# Scaling Multinomial Logistic Regression via Hybrid Parallelism

Parameswaran Raman
University of California, Santa Cruz
params@ucsc.edu

Sriram Srinivasan
University of California, Santa Cruz
ssriniv9@ucsc.edu

Shin Matsushima
University of Tokyo, Japan
shin_matsushima@mist.i.u-tokyo.ac.jp

Xinhua Zhang
University of Illinios, Chicago
zhangx@uic.edu

Hyokun Yun
Amazon
yunhyoku@amazon.com

S.V.N. Vishwanathan
Amazon
vishy@amazon.com

## ABSTRACT

We study the problem of scaling Multinomial Logistic Regression (MLR) to datasets with very large number of data points in the presence of large number of classes. At a scale where neither data nor the parameters are able to fit on a single machine, we argue that *simultaneous data and model parallelism (Hybrid Parallelism)* is inevitable. The key challenge in achieving such a form of parallelism in MLR is the log-partition function which needs to be computed *across all K classes* per data point, thus making model parallelism non-trivial.

To overcome this problem, we propose a reformulation of the original objective that exploits *double-separability*, an attractive property that naturally leads to hybrid parallelism. Our algorithm (DS-MLR) is *asynchronous and completely de-centralized*, requiring minimal communication across workers while keeping both data and parameter workloads partitioned. Unlike standard data parallel approaches, DS-MLR *avoids bulk-synchronization* by maintaining local normalization terms on each worker and accumulating them incrementally using a token-ring topology.

We demonstrate the versatility of DS-MLR under various scenarios in data and model parallelism, through an empirical study consisting of real-world datasets. In particular, to demonstrate scaling via hybrid parallelism, we created a new benchmark dataset (Reddit-Full) by pre-processing 1.7 billion reddit user comments spanning the period 2007-2015. We used DS-MLR to solve an extreme multi-class classification[1] problem of classifying 211 million data points into their corresponding subreddits. Reddit-Full is a massive data set with data occupying 228 GB and 44 billion parameters occupying 358 GB. To the best of our knowledge, no other existing methods can handle MLR in this setting.

## KEYWORDS

Multinomial Logistic Regression; Stochastic Optimization; Large Scale Machine Learning

[1]Extreme classification is defined as multi-class / multi-label classification in the presence of very large number of examples and classes / labels.


*KDD '19, August 4–8, 2019, Anchorage, AK, USA*

ACM ISBN 978-1-4503-6201-6/19/08…$15.00
https://doi.org/10.1145/3292500.3330837

**ACM Reference Format:**
Parameswaran Raman, Sriram Srinivasan, Shin Matsushima, Xinhua Zhang, Hyokun Yun, and S.V.N. Vishwanathan. 2019. Scaling Multinomial Logistic Regression via Hybrid Parallelism. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3292500.3330837

## 1 INTRODUCTION

In this paper, we focus on *multinomial logistic regression* (MLR), also known as softmax regression which computes the probability of a $D$-dimensional data point $\mathbf{x}_i \in \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N\}$ belonging to a class $k \in \{1, 2, \ldots, K\}$. The model is parameterized by a parameter matrix $W \in \mathbb{R}^{D \times K}$. MLR is a method of choice for several real-world tasks such as Image Classification [20] and Video Recommendation [8]. It also manifests as the final output layer in Feed-Forward Deep Neural Networks [10]. Therefore, it has received significant research attention [11], [26]. We concern ourselves with running MLR in the presence of large number of data points $N$ and large number of classes $K$ - a setting which often requires distributing computation over $P$ machines (viz. workers).

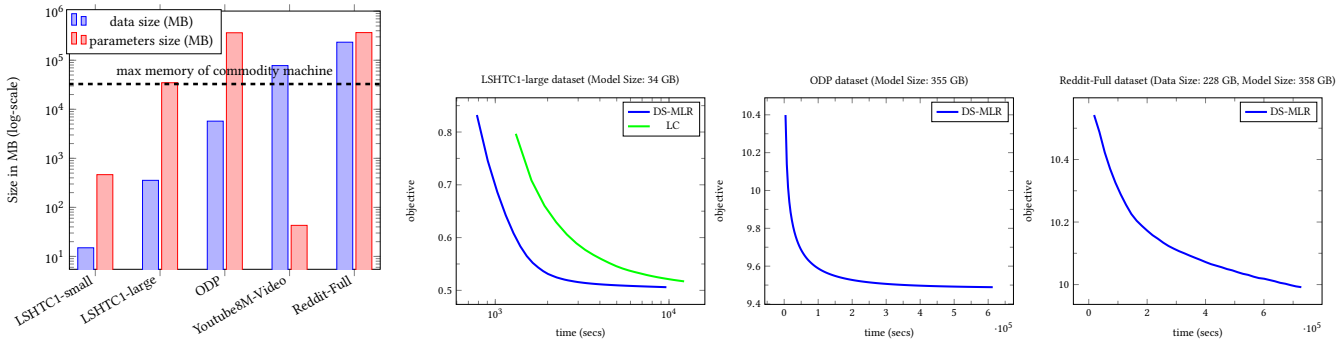
### 1.1 Motivation for Hybrid Parallelism

Traditional methods to perform distributed MLR typically fall into two categories: (a) *data parallel* methods such as L-BFGS [17] which partition the data workload across $P$ workers, however, duplicate the model workload across all workers, and (b) *model parallel* methods such as LC [11], which partition the model workload across $P$ workers, but need to duplicate the data across all of them. This is illustrated in Table 1.

| | Storage per worker | | Communication |
|---|---|---|---|
| | Data | Parameters | |
| L-BFGS | $O(\frac{ND}{P})$ | $O(KD)$ | $O(KD)$ |
| LC | $O(ND)$ | $O(\frac{KD}{P}) + O(N)$ | $O(N)$ |
| **DS-MLR** | $O(\frac{ND}{P})$ | $O(\frac{KD}{P}) + O(\frac{N}{P})$ | $O(\frac{KD}{P})$ |

**Table 1: Memory requirements of various algorithms in MLR ($N$ data points, $D$ features, $K$ classes, $P$ workers).**

The growing acclaim of machine learning is witnessing a surge of novel prediction tasks in diverse domains such as natural language, speech, image and video. These tasks not only involve *humongous amounts of data*, but also are powered by *sophisticated models*, thus




**(a) Data and Model requirements (MB) of real-world datasets for MLR.**

**(b) DS-MLR being a hybrid-parallel method can be run on all datasets with large model storage requirements.**

**Figure 1: Training sophisticated models on large real-world datasets (e.g. Reddit-Full dataset in Figure 1a) can easily cause both the data and parameter sizes (shown as blue and red bars) to well exceed the memory capacity of a commodity cluster machine (shown in dotted black lines). In Figure 1b, we show how this challenge can be met by designing a hybrid-parallel method such as DS-MLR which partitions both data and parameters simultaneously. This enables us to handle data and model workloads of all sizes - in such situations, popular baselines such as L-BFGS [17] (data parallel only) and LC [11] (model parallel only) struggle to run.**

demanding larger storage footprints for the model itself. Such memory requirements typically exceed the capacity of a single machine in a commodity cluster easily. Figure 1a illustrates this fundamental challenge in large-scale machine learning. As seen in the figure, real-world datasets exhibit varying storage requirements for the data and model. While the smaller ones are within the capacity of a single machine, larger datasets such as ODP and Reddit are impossible to run on a commodity cluster with just traditional model parallelism approaches. This is because ODP has a massive requirement of 355 GB for the model itself, while Reddit-Full dataset is even bigger, requiring 228 GB for the data and 358 GB for its model.

**One versatile method (DS-MLR)**: We propose a universal method which acts as a swiss army knife to get the best of both worlds. As a consequence of partitioning both data and model workloads simultaneously (*Hybrid Parallelism*) across $P$ processors, DS-MLR is able to handle data and model parameters of varying sizes, without incurring any storage costs due to duplication. In Figure 1b, we show results of running DS-MLR on three representative datasets which have large model storage requirements: (i) Despite being a modestly-sized dataset, **LSHTC-large** requires 34 GB for its model parameters. As a result, only model-parallel methods such as LC can be run on it. When compared against DS-MLR, we observed that DS-MLR is able to achieve a much faster convergence than LC as seen in the plot (ii) **ODP** is a much larger dataset requiring 355 GB for the model parameters. Even though LC could theoretically be run on it, we observed that it took an enormous amount of time to complete even a single iteration. We believe this is because LC is a second-order method and therefore its per iteration cost is significantly higher than a stochastic method such as DS-MLR (iii) Finally, **Reddit-Full** is a new benchmark dataset pushing the limits of data and model storage. Running MLR on this dataset requires 228 GB of data and 358 GB of model storage. This is a prototypical use-case where a hybrid-parallel method shines over its vanilla data/model parallel counterparts. To the best of our knowledge *no other existing methods* are able to handle such large workloads. Figure 1b shows that DS-MLR is able to handle this scale. However, since each iteration on this massive dataset takes 5 hours (this is excluding the time spent in data loading and initializing the optimizer), we could not keep our experiment running beyond 10 days due to job time limitations on our HPC cluster.

**Cost Analysis of using commodity hardware**: Massive real-world datasets demand high memory, e.g. Reddit-Full consumes 600 GB of total memory (228 GB data, 358 GB model parameters). *When using hybrid parallelism*, one can get away with using cheap commodity hardware. Since the load for RedditFull dataset demands 50,000 compute hours per iteration, using 20 `c5.4xlarge` EC2 instances (32 GB RAM, 16 CPUs, $0.68 / hr per machine) each running 16 threads, one iteration requires 156 hours with a cost of $2,121 per iteration. On the other hand, *if we use data parallelism only (or model parallelism only)*, high-memory instances such as `x1.16xlarge` (900 GB RAM, 64 CPUs, $6.67 per hr per machine) are inevitable. A rough calculation shows that to achieve the same per iteration time, one would have to spend $111,335. This is mainly because, either data or parameters would need to be replicated across each processor, thus making it impossible to use all 64 cores. Even if we make use of a clever data or parameter sharing mechanism to avoid replication, the resulting cost comes down to $5,000 roughly, which is still twice the earlier case.

## 1.2 Our main contributions

**Hybrid Parallel reformulation for MLR**: We present DS-MLR, a novel distributed stochastic optimization algorithm that can partition both data as well as model parameters *simultaneously* (*hybrid parallel*) across its workers. DS-MLR is able to perfectly partition the workload across $P$ workers, costing $O(\frac{ND}{P})$ storage for data and $O(\frac{KD}{P} + \frac{N}{P})$ for the model. As a result, DS-MLR can scale to arbitrarily large datasets where to the best of our knowledge, many of the existing distributed algorithms cannot be applied since they

need to either duplicate $O(KD)$ storage (data parallel methods) or $O(ND)$ storage (model parallel methods) across their workers.

**Asynchronous and De-centralized Implementation**: To deploy DS-MLR on real-world datasets, we develop a *non-blocking* and *asynchronous* variant (DS-MLR Async), which provides further speedups in the multi-core, multi-machine setting by interleaving the computation and communication phases during every iteration. DS-MLR avoids expensive *bulk-synchronization* operations by maintaining local normalization terms on each worker and accumulating them incrementally using a token-ring topology. DS-MLR is implemented in C++ making use of intra-machine parallelism (multi-threading using Intel TBB) as well as inter-machine parallelism (using MPI).

**Large-scale real world experiments**: We present an exhaustive empirical study running DS-MLR on real-world datasets with varying data and model footprints, showing that DS-MLR readily applies in all cases. In particular, to demonstrate applicability of DS-MLR to the scenario where *both data and model do not fit* on a single machine, we created a new benchmark dataset Reddit-Full that has data and model footprints of *228 GB and 358 GB* respectively.

## 2 RELATED WORK

There has been a flurry of work in the past few years on developing distributed optimization algorithms for machine learning. In this section, we characterize some of this related work and put our method DS-MLR in perspective.

**Data Parallelism vs Model Parallelism**: The classic paradigm in distributed machine learning is to perform *data partitioning*, using, for instance, a map reduce style architecture [7] where data is distributed across multiple slaves. In each iteration, the slaves gather the parameter vector from the master, compute gradients locally and transmit them back to the master. The L-BFGS optimization algorithm [17] is typically used in the master to update the parameters after every iteration. *The main drawback of this strategy is that the model parameters need to be replicated on every machine*. For a $D$ dimensional problem involving $K$ classes, this demands $O(K \times D)$ storage. In many cases, this is too large to fit on a single machine. An orthogonal approach is to use *model partitioning*. Here again, a master slave architecture is used, but now, the data is replicated across each slave. The model parameters are partitioned and distributed to each machine. During each iteration, the model parameters on the individual machines are updated, and some auxiliary variables are computed and distributed to the other slaves, which use these variables in their parameter updates. See the Log-Concavity (LC) method [11] for an example of such a strategy. *The main drawback of this approach, however, is that the data needs to be replicated on each machine*, and consequently it is not applicable when the data is too large to fit on a single machine.

**Distributed Stochastic Gradient Methods**: Stochastic gradient descent based approaches have proven to be very fruitful since they make frequent parameter updates and converge much more rapidly [3]. Several algorithms for parallelizing SGD have been proposed in the past such as Hogwild [18], Parallel SGD [32], DSGD [9], FPSGD [31] and more recently, Parameter Server [14] and Petuum [25]. Although the importance of data and model parallelism has been recognized in Parameter Server and the Petuum framework [25], to the best of our knowledge this has not been exploited in their specific instantiations such as applications to multinomial logistic regression [24]. We believe this is because [24] does not reformulate the problem the way DS-MLR does. *Several problems in machine learning are not naturally well-suited for simultaneous data and model parallelism, and therefore such reformulations are essential in uncovering a suitable structure.*

**De-centralized vs Parameter Server Based**: Parameter Server [14] a widely popular architecture for distributed machine learning, makes use of two types of nodes: *workers* and *parameter servers*. The former is used to store the partitioned data and the latter to store the partitioned model. Workers communicate with the parameter servers and push/ pull gradient updates. Therefore, this architecture can be leveraged for hybrid parallelism (simultaneous data and model parallelism). [23] is one such work where parameter servers have been used to provide simultaneous data and model parallel formulation for binary regularized risk minimization problems. However parameter server has its own challenges: (1) There is an added overhead in network bandwidth arising due to communication between the layers of workers and parameter servers, (2) There is some effort required to strike the right balance between hardware efficiency and statistical efficiency while setting up the resource allocation (ratio of # of worker nodes to # of parameter servers). Adding too few parameter servers could cause the model to converge very slowly or not converge at all (poor statistical efficiency) due to insufficient rounds of synchronization. On the other hand, adding too many of these servers to enable frequent model synchronization, could take hardware resources away from the workers (poor hardware efficiency), (3) Moreover, the optimal resource allocation of workers and parameter servers depends on several factors such as the cluster size, hardware characteristics, and the training data. These challenges have been explored in much more detail in [22] with some empirical study. The work in [22] provides a motivation towards exploring de-centralized architectures for distributed machine learning. *Our proposed method DS-MLR is a step in this direction, where at any given point of time, both data as well as model stays truly partitioned into mutually exclusive blocks across the workers. Parameter updates are directly exchanged across workers, eliminating the need for any intermediate servers.*

**Asynchronous vs Synchronous**: Parameter Server and HogWild [18] are asynchronous approaches. In Hogwild, parameter updates are executed in parallel using different threads under the assumption that any two serial updates are not likely to collide on the same data point when the data is sparse. *DS-MLR does not make any such assumptions. It has both synchronous and asynchronous variants and the latter is in the spirit of NOMAD* [29].

Alternating direction method of multipliers (ADMM) [5] is another popular technique to parallelize convex optimization problems. The key idea in ADMM is to reformulate the original optimization problem by introducing redundant linear constraints. This makes the new objective easily *data parallel*. However, *ADMM suffers from a similar drawback as L-BFGS when applied to multinomial logistic regression*. The number of redundant constraints that need to be introduced are $N$ (# data points) $\times$ $K$ (# classes) which is a major bottleneck to model parallelism. Moreover, the convergence rate of ADMM for MLR is known to be slow as discussed in [11].

Log-Concavity (LC) method [11] proposed a distributed *model parallel* approach to solve the multinomial logistic regression problem by linearizing the log-partition function based on its variational form [4]. However, because their formulation is only model parallel - *the entire data has to be replicated across all the workers, and a bulk-synchronization step is required per iteration to accumulate the partial models from various machines.* This is not practical for real world applications when both the data and model sizes get larger. Interestingly, we noticed that the objective function of the LC method can also be recovered from the objective function of DS-MLR (5).

**Doubly-Separable formulations**: Our reformulation in DS-MLR exploits the doubly-separable [27] structure in terms of global model parameters and some local auxiliary variables. Other doubly-separable methods also exist such as NOMAD [29] for matrix completion and RoBiRank [28] for latent collaborative retrieval. NOMAD [29] is a distributed-memory, asynchronous and decentralized algorithm and RoBiRank [28] is also a distributed-memory but synchronous algorithm.

## 3 MULTINOMIAL LOGISTIC REGRESSION

Consider training data of the form $(\mathbf{x}_i, y_i)_{i=1,\ldots,N}$ where $\mathbf{x}_i \in \mathbb{R}^d$ is a $d$-dimensional feature vector and $y_i \in \{1, 2, \ldots, K\}$ is a label associated with it; $K$ denotes the number of class labels. Let $y_{ik} = I(y_i = k)$ denote the membership of data point $\mathbf{x}_i$ to class $k$. The probability that $\mathbf{x}_i$ belongs to class $k$ is given by:

$$p(y = k|\mathbf{x}_i) = \frac{\exp(\mathbf{w}_k^T \mathbf{x}_i)}{\sum_{j=1}^K \exp(\mathbf{w}_j^T \mathbf{x}_i)}, \tag{1}$$

where $W = \{\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K\}$ denotes the parameter vector for each of the $K$ classes. Using the negative log-likelihood of (1) as a loss function, the L2-regularized objective function of MLR can be written as:

$$L_1(W) = \frac{\lambda}{2} \sum_{k=1}^K \|\mathbf{w}_k\|^2 - \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \mathbf{w}_k^T \mathbf{x}_i + \frac{1}{N} \sum_{i=1}^N \log \left( \sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x}_i) \right), \tag{2}$$

where $\lambda$ is the regularization hyper-parameter. Table 2 summarizes these notations. Optimizing the above objective function (2) when the number of classes $K$ is large, is extremely challenging as computing the *log partition function* involves summing up over a large number of classes. In addition, it couples the class level parameters $\mathbf{w}_k$ together, making it difficult to distribute computation. In this paper, we present an alternative formulation for MLR, to address this challenge.

## 4 DOUBLY-SEPARABLE MULTINOMIAL LOGISTIC REGRESSION (DS-MLR)

In this section, we present a reformulation of the MLR problem, which is closer in spirit to dual-decomposition methods [6]. We begin by first rewriting (2) as,

$$L_1(W) = \frac{\lambda}{2} \sum_{k=1}^K \|\mathbf{w}_k\|^2 - \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \mathbf{w}_k^T \mathbf{x}_i - \frac{1}{N} \sum_{i=1}^N \log \frac{1}{\sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x}_i)}, \tag{3}$$

This can be expressed as a constrained optimization problem,

$$L_1(W, A) = \frac{\lambda}{2} \sum_{k=1}^K \|\mathbf{w}_k\|^2 - \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \mathbf{w}_k^T \mathbf{x}_i - \frac{1}{N} \sum_{i=1}^N \log a_i, \tag{4}$$

$$\text{s.t.} \quad a_i = \frac{1}{\sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x}_i)}, \quad i = 1, 2, \ldots N$$

where $A = \{a_i\}_{i=1,\ldots,N}$.

Observe that this resembles dual-decomposition methods of the form:
$\min_{x,z} f(x) + g(z)$ s.t. $Ax + Bz = c$, where $f$ and $g$ are convex functions. In our objective function (4), the decomposable functions are $f(W)$ and $g(A)$ respectively. Introducing Lagrange multipliers, $\beta_i$, $\quad i = 1, 2 \ldots N$, we obtain the equivalent unconstrained minimax problem [6],

$$L_2(W, A, \beta) = \frac{\lambda}{2} \sum_{k=1}^K \|\mathbf{w}_k\|^2 - \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K y_{ik} \mathbf{w}_k^T \mathbf{x}_i - \frac{1}{N} \sum_{i=1}^N \log a_i + \frac{1}{N} \sum_{i=1}^N \sum_{k=1}^K \beta_i \; a_i \exp(\mathbf{w}_k^T \mathbf{x}_i) - \frac{1}{N} \sum_{i=1}^N \beta_i \tag{5}$$

It is known that dual-decomposition methods can reliably find a stationary point, therefore the solution obtained by our method is also globally optimal. The updates for the primal variables $W$, $A$ and dual variable $\beta$ can be written as follows:

$$W_k^{t+1} \leftarrow \underset{W_k}{\text{argmin}} \, L_2(W_k, a^t, \beta^t), \tag{6}$$

$$a_i^{t+1} \leftarrow \underset{a_i}{\text{argmin}} \, L_2(W_k^{t+1}, a_i^t, \beta_i^t), \tag{7}$$

$$\beta_i^{t+1} \leftarrow \beta_i^t + \rho \left( a_i^{t+1} \sum_{k=1}^K \exp\left( {w_k^T}^{t+1} x_i \right) - 1 \right) \tag{8}$$

Here, $W_k^{t+1}$ and $a_i^{t+1}$ can be obtained by any black-box optimization procedure, while $\beta_i^{t+1}$ is updated via dual-ascent using a step-length $\rho$. Intuitively, the dual-ascent update of $\beta$ penalizes any violation of the constraint in problem (4).

We make the following interesting observations in these updates:

**Update for $a_i^{t+1}$**: When (7) is solved to optimality, $a_i$ admits an exact closed-form solution given by,

$$a_i = \frac{1}{\beta_i \sum_{k=1}^K \exp(\mathbf{w}_k^T \mathbf{x}_i)}, \tag{9}$$

**Update for $\beta_i^{t+1}$**: As a consequence of the above exact solution for $a_i$, the dual-ascent update for $\beta_i$ is no longer needed, since the penalty is always zero during such a projection if $\beta_i$ is set to a

| Symbol | Definition |
|---|---|
| $N$ | total number of observations |
| $D$ | total number of dimensions |
| $K$ | total number of classes |
| $x = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$, $\mathbf{x}_i \in \mathbb{R}^D$ | data points |
| $y = \{y_1, \ldots, y_N\}$, $y_i \in \{1, 2, \ldots, K\}$ | class the data point $x_i$ belongs to (i.e. label) |
| $W = \{\mathbf{w}_1, \ldots, \mathbf{w}_K\}$, $\mathbf{w}_k \in \mathbb{R}^D$ | parameters of the model |
| $a = \{a_1, \ldots, a_N\}$, $a_i \in \mathbb{R}$ | auxiliary variables mapping one-one to the observations |
| $b = \{b_1, \ldots, b_N\}$, $b_i \in \mathbb{R}$ | auxiliary variables used to represent $\log a_i$ for convenience |
| $y_{ik} = I(y_i = k)$ | Indicator variable denoting the membership of data point $x_i$ to class $k$ |
| $\lambda$ | regularization hyper-parameter |
| $\eta$ | learning rate hyper-parameter |

**Table 2: Notations for Multinomial Logistic Regression**

constant equal to 1.

**Update for $W_k^{t+1}$**: This is the only update that we need to handle numerically.

$L_2(W, A)$ can be first written in this form,

$$L_2(W, B) = \sum_{i=1}^{N}\sum_{k=1}^{K}\left(\frac{\lambda}{2N}\|\mathbf{w}_k\|^2 - \frac{1}{N}y_{ik}\mathbf{w}_k^T\mathbf{x}_i - \frac{1}{NK}b_i + \frac{1}{N}\exp(\mathbf{w}_k^T\mathbf{x}_i + b_i) - \frac{1}{NK}\right) \quad (10)$$

where we denote $b_i = \log(a_i)$ for convenience and $B = \{b_i\}_{i=1,\ldots,N}$. The objective function is now *doubly-separable* [27] since,

$$L_2(w_1, \ldots, w_K, b_1, \ldots, b_N) = \sum_{i=1}^{N}\sum_{k=1}^{K} f_{ki}(\mathbf{w}_k, b_i) \quad (11)$$

where

$$f_{ki}(\mathbf{w}_k, b_i) = \frac{\lambda}{2N}\|\mathbf{w}_k\|^2 - \frac{y_{ik}\mathbf{w}_k^T\mathbf{x}_i}{N} + \frac{\exp(\mathbf{w}_k^T\mathbf{x}_i + b_i)}{N} - \frac{b_i}{NK} - \frac{1}{NK} \quad (12)$$

Obtaining such a form for the objective function is key to achieving simultaneous data and model parallelism. It is worth pointing out that such an objective function can also be derived using the variational form for the log-partition function [4].

**Stochastic Optimization:** Minimizing $L_2(W, B)$ involves computing the gradients of eqn (10) w.r.t. $\mathbf{w}_k$ which is often computationally expensive. Instead, one can compute *stochastic gradients* [19] which are computationally cheaper than the exact gradient, and perform stochastic updates as follows:

$$\mathbf{w}_k \leftarrow \mathbf{w}_k - \eta_t K\left(\lambda\mathbf{w}_k - y_{ik}\mathbf{x}_i + \exp(\mathbf{w}_k^T\mathbf{x}_i + b_i)\mathbf{x}_i\right) \quad (13)$$

where $\eta_t$ is the learning rate for $\mathbf{w}_k$ in the $t$-th iteration.

**Advantages of our reformulation of DS-MLR in eqn (10)**:

(1) The objective function $L_2(W, B)$ splits as summations over $N$ data points and $K$ classes. Therefore, each term in the stochastic updates only depends one data point $i$ and one class $k$. We exploit this to achieve simultaneous data and model parallelism.
(2) We are able to update the variational parameters $b_i$ in closed-form, avoiding noisy stochastic updates. This improves our overall convergence.
(3) Our formulation lends itself nicely to an asynchronous implementation. Section 5.2 describes this in more detail.

# 5 DISTRIBUTING THE COMPUTATION OF DS-MLR

## 5.1 DS-MLR Synchronous

We first describe the distributed DS-MLR Synchronous algorithm in Algorithm 1. The data and parameters are distributed among the $P$ processors as illustrated in Figure 2 where the row-blocks and column-blocks represent data $X^{(p)}$ and weights $W^{(p)}$ on each local processor respectively. The algorithm proceeds by running $T$ iterations in parallel on each of the $P$ workers arranged in a ring network topology.

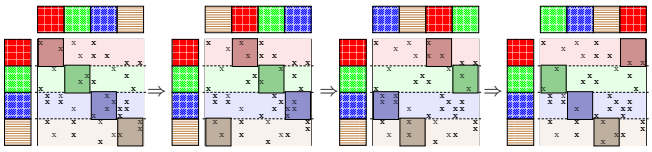


**Figure 2: $P = 4$ inner-epochs of distributed SGD. Each worker updates mutually-exclusive blocks of data and parameters as shown by the dark colored diagonal blocks [9].**

Each iteration consist of $2P$ inner-epochs. During the first $P$ inner-epochs, each worker sends/receives its parameters $W^{(p)}$ to/from the adjacent machine and performs stochastic $W^{(p)}$ updates using the block of data $X^{(p)}$ and parameters $W^{(p)}$ that it owns. The second $P$ inner-epochs are used to pass around the $W^{(p)}$ to compute the $b^{(p)}$ exactly using (9).

## 5.2 DS-MLR Asynchronous

The performance of DS-MLR Sync can be significantly improved by performing computation and communication in parallel. Thanks to the double-separable nature of our objective function (10), this can be easily achieved by applying the NOMAD algorithm [29]. The entire DS-MLR Async algorithm is described in Algorithm 2.

The algorithm begins by distributing the data and parameters among $P$ workers in the same fashion as in the synchronous version. However, here we also maintain $P$ worker queues. Initially

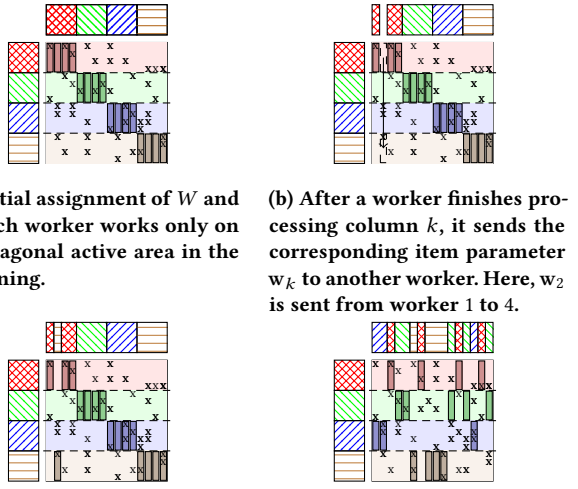
**Algorithm 1** DS-MLR Synchronous

$K$: # classes, $P$: # workers, $T$: total outer iterations, $t$: outer iteration index, $s$: inner epoch index
$W^{(p)}$: weights per worker, $b^{(p)}$: variational parameters per worker
Initialize $W^{(p)} = 0$, $b^{(p)} = \frac{1}{K}$
**for all** $p = 1, 2, \ldots, P$ in parallel **do**
  **for all** $t = 1, 2, \ldots, T$ **do**
    **for all** $s = 1, 2, \ldots, P$ **do**
      Send $W^{(p)}$ to worker on the right
      Receive $W^{(p)}$ from worker on the left
      Update $W^{(p)}$ stochastically using (13)
    **end for**
    **for all** $s = 1, 2, \ldots, P$ **do**
      Send $W^{(p)}$ to worker on the right
      Receive $W^{(p)}$ from worker on the left
      Compute partial sums
    **end for**
    Update $b^{(p)}$ exactly (9) using the partial sums
  **end for**
**end for**

**Algorithm 2** DS-MLR Asynchronous

$K$: total # classes, $P$: total # workers, $T$: total outer iterations, $W^{(p)}$: weights per worker
$b^{(p)}$: variational parameters per worker, queue[$P$]: array of $P$ worker queues
Initialize $W^{(p)} = 0$, $b^{(p)} = \frac{1}{K}$ `//Initialize parameters`
**for** $k \in W^{(p)}$ **do**
  Pick $q$ uniformly at random
  queue[$q$].push($(k, \mathbf{w}_k)$) `//Initialize worker queues`
**end for**
`//Start P workers`
**for all** $p = 1, 2, \ldots, P$ in parallel **do**
  **for all** $t = 1, 2, \ldots, T$ **do**
    **repeat**
      $(k, \mathbf{w}_k) \leftarrow$ queue[p].pop()
      Update $\mathbf{w}_k$ stochastically using (13)
      Compute partial sums
      Compute index of next queue to push to: $\hat{q}$
      queue[$\hat{q}$].push($(k, \mathbf{w}_k)$)
    **until** # of updates is equal to $K$
    Update $b^{(p)}$ exactly (9) using the partial sums
  **end for**
**end for**

the parameters $W^{(p)}$ are distributed uniformly at random across the queues. The workers subsequently can run their updates in parallel as follows: each one pops a parameter $\mathbf{w}_k$ out the queue, updates it stochastically and pushes it into the queue of the next worker. Simultaneously, each worker also records the partial sum (the local contribution of each worker towards the global normalization constant $\sum_{k=1}^{K} exp(w_k^T x_i)$) that is required for updating the variational parameters. This process repeats until $K$ updates have been made which is equivalent to saying that each worker has updated every parameter $\mathbf{w}_k$. Following this, the worker updates all its variational parameters $b^{(p)}$ exactly using the partial sums (9). For simplicity of explanation, we restricted Algorithm 2 to $P$ workers on a single-machine. However, in our actual implementation, there are multiple threads running on a single machine in addition to multiple machines sharing the load across the network. Therefore, in this setting, each worker (thread) first passes around the parameter $\mathbf{w}_k$ across all the threads on its machine. Once this is completed, the parameter is tossed onto the queue of the first thread on the next machine.



**(a) Initial assignment of $W$ and $X$. Each worker works only on the diagonal active area in the beginning.**

**(b) After a worker finishes processing column $k$, it sends the corresponding item parameter $\mathbf{w}_k$ to another worker. Here, $\mathbf{w}_2$ is sent from worker 1 to 4.**

**(c) Upon receipt, the column is processed by the new worker. Here, worker 4 can now process column 2 since it owns the column.**

**(d) During the execution of the algorithm, the ownership of the global parameters (weight vectors) $\mathbf{w}_k$ changes.**

**Figure 3: Illustration of the communication pattern in DS-MLR Async algorithm. Parameter vector $\mathbf{w}_k$ is exchanged in a de-centralized manner across workers without the use of any parameter servers [14].**

## 6 CONVERGENCE ANALYSIS

In this section, we present the convergence analysis of DS-MLR. The flavor of stochasticity we use in DS-MLR is *sampling without replacement* [21], which is also popularly known as *Incremental Gradient Descent* [15], [2] and is found to converge faster[2] in practice than vanilla *sampling with replacement* SGD [12]. For the asynchronous case, we make an additional assumption which is a sufficient condition to characterize gradient delays. Such a condition has been widely used to prove convergence of asynchronous SGD algorithms as discussed in [30]. Theorem 1 presents the rate for the synchronous version of DS-MLR.

Theorem 1. *Suppose all* $\|\mathbf{x}_i\| \leq r$ *for a constant* $r > 0$. *Let the step size* $\eta_t$ *in* (13) *decay at the rate of* $\frac{\eta_0}{\sqrt{t}}$ *where* $\eta_0$ *is a carefully tuned*

[2] [2] outlines exact conditions under which Incremental Gradient Descent converges namely: diminishing step sizes and choosing indices in a cyclic order, and re-shuffling at the end of cycle. Our implementation of DS-MLR follows these guidelines closely.

*hyper-parameter. Then, under standard assumptions of smoothness, strong convexity, lipschitz hessian and bounded gradients,*

$$\mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2] \leq \frac{\|\mathbf{w}_k^{1,n} - \mathbf{w}_k^*\|^2 + \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1}}{\sqrt{t}}, \quad \forall t = \{1, 2, \ldots, T\} \tag{14}$$

*where $\mathbf{w}_k^{t,n}$ is value of parameter vector $\mathbf{w}_k$ at outer and inner iterations indexed by $t$ and $n$ respectively, $\mathbf{w}_k^*$ is the optimal solution, $\mathbf{x}_i$ denotes the data point, $M_1 = nC_4$ and $M_2 = n^2C_5$ (where constants $C_4$ and $C_5$ depend on $L$ and $\mu$).*

Proof is available in Appendix A. The key steps in our analysis are as follows:

- *First*, for the $t$-th iteration, we introduce a random variable $R^t$ to absorb the effects of re-shuffling the indices within the epoch by closely following results in [12].
- *Second*, to account for the delay and staleness in updates for $\mathbf{w}_k$ in the inner-iterations, we prove and make use of Lemma 2 to bound the staleness in $\nabla f_i$.
- *Finally*, we prove our main result using a proof by induction (see Lemma 3) argument revealing the $\frac{1}{\sqrt{t}}$ rate.

**Remark**: Our analysis can also be easily adapted to prove $\frac{1}{t}$ rate using step-size of $\frac{\eta_0}{t}$. However, in practice, we found $\frac{\eta_0}{\sqrt{t}}$ to be slightly more stable. Using an assumption on boundedness of the delay, we can use of results in [30] to achieve $\frac{1}{\sqrt{t}}$ rates for DS-MLR Async for a suitable diminishing step-size sequence.

## 7 EXPERIMENTAL RESULTS

In our empirical study, we analyze the behavior of DS-MLR Async by running it on several real-world datasets of varying scale. Table 3 provides a summary of their characteristics.

**Hardware:** All single-machine experiments were run on a cluster with the configuration of two 8-core Intel Xeon-E5 processors and 32 GB memory per node. For multi-machine multi-core, we used Intel vLab Knights Landing (KNL) cluster with node configuration of Intel Xeon Phi 7250 CPU (64 cores, 200GB memory), connected through Intel Omni-Path (OPA) Fabric.

**Implementation Details:** We implemented DS-MLR in C++ using MPI for communication across nodes and Intel TBB for concurrent queues and multi-threading. To make the comparison fair, we re-implemented the LC [11] method in C++ and MPI using ALGLIB for inner optimization. For the L-BFGS baseline, we used the TAO solver (from PETSc). Although, there exist numerous data and model parallel methods, we use these as representative baselines.

**Reproducibility:** The hyper-parameter values and node configuration used in our experiments are in Table 3. Code and scripts required for reproducing the experiments are readily available for download from https://bitbucket.org/params/dsmlr. The repository includes instructions to compile and run the code and scripts to launch the jobs on a HPC cluster with similar capability as ours.

### 7.1 Comparison with other methods

#### 7.1.1 *SMALL SCALE DATASETS.*

**CLEF, NEWS20, LSHTC1-small**: For this experiment, we compare DS-MLR, L-BFGS and the LC methods on small scale datasets which can easily fit in the memory of a single machine and therefore require no parallelism. In such scenarios, a second order methods

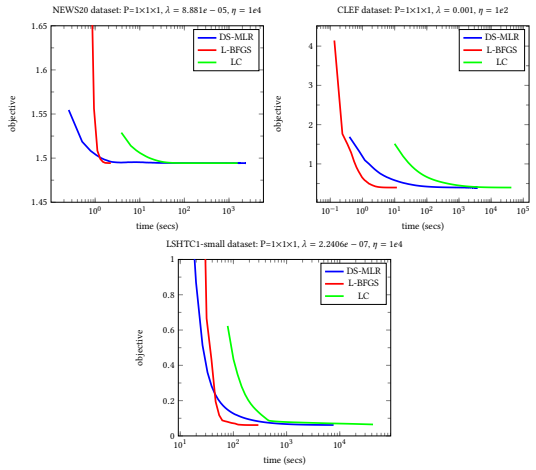



**Figure 4: Data and Model both fit in memory. In each plot, P=N×M×T denotes that there are $N$ nodes each running $M$ mpi tasks, with $T$ threads each. $\lambda$ and $\eta$ refer to regularization and learning-rate.**

such as L-BFGS are theoretically expected to out-perform stochastic methods due their superior quadratic convergence rates. In our experiments, we observed that it is indeed the case. When comparing DS-MLR against LC (which is our model parallel baseline method) we found that DS-MLR consistently shows a faster decrease in objective value compared to LC on all three datasets: NEWS20, LSHTC1-small and CLEF. LC stalls towards the end and progresses very slowly as seen in the plots. Figure 4 shows the progress of objective function as a function of time for DS-MLR, L-BFGS and LC on the three datasets.

#### 7.1.2 *LARGE SCALE DATASETS.*

**LSHTC1-large:** L-BFGS requires all its parameters to fit on one machine and is therefore not suited for model parallelism (even a modest dataset such as LSHTC1-large requires ≈ 4.2 billion parameters or ≈ 34GB). Thus, parallelizing L-BFGS would involve duplicating 34 GB of parameters across all its processors. We ran both DS-MLR and LC using 48 workers. Figure 1b (left) shows how the objective function changes vs time for DS-MLR and LC. As can be seen, DS-MLR out performs LC by a wide-margin despite the advantage LC has by duplicating data across all its processors.

**ODP**: We ran DS-MLR on ODP dataset [3] which has a huge model parameter size of 355 GB. For this experiment we used 20 nodes × 1 mpi task × 260 threads. The progress in decreasing the objective function value is shown in Figure 1b (center). LC method being a second-order method has a very high per-iteration cost and it takes an enormous amount of time to finish even a single iteration.

**YouTube8M-Video**: This dataset was created by pre-processing the publicly available dataset of youtube video embeddings [4] into

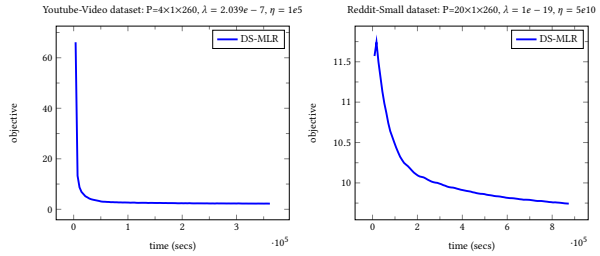
[3] https://github.com/JohnLangford/vowpal_wabbit/tree/master/demo/recall_tree/odp

[4] https://research.google.com/youtube8m/

| Dataset | # instances | # features | #classes | data (train + test) | parameters | sparsity (% nnz) | Node configuration | Hyper-parameter values |
|---|---|---|---|---|---|---|---|---|
| CLEF | 10,000 | 80 | 63 | 9.6 MB + 988 KB | 40 KB | 100 | P=1×1×1 | $\lambda = 0.001, \eta = 1e2$ |
| NEWS20 | 11,260 | 53,975 | 20 | 21 MB + 14 MB | 9.79 MB | 0.21 | P=1×1×1 | $\lambda = 8.881e-05, \eta = 1e4$ |
| LSHTC1-small | 4,463 | 51,033 | 1,139 | 11 MB + 4 MB | 465 MB | 0.29 | P=1×1×1 | $\lambda = 2.2406e-07, \eta = 1e4$ |
| LSHTC1-large | 93,805 | 347,256 | 12,294 | 258 MB + 98 MB | 34 GB | 0.049 | P=4×1×12 | $\lambda = 1e-7, \eta = 20e4$ |
| ODP | 1,084,404 | 422,712 | 105,034 | 3.8 GB + 1.8 GB | 355 GB | 0.0533 | P=20×1×260 | $\lambda = 9.221e-7, \eta = 1e5$ |
| YouTube8M-Video | 4,902,565 | 1,152 | 4,716 | 59 GB + 17 GB | 43 MB | 100 | P=4×1×260 | $\lambda = 2.039e-7, \eta = 1e5$ |
| Reddit-Small | 52,883,089 | 1,348,182 | 33,225 | 40 GB + 18 GB | 358 GB | 0.0036 | P=20×1×260 | $\lambda = 1e-19, \eta = 5e10$ |
| Reddit-Full | 211,532,359 | 1,348,182 | 33,225 | 159 GB + 69 GB | 358 GB | 0.0036 | P=40×1×250 | $\lambda = 1e-19, \eta = 4e10$ |

**Table 3: Characteristics of the datasets used and experimental settings ($P = N \times M \times T$ denotes $N$ nodes each running $M$ mpi tasks, with $T$ threads each)**

a multi-class classification dataset consisting of 4,716 classes and 1,152 features. Since it was created from features derived from embeddings, it is a dense dataset. We used the configuration of 4




**(a) Data does not fit and Model fits.** **(b) Data does not fit and Model does not fit.**

**Figure 5**

nodes × 1 mpi tasks × 260 threads to run DS-MLR on this dataset and we observed a fast convergence as shown in Figure 5a. This is likely because DS-MLR being non-blocking and asynchronous in nature runs at its peak performance on a dense dataset such as YouTube8M-Video, since the number of non-zeros in the data remains uniform across all its workers.

**Reddit datasets**: In this sub-section, we demonstrate the capability of DS-MLR to solve a multi-class classification problem of massive scale, on a new benchmark dataset *RedditFull* which we created out of 1.7 billion reddit user comments spanning the period 2007-2015. Our aim is to classify a particular reddit comment into a suitable sub-reddit. The data and model parameters occupy 228 GB and 358 GB respectively. Therefore, both L-BFGS and LC cannot be applied here. We also created a smaller subset of this dataset *Reddit-Small* by sub-sampling around 50 million data points. The result of running DS-MLR on these two datasets are shown in Figure 5b and Figure 1b (right) respectively.

## 7.2 Predictive performance of DS-MLR

In this section, we plot the cumulative distribution function (CDF) of ranks of test labels. This is a proxy for the *precision@k* curve and gives a more closer indication of the predictive performance of a multinomial classification algorithm. In Figure 6, we plot the precision obtained after the first 5 iterations (denoted by dashed lines), and after the end of optimization (denoted by solid lines). As seen, DS-MLR performs competitively well compared to other methods in all datasets, and in general tends to give a good accuracy within the first 5 iterations. *Using roughly $\frac{1}{4}$ top-k classes was enough to get a predictive performance of around 95% in all datasets.*

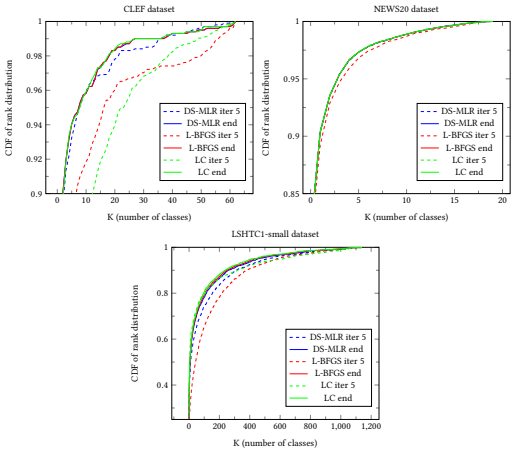



**Figure 6: Cumulative distribution function (CDF) of predictive ranks of the test labels for three sample datasets. DS-MLR performs competitively well within the first 5 iterations. Using roughly $\frac{1}{4}$ top-k classes was enough to get a predictive performance of around 95% in all datasets.**

## 7.3 Scaling behavior of DS-MLR

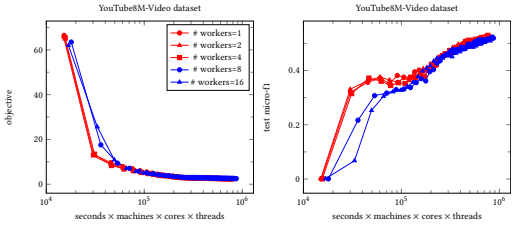



**Figure 7: Scalability analysis of DS-MLR on YouTube8M-Video dataset: Change in objective function and test f1-score vs computation time varying the # of workers (machines).**

In Figures 7 and 8, we analyze the scaling behavior of DS-MLR under the settings of multi-machine and multi-thread parallelism. We picked a dataset for each of these scenarios: YouTube8M-Video and

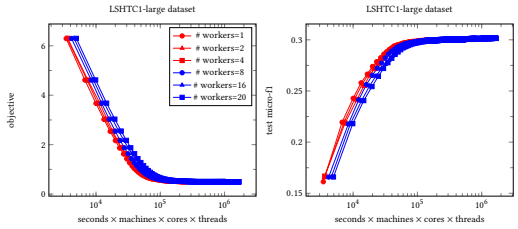



**Figure 8: Scalability analysis of DS-MLR on LSHTC1-large dataset - Change in objective function and test f1-scores vs computation time varying the # of workers (threads).**

LSHTC1-large respectively. We plot the rate of change in objective function as well as the f-score as the number of workers (# machines × # cores × # threads) is varied. For YouTube8M-Video dataset, we vary the number of machines as 1, 2, 4, 8, 16. For LSHTC1-large, DS-MLR can handle this dataset on a single machine, therefore, we simply vary the number of threads on a single machine (as a single mpi task) as 1, 2, 4, 8, 16, 20. In an ideal scenario with linear scaling, we would expect all the figures to overlap with each other. From the plot we observe that multi-thread behavior is pretty close to the ideal behavior while in multi-machine case there is some slowdown with 8 and 10 workers. This is most likely due to the communication and network overheads in the cluster.

## 8 CONCLUSION

In this paper, we present a novel distributed stochastic optimization algorithm DS-MLR to solve multinomial logistic regression problems having large number of examples and classes. By exploiting double-separability, we present a reformulation that is hybrid parallel (both data and model parallel simultaneously). DS-MLR is able to perfectly partition the workload across $P$ workers, costing $O(\frac{ND}{P})$ storage for data and $O(\frac{KD}{P} + \frac{N}{P})$ for the model. As a result, DS-MLR can scale to arbitrarily large datasets. DS-MLR is fully de-centralized unlike the parameter-server architecture. Parameter updates are directly exchanged asynchronously across workers, eliminating the need for any intermediate servers. We provide empirical results showing DS-MLR applies to all regimes of distributed machine learning, especially the case where both data and model sizes exceed the memory capacity of a single machine. To show this, we created a benchmark dataset (Reddit-Full) to run extreme multi-class classification with 228 GB data and 358 GB parameters. Future directions of work include topics such as extreme multi-label classification [1], [13] and log-linear parameterization for undirected graphical models which exhibit similar computational challenges.

## 9 ACKNOWLEDGMENTS

The authors would like to thank C. Seshadhri for valuable inputs on the manuscript. This research was supported by NSF grant 1546459.

## REFERENCES

[1] R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma. Multi-label learning with millions of labels: Recommending advertiser bid phrases for web pages. In *Proceedings of the 22nd international conference on World Wide Web*, pages 13–24. ACM, 2013.

[2] D. P. Bertsekas. Incremental gradient, subgradient, and proximal methods for convex optimization: A survey. *Optimization for Machine Learning*, 2010(1-38):3, 2011.

[3] L. Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*. Springer, 2010.

[4] G. Bouchard. Efficient bounds for the softmax function, applications to inference in hybrid models. 2007.

[5] 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*, 3(1):1–122, 2011.

[6] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, Cambridge, England, 2004.

[7] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, K. Olukotun, and A. Y. Ng. Map-reduce for machine learning on multicore. In *Advances in neural information processing systems*, pages 281–288, 2007.

[8] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, et al. The youtube video recommendation system. In *Proceedings of the fourth ACM conference on Recommender systems*, pages 293–296. ACM, 2010.

[9] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. ACM, 2011.

[10] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[11] S. Gopal and Y. Yang. Distributed training of large-scale logistic models. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 289–297, 2013.

[12] J. Z. HaoChen and S. Sra. Random shuffling beats sgd after finite epochs. *arXiv preprint arXiv:1806.10077*, 2018.

[13] H. Jain, Y. Prabhu, and M. Varma. Extreme multi-label loss functions for recommendation, tagging, ranking & other missing label applications. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 935–944. ACM, 2016.

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

[15] A. Nedić and D. Bertsekas. Convergence rate of incremental subgradient algorithms. In *Stochastic optimization: algorithms and applications*, pages 223–264. Springer, 2001.

[16] Y. Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[17] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer Series in Operations Research. Springer, 2nd edition, 2006.

[18] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[19] H. E. Robbins and S. Monro. A stochastic approximation method. *Annals of Mathematical Statistics*, 22:400–407, 1951.

[20] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[21] O. Shamir. Without-replacement sampling for stochastic gradient methods. In *Advances in Neural Information Processing Systems*, pages 46–54, 2016.

[22] P. Watcharapichat, V. L. Morales, R. C. Fernandez, and P. Pietzuch. Ako: Decentralised deep learning with partial gradient exchange. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 84–97. ACM, 2016.

[23] L. Xiao, A. W. Yu, Q. Lin, and W. Chen. Dscovr: Randomized primal-dual block coordinate algorithms for asynchronous distributed optimization. *arXiv preprint arXiv:1710.05080*, 2017.

[24] P. Xie, J. K. Kim, Y. Zhou, Q. Ho, A. Kumar, Y. Yu, and E. P. Xing. Distributed machine learning via sufficient factor broadcasting. *CoRR*, 2015.

[25] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: a new platform for distributed machine learning on big data. *Big Data, IEEE Transactions on*, 2015.

[26] I. E.-H. Yen, X. Huang, P. Ravikumar, K. Zhong, and I. Dhillon. Pd-sparse : A primal and dual sparse approach to extreme multiclass and multilagrawal2013multoabel classification. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 3069–3077, 2016.

[27] H. Yun. *Doubly Separable Models*. PhD thesis, Purdue University West Lafayette, 2014.

[28] H. Yun, P. Raman, and S. Vishwanathan. Ranking via robust binary classification. In *Advances in Neural Information Processing Systems*, 2014.

[29] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon. Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. 2013.

[30] X. Zhang, J. Liu, and Z. Zhu. Taming convergence for asynchronous stochastic gradient descent with unbounded delay in non-convex learning. *arXiv preprint arXiv:1805.09470*, 2018.

[31] Y. Zhuang, Y.-C. Juan, and C.-J. Lin. A fast parallel stochastic gradient method for matrix factorization in shared memory systems. 2013.

[32] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola. Parallelized stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 2595–2603, 2010.

# A DETAILS OF THE ANALYSIS AND PROOF OF THEOREM 1

In DS-MLR, for each epoch $t \in \{1, \ldots, T\}$, where $T$ is the total number of iterations, the algorithm performs the following Incremental Gradient Descent steps: *(a) Re-shuffle the indices of the data points* $\{\mathbf{x}\}_{i=1}^n$*, (b) Cyclically scan the index-set picking each data point* $x_i$ *and make stochastic gradient updates for* $\mathbf{w}_k$ *using* $(\mathbf{x}_i, \mathbf{w}_k)\ \forall k \in \{1, \ldots, K\}$.

Since in Algorithm 1, $b_i$ is updated in closed-form using (9), we can optimize $B$ out from $F(W, B)$, thereby working with $F(W)$.

$$F(W) = F(\mathbf{w}_1, \ldots, \mathbf{w}_K) = \sum_{i=1}^{N}\sum_{k=1}^{K} f_{ki}(\mathbf{w}_k) = \frac{1}{n}\sum_{i=1}^{n} f_i(W) \quad (15)$$

where $f_i(W) = \frac{\lambda}{2}\|W\|^2 - w_{y_i}^T\mathbf{x}_i + \log\sum_{k=1}^K \exp(\mathbf{w}_k^T\mathbf{x}_i)$. Clearly, $f_i$ has the variational representation,

$$f_i(W) = \frac{\lambda}{2}\|W\|^2 - w_{y_i}^T\mathbf{x}_i + \min_{a_i\in\mathbb{R}}\left\{-b_i + \sum_{k=1}^{K}\exp\left(\mathbf{w}_k^T\mathbf{x}_i + b_i\right)\right\} - 1 \quad (16)$$

**Modeling the effects of re-shuffling indices per epoch:** Let us assume a permutation function $\sigma_t(\cdot)$: [n] → [n], that produces a permutation of the indices of the data points $\{\mathbf{x}_i\}_{i=1}^n$ at the beginning of every epoch $t$. The update for $\mathbf{w}_k$ can be then written as follows:

$$\mathbf{w}_k^{t,m} = \mathbf{w}_k^{t,m-1} - \eta_t\nabla f_{\sigma_t(m)}\left(\mathbf{w}_k^{t,m-1}\right) \quad (17)$$

for $1 \le m \le n$. Here $\mathbf{w}_k^{t,m}$ represents the $m$-th iterate within the $t$-th epoch. $\eta_t$ is the step-size per epoch which is decayed as per the rule: $\eta_t = \frac{\eta_0}{\sqrt{t}}$, after choosing a carefully tuned value for $\eta_0$. We define a random variable $R^t$ to capture the gradient error due to the random re-shuffling of indices every epoch. $R^t$ is defined as,

$$R^t = \sum_{i=1}^{n}\nabla f_{\sigma_t(i)}\left(\mathbf{w}_k^{t,i-1}\right) - \sum_{i=1}^{n}\nabla f_{\sigma_t(i)}\left(\mathbf{w}_k^{t,0}\right) \quad (18)$$

**Modeling staleness in incremental gradient updates:** We are not able to make increment gradient updates using the optimal $\mathbf{w}_k^{t,n}$ for the inner-iterations $\{1, 2, \ldots, n\}$ as they remain stale during the inner-iterations. As a result of this, the variational parameter $b_i$ calculated are also stale. This could affect the convergence and needs to be factored into the analysis. Lemma 2 shows that despite the staleness, the incremental gradient computed in these $n$ inner-iterations at $\mathbf{w}_k^{t,n}$ is not too far from the true gradient at $\mathbf{w}_k^t$.

Lemma 2. *Denote the approximate gradient of* $f_{\sigma(i)}$ *evaluated at* $\mathbf{w}_k^{t,n}$ *based on* $b_i^t$ *as*

$$\tilde{G}_k^t = (\tilde{\mathbf{g}}_1, \ldots, \tilde{\mathbf{g}}_K), \quad (19)$$

*where* $\tilde{\mathbf{g}}_c = \lambda w_{k,c}^t - [y_i = c]\mathbf{x}_i + \exp(\mathbf{x}_i^T w_{k,c}^t + a_i^t)\mathbf{x}_i$.
*Then* $\left\|\tilde{G}_k^t - \nabla f_i(w_k^t)\right\| \le \frac{r}{K}\left\|w_k^{t,n} - w_k^t\right\|$.

Proof. Unfolding the term $b_i^t$,

$$\tilde{\mathbf{g}}_c - \frac{\partial}{\partial w_c}f_i(w_k^t) = \left(\frac{\exp(\mathbf{x}_i^T w_{k,c}^t)}{\sum_{c=1}^K\exp(\mathbf{x}_i^T w_c^t)} - \frac{\exp(\mathbf{x}_i^T w_{k,c}^t)}{\sum_{c=1}^K\exp(\mathbf{x}_i^T w_{k,c}^t)}\right)\mathbf{x}_i$$

Therefore

$$\left\|\tilde{G} - \nabla f_i(w_k^t)\right\| \le r\sqrt{K}\left|\frac{1}{\sum_{c=1}^K\exp(\mathbf{x}_i^T w_c^t)} - \frac{1}{\sum_{c=1}^K\exp(\mathbf{x}_i^T w_{k,c}^t)}\right|$$

So it suffices to upper bound the gradient of $1/\sum_{c=1}^K\exp(\mathbf{x}_i^T w_c)$. Since $\mathbf{x}_i$ and $w_c$ are bounded, $\exp(\mathbf{x}_i^T w_c)$ is lower bounded by a positive universal constant [5]. Now,

$$\begin{aligned}&\left\|\nabla_w\frac{1}{\sum_{c=1}^K\exp(\mathbf{x}_i^T w_c)}\right\|\\ &= \frac{1}{(\sum_{c=1}^K\exp(\mathbf{x}_i^T w_c))^2}\left\|(\exp(\mathbf{x}_i^T w_1)\mathbf{x}_i, \ldots, \exp(\mathbf{x}_i^T w_K)x_i)\right\|\\ &\le \frac{\sqrt{K}}{K^2}r\end{aligned}$$

□

Next, we bound some quantities that will prove to be useful later. $\|\nabla f_i(w)\| \le B$ by assumption of bounded gradient Without loss of generality, suppose $f_k$ is used for update at step $k$. Then $w_k^t$ is subtracted by $\frac{\eta_t}{N}(\lambda w_k^t - x_k \otimes \mathbf{e}'_{y_k} + \tilde{G}_k^t)$, where $\otimes$ is Kroneker product and $\mathbf{e}_c$ is a canonical vector. As long as $\eta_t \le \frac{1}{\lambda}$, we can recursively apply Lemma 2 and derive bounds

$$\left\|w_k^t - w^t\right\| \le \frac{k}{N}\eta_t r, \quad (20)$$

$$\left\|\nabla f_k(w_k^t) - \tilde{G}_k^t\right\| \le \eta_t r, \quad (21)$$

$$\left\|\tilde{G}_k^t\right\| \le r, \quad (22)$$

for all $k$.

**Deriving the main result:** For one epoch, we have the following inequality,

$$\begin{aligned}&\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2\\ &= \|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 - 2\eta_t\left\langle\mathbf{w}_k^{t,0} - \mathbf{w}_k^*, \sum_{i=1}^{n}\nabla f_{\sigma_t(i)}\left(\mathbf{w}_k^{t,i-1}\right)\right\rangle\\ &+ \eta_t^2\left\|\sum_{i=1}^{n}\nabla f_{\sigma_t(i)}\left(\mathbf{w}_k^{t,i-1}\right)\right\|^2 \quad (23)\\ &= \|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 - \underbrace{2\eta_t\left\langle\mathbf{w}_k^{t,0} - \mathbf{w}_k^*, n\nabla F\left(\mathbf{w}_k^{t,0}\right)\right\rangle}\\ &- 2\eta_t\left\langle\mathbf{w}_k^{t,0} - \mathbf{w}_k^*, R^t\right\rangle + \eta_t^2\left\|n\nabla F\left(\mathbf{w}_k^{t,0}\right) + R^t\right\|^2 \quad (24)\\ &\le \|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 - \underbrace{2n\eta_t\left(\frac{L\mu}{L+\mu}\|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 + \frac{1}{L+\mu}\|\nabla F\left(\mathbf{w}_k^{t,0}\right)\|^2\right)}\\ &- 2\eta_t\left\langle\mathbf{w}_k^{t,0} - \mathbf{w}_k^*, R^t\right\rangle + 2n^2\eta_t^2\|\nabla F\left(\mathbf{w}_k^{t,0}\right)\|^2 + 2\eta_t^2\|R^t\|^2 \quad (25)\\ &= \left(1 - 2n\eta_t\frac{L\mu}{L+\mu}\right)\|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 - \left(\frac{2n\eta_t}{L+\mu} - 2n^2\eta_t^2\right)\|\nabla F\left(\mathbf{w}_k^{t,0}\right)\|^2\\ &- 2\left\langle\mathbf{w}_k^{t,0} - \mathbf{w}_k^*, R^t\right\rangle + 2\eta_t^2\|R^t\|^2 \quad (26)\end{aligned}$$

where the inequality is due to Theorem 2.1.11 in [16].

[5] If one is really meticulous and notes that $\|w\|^2 \le 2\lambda\log K$ which does involve $K$, one should be appeased that $\exp(\sqrt{\log K})$ is $o(K^\alpha)$ for any $\alpha > 0$.

Take the expectation of (26) over permutation $\sigma_t(\cdot)$, we get:

$$\mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2] \leq \left(1 - 2n\eta_t \frac{L\mu}{L+\mu}\right)\|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 - \left(\frac{2n\eta_t}{L+\mu} - 2n^2\eta_t^2\right)\|\nabla F\left(\mathbf{w}_k^{t,0}\right)\|^2 - \underbrace{2\left\langle \mathbf{w}_k^{t,0} - \mathbf{w}_k^*, \mathbb{E}[R^t]\right\rangle}_{T_1} + \underbrace{2\eta_t^2\mathbb{E}[\|R^t\|^2]}_{T_2} \quad (27)$$

Terms $T_1$ and $T_2$ which involve $R^t$ capture the random effects. To bound them, we make use of Lemma 1 and Lemma 3 presented in [12] and obtain higher-order powers of $\eta_t$ as $O(\eta_t^3)$, $O(\eta_t^4)$, and $O(\eta_t^5)$. Following these steps, the expectation in (27) can be written as,

$$\mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2] \leq \left(1 - 2n\eta_t \frac{L\mu}{L+\mu}\right)\|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 + \left(2n^2\eta_t^2 - \frac{2n\eta_t}{L+\mu}\right)\|\nabla F\left(\mathbf{w}_k^{t,0}\right)\|^2 + \eta_t^3 nC_1 + \eta_t^5 n^5 C_2 + \eta_t^4 n^4 C_3 \quad (28)$$

where $C_1 = \frac{2}{\mu}L^2B^2$, $C_2 = \frac{2}{\mu}L^4B^2$, $C_3 = \frac{1}{2}L^2B^2$.

Using (22) to bound $\|\nabla F\left(\mathbf{w}_k^{t,0}\right)\|^2$ and using constants $C_4 = \frac{2L\mu}{L+\mu}$, $C_5 = 2r^2$ and $C_6 = \frac{2r^2}{L+\mu}$, we simplify (28) as:

$$\mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2] \leq \left(1 - \eta_t nC_4\right)\|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 + \eta_t^2 n^2 C_5 - \eta_t nC_6 + \eta_t^3 nC_1 + \eta_t^5 n^5 C_2 + \eta_t^4 n^4 C_3 \quad (29)$$

The term involving $C_6$ can be dropped to maintain the inequality, since $nC_6 > 0$. Since $\eta_t$ is monotonically decreasing, we can ignore the higher-order terms and further simplify the expectation as:

$$\mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2] \leq \left(1 - \eta_t nC_4\right)\|\mathbf{w}_k^{t,0} - \mathbf{w}_k^*\|^2 + \eta_t^2 n^2 C_5 \quad (30)$$

For simplicity, we denote $M_1 = nC_4$ and $M_2 = n^2C_5$. In addition, let $E_t = \sqrt{t}\,\mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2]$. Also note that, the iterate during the first inner-iteration of $t$-th epoch $\mathbf{w}_k^{t,0}$ is the same as the iterate during the last inner-iteration of the $(t-1)$-th epoch $\mathbf{w}_k^{t-1,n}$. Using these, (30) can be written in the form of a recursive inequality,

$$\frac{E_{t+1}}{\sqrt{t+1}} \leq (1 - \eta_t M_1)\frac{E_t}{\sqrt{t}} + \eta_t^2 M_2 \quad (31)$$

$$= \left(1 - \frac{\eta_0}{\sqrt{t}}M_1\right)\frac{E_t}{\sqrt{t}} + \frac{\eta_0^2}{t}M_2 \quad (32)$$

Multiplying both sides by $\sqrt{t+1}$,

$$E_{t+1} \leq \left(1 - \frac{\eta_0}{\sqrt{t}}M_1\right)\frac{\sqrt{t+1}}{\sqrt{t}}E_t + \frac{\sqrt{t+1}}{t}\eta_0^2 M_2 \quad (33)$$

We now state the following inequalities which we will use to further simplify the bound in (33),

$$\frac{\sqrt{t+1}}{\sqrt{t}} \leq \frac{\sqrt{t}+1}{\sqrt{t}} \qquad \forall t > 0, \quad (34)$$

$$\frac{\sqrt{t+1}}{t} \leq \frac{2}{\sqrt{t}} \qquad \forall t > 0, \quad (35)$$

$$\frac{\sqrt{t}+1}{\sqrt{t}} \geq 1 \qquad \forall t > 0 \quad (36)$$

Using (34) and (35), the recursive expectation in (33) becomes,

$$E_{t+1} \leq \left(1 - \frac{\eta_0}{\sqrt{t}}M_1\right)\frac{\sqrt{t}+1}{\sqrt{t}}E_t + \frac{2}{\sqrt{t}}\eta_0^2 M_2 \quad (37)$$

$$= \left(\frac{\sqrt{t}+1}{\sqrt{t}} - \frac{\sqrt{t}+1}{\sqrt{t}}\frac{\eta_0}{\sqrt{t}}M_1\right)E_t + \frac{2}{\sqrt{t}}\eta_0^2 M_2 \quad (38)$$

$$\leq \left(\frac{\sqrt{t}+1}{\sqrt{t}} - \frac{\eta_0}{\sqrt{t}}M_1\right)E_t + \frac{2}{\sqrt{t}}\eta_0^2 M_2 \quad (39)$$

where the last inequality uses (36). Assuming $\eta_0 M_1 > 1$, we get

$$E_{t+1} \leq \left(1 - \frac{\eta_0 M_1 - 1}{\sqrt{t}}\right)E_t + \frac{2}{\sqrt{t}}\eta_0^2 M_2 \quad (40)$$

We now apply the following Lemma 3 to (40) which finally leads to the main result presented in Theorem 1. Lemma 3 is proved by proof of induction.

Lemma 3. *With $E_t = \sqrt{t}\ \mathbb{E}[\|\mathbf{w}_k^{t,n} - \mathbf{w}_k^*\|^2]$, $M_1 = nC_4$ and $M_2 = n^2C_5$ (where constants $C_4$ and $C_5$ depend on $L$ and $\mu$), we can bound the expectation for $t$-th iteration relative to the first iteration as follows,*

$$E_t \leq E_1 + \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1} \quad (41)$$

Proof. (40) can be written as,

$$E_{t+1} \leq \left(1 - \frac{\eta_0 M_1 - 1}{\sqrt{t}}\right)E_t + \frac{2}{\sqrt{t}}\eta_0^2 M_2 \quad (42)$$

$$\leq \left(1 - \frac{\eta_0 M_1 - 1}{\sqrt{t}}\right)\left(E_1 + \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1}\right) + \frac{2}{\sqrt{t}}\eta_0^2 M_2 \quad (43)$$

$$\leq \left(1 - \frac{\eta_0 M_1 - 1}{\sqrt{t}}\right)E_1 + \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1} - \left(\frac{\eta_0 M_1 - 1}{\sqrt{t}} \cdot \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1}\right) + \frac{2\eta_0^2 M_2}{\sqrt{t}} \quad (44)$$

$$= \left(1 - \frac{\eta_0 M_1 - 1}{\sqrt{t}}\right)E_1 + \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1} \leq E_1 + \frac{2\eta_0^2 M_2}{\eta_0 M_1 - 1} \quad (45)$$

□