteration of Algorithm 1 is to construct and solve the linear system (2.11). For ERM, the left-hand side of (2.11) has the following structure.

$$P_k^T\nabla^2 f(\boldsymbol{w}_k)P_k = P_k^T P_k + C(XP_k)^T D_{\boldsymbol{w}_k}(XP_k). \tag{4.24}$$

We discuss 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 (4.24) can be cheaply computed in $O(m^2 l)$.

[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.

Other wise, $O(\#\text{nnz} \cdot m)$ for calculating $XP_k$ is generally more expensive. We discuss how to easily obtain $XP_k$ using $XP_{k-1}$ and some cheap operations. We begin with the case of a single machine. From (2.10), three new vectors

$$X\boldsymbol{u}_{k-1}, \quad X\boldsymbol{s}_{k-1}, \quad X\nabla f(\boldsymbol{w}_k) \tag{4.25}$$

must be calculated. From (2.4) and (2.13),

$$X\boldsymbol{s}_{k-1} = X\nabla f(\boldsymbol{w}_k) - X\nabla f(\boldsymbol{w}_{k-1}), \tag{4.26}$$

$$X\boldsymbol{u}_{k-1} = \theta_{k-1}(XP_{k-1})\boldsymbol{t}_{k-1}. \tag{4.27}$$

Because $X\nabla f(\boldsymbol{w}_{k-1})$ is maintained in $XP_{k-1}$, the main operation for (4.25) is $X\nabla f(\boldsymbol{w}_k)$ that costs $O(\#\text{nnz})$. For maintaining $X\boldsymbol{w}_k$, we have

$$\begin{aligned} X\boldsymbol{w}_k &= X\boldsymbol{w}_{k-1} + \theta_{k-1}XP_{k-1}\boldsymbol{t}_{k-1} \\ &= X\boldsymbol{w}_{k-1} + X\boldsymbol{u}_{k-2}. \end{aligned} \tag{4.28}$$

Because $X\boldsymbol{u}_{k-2}$ is part of $XP_{k-1}$, which has been maintained, (4.28) costs only $O(l)$. In summary, calculating the second term in (4.24) needs $O(m^2l + \#\text{nnz})$.

In our distributed setting, (4.26)-(4.28) become

$$\begin{aligned} (X\boldsymbol{s}_{k-1})_{J_r} &= (X\nabla f(\boldsymbol{w}_k))_{J_r} - (X\nabla f(\boldsymbol{w}_{k-1}))_{J_r}, \\ (X\boldsymbol{u}_{k-1})_{J_r} &= \theta_{k-1}(XP_{k-1})_{J_r,:}\boldsymbol{t}_{k-1}, \\ (X\boldsymbol{w}_k)_{J_r} &= (X\boldsymbol{w}_{k-1})_{J_r} + (X\boldsymbol{u}_{k-2})_{J_r}. \end{aligned}$$

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 have been available, all we need is $\nabla f(\boldsymbol{w}_k)$, which is computed and made available to all nodes through an *allreduce* operation.

$$\nabla f(\boldsymbol{w}_k) = \bigoplus_{r=1}^{K} \left( CX_{J_r,:}^T(\boldsymbol{v}_{\boldsymbol{w}_k})_{J_r} + \begin{bmatrix} \mathbf{0} \\ (\boldsymbol{w}_k)_{\tilde{J}_r} \\ \mathbf{0} \end{bmatrix} \right), \tag{4.29}$$

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 (4.24) is fully parallelized and thus costs $O((\#\text{nnz} + m^2l)/K)$ computation for each machine, with one round of $O(n)$ communication to generate the gradient.

In (4.29), $(\boldsymbol{v}_{\boldsymbol{w}_k})_{J_r}$ is needed. 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 an *allreduce* operation on local matrices of size $O(m^2)$ is conducted to get the second term in (4.24).

$$\bigoplus_{r=1}^{K} (XP_k)_{J_r,:}^T (D_{\boldsymbol{w}_k})_{J_r,J_r} (XP_k)_{J_r,:}. \tag{4.30}$$

Similarly, in the line search procedure, even if it takes several backtracking steps, we can evaluate the loss function by

$$\bigoplus_{r=1}^{K} (\sum\nolimits_{i \in J_r} \xi_i((X\boldsymbol{w}_k)_i + \theta(XP_k)_{i,:}\boldsymbol{t}_k)). \tag{4.31}$$

With the availability of $(X\boldsymbol{w}_k)_{J_r}$ and $(XP_k)_{J_r,:}$, (4.31) is cheap with $O(1)$ communication cost for *allreduce*.

Once the matrix in (4.24) is generated, the remaining task is to solve a linear system of $m$ variables. Because $m$ is small from (4.23), it may not be cost-effective to solve (2.11) distributedly. We thus let the matrix and the right-hand side vector $-P_k^T\nabla f(\boldsymbol{w}_k)$ be available at all computing nodes, and each node takes $O(m^3)$ to solve the same linear system.

**Algorithm 2:** Distributed Limited-memory Common-directions Method for solving the ERM problem (4.18)

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 (4.22) and (4.21)
Compute $\nabla f(\boldsymbol{w}_0)$ by (4.29)
$P_{\tilde{J}_r,:} = [(\nabla f(\boldsymbol{w}_0))_{\tilde{J}_r}], M^0 = \bigoplus_{r=1}^{K} \|\nabla f(\boldsymbol{w}_0)_{\tilde{J}_r}\|^2$
$(\hat{\boldsymbol{u}}_0)_{J_r} = X_{J_r,:}\nabla f(\boldsymbol{w}_0), U_{J_r} = (\hat{\boldsymbol{u}}_0)_{J_r}$
**for** *k=0,1,…* **do**
  Compute $U^T D_{\boldsymbol{w}_k} U$ by (4.30) ▷ $O(m^2l/K)$; $O(m^2)$ communication
  Obtain $\boldsymbol{t}$ by solving (2.11) with $\boldsymbol{t} = [M^k + CU^T D_{\boldsymbol{w}_k} U]^+[-P^T\nabla f(\boldsymbol{w}_k)]$ ▷ $O(m^3)$
  Compute $\Delta\boldsymbol{w}_{\tilde{J}_r} = P_{\tilde{J}_r,:}\boldsymbol{t}$ ▷ $O(mn/K)$
  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 $\theta$ that satisfied (2.12) using $\boldsymbol{z}_{J_r}$ and $\Delta\boldsymbol{z}_{J_r}$ for (4.31)
  $(\boldsymbol{u}_k)_{\tilde{J}_r} = \theta\Delta\boldsymbol{w}_{\tilde{J}_r}$
  $(X\boldsymbol{u}_k)_{J_r} = \theta\Delta\boldsymbol{z}_{J_r}$
  $(\boldsymbol{w}_{k+1})_{\tilde{J}_r} = (\boldsymbol{w}_k)_{\tilde{J}_r} + (\boldsymbol{u}_k)_{\tilde{J}_r}$
  $\boldsymbol{z}_{J_r} = \boldsymbol{z}_{J_r} + (X\boldsymbol{u}_k)_{J_r}$
  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 (4.29) ▷ $O(\#\text{nnz}/K)$; $O(n)$ 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 (2.10)
  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 $P^T\nabla f(\boldsymbol{w}_{k+1})$ and $\boldsymbol{s}_k^T\boldsymbol{s}_k$ by (4.34) and (4.35) ▷ $O(mn/K)$; $O(m)$ communication
  Update $M^{k+1}$ using (4.33) ▷ $O(m^2)$
**end**

**4.2 Reducing Cost for $P_k^T P_k$.** Let $M^k \equiv P_k^T P_k$. We mentioned in Section 2.3 that in [3] for distributed L-BFGS, their method, called VL-BFGS, requires calculating $M^k$ (i.e., inner products between $P_k$'s columns) as well. They pointed out that $M^{k-1}$ and $M^k$ share most elements, so only

the following new entries must be calculated.

$$P_k^T[\boldsymbol{u}_{k-1}, \boldsymbol{s}_{k-1}, \nabla f(\boldsymbol{w}_k)]. \tag{4.32}$$

With $P_k \in \mathbf{R}^{n\times(2m+1)}$, clearly $6m+6$ inner products are needed for (4.32). Besides, their method also involves $2m$ vector additions (details omitted). The cost is therefore higher than $2m$ inner products and $2m$ vector additions in the standard L-BFGS.[2] However, the advantage of [3]'s setting is that the $6m+6$ inner products can be fully parallelized when $P_k$ is distributedly stored. In this section, we will show that by a careful design, (4.32), or $M^k$, can be done by $2m+2$ parallelizable inner products: $P_k^T\nabla f(\boldsymbol{w}_k)$ and $\boldsymbol{s}_{k-1}^T\boldsymbol{s}_{k-1}$. Therefore, our technique is useful not only for our method but also for improving upon VL-BFGS.

To efficiently calculate (4.32), we begin with considering $P_k^T\boldsymbol{s}_{k-1}$. From (2.4) 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

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

Besides $\boldsymbol{s}_{k-1}^T\boldsymbol{s}_{k-1}$, other operations cost no more than $O(m)$, which is negligible from (4.23).

Next, we consider $P_k^T\boldsymbol{u}_{k-1}$. From (2.13),

$$\boldsymbol{v}^T\boldsymbol{u}_{k-1} = \theta_{k-1}\boldsymbol{v}^T P_{k-1}\boldsymbol{t}_{k-1}, \forall \boldsymbol{v} \in \mathbf{R}^n.$$

This then leads to

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

The computation of $P_k^T\boldsymbol{u}_{k-1}$ thus mainly requires computing $M^{k-1}\boldsymbol{t}_{k-1}$, which costs only $O(m^2)$.

The rest terms are available from $M^{k-1}$ because they are inner products between vectors in $P_{k-1}$. In summary,

$$M^k_{i,j} = \begin{cases} M^{k-1}_{i+2,j+2}, & \text{if } i,j < 2m-1 \\ \left(P_k^T\boldsymbol{u}_{k-1}\right)_i, & \text{if } j = 2m-1 \\ \left(P_k^T\boldsymbol{s}_{k-1}\right)_i, & \text{if } j = 2m \\ \left(P_k^T\nabla f\left(\boldsymbol{w}_k\right)\right)_i, & \text{if } j = 2m+1 \\ M^k_{j,i}, & \text{if } i \geq 2m-1 > j \end{cases}. \tag{4.33}$$

In our distributed setting, $P_k^T\nabla f(\boldsymbol{w}_k)$ and $\boldsymbol{s}_{k-1}^T\boldsymbol{s}_{k-1}$ are computed by the following *allreduce* operations.

$$P_k^T\nabla f\left(\boldsymbol{w}_k\right) = \bigoplus\nolimits_{r=1}^K \left(P_k\right)^T_{\tilde{J}_r,:}\nabla f\left(\boldsymbol{w}_k\right)_{\tilde{J}_r}, \tag{4.34}$$

$$\boldsymbol{s}_{k-1}^T\boldsymbol{s}_{k-1} = \bigoplus\nolimits_{r=1}^K \left(\boldsymbol{s}_{k-1}\right)^T_{\tilde{J}_r}\left(\boldsymbol{s}_{k-1}\right)_{\tilde{J}_r}, \tag{4.35}$$

with $O(mn/K)$ cost and $O(m)$ communication. Because $M^k$ and $\boldsymbol{t}_k$ are stored on all nodes, all other operations are conducted on all nodes without communication.

For the $2m$ vector additions to obtain $\boldsymbol{p}_k$, as we use $XP_k$ for line search, forming the full $\boldsymbol{p}_k$ is not needed. We only locally conduct vector additions on $(P_k)_{J_r,:}$ to get $(\boldsymbol{p}_k)_{\tilde{J}_r}$.

With the availability of $M^k$, some other operations can also be saved. For example, $\nabla f(\boldsymbol{w}_k)^T\boldsymbol{p}_k$ needed in (2.12) 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.

**4.3 Cost Analysis.** We list details of applying Algorithm 1 to (4.18) in a distributed environment with instance-wise data split in Algorithm 2. The computational cost per iteration at each machine is

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

where from (4.23), #nnz is in general the dominant term. For the communication, the cost 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 (2.12) is often satisfied at $\theta_k = 1$.

## 5 Related Works

Because of the need of distributed optimization in the machine learning area, there have been many approaches proposed, among which some are for general optimization problems and others are specially designed for ERM problems. We will only focus on those whose convergence are independent of the number of machines being used, as we consider the scenario of using a large number of machines.

Besides L-BFGS, another effective batch distributed optimization method is the truncated Newton method. At each iteration, the update direction $\boldsymbol{p}_k$ is obtained by approximately solving the following linear system.

$$\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{p}_k = -\nabla f(\boldsymbol{w}_k). \tag{5.36}$$

[2] In the two-loop procedure of L-BFGS, both loops have $m$ iterations, each involves one inner product and one vector addition.

| Data set | #instances | #features | #nonzeros |
|---|---|---|---|
| criteo | 45,840,617 | 999,999 | 1,787,773,969 |
| kdd2012 | 14,664,569 | 19,144,857 | 161,310,259 |
| url | 2,396,130 | 3.221,961 | 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 |

Table 1: Data statistics.

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 (5.36), a Hessian-free approach is considered by using the conjugate gradient (CG) method, where a sequence of Hessian-vector products is needed. Take the ERM problem (4.18) as an example, from (4.20),

$$\nabla^2 f(\boldsymbol{w}_k)\boldsymbol{d} = \boldsymbol{d} + X^T\left(D_{\boldsymbol{w}_k}\left(X\boldsymbol{d}\right)\right), \tag{5.37}$$

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 (5.37) requires the communication. For example, if an instance-wise split is used, we need

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

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 ADMM [18, 1] when both are implemented in MPI.

## 6 Experiments

We present results on solving (4.18) distributedly. We consider logistic regression, whose loss term $\xi_i(\cdot)$ in (4.18) is $\xi_i(z) \equiv \log\left(1+\exp\left(-y_i z\right)\right)$, with $y_i \in \{-1, 1\}$. The distributed environment is a cluster with 16 nodes, where each node runs MVAPICH2 [4] and has two Intel Xeon X5650 2.66GHZ 6C Processors. We use one core per node. We consider data sets shown in Table 1. All data sets except kdd2012 and criteo are publicly available, while these two are obtained using the parsing script in the experiment code of [5]. We compare the relative difference to the optimal objective value: $\left|(f(\boldsymbol{w}) - f(\boldsymbol{w}^*))/f(\boldsymbol{w}^*)\right|$, where $\boldsymbol{w}^*$ is the optimum obtained by running our algorithm long enough.

We compare the following state-of-the-art batch distributed optimization methods for (4.18).

- TRON [19, 8]: the distributed trust region Newton method. We use the solver in MPI-LIBLINEAR.[3]
- VL-BFGS: vector-free distributed L-BFGS. We use the techniques discussed in Section 4.2 to reduce the cost from the original algorithm by [3]. We set $m = 10$.
- L-CommDir: our method with different selections of $P_k$, including GRAD (2.8), STEP (2.9), and BFGS (2.10). We fix the number of columns of $P_k$ to be 11 (current gradient plus 10 historical vectors). Specifically, for BFGS we take the last five pairs of $(\boldsymbol{u}_j, \boldsymbol{s}_j)$, and for the rest two, we take the last 10 gradients/steps.

The reason of using fewer columns in $P_k$ of L-CommDir than that of VL-BFGS is that our method needs to additionally store $XP_k$ and thus takes more memory. We ensure a fair comparison such that both methods consume a similar amount of memory. All algorithms are implemented in C/C++ and MPI. We set the initial point $\boldsymbol{w}_0 = \boldsymbol{0}$ in all experiments. 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 the choices of using GRAD (2.8), STEP (2.9), or BFGS (2.10) for $P_k$ under the limited-memory setting, we can see that the natural modification from the common-directions method [17] of using (2.8) is significantly slower than the other two choices, showing that there is a necessity of using different information.

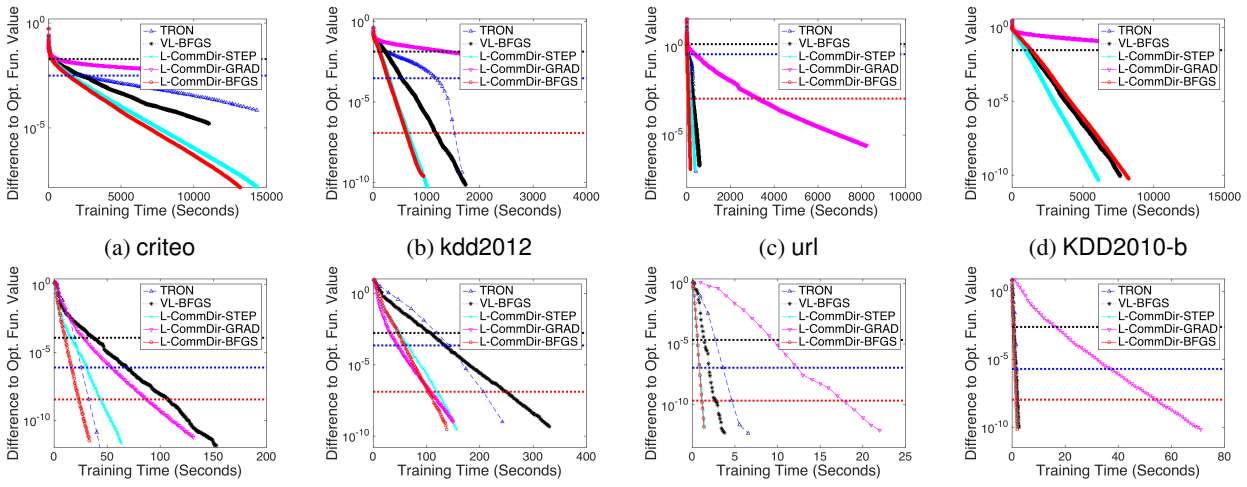
## 7 Conclusions

In this work, we present an efficient distributed optimization algorithm that is inspired by the common-directions method, and overcome its shortage of excessive memory consumption and unlimitedly growing per iteration cost. 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.

## 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.

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




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

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

Figure 1: Comparison of different algorithms with $C = 1$. We present training time v.s. relative difference to the optimal objective value. The horizontal lines mark 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 $10^{-4}$.


[2] R. H. Byrd, G. M. Chin, W. Neveitt, and J. Nocedal. On the use of stochastic Hessian information in optimization methods for machine learning. *SIAM Journal on Optimization*, 21(3):977–995, 2011.

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

[4] W. Huang, G. Santhanaraman, H.-W. Jin, Q. Gao, and D. K. Panda. Design of high performance MVAPICH2: MPI2 over infiniband. In *CCGrid*. IEEE, 2006.

[5] Y.-C. Juan, Y. Zhuang, W.-S. Chin, and C.-J. Lin. Field-aware factorization machines for CTR prediction. In *RecSys*, 2016.

[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. *Optimization Methods and Software*, 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. *USSR Computational Mathematics and Mathematical Physics*, 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 Computation*, 27:1766–1795, 2015.

[17] P.-W. Wang, C.-P. Lee, and C.-J. Lin. The common directions method for regularized 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.