Journal of Machine Learning Research 19 (2018) 1-37 Submitted 1/15; Revised 4/16; Published 12/18

An efficient distributed learning algorithm based on effective local functional approximations

Dhruv Mahajan DHRUVM@FB.COM
Facebook AI
Menlo Park, CA 94025, USA

Nikunj Agrawal NIKUNJ157@GMAIL.COM
Indian Institute of Technology
Dept. of Computer Science & Engineering
Kanpur, India

S. Sathiya Keerthi KEERTHI@MICROSOFT.COM
Office Data Science Group
Microsoft
Mountain View, CA 94043, USA

Sundararajan Sellamanickam SSRAJAN@MICROSOFT.COM
Microsoft Research
Bangalore, India

Léon Bottou LEON@BOTTOU.ORG
Facebook AI Research
New York, NY, USA

Editor: Inderjit Dhillon

Abstract

Scalable machine learning over big data is an important problem that is receiving a lot of attention in recent years. On popular distributed environments such as Hadoop running on a cluster of commodity machines, communication costs are substantial and algorithms need to be designed suitably considering those costs. In this paper we give a novel approach to the distributed training of linear classifiers (involving smooth losses and L_2 regularization) that is designed to reduce the total communication costs. At each iteration, the nodes minimize locally formed approximate objective functions; then the resulting minimizers are combined to form a descent direction to move. Our approach gives a lot of freedom in the formation of the approximate objective function as well as in the choice of methods to solve them. The method is shown to have $O(\log(1/\epsilon))$ time convergence. The method can be viewed as an iterative parameter mixing method. A special instantiation yields a parallel stochastic gradient descent method with strong convergence. When communication times between nodes are large, our method is much faster than the Terascale method (Agarwal et al., 2011), which is a state of the art distributed solver based on the statistical query model (Chu et al., 2006) that computes function and gradient values in a distributed fashion. We also evaluate against other recent distributed methods and demonstrate superior performance of our method.

Keywords: Distributed learning, Example partitioning, L_2 regularization

©2018 Dhruv Mahajan, Nikunj Agrawal, S. Sathiya Keerthi, Sundararajan Sellamanickam and Léon Bottou.

License: CC-BY 4.0, see https://creativecommons.org/licenses/by/4.0/. Attribution requirements are provided at http://jmlr.org/papers/v19/15-020.html.

1. Introduction

In recent years, machine learning over Big Data has become an important problem, not only in web related applications, but also more commonly in other applications, e.g., in the data mining over huge amounts of user logs. The data in such applications are usually collected and stored in a decentralized fashion over a cluster of commodity machines (nodes) where communication times between nodes are significantly large. Examples of applications involving high communication costs include: Click prediction on advertisement data where the number of examples is huge and features are words which can run in to billions, and Geo distributed data across different countries/continents where cross data-center speeds are extremely slow. In such a settings, it is natural for the examples to be partitioned over the nodes.

Distributed machine learning algorithms that operate on such data are usually iterative. Each iteration involves some *computation* that happens locally in each node. In each iteration, there is also *communication* of information between nodes and this is special to the distributed nature of solution. Distributed systems such as those based on the Map-Reduce framework (Dean and Ghemawat, 2008) involve additional special operations per iteration, such as the loading of data from disk to RAM. Recent frameworks such as Spark (Zaharia et al., 2010) and REEF (Weimer et al., 2015) avoid such unnecessary repeated loading of data from disk. Still, communication between nodes in each iteration is unavoidable and, its cost can be substantial when working with Big Data. Therefore, the development of efficient distributed machine learning algorithms that minimize communication between nodes is an important problem. The key is to come up with algorithms that minimize the number of iterations.

In this paper we consider the distributed batch training of linear classifiers in which: (a) both, the number of examples and the number of features are large; (b) the data matrix is sparse; (c) the examples are partitioned over the nodes; (d) the loss function is convex and differentiable; and, (e) the L_2 regularizer is employed. This problem involves the large scale unconstrained minimization of a convex, differentiable objective function $f(w)$ where w is the weight vector. The minimization is usually performed using an iterative descent method in which an iteration starts from a point w^r, computes a direction d^r that satisfies

$$\textbf{sufficient angle of descent: } \underline{\angle -g^r, d^r} \leq \theta \tag{1}$$

where $g^r = g(w^r)$, $g(w) = \nabla f(w)$, $\underline{\angle a, b}$ is the angle between vectors a and b, and $0 \leq \theta < \pi/2$, and then performs a line search along the direction d^r to find the next point, $w^{r+1} = w^r + td^r$. Let $w^\star = \arg\min_w f(w)$. A key side contribution of this paper is the proof that, when f is convex and satisfies some additional weak assumptions, the method has global linear rate of convergence (*glrc*)[1] and so it finds a point w^r satisfying $f(w^r) - f(w^\star) \leq \epsilon$ in $O(log(1/\epsilon))$ iterations. *The main theme of this paper is that the flexibility offered by this method with strong convergence properties allows us to build a class of useful distributed learning methods with good computation and communication trade-off capabilities.*

Take one of the most effective distributed methods, viz., SQM (Statistical Query Model) (Chu et al., 2006; Agarwal et al., 2011), which is a batch, gradient-based descent method. The gradient is computed in a distributed way with each node computing the gradient

1. We say a method has *glrc* if $\exists\ 0 < \delta < 1$ such that $(f(w^{r+1}) - f(w^\star)) \leq \delta(f(w^r) - f(w^\star))\ \forall r$.

component corresponding to its set of examples. This is followed by an aggregation of the components. We are interested in systems in which the communication time between nodes is large relative to the computation time in each node.[2] For iterative algorithms such as SQM, the total training time is given by

$$\text{Training time} = (T^{cmp} + T^{com})\, T^{iter} \tag{2}$$

where T^{cmp} and T^{com} are respectively, the computation time and the communication time per iteration and T^{iter} is the total number of iterations. When T^{com} is large, it is not optimal to work with an algorithm such as SQM that has T^{cmp} small and due to which, T^{iter} is large. In such a scenario, it is useful to ask: **Q1.** *In each iteration, can we do more computation in each node so that the number of iterations and hence the number of communication passes are decreased, thus reducing the total computing time?*

There have been some efforts in the literature to reduce the amount of communication. In one class of such methods, the current w^r is first passed on to all the nodes. Then, each node p forms an approximation $\tilde{f}_p$ of f using only its examples, followed by several optimization iterations (local passes over its examples) to decrease $\tilde{f}_p$ and reach a point w_p. The $w_p\ \forall p$ are averaged to form the next iterate w^{r+1}. One can stop after just one major iteration (going from $r = 0$ to $r = 1$); such a method is referred to as *parameter mixing (PM)* (Mann et al., 2009). Alternatively, one can do many major iterations; such a method is referred to as *iterative parameter mixing (IPM)* (Hall et al., 2010). Convergence theory for such methods is inadequate (Mann et al., 2009; McDonald et al., 2010), which prompts us to ask: **Q2.** *Is it possible to devise an IPM method that produces* $\{w^r\} \to w^\star$*?*

In another class of methods, the dual problem is solved in a distributed fashion (Pechyony et al., 2011; Yang, 2013; Yang et al., 2013; Jaggi et al., 2014). Let α_p denote the dual vector associated with the examples in node p. The basic idea is to optimize $\{\alpha_p\}$ in parallel and then use a combination of the individual directions thus generated to take an overall step. In practice these methods tend to have slow convergence; see Section 4 for details.

We make a novel and simple use of the iterative descent method mentioned at the beginning of this section to design a distributed algorithm that answers Q1-Q2 positively. The main idea is to use distributed computation for generating a good search direction d^r and not just for forming the gradient as in SQM. At iteration r, let us say each node p has the current iterate w^r and the gradient g^r. This information can be used together with the examples in the node to form a function $\hat{f}_p(\cdot)$ that approximates $f(\cdot)$ and satisfies $\nabla \hat{f}_p(w^r) = g^r$. One simple and effective suggestion is:

$$\hat{f}_p(w) = f_p(w) + (g^r - \nabla f_p(w^r)) \cdot (w - w^r) \tag{3}$$

where f_p is the part of f that does not depend on examples outside node p; the second term in (3) can be viewed as an approximation of the objective function part associated with data from the other nodes. In Section 3 we give other suggestions for forming $\hat{f}_p$. Now $\hat{f}_p$ can be optimized within node p using any method $\mathcal{M}$ which has *glrc*, e.g., Trust region method, L-BFGS, etc. There is no need to optimize $\hat{f}_p$ fully. We show (see Section 3) that, in a constant number of local passes over examples in node p, an approximate minimizer

2. This is the case when feature dimension is huge. Many applications gain performance when the feature space is expanded, say, via feature combinations, explicit expansion of nonlinear kernels etc.

w_p of $\hat{f}_p$ can be found such that the direction $d_p = w_p - w^r$ satisfies the sufficient angle of descent condition, (1). A convex combination of the set of directions generated in the nodes, $\{d_p\}$ forms the overall direction d^r for iteration r. Note that d^r also satisfies (1). The result is an overall distributed method that finds a point w satisfying $f(w) - f(w^\star) \leq \epsilon$ in $O(\log(1/\epsilon))$ time. This answers **Q2**.

The method also reduces the number of communication passes over the examples compared with SQM, thus also answering **Q1**. The intuition here is that, if each $\hat{f}_p$ is a good approximation of f, then d^r will be a good global direction for minimizing f at w^r, and so the method will move towards $w^\star$ much faster than SQM.

In summary, the paper makes the following contributions. First, for convex f we establish *glrc* for a general iterative descent method. Second, and more important, we propose a distributed learning algorithm that: (a) converges in $O(\log(1/\epsilon))$ time, thus leading to an IPM method with strong convergence; (b) is more efficient than SQM when communication costs are high; and (c) flexible in terms of the local optimization method $\mathcal{M}$ that can be used in the nodes.

There is also another interesting side contribution associated with our method. It is known that example-wise methods such as stochastic gradient descent (SGD) are inherently sequential and hard to parallelize (Zinkevich et al., 2010). By employing SGD as $\mathcal{M}$, the local optimizer for $\hat{f}_p$ in our method, we obtain a parallel SGD method with good performance as well as strong convergence properties. This contribution is covered in detail in Mahajan et al. (2013b); we give a summarized view in Subsection 3.5.

Experiments (Section 4) validate our theory as well as show the benefits of our method for large dimensional datasets where communication is the bottleneck. We give a discussion on unexplored possibilities for extending our distributed learning method in Section 5 and conclude the paper in Section 6.

2. Basic descent method

Let $f \in \mathcal{C}^1$, the class of continuously differentiable functions[3], f be convex, and the gradient g satisfy the following assumptions.

A1. g is Lipschitz continuous, i.e., $\exists\ L > 0$ such that $\|g(w) - g(\tilde{w})\| \leq L\|w - \tilde{w}\| \quad \forall\ w, \tilde{w}$.

A2. $\exists\ \sigma > 0$ such that $(g(w) - g(\tilde{w})) \cdot (w - \tilde{w}) \geq \sigma\|w - \tilde{w}\|^2 \quad \forall\ w, \tilde{w}$.

A1 and A2 are essentially second order conditions: if f happens to be twice continuously differentiable, then L and σ can be viewed as upper and lower bounds on the eigenvalues of the Hessian of f. A convex function f is said to be σ- strongly convex if $f(w) - \frac{\sigma}{2}\|w\|^2$ is convex. In machine learning, all convex risk functionals in $\mathcal{C}^1$ having the L_2 regularization term, $\frac{\lambda}{2}\|w\|^2$ are σ- strongly convex with $\sigma = \lambda$. It can be shown (Smola and Vishwanathan, 2008) that, if f is σ-strongly convex, then f satisfies assumption A2.

Let $f^r = f(w^r)$, $g^r = g(w^r)$ and $w^{r+1} = w^r + td^r$. Consider the following standard line search conditions.

$$\textbf{Armijo:} \quad f^{r+1} \leq f^r + \alpha g^r \cdot (w^{r+1} - w^r) \tag{4}$$

$$\textbf{Wolfe:} \quad g^{r+1} \cdot d^r \geq \beta g^r \cdot d^r \tag{5}$$

3. It would be interesting future work to extend all the theory developed in this paper to non-differentiable convex functions, using sub-gradients.

where $0 < \alpha < \beta < 1$.

Algorithm 1: Descent method for f

Choose w^0;
for $r = 0, 1 \ldots$ **do**
1. Exit if $g^r = 0$;
2. Choose a direction d^r satisfying (1);
3. Do line search to choose $t > 0$ so that $w^{r+1} = w^r + td^r$ satisfies the Armijo-Wolfe conditions (4) and (5);

end

Let us now consider the general descent method in Algorithm 1 for minimizing f. The following result shows that the algorithm is well-posed. A proof is given in the appendix B.
Lemma 1. Suppose $g^r \cdot d^r < 0$. Then $\{t : (4) \text{ and } (5) \text{ hold for } w^{r+1} = w^r + td^r\} = [t_\beta, t_\alpha]$, where $0 < t_\beta < t_\alpha$, and t_β, t_α are the unique roots of

$$g(w^r + t_\beta d^r) \cdot d^r = \beta g^r \cdot d^r, \tag{6}$$

$$f(w^r + t_\alpha d^r) = f^r + t_\alpha \alpha g^r \cdot d^r, \;\; t_\alpha > 0. \tag{7}$$

Theorem 2. Let $w^\star = \arg\min_w f(w)$ and $f^\star = f(w^\star)$.[4] Then $\{w^r\} \to w^\star$. Also, we have *glrc*, i.e., $\exists\ \delta$ satisfying $0 < \delta < 1$ such that $(f^{r+1} - f^\star) \leq \delta\,(f^r - f^\star)\ \forall\ r \geq 0$, and, $f^r - f^\star \leq \epsilon$ is reached after at most $\frac{\log((f^0 - f^\star)/\epsilon)}{\log(1/\delta)}$ iterations. An upper bound on δ is $(1 - 2\alpha(1-\beta)\frac{\sigma^2}{L^2}\cos^2\theta)$.

A proof of Theorem 2 is given in the appendix B. If one is interested only in proving convergence, it is easy to establish under the assumptions made; such theory goes back to the classical works of Wolfe (Wolfe, 1969, 1971). But proving *glrc* is harder. There exist proofs for special cases such as the gradient descent method (Boyd and Vandenberghe, 2004). The *glrc* result in Wang and Lin (2013) is only applicable to descent methods that are "close" (see equations (7) and (8) in Wang and Lin (2013)) to the gradient descent method. Though Theorem 2 is not entirely surprising, as far as we know, such a result does not exist in the literature.

Remark 1. It is important to note that the rate of convergence indicated by the upper bound on δ given in Theorem 2 is pessimistic since it is given for a very general descent algorithm that includes plain batch gradient descent which is known to have a slow rate of convergence. *Therefore, it should not be misconstrued that the indicated slow convergence rate would hold for all methods falling into the class of methods covered here.* Depending on the method used for choosing d^r, the actual rate of convergence can be a lot better. For example, we observe very good rates for our distributed method, FADL; see the empirical results in Section 4 and also the comments made in Remark 2 of Section 3.

3. Distributed training

In this section we discuss full details of our distributed training algorithm. Let $\{x_i, y_i\}$ be the training set associated with a binary classification problem ($y_i \in \{1, -1\}$). Consider a

4. Assumption A2 implies that $w^\star$ is unique.

linear classification model, $y = \text{sgn}(w^T x)$. Let $l(w \cdot x_i, y_i)$ be a continuously differentiable loss function that has Lipschitz continuous gradient. This allows us to consider loss functions such as least squares ($l(s, y) = (s - y)^2$), logistic loss ($l(s, y) = \log(1 + \exp(-sy))$ and squared hinge loss ($l(s, y) = \max\{0, 1 - ys\}^2$). Hinge loss is not covered by our theory since it is non-differentiable.

Suppose the training examples are distributed in P nodes. Let: I_p be the set of indices i such that (x_i, y_i) sits in the p-th node; $L_p(w) = \sum_{i \in I_p} l(w; x_i, y_i)$ be the total loss associated with node p; and, $L(w) = \sum_p L_p(w)$ be the total loss over all nodes. Our aim is to minimize the regularized risk functional $f(w)$ given by

$$f(w) = \frac{\lambda}{2}\|w\|^2 + L(w) = \frac{\lambda}{2}\|w\|^2 + \sum_p L_p(w), \tag{8}$$

where $\lambda > 0$ is the regularization constant. It is easy to check that $g = \nabla f$ is Lipschitz continuous.

3.1. Our approach

Our distributed method is based on the descent method in Algorithm 1. We use a master-slave architecture.[5] Let the examples be partitioned over P slave nodes. Distributed computing is used to compute the gradient g^r as well as the direction d^r. In the r-th iteration, let us say that the master has the current w^r and gradient g^r. One can communicate these to all P (slave) nodes. The direction d^r is formed as follows. Each node p constructs an approximation of $f(w)$ using only information that is available in that node, call it $\hat{f}_p(w)$, and (approximately) optimizes it (starting from w^r) to get the point w_p. Let $d_p = w_p - w^r$. Then d^r is chosen to be any convex combination of $d_p \; \forall p$. Doing line search along the d^r direction completes the r-th iteration. Line search involves distributed computation, but it is inexpensive; we give details in Subsection 3.4.

We want to point out that $\hat{f}_p$ can change with r, i.e., one is allowed to use a different $\hat{f}_p$ in each outer iteration. We just don't mention it as $\hat{f}_p^r$ to avoid clumsiness of notation. In fact, all the choices for $\hat{f}_p$ that we discuss below in Subsection 3.2 are such that $\hat{f}_p$ depends on the current iterate, w^r.

3.2. Choosing $\hat{f}_p$

Our method offers great flexibility in choosing $\hat{f}_p$ and the method used to optimize it. We only require $\hat{f}_p$ to satisfy the following.

A3. $\hat{f}_p$ is σ-strongly convex, has Lipschitz continuous gradient and satisfies *gradient consistency at* w^r: $\nabla \hat{f}_p(w^r) = g^r$.

Below we give several ways of forming $\hat{f}_p$. The σ-strongly convex condition is easily taken care of by making sure that the L_2 regularizer is always a part of $\hat{f}_p$. This condition implies that

$$\hat{f}_p(w_p) \geq \hat{f}_p(w^r) + \nabla \hat{f}_p(w^r) \cdot (w_p - w^r) + \frac{\sigma}{2}\|w_p - w^r\|^2. \tag{9}$$

5. An *AllReduce* arrangement of nodes (Agarwal et al., 2011) may also be used.

The gradient consistency condition is motivated by the need to satisfy the angle condition (1). Since w_p is obtained by starting from w^r and optimizing $\hat{f}_p$, it is reasonable to assume that $\hat{f}_p(w_p) < \hat{f}_p(w^r)$. Using these in (9) gives $-g^r \cdot d_p > 0$. Since d^r is a convex combination of the d_p it follows that $-g^r \cdot d^r > 0$. Later we will formalize this to yield (1) precisely.

A general way of choosing the approximating functional $\hat{f}_p$ is

$$\hat{f}_p(w) = \frac{\lambda}{2}\|w\|^2 + \tilde{L}_p(w) + \hat{L}_p(w), \tag{10}$$

where $\tilde{L}_p$ is an approximation of L_p and $\hat{L}_p(w)$ is an approximation of $L(w) - L_p(w) = \sum_{q \neq p} L_q(w)$. A natural choice for $\tilde{L}_p$ is L_p itself since it uses only the examples within node p; but there are other possibilities too. To maintain communication efficiency, we would like to design an $\hat{L}_p$ such that it does not explicitly require any examples outside node p. To satisfy A3 we need $\hat{L}_p$ to have Lipschitz continuous gradient. Also, to aid in satisfying gradient consistency, appropriate linear terms are added. We now suggest five choices for $\hat{f}_p$.

Linear Approximation. Set $\tilde{L}_p = L_p$ and choose $\hat{L}_p$ based on the first order Taylor series. Thus,

$$\tilde{L}_p(w) = L_p(w), \qquad \hat{L}_p(w) = (\nabla L(w^r) - \nabla L_p(w^r)) \cdot (w - w^r). \tag{11}$$

(The zeroth order term needed to get $f(w^r) = \hat{f}(w^r)$ is omitted everywhere because it is a constant that plays no role in the optimization.) Note that $\nabla L(w^r) = g^r - \lambda w^r$ and so it is locally computable in node p; this comment also holds for the methods below.

Hybrid approximation. This is an improvement over the linear approximation where we add a quadratic term to $\hat{L}_p$. Since the loss term in (8) is total loss (and not averaged loss), this is done by using $(P-1)$ copies of the quadratic term of L_p to approximate the quadratic term of $\sum_{q \neq p} L_q$.

$$\tilde{L}_p(w) = L_p(w), \tag{12}$$

$$\hat{L}_p(w) = (\nabla L(w^r) - \nabla L_p(w^r)) \cdot (w - w^r) + \frac{P-1}{2}(w - w^r)^T H_p^r (w - w^r), \tag{13}$$

where H_p^r is the Hessian of L_p at w^r. This corresponds to using subsampling to approximate the Hessian of $L(w) - L_p(w)$ at w^r utilizing only the local examples. Subsampling based Hessian approximation is known to be very effective in optimization for machine learning (Byrd et al., 2012).

Quadratic approximation. This is a pure quadratic variant where a second order approximation is used for $\tilde{L}_p$ too.

$$\tilde{L}_p(w) = \nabla L_p(w^r) \cdot (w - w^r) + \frac{1}{2}(w - w^r)^T H_p^r (w - w^r), \tag{14}$$

$$\hat{L}_p(w) = (\nabla L(w^r) - \nabla L_p(w^r)) \cdot (w - w^r) + \frac{P-1}{2}(w - w^r)^T H_p^r (w - w^r). \tag{15}$$

The comment made earlier on the goodness of subsampling based Hessian for the Hybrid approximation applies here too.

Nonlinear approximation. Here the idea is to use $P-1$ copies of L_p to approximate $\sum_{q \neq p} L_q$.

$$\tilde{L}_p(w) = L_p(w), \tag{16}$$

$$\hat{L}_p(w) = (\nabla L(w^r) - P\nabla L_p(w^r)) \cdot (w - w^r) + (P-1)L_p(w). \tag{17}$$

A somewhat similar approximation is used in Sharir et al. (2014). But the main algorithm where it is used does not have deterministic monotone descent like our algorithm. The gradient consistency condition, which is essential for establishing function descent, is not respected in that algorithm. In Section 4 we compare our methods against the method in Sharir et al. (2014).

BFGS approximation. For $\tilde{L}_p$ we can either use L_p or a second order approximation, like in the approximations given above. For $\hat{L}_p$ we can use a second order term, $\frac{1}{2}(w - w^r) \cdot H(w - w^r)$ where H is a positive semi-definite matrix; for H we can use a diagonal approximation or keep a limited history of gradients and form a BFGS approximation of $L - L_p$.

Remark 2. The distributed method described above is an instance of Algorithm 1 and so Theorem 2 can be used. In Theorem 2 we mentioned a convergence rate, δ. For $\cos\theta = \sigma/L$ this yields the rate $\delta = (1 - 2\alpha(1-\beta)(\frac{\sigma}{L})^4)$. This rate is obviously pessimistic given that it applies to general choices of $\hat{f}_p$ satisfying minimal assumptions. Actual rates of convergence depend a lot on the choice made for $\hat{f}_p$. Suppose we choose $\hat{f}_p$ via Hybrid, Quadratic or Nonlinear approximation choices mentioned in Subsection 3.2 and minimize $\hat{f}_p$ exactly in the inner optimization. These approximations are invariant to coordinate transformations such as $w' = Bw$, where B is a positive definite matrix. Note that Armijo-Wolfe line search conditions are also unaffected by such transformations. What this means is that, at each iteration, we can, without changing the algorithm, choose for analysis a coordinate transformation that gives the best rate. The Linear approximation choice for $\hat{f}_p$ does not enjoy this property. This explains why the Hybrid, Quadratic and Nonlinear approximations perform so well and give great rates of convergence in practice; see the experiments in Section 4.7 and Subsection 4.9.1. Proving much better convergence rates for FADL using these approximations would be interesting future work.

In Section 4 we evaluate some of these approximations in detail.

3.3. Convergence theory

In practice, exactly minimizing $\hat{f}_p$ is infeasible. For convergence, it is not necessary for w_p to be the minimizer of $\hat{f}_p$; we only need to find w_p such that the direction $d_p = w_p - w^r$ satisfies (1). The angle θ needs to be chosen right. Let us discuss this first. Let $\hat{w}_p^\star$ be the minimizer of $\hat{f}_p$. It can be shown (see appendix B) that $\angle(\hat{w}_p^\star - w^r, -g^r) \leq \cos^{-1}\frac{\sigma}{L}$. To allow for w_p being an approximation of $\hat{w}_p^\star$, we choose θ such that

$$\frac{\pi}{2} > \theta > \cos^{-1}\frac{\sigma}{L}. \tag{18}$$

The following result shows that if an optimizer with *glrc* is used to minimize $\hat{f}_p$, then, only a constant number of iterations is needed to satisfy the sufficient angle of descent condition.

Lemma 3. Assume $g^r \neq 0$. Suppose we minimize $\hat{f}_p$ using an optimizer $\mathcal{M}$ that starts from $v^0 = w^r$ and generates a sequence $\{v^k\}$ having *glrc*, i.e., $\hat{f}_p(v^{k+1}) - \hat{f}_p^\star \leq \delta(\hat{f}_p(v^k) - \hat{f}_p^\star)$, where $\hat{f}_p^\star = \hat{f}_p(\hat{w}_p^\star)$. Then, there exists $\hat{k}$ (which depends only on σ and L) such that $\angle(-g^r, v^k - w^r) \leq \theta \ \ \forall k \geq \hat{k}$.

Lemma 3 can be combined with Theorem 2 to yield the following convergence theorem.

Theorem 4. Suppose θ satisfies (18), $\mathcal{M}$ is as in Lemma 3 and, in each iteration r and for each p, $\hat{k}$ or more iterations of $\mathcal{M}$ are applied to minimize $\hat{f}_p$ (starting from w^r) and get w_p. Then the distributed method converges to a point w satisfying $f(w) - f(w^\star) \leq \epsilon$ in $O(\log(1/\epsilon))$ time.

Proofs of Lemma 3 and Theorem 4 are given in appendix B.

3.4. Practical implementation.

We refer to our method by the acronym, FADL - Function Approximation based Distributed Learning. Going with the practice in numerical optimization, we replace (1) by the condition, $-g^r \cdot d^r > 0$ and use $\alpha = 10^{-4}$, $\beta = 0.9$ in (4) and (5). In actual usage, Algorithm 1 can be terminated when $\|g^r\| \leq \epsilon_g \|g^0\|$ is satisfied at some r. Let us take line search next. On $w = w^r + td^r$, the loss has the form $l(z_i + te_i, y_i)$ where $z_i = w^r \cdot x_i$ and $e_i = d^r \cdot x_i$. Once we have computed $z_i \ \forall i$ and $e_i \ \forall i$, the distributed computation of $f(w^r + td^r)$ and its derivative with respect to t is cheap as it does not involve any computation involving the data, $\{x_i\}$. Thus, many t values can be explored cheaply. Since d^r is determined by approximate optimization, $t = 1$ is expected to give a decent starting point. We first identify an interval $[t_1, t_2] \subset [t_\beta, t_\alpha]$ (see Lemma 1) by starting from $t = 1$ and doing forward and backward stepping. Then we check if t_1 or t_2 is the minimizer of $f(w^r + td^r)$ on $[t_1, t_2]$; if not, we do several bracketing steps in (t_1, t_2) to locate the minimizer approximately. Finally, when using method $\mathcal{M}$, we terminate it after a fixed number of steps, $\hat{k}$; we have found that, setting $\hat{k}$ to match computation and communication costs works effectively. Algorithm 2 gives all the steps of FADL while also mentioning the distributed communications and computations involved.

Choices for $\mathcal{M}$. There are many good methods having (deterministic) *glrc*: L-BFGS, TRON (Lin et al., 2008), Primal coordinate descent (Chang et al., 2008), etc. One could also use methods with *glrc* in the expectation sense (in which case, the convergence in Theorem 4 should be interpreted in some probabilistic sense). This nicely connects our method with recent literature on parallel SGD. We discuss this in the next subsection only briefly as it is outside the scope of the current paper. See our related work (Mahajan et al., 2013b) for details. For the experiments of this paper we do the following. For optimizing the quadratic approximation in (10) with (14)-(15), we used the conjugate-gradient method (Shewchuk, 1994). For all other (non-quadratic) approximations of FADL as well as all nonlinear solvers needed by other methods, we used TRON.

3.5. Connections with parallel SGD

For large scale learning on a single machine, example-wise methods[6] such as stochastic gradient descent (SGD) and its variations (Bottou, 2010; Johnson and Zhang, 2013) and dual

6. These methods update w after scanning each example.

Algorithm 2: FADL - Function Approximation based Distributed Learning. *com:* communication; *cmp:* = computation; *agg:* aggregation. $\mathcal{M}$ is the optimizer used for minimizing $\hat{f}_p$.

Choose w^0;
for $r = 0, 1 \ldots$ **do**
  1. Compute g^r (*com:* w^r; *cmp:* Two passes over data; *agg:* g^r); By-product: $\{z_i = w^r \cdot x_i\}$;
  2. Exit if $\|g^r\| \leq \epsilon_g \|g^0\|$;
  3. **for** $p = 1, \ldots, P$ *(in parallel)* **do**
    4. Set $v^0 = w^r$;
    5. **for** $k = 0, 1, \ldots, \hat{k}$ **do**
      6. Find v^{k+1} using one iteration of $\mathcal{M}$;
    end
    7. Set $w_p = v^{\hat{k}+1}$;
  end
  8. Set d^r as any convex combination of $\{w_p\}$ (*agg:* w_p);
  9. Compute $\{e_i = d^r \cdot x_i\}$ (*com:* d^r; *cmp:* One pass over data);
  10. Do line search to find t (for each t: *com:* t; *cmp:* l and $\partial l/\partial t$ *agg:* $f(w^r + td^r)$ and its derivative wrt t);
  11. Set $w^{r+1} = w^r + td^r$;
end

coordinate ascent (Hsieh et al., 2008) perform quite well. However, example-wise methods are inherently sequential. If one employs a method such as SGD as $\mathcal{M}$, the local optimizer for $\tilde{f}_p$, the result is, in essence, a parallel SGD method. However, with parameter mixing and iterative parameter mixing methods (Mann et al., 2009; Hall et al., 2010; McDonald et al., 2010) (we briefly discussed these methods in Section 1) that do not do line search, convergence theory is limited, even that requiring a complicated analysis (Zinkevich et al., 2010); see also Mann et al. (2009) for some limited results. Thus, the following has been an unanswered question: **Q3.** *Can we form a parallel SGD method with strong convergence properties such as glrc?*

As one special instantiation of our distributed method, we can use, for the local optimization method $\mathcal{M}$, any variation of SGD with *glrc* (in expectation), e.g., the one in Johnson and Zhang (2013). For this case, in a related work of ours (Mahajan et al., 2013b) we show that our method has $O(\log(1/\epsilon))$ time convergence in a probabilistic sense. The result is a strongly convergent parallel SGD method, which answers **Q3**. An interesting side observation is that, the single machine version of this instantiation is very close to the variance-reducing SGD method in Johnson and Zhang (2013). We discuss this next.

Connection with SVRG. Let us take the $\hat{f}_p$ in (11). Let $n_p = |I_p|$ be the number of examples in node p. Define $\psi_i(w) = n_p l(w \cdot x_i, y_i) + \frac{\lambda}{2}\|w\|^2$. It is easy to check that

$$\nabla \hat{f}_p(w) = \frac{1}{n_p} \sum_{i \in I_p} (\nabla \psi_i(w) - \nabla \psi_i(w^r) + g^r). \tag{19}$$

Thus, plain SGD updates applied to $\hat{f}_p$ has the form

$$w = w - \eta(\nabla\psi_i(w) - \nabla\psi_i(w^r) + g^r), \tag{20}$$

which is precisely the update in SVRG. In particular, the single node ($P = 1$) implementation of our method using plain SGD updates for optimizing $\hat{f}_p$ is very close to the SVRG method.[7] While Johnson and Zhang (2013) motivate the update in terms of variance reduction, we derive it from a functional approximation viewpoint.

3.6. Computation-Communication tradeoff

In this subsection we do a rough analysis to understand the conditions under which our method (FADL) is faster than the SQM method (Chu et al., 2006; Agarwal et al., 2011) (see Section 1). This analysis is only for understanding the role of various parameters and not for getting any precise comparison of the speed of the two methods.

Compared to the SQM method, FADL does a lot more computation (optimize $\hat{f}_p$) in each node. On the other hand FADL reaches a good solution using a much smaller number of outer iterations. Clearly, FADL will be attractive for problems with high communication costs, e.g., problems with a large feature dimension. For a given distributed computing environment and specific implementation choices, it is easy to do a rough analysis to understand the conditions in which FADL will be more efficient than SQM. Consider a distributed grid of nodes in an *AllReduce* tree. Let us use a common method such as TRON for implementing SQM as well as for $\mathcal{M}$ in FADL. Assuming that $T^{\text{outer}}_{\text{SQM}} > 3.0 T^{outer}_{\text{FADL}}$ (where $T^{\text{outer}}_{\text{FADL}}$ and T^{outer}_{SQM} are the number of outer iterations required by SQM and FADL), we can do a rough analysis of the costs of SQM and FADL (see appendix A for details) to show that FADL will be faster when the following condition is satisfied.

$$\frac{nz}{m} < \frac{\gamma P}{2\hat{k}} \tag{21}$$

where: nz is the number of nonzero elements in the data, i.e., $\{x_i\}$; m is the feature dimension; γ is the relative cost of communication to computation (e.g. $100 - 1000$); P is the number of nodes; and $\hat{k}$ is the number of inner iterations of FADL. Thus, the larger the dimension (m) is, and the higher the sparsity in the data is, FADL will be better than SQM.

4. Experiments

In this section, we demonstrate the effectiveness of our method by comparing it against several existing distributed training methods on five large data sets. We first discuss our experimental setup. We then briefly list each method considered and then do experiments to decide the best overall setting for each method. This applies to our method too, for which the setting is mainly decided by the choice made for the function approximation, $\hat{f}_p$; see Subsection 3.2 for details of these choices. Finally, we compare, in detail, all the methods under their best settings. This study clearly demonstrates scenarios under which our method performs better than other methods.

7. Note the subtle point that applying SVRG method on $\hat{f}_p$ is different from doing (20), which corresponds to plain SGD. It is the former that assures *glrc* (in expectation).

4.1. Experimental Setup

We ran all our experiments on a Hadoop cluster with 379 nodes and 10 Gbit interconnect speed. Each node has Intel (R) Xeon (R) E5-2450L (2 processors) running at 1.8 GHz. Since iterations in traditional *MapReduce* are slower (because of job setup and disk access costs), as in Agarwal et al. (Agarwal et al., 2011), we build an *AllReduce* binary tree between the mappers[8]. The communication bandwidth is *1 Gbps* (gigabits per sec).

Dataset	#Examples (n)	#Features (m)	#Non-zeros (nz)	λ/n
kdd2010	8.41×10^6	20.21×10^6	0.31×10^9	1.25×10^{-6}
url	1.91×10^6	3.23×10^6	0.22×10^9	0.11×10^{-6}
webspam	0.35×10^6	16.6×10^6	0.98×10^9	1.0×10^{-4}
mnist8m	8.1×10^6	784	6.35×10^9	1.0×10^{-4}
rcv	0.5×10^6	47236	0.50×10^8	1.0×10^{-4}

Table 1: Properties of datasets.

Data Sets. We consider the following publicly available datasets having a large number of examples:[9] *kdd2010*, *url*, *webspam*, *mnist8m* and *rcv*. Table 1 shows the numbers of examples, features, nonzero in data matrix and the values of regularizer λ used. The regularizer for each dataset is chosen to be the optimal value that gives the best performance on a small validation set. We use these datasets mainly to illustrate the validity of theory, and its utility to distributed machine learning. In real scenarios of Big data, the datasets are typically much larger. Note that *kdd2010*, *url* and *webspam* are large dimensional (m is large) while *mnist8m* and *rcv* are low/medium dimensional (m is not high). This division of the datasets is useful because communication cost in example-partitioned distributed methods is mainly dependent on m (see Appendix A) and so these datasets somewhat help to see the effect of communications cost.

We use the *squared-hinge loss* function for all the experiments. Unless stated differently, for all numerical optimizations we use the Trust Region Newton method (TRON) proposed in Lin et al. (2008).

Evaluation Criteria. We use the relative difference to the optimal function value and the Area under Precision-Recall Curve (AUPRC) (Sonnenburg and Franc, 2010; Agarwal et al., 2013)[10] as the evaluation criteria. The former is calculated as $(f - f^*)/f^*$ in log scale, where f^* is the optimal function value obtained by running the *TERA* algorithm (see below) for a very large number of iterations.

4.2. Methods for comparison

We compare the following methods.

8. Note that we do not use the pipelined version and hence we incur an extra multiplicative $logP$ cost in communication.
9. These datasets are available at: `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/`. For *mnist8m* we solve the binary problem of separating class "3" from others.
10. We employed AUPRC instead of AUC because it differentiates methods more finely.

- *TERA:* The Terascale method (TERA) (Agarwal et al., 2011) is the best representative method from the SQM class (Chu et al., 2006). It can be considered as the state-of-the-art distributed solver and therefore an important baseline.

- *ADMM:* We use the example partitioning formulation of the Alternating Direction Method of Multipliers (ADMM) (Boyd et al., 2011; Zhang et al., 2012). ADMM is a dual method which is very different from our primal method; however, like our method, it solves approximate problems in the nodes and iteratively reaches the full batch solution.

- *CoCoA:* This method (Jaggi et al., 2014) represents the class of distributed dual methods (Pechyony et al., 2011; Yang, 2013; Yang et al., 2013; Jaggi et al., 2014) that, in each outer iteration, solve (in parallel) several local dual optimization problems.

- *DANE:* This is the Newton based method described in Sharir et al. (2014) that uses a function approximation similar to FADL.

- *DiSCO:* This (Zhang and Xiao, 2015) is a distributed Newton method designed with communication-efficiency in mind.

- *Our method (FADL):* This is our method described in detail in Section 3 and more specifically, in Algorithm 2.

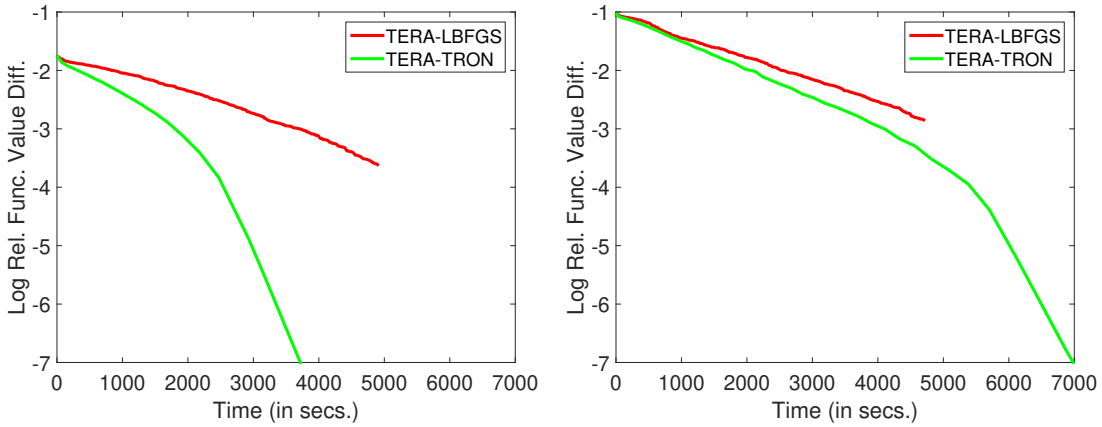
4.3. Study of TERA

A key attractive property of TERA is that the number of outer iterations pretty much remains constant with respect to the number of distributed nodes used. As recommended by Agarwal et al. (2011), we find a local weight vector per node by minimizing the local objective function (based only on the examples in that node) using five epochs of SGD (Bottou, 2010). (The optimal step size is chosen by running SGD on a subset of data.) We then average the weights from all the nodes (on a per-feature basis as explained in Agarwal et al. (2011)) and use the averaged weight vector to warm start *TERA*.[11] Agarwal et al. (2011) use the LBFGS method as the trainer whereas we use TRON. To make sure that this does not lead to bias, we try both, TERA-LBFGS and TERA-TRON. Figure 1 compares the progress of objective function for these two choices. Clearly, TERA-TRON is superior. We observe similar behavior on the other datasets also. Given this, we restrict ourselves to TERA-TRON and simply refer to it as TERA.

4.4. Study of ADMM

The ADMM objective function (Boyd et al., 2011) has a quadratic proximal term called augmented Lagrangian with a penalty parameter ρ multiplying it. In general, the performance of ADMM is very sensitive to the value of ρ and hence making a good choice for it is crucial. We consider three methods for choosing ρ.

11. We use this inexpensive initialization for FADL and ADMM too. It is not applicable to CoCoA. Because of this, CoCoA starts with a different primal objective function value than others.



(a) *kdd2010* - 8 nodes

(b) *kdd2010* - 128 nodes

Figure 1: Plots showing the time efficiency of TERA methods for *kdd2010*.

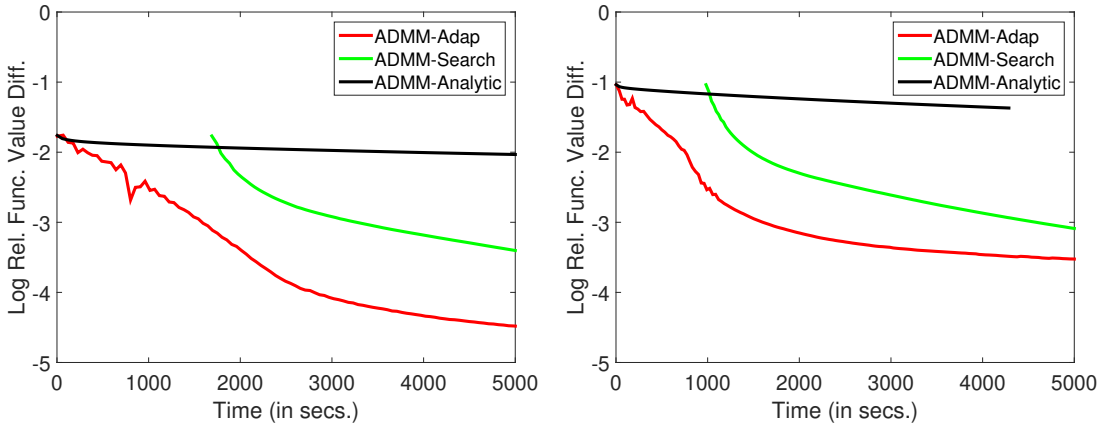
Even though there is no supporting theory, Boyd et al (Boyd et al., 2011) suggest an approach by which ρ is adapted in each iteration; see Equation (3.13) in Section 3.4.1 of that paper. We will refer to this choice as *Adap*.

Recently, Deng and Yin (2012) proved a linear rate of convergence for ADMM under assumptions A1 and A2 (see Section 2) on ADMM functions. As a result, their analysis also hold for the objective function in (8). They also give an analytical formula to set ρ in order to get the best theoretical linear rate constant. We will refer to this choice of ρ as *Analytic*.

We also consider a third choice, *ADMM-Search* in which, we start with the value of ρ given by *Analytic*, choose several values of ρ in its neighborhood and select the best ρ by running ADMM for 10 iterations and looking at the objective function value. Note that this step takes additional time and causes a late start of ADMM.

Figure 2 compares the progress of the training objective function for the three choices on *kdd2010* for $P = 8$ and $P = 128$. *Analytic* is an order of magnitude slower than the other two choices. *Search* works well. However, a significant amount of initial time is spent on finding a good value for ρ, thus making the overall approach slow. *Adap* comes out to be the best performer among the three choices. Similar observations hold for other datasets and other choices of P. So, for ADMM, we will fix *Adap* as the way of choosing ρ and refer to ADMM-*Adap* simply as ADMM.

It is also worth commenting on methods related to ADMM. Apart from ADMM, Bertsekas and Tsitsiklis (1997) discuss several other classic optimization methods for separable convex programming, based on proximal and Augmented Lagrangian ideas which can be used for distributed training of linear classifiers. ADMM represents the best of these methods. Also, Gauss-Seidel and Jacobi methods given in Bertsekas and Tsitsiklis (1997) are related to feature partitioning, which is very different from the example partitioning scenario studied in this paper. Therefore we do not consider these methods.



(a) *kdd2010* - 8 nodes

(b) *kdd2010* - 128 nodes

Figure 2: Plots showing the time efficiency of ADMM methods for *kdd2010*.

4.5. Study of CoCoA

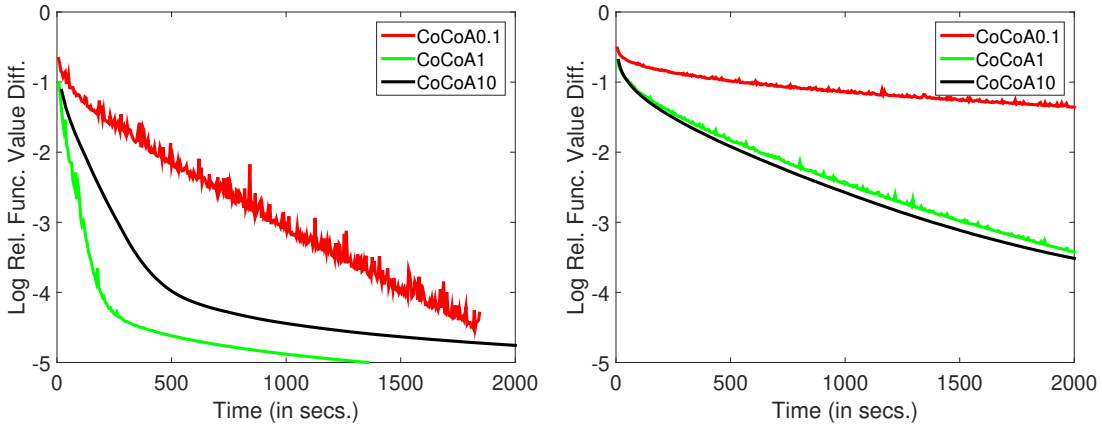
In CoCoA (Jaggi et al., 2014) the key parameter is the approximation level of the inner iterations used to solve each projected dual sub-problem. The number of epochs of coordinate dual ascent inner iterations plays a crucial role. We try the following choices for it: 0.1, 1 and 10.[12] Figure 3 compares the progress of the objective function on *kdd2010* for two choices of nodes, $P = 8$ and $P = 128$. We find the choice of 1 epoch to work well reasonably consistently over all the five datasets and varying number of nodes. So we fix this choice and refer to the resulting method simply as CoCoA. Note in Figure 3 that the (primal) objective function does not decrease continuously with time. This is because it is a dual method and so monotone descent of the objective function is not assured.[13]

4.6. Study of DANE

In our detailed comparison study later in this section, we will also include another method called DANE, which has some resemblance to FADL. This is the Newton based method described in Sharir et al. (2014). Even though this method is very different in spirit from our method, it uses a function approximation similar to our Nonlinear approximation idea; we briefly discussed this in Subsection 3.2. DANE is a non-monotone method that is based on fixed step sizes, with a probabilistic convergence theory to support it.

DANE has two parameters, μ and η in the function approximation; μ is the coefficient for the proximal term and η is used for defining the direction. Sharir et al. (2014) do not prove convergence for any possible choices of μ and η values; their practical recommendation is to use $\mu = 3\lambda$ and $\eta = 1$. The choice of μ in particular, turns out to be quite sub-optimal. We found that there was no single value of μ that is good for all datasets, and so it needs to be tuned for each dataset. So, to improve DANE, we included an initial μ-tuning step.

12. The examples were randomly shuffled for presentation. When the number of epochs is a fraction, e.g., 0.1, the inner iteration was stopped after 10% of the examples were presented.
13. The same comment holds for ADMM which is also a dual method; see for example, the jumps in objective function values for ADMM in Figures 6 and 8 for *kdd2010*.



(a) *kdd2010* - 8 nodes (b) *kdd2010* - 128 nodes

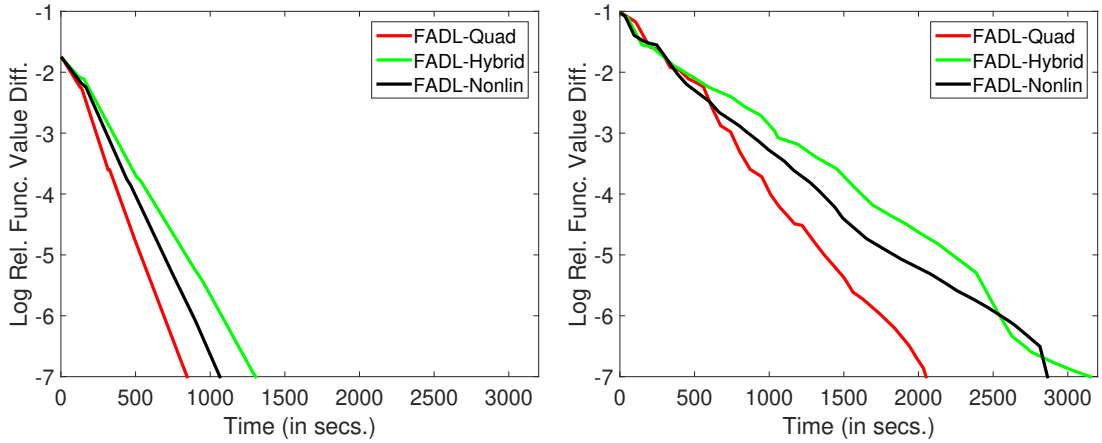
Figure 3: Plots showing the time efficiency of CoCoA settings for *kdd2010*.

We start with $\mu = 3 * \lambda$ and use an initial set of four outer iterations to choose the μ value that gives the best improvement in objective function. Starting from $\mu = 3 * \lambda$ we try several values of μ in the direction of μ change that leads to improvement. After the first four outer iterations, we fix μ at the chosen best value for the remaining iterations. (Note that the cost associated with tuning μ has to be included in the overall cost of DANE.) As we will see later in Subsection 4.9, in spite of all this tuning, DANE did not converge in several situations. Essentially, the issue is that μ has to be adaptive - DANE needs to use different choices of μ in the early, middle and end stages of one training. But then, this is nearly akin to doing a kind of line search in each outer iteration. As opposed to DANE, note that FADL is a monotone method directly based on line search, and it does not need the proximal term to restrict step-sizes. Also, unlike DANE, all the parameters of FADL are fixed to default values for all datasets.

The choice of inner optimizer for DANE is crucial for its efficiency. We tried SVRG (Johnson and Zhang, 2013) and also TRON (Lin et al., 2008). Both did well, but SVRG was better, especially for small number of nodes. So we did a more detailed study of SVRG. We tried two choices. (a) Fix the number of epochs for the local (inner) optimization to some good value, e.g., 10. (b) Always choose the number of epochs for the local (inner) optimization to be such that the cost of local computation in a node is matched with the communications cost. Choice (b) gave a better solution. So we have used it for DANE.

4.7. Study of FADL

Recall from Subsection 3.2 the various choices that we suggested for $\hat{f}_p$. We are yet to implement and study the BFGS approximation; we leave it out for future work. Our original work (Mahajan et al., 2013a) focused on the Linear approximation, but we found the Quadratic, Hybrid and Nonlinear choices to work much better. So we study only these three methods. The implementation of these methods is as described in Subsection 3.4 and Algorithm 2.



(a) *kdd2010* - 8 nodes (b) *kdd2010* - 128 nodes

Figure 4: Plots showing the time efficiency of the three function approximations of FADL for *kdd2010*.

Figure 4 compares the progress of the training objective function for various choices of $\hat{f}_p$. Among the choices, the quadratic approximation for $\hat{f}_p$ gives the best performance, although the Hybrid and Nonlinear approximations also do quite well. We observe this reasonably consistently in other datasets too. Hence, from the set of methods considered in this subsection we choose FADL-Quadratic approximation as the only method for further analysis, and simply refer to this method as FADL hereafter.

Why does the quadratic approximation do better than hybrid and nonlinear approximations? We do not have a precise answer to this question, but we give some arguments in support. In each outer iteration, the function approximation idea is mainly used to get a good direction. Recall from Subsection 3.2 that different choices use different approximations for $\tilde{L}_p$ and $\hat{L}_p$. Using the same "type" (meaning linear, nonlinear or quadratic) for both, $\tilde{L}_p$ and $\hat{L}_p$ is possibly better for direction finding. Second, the direction finding could be more sensitive to the nonlinear approximation compared to the quadratic approximation; this could become more severe as the number of nodes becomes larger. Literature shows that quadratic approximations have good robustness properties; for example, subsampling in Hessian computation (Byrd et al., 2012) doesn't worsen direction finding much.

4.8. Comparison of FADL against DiSCO

DiSCO (Zhang and Xiao, 2015) is an inexact damped Newton method, where the inexact Newton steps are computed using a distributed preconditioned conjugate gradient method. Like TERA, it belongs to the SQM class (Chu et al., 2006). The DiSCO method as described in Zhang and Xiao (2015) is not designed for squared hinge loss. Therefore, in this subsection we separately compare FADL against DiSCO using the logistic loss. With communication efficiency in mind we compare the two methods in terms of the number of communication passes. Figure 5 gives the progress in objective function as a function of the number of communication passes for the two methods. FADL is clearly better than DiSCO, the

difference being quite large on datasets such as *kdd2010*. In terms of total computing time, the superiority of FADL turns out to be even better than what is seen in the plots of communication passes. This is because DiSCO requires more extensive computational steps than FADL within one communication pass.

4.9. Comparison of FADL against TERA, ADMM, CoCoA and DANE

Having made the best choice of settings for the methods, we now evaluate FADL against TERA, ADMM, CoCoA and DANE in more detail. We do this using three sets of plots. We give details only for $P = 8$ and $P = 128$ to give an idea of how performance varies for small and large number of nodes.

1. *Communication passes.* We plot the variation of the training objective function as a function of the number of communication passes. For the x-axis we prefer the number of communication passes instead of the number of outer iterations since the latter has a different meaning for different methods while the former is quite uniform for all methods. Figures 6 and 7 give the plots respectively for the large dimensional (m large) datasets (*kdd2010*, *url* and *webspam*) and medium/small dimensional (m medium/small) datasets (*mnist8m* and *rcv*).

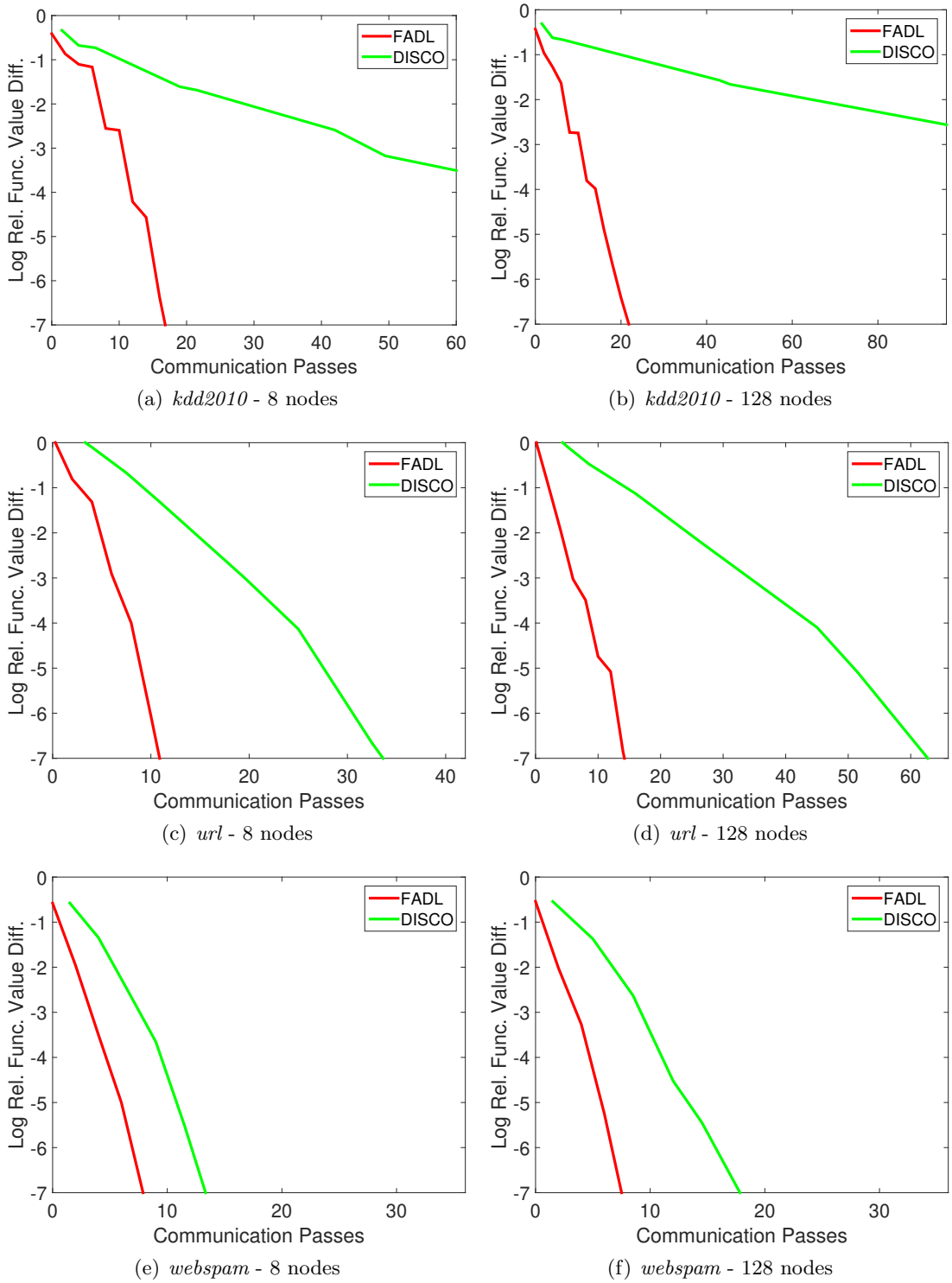
2. *Time.* We plot the variation of the training objective function as a function of the actual solution time. Figures 8 and 9 give the plots respectively for the large dimensional and medium/small dimensional datasets.

3. *Speed-up over TERA.* TERA is an established strong baseline method. So it is useful to ask how other methods fare relative to TERA and study this as a function of the number of nodes. For doing this we need to represent each method by one or two real numbers that indicate performance. Since generalization performance is finally the quantity of interest, we stop a method when it reaches within 0.1% of the steady state AUPRC value achieved by full, perfect training of (8) and record the following two measures: the total number of communication passes and the total time taken. For each measure, we plot the ratio of the measure's value for TERA to the corresponding measure's value for a method, as a function of the number of nodes, and repeat this for each method. Larger this ratio, better is a method; also, ratio greater than one means a method is faster than TERA. Figures 10 and 11 give the plots for all the five datasets.

Let us now use these plots to compare the methods. In the plots, DANE starts later than other methods due to the extra work needed for tuning the proximal parameter μ.

4.9.1. Rate of Convergence

Analysis of the rate of convergence is better done by studying the behavior of the training objective function with respect to the number of communication passes. So it is useful to look at Figures 6 and 7. Clearly, as predicted by theory, the rate of convergence is linear for all methods.



(a) *kdd2010* - 8 nodes

(b) *kdd2010* - 128 nodes

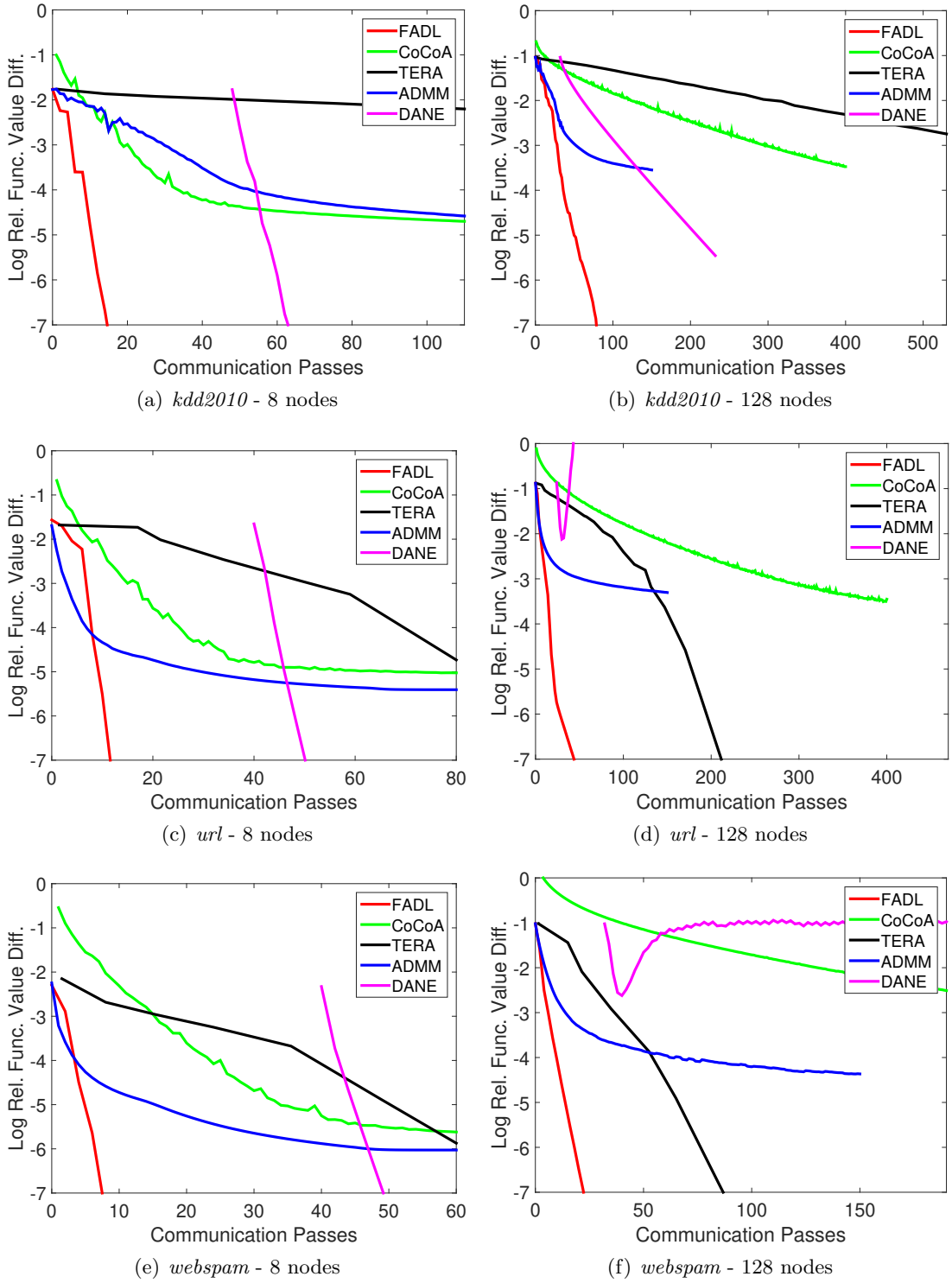
(c) *url* - 8 nodes

(d) *url* - 128 nodes

(e) *webspam* - 8 nodes

(f) *webspam* - 128 nodes

Figure 5: Plots comparing the number of communication passes of DiSCO and FADL on three datasets.



(a) *kdd2010* - 8 nodes

(b) *kdd2010* - 128 nodes

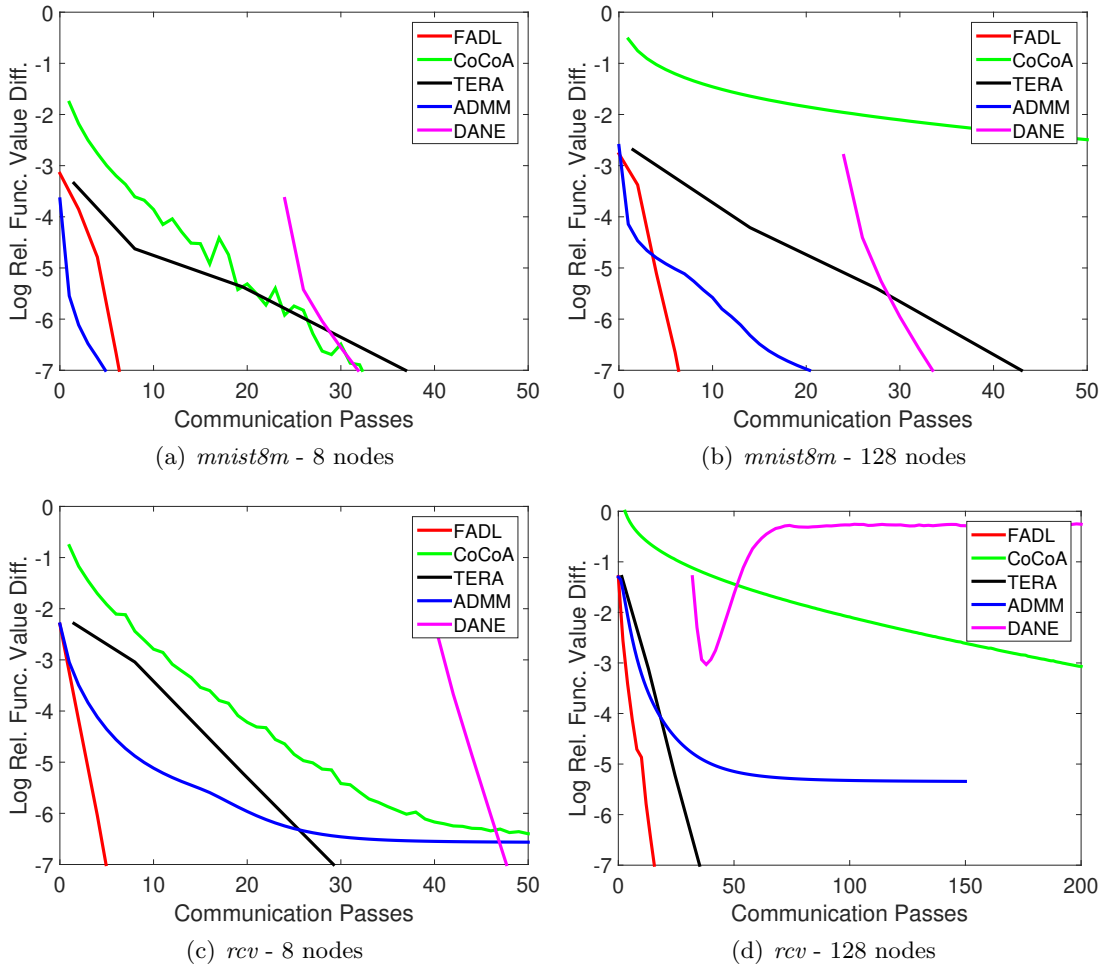
(c) *url* - 8 nodes

(d) *url* - 128 nodes

(e) *webspam* - 8 nodes

(f) *webspam* - 128 nodes

Figure 6: Plots showing the rate of convergence of various methods for the three high dimensional datasets.



(a) *mnist8m* - 8 nodes

(b) *mnist8m* - 128 nodes

(c) *rcv* - 8 nodes

(d) *rcv* - 128 nodes

Figure 7: Plots showing the linear convergence of various methods for the two low/medium dimensional datasets.

TERA uses distributed computation only to compute the gradient and so the plots should be unaffected by P. But in the plots we do see differences between the plots for $P = 8$ and $P = 128$. This is because of their different initialization (average of one pass SGD solutions from nodes): the initialization with lower number of nodes is better due its smaller variance; note also the better starting objective function value at the start (left most point) for $P = 8$.

For FADL, the rate is steeper for $P = 8$ than for $P = 128$. This steeper behavior for lower number of nodes is expected because the functional approximation in each node becomes better as the number of nodes decreases.

Recall from Section 1 that, our main aim behind the design of the function approximation based methods is to reduce the number of communication passes significantly. The plots in Figures 6 and 7 clearly confirm such a reduction.

Even though the end convergence rate of ADMM is slow, it generally shows good rates of convergence in the initial stages of training. This is a useful behavior because generalization measures such as AUPRC tend to achieve steady state values quickly in the early stages. This usefulness is seen in Figure 10 too.

DANE has good rates of convergence when P is small ($P = 8$), but the cost associated with tuning μ makes it poor; note that, if μ is not tuned, that will affect the rate of convergence. DANE tends to be unstable for large P values.

Overall, FADL gives much better rates of convergence (both, in the early training stage as well as in the end stage) compared to TERA, CoCoA, ADMM and DANE methods.[14] FADL shows a large reduction in the number of communication passes over TERA, especially when the number of nodes is small. Against CoCoA the trend is the other way: FADL needs a much smaller number of communication passes than CoCoA especially when the number of nodes is large. These observations can also be seen from Figure 10. Clearly CoCoA seems to be very slow with increasing number of nodes.

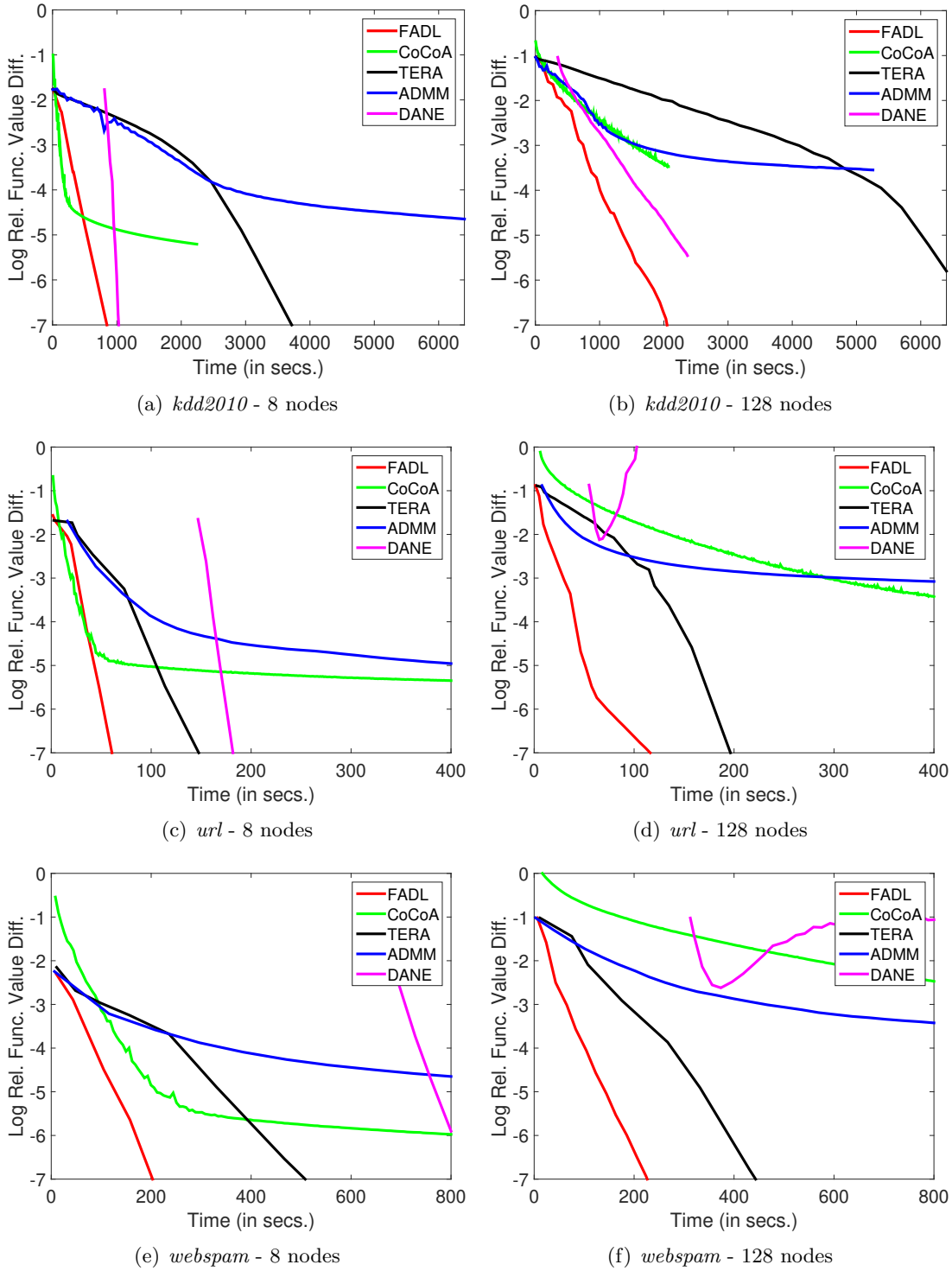
4.9.2. Time Taken

In the previous analysis we ignored computation costs within each iteration. But these costs play a key role when we analyze overall efficiency in terms of the actual time taken. We study this next. Figures 8 and 9 are relevant for this study. FADL, ADMM, CoCoA and DANE involve much more extensive computations in the inner iterations than TERA; this is especially true when the number of nodes is small because of the large amount of local data in each node. TERA fares much better in the time analysis than what we saw while studying using communication passes only. Thus the gap between TERA and other methods becomes a lot smaller in the time analysis. Compare, for example, TERA and ADMM with respect to communication passes and time. Although ADMM is much more efficient than TERA with respect to the number of communication passes, TERA catches up nicely with ADMM on the time taken.

CoCoA does well sometimes; for example, on *kdd2010* and *url*, when the number of nodes is small, say $P = 8$. But it is slow otherwise, especially when the number of nodes is large.

FADL is uniformly better than ADMM with respect to the total time taken. Overall, FADL shows the best performance, performing equally or much better than other methods in different situations. With medium/low dimensional datasets (see Figure 9), communication time is less of an issue and so the expectation is that FADL is less of value for them. Even on these datasets, FADL does equally or better than TERA. DANE is not competitive due to the time involved for tuning μ.

14. Here, it is useful to recall the comments made in Remark 2 of Section 3 on the goodness of the convergence rate of FADL.



(a) *kdd2010* - 8 nodes

(b) *kdd2010* - 128 nodes

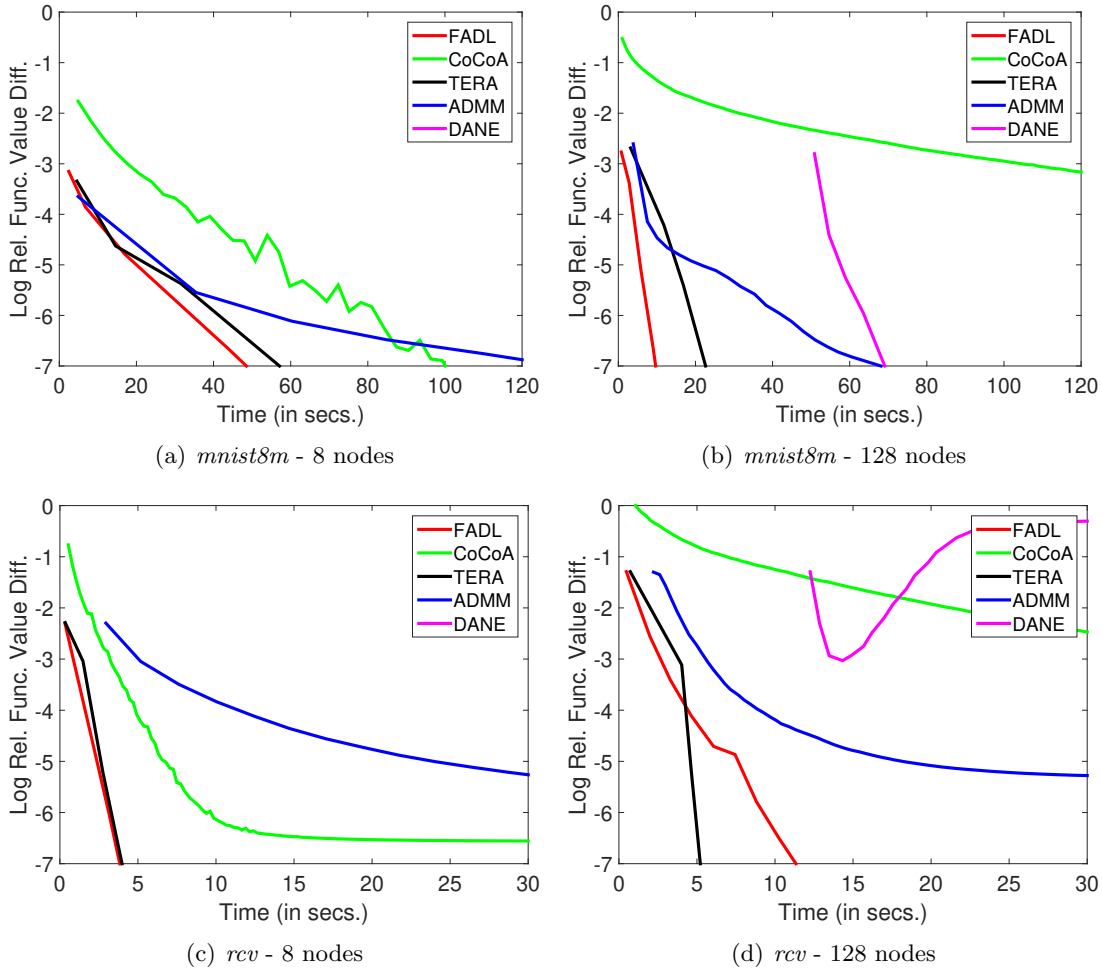
(c) *url* - 8 nodes

(d) *url* - 128 nodes

(e) *webspam* - 8 nodes

(f) *webspam* - 128 nodes

Figure 8: Plots showing the time efficiency of various methods for the three high dimensional datasets.



(a) *mnist8m* - 8 nodes

(b) *mnist8m* - 128 nodes

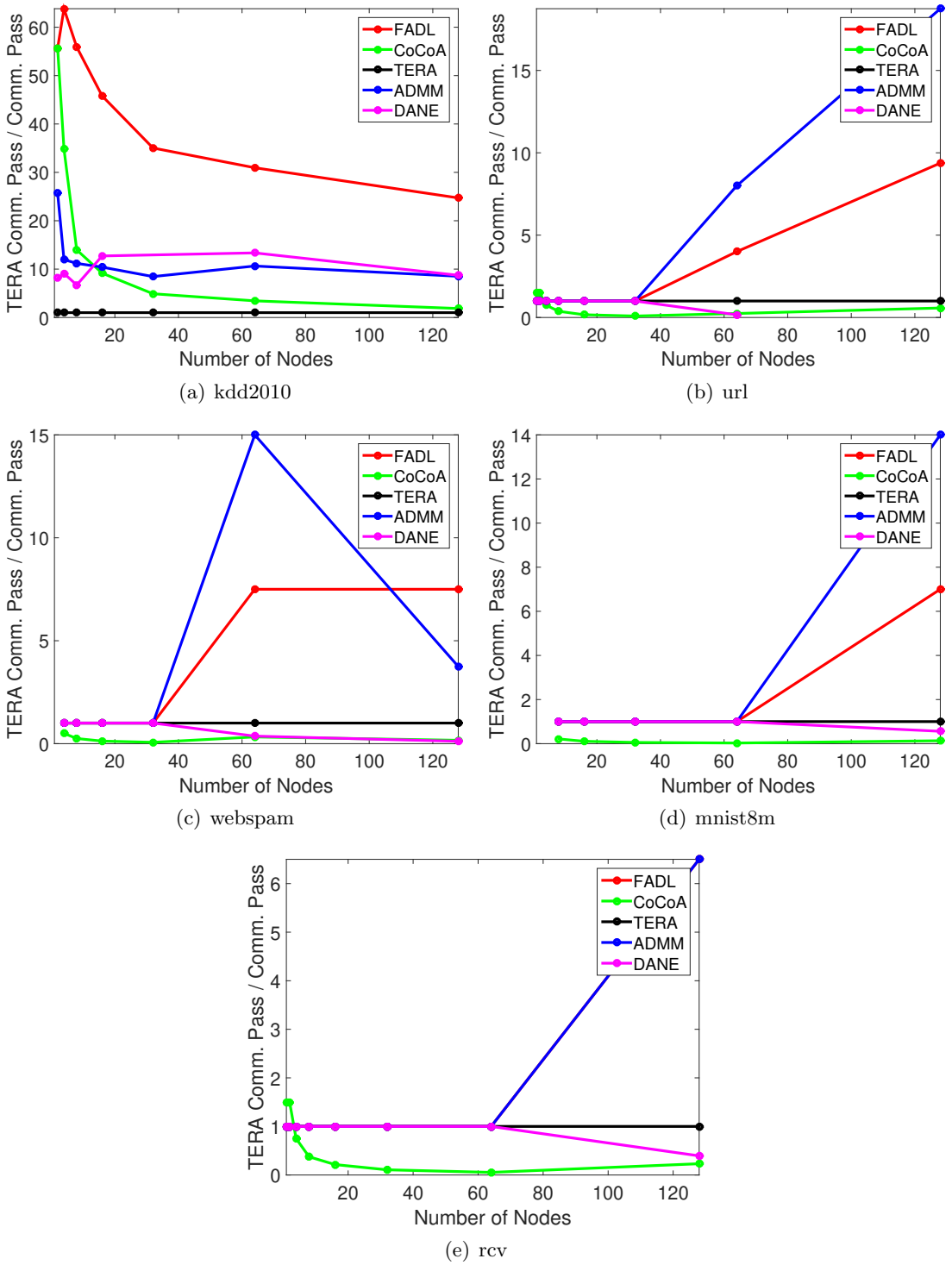
(c) *rcv* - 8 nodes

(d) *rcv* - 128 nodes

Figure 9: Plots showing the time efficiency of various methods for the two low/medium dimensional datasets. For *rcv* and *mnist8m* - 8 nodes, the plot of DANE is invisible since the time taken for tuning μ is larger than the time window displayed.

4.9.3. Relative performance of the methods

Figure 11 is relevant for this study. CoCoA shows impressive speed-up over TERA on *kdd2010* but it is much slower on all the other datasets. It is unclear why CoCoA fares so well on *kdd2010* but not on the other datasets. ADMM gives an overall decent performance when compared to TERA. FADL is consistently faster than TERA, with speed-ups ranging anywhere from 1-10. In communication-heavy scenarios where reducing the number of communication passes is most important, methods such as FADL and ADMM have great value (see Figure 10), with the possibility of getting even higher speed-ups over TERA. Except for *kdd2010* for which FADL is slower than CoCoA for small number of nodes, it is generally the fastest method. DANE does not seem to have any special scenario where it is better than others.



(a) kdd2010

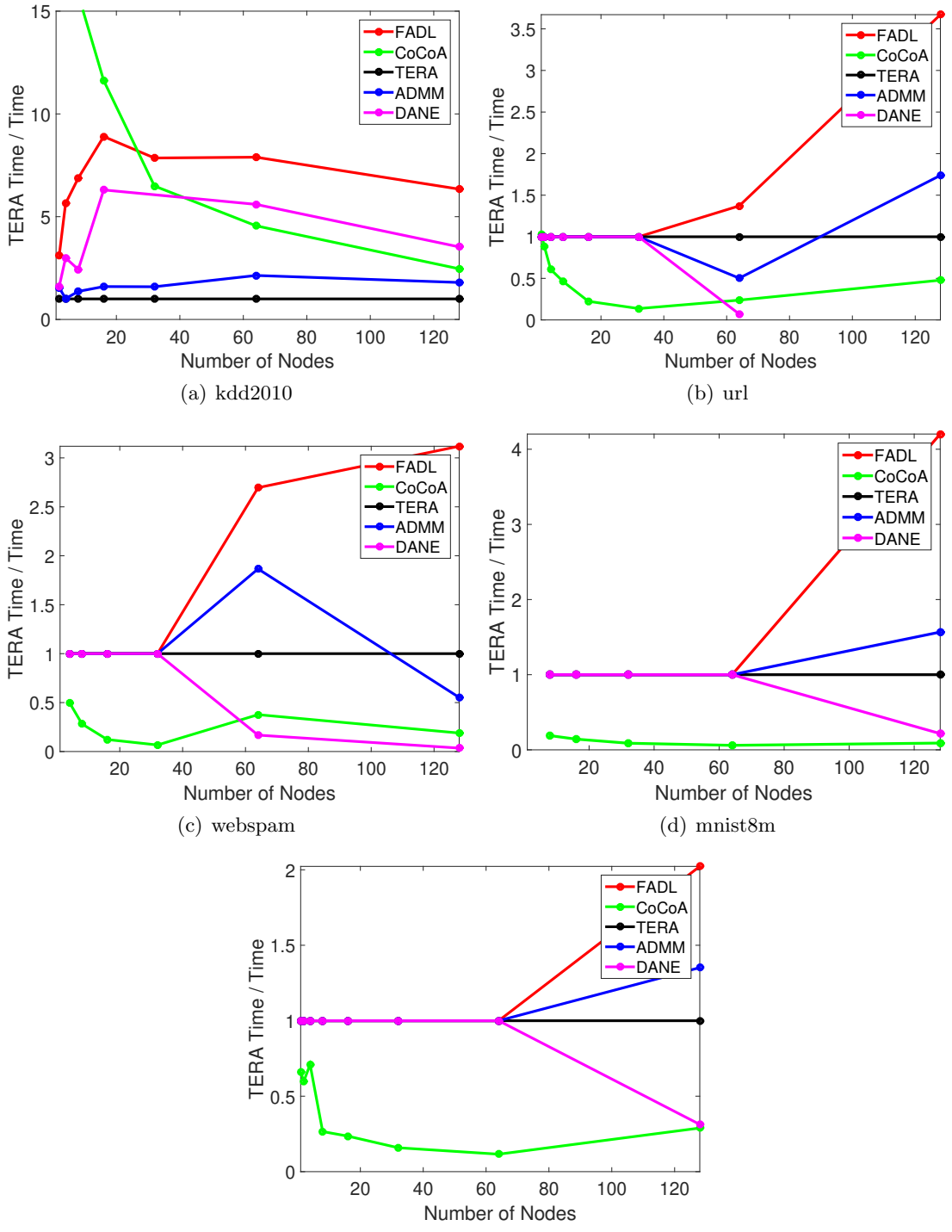
(b) url

(c) webspam

(d) mnist8m

(e) rcv

Figure 10: Plots showing communication passes (relative to TERA) as a function of the number of nodes. Each method was terminated when it reached within 0.1% of the steady state AUPRC value achieved by full, perfect training of (8). For *rcv*, the FADL and ADMM curves coincide.



(a) kdd2010

(b) url

(c) webspam

(d) mnist8m

(e) rcv

Figure 11: Plots showing time (relative to TERA) as a function of the number of nodes. Each method was terminated when it reached within 0.1% of the steady state AUPRC value achieved by full, perfect training of (8).

4.9.4. Speed-up as a function of P

Let us revisit Figure 8 and look at the plots corresponding to *kdd2010* for FADL.[15] It can be observed that the time needed for reaching a certain tolerance, say *Log Rel. Func. Value Diff. = -3*, is two times smaller for $P = 8$ than for $P = 128$. This means that using a large number of nodes is not useful, which prompts the question: *Is a distributed solution really necessary?* There are two answers to this question. First, as we already mentioned, when the training data is huge[16] and *the data is generated and forced to reside in distributed nodes (moving data between machines is not an efficient option)*, the right question to ask is not whether we get great speed-up, but to ask which method is the fastest. Second, for a given dataset and method, if the time taken to reach a certain approximate stopping tolerance (e.g., based on AUPRC) is plotted as a function of P, it usually has a minimum at a value $P > 1$. Given this, it is appropriate to choose a P optimally to minimize training time. A large fraction of Big data machine learning applications involve periodically repeated model training involving newly added data. For example, in Advertising, logistic regression based click probability models are retrained on a daily basis on incrementally varying datasets. In such scenarios it is worthwhile to spend time to tune P in an early deployment phase to minimize time, and then use this choice of P for future runs.

4.9.5. Computation and Communication Costs

Table 2 shows the ratio of computational cost to communication cost for the three high dimensional datasets for all the methods.[17] Note that the ratio is small for *TERA* and so communication cost dominates the time for it. On the other hand, both the costs are well balanced for FADL. Note that ratio varies in the range of $0.625 - 2.845$. This clearly shows that FADL trades-off computation with communication, while significantly reducing the number of communication passes (Figures 6 and 7) and time (Figures 8 and 9).

	FADL	CoCoA	TERA	ADMM
kdd2010	1.6333	0.1416	0.1422	1.8499
url	1.3650	0.1040	0.2986	3.4886
webspam	1.2082	0.1570	0.2423	1.2543

Table 2: Ratio of the total computation cost to the total communication cost for various methods which were terminated when AUPRC reached within 0.1% of the AUPRC value for 128 nodes.

15. We choose FADL as an example, but the comments made in the discussion apply to other methods too.
16. The datasets, *kdd2010*, *url* and *webspam* are really not huge in the *Big data* sense. In this paper we used them only because of lack of availability of much bigger public datasets.
17. For the medium/low dimensional datasets *rcv* and *mnist8m*, communication latencies, line search cost etc. also play a key role and an analysis of computation cost versus communication cost does not provide any great insight.

4.10. Experiment on a much larger dataset

To verify the goodness of FADL, we also did an experiment evaluating FADL against other methods, on a large dataset that is more than an order of magnitude bigger than the largest dataset in Table 1. The dataset is the Splice site recognition dataset from the bioinformatics domain (Sonnenburg and Franc, 2010). In this dataset each example is a sequence; we considered all positional features upto 9 grams. This led to a dataset of 49 million features and 50 million examples. The size of the dataset is larger than 0.65 Terabytes. We employed a cluster of 100 nodes to solve this problem. Figure 12 compares the various methods on (a) the reduction of the objective function as a function of communication passes; (b) the reduction of the objective function as a function of time; and (c) the improvement of generalization performance (AUPRC) as a function of time. It is clear that FADL makes great reductions over other methods, on the number of communication passes. FADL is also the best performer when we measure by the time taken. On clusters with slower communication speeds and iteration set-up times, the value of FADL over other methods will be even higher. Interestingly, CoCoA comes out as the next best performer. CoCoA has slower end convergence than FADL, but it shows up equally well in plot (c) on the improvement of generalization performance. As we saw earlier with other datasets, the value of CoCoA varies a lot; for the current scenario of solving the splice site recognition on 100 nodes it seems to be well-suited. FADL, on the other hand, is uniformly good in varying scenarios of several datasets, number of nodes etc.

4.11. Summary

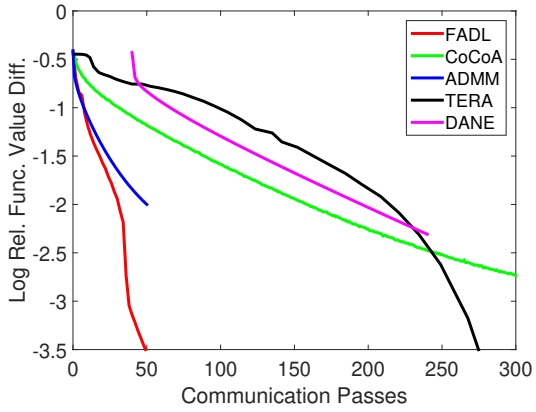
It is useful to summarize the findings of the empirical study.

- FADL gives a great reduction in the number of communication passes, making it clearly superior to other methods in communication heavy settings.
- In spite of higher computational costs per iteration FADL shows the overall best performance on the total time taken. This is true even for medium and low dimensional datasets.
- FADL shows a speed-up of 1-10 over TERA, the actual speed-up depending on the dataset and the setting.
- FADL nicely balances computation and communication costs.

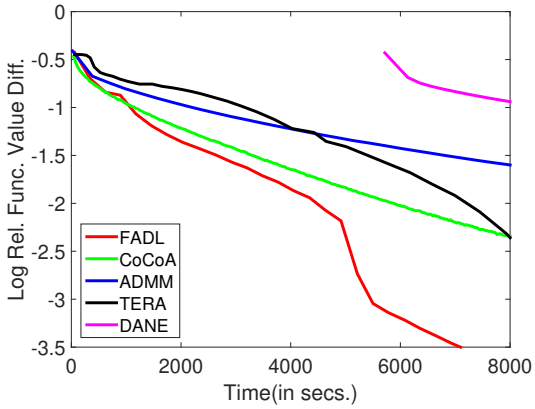
5. Discussion

In this section, we discuss briefly, other different distributed settings made possible by our algorithm. The aim is to show the flexibility and generality of our approach while ensuring *glrc*.

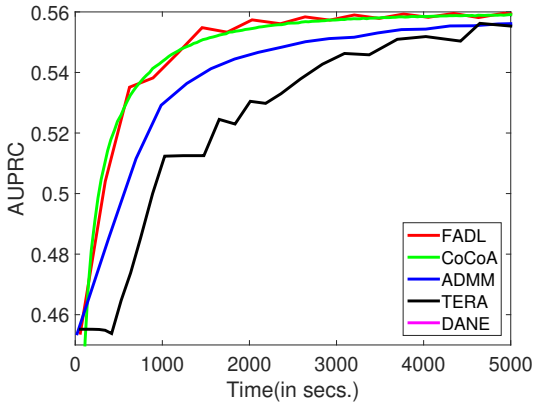
Section 3 considered example partitioning where examples are distributed across the nodes. First, it is worth mentioning that, due to the gradient consistency condition, *partitioning* is not a necessary constraint; our theory allows examples to be resampled, i.e., each example is allowed to be a part of any number of nodes arbitrarily. For example, to reduce the number of outer iterations, it helps to have more examples in each node.



(a) Communication efficiency (objective function)



(b) Time efficiency (objective function)



(c) Time efficiency (AUPRC)

Figure 12: Plots comparing various methods on the *Splice site recognition* dataset.

Second, the theory proposed in Section 3 holds for feature partitioning also. Suppose, in each node p we restrict ourselves to a subset of features, $J_p \subset \{1, \ldots, d\}$, i.e., include the constraint, $w_p \in \{w : w(j) = w^r(j) \;\; \forall r \notin J_p\}$, where $w(j)$ denotes the weight of the j^{th} feature. Note that we do not need $\{J_p\}$ to form a partition. This is useful since important features can be included in all the nodes.

Gradient sub-consistency. Given w^r and J_p we say that $\hat{f}_p(w)$ has gradient sub-consistency with f at w^r on J_p if $\frac{\partial f}{\partial w(j)}(w^r) = \frac{\partial \hat{f}}{\partial w(j)}(w^r) \;\; \forall \, j \in J_p$.

Under the above condition, we can modify the algorithm proposed in Section 3 to come up with a feature decomposition algorithm with *glrc*.

Several feature decomposition based approaches (Richtárik and Takác, 2012; Patriksson, 1998b) have been proposed in the literature. The one closest to our method is the work by Patrikkson on a synchronized parallel algorithm (Patriksson, 1998b) which extends a generic cost approximation algorithm (Patriksson, 1998a) that is similar to our functional approximation. The sub-problems on the partitions are solved in parallel. Although the objective function is not assumed to be convex, the cost approximation is required to satisfy

a monotone property, implying that the approximation is convex. The algorithm only has asymptotic linear rate of convergence and it requires the feature partitions to be disjoint. In contrast, our method has *glrc* and works even if features overlap in partitions. Moreover, there does not exist any counterpart of our example partitioning based distributed algorithm discussed in Section 3.

Recently Mairal (2013) has developed an algorithm called MISO. The main idea of MISO (which is in the spirit of the EM algorithm) is to build majorization approximations with good properties so that line search can be avoided, which is interesting. MISO is a serial method. Developing a distributed version of MISO is an interesting future direction; but, given that line search is inexpensive communication-wise, it is unclear if such a method would give great benefits.

Our approach can be easily generalized to joint example-feature partitioning as well as non-convex settings.[18] The exact details of all the extensions mentioned above and related experiments are left for future work.

Recently, a powerful divide and conquer approach Hsieh et al. (2014) has been suggested for training kernel methods. The idea is to partition the input space such that the restrictions of training on the partitioned input spaces are as decoupled as possible. If, in FADL, we had the ability to choose the parts of data that are placed in the nodes, then we would also gain by choosing decoupled partitions. However, in the distributed case, this requires pre-processing as well as shuffling of data, which are expensive.

6. Conclusion

To conclude, we have proposed FADL, a novel functional approximation based distributed algorithm with provable global linear rate of convergence. The algorithm is general and flexible in the sense of allowing different local approximations at the node level, different algorithms for optimizing the local approximation, early stopping and general data usage in the nodes. We also established the superior efficiency of FADL by evaluating it against key existing distributed methods. We believe that FADL has great potential for solving machine learning problems arising in Big data.

Appendix A: Complexity analysis

Let us use the notations of section 3 given around (21). We define the overall cost of any distributed algorithm as

$$[(c_1 \frac{nz}{P} + c_2 m) T^{\text{inner}} + c_3 \gamma m] T^{\text{outer}}, \tag{22}$$

where T^{outer} is the number of outer iterations, T^{inner} is the number of inner iterations at each node before communication happens and c_1 and c_2 denote the number of passes over the data and m-dimensional dot products per inner iteration respectively. For communication, we assume an *AllReduce* binary tree as described in Agarwal et al. (2011) with

18. For non-convex settings *glrc* is hard to establish, but proving a simpler convergence theory is quite possible.

pipelining. As a result, we do not have a multiplicative factor of $log_2 P$ in our cost[19]. γ is the relative computation to communication speed in the given distributed system; more precisely, it is the ratio of the times associated with communicating a floating point number and performing one floating point operation; γ is usually much larger than 1. c_3 is the number of m-dimensional vectors (gradients, Hessian-vector computations etc.) we need to communicate.

Method	c_1	c_2	c_3	T^{inner}
SQM	2	$\approx 5-10$	1	1
FADL	2	$\approx 5-7$	2	$\hat{k}$

Table 3: Value of cost parameters

The values of different parameters for SQM and FADL are given in Table 3. $T^{\text{outer}}_{\text{SQM}}$ is the number of overall conjugate gradient iterations plus gradient computations. $\hat{k}$ is the average number of conjugate gradient iterations (for the inner minimization of $\hat{f}_p$ using TRON) required per outer iteration in FADL. Typically $\hat{k}$ is between 5 and 20.

Since dense dot products are extremely fast $c_2 m$ is small compared to $c_1 nz/P$ for both the approaches, we ignore it from (22) for simplicity. Now for FADL to have lesser cost than TERA, we can use (22) to get the condition,

$$2.0(\hat{k}T^{\text{outer}}_{\text{FADL}} - T^{\text{outer}}_{\text{SQM}})\frac{nz}{P} \leq (T^{\text{outer}}_{\text{SQM}} - 2T^{\text{outer}}_{\text{FADL}})\gamma m \tag{23}$$

Let us ignore $T^{\text{outer}}_{\text{SQM}}$ on the left side of this inequality (in favor of SQM) and rearrange to get the looser condition,

$$\frac{nz}{m} \leq \frac{\gamma P}{\hat{k}}\frac{1}{2.0}(\frac{T^{\text{outer}}_{\text{SQM}}}{T^{\text{outer}}_{\text{FADL}}} - 2) \tag{24}$$

Assuming $T^{\text{outer}}_{\text{SQM}} > 3.0 T^{\text{outer}}_{\text{FADL}}$, we arrive at the final condition in (21).

Appendix B: Proofs

Proofs of the results in section 2

Let us now consider the establishment of the convergence theory given in section 2.

Proof of Lemma 1. Let $\rho(t) = f(w^r + td^r)$ and $\gamma(t) = \rho(t) - \rho(0) - \alpha t\rho'(0)$. Note the following connections with quantities involved in Lemma 1: $\rho(t) = f^{r+1}$, $\rho(0) = f^r$, $\rho'(t) = g^{r+1} \cdot d^r$ and $\gamma(t) = f^{r+1} - f^r - \alpha g^r \cdot (w^{r+1} - w^r)$. (4) corresponds to the condition $\gamma(t) \leq 0$ and (5) corresponds to the condition $\rho'(t) \geq \beta\rho'(0)$.

$\gamma'(t) = \rho'(t) - \alpha\rho'(0)$. $\rho'(0) < 0$. ρ' is strictly monotone increasing because, by assumption A2,

$$\rho'(t) - \rho'(\tilde{t}) \geq \sigma(t - \tilde{t})\|d^r\|^2 \;\; \forall\, t, \tilde{t} \tag{25}$$

This implies that γ' is also strictly monotone increasing and, all four, ρ, ρ', γ' and γ tend to infinity as t tends to infinity.

19. Actually, there is another communication term, $\gamma b\, log_2 P$, where b is the size of first block of communicated doubles in the pipeline. However, typically $b \ll m$ and hence we ignore it.

Let t_β be the point at which $\rho'(t) = \beta\rho'(0)$. Since $\rho'(0) < 0$ and ρ' is strictly monotone increasing, t_β is unique and $t_\beta > 0$. This validates the definition in (6). Monotonicity of ρ' implies that (5) is satisfied iff $t \geq t_\beta$.

Note that $\gamma(0) = 0$ and $\gamma'(0) < 0$. Also, since γ' is monotone increasing and $\gamma(t) \to \infty$ as $t \to \infty$, there exists a unique $t_\alpha > 0$ such that $\gamma(t_\alpha) = 0$, which validates the definition in (7). It is easily checked that $\gamma(t) \leq 0$ iff $t \in [0, t_\alpha]$.

The properties also imply $\gamma'(t_\alpha) > 0$, which means $\rho'(t_\alpha) \geq \alpha\rho'(0)$. By the monotonicity of ρ' we get $t_\alpha > t_\beta$, proving the lemma.

Proof of Theorem 2. Using (5) and A1,

$$(\beta - 1)g^r \cdot d^r \leq (g^{r+1} - g^r) \cdot d^r \leq Lt\|d^r\|^2 \tag{26}$$

This gives a lower bound on t:

$$t \geq \frac{(1-\beta)}{L\|d^r\|^2}(-g^r \cdot d^r) \tag{27}$$

Using (4), (27) and (1) we get

$$f^{r+1} \leq f^r + \alpha t g^r \cdot d^r \leq f^r - \frac{\alpha(1-\beta)}{L\|d^r\|^2}(-g^r \cdot d^r)^2 \leq f^r - \frac{\alpha(1-\beta)}{L}\cos^2\theta\|g^r\|^2 \tag{28}$$

Subtracting $f^\star$ gives

$$(f^{r+1} - f^\star) \leq (f^r - f^\star) - \frac{\alpha(1-\beta)}{L}\cos^2\theta\|g^r\|^2 \tag{29}$$

A2 together with $g(w^\star) = 0$ implies $\|g^r\|^2 \geq \sigma^2\|w^r - w^\star\|^2$. Also A1 implies $f^r - f^\star \leq \frac{L}{2}\|w^r - w^\star\|^2$ Smola and Vishwanathan (2008). Using these in (29) gives

$$\begin{aligned}(f^{r+1} - f^\star) &\leq (f^r - f^\star) - 2\alpha(1-\beta)\frac{\sigma^2}{L^2}\cos^2\theta(f^r - f^\star) \\ &\leq (1 - 2\alpha(1-\beta)\frac{\sigma^2}{L^2}\cos^2\theta)(f^r - f^\star)\end{aligned} \tag{30}$$

Let $\delta = (1 - 2\alpha(1-\beta)\frac{\sigma^2}{L^2}\cos^2\theta)$. Clearly $0 < \delta < 1$. Theorem 2 follows.

Proofs of the results in section 3

Let us now consider the establishment of the convergence theory given in section 3. We begin by establishing that the exact minimizer of $\hat{f}_p$ makes a sufficient angle of descent at w^r.

Lemma 5. Let $\hat{w}_p^\star$ be the minimizer of $\hat{f}_p$. Let $d_p = (\hat{w}_p^\star - w^r)$. Then

$$-g^r \cdot d_p \geq (\sigma/L)\|g^r\|\|d_p\| \tag{31}$$

Proof. First note, using gradient consistency and $\nabla f_p(\hat{w}_p^\star) = 0$ that

$$\|g^r\| = \|\nabla\hat{f}_p(w^r) - \nabla\hat{f}_p(\hat{w}_p^\star)\| \leq L\|d_p\| \tag{32}$$

Now,

$$-g^r \cdot d_p = (\nabla \hat{f}_p(w^r) - \nabla \hat{f}_p(\hat{w}_p^\star))^T (w^r - \hat{w}_p^\star) \geq \sigma \|d_p\|^2 = \quad \sigma \|g^r\| \|d_p\| \frac{\|d_p\|}{\|g^r\|} \geq \frac{\sigma}{L} \|g^r\| \|d_p\| \quad (33)$$

where the second line comes from σ-strong convexity and the fourth line follows from (32).

Proof of Lemma 3. Let us now turn to the question of approximate stopping and establish Lemma 3. Given θ satisfying (18) let us choose $\zeta \in (0, 1)$ such that

$$\frac{\pi}{2} > \theta > \cos^{-1} \frac{\sigma}{L} + \cos^{-1} \zeta \quad (34)$$

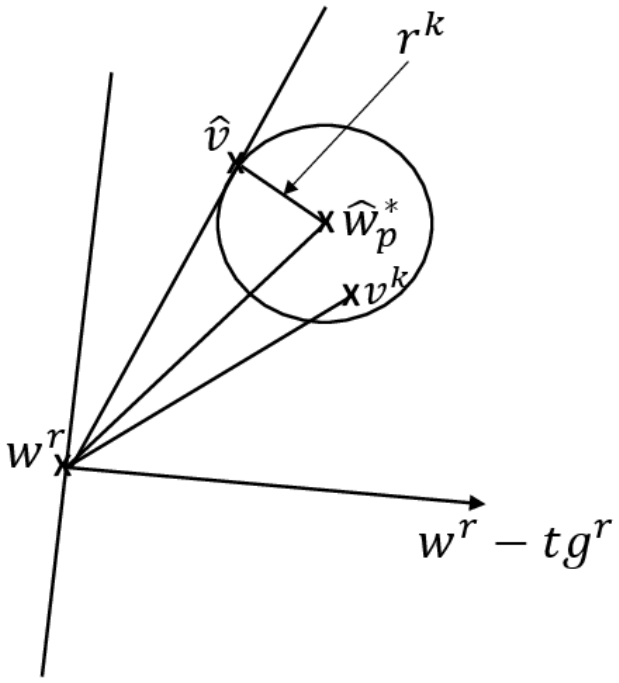


Figure 13: Construction used in the proof of Lemma 3.

By A3 and equations (3.16) and (3.22) in Smola and Vishwanathan (2008), we get

$$\frac{\sigma}{2} \|v - \hat{w}_p^\star\|^2 \leq \hat{f}_p(v) - \hat{f}_p^\star \leq \frac{L}{2} \|v - \hat{w}_p^\star\|^2 \quad (35)$$

After k iterations we have

$$\hat{f}_p(v^k) - \hat{f}_p^\star \leq \delta^k (\hat{f}_p(w^r) - \hat{f}_p^\star) \quad (36)$$

We can use these to get

$$\|v^k - \hat{w}_p^\star\|^2 \leq \frac{2(\hat{f}_p(v^k) - \hat{f}_p^\star)}{\sigma} \leq \frac{2\delta^k (\hat{f}_p(w^r) - \hat{f}_p^\star)}{\sigma} \leq \frac{\delta^k L}{\sigma} \|w^r - \hat{w}_p^\star\|^2 \stackrel{\text{def}}{=} (r^k)^2 \quad (37)$$

For now let us assume the following:

$$\|v^k - \hat{w}_p^\star\|^2 \leq \|w^r - \hat{w}_p^\star\|^2 \tag{38}$$

Using (37) note that (38) holds if

$$\frac{\delta^k L}{\sigma} \leq 1 \tag{39}$$

Let S^k be the sphere, $S^k = \{v : \|v - \hat{w}_p^\star\|^2 \leq (r^k)^2\}$. By (37) we have $v^k \in S^k$. See Figure 6. Therefore,

$$\phi^k \leq \max_{v \in S^k} \phi(v) \tag{40}$$

where ϕ^k is the angle between $\hat{w}_p^\star - w^r$ and $v^k - w^r$, and $\phi(v)$ is the angle between $v - w^r$ and $\hat{w}_p^\star - w^r$. Given the simple geometry, it is easy to see that $\max_{v \in S^k} \phi(v)$ is attained by a point $\hat{v}$ lying on the boundary of S^k (i.e., $\|\hat{v} - \hat{w}_p^\star\|^2 = (r^k)^2$) and satisfying $(\hat{v} - \hat{w}_p^\star) \perp (\hat{v} - w^r)$. This geometry yields

$$\cos^2 \phi(\hat{v}) = \frac{\|\hat{v} - w^r\|^2}{\|\hat{w}_p^\star - w^r\|^2} = \frac{\|\hat{w}_p^\star - w^r\|^2 - (r^k)^2}{\|\hat{w}_p^\star - w^r\|^2} = 1 - \frac{(r^k)^2}{\|\hat{w}_p^\star - w^r\|^2} = 1 - \frac{\delta^k L}{\sigma} \tag{41}$$

Since $\phi^k \leq \phi(\hat{v})$,

$$\cos^2 \phi^k \geq 1 - \frac{\delta^k L}{\sigma} \tag{42}$$

Thus, if

$$1 - \frac{\delta^k L}{\sigma} \geq \zeta^2 \tag{43}$$

then

$$\cos \phi^k \geq \zeta \;\; \forall k \geq \hat{k} \tag{44}$$

holds. By (34) this yields $\angle(-g^r, v^k - w^r) \leq \theta$, the result needed in Lemma 3. Since $\zeta > 0$, (43) implies (39), so (38) holds and there is no need to separately satisfy it. Now (43) holds if

$$k \geq \hat{k} \stackrel{\text{def}}{=} \frac{\log(L/(\sigma(1-\zeta^2)))}{\log(1/\delta)} \tag{45}$$

which proves the lemma.

Proof of Theorem 4. It trivially follows from a combination of Lemma 3 and Theorem 2.

References

A. Agarwal, O. Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear learning system. In *arXiv:1140.4198*, 2011.

A. Agarwal, O Chapelle, M. Dudik, and J. Langford. A reliable effective terascale linear system. *arXiv:1110.4198*, 2013.

D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and distributed computation: Numerical methods*. Athena Scientific, Cambridge, MA, 1997.

L. Bottou. Large-scale machine learning with stochastic gradient descent. In *COMPSTAT'2010*, pages 177–187, 2010.

S. Boyd and L. Vandenberghe. *Convex optimization*. Cambridge University Press, Cambridge, UK, 2004.

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

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 of Optimization*, pages 977–995, 2012.

K.W. Chang, C.J. Hsieh, and C.J. Lin. Coordinate descent method for large-scale l2-loss linear SVM. *JMLR*, pages 1369–1398, 2008.

C.T. Chu, S.K. Kim, Y.A. Lin, Y.Y. Yu, G. Bradski, A.Y. Ng, and K. Olukotun. Map-reduce for machine learning on multicore. *NIPS*, pages 281–288, 2006.

J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, pages 107–113, 2008.

W. Deng and W. Yin. On the global linear convergence of the generalized alternating direction method of multipliers. *Rice University CAAM Technical Report*, TR12-14, 2012.

K.B. Hall, S. Gilpin, and G. Mann. Mapreduce/bigtable for distributed optimization. In *NIPS Workshop on Leaning on Cores, Clusters, and Clouds*, 2010.

C.J. Hsieh, K.W. Chang, C.J. Lin, S.S. Keerthi, and S. Sundararajan. A dual coordinate descent method for large-scale linear SVM. In *ICML*, pages 408–415, 2008.

C.J. Hsieh, S. Si, and I.S. Dhillon. A divide-and-conquer solver for kernel support vector machines. *ICML*, 2014.

M. Jaggi, V. Smith, M. Takáč, J. Terhorst, S. Krishnan, T. Hofmann, and M.I. Jordan. Communication-efficient distributed dual coordinate ascent. *arXiv:1409.1458*, 2014.

R. Johnson and T. Zhang. Accelerating stochastic gradient descent using predictive variance reduction. *NIPS*, 2013.

C.J. Lin, R.C. Weng, and S.S. Keerthi. Trust region newton method for large-scale logistic regression. *JMLR*, pages 627–650, 2008.

D. Mahajan, S. S. Keerthi, S. Sundararajan, and L. Bottou. A functional approximation based distributed learning algorithm. *arXiv:1310.8418*, 2013a.

D. Mahajan, S. S. Keerthi, S. Sundararajan, and L. Bottou. A parallel SGD method with strong convergence. *NIPS Workshop on Optimization in Machine Learning*, 2013b.

J. Mairal. Optimization with first order surrogate functions. *ICML*, 2013.

G. Mann, R.T. McDonald, M. Mohri, N. Silberman, and D. Walker. Efficient large-scale distributed training of conditional maximum entropy models. In *NIPS*, pages 1231–1239, 2009.

R.T. McDonald, K. Hall, and G. Mann. Distributed training strategies for the structured perceptron. In *HLT-NAACL*, pages 456–464, 2010.

M. Patriksson. Cost approximation: A unified framework of descent algorithms for nonlinear programs. *SIAM J. on Optimization*, 8:561–582, 1998a.

M. Patriksson. Decomposition methods for differentiable optimization problems over cartesian product sets. *Comput. Optim. Appl.*, 9:5–42, 1998b.

D. Pechyony, L. Shen, and R. Jones. Solving large scale linear SVM with distributed block minimization. *NIPS workshop on Big Learning*, 2011.

P. Richtárik and M. Takác. Parallel coordinate descent methods for big data optimization. *CoRR*, abs/1212.0873, 2012.

O. Sharir, N. Srebro, and T. Zhang. Communication efficient distributed optimization using an approximate newton-type method. *arXiv:1312.7853v4*, 2014.

J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, 1994.

A. Smola and S.V.N. Vishwanathan. *Introduction to Machine Learning*. Cambridge University Press, Cambridge, UK, 2008.

S. T. Sonnenburg and V. Franc. COFFIN: a computational framework for linear SVMs. *ICML*, 2010.

P.W. Wang and C.J. Lin. Iteration complexity of feasible descent methods for convex optimization. *Technical Report, National Taiwan University*, 2013.

M. Weimer, Y. Chen, B.G. Chun, T. Condie, C. Curino, C. Douglas, Y. Lee, T. Majestro, D. Malkhi, S. Matusevych, et al. Reef: Retainable evaluator execution framework. In *ACM SIGMOD*, pages 1343–1355, 2015.

P. Wolfe. Convergence conditions for ascent methods. *SIAM Review*, 11:226–235, 1969.

P. Wolfe. Convergence conditions for ascent methods: II: Some corrections. *SIAM Review*, 13:185–188, 1971.

T. Yang. Trading computation for communication: distributed stochastic dual coordinate ascent. *NIPS*, 2013.

T. Yang, S. Zhu, R. Jin, and Y. Lin. Analysis of distributed stochastic dual coordinate ascent. *arXiv:1312.1031*, 2013.

M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *USENIX Conference*, 2010.

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

Y. Zhang and L. Xiao. DiSCO: Communication-efficient distributed optimization of self-concordant loss. *ICML*, 2015.

M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.