Supplementary Materials for ”Fast Matrix-vector Multiplications for Large-scale Logistic Regression on Shared-memory Systems”

Mu-Chu Lee
Department of Computer Science
National Taiwan University
Taipei, Taiwan
Email: b01902082@ntu.edu.tw

Wei-Lin Chiang
Department of Computer Science
National Taiwan University
Taipei, Taiwan
Email: b02902056@ntu.edu.tw

Chih-Jen Lin
Department of Computer Science
National Taiwan University
Taipei, Taiwan
Email: cjlin@csie.ntu.edu.tw

1. Matrix-vector Multiplications in Newton Methods for Logistic Regression

We briefly introduce Newton methods for logistic regression and then explain that matrix-vector Multiplications are the computational bottleneck.

A. Newton and Truncated Newton Methods

To minimize a twice-differentiable function $f(\boldsymbol{w})$ in (1), at the current iterate $\boldsymbol{w}^k$, a Newton method minimizes the following second-order Taylor expansion of the function-value reduction $f(\boldsymbol{w}^k+\boldsymbol{s})-f(\boldsymbol{w}^k)$ to obtain a direction $\boldsymbol{s}^k$.

$$\min_{\boldsymbol{s}} q_k(\boldsymbol{s}), \text{ where } q_k(\boldsymbol{s}) \equiv \nabla f(\boldsymbol{w}^k)^T\boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T\nabla^2 f(\boldsymbol{w}^k)\boldsymbol{s}. \tag{A.1}$$

For logistic regression, we can easily derive the following gradient and Hessian of $f(\boldsymbol{w})$:

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + C\sum_{i=1}^{l}(\sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i)-1)y_i\boldsymbol{x}_i, \tag{A.2}$$

$$\nabla^2 f(\boldsymbol{w}) = \mathcal{I} + CX^TDX, \tag{A.3}$$

where $\mathcal{I}$ is the identity matrix,

$$\sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i) = (1+e^{-y_i\boldsymbol{w}^T\boldsymbol{x}_i})^{-1},$$

and D is a diagonal matrix with

$$D_{ii} = \sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i)(1-\sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i)).$$

We can see that $X\boldsymbol{w}$ obtained in function value evaluation can be continually used for $\sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i), \forall i$ in (A.2) for gradient evaluation. In (A.2) we need another matrix-vector multiplication

$$X^T\boldsymbol{v}, \text{ where } v_i = (\sigma(y_i\boldsymbol{w}^T\boldsymbol{x}_i)-1)y_i.$$

For obtaining the direction $\boldsymbol{s}^k$, because $\nabla^2 f(\boldsymbol{w}^k)$ is positive definite, the minimization problem in (A.1) is equivalent to solving the following Newton linear system.

$$\nabla^2 f(\boldsymbol{w}^k)\boldsymbol{s}^k = -\nabla f(\boldsymbol{w}^k). \tag{A.4}$$

For data sets with many features, $\nabla^2 f(\boldsymbol{w}^k)$ is a huge and dense matrix that is too large to be stored. Machine learning researchers [1], [2], [3] have applied iterative methods such as Conjugate Gradient (CG) to solve (A.4) without explicitly forming the Hessian. A CG procedure includes some iterations, where each one has the following main computational task

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{d} = \boldsymbol{d} + C\cdot X^T(D(X\boldsymbol{d})). \tag{A.5}$$

Algorithm 1 CG for approximately solving (A.4)

```
1) Let $\boldsymbol{s}=\boldsymbol{0}, \boldsymbol{r}=-\nabla f(\boldsymbol{w}^k)$, $\boldsymbol{d}=\boldsymbol{r}$, and $r_{\text{sq}}=\|\boldsymbol{r}\|^2$.
2) For $i=0,1,\ldots$
    • If some stopping conditions hold, stop and output $\boldsymbol{s}$ as $\boldsymbol{s}^k$.
    • Calculate $\boldsymbol{v}=\nabla^2 f(\boldsymbol{w}^k)\boldsymbol{d}$ by (A.5)
    • $\alpha \leftarrow r_{\text{sq}}/\boldsymbol{d}^T\boldsymbol{v}$
    • $\boldsymbol{s}\leftarrow\boldsymbol{s}+\alpha\boldsymbol{d}$ and $\boldsymbol{r}\leftarrow\boldsymbol{r}-\alpha\boldsymbol{v}$.
    • $r_{\text{sq}}^{\text{new}}\leftarrow\|\boldsymbol{r}\|^2$
    • $\beta\leftarrow r_{\text{sq}}^{\text{new}}/r_{\text{sq}}$ and $r_{\text{sq}}\leftarrow r_{\text{sq}}^{\text{new}}$
    • $\boldsymbol{d}\leftarrow\boldsymbol{r}+\beta\boldsymbol{d}$.
```

Here $\boldsymbol{d}$ is an intermediate vector in the CG procedure. Note that (A.5) is performed by using only X and D without forming the Hessian X^TDX. This Hessian-free strategy effectively alleviates the memory difficulty. For completeness, we show the CG procedure in Algorithm 1, where details can be seen in, for example, [4]. In Algorithm 1, we can clearly see that except (A.5), all others are cheaper vector operations.

By using the CG method, a Newton method has two layers of iterations: at each outer iteration an inner CG procedure finds the Newton direction. To save the computational cost, in practice we only approximately solve (A.5), so the CG procedure stops after satisfying some stopping conditions. Therefore, we have a “truncated” Newton method in optimization because of not using the full Newton direction of exactly solving (A.5).

The obtained direction $\boldsymbol{s}^k$ cannot be directly used to update $\boldsymbol{w}^k$ because we may not have the decrease of the function value; that is, $f(\boldsymbol{w}^k+\boldsymbol{s}^k) < f(\boldsymbol{w}^k)$ may not hold. The decreasing property is required in proving the convergence.

Two major optimization techniques to obtain this property are line search and trust-region methods. We investigate their needed operations.

B. Line Search Procedure

A line search procedure finds a step size α_k such that the following sufficient decrease condition holds.

$$f(\boldsymbol{w}^k + \alpha_k \boldsymbol{s}^k) \leq f(\boldsymbol{w}^k) + \eta \alpha_k \nabla f(\boldsymbol{w}^k)^T \boldsymbol{s}^k,$$

where $\eta \in (0,1)$ is a pre-specified constant. Usually α_k is obtained by a backtrack search of trying

$$\alpha_k = 1, \beta, \beta^2, \ldots,$$

where $\beta \in (0,1)$. Although several evaluations of $f(\boldsymbol{w}^k + \alpha \boldsymbol{s}^k)$ are conducted, we need only one matrix-vector multiplication $X\boldsymbol{s}^k$ and have

$$X(\boldsymbol{w}^k + \alpha \boldsymbol{s}^k) = X\boldsymbol{w}^k + \alpha X \boldsymbol{s}^k.$$

Proofs of convergence for truncated Newton methods using line search can be found in, for example, [5].

C. Trust Region Methods

This approach restricts the truncated Newton direction s^k to be within a "trust region" of size Δ_k.

$$\min_{\boldsymbol{s}} \; q_k(\boldsymbol{s}) \quad \text{subject to} \|\boldsymbol{s}\| \leq \Delta_k.$$

The step $\boldsymbol{s}^k$ is taken if the function value at $\boldsymbol{w}^k + \boldsymbol{s}^k$ is sufficiently decreased. The calculation involves one matrix-vector multiplication $X(\boldsymbol{w}^k + \boldsymbol{s}^k)$. However, if $\boldsymbol{s}^k$ is accepted for update, then $X(\boldsymbol{w}^k + \boldsymbol{s}^k)$ can be passed to the next iteration for use.

In our implementation, we follow the setting in [6] to update the size Δ_k of the trust region. Because of the constraint $\|\boldsymbol{s}\| \leq \Delta_k$, the CG procedure must be modified to take care of it. See details in Algorithm 2 of [3]. The convergence of trust region Newton methods using CG procedures has been established in [7] and subsequent works.

D. Computational Cost

From the above discussion, we see that function evaluations, gradient evaluations, and CG iterations all involve matrix-vector multiplications. Because each outer iteration involves one function/gradient evaluation but several CG iterations, the cost of a Newton method is almost proportional to the total number of CG iterations (i.e., the total number of Hessian-vector multiplications).

2. Detailed Analysis on Implementation using OpenMP

An OpenMP loop must assign tasks to different threads. For example, the default `schedule(static)` splits indices to P blocks in a fixed way, where each contains l/P elements. However, because tasks may be unbalanced, we can have a dynamic scheduling so that available threads are assigned to the next needed tasks. A chunk size decides the number of indices handled by a thread at a time. For example, `schedule(dynamic,256)` implies that a thread works on a chunk of 256 elements each time. However, overheads occur for doing the dynamic task assignment.

Deciding what kind of scheduling to use is not trivial. We check the implementation of (3) as an example. This operation involves the following three loops.

1) Initializing $\hat{\boldsymbol{u}}^p = \boldsymbol{0}, \forall p = 1, \ldots, P$.
2) Calculating $\hat{\boldsymbol{u}}^p, \forall p$ by

$$\hat{\boldsymbol{u}}^p = \sum \{\boldsymbol{u}_i \boldsymbol{x}_i \mid i \text{ run by thread } p\}$$

3) Calculating $\bar{\boldsymbol{u}} = \sum_{p=1}^{P} \hat{\boldsymbol{u}}^p$.

At first glance we would think that the first and the third loops are not important because they involve only $O(Pl)$ operations. The second loop needs $O(\text{nnz})$, where nnz is the total number of non-zero entries in X. In general

$$\text{nnz} \gg Pl,$$

because our number of cores is no more than 20, but the average number of non-zero features per instance is easily hundreds or more. However, we show that a casual implementation may cause the first or the third loop equally time consuming. For simplicity, we check only the third loop because the first one is similar. The summation $\sum_{p=1}^{P} \hat{\boldsymbol{u}}^p$ is indeed similar to $X^T \boldsymbol{u}$, where the output is shared. If $n \gg P$, we can sequentially conduct vector additions but parallelize each vector addition:

```
1: for $p = 1, \ldots, P$ do
2:     for $i = 1, \ldots, n$ do in parallel
3:         $u_i \leftarrow u_i + \hat{u}_i^p$
```

We call this a core-oriented approach. A concern is that when n is very small, we do not achieve parallelism. Instead we can consider:

```
1: for $i = 1, \ldots, n$ do in parallel
2:     $u_i = \sum_{p=1}^{P} \hat{u}_i^p$
```

We call this a feature-oriented approach. A disadvantage is that $\hat{\boldsymbol{u}}^p$ is not continuously accessed.

Now for the second loop and the two approaches for the third loop, we try the following three settings,

1) `schedule(static)`
2) `schedule(dynamic)`
3) `schedule(dynamic,256)`

and present the running time in Table I. We consider one problem (covtype_binary) with small n and one problem (rcv1_binary) with large n. Results show that not only does scheduling significantly affect the running time, but also it is data dependent. For the second loop by `schedule(static)`, the running time is short for covtype_binary, but is unacceptable for rcv1_binary. For the third loop (core-oriented approach) with `schedule(dynamic)`, where the default chunk size is one, the running time is huge for rcv1_binary. Apparently this is because of the overheads of doing n assignments. This experiment fully illustrates the importance of having appropriate settings. In experiments in Section III-E, the OpenMP implementation considers `schedule(dynamic,256)` for

TABLE I
RUNNING TIME(IN SECONDS) WITH DIFFERENT SCHEDULER. EIGHT THREADS ARE USED.

Second loop	covtype_binary	rcv1_binary
`schedule(static)`	0.2879	2.9387
`schedule(dynamic)`	1.2611	2.6084
`schedule(dynamic, 256)`	0.2558	1.6505
Third loop (core-oriented)	**covtype_binary**	**rcv1_binary**
`schedule(static)`	0.0008	0.0056
`schedule(dynamic)`	0.0017	0.5094
`schedule(dynamic, 256)`	0.0010	0.0062
Third loop (feature-oriented)	**covtype_binary**	**rcv1_binary**
`schedule(static)`	0.0005	0.0062
`schedule(dynamic)`	0.0007	0.0910
`schedule(dynamic, 256)`	0.0005	0.0039

the second loop, while the feature-oriented approach with `schedule(static)` for the third loop.

A. Relations with Asynchronous Coordinate Descent Methods

Very recently, asynchronous coordinate descent methods [8] have been proposed for efficiently solving the dual problem of linear classifiers on shared-memory systems. While a dual-based method looks very different from the Newton method here that solves the primal problem, surprisingly their operations are very related. In this section we discuss some interesting connections.

If $l1$ loss rather than logistic loss is used, problem becomes linear SVM.

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum\nolimits_{i=1}^{l} \max(0, 1 - y_i\boldsymbol{w}^T\boldsymbol{x}_i). \qquad \text{(A.6)}$$

The dual problem takes the following form.

$$\min_{\boldsymbol{\alpha}} \quad f(\boldsymbol{\alpha}) \quad \text{subject to } 0 \le \alpha_i \le C, \forall i, \qquad \text{(A.7)}$$

where

$$f(\boldsymbol{\alpha}) = \frac{1}{2}\sum\nolimits_{i=1}^{l}\sum\nolimits_{j=1}^{l} \boldsymbol{x}_i^T\boldsymbol{x}_j\alpha_i\alpha_j - \sum\nolimits_{i=1}^{l}\alpha_i.$$

The dual CD method to solve (A.7) is the following procedure.

```
1: while $\boldsymbol{w}$ is not optimal do
2:     for $i = 1, \dots, l$ do
3:         $r = \min(\max(\alpha_i - \frac{\nabla_i f(\boldsymbol{\alpha})}{\|\boldsymbol{x}_i\|^2}, 0), C) - \alpha_i$
4:         $\alpha_i \leftarrow \alpha_i + r$
5:         $\boldsymbol{w} \leftarrow \boldsymbol{w} + r\boldsymbol{x}_i$
```

By initializing $\boldsymbol{w} = \boldsymbol{0}$, we maintain the relationship

$$\boldsymbol{w} = \sum\nolimits_{j=1}^{l} \alpha_j\boldsymbol{x}_j,$$

and $\boldsymbol{w}$ will eventually converge to the optimal solution of the primal problem (A.6). Then

$$\nabla_i f(\boldsymbol{\alpha}) = \sum\nolimits_{j=1}^{l} \alpha_j\boldsymbol{x}_j^T\boldsymbol{x}_i - 1 = \boldsymbol{x}_i^T\boldsymbol{w} - 1.$$

A comparison with the **for** loop in Section II-C shows that, in the **for** loop here for updating $\alpha_1, \dots, \alpha_l$, the number of operations is about the same as that of the two matrix-vector operations:

$$X\boldsymbol{d} \text{ and } X^T\boldsymbol{u}, \text{ where } \boldsymbol{u} = DX\boldsymbol{d}.$$

The work [8] parallelizes the **for** loop by

```
1: for $i = 1, \dots, l$ do in parallel
2:     $r = \min(\max(\alpha_i - \frac{\boldsymbol{w}^T\boldsymbol{x}_i - 1}{\|\boldsymbol{x}_i\|^2}, 0), C) - \alpha_i$
3:     $\alpha_i \leftarrow \alpha_i + r$
4:     for $(x_i)_s \neq 0$ do
5:         atomic: $w_s \leftarrow w_s + r(x_i)_s$
```

We pointed out in Section III-C that the atomic operations cause serious difficulties in the parallel computation of $X^T\boldsymbol{u}$. However, [8] reports good speedup on some large document data sets such as rcv1_binary used in Section III. Naturally we wonder why atomic operations do not cause difficulties for the dual CD method.

An investigation shows that the reason is because the dual CD method may not need to update $\boldsymbol{w}$ for all $i \in \{1, \dots, l\}$. Specifically, the **for** loop is implemented in the following way.

```
1: for $i = 1, \dots, l$ do in parallel
2:     $r = \min(\max(\alpha_i - \frac{\boldsymbol{w}^T\boldsymbol{x}_i - 1}{\|\boldsymbol{x}_i\|^2}, 0), C) - \alpha_i$
3:     $\alpha_i \leftarrow \alpha_i + r$
4:     if $r \neq 0$ then
5:         for $(x_i)_s \neq 0$ do
6:             atomic: $w_s \leftarrow w_s + r(x_i)_s$
```

This setting is reasonable because if $r = 0$, $\boldsymbol{w}$ remains the same. An important property of the $l1$ loss SVM is that the optimal $\boldsymbol{\alpha}$ has some bounded elements (i.e., $\alpha_i = 0$ or C). These elements may reach the final bounded value in just a few iterations. Their corresponding r is zero, so there is no need to update $\boldsymbol{w}$. The situation is like that only part of $X^T\boldsymbol{u}$ calculation is conducted. More precisely, the two loops in Section II-B have

$$l \text{ of every } 2l \text{ vector operations}$$

involving atomic operations, while the dual CD method has

$$\delta l \text{ of every } (1+\delta)l \text{ vector operations,}$$

where $\delta \in [0, 1)$ depends on the number of bounded components in the solution. Thus, the reason why the asynchronous parallel CD method can achieve good speedup is because atomic operations are a smaller portion of the total computation.

In the dual CD method, if we know $r = 0$ beforehand, than the whole step at index i can be waived and the calculation of $\boldsymbol{w}^T\boldsymbol{x}_i$ can also be saved. In practice, we can conjecture if an α_i will remain the same or not. This technique, referred to as shrinking [9], [10], is effective in some situations. Then a bigger portion of inner products involves atomic operations, so the speedup may become less dramatic. Further, the speedup may be poor if the dual CD method is applied to train logistic regression [11] because $0 < \alpha_i < 1, \forall i$ and $r \neq 0$ in general.

REFERENCES

[1] P. Komarek and A. W. Moore, "Making logistic regression a core data mining tool," Carnegie Mellon University, Tech. Rep., 2005.

[2] S. S. Keerthi and D. DeCoste, "A modified finite Newton method for fast solution of large scale linear SVMs," *JMLR*, vol. 6, pp. 341–361, 2005.

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

[4] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. The Johns Hopkins University Press, 1996.

[5] I. Griva, S. G. Nash, and A. Sofer, *Linear and nonlinear optimization*, 2nd ed. SIAM, 2009.

[6] C.-J. Lin and J. J. Moré, "Newton's method for large-scale bound constrained problems," *SIAM J. Optim.*, vol. 9, pp. 1100–1127, 1999.

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

[8] C.-J. Hsieh, H.-F. Yu, and I. S. Dhillon, "PASSCoDe: Parallel asynchronous stochastic dual coordinate descent," in *ICML*, 2015.

[9] T. Joachims, "Making large-scale SVM learning practical," in *Advances in Kernel Methods - Support Vector Learning*. MIT Press, 1998.

[10] 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*, 2008.

[11] H.-F. Yu, F.-L. Huang, and C.-J. Lin, "Dual coordinate descent methods for logistic regression and maximum entropy models," *MLJ*, vol. 85, pp. 41–75, 2011.