Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent

Rainer Gemulla[1] Peter J. Haas[2] Erik Nijkamp[2] Yannis Sismanis[2]

[1]Max-Planck-Institut für Informatik
Saarbrücken, Germany
rgemulla@mpi-inf.mpg.de

[2]IBM Almaden Research Center
San Jose, CA, USA
{phaas, enijkam, syannis}@us.ibm.com

Revision of IBM Technical Report RJ10481 (March 16, 2011)
February 20, 2013

We provide a novel algorithm to approximately factor large matrices with millions of rows, millions of columns, and billions of nonzero elements. Our approach rests on stochastic gradient descent (SGD), an iterative stochastic optimization algorithm. We first develop a novel "stratified" SGD variant (SSGD) that applies to general loss-minimization problems in which the loss function can be expressed as a weighted sum of "stratum losses." We establish sufficient conditions for convergence of SSGD using results from stochastic approximation theory and regenerative process theory. We then specialize SSGD to obtain a new matrix-factorization algorithm, called DSGD, that can be fully distributed and run on web-scale datasets using MapReduce. DSGD has good speed-up behavior and handles a wide variety of matrix factorizations. We describe the practical techniques used to optimize performance in our DSGD implementation. Experiments suggest that DSGD converges significantly faster and has better scalability properties than alternative algorithms.

Contents

1. Introduction

As Web 2.0 and enterprise-cloud applications proliferate, data mining algorithms need to be (re)designed to handle web-scale datasets. For this reason, low-rank matrix factorization has received much attention in recent years, since it is fundamental to a variety of mining tasks that are increasingly being applied to massive datasets [12, 17, 20, 22, 25]. Specifically, low-rank matrix factorizations are effective tools for analyzing "dyadic data" in order to discover and quantify the interactions between two given entities. Successful applications include topic detection and keyword search (where the corresponding entities are documents and terms), news personalization (users and stories), and recommendation systems (users and items). In large applications, these problems can involve matrices with millions of rows (e.g., distinct customers), millions of columns (e.g., distinct items), and billions of entries (e.g., transactions between customers and items). At such scales, distributed algorithms for matrix factorization are essential to achieving reasonable performance [12, 13, 25, 32]. In this paper, we provide a novel, effective distributed factorization algorithm based on stochastic gradient descent.

In practice, exact factorization is generally neither possible nor desired, so virtually all "matrix factorization" algorithms actually produce low-rank approximations, attempting to minimize a "loss function" that measures the discrepancy between the original input matrix and product of the factors returned by the algorithm; we use the term "matrix factorization" throughout to refer to such an approximation.

With the recent advent of programmer-friendly parallel processing frameworks such as MapReduce, web-scale matrix factorizations have become practicable and are of increasing interest to web companies, as well as other companies and enterprises that deal with massive data. Indeed, MapReduce can be used not only to factor an input matrix, but also to efficiently construct the input matrix from massive, detailed raw data, such as customer transactions. To facilitate distributed processing, prior approaches would pick an embarrassingly parallel matrix factorization algorithm and implement it on a MapReduce cluster; the choice of algorithm was driven by the ease with which it could be distributed. In this paper, we take a different approach and start with an algorithm that is known to have good performance in non-parallel environments. Specifically, we start with stochastic gradient descent (SGD), an iterative optimization algorithm which has been shown, in a sequential setting, to be very effective for matrix factorization [20]. Although the generic SGD algorithm is not embarrassingly parallel, we can exploit the special structure of the factorization problem to obtain a version of SGD that is fully distributed and scales to extremely large matrices.

The key idea is to first develop a "stratified" variant of SGD, called SSGD, that is applicable to general loss-minimization problems in which the loss function $L(\theta)$ can be expressed as a weighted sum of "stratum losses," so that $L(\theta) = w_1 L_1(\theta) + \cdots + w_q L_q(\theta)$. At each iteration, the algorithm takes a downhill step with respect to one of the stratum losses L_s, i.e., approximately in the direction of the negative gradient $-L'_s(\theta)$. Although each such direction is "wrong" with respect to minimization of the overall loss L, we prove that, under appropriate regularity conditions, SSGD will converge to a good solution for L if the sequence of strata is chosen carefully.

We then specialize SSGD to obtain a novel distributed matrix-factorization algorithm, called DSGD. Specifically, we express the input matrix as a union of (possibly overlapping) pieces, called

"strata." For each stratum, the stratum loss is defined as the loss computed over only the data points in the stratum (and appropriately scaled). The strata are carefully chosen so that each stratum has "d-monomial" structure, which allows SGD to be run on the stratum in a distributed manner. For example, a stratum corresponding to the nonzero entries in a block-diagonal matrix with k blocks is d-monomial for all $d \leq k$. The DSGD algorithm repeatedly selects a stratum according to the general SSGD procedure and processes the stratum in a distributed fashion. Stratification is a technique commonly used to reduce the variance of noisy estimates [4, Sec. V.7], such as gradient estimates in SGD; here we re-purpose the stratification technique to derive a distributed factorization algorithm with provable convergence guarantees.

Our contributions are as follows:

1. We present SSGD, a novel stratified version of SGD, that is applicable to any optimization problem in which the loss function can be represented as a weighted sum of stratum losses.
2. We formally establish sufficient conditions for the convergence of SSGD using results from stochastic approximation theory and regenerative process theory.
3. We specialize SSGD to obtain DSGD, a novel distributed algorithm for low-rank matrix factorization. Both data and factors are fully distributed. DSGD has low memory requirements and scales to matrices with millions of rows, millions of columns, and billions of nonzero elements.
4. We describe practical techniques for implementing DSGD and optimizing its performance.
5. We show that DSGD is amenable to MapReduce, a popular framework for distributed processing.
6. We compare DSGD to state-of-the-art distributed algorithms for matrix factorization. Our experiments suggest that DSGD converges significantly faster, and has better scalability.

Unlike many prior algorithms, DSGD is a generic algorithm in that it can be used for a variety of different loss functions. In this paper, we focus primarily on the class of factorizations that minimize a "nonzero loss." This class of loss functions is important for applications in which a zero represents missing data and hence should be ignored when computing loss. A typical motivation for factorization in this setting is to estimate the missing values, e.g., the rating that a customer would likely give to a previously unseen movie.

The rest of the paper is organized as follows. In Sec. 2, we introduce the factorization problem by means of an example, and discuss prior approaches to its solution. Sec. 3 describes the basic (non-parallel) SGD algorithm and its application to the matrix factorization problem. In Sec. 4 we develop the stratified variant of SGD and establish sufficient conditions for convergence. We specialize SSGD in Sec. 5 to obtain our DSGD matrix factorization algorithm. We first discuss the special "interchangeability" structure that we exploit to permit distributed execution of SGD within a stratum, and then show how to exploit this structure by means of a simple example that corresponds to processing of a single stratum by DSGD. We then give the general algorithm, which combines distributed processing within strata and careful selection of a stratum sequence. Practical implementation considerations are discussed in Sec. 6 and our empirical study of DSGD is described in Sec. 7. We conclude in Sec. 8.

2. Example and Prior Work

To gain understanding about applications of matrix factorizations, consider the "Netflix problem" [5] of recommending movies to customers. Netflix is a company that offers tens of thousands of movies for rental. The company has more than 15M customers, each of whom can provide feedback about their personal taste by rating movies with 1 to 5 stars. The feedback can be represented in a feedback matrix such as

$$\begin{array}{c} \\ \textit{Alice} \\ \textit{Bob} \\ \textit{Charlie} \end{array} \begin{array}{c} \begin{array}{ccc} \textit{Avatar} & \textit{The Matrix} & \textit{Up} \end{array} \\ \begin{pmatrix} ? & 4 & 2 \\ 3 & 2 & ? \\ 5 & ? & 3 \end{pmatrix} \end{array}.$$

Each entry may contain additional data, e.g., the date of rating or other forms of feedback such as click history. The goal of the factorization is to predict missing entries (?); entries with a high predicted rating are then recommended to users for viewing. This is an instance of a recommender system based on matrix factorization, and has been successfully applied in practice. See [20] for an excellent discussion of the intuition behind this approach.

The traditional matrix factorization problem can be stated as follows. Given an $m \times n$ matrix $\boldsymbol{V}$ and a rank r, find an $m \times r$ matrix $\boldsymbol{W}$ and an $r \times n$ matrix $\boldsymbol{H}$ such that $\boldsymbol{V} = \boldsymbol{W}\boldsymbol{H}$. As discussed previously, our actual goal is to obtain a low-rank approximation $\boldsymbol{V} \approx \boldsymbol{W}\boldsymbol{H}$, where the quality of the approximation is described by an application-dependent loss function L. We seek to find

$$\operatorname*{argmin}_{\boldsymbol{W},\boldsymbol{H}} L(\boldsymbol{V}, \boldsymbol{W}, \boldsymbol{H}),$$

i.e., the choice of $\boldsymbol{W}$ and $\boldsymbol{H}$ that give rise to the smallest loss. For example, assuming that missing ratings are coded with the value 0, loss functions for recommender systems are often based on the *nonzero squared loss*

$$L_{\text{NZSL}} = \sum_{i,j:\boldsymbol{V}_{ij}\neq 0} (\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2 \tag{1}$$

and usually incorporate regularization terms, user and movie biases, time drifts, and implicit feedback.

In the following, we restrict attention to loss functions that, like L_{NZSL}, can be decomposed into a sum of *local losses* over (a subset of) the entries in $\boldsymbol{V}_{ij}$. I.e., we require that the loss can be written as

$$L = \sum_{(i,j)\in Z} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}) \tag{2}$$

for some *training set* $Z \subseteq \{1, 2, \ldots, m\} \times \{1, 2, \ldots, n\}$ and *local loss function* l, where $\boldsymbol{A}_{i*}$ and $\boldsymbol{A}_{*j}$ denote row i and column j of matrix $\boldsymbol{A}$, respectively. Many loss functions used in practice—such as squared loss, generalized Kullback-Leibler divergence (GKL), and L_p regularization—can be decomposed in such a manner [28]; see Appendix C for more examples. Note that a given loss function L can potentially be decomposed in multiple ways. In this paper, we focus primarily on the

class of *nonzero decompositions*, in which $Z = \{ (i,j) : \boldsymbol{V}_{ij} \neq 0 \}$ refers to the nonzero entries in $\boldsymbol{V}$. As mentioned above, such decompositions naturally arise when zeros represent missing data. Our algorithms can handle other decompositions as well; see our preliminary results for GKL in Sec. 7. To avoid trivialities, we assume throughout that there is at least one training point in every row and in every column of $\boldsymbol{V}$; e.g., every customer has rated at least one movie and every movie has been rated at least once.[1]

To compute $\boldsymbol{W}$ and $\boldsymbol{H}$ on MapReduce, all known algorithms start with some initial factors $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$ and iteratively improve them. The $m \times n$ input matrix $\boldsymbol{V}$ is partitioned into $d_1 \times d_2$ blocks, which are distributed in the MapReduce cluster. Both row and column factors are blocked conformingly:

$$\begin{array}{c} \\ \boldsymbol{W}^1 \\ \boldsymbol{W}^2 \\ \vdots \\ \boldsymbol{W}^{d_1} \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{H}^1 & \boldsymbol{H}^2 & \cdots & \boldsymbol{H}^{d_2} \end{array} \\ \begin{pmatrix} \boldsymbol{V}^{11} & \boldsymbol{V}^{12} & \cdots & \boldsymbol{V}^{1d_2} \\ \boldsymbol{V}^{21} & \boldsymbol{V}^{22} & \cdots & \boldsymbol{V}^{2d_2} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{V}^{d_1 1} & \boldsymbol{V}^{d_1 2} & \cdots & \boldsymbol{V}^{d_1 d_2} \end{pmatrix} \end{array},$$

where we use superscripts to refer to individual blocks. The algorithms are designed such that each block $\boldsymbol{V}^{ij}$ can be processed independently in the map phase, taking only the corresponding blocks of factors $\boldsymbol{W}^i$ and $\boldsymbol{H}^j$ as input. Some algorithms directly update the factors in the map phase (then either $d_1 = m$ or $d_2 = n$ to avoid overlap), whereas others aggregate the factor updates in a reduce phase.

Existing algorithms can be classified into *specialized algorithms*, which are designed for a particular loss, and *generic algorithms*, which work for a wide variety of loss functions. Specialized algorithms currently exist for only a small class of loss functions. For GKL loss, Das et al. [12] provide an EM-based algorithm, and Liu et al. [25] provide a multiplicative-update method. In [25], the latter MULT approach is also applied to squared loss and nonnegative matrix factorization with an "exponential" loss function (exponential NMF). Each of these algorithms in essence takes an embarrassingly parallel matrix factorization algorithm developed previously—in [17, 18] for the EM algorithm and in [22, 23] for the MULT methods—and directly distributes it across the MapReduce cluster. Zhou et al. [32] show how to distribute the well-known alternating least squares (ALS) algorithm to handle factorization problems with a nonzero squared loss function and an optional weighted L_2 regularization term. Their approach requires a double-partitioning of $\boldsymbol{V}$: once by row and once by column. Moreover, ALS requires that each of the factor matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ can (alternately) fit in main memory. More details on each of the foregoing algorithms can be found in Appendix A.

Generic algorithms are able to handle all differentiable loss functions that decompose into summation form. A simple approach is distributed gradient descent (DGD [13, 16, 26]), which distributes gradient computation across a compute cluster, and then performs centralized parameter updates using, for example, quasi-Newton methods such as L-BFGS-B [8]. Partitioned SGD

[1] Clearly, recommendation is impossible for a customer who has never rated a movie or a movie that has never been rated; mathematically, the $\boldsymbol{W}$ factors for an empty row or the $\boldsymbol{H}$ factors for an empty column can be set to any arbitrary value without affecting the loss, so the factorization problem is not well posed in this case.

approaches make use of a similar idea: SGD is run independently and in parallel on partitions of the dataset, and parameters are averaged after each pass over the data (PSGD [16, 27]) or once at the end (ISGD [26, 27, 33]). These approaches have not been applied to matrix factorization before. Similarly to L-BFGS-B, they exhibit slow convergence in practice (see Sec. 7) and need to store the full factor matrices in memory. This latter limitation can be a serious drawback: for large factorization problems, it is crucial that both matrix and factors be distributed. Our present work on DSGD is a first step towards such a *fully distributed* generic algorithm with good convergence properties.

3. Stochastic Gradient Descent

In this section, we discuss how to factorize a given matrix via standard (non-parallel) SGD.

3.1. Preliminaries

The goal of SGD is to find the value $\theta^* \in \Re^k$ ($k \geq 1$) that minimizes a given loss $L(\theta)$. The algorithm makes use of noisy observations $\hat{L}'(\theta)$ of $L'(\theta)$, the function's gradient with respect to θ. Starting with some initial value θ_0, SGD refines the parameter value by iterating the stochastic difference equation

$$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n), \tag{3}$$

where n denotes the step number and $\{ \epsilon_n \}$ is a sequence of decreasing step sizes. (We assume throughout that each ϵ_n is nonnegative and finite.) Since $-L'(\theta_n)$ is the direction of steepest descent, (3) constitutes a noisy version of gradient descent. Figure 1 illustrates this process with an example in which θ is 2-dimensional.

Stochastic approximation theory can be used to show that, under certain regularity conditions [21], the noise in the gradient estimates "averages out" and SGD converges to the set of stationary points satisfying $L'(\theta) = 0$. Of course, these stationary points can be minima, maxima, or saddle points. One may argue that convergence to a maximum or saddle point is unlikely because the noise in the gradient estimates reduces the likelihood of getting stuck at such a point. Thus $\{ \theta_n \}$ typically converges to a (local) minimum of L. A variety of methods can be used to increase the likelihood of finding a global minimum, e.g., running SGD multiple times, starting from a set of randomly chosen initial solutions.

In practice, one often makes use of an additional projection Π_H that keeps the iterate in a given constraint set H. For example, there is considerable interest in nonnegative matrix factorizations [22], which corresponds to setting $H = \{ \theta : \theta \geq 0 \}$. The projected algorithm takes form

$$\theta_{n+1} = \Pi_H \left[\theta_n - \epsilon_n \hat{L}'(\theta_n) \right]. \tag{4}$$

In addition to the set of stationary points, the projected process may converge to the set of "chain recurrent" points [21], which are influenced by the boundary of the constraint set H.

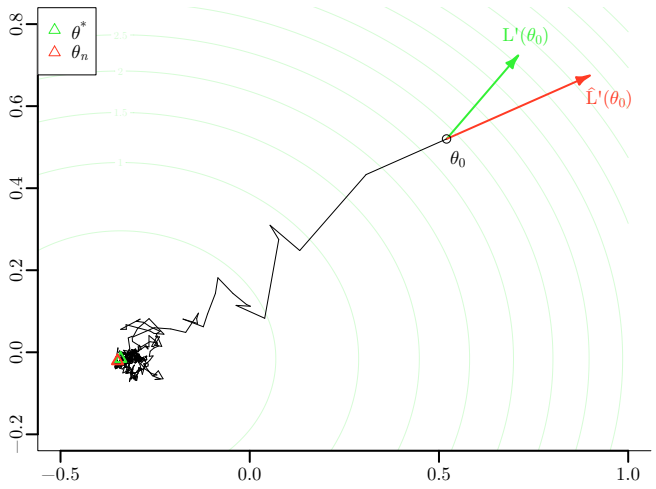


Figure 1: Example of SGD

3.2. SGD for Matrix Factorization

To apply SGD to matrix factorization, we take θ to be $(\boldsymbol{W}, \boldsymbol{H})$ and decompose the loss L as in (2) for an appropriate training set Z and local loss function l. For brevity, we suppress the constant matrix $\boldsymbol{V}$ in our notation. Denote by $L_z(\boldsymbol{W}, \boldsymbol{H}) = L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$ the local loss at position $z = (i, j)$. Then $L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{z \in Z} L_z(\boldsymbol{W}, \boldsymbol{H})$ and hence $L'(\boldsymbol{W}, \boldsymbol{H}) = \sum_{z \in Z} L'_z(\boldsymbol{W}, \boldsymbol{H})$ by the sum rule for differentiation. DGD methods exploit the summation form of L': they compute the local gradients L'_z in parallel and sum up. In contrast, SGD obtains noisy gradient estimates by scaling up *just one* of the local gradients, i.e.,

$$\hat{L}'(\boldsymbol{W}, \boldsymbol{H}) = N L'_z(\boldsymbol{W}, \boldsymbol{H}),$$

where $N = |Z|$ and the training point z is chosen randomly from the training set. Algorithm 1 uses SGD to perform matrix factorization.

Algorithm 1 SGD for Matrix Factorization

Require: A training set Z, initial values $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$
  while not converged **do** */* step */*
    Select a training point $(i, j) \in Z$ uniformly at random.
    $\boldsymbol{W}'_{i*} \leftarrow \boldsymbol{W}_{i*} - \epsilon_n N \frac{\partial}{\partial \boldsymbol{W}_{i*}} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$
    $\boldsymbol{H}_{*j} \leftarrow \boldsymbol{H}_{*j} - \epsilon_n N \frac{\partial}{\partial \boldsymbol{H}_{*j}} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$
    $\boldsymbol{W}_{i*} \leftarrow \boldsymbol{W}'_{i*}$
  end while

Note that, after generating a random training point $(i, j) \in Z$, we need to update only $\boldsymbol{W}_{i*}$ and $\boldsymbol{W}_{*j}$, and do not need to update factors of the form $\boldsymbol{W}_{i'*}$ for $i' \neq i$ or $\boldsymbol{H}_{*j'}$ for $j' \neq j$. This

computational savings follows from our representation of the global loss as a sum of local losses. Specifically, we have used the fact that

$$\frac{\partial}{\partial \boldsymbol{W}_{i'k}} L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = \begin{cases} 0 & \text{if } i \neq i' \\ \frac{\partial}{\partial \boldsymbol{W}_{ik}} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}) & \text{otherwise} \end{cases} \tag{5}$$

and

$$\frac{\partial}{\partial \boldsymbol{H}_{kj'}} L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = \begin{cases} 0 & \text{if } j \neq j' \\ \frac{\partial}{\partial \boldsymbol{H}_{kj}} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}) & \text{otherwise} \end{cases} \tag{6}$$

for $1 \leq k \leq r$. SGD is sometimes referred to as *online learning* or *sequential gradient descent* [6]. Batched versions, in which multiple local losses are averaged, are also feasible but often have inferior performance in practice.

One might wonder why replacing exact gradients (GD) by noisy estimates (SGD) can be beneficial. The main reason is that exact gradient computation is costly, whereas noisy estimates are quick and easy to obtain. In a given amount of time, we can perform many quick-and-dirty SGD updates instead of a few, carefully planned GD steps. The noisy process also helps in escaping local minima (especially those with a small basin of attraction and more so in the beginning, when the step sizes are large). Moreover, SGD is able to exploit repetition within the data. Parameter updates based on data from a certain row or column will also decrease the loss in similar rows and columns. Thus the more similarity there is, the better SGD performs. Ultimately, the hope is that the increased number of steps leads to faster convergence. This behavior can be proven for some problems [7], and it has been observed in the case of large-scale matrix factorization [20].

4. Stratified SGD

In this section we develop a general stratified stochastic gradient descent (SSGD) algorithm, and give sufficient conditions for convergence. In Sec. 5 we specialize SSGD to obtain an efficient distributed algorithm (DSGD) for matrix factorization.

4.1. The SSGD Algorithm

In SSGD, the loss function $L(\theta)$ is decomposed into a weighted sum of loss functions $L_s(\theta)$ as follows:

$$L(\theta) = w_1 L_1(\theta) + w_2 L_2(\theta) + \ldots + w_q L_q(\theta), \tag{7}$$

where we assume without loss of generality that $0 < w_s \leq 1$ and $\sum w_s = 1$. We refer to index s as a *stratum*, L_s as the *stratum loss* for stratum s, and w_s as the weight of stratum s. In practice, a stratum often corresponds to a part or partition of some underlying dataset. In this case, one can think of L_s as the loss incurred on the respective partition; the overall loss is obtained by summing up the per-partition losses. In general, however, the decomposition of L can be arbitrary; there may or may not be an underlying data partitioning. Also note that there is some freedom in the choice of

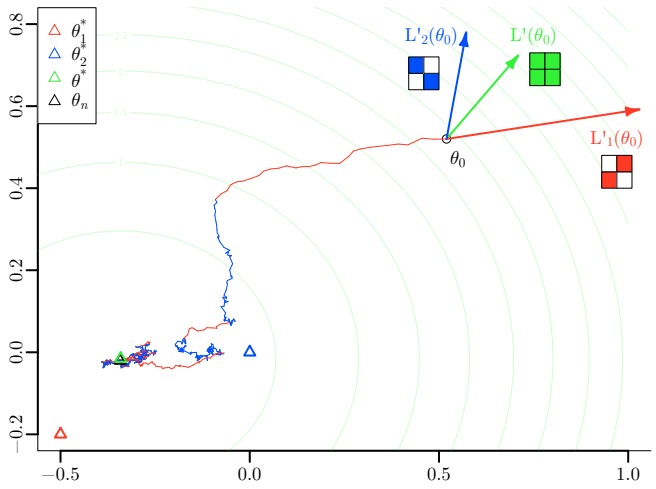


Figure 2: Example of stratified SGD

the w_s; they may be altered to arbitrary values (subject to the constraints above) by appropriately modifying the stratum loss functions. This freedom gives room for optimization.

SSGD runs standard stochastic gradient descent on a single stratum at a time, but switches strata in a way that guarantees correctness. The algorithm can be described as follows. Suppose that there is a (potentially random) *stratum sequence* $\{\gamma_n\}$, where each γ_n takes values in $\{1, \ldots, q\}$ and determines the stratum to use in the nth iteration. Using a noisy observation $\hat{L}'_{\gamma_n}$ of the gradient L'_{γ_n}, we obtain the update rule

$$\theta_{n+1} = \Pi_H\left[\theta_n - \epsilon_n \hat{L}'_{\gamma_n}(\theta_n)\right]. \tag{8}$$

The sequence $\{\gamma_n\}$ has to be chosen carefully to establish convergence to the stationary (or chain-recurrent) points of L. Indeed, because each step of the algorithm proceeds approximately in the "wrong" direction, i.e., $-L'_{\gamma_n}(\theta_n)$ rather than $-L'(\theta_n)$, it is not obvious that the algorithm will converge at all. We show in Sec. 4.2 and 4.3, however, that SSGD will indeed converge under appropriate regularity conditions provided that, in essence, the "time" spent on each stratum is proportional to its weight.

Figure 2 shows an example of SSGD, where the loss (shown in green) has been decomposed into two strata (blue and red). When $\gamma_n =$ red, the iterate is "pulled" towards the red optimum; when $\gamma_n =$ blue, it moves towards the blue optimum. In the long run, the process converges to the overall optimum.

4.2. Convergence of SSGD

Appropriate sufficient conditions for the convergence of SSGD can be obtained from general results on stochastic approximation in Kushner and Yin [21, Sec. 5.6]. These conditions are satisfied in most

matrix factorization problems. We distinguish step-size conditions, loss conditions, stratification conditions, and stratum-sequence conditions.

The two step-size conditions involve the sequence $\{ \epsilon_n \}$.

Condition 1. *The step sizes slowly approach zero in that* $\epsilon_n \to 0$ *and* $\sum \epsilon_n = \infty$.

Condition 2. *The step sizes decrease "quickly enough" in that* $\sum \epsilon_i^2 < \infty$

Clearly, the step sizes must decrease to 0 in order for the algorithm to converge (as specified in Condition 1). However, this convergence must occur at the correct rate. Condition 1 ensures that the SGD algorithm can move across arbitrarily large distances and thus cannot get stuck halfway to a stationary point; Condition 2 ("square summability") ensures that the step sizes decrease to 0 fast enough so that converge occurs. The simplest valid choice is $\epsilon_n = 1/n$.

The next pair of conditions involve the loss function.

Condition 3. *The constraint set* H *on which* L *is defined is a hyperrectangle.*

Condition 4. $L'(\theta)$ *is continuous on* H.

Note that non-differentiable points may arise in matrix factorization, e.g., when L_1 regularization is used. SSGD is powerful enough to deal with such points under appropriate regularity conditions, using "subgradients"; see [21] for details.

With respect to stratification, we require that estimates of the gradient defined with respect to a given stratum are unbiased, have bounded second moment for $\theta \in H$, and do not depend on the past. DSGD satisfies these conditions by design. More precisely, the conditions are as given below. Denote by $\hat{L}'_{\gamma_n,n}(\theta_n)$ the gradient estimate used in the nth step.

Condition 5. *The gradient estimates have bounded second moment, i.e.,*

$$\sup_n \mathrm{E}\left[|\hat{L}'_{\gamma_n,n}(\theta_n)|^2 \right] < \infty$$

for all $\theta \in H$.

This condition ensures that the noise in the gradient estimates is small enough to eventually be averaged out, and is often fulfilled by choosing the constraint set H appropriately (so that $L'_s(\theta)$ is bounded for all $s \in \{ 1, \ldots, q \}$ and $\theta \in H$).

Condition 6. *The noise in the gradient estimates is a martingale difference, i.e.,*

$$\mathrm{E}\left[\hat{L}'_{\gamma_n,n}(\theta_n) \mid \mathscr{F}_n \right] = L'_{\gamma_n}(\theta_n),$$

where $\hat{L}'_{\gamma_n,n}$ *is the gradient estimate in the* n*th step and* $\mathscr{F}_n = \sigma(\{ \epsilon_i, \theta_i, \gamma_i, i \leq n \})$ *is the* σ*-field that represents what is known at step* n.

Thus, accounting for the entire history, the gradient estimate is required to be unbiased for the gradient.

Finally, we give a sufficient condition on the stratum sequence.

Condition 7. *The step sizes satisfy* $(\epsilon_n - \epsilon_{n+1})/\epsilon_n = O(\epsilon_n)$ *and the* γ_n *are chosen such that the directions "average out correctly" in the sense that, for any* $\theta \in H$*,*

$$\lim_{n\to\infty} \epsilon_n \sum_{i=0}^{n-1} \left[L'_{\gamma_i}(\theta) - L'(\theta) \right] = 0$$

almost surely.

For example, if ϵ_n were equal to $1/n$, then the nth term would represent the empirical average deviation from the true gradient over the first n steps.

We can now state our correctness result, which asserts that, under the foregoing conditions, the θ_n sequence converges almost surely to the set of limit points of an ODE that is a smoothed version of the basic SGD recursion. As shown in [21], these limit points comprise the set of stationary points of L in H, as well as a set of chain-recurrent points on the boundary of H. In our setting, the limit point to which SSGD converges is typically a good local minimum.

Theorem 1. *Suppose that Conditions 1–7 hold. Then the sequence* $\{ \theta_n \}$ *converges almost surely to the set of limit points of the projected ODE*

$$\dot{\theta} = -L'(\theta) + z$$

in H*, taken over all initial conditions. Here,* z *is the "minimum force" to keep the solution in* H *[21, Sec. 4.3].*

The conditions used in Theorem 1 can be weakened considerably, but suffice for our purposes. The theorem follows directly from results in [21]. Indeed, as a special case of [21, Th. 6.1.1], the desired result follows from Conditions 1, 3, 4, and 5, and two "asymptotic rate of change" (ARC) assumptions: one on the gradient-estimation noise, given by (6.1.4) in [21], and one on the differences $\{L'_{\gamma_n}(\theta) - L'(\theta)\}_{n=0}^{\infty}$, given by (A6.1.3) in [21]. As discussed on pp. 137–138 of [21], the first ARC assumption is implied by Conditions 2, 5, and 6. The second ARC assumption is implied by Condition 7; see [21, p. 171].

All but the last of the sufficient conditions for convergence hold by design. Therefore, the crux of showing that SSGD converges is showing that Condition 7 holds. We address this issue next.

4.3. Conditions for Stratum Selection

The following result gives sufficient conditions on $L(\theta)$, the step sizes $\{ \epsilon_n \}$, and the stratum sequence $\{ \gamma_n \}$ such that Condition 7 holds. Our key assumption is that the sequence $\{ \gamma_n \}$ is *regenerative* [3, Ch. VI], in that there exists an increasing sequence of almost-surely finite random indices $0 = \beta(0) < \beta(1) < \beta(2) < \cdots$ that serves to decompose $\{ \gamma_n \}$ into consecutive, independent and identically distributed (i.i.d.) cycles $\{ C_k \}$,[2] with $C_k = \{ \gamma_{\beta(k-1)}, \gamma_{\beta(k-1)+1},$

[2] The cycles need not directly correspond to strata. Indeed, we make use of strategies in which a cycle comprises multiple strata.

$\ldots, \gamma_{\beta(k)-1}\}$ for $k \geq 1$. I.e., at each $\beta(i)$, the stratum is selected according to a probability distribution that is independent of past selections, and the future sequence of selections after step $\beta(i)$ looks probabilistically identical to the sequence of selections after step $\beta(0)$. The length τ_k of the kth cycle is given by $\tau_k = \beta(k) - \beta(k-1)$. Letting $I_{\gamma_n=s}$ be the indicator variable for the event that stratum s is chosen in the nth step, set

$$X_k(s) = \sum_{n=\beta(k-1)}^{\beta(k)-1} (I_{\gamma_n=s} - w_s)$$

for $1 \leq s \leq q$. It follows from the regenerative property that the pairs $\{ (X_k(s), \tau_k) \}$ are i.i.d. for each s. The following theorem asserts that, under regularity conditions, we may pick any regenerative sequence γ_n such that $\mathrm{E}[X_1(s)] = 0$ for all strata.

Theorem 2. *Suppose that $L(\theta)$ is differentiable on H and $\sup_{\theta \in H} |L'_s(\theta)| < \infty$ for $1 \leq s \leq q$ and $\theta \in H$. Also suppose that $\epsilon_n = O(n^{-\alpha})$ for some $\alpha \in (0.5, 1]$ and that $(\epsilon_n - \epsilon_{n+1})/\epsilon_n = O(\epsilon_n)$. Finally, suppose that $\{\gamma_n\}$ is regenerative with $\mathrm{E}[\tau_1^{1/\alpha}] < \infty$ and $\mathrm{E}[X_1(s)] = 0$ for $1 \leq s \leq q$. Then Condition 7 holds.*

The condition $\mathrm{E}[X_1(s)] = 0$ essentially requires that, for each stratum s, the expected fraction of visits to s in a cycle equals w_s. By the strong law of large numbers for regenerative processes [3, Sec. VI.3], this condition—in the presence of the finite-moment condition on τ_1—also implies that the long-term fraction of visits to s equals w_s. The finite-moment condition is typically satisfied whenever the number of successive steps taken within a stratum is bounded with probability 1.

Proof. Fix $\theta \in H$ and observe that

$$\begin{aligned}\epsilon_n \sum_{i=0}^{n-1} \big(L'_{\gamma_i}(\theta) - L'(\theta)\big) &= \epsilon_n \sum_{i=0}^{n-1} \sum_{s=1}^{q} \big(L'_s(\theta) I_{\gamma_i=s} - L'_s(\theta) w_s\big) \\ &= \sum_{s=1}^{q} L'_s(\theta) \epsilon_n \sum_{i=0}^{n-1} \big(I_{\gamma_i=s} - w_s\big).\end{aligned}$$

Since $|L'_s(\theta)| < \infty$ for each s, it suffices to show that $n^{-\alpha} \sum_{i=0}^{n-1} \big(I_{\gamma_i=s} - w_s\big) \xrightarrow{\text{a.s.}} 0$ for $1 \leq s \leq q$. To this end, fix s and denote by $c(n)$ the (random) number of complete cycles up to step n. We have

$$\sum_{i=0}^{n} (I_{\gamma_i=s} - w_s) = \sum_{k=1}^{c(n)} X_k(s) + R_{1,n},$$

where $R_{1,n} = \sum_{i=\beta(c(n))}^{n} (I_{\gamma_i=s} - w_s)$. I.e., the sum can be broken up into sums over complete cycles plus a remainder term corresponding to a sum over a partially completed cycle. Similar

calculations let us write $n = \sum_{k=1}^{c(n)} \tau_k + R_{2,n}$, where $R_{2,n} = n - \beta\big(c(n)\big) + 1$. Thus

$$\begin{aligned}\frac{\sum_{i=0}^{n}(I_{\gamma_i=s} - w_s)}{n^\alpha} &= \frac{\sum_{k=1}^{c(n)} X_k(s) + R_{1,n}}{\left(\sum_{k=1}^{c(n)} \tau_k + R_{2,n}\right)^\alpha} \\ &= \frac{\sum_{k=1}^{c(n)} X_k(s)}{c(n)^\alpha} \left(\frac{\sum_{k=1}^{c(n)} \tau_k}{c(n)} + \frac{R_{2,n}}{c(n)}\right)^{-\alpha} \\ &\quad + \frac{R_{1,n}/c(n)^\alpha}{\left(\sum_{k=1}^{c(n)} \tau_k/c(n) + R_{2,n}/c(n)\right)^\alpha}.\end{aligned} \tag{9}$$

By assumption, the random variables $\{ X_k(s) \}$ are i.i.d. with common mean 0. Moreover, $|X_k(s)| \leq (1 + w_s)\tau_k$, which implies that $\mathrm{E}[\,|X_1(s)|^{1/\alpha}\,] \leq (1 + w_s)^{1/\alpha}\,\mathrm{E}[\,\tau_1^{1/\alpha}\,] < \infty$. It then follows from the Marcinkiewicz-Zygmund strong law [9, Th. 5.2.2] that $n^{-\alpha}\sum_{k=1}^{n} X_k(s) \xrightarrow{\text{a.s.}} 0$. Because each regeneration point, and hence each cycle length, is assumed to be almost surely finite, it follows that $c(n) \xrightarrow{\text{a.s.}} \infty$, so that $\sum_{k=1}^{c(n)} X_k(s)/c(n)^\alpha \xrightarrow{\text{a.s.}} 0$ as $n \to \infty$. Similarly, an application of the ordinary strong law of large numbers shows that $\sum_{k=1}^{c(n)} \tau_k/c(n) \xrightarrow{\text{a.s.}} \mathrm{E}[\,\tau_1\,] > 0$. Next, note that $|R_{1,n}| \leq (1 + w_s)\tau_{c(n)+1}$, so that $R_{1,n}/c(n)^\alpha \xrightarrow{\text{a.s.}} 0$ provided that $\tau_k/k^\alpha \xrightarrow{\text{a.s.}} 0$. To establish this latter limit result, observe that for any $\epsilon > 0$, the assumed finiteness of $\mathrm{E}[\,(\tau_k/\epsilon)^{1/\alpha}\,]$ implies [10, Th. 3.2.1] that $\sum_{k=1}^{\infty} \Pr[\,(\tau_k/\epsilon)^{1/\alpha} \geq k\,] < \infty$, and hence $\sum_{k=1}^{\infty} \Pr[\,\tau_k/k^\alpha \geq \epsilon\,] < \infty$. It then follows from the first Borel-Cantelli Lemma (see [10, Th. 4.2.1]) that $\Pr[\,\tau_k/k^\alpha \geq \epsilon \text{ infinitely often}\,] = 0$, which in turn implies [10, Th. 4.2.2] that $\tau_k/k^\alpha \xrightarrow{\text{a.s.}} 0$. A similar argument shows that $R_{2,n}/c(n) \xrightarrow{\text{a.s.}} 0$, and the desired result follows after letting $n \to \infty$ in the rightmost expression in (9). □

The conditions on $\{ \epsilon_n \}$ in Theorem 2 are often satisfied in practice, e.g., when $\epsilon_n = 1/n$ or when $\epsilon_n = 1/\lceil n/k \rceil$ for some $k > 1$ with $\lceil x \rceil$ denoting the smallest integer greater than or equal to x (so that the step size remains constant for some fixed number of steps, as in Algorithm 2 below). See Sec. 6 for further discussion.

Similarly, a wide variety of strata-selection schemes satisfy the conditions of Theorem 2. Some simple examples include (1) running precisely cw_s steps on stratum s in every "chunk" of c steps, and (2) repeatedly picking a stratum according to some fixed distribution $\{ p_s > 0 \}$ and running cw_s/p_s steps on the selected stratum s. (E.g., we can set $p_s = w_s$ so that strata are chosen proportional to their weight and a constant number of steps is run on the selected stratum, or $p_s = 1/q$ so that strata are chosen uniformly and at random but the number of steps run on the selected stratum is proportional to its weight.) For example (1), assume initially that the order of stratum visits within any two chunks is the same. Then, in the notation of Theorem 2, we have $\tau_k = c$ and $X_k(s) = 0$ with probability 1 for all k, so the conditions of the theorem hold trivially, with the chunks playing the role of regenerative cycles. In fact, the order within each chunk is irrelevant, since for any ordering the pairs $\{ \big(Y_k(s), \tau_k\big) \}$ are trivially i.i.d., so that the proof of the theorem goes through essentially unchanged. For example (2), the steps at which a stratum is randomly selected clearly

form a sequence of regeneration points for $\{\gamma_n\}$. We have

$$\mathrm{E}[\tau_1] = \sum_{s=1}^{q} p_s \frac{cw_s}{p_s} = c\sum_{s=1}^{q} w_s = c$$

The sum of the random variables $I_{\gamma_n=s}$ over a cycle is cw_s/p_s if stratum s is selected and 0 otherwise, so that the expected sum is $p_s(cw_s/p_s) + (1-p_s)0 = cw_s$ and hence

$$\begin{aligned}\mathrm{E}[X_1(s)] &= \mathrm{E}\left[\sum_{\gamma_n \in \text{cycle 1}} (I_{\gamma_n=s} - w_s)\right] \\ &= \mathrm{E}\left[\sum_{\gamma_n \in \text{cycle 1}} I_{\gamma_n=s}\right] - \mathrm{E}[w_s\tau_k] = cw_s - cw_s = 0.\end{aligned}$$

Moreover, τ_1 is bounded above by $\max_s cw_s/p_s < \infty$, and hence has finite moments of all orders.

To give a better idea of the scope of stratum-selection schemes covered by Theorem 2, we discuss a randomized procedure in which, after visiting a stratum s, we visit the same stratum at the next step with probability $p_s = 1-(cw_s)^{-1}$, and with probability $1-p_s$ we select a new stratum randomly and uniformly from the q strata. (Thus the new stratum may correspond to the old stratum.) Here c is a constant that is large enough to ensure that $cw_s > 1$ for each s. Denote by $s_0 \in \{1,2,\ldots,q\}$ the initial stratum to be visited; we fix s_0 a priori. Then $\{\gamma_n\}$ is regenerative, with the regeneration points corresponding to the successive steps at which a new stratum is selected and the new stratum is s_0. We can write $\tau_1 = \sum_{i=1}^{N} V_i$, where N is the total number of new-stratum selections, and V_i is the number of successive visits to the ith selected stratum. The random variable N has a geometric distribution with mean q and, given that stratum s is selected at the ith selection epoch, V_i has a geometric distribution with mean cw_s. Moreover, the V_i's are mutually independent and independent of N, and $V_2,\ldots,V_N$ are i.i.d., specifically, each V_i ($i>1$) is distributed as an average of $q-1$ independent geometric random variables with means $\{w_s : s \neq s_0\}$. It follows that

$$\mathrm{E}[\tau_1] = \mathrm{E}[V_1] + \mathrm{E}\left[\sum_{i=2}^{N} V_i\right] = cw_{s_0} + \mathrm{E}[N-1]\left((q-1)^{-1}\sum_{s\neq s_0} cw_s\right) = \sum_s cw_s = c.$$

Let N_s denote the number of new-stratum selections equal to s in the first cycle. It is not hard to see that $N_{s_0} = 1$ with probability 1 and that $\mathrm{E}[N_s] = 1$ for $s \neq s_0$. Thus, for each s,

$$\mathrm{E}\left[\sum_{\gamma_n \in \text{cycle 1}} I_{\gamma_n=s}\right] = cw_s\,\mathrm{E}[N_s] = cw_s,$$

so that $\mathrm{E}[X_1(s)] = 0$. Finally, we show that τ_1 has finite moments of all orders. Using the simple bound $(\sum_{i=1}^n x_i)^\beta \le (n\max_{1\le i\le n} x_i)^\beta \le n^\beta \sum_{i=1}^n x_i^\beta$ for $x_1,x_2,\ldots,x_n \ge 0$ and $\beta \ge 0$, we have $\mathrm{E}[\tau_1^\beta] \le \mathrm{E}[N^\beta]m(\beta,s_0) + \mathrm{E}[N^\beta(N-1)]\sum_{s\neq s_0} m(\beta,s)/(q-1)$. Here $m(\beta,s)$ denotes the (finite) βth moment of a geometric random variable with mean cw_s. The desired result then

follows from the fact that N also has a geometric distribution, and hence has finite moments of all orders.

The above examples are primarily of theoretical interest. In Sec. 6, we focus on some schemes that are particularly suitable for practical implementation in the context of DSGD.

We conclude by noting that, if L is well behaved, Liapunov-function arguments can be used to show that, for any sufficiently large hyperrectangle H, the θ_n sequence will fall within H infinitely often with probability 1, so that the foregoing arguments apply. Moreover, if L is in fact convex, then the global minimum is the unique limit point; see [15].

5. The DSGD Algorithm

We can exploit the structure of the matrix factorization problem to derive a distributed algorithm for rank-r matrix factorization via SGD. The idea is to specialize the SSGD algorithm, choosing the strata such that SGD can be run on each stratum in a distributed manner. We first discuss the "interchangeability" structure that we will exploit for distributed processing within a stratum.

5.1. Interchangeability

In general, distributing SGD is hard because the individual steps depend on each other: from (4), we see that θ_n has to be known before θ_{n+1} can be computed. However, in the case of matrix factorization, the SGD process has some structure that we can exploit.

We focus on loss-minimization problems of the form $\text{minimize}_{\theta \in H}\, L(\theta)$ where the loss function L has *summation form*: $L(\theta) = \sum_{z \in Z} L_z(\theta)$.

Definition 1. *Two training points $z_1, z_2 \in Z$ are* interchangeable *if for all loss functions L having summation form, all $\theta \in H$, and $\epsilon > 0$,*

$$\begin{aligned} & L'_{z_1}(\theta) = L'_{z_1}(\theta - \epsilon L'_{z_2}(\theta)) \\ \textit{and} \qquad & L'_{z_2}(\theta) = L'_{z_2}(\theta - \epsilon L'_{z_1}(\theta)). \end{aligned} \tag{10}$$

Two disjoint sets of training points $Z_1, Z_2 \subset Z$ are interchangeable if z_1 and z_2 are interchangeable for every $z_1 \in Z_1$ and $z_2 \in Z_2$.

As described in Sec. 5.2 below, we can swap the order of consecutive SGD steps that involve interchangeable training points without affecting the final outcome.

Now we return to the setting of matrix factorization, where the loss function has the form $L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{(i,j) \in Z} L_{ij}(\boldsymbol{W}, \boldsymbol{H})$ with $L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$. The following theorem gives a simple criterion for interchangeability.

Theorem 3. *Two training points $z_1 = (i_1, j_1) \in Z$ and $z_2 = (i_2, j_2) \in Z$ are interchangeable if they share neither row nor column, i.e., $i_1 \neq i_2$ and $j_1 \neq j_2$.*

Proof. The result is a direct consequence of the decomposition of the global loss into a sum of local losses. Specifically, it follows from (5) and (5) that the partial derivatives of L_{ij} (1) depend only on $\boldsymbol{V}_{ij}$, $\boldsymbol{W}_{i*}$, and $\boldsymbol{H}_{*j}$, and (2) are nonzero only with respect to $\boldsymbol{W}_{i1}, \ldots, \boldsymbol{W}_{ir}, \boldsymbol{H}_{1j}, \ldots, \boldsymbol{H}_{rj}$. When $i_1 \neq i_2$ and $j_1 \neq j_2$, both $(\boldsymbol{W}, \boldsymbol{H})$ and $(\boldsymbol{W}, \boldsymbol{H}) - \epsilon L'_{z_1}(\boldsymbol{W}, \boldsymbol{H})$ agree on the values of $\boldsymbol{W}_{i_2*}$ and $\boldsymbol{H}_{*j_2}$ for any choice of $(\boldsymbol{W}, \boldsymbol{H})$, which establishes the second part of (10). Analogous arguments hold for the first part. □

It follows that if two blocks of $\boldsymbol{V}$ share neither rows or columns, then the sets of training points contained in these blocks are interchangeable.

5.2. A Simple Case

We introduce the DSGD algorithm by considering a simple case that essentially corresponds to running DSGD using a single "d-monomial" stratum (see Sec. 5.3). The goal is to highlight the technique by which DSGD runs the SGD algorithm in a distributed manner within a stratum. For a given training set Z, denote by $\boldsymbol{Z}$ the corresponding *training matrix*, which is obtained by zeroing out the elements in $\boldsymbol{V}$ that are not in Z; these elements usually represent missing data or held-out data for validation. In our simple scenario, Z corresponds to our single stratum of interest, and the corresponding training matrix $\boldsymbol{Z}$ is block-diagonal:

$$\begin{array}{c} \\ \boldsymbol{W}^1 \\ \boldsymbol{W}^2 \\ \vdots \\ \boldsymbol{W}^d \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{H}^1 & \boldsymbol{H}^2 & \cdots & \boldsymbol{H}^d \end{array} \\ \begin{pmatrix} \boldsymbol{Z}^1 & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{0} & \boldsymbol{Z}^2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \mathbf{0} \\ \mathbf{0} & \cdots & \mathbf{0} & \boldsymbol{Z}^d \end{pmatrix} \end{array}, \tag{11}$$

where $\boldsymbol{W}$ and $\boldsymbol{H}$ are blocked conformingly. Denote by Z^b the set of training points in block $\boldsymbol{Z}^b$. We exploit the key property that, by Theorem 3, sets Z^i and Z^j are interchangeable for $i \neq j$. For some $T \in [1, \infty)$, suppose that we run T steps of SGD on $\boldsymbol{Z}$, starting from some initial point $\theta_0 = (\boldsymbol{W}_0, \boldsymbol{H}_0)$ and using a fixed step size ϵ. We can describe an instance of the SGD process by a *training sequence* $\omega = (z_0, z_1, \ldots, z_{T-1})$ of T training points. Figure 1 shows an example of such a training sequence. Define $\theta_0(\omega) = \theta_0$ and

$$\theta_{n+1}(\omega) = \theta_n(\omega) + \epsilon Y_n(\omega),$$

where the update term $Y_n(\omega) = -NL'_{\omega_n}(\theta_n(\omega))$ is the scaled negative gradient estimate as in standard SGD. We can write

$$\theta_T(\omega) = \theta_0 + \epsilon \sum_{n=0}^{T-1} Y_n(\omega). \tag{12}$$

To see how to exploit the interchangeability structure, consider the subsequence $\sigma_b(\omega) = \omega \cap Z^b$ of training points from block $\boldsymbol{Z}^b$; the subsequence has length $T_b(\omega) = |\sigma_b(\omega)|$. The following theorem asserts that we can run SGD on each block independently, and then sum up the results.

Theorem 4. *Using the definitions above,*

$$\theta_T(\omega) = \theta_0 + \epsilon \sum_{b=1}^{d} \sum_{k=0}^{T_b(\omega)-1} Y_k(\sigma_b(\omega)). \tag{13}$$

Proof. We establish a one-to-one correspondence between the update terms $Y_n(\omega)$ in (12) and $Y_k(\sigma_b(\omega))$ in (13). Denote by $z_{b,k}$ the $(k+1)$st element in $\sigma_b(\omega)$, i.e., the $(k+1)$st element from block $\boldsymbol{Z}^b$ in ω. Denote by $\pi(z_{b,k})$ the 0-based position of this element in ω. We have $\omega_{\pi(z_{b,k})} = z_{b,k}$. Now consider the first element $z_{b,0}$ from block b. We have $z_n \notin Z^b$ for all previous elements $n < \pi(z_{b,0})$. Since the training matrix is block-diagonal, blocks have pairwise disjoint rows and pairwise disjoint columns. Thus by Theorem 3, $z_{b,0}$ is interchangeable with each of the z_n for $n < \pi(z_{b,0})$. We can therefore eliminate the z_n one by one:

$$\begin{aligned} Y_{\pi(z_{b,0})}(\omega) &= -NL'_{z_{b,0}}(\theta_{\pi(z_{b,0})}(\omega)) = -NL'_{z_{b,0}}(\theta_{\pi(z_{b,0})-1}(\omega)) \\ &= \cdots = -NL'_{z_{b,0}}(\theta_0) = Y_0(\sigma_b(\omega)). \end{aligned}$$

Using the same argument, we can safely remove *all* update terms from elements not in block $\boldsymbol{Z}^b$. By induction on k, we obtain

$$\begin{aligned} Y_{\pi(z_{b,k})}(\omega) &= -NL'_{z_{b,k}} \left(\theta_0 + \epsilon \sum_{n=0}^{\pi(z_{b,k})-1} Y_n(\omega) \right) \\ &= -NL'_{z_{b,k}} \left(\theta_0 + \epsilon \sum_{l=0}^{k-1} Y_{\pi(z_{b,l})}(\sigma_b(\omega)) \right) \\ &= Y_k(\sigma_b(\omega)). \end{aligned} \tag{14}$$

The assertion of the theorem now follows from

$$\begin{aligned} \theta_T(\omega) &= \theta_0 + \epsilon \sum_{n=0}^{T-1} Y_n(\omega) = \theta_0 + \epsilon \sum_{b=1}^{d} \sum_{k=0}^{T_b(\omega)-1} Y_{\pi(z_{b,k})}(\omega) \\ &= \theta_0 + \epsilon \sum_{b=1}^{d} \sum_{k=0}^{T_b(\omega)-1} Y_k(\sigma_b(\omega)), \end{aligned}$$

where we first reordered the update terms and then used (14). □

We used the fact that $\boldsymbol{Z}$ is block-diagonal only to establish interchangeability between blocks. This means that Theorem 4 also applies when the matrix is not block-diagonal, but can be divided into a set of interchangeable submatrices in some other way.

We now describe how to exploit Theorem 4 for distributed processing. We block $\boldsymbol{W}$ and $\boldsymbol{H}$ conformingly to $\boldsymbol{Z}$—as in (11)—and divide processing into d independent tasks $\Gamma_1, \ldots, \Gamma_d$ as

follows. Task Γ_b is responsible for subsequence $\sigma_b(\omega)$: It takes $\boldsymbol{Z}^b$, $\boldsymbol{W}^b$, and $\boldsymbol{H}^b$ as input, performs the block-local updates $\sigma_b(\omega)$, and outputs updated factor matrices $\boldsymbol{W}^b_{\text{new}}$ and $\boldsymbol{H}^b_{\text{new}}$.[3] By Theorem 4, we have

$$\boldsymbol{W}' = \begin{pmatrix} \boldsymbol{W}^1_{\text{new}} \\ \vdots \\ \boldsymbol{W}^d_{\text{new}} \end{pmatrix} \qquad \text{and} \qquad \boldsymbol{H}' = \begin{pmatrix} \boldsymbol{H}^1_{\text{new}} & \cdots & \boldsymbol{H}^d_{\text{new}} \end{pmatrix},$$

where $\boldsymbol{W}'$ and $\boldsymbol{H}'$ are the matrices that one would obtain by running sequential SGD on ω. Since each task accesses different parts of both training data and factor matrices, the data can be distributed across multiple nodes and the tasks can run simultaneously. In Sec. 6, we describe how to efficiently implement the above idea.

5.3. The General Case

We now present the complete DSGD matrix-factorization algorithm. The key idea is to stratify the training set Z into a set $S = \{ Z_1, \ldots, Z_q \}$ of q strata so that each individual stratum $Z_s \subseteq Z$ can be processed in a distributed fashion. We do this by ensuring that each stratum is "d-monomial" as defined below. The d-monomial property generalizes the block-diagonal structure of the example in Sec. 5.2, while still permitting the techniques of that section to be applied. The strata must *cover* the training set in that $\bigcup_{s=1}^{q} Z_s = Z$, but overlapping strata are allowed. The parallelism parameter d is chosen to be greater than or equal to the number of available processing tasks.

Definition 2. *A stratum Z_s is* d-monomial *if it can be partitioned into d nonempty subsets $Z_s^1, Z_s^2, \ldots, Z_s^d$ such that $i \neq i'$ and $j \neq j'$ whenever $(i,j) \in Z_s^{b_1}$ and $(i',j') \in Z_s^{b_2}$ with $b_1 \neq b_2$. A training matrix $\boldsymbol{Z}_s$ is* d-monomial *if it is constructed from a d-monomial stratum Z_s.*

There are many ways to stratify the training set according to Def. 2. In our current work, we perform *data-independent blocking*; more advanced strategies may improve the speed of convergence further. We first randomly permute the rows and column of $\boldsymbol{Z}$, and then create $d \times d$ blocks of size $(m/d) \times (n/d)$ each; the factor matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ are blocked conformingly. This procedure ensures that the expected number of training points in each of the blocks is the same, namely, N/d^2. Then, for a permutation $j_1, j_2, \ldots, j_d$ of $1, 2, \ldots, d$, we can define a stratum as $Z_s = Z^{1j_1} \cup Z^{2j_2} \cup \cdots \cup Z^{dj_d}$, where the *substratum* Z^{ij} denotes the set of training points that fall within block $\boldsymbol{Z}^{ij}$. We can represent a stratum Z_s by a template $\tilde{\boldsymbol{Z}}_s$ that displays each block $\boldsymbol{Z}^{ij}$ corresponding to a substratum Z^{ij} of Z_s, with all other blocks represented by zero matrices. When $d = 2$, for example, we obtain two strata represented by the templates

$$\tilde{\boldsymbol{Z}}_1 = \begin{pmatrix} \boldsymbol{Z}^{11} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Z}^{22} \end{pmatrix} \quad \text{and} \quad \tilde{\boldsymbol{Z}}_2 = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{Z}^{12} \\ \boldsymbol{Z}^{21} & \boldsymbol{0} \end{pmatrix}.$$

The set S of possible strata contains $d!$ elements, one for each possible permutation of $1, 2, \ldots, d$. Note that different strata may overlap when $d > 2$. Also note that there is no need to materialize these strata: They are constructed on-the-fly by processing only the respective blocks of $\boldsymbol{Z}$.

[3] Since training data is sparse, a block $\boldsymbol{Z}^b$ may contain no training points; in this case we cannot execute SGD on the block, so the corresponding factors simply remain at their initial values.

Given a set of strata and associated weights $\{ w_s \}$, we decompose the loss into a weighted sum of per-stratum losses as in (7): $L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{s=1}^{q} w_s L_s(\boldsymbol{W}, \boldsymbol{H})$. (As in Sec. 3.2, we suppress the fixed matrix $\boldsymbol{V}$ in our notation for loss functions.) We use per-stratum losses of form

$$L_s(\boldsymbol{W}, \boldsymbol{H}) = c_s \sum_{(i,j) \in Z_s} L_{ij}(\boldsymbol{W}, \boldsymbol{H}), \tag{15}$$

where c_s is a stratum-specific constant; see the discussion below. When running SGD on a stratum, we use the gradient estimate

$$\hat{L}'_s(\boldsymbol{W}, \boldsymbol{H}) = N_s c_s L'_{ij}(\boldsymbol{W}, \boldsymbol{H}) \tag{16}$$

of $L'_s(\boldsymbol{W}, \boldsymbol{H})$ in each step, i.e., we scale up the local loss of an individual training point by the size $N_s = |Z_s|$ of the stratum. For example, from the $d!$ strata described previously, we can select d disjoint strata $Z_1, Z_2, \ldots, Z_d$ such that they cover Z. Then any given loss function L of the form (2) can be represented as a weighted sum over these strata by choosing w_s and c_s subject to $w_s c_s = 1$. Recall that w_s can be interpreted as the "time" spent on each stratum in the long run. A natural choice is to set $w_s = N_s/N$, i.e., proportional to the stratum size. This particular choice leads to $c_s = N/N_s$ and we obtain the standard SGD gradient estimator $\hat{L}'_s(\boldsymbol{W}, \boldsymbol{H}) = N L'_{ij}(\boldsymbol{W}, \boldsymbol{H})$. As another example, we can represent L as a weighted sum in terms of all $d!$ strata; in light of the fact that each substratum Z^{ij} lies in exactly $(d-1)!$ of these strata, we choose $w_s = N_s / \big((d-1)! N\big)$ and use the value of $c_s = N/N_s$ as before.

The individual steps in DSGD are grouped into *subepochs*, each of which amounts to processing one of the strata. In more detail, DSGD makes use of a sequence $\{ (\xi_k, T_k) \}$, where ξ_k denotes the *stratum selector* used in the kth subepoch, and T_k the number of steps to run on the selected stratum. Note that this sequence of pairs uniquely determines an SSGD stratum sequence as in Sec. 4.1: $\gamma_1 = \cdots = \gamma_{T_1} = \xi_1$, $\gamma_{T_1+1} = \cdots = \gamma_{T_1+T_2} = \xi_2$, and so forth. The $\{ (\xi_k, T_k) \}$ sequence is chosen such that the underlying SSGD algorithm, and hence the DSGD factorization algorithm, is guaranteed to converge; see Sec. 4.3. Once a stratum ξ_k has been selected, we perform T_k SGD steps on Z_{ξ_k}; this is done in a parallel and distributed way using the technique of Sec. 5.2. DSGD is shown as Algorithm 2, where we define an *epoch* as a sequence of d subepochs. As will become evident in Sec. 6 below, an epoch roughly corresponds to processing the entire training set once.

When executing Algorithm 2 on d nodes in a shared-nothing environment such as MapReduce, the input matrix need only be distributed once. Then the only data that are transmitted between nodes during subsequent processing are (small) blocks of factor matrices. Indeed, if node i stores blocks $\boldsymbol{W}^i, \boldsymbol{Z}^{i1}, \boldsymbol{Z}^{i2}, \ldots, \boldsymbol{Z}^{id}$ for $1 \le i \le d$, then only matrices $\boldsymbol{H}^1, \boldsymbol{H}^2, \ldots, \boldsymbol{H}^d$ need be transmitted. (If the $\boldsymbol{W}^i$ matrices are smaller, then we transmit these instead.)

Since, by construction, parallel processing within the kth selected stratum leads to the same update terms as for the corresponding sequential SGD algorithm on Z_{ξ_k}, we have established the connection between DSGD and SSGD. Thus the convergence of DSGD is implied by the convergence of the underlying SSGD algorithm; see Sec. 4.2.

Algorithm 2 DSGD for Matrix Factorization

Require: $\boldsymbol{Z}$, $\boldsymbol{W}_0$, $\boldsymbol{H}_0$, cluster size d
  $\boldsymbol{W} \leftarrow \boldsymbol{W}_0$
  $\boldsymbol{H} \leftarrow \boldsymbol{H}_0$
  Block $\boldsymbol{Z}$ / $\boldsymbol{W}$ / $\boldsymbol{H}$ into $d \times d$ / $d \times 1$ / $1 \times d$ blocks
  while not converged **do** /* *epoch* */
    Pick step size ϵ
    for $s = 1, \ldots, d$ **do** /* *subepoch* */
      Pick d blocks $\{\boldsymbol{Z}^{1j_1}, \ldots, \boldsymbol{Z}^{dj_d}\}$ to form a stratum
      for $b = 1, \ldots, d$ **do** /* *in parallel* */
        Run SGD on the training points in $\boldsymbol{Z}^{bj_b}$ (step size $= \epsilon$)
      end for
    end for
  end while

6. DSGD Implementation

In this section, we discuss some practical issues around DSGD. We first discuss some general algorithmic details, including initialization considerations and practical methods for choosing the training sequence for the parallel SGD step, selecting strata, and picking the step size ϵ. We then discuss issues specific to MapReduce platforms, specifically, Hadoop.

6.1. General Algorithmic Details

As above, a "subepoch" corresponds to processing a stratum and an "epoch"—roughly equivalent to a complete pass through the training data—corresponds to processing a sequence of d strata.

Initialization. Some care must be taken when choosing initial factor values $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$. In the case of nonzero squared loss L_{NZSL}, for example, choosing $\boldsymbol{W}_0 = \boldsymbol{0}$ and $\boldsymbol{H}_0 = \boldsymbol{0}$ results in the factors remaining equal to zero at all future DSGD iterations. For GKL loss, we cannot have $\boldsymbol{W}_{i*} = \boldsymbol{0}$ for any i or $\boldsymbol{H}_{*j}$ for any j, since then the loss function is ill defined. In our implementation, we generate initial factor values using a pseudorandom number generator, which ensures that all initial values are nonzero.

Training sequence. When processing a subepoch (i.e., a stratum), we do not generate a global training sequence and then distribute it among blocks. Instead, each task generates a *local* training sequence directly for its corresponding block. This reduces communication cost and avoids the bottleneck of centralized computation. Practical experience suggests that good results are achieved when (1) the local training sequence covers a large part of the local block, and (2) the training sequence is randomized. We consider the following strategies for processing block $\boldsymbol{Z}^{ij}$:

- *Sequential selection (SEQ).* Scan $\boldsymbol{Z}^{ij}$ in the order it is stored.
- *With-replacement selection (WR).* Randomly select training points from $\boldsymbol{Z}^{ij}$; each point may be selected multiple times.

- *Without-replacement selection (WOR).* Randomly select training points from $\boldsymbol{Z}^{ij}$ such that each point is selected precisely once; i.e., generate an ordering of the points randomly and uniformly from the set of all such orderings, and select points according to this ordering.

The first two strategies are extremes: SEQ satisfies (1) but not (2), whereas WR satisfies (2) but not (1). A compromise—which worked best in our experiments—is WOR; it ensures that many different training points are selected while at the same time maximizing randomness. Note that Theorem 2 implicitly assumes WR but can be extended to cover SEQ and WOR as well. (In brief, redefine a stratum to consist of a single training point and redefine the stratum weights w_s accordingly.)

Update terms. When processing a training point (i, j) during an SGD step on stratum s, we use the gradient estimate $\hat{L}'_s(\theta) = NL'_{ij}(\theta)$ as in standard SGD; this corresponds to a choice of $c_s = N/N_s$ in (16). For (i, j) picked uniformly and at random from Z_s, the estimate is unbiased for the gradient of the stratum loss $L_s(\theta)$ given in (15).

Stratum selection. Recall that the stratum sequence (ξ_k, T_k) determines which of the strata is chosen in each subepoch and how many steps are run on that stratum. We choose training sequences such that $T_k = N_{\xi_k} = |Z_{\xi_k}|$; this ensures that we can make use of all the training points in the stratum. For the data-independent blocking scheme given in Sec. 5.3, each block $\boldsymbol{Z}^{ij}$ occurs in $(d-1)!$ of the $d!$ strata. Thus we do not need to process all strata to cover the entire training set. As above, we want to process a large part of the training set in each epoch, while at the same time maximizing randomization. To select a set of d strata to visit during an epoch, we use strategies similar to those for intra-block training point selection:

- *Sequential selection (SEQ).* Pick a sequence of d strata that jointly cover the entire training matrix. Then cycle through this sequence and ignore all other strata.
- *With replacement selection (WR).* Repeatedly pick a stratum uniformly and at random from the set of all strata until d strata have been processed.
- *Without replacement selection (WOR).* Pick a sequence of d strata such that the d strata jointly cover the entire training set; the sequence is picked uniformly and at random from all such sequences of d strata.[4]

Taking the scaling constant c_s in (15) as N/N_s, we can see that all three strategies are covered by Theorem 2, where each epoch corresponds to a regenerative cycle. We argue informally as follows. Recall that if Theorem 2 is to apply, then w_s must correspond to the long-term fraction of steps run on stratum Z_s. For SEQ, this means that all but d of the weights are zero, and the remaining weights satisfy $w_s = N_s/N$. For WR and WOR, we have $w_s = N_s/((d-1)!N)$, since we select each stratum s equally often in the long run, and always perform N_s steps on stratum s. The question is then whether these choices of w_s lead to a legitimate representation of L as in (7). One can show

[4] This can be performed efficiently by randomly permuting the rows and columns of a matrix of form

$$\begin{pmatrix} 1 & 2 & \cdots & d \\ 2 & 3 & \cdots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ d & 1 & \cdots & d-1 \end{pmatrix}.$$

The (k, i)-entry then contains the column index of the block to pick from row i in the kth subepoch.

that $\{ w_s \}$ satisfies (7) for all $\boldsymbol{Z}$ and L of form (2) if and only if

$$\sum_{s:Z_s \supseteq Z^{ij}} w_s c_s = 1 \tag{17}$$

for each substratum Z^{ij}. Direct verification shows that (17) holds for the above choices of w_s when $c_s = N/N_s$.

Step sizes. The stochastic approximation literature often works with step size sequences roughly of form $\epsilon_n = 1/n^\alpha$ with $\alpha \in (0.5, 1]$; and Theorem 2 guarantees asymptotic convergence for such choices. In practice, one may want to deviate from these choices to achieve faster convergence over the finite number of steps that are actually executed. We use an adaptive method for choosing the step size sequence. We exploit the fact that—in contrast to SGD in general—we can determine the current loss after every epoch. Thus we can check whether an epoch decreased or increased the loss. With this observation in mind, we employ a heuristic called *bold driver*, which is often used for gradient descent. Starting from an initial step size ϵ_0, we (1) increase the step size by a small percentage (say, 5%) whenever we see a decrease of loss, and (2) we drastically decrease the step size (say, by 50%) if we observe an increase of loss. Within each epoch, the step size remains fixed. Given a reasonable choice of ϵ_0, the bold driver method worked extremely well in our experiments. To pick ϵ_0, we leverage the fact that we have many compute nodes available. We replicate a small sample of $\boldsymbol{Z}$ (say, 0.1%) to each node. We then try different step sizes in parallel. Initially, we make a pass over the sample for step sizes $1, 1/2, 1/4, \ldots, 1/2^{d-1}$; this is done in parallel at all d nodes. The step size that gives the best result is selected as ϵ_0. As long as our loss decreases, we repeat a variation of this process after every epoch, where we try step sizes within a factor of $[1/2, 2]$ of the current step size. Eventually, the so-chosen step size will become too large and the value of the loss will increase. Intuitively, this happens when the iterate has moved closer to the global solution than to the local solution of the sample. As soon as we observe an increase of loss, we switch to the bold driver method for the rest of the process.

6.2. MapReduce/Hadoop Implementation

MapReduce is a parallel computation framework that was originally developed at Google [14], and implemented later as part of the Apache Hadoop open-source project [2]. The MapReduce framework was originally designed to scan and aggregate large datasets in a robust and scalable manner in a shared-nothing cluster of commodity servers, where each server has its own local memory and disk storage, and inter-server communication occurs over a network. The framework processes *jobs*, where each job consists of a map stage and a reduce stage. In the Hadoop implementation of MapReduce, data is physically stored as a collection of files on the Hadoop Distributed File System (HDFS). A *namenode* coordinates access to the file system data and maintains a directory tree of all files in the system. The Hadoop *InputFormat* operator partitions the raw input data into logical *splits*, and allows Hadoop to correctly parse splits into input records. The *map* stage scans the splits of the input data set, transforming each input record according to a user-defined map function and also extracting a grouping key for the record; the splits are processed in parallel by a set of independent *mapper tasks*. The *reduce* stage *shuffles* the output records from the map stage across the network

and groups them according to the grouping key, aggregates each group according to a user-defined reduce function, and writes out the result; the groups are processed and written out in parallel by a set of independent *reducer tasks*. As mentioned above, mapper and reducer tasks are executed on a cluster of servers, each of which has a fixed number of concurrent processing *slots*; a task is assigned to a single slot and processes one split or one group at a time. Whenever there are not enough map slots to simultaneously process all of the input splits during the map stage, Hadoop processes the data in a sequence of *waves*, where the number of waves roughly equals the total number of splits divided by the total number of slots. (Waves can overlap a bit when different splits have different processing times). Since tasks can run independently, Hadoop can make progress on a job as slots become available, can load balance across heterogeneous environments, and can tolerate failures.

A key advantage of the DSGD algorithm is that it can be implemented on Hadoop using map-only jobs, avoiding the expensive shuffling of data over the network that would be required in a reduce phase. A straightforward DSGD implementation, as discussed in Section 5.2, uses a single job for parallel processing of a single stratum. In this case, multiple jobs are required to perform a single DSGD iteration. Although this solution achieves scalability and fault-tolerance, its performance suffers because of Hadoop's internal overheads for spawning and coordinating jobs.

Although we expect that some of these overheads will diminish as Hadoop matures, we nonetheless employ several optimizations in order to achieve good preliminary performance. Key to these optimizations is careful organization and management of the data. Recall from Section 5.2 that we partition the $\boldsymbol{Z}$ matrix into $d \times d$ blocks, and block $\boldsymbol{W}$ and $\boldsymbol{H}$ conformingly (into d blocks each), as in (11). In our implementation, the input data to the Hadoop mappers is the data in $\boldsymbol{Z}$, and we configure the InputFormat operator so that the splits of $\boldsymbol{Z}$ correspond one-to-one with the blocks of $\boldsymbol{Z}$ (so that we can refer to "splits" or "blocks" of $\boldsymbol{Z}$ interchangeably). The matrices $\boldsymbol{W}$ and $\boldsymbol{H}$ are also stored in HDFS but are not directly handled by InputFormat. Instead, we manually ensure that these matrices are stored in multiple files $\boldsymbol{W}^1, \boldsymbol{W}^2, \ldots, \boldsymbol{W}^d, \boldsymbol{H}^1, \boldsymbol{H}^2, \ldots, \boldsymbol{H}^d$, where each file corresponds to a single block of $\boldsymbol{W}$ or $\boldsymbol{H}$ (so that we can refer to "files" or "blocks" of $\boldsymbol{W}$ and $\boldsymbol{H}$ interchangeably). The naming conventions of the files allow a mapper that is processing a given block $\boldsymbol{Z}^{ij}$ of $\boldsymbol{Z}$ to identify and retrieve the corresponding files (and hence blocks) $\boldsymbol{W}^i$ of $\boldsymbol{W}$ and $\boldsymbol{H}^j$ of $\boldsymbol{H}$ using the namenode. As mentioned in Sec. 5.3, the data can be arranged so that, for each block $\boldsymbol{Z}^{ij}$, the corresponding block $\boldsymbol{W}^i$ is stored locally, so that only the block $\boldsymbol{H}^j$ might need to be transmitted over the network. (The $\boldsymbol{W}^i$ block is transmitted and the $\boldsymbol{H}^j$ block is stored locally if the $\boldsymbol{W}$ blocks are smaller than the $\boldsymbol{H}$ blocks.)

With this setup, our optimizations are as follows. First, data is stored in Java primitive arrays rather than Java objects to avoid the performance bottleneck caused by "immutable record decoders" [19]. Next, we use block-wise I/O to read entire matrix blocks at once; this avoids expensive per-record processing costs.

Finally, we submit a single map-only job per epoch (complete data-matrix scan) rather than a map-only job per subepoch (stratum scan) to reduce Hadoop's high overhead in spawning jobs and to allow some overlapping of subepoch processing. In other words, the job processes all the blocks of $\boldsymbol{Z}$. The details are as follows.

1. In general, Hadoop processes splits in descending size order (in a bin-packing greedy fashion)

during the map stage in order to minimize the number of waves it requires. We "fool" the system by overloading the split `getLength()` function and manipulating the reported split sizes, thus gaining control over the order in which splits are processed. In this way, we can ensure that, at any time point, the current wave is processing only those splits of $\boldsymbol{Z}$ belonging to the stratum for the current subepoch (or for the next subepoch, as discussed below). Each map task processing a block $\boldsymbol{Z}^{ij}$ explicitly fetches (using low-level HDFS calls) the corresponding blocks $\boldsymbol{W}^i$ and $\boldsymbol{H}^j$ as described above, performs the local SGD updates, and outputs the new files $\boldsymbol{W}^i_{\text{new}}$ and $\boldsymbol{H}^j_{\text{new}}$, which represent the updated versions of blocks $\boldsymbol{W}^i$ and $\boldsymbol{H}^j$.

2. If there are idle map slots available while processing the current stratum, then we can use these slots to partially process splits belonging to the next stratum. Specifically, a map task assigned to a block $\boldsymbol{Z}^{kl}$ for the next stratum immediately begins to parse the corresponding split received from HDFS into individual records and then—if using the WOR or WR training sequence for records in the block (Sec. 6.1)—randomly shuffles the records. Such parsing and shuffling operations take quite some time, so overlapping this task with the processing of the current stratum yields significant performance improvements. To ensure correctness, the map task waits for the files $\boldsymbol{W}^k_{\text{new}}$ and $\boldsymbol{H}^l_{\text{new}}$ from the previous stratum processing to appear in the HDFS namenode before starting to perform the actual SGD updates. In this way, processing of the next stratum can overlap partially with processing of the current stratum to maximize performance, without loss of correctness.

7. Experiments

We compared various matrix factorization algorithms with respect to their convergence properties, runtime efficiency, and scalability. We found that the convergence speed of DSGD is on par or better than alternative methods, even when these methods are specialized to the loss function. In terms of overall performance, we found that DSGD is significantly faster, produces more stable results, and has better scalability properties.

7.1. Setup

We implemented our new DSGD method on top of MapReduce, along with the PSGD, ISGD, DGD, ALS, and MULT methods discussed in Sec. 2. The DGD algorithm uses the L-BFGS quasi-Newton method as in [13]. DSGD, PSGD, and L-BFGS are generic methods that work with a wide variety of loss functions, whereas ALS and MULT are restricted to quadratic loss functions and GKL, respectively. We used two different implementations and compute clusters; one for in-memory experiments and one for large scale-out experiments on very large datasets.

The in-memory implementation is based on R and C, and uses R's `snowfall` package to implement MapReduce. It targets datasets that are small enough to fit in aggregate memory, i.e., with up to a few billion nonzero entries. We block and distribute the input matrix across the cluster before running each experiment. The factor matrices are communicated via Samba mount points.

The R cluster consists of 16 nodes, each running two Intel Xeon E5530 processors with 8 cores at 2.4GHz. Every node has 48GB of memory.

The second implementation is based on Hadoop [2], an open-source MapReduce implementation. The Hadoop cluster is equipped with 40 nodes, each with two Intel Xeon E5440 processors and 4 cores at 2.8GHz and 32 GB of memory.

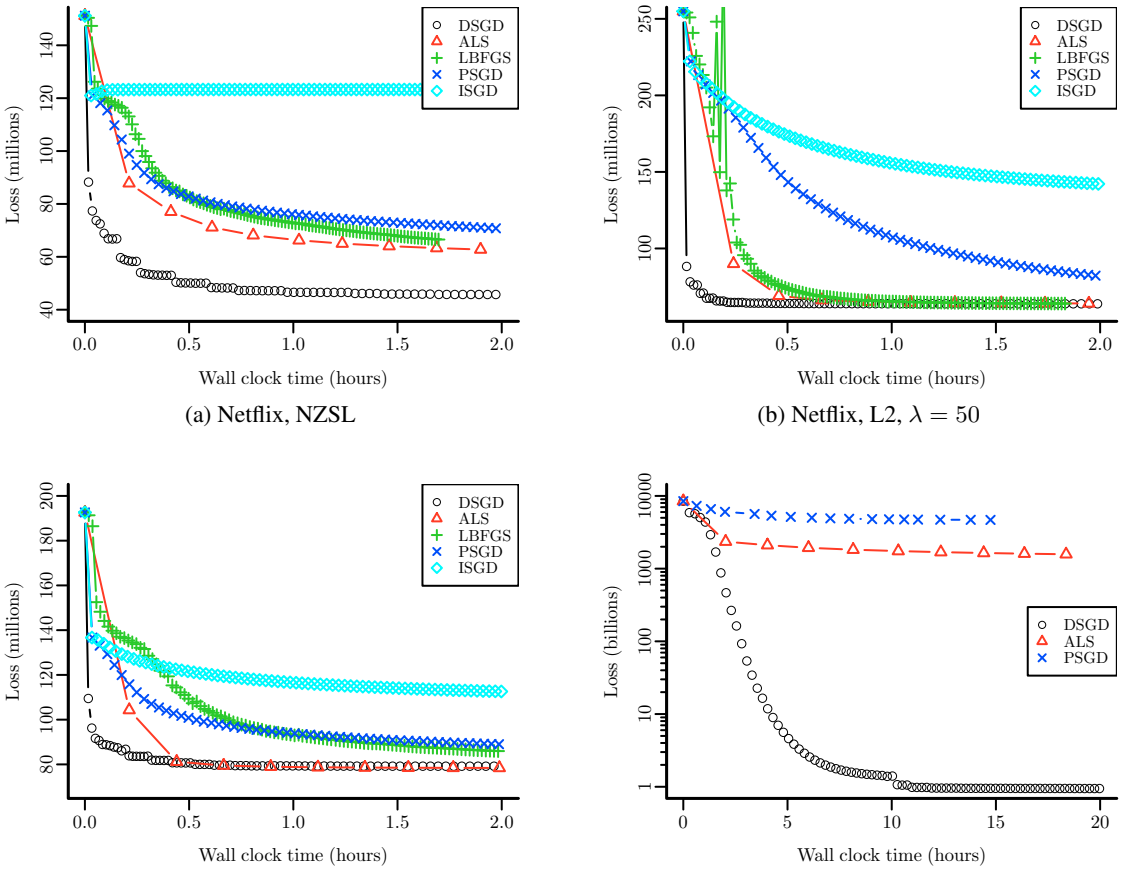
For our experiments with all SGD-based approaches, we used adaptive step size computation based on a sample of roughly 1M data points, switching to the bold driver as soon as an increase in loss was observed. The time for step size selection is included in all of our performance plots. For ISGD, we performed (parallel) step size computation using the parameters of only the first partition; this step size worked well for all partitions. Unless stated otherwise, we used WOR selection for both training sequences (all approaches) and stratum sequences (DSGD only).

We used the Netflix competition dataset [5] for our experiments on real data. The dataset contains a small subset of movie ratings given by Netflix users, specifically, 100M anonymized, time-stamped ratings from roughly 480k customers on roughly 18k movies. For larger-scale experiments on the in-memory implementation, we used a synthetic dataset with 10M rows, 1M columns, and 1B nonzero entries. We first generated matrices $\boldsymbol{W}^*$ and $\boldsymbol{H}^*$ by repeatedly sampling values from the Gaussian(0,10) distribution. We then sampled 1B entries from the product $\boldsymbol{W}^*\boldsymbol{H}^*$, and added Gaussian(0,1) noise to each sample; this procedure ensured the existence of a reasonable low-rank factorization. For all experiments, we centered the input matrix around its mean. The starting points $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$ were chosen by sampling entries uniformly and at random from $[-0.5, 0.5]$; we used the same starting point for each algorithm to ensure fair comparison. Unless stated otherwise, we used rank $r = 50$.

We used four well-known loss functions in our experiments: plain nonzero squared loss (L_{NZSL}), nonzero squared loss with an L_2 regularization term (L_{L2}), nonzero squared loss with a nonzero-weighted L_2 term (L_{NZL2}), and generalized KL divergence (L_{GKL}):

$$
\begin{aligned}
L_{\text{NZSL}} &= \sum_{(i,j)\in Z} (\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2 \\
L_{\text{L2}} &= L_{\text{NZSL}} + \lambda\big(\|\boldsymbol{W}\|_{\text{F}}^2 + \|\boldsymbol{H}\|_{\text{F}}^2\big) \\
L_{\text{NZL2}} &= L_{\text{NZSL}} + \lambda\big(\|\boldsymbol{N}_1\boldsymbol{W}\|_{\text{F}}^2 + \|\boldsymbol{H}\boldsymbol{N}_2\|_{\text{F}}^2\big) \\
L_{\text{GKL}} &= \sum_{(i,j)\in Z} (\boldsymbol{V}_{ij} \log \boldsymbol{V}_{ij}/[\boldsymbol{W}\boldsymbol{H}]_{ij} - \boldsymbol{V}_{ij}) + \sum_{i,j} [\boldsymbol{W}\boldsymbol{H}]_{ij},
\end{aligned}
\tag{18}
$$

where $\boldsymbol{N}_1$ ($\boldsymbol{N}_2$) is a diagonal matrix that rescales each row (column) of $\boldsymbol{W}$ ($\boldsymbol{H}$) by the number of nonzero entries of $\boldsymbol{Z}$ in that row (column), and $\|\boldsymbol{A}\|_{\text{F}}$ denotes the Frobenius norm of a matrix $\boldsymbol{A}$ (see App. C). L_{NZL2} has been used successfully on the Netflix data [20], and L_{GKL} has applications in text indexing [17]. We used "principled" values of λ throughout. E.g., we used values of $\lambda = 50$ and $\lambda = 0.05$ for L_{L2} and L_{NZL2} on Netflix data, respectively, and $\lambda = 0.1$ for L_{L2} on synthetic data, the former values because they yielded the best movie recommendations on held-out data, and the latter because it is "natural" in that the resulting minimum-loss factors correspond to the "maximum a posteriori" Bayesian estimator of $\boldsymbol{W}$ and $\boldsymbol{H}$ under the Gaussian-based procedure used to generate the synthetic data.



(a) Netflix, NZSL

(b) Netflix, L2, $\lambda = 50$

(c) Netflix, NZL2, $\lambda = 0.05$

(d) Synthetic data, L2, $\lambda = 0.1$

Figure 3: Performance in terms of wall-clock time

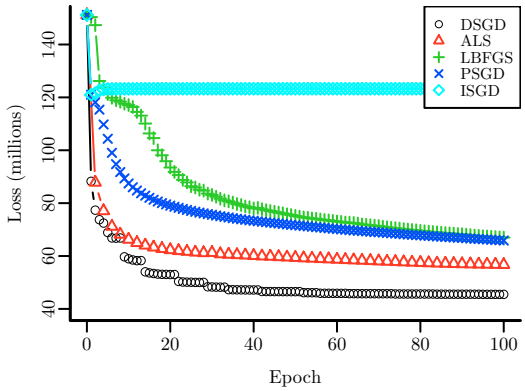
7.2. Relative Performance

We first evaluated the relative performance of the matrix factorization algorithms. For various loss functions and datasets, we ran 100 epochs—i.e, scans of the data matrix—with each algorithm and measured the elapsed wall-clock time, as well as the value of the loss after every epoch. We used 64-way distributed processing on 8 nodes (with 8 concurrent map tasks per node).[5]

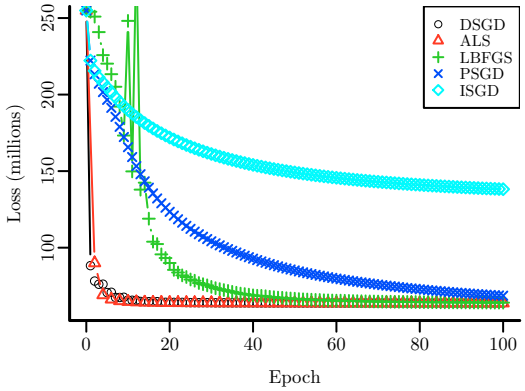
Representative results are given in Figure 3, which displays the achieved loss as a function of wall clock time, and in Figure 4, which displays the loss as a function of the number of epochs.

As can be seen from Figure 3, DSGD converges about as fast as—or faster than—alternative

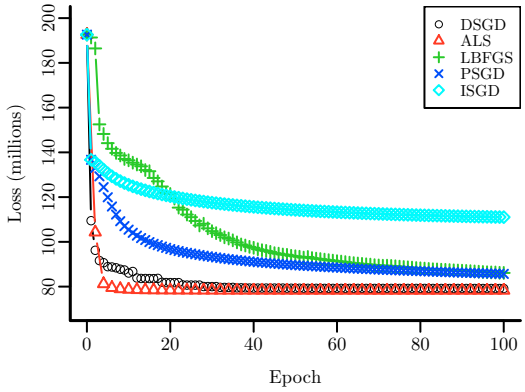
[5]Note that for all approaches but ISGD and ALS, 64-way distributed processing is excessive for the Netflix data; the execution time is dominated by latencies. We nonetheless used 64-way processing to get a consistent view over datasets of various sizes.



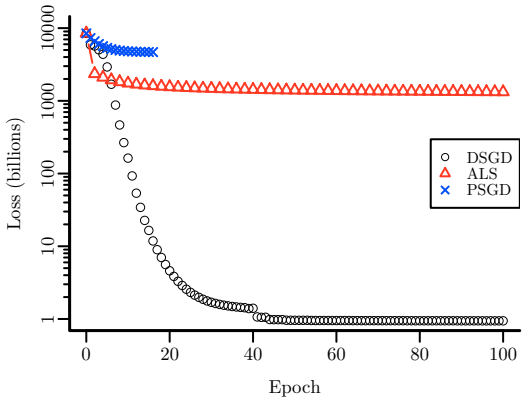
(a) Netflix, NZSL



(b) Netflix, L2, $\lambda = 50$



(c) Netflix, NZL2, $\lambda = 0.05$



(d) Synthetic data, L2, $\lambda = 0.1$

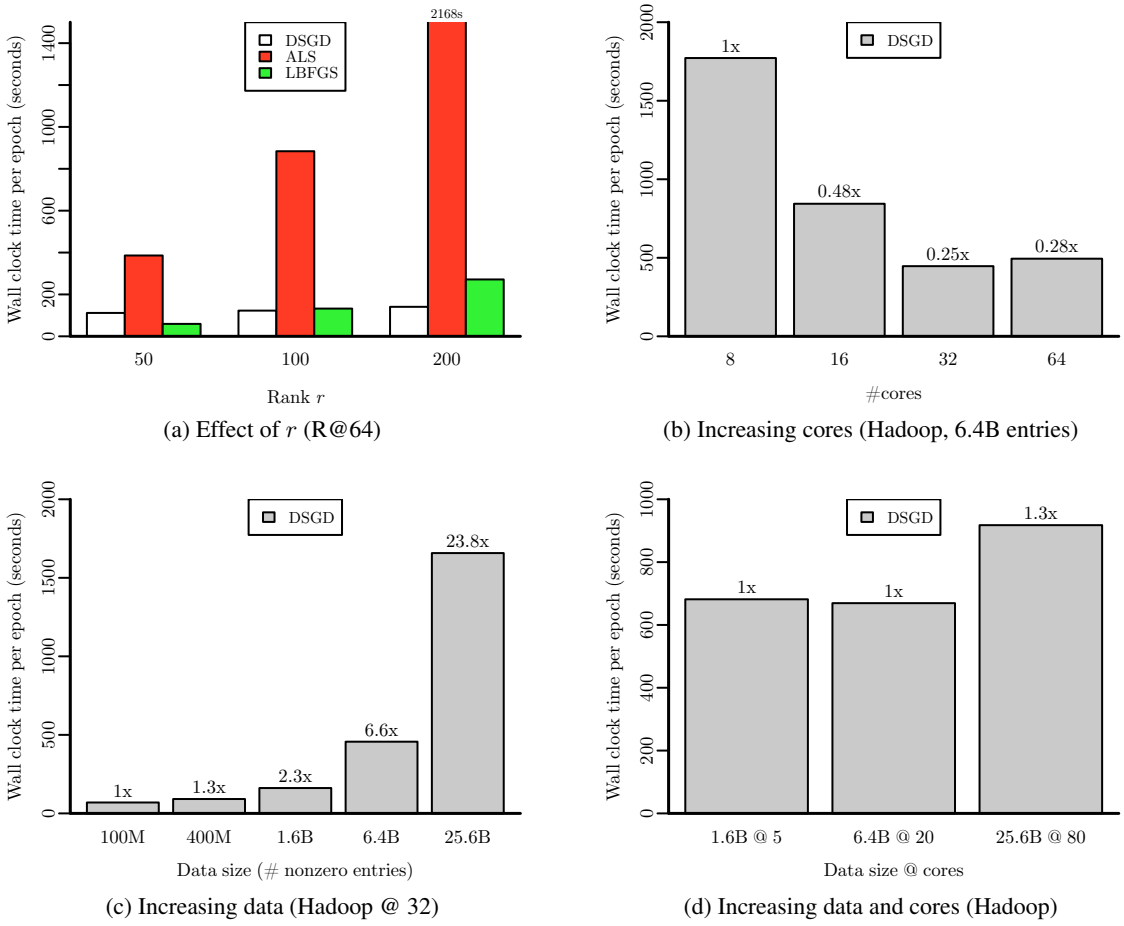
Figure 4: Performance in terms of epochs

methods, with the DSGD performing markedly better for L_{NZSL} loss over the Netflix data and for L_{L2} loss over both the Netflix dataset and the large synthetic dataset. Comparing the various SGD-based approaches, we observe that ISGD and PSGD exhibit consistently inferior performance, and offer the following explanation. The matrix-factorization problem is "non-identifiable" in that the loss function has many global minima that correspond to widely different values of $(\boldsymbol{W}, \boldsymbol{H})$. Averages of $(\boldsymbol{W}, \boldsymbol{H})$ values from different partitions, as computed by ISGD and PSGD, do not correspond to good overall solutions, and the algorithms may not converge to a local minimum of the global loss, or may converge very slowly. E.g., in the example of Figure 2, ISGD would converge to a point on the line between the red and blue minima. Of the ISGD and PSGD algorithms, it is not surprising that ISGD has the worst convergence behavior; recall that ISGD computes a local optimum of each partition of the dataset via SGD, and averages only once after all 100 epochs.[6] PSGD improves on ISGD by averaging parameters after every epoch, but it is still outperformed by most other approaches. DSGD performs best; its usage of stratification instead of averaging significantly improves convergence speed. For the remainder of our discussion, we focus on DSGD as the best SGD-based algorithm and compare it with L-BFGS and ALS. L-BFGS is clearly inferior to the other two algorithms. Indeed, we do not give results for L-BFGS in Figures 3d or 4d because its centralized parameter-update step ran out of memory when faced with very large data. In the other three experiments, L-BFGS is able to execute more epochs per unit time than the other algorithms—e.g., for the L_{NZSL} experiment, DSGD ran 43 epochs, ALS ran 10 epochs, and L-BFGS ran 61 epochs in the first hour—but the per-epoch decrease in loss is relatively small. In general, the foregoing differences in runtime are explained by different computational costs (highest for ALS, which has to solve $m + n$ least-squares problems per epoch) and synchronization costs (highest for PSGD, which has to average all parameters in each epoch).

ALS achieves performance roughly comparable to DSGD for L_{NZL2} loss over Netflix data, taking about 15 more minutes to get within the vicinity of the minimal loss but ultimately yielding a slightly lower loss (about 1% less than that of DSGD) after two hours. ALS is clearly inferior to DSGD, however, in the other three experiments. The differences are more noticeable in Figure 3 than in Figure 4, since they reflect the larger execution time per epoch for ALS as indicated above; see also Figure 5a. For the first experiment—L_{NZSL} loss over Netflix data—the lack of a regularization term makes the factorization difficult for ALS because the search space is relatively large and there are many equivalent solutions. Specifically, we observed that ALS conducted large moves through the parameter space; the factors grew without bound. On the synthetic data with L_{L2}, ALS is very effective in the first epoch, but then converges slowly. We have observed similar behavior on very small matrices, with ALS getting stuck when moving along "valleys" of small loss in which both $\boldsymbol{W}$ and $\boldsymbol{H}$ change simultaneously. DSGD does not suffer from these problems and has superior convergence properties.

In summary, the overall performance of DSGD was consistently more stable than that of alternative algorithms. The speed of convergence was comparable or faster.

[6] The intermediate points shown in Figure 3 have been obtained by pausing ISGD after every epoch in order to average parameters and compute the loss. The time to do so is not included in the wall-clock time of ISGD.



(a) Effect of r (R@64)

(b) Increasing cores (Hadoop, 6.4B entries)

(c) Increasing data (Hadoop @ 32)

(d) Increasing data and cores (Hadoop)

Figure 5: Scalability

7.3. Scalability

Next, we studied various scalability issues in our shared-nothing MapReduce environment. We examined the effect of scaling up the approximation rank r, and then explored the scalability of DSGD on a Hadoop cluster by scaling up the dataset size, the number of cores, and then both the dataset size and number of cores. Overall, the gradient descent methods scale better with increasing rank than ALS, and DSGD has good scalability properties on Hadoop, provided that the amount of data processed per core does not become so small that system overheads start to dominate.

To explore the effect of increasing the approximation rank, we used the Netflix data with L_{NZL2} loss and the same R-cluster setup as before (64 cores, 8 nodes); note that the value $r = 50$ corresponds to our relative-performance experiments. The results are displayed in Figure 5a. As observed previously, ALS is significantly slower than DSGD and L-BFGS. ALS spends most

of its time on constructing and solving (in the least-squares sense) systems of linear equations. Since the number of both equations and variables increases with rank—construction is $O(Nr^2)$, solving is $O((m+n)r^3)$—the performance degrades significantly as r increases. L-BFGS performs centralized updates of the factors; these centralized updates become a bottleneck as the rank (and thus factor size) increases. The impact of increased rank on DSGD appears rather mild, mainly because factors are fully distributed. As the rank increases further and gradient estimation becomes the major bottleneck, we expect to see a more pronounced increase in runtime.

Our remaining experiments focus on the performance of DSGD on the Hadoop cluster. Figure 5b depicts scale-up results as we repeatedly double the number of cores while keeping the data size constant at 6.4B entries. Figure 5c plots the runtime per epoch as we repeatedly quadruple the data (while keeping the number of cores constant at 32), and Figure 5d shows scale-out results, in which we scale both data and cores simultaneously.

As can be seen in Figure 5b, DSGD initially achieves roughly linear speed-up as the number of cores is repeatedly doubled, up to 32 cores. After this point, speed-up performance starts to degrade. The reason for this behavior is that, when the number of cores becomes large, the amount of data processed per core becomes small—e.g., 64-way DSGD requires 64^2 blocks, so that the amount of data per block is only ≈ 25MB. The actual time to execute DSGD on the data becomes negligible, and the overall processing times become dominated by Hadoop overheads, especially the time required to spawn a task. (Hadoop is designed for tasks that run at least on the order of minutes.) A similar phenomenon can be seen in Figure 5c, where the elapsed time is sublinear in the data size for small datasets (≤ 1.6B entries); for larger datasets, overheads no longer mask performance, and the runtime increases linearly with dataset size. In our scale-out experiments (Figure 5d), the impact of Hadoop overheads is more muted, since scaling up the data size offsets the overhead effect caused by increasing the number of cores. E.g., the processing time initially remains constant as the dataset size and number of cores are each scaled up by a factor of 4, with the overall runtime increasing by a modest 30% as we scale to very large datasets on large clusters. The foregoing overhead effects can potentially be ameliorated by improving scheduling in Hadoop or using an alternative parallel runtime system such as Spark [31].

7.4. Selection Schemes

Finally, we evaluated the impact of different strategies for selecting both strata and training sequences. The runtime cost for the various alternatives are comparable, but the speed of convergence differs significantly. Figure 6 shows exemplary results for 64-way SGD on the Netflix data with L_{NZL2}, where we plot all combinations of options for training-point and stratum selection. Sequential stratum selection performed worst: the curves corresponding to this scheme cluster at the upper right of the plot, whereas the curves for the randomized strategies cluster at the lower left; i.e., any form of randomized stratum selection helped significantly. For randomized selection schemes, the WOR strategy for selecting training points yielded the best results. Overall, WOR selection for both training points and strata ensures a good balance between randomization and processing many different data points.

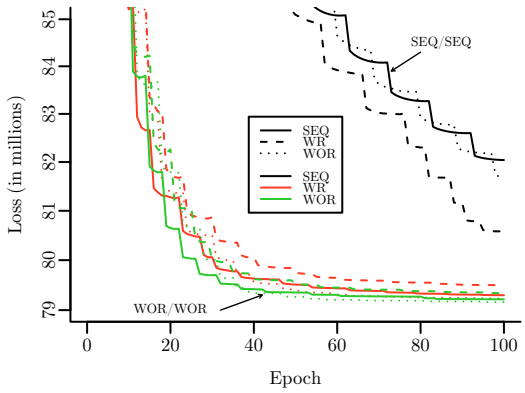


Figure 6: Effect of stratum selection (line color) and training sequence (line type)

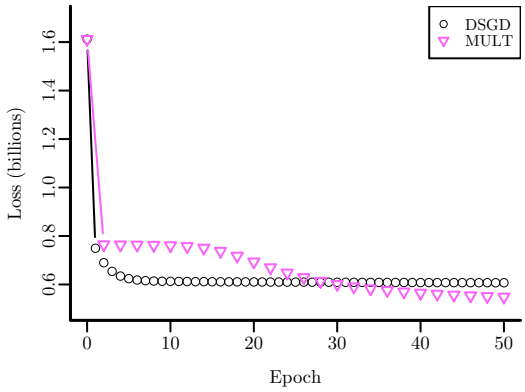


Figure 7: GKL on Netflix data (R@64)

7.5. Other Loss Functions

To show that DSGD can be applied to a variety of loss functions, we implemented L_{GKL}—a loss function that is used for nonnegative matrix factorization and that does *not* ignore zeros in the $\boldsymbol{V}$ matrix—and ran the resulting factorization algorithm on the Netflix data, along with the loss-specific MULT algorithm of Das et al. [12]. DSGD performs respectably compared to MULT (Figure 7), reaching the vicinity of the minimum loss more rapidly—roughly 7 epochs for DSGD versus 27 epochs for MULT—and achieving an ultimate loss that is only modestly greater than that of MULT. Our implementation is a first cut; we are currently refining it further.

8. Conclusions

We introduced DSGD, a novel algorithm for large-scale matrix factorization. DSGD is fully distributed and can handle matrices with millions of rows, millions of columns, and billions of nonzero entries. In contrast to most alternative algorithms, DSGD is generic in that it supports a wide variety of loss functions that arise in practice. Our experiments indicate that DSGD is on par or faster than specialized algorithms in terms of runtime, convergence properties, and memory requirements. Recent work [29, 30] is yielding versions of the DSGD algorithm for environments besides MapReduce, such as shared-nothing MPI and multithreaded shared memory architectures, as well as platforms such as Spark [24]. The DSGD idea is also being adapted to solve other problems, such as cubic spline interpolation for massive time series [15].

References

[1] R. Albright, J. Cox, D. Duling, A. Langville, and C. Meyer. Algorithms, initializations, and convergence for the nonnegative matrix factorization. Technical Report Math 81706, NCSU, 2006.

[2] Apache Hadoop. https://hadoop.apache.org.

[3] S. Asmussen. *Applied Probability and Queues*. Springer, 2nd edition, 2003.

[4] S. Asmussen and P. W. Glynn. *Stochastic Simulation: Algorithms and Analysis*. Springer, 2007.

[5] J. Bennett and S. Lanning. The Netflix prize. In *KDD Cup and Workshop*, 2007.

[6] C. M. Bishop. *Pattern Recognition and Machine Learning*. Information Science and Statistics. Springer, 2007.

[7] L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In *NIPS*, volume 20, pages 161–168. 2008.

[8] R. H. Byrd, P. Lu, J. Nocedal, and C. Zhu. A limited memory algorithm for bound constrained optimization. *SIAM J. Sci. Comput.*, 16(5):1190–1208, 1995.

[9] Y. S. Chow and H. Teicher. *Probability Theory: Independence, Interchangeability, Martingales*. Springer, 2nd edition, 1988.

[10] K. L. Chung. *A Course in Probability Theory*. Elsevier, third edition, 2001.

[11] A. Cichocki and R. Zdunek. Regularized alternating least squares algorithms for non-negative matrix/tensor factorization. In *ISNN '07: Proc. of the 4th international symposium on Neural Networks*, pages 793–802, 2007.

[12] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. In *WWW*, pages 271–280, 2007.

[13] S. Das, Y. Sismanis, K. S. Beyer, R. Gemulla, P. J. Haas, and J. McPherson. Ricardo: Integrating R and Hadoop. In *SIGMOD*, pages 987–998, 2010.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[15] P. J. Haas and Y. Sismanis. On aligning massive time-series data in Splash. In *VLDB BigData Workshop*, 2012.

[16] K. B. Hall, S. Gilpin, and G. Mann. MapReduce/Bigtable for distributed optimization. In *NIPS LCCC Workshop*, 2010.

[17] T. Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57, 1999.

[18] T. Hofmann. Latent semantic models for collaborative filtering. *ACM Trans. Inf. Syst.*, 22(1):89–115, 2004.

[19] D. Jiang, B. C. Ooi, L. Shi, and S. Wu. The performance of MapReduce: An in-depth study. *PVLDB*, 3(1):472–483, 2010.

[20] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *IEEE Computer*, 42(8):30–37, 2009.

[21] H. J. Kushner and G. Yin. *Stochastic Approximation and Recursive Algorithms and Applications*. Springer, 2nd edition, 2003.

[22] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, 1999.

[23] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *NIPS*, pages 556–562, 2000.

[24] B. Li, S. Tata, and Y. Sismanis. Sparkler: Supporting large-scale matrix factorization. In *EDBT*, 2013. To appear.

[25] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, pages 681–690, 2010.

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

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

[28] A. P. Singh and G. J. Gordon. A unified view of matrix factorization models. In *ECML PKDD*, pages 358–373, 2008.

[29] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. In *ICDM*, pages 655–664, 2012.

[30] C. Teflioudi, F. Makari, R. Gemulla, P. J. Haas, and Y. Sismanis. Shared-memory and shared-nothing algorithms for matrix completion, 2013. Submitted.

[31] M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010.

[32] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the Netflix Prize. In *AAIM*, pages 337–348, 2008.

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

A. MapReduce Algorithms for Matrix Factorization

We review some algorithms for finding an $m \times r$ matrix $\boldsymbol{W}$ and an $r \times n$ matrix $\boldsymbol{H}$ such that $\boldsymbol{V} \approx \boldsymbol{W}\boldsymbol{H}$ for a given $m \times n$ input matrix $\boldsymbol{V}$, in the sense of minimizing a specified loss function $L(\boldsymbol{V}, \boldsymbol{W}\boldsymbol{H})$ computed over N training points. We focus on algorithms that have been shown to work in a distributed setting—in which d tasks can perform computations in parallel—on very large matrices. All algorithms are of an iterative nature. They start with some initial factors $\boldsymbol{W}_0$ and $\boldsymbol{H}_0$, which are repeatedly updated until some convergence criteria is met. Interesting properties of such algorithms include supported loss functions, convergence properties, whether they support non-negativity or box constraints, time and space complexity, and how distribution is achieved. An overview of the algorithms is given in Table 1.

A.1. Specialized Algorithms

The specialized algorithms below are each designed for a certain class of loss functions. In all cases, distributed processing is achieved by partitioning the input matrix and splitting up linear algebra computations across nodes.

Alternating Least Squares. In its standard form, the method of alternating least squares optimizes

$$L_{\text{SL}} = \sum_{i,j} (\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2.$$

Table 1: MapReduce Algorithms for Matrix Factorization*

Algorithm	**Loss**	**Regularizer**	**Partitioning**	**[Scans**	**M**	**R**	**Time] per Iteration**
ALS [32]	SL, NZSL	L2, NZL2	$\boldsymbol{V}$ & $\boldsymbol{W}$ by rows, $\boldsymbol{V}$ & $\boldsymbol{H}$ by columns	2	2	0	$O(d^{-1}[Nr^2 + (m+n)r^3])$
	SL	L2	(as above)	2	2	0	$O(d^{-1}[Nr + (m+n)r^2] + r^3)$
EM [12]	GKL	-	$\boldsymbol{V}$ blocked rect., $\boldsymbol{W}$ & $\boldsymbol{H}$ conformingly	1	1	1	$O(d^{-1}Nr)$
MULT [25]	GKL	-	$\boldsymbol{V}$ blocked rect., $\boldsymbol{W}$ & $\boldsymbol{H}$ conformingly	2	2	2	$O(d^{-1}Nr)$
	SL	-	(as above)	2	2	2	$O(d^{-1}[Nr + (m+n)r^2])$
GD [13]	diff.	diff.	$\boldsymbol{V}$ blocked rect., $\boldsymbol{W}$ & $\boldsymbol{H}$ conformingly	1	1	1	$O(d^{-1}N)T_{L'} + T_{\text{UPD}}$
PSGD	diff.	diff.	$\boldsymbol{V}$ partitioned arbitrary, $\boldsymbol{W}$ & $\boldsymbol{H}$ replicated	1	1	1	$O(d^{-1}N)T_{L'} + O((m+n)r)$
DSGD	diff.	diff.	$\boldsymbol{V}$ blocked square, $\boldsymbol{W}$ & $\boldsymbol{H}$ conformingly	1	d**	0	$O(d^{-1}N)T_{L'}$
	NZSL	L2, NZL2	(as above)	1	d**	0	$O(d^{-1}Nr)$

*The Scans, M, and R columns refer to the number of data scans, map jobs, and reduce jobs per iteration of the algorithm, respectively. "diff." means that the algorithm can handle arbitrary differentiable loss functions. $T_{L'}$ and T_{UPD} denote the time required by gradient-based methods to compute a local gradient L'_{ij} and the time to update all parameters once gradients have been computed, respectively. Some algorithms have multiple entries; these entries may refer to different losses or regularizers (MULT), to specific loss or regularizers on which improved runtime properties are achieved (ALS), or to an example (DSGD).

**One Map job per subepoch, each processing d blocks. In our Hadoop implementation, we use customized locking strategies to run just one Map job per iteration.

The method alternates between finding the best value for $\boldsymbol{W}$ given $\boldsymbol{H}$, and finding the best value of $\boldsymbol{H}$ given $\boldsymbol{W}$. This amounts to computing the least squares solutions to the following systems of linear equations

$$\begin{aligned}\boldsymbol{V}_{i*} - \underline{\boldsymbol{W}_{i*}}\boldsymbol{H}_n &= \boldsymbol{0},\\ \boldsymbol{V}_{*j} - \boldsymbol{W}_{n+1}\underline{\boldsymbol{H}_{*j}} &= \boldsymbol{0},\end{aligned}$$

where the unknown variable is underlined. This specific form suggests that each row of $\boldsymbol{W}$ can be updated by accessing only the corresponding row in the data matrix $\boldsymbol{V}$, while each column in $\boldsymbol{H}$ can be updated by accessing the corresponding column in $\boldsymbol{V}$. This facilitates distributed processing; see below. The equations can be solved using a method of choice. We obtain

$$\begin{aligned}\boldsymbol{W}_{n+1}^{\mathrm{T}} &\leftarrow (\boldsymbol{H}_n\boldsymbol{H}_n^{\mathrm{T}})^{-1}\boldsymbol{H}_n\boldsymbol{V}^{\mathrm{T}},\\ \boldsymbol{H}_{n+1} &\leftarrow (\boldsymbol{W}_{n+1}^{\mathrm{T}}\boldsymbol{W}_{n+1})^{-1}\boldsymbol{W}^{\mathrm{T}}\boldsymbol{V}.\end{aligned}$$

for the unregularized loss shown above. When an additional L_2 regularization term of form $\lambda(\|\boldsymbol{W}\|_{\mathrm{F}}^2 + \|\boldsymbol{H}\|_{\mathrm{F}}^2)$ is added, we obtain

$$\boldsymbol{W}_{n+1}^{\mathrm{T}} \leftarrow (\boldsymbol{H}_n\boldsymbol{H}_n^{\mathrm{T}} + \lambda\boldsymbol{I})^{-1}\boldsymbol{H}_n\boldsymbol{V}^{\mathrm{T}}, \tag{19a}$$

$$\boldsymbol{H}_{n+1} \leftarrow (\boldsymbol{W}_{n+1}^{\mathrm{T}}\boldsymbol{W}_{n+1} + \lambda\boldsymbol{I})^{-1}\boldsymbol{W}^{\mathrm{T}}\boldsymbol{V}. \tag{19b}$$

Since the update term of $\boldsymbol{H}_{n+1}$ depends on $\boldsymbol{W}_{n+1}$, the input matrix has to be processed twice to update both factor matrices.

In contrast to SVD, ALS does not produce an orthogonal factorization and it might get stuck in local minima. However, ALS can handle a wide range of variations for which SVD is not applicable, but which are important in practice. Examples include non-negativity constraints [1], sparsity constraints [1, 11], weights [11], regularization [11, 32], or the restriction to nonzero entries [32]. In general, ALS is applicable when the loss function is quadratic in both $\boldsymbol{W}$ and $\boldsymbol{H}$.

Zhou et al. [32] proposed a distributed version of ALS for L_{NZSL}, see Eq. (1). We first describe a variation for L_{SL}, and then outline how to optimize L_{NZSL}. In both cases, the algorithm runs two Map-only jobs per iteration, one to compute $\boldsymbol{W}_{n+1}$ and one to compute $\boldsymbol{H}_{n+1}$. Each of the jobs uses a different partitioning of $\boldsymbol{V}$, either by rows or by columns. For example, we use the row partitioning to compute $\boldsymbol{W}_{n+1}$: Each mapper reads a set of rows of $\boldsymbol{V}$, the corresponding rows of $\boldsymbol{W}_n$, and the entire matrix $\boldsymbol{H}_n$. The ith mapper then solves the part of equation (19a) that concerns its rows:

$$\boldsymbol{W}_{n+1,i*}^{\mathrm{T}} \leftarrow \underbrace{(\boldsymbol{H}_n\boldsymbol{H}_n^{\mathrm{T}} + \lambda\boldsymbol{I})}_{\text{coefficient matrix}}^{-1} \underbrace{\boldsymbol{H}_n\boldsymbol{V}_{i*}^{\mathrm{T}}}_{\text{right-hand side}},$$

where $\boldsymbol{W}_{n+1,i*}$ denotes the rows of $\boldsymbol{W}_{n+1}$ read by the ith mapper, and similarly for $\boldsymbol{V}_{i*}$. For L_{SL}, the coefficient matrix is shared across rows, which allows us to reuse computation (e.g., a QR factorization of the coefficient matrix). The matrix $\boldsymbol{H}_{n+1}$ is computed analogously. The overall time complexity is $O(d^{-1}[Nr + (m+n)r^2] + r^3)$ time.[7] For L_{NZSL}, we modify the algorithm so that it

[7] The right-hand sides can be constructed in $O(Nr)$ time. The coefficient matrix can be constructed in $O((m+n)r^2)$ time, and reduced to an upper-triangular form in $O(r^3)$ time (not parallel). For each of the $m+n$ equation systems, back-substitution takes $O(r^2)$ time.

uses a different coefficient matrix for each system of equations, i.e., for each mapper. Intuitively, we remove equations that correspond to zero entries of $\boldsymbol{V}$ from the least-squares problem. This is achieved by using

$$\boldsymbol{W}_{n+1,i*}^{\mathrm{T}} \leftarrow \underbrace{(\boldsymbol{H}_n^{(i)}[\boldsymbol{H}_n^{(i)}]^{\mathrm{T}} + \lambda \boldsymbol{I})}_{\text{coefficient matrix}}^{-1} \underbrace{\boldsymbol{H}_n \boldsymbol{V}_{i*}^{\mathrm{T}}}_{\text{right-hand side}},$$

where $\boldsymbol{H}_n^{(i)}$ consists of just the columns of $\boldsymbol{H}_n$ that have non-zero entries in the respective column of $\boldsymbol{V}_{i*}$. The computation of $\boldsymbol{H}_{n+1}$ is analogous. The time complexity increases to $O(d^{-1}[Nr^2 + (m+n)r^3])$.[8]

Expectation Maximization (EM). Hofmann et al. [17, 18] proposed an EM algorithm to minimize the KL divergence L_{KL} in the context of "probabilistic latent semantic analysis" (pLSA). As we discuss below, the algorithm can be seen as a matrix factorization algorithm. Let $\boldsymbol{V}$ be non-negative and $\sum_{i,j} \boldsymbol{V}_{ij} = 1$.[9] Then, $\boldsymbol{V}$ corresponds to a probability distribution over pairs (i, j). pLSA factors this probability distribution as follows

$$\Pr[\, i, j \,] \approx \sum_z \Pr[\, z \,] \Pr[\, i \mid z \,] \Pr[\, j \mid z \,], \tag{20}$$

where $z \in \{\, z_1, \ldots, z_r \,\}$ is a latent variable and follows a multinomial distribution over r topics. If we identify $\Pr[\, i, j \,] = \boldsymbol{V}_{ij}$, $\Pr[\, z \,] = \boldsymbol{Z}_{zz}$ (where $\boldsymbol{Z}$ is a diagonal $r \times r$ matrix), $\Pr[\, i \mid z \,] = \boldsymbol{W}'_{iz}$, and $\Pr[\, j \mid z \,] = \boldsymbol{H}'_{zj}$, we obtain the following equivalent matrix factorization

$$\boldsymbol{V} \approx \boldsymbol{W}' \boldsymbol{Z} \boldsymbol{H}'.$$

Note that $\boldsymbol{W}'$ ($\boldsymbol{H}'$) describes a conditional probability distribution; thus each of its columns (rows) sums to 1.

Model fitting is performed as follows. In the E-step, we compute the probability $\Pr[\, z \mid i, j \,]$ that entry (i, j) is explained by topic z:

$$\Pr[\, z \mid i, j \,] = \frac{\Pr[\, z \,] \Pr[\, i \mid z \,] \Pr[\, j \mid z \,]}{\sum_{z'} \Pr[\, z' \,] \Pr[\, i \mid z' \,] \Pr[\, j \mid z' \,]}. \tag{21}$$

In the M-step, the parameters are updated. Using normalization constants $K_z = \sum_{i,j} \boldsymbol{V}_{ij} \Pr[\, z \mid i, j \,]$, we set

$$\Pr[\, j \mid z \,] = \frac{1}{K_z} \sum_i \boldsymbol{V}_{ij} \Pr[\, z \mid i, j \,] \tag{22a}$$

$$\Pr[\, i \mid z \,] = \frac{1}{K_z} \sum_j \boldsymbol{V}_{ij} \Pr[\, z \mid i, j \,] \tag{22b}$$

$$\Pr[\, z \,] = K_z \tag{22c}$$

[8] We can construct all coefficient matrices in $O(Nr^2)$ time. Solving each system takes $O(r^3)$ time.

[9] If $R = \sum_{i,j} \boldsymbol{V}_{ij} \neq 1$, we normalize $\boldsymbol{V}$ by dividing each element by R.

It can be shown that the KL divergence between the distribution $\boldsymbol{V}$ and the fitted distribution (20) is non-increasing in every EM iteration. The EM algorithm converges to a stationary point of L_{KL}.

We now transform the EM algorithm into the language of linear algebra. This will allow us to uncover similarities between the EM algorithm and the multiplicative update rules described later, and also facilitates exposition of distributed EM. In what follows, we set $\boldsymbol{W} = \boldsymbol{W}'$ and $\boldsymbol{H} = \boldsymbol{Z}\boldsymbol{H}'$, i.e., we factor $\boldsymbol{Z}$ into the parameter matrix $\boldsymbol{H}$. $\boldsymbol{Z}$ can be readily factored out after convergence, if desired. The E-step (21) becomes

$$\Pr[\, z \mid i, j \,] = \left[\frac{\boldsymbol{W}_{*z}\boldsymbol{H}_{z*}}{\boldsymbol{W}\boldsymbol{H}}\right]_{ij}, \tag{23}$$

where division is performed element-wise. Let $\boldsymbol{K}_n = \mathrm{diag}(K_{n,1}, \ldots, K_{n,r})$ be the matrix of the normalization constants used in the $(n+1)$st M-step. Inserting (23) directly into the equations (22), we obtain the following update rules:

$$\begin{aligned}
\boldsymbol{W}'_{n+1} &\leftarrow \left[\boldsymbol{W}_n \circ (\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n)\boldsymbol{H}_n^{\mathrm{T}}\right]\boldsymbol{K}_n^{-1} && \text{(24a)}\\
\boldsymbol{H}'_{n+1} &\leftarrow \boldsymbol{K}_n^{-1}\left[\boldsymbol{W}_n^{\mathrm{T}}(\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n) \circ \boldsymbol{H}_n\right] && \text{(24b)}\\
\boldsymbol{Z}_{n+1} &= \boldsymbol{K}_n, && \text{(24c)}
\end{aligned}$$

where $\circ$ denotes element-wise multiplication. Note that $\boldsymbol{K}_n^{-1}$ is easy to compute as $\boldsymbol{K}_n$ is a diagonal matrix. Since all resulting matrices describe (conditional) probability distributions, they are normalized appropriately. By construction, we have

$$\begin{aligned}
\boldsymbol{K}_n &= \mathrm{diag}(\mathrm{colSums}\left[\boldsymbol{W}_n \circ (\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n)\boldsymbol{H}_n^{\mathrm{T}}\right])\\
&= \mathrm{diag}(\mathrm{rowSums}\left[\boldsymbol{W}_n^{\mathrm{T}}(\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n) \circ \boldsymbol{H}_n\right]),
\end{aligned}$$

where $\mathrm{colSums}[\boldsymbol{A}]_j = \sum_i \boldsymbol{A}_{ij}$ and $\mathrm{rowSums}[\boldsymbol{A}]_i = \sum_j \boldsymbol{A}_{ij}$ for a matrix $\boldsymbol{A}$. is the If we compute $\boldsymbol{W}_{n+1}$ and $\boldsymbol{H}_{n+1}$ directly, we arrive at the following final update rules

$$\begin{aligned}
\tilde{\boldsymbol{W}}_{n+1} &\leftarrow \boldsymbol{W}_n \circ (\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n)\boldsymbol{H}_n^{\mathrm{T}} && \text{(25a)}\\
\boldsymbol{W}_{n+1} &\leftarrow \tilde{\boldsymbol{W}}_{n+1}\,\mathrm{diag}(1/\,\mathrm{colSums}[\tilde{\boldsymbol{W}}_{n+1}]) && \text{(25b)}\\
\boldsymbol{H}_{n+1} &\leftarrow \boldsymbol{W}_n^{\mathrm{T}}(\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n) \circ \boldsymbol{H}_n, && \text{(25c)}
\end{aligned}$$

where $\tilde{\boldsymbol{W}}_{n+1}$ is an intermediate variable that is normalized to obtain $\boldsymbol{W}_{n+1}$.

Das et al. [12] show how to distribute the EM algorithm using MapReduce. The idea is to partition matrix $\boldsymbol{V}$ into $d_1 \times d_2$ blocks, $\boldsymbol{W}$ into d_1 conforming blocks, and $\boldsymbol{H}$ into d_2 conforming blocks. Only one MapReduce job is needed to perform one EM iteration. To make this work, we first push the normalization of $\tilde{\boldsymbol{W}}_{n+1}$ in equations (25) into the $(n+2)$nd iteration:

$$\begin{aligned}
\tilde{\boldsymbol{W}}_{n+1} &\leftarrow \tilde{\boldsymbol{W}}_n\boldsymbol{K}_n^{-1} \circ (\boldsymbol{V}/\tilde{\boldsymbol{W}}_n\boldsymbol{K}_n^{-1}\boldsymbol{H}_n)\boldsymbol{H}_n^{\mathrm{T}} && \text{(26a)}\\
\boldsymbol{K}_{n+1} &\leftarrow \mathrm{diag}(\mathrm{colSums}[\tilde{\boldsymbol{W}}_{n+1}]) && \text{(26b)}\\
\boldsymbol{H}_{n+1} &\leftarrow [\tilde{\boldsymbol{W}}_n\boldsymbol{K}_n^{-1}]^{\mathrm{T}}(\boldsymbol{V}/\tilde{\boldsymbol{W}}_n\boldsymbol{K}_n^{-1}\boldsymbol{H}_n) \circ \boldsymbol{H}_n. && \text{(26c)}
\end{aligned}$$

where $\tilde{\boldsymbol{W}}_0 = \boldsymbol{W}_0$ and $\boldsymbol{K}_0 = \operatorname{diag}(\operatorname{colSums} \boldsymbol{W}_0)$. Each mapper reads a block of $\boldsymbol{V}$, the corresponding blocks of $\tilde{\boldsymbol{W}}_n$ and $\boldsymbol{H}_n$, and the entire matrix $\boldsymbol{K}_n$. For each nonzero element of $\boldsymbol{V}$, the mapper computes the quantity

$$p_{z|ij} = \Pr[\, z \mid i, j \,] = \boldsymbol{W}_{n,iz} \boldsymbol{K}_{n,zz}^{-1} (\boldsymbol{V}_{ij} / \boldsymbol{W}_{n,i*} \boldsymbol{K}_n^{-1} \boldsymbol{H}_{n,*j}) \boldsymbol{H}_{n,zj}^{\mathrm{T}},$$

and outputs pairs $(i, p_{z|ij})$, $(j, p_{z|ij})$, $(z, p_{z|ij})$. Thus there is one *group* per row, one per column, and one per topic. The reducer for row i computes $\tilde{\boldsymbol{W}}_{n+1,i*}$ using the transformation:

$$\begin{aligned}
\tilde{\boldsymbol{W}}_{n+1,iz} &\leftarrow \tilde{\boldsymbol{W}}_{n,iz} \boldsymbol{K}_{n,zz}^{-1} \circ (\boldsymbol{V}_{i*} / \tilde{\boldsymbol{W}}_{n,i*} \boldsymbol{K}_n^{-1} \boldsymbol{H}_n) \boldsymbol{H}_{n,z*}^{\mathrm{T}} \\
&= \sum_j \tilde{\boldsymbol{W}}_{n,iz} \boldsymbol{K}_{n,zz}^{-1} \circ (\boldsymbol{V}_{ij} / \tilde{\boldsymbol{W}}_{n,i*} \boldsymbol{K}_n^{-1} \boldsymbol{H}_{n,*j}) \boldsymbol{H}_{n,zj}^{\mathrm{T}} \\
&= \sum_j p_{z|ij}.
\end{aligned}$$

Similarly,

$$\begin{aligned}
\boldsymbol{H}_{n+1,zj} &\leftarrow \sum_i p_{z|ij} \\
\boldsymbol{K}_{n+1,zz} &\leftarrow \sum_{i,j} p_{z|ij}
\end{aligned}$$

Thus reducers simply sum up the entries within each group. Since summation is distributive, preaggregation can be performed in a combine step at each mapper. To output the result in blocked form, groups are assigned to reducers according to the blocking—e.g., a single reducer processes all rows (groups) of the first block of $\tilde{\boldsymbol{W}}_{n+1}$. Assuming $m, n = O(N)$, the overall time complexity per iteration is $O(d^{-1} N r)$.

Multiplicative updates (MULT). Lee et.al [22] proposed multiplicative update rules for non-negative matrix factorization under L_{GKL}. Later [23], they refined the L_{GKL} rules and developed rules for L_{SL}. The refined update rules can be seen as a rescaled version of gradient descent, where the step sizes are computed individually for each parameter. In all cases, the rules are multiplicative in that each factor gets multiplied by some update term, which varies from method to method. Each iteration is non-increasing in the loss, and the factor matrices converge to a stationary point of the loss function.

We first consider the rules for GKL given in [22]:

$$\tilde{\boldsymbol{W}}_{n+1} \leftarrow \boldsymbol{W}_n \circ (\boldsymbol{V} / \boldsymbol{W}_n \boldsymbol{H}_n) \boldsymbol{H}_n^{\mathrm{T}} \tag{27a}$$

$$\boldsymbol{W}_{n+1} \leftarrow \tilde{\boldsymbol{W}}_{n+1} \operatorname{diag}(1 / \operatorname{colSums}[\tilde{\boldsymbol{W}}_{n+1}]) \tag{27b}$$

$$\boldsymbol{H}_{n+1} \leftarrow \boldsymbol{W}_{n+1}^{\mathrm{T}} (\boldsymbol{V} / \boldsymbol{W}_{n+1} \boldsymbol{H}_n) \circ \boldsymbol{H}_n, \tag{27c}$$

These are almost the update rules (25) of EM, but $\boldsymbol{W}_{n+1}$ is used instead of $\boldsymbol{W}_n$ when computing $\boldsymbol{H}_{n+1}$. As a consequence, updates of $\boldsymbol{W}$ and $\boldsymbol{H}$ cannot be performed simultaneously anymore, and

two scans of $\boldsymbol{V}$ are required. The refined rules for GKL are

$$\boldsymbol{W}_{n+1} \leftarrow \boldsymbol{W}_n \circ (\boldsymbol{V}/\boldsymbol{W}_n\boldsymbol{H}_n)\boldsymbol{H}_n^{\mathrm{T}} \operatorname{diag}(1/\operatorname{rowSums}[\boldsymbol{H}_n]),$$
$$\boldsymbol{H}_{n+1} \leftarrow \boldsymbol{H}_n \circ \operatorname{diag}(1/\operatorname{colSums}[\boldsymbol{W}_{n+1}])\boldsymbol{W}_{n+1}^{\mathrm{T}}(\boldsymbol{V}/\boldsymbol{W}_{n+1}\boldsymbol{H}_n).$$

These update rules can be seen as symmetric versions of (27); they work better in practice. Thus, when we refer to MULT for L_{GKL}, we refer to the refined rules above. The time complexity remains $O(Nr)$. The rules for L_{SL} are given by:

$$\boldsymbol{H}_{n+1} \leftarrow \boldsymbol{H}_n \circ \frac{\boldsymbol{W}_n^{\mathrm{T}}\boldsymbol{V}}{\boldsymbol{W}_n^{\mathrm{T}}\boldsymbol{W}_n\boldsymbol{H}_n},$$
$$\boldsymbol{W}_{n+1} \leftarrow \boldsymbol{W}_n \circ \frac{\boldsymbol{V}\boldsymbol{H}_{n+1}^{\mathrm{T}}}{\boldsymbol{W}_n\boldsymbol{H}_{n+1}\boldsymbol{H}_{n+1}^{\mathrm{T}}}.$$

Sparsity of $\boldsymbol{V}$ is readily exploited: $\boldsymbol{W}^{\mathrm{T}}\boldsymbol{V}$ and $\boldsymbol{V}\boldsymbol{H}_{n+1}^{\mathrm{T}}$ each can be computed in a single scan of the nonzero elements of $\boldsymbol{V}$. The overall time complexity is $O(Nr + (m+n)r^2)$.

Liu et al. [25] give MapReduce versions of MULT; the underlying ideas are the same as for distributed EM [12]. Minor modifications are needed to match the refined rules or the L_{SL} rules.

A.2. Generic Algorithms

Generic algorithms can be used to solve our generalized problem statement. We first discuss gradient descent, which has been applied successfully to large-scale matrix factorization. We then summarize some recent work in the area of distributed SGD.

Gradient descent (GD). Gradient descent is a well-known optimization technique. The idea is to compute the direction of steepest descent at the current value of the parameters, and then take a small step in this direction. We describe the algorithm in terms of the local loss functions L_{ij}. The gradients are given by:

$$\frac{\partial}{\partial \boldsymbol{W}_{i*}} L(\boldsymbol{W}, \boldsymbol{H}) = \frac{\partial}{\partial \boldsymbol{W}_{i*}} \sum_{(i',j)\in Z} L_{i'j}(\boldsymbol{W}_{i'*}, \boldsymbol{H}_{*j}) = \sum_{j \in Z_{i*}} \frac{\partial}{\partial \boldsymbol{W}_{i*}} L_{ij}(\boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}),$$

where $Z_{i*} = \{\, j \colon (i,j) \in Z \,\}$. This means that the gradient w.r.t. $\boldsymbol{W}_{i*}$ depends on $\boldsymbol{H}$ and row i of both the loss matrix $\boldsymbol{L}$ and $\boldsymbol{W}$. Similarly, we have

$$\frac{\partial}{\partial \boldsymbol{H}_{*j}} L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{i \in Z_{*j}} \frac{\partial}{\partial \boldsymbol{W}_{*j}} L_{ij}(\boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}),$$

where $Z_{*j} = \{\, i \colon (i,j) \in Z \,\}$. The gradient w.r.t. to $\boldsymbol{W}$ is given by the matrix of first-order partial derivatives

$$\boldsymbol{W}_n^{\nabla} = \frac{\partial}{\partial \boldsymbol{W}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) = \begin{pmatrix} \frac{\partial}{\partial \boldsymbol{W}_{1*}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) \\ \frac{\partial}{\partial \boldsymbol{W}_{2*}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) \\ \vdots \\ \frac{\partial}{\partial \boldsymbol{W}_{m*}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) \end{pmatrix}.$$

Similarly,

$$\boldsymbol{H}_n^{\nabla} = \frac{\partial}{\partial \boldsymbol{H}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) = \left(\frac{\partial}{\partial \boldsymbol{H}_{*1}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) \quad \cdots \quad \frac{\partial}{\partial \boldsymbol{H}_{*n}} L(\boldsymbol{W}_n, \boldsymbol{H}_n) \right).$$

Then, GD performs the following iteration

$$\boldsymbol{W}_{n+1} = \boldsymbol{W}_n - \epsilon_n \boldsymbol{W}_n^{\nabla}$$
$$\boldsymbol{H}_{n+1} = \boldsymbol{H}_n - \epsilon_n \boldsymbol{H}_n^{\nabla}.$$

If the conditions on $\{ \epsilon_n \}$ and L given in Sec. 4.2 hold, and a projection term is added to the above equations, the algorithm converges asymptotically to the set of limit points of the projected ODE $\dot{\theta} = -L'(\theta) + z$, where $\theta = (\boldsymbol{W}, \boldsymbol{H})$ and, as before, z is the minimum force to keep the solution in the constraint region.

GD converges very slowly and many iterations may be required to approach a stationary point. To get reasonable convergence speed, Newton or quasi-Newton methods replace the step sizes by (estimates of) the inverse Hessian of L at the current parameter estimate. A scalable and memory-efficient method is L-BFGS-B [8]. If L is non-convex, GD may get stuck in a local optimum. Especially in its standard form, GD is thus not well suited for loss functions that have many "small bumps", i.e., many bad local minima.

Both first-order and second-order GD methods can be distributed using MapReduce [26, 13]. The key idea of this approach to arbitrarily partition the loss matrix across a cluster of nodes. Each node sums up the partial gradients of its part of the data, the partial gradients are then summed up at the reducers. The authors argue that this can be done conveniently by using a query language on top of MapReduce. Once the gradient has been computed, a master node will perform the update of the parameter values using, for example, the L-BFGS-B method.

Algorithm 3 PSGD/ISGD for Matrix Factorization

Require: $\boldsymbol{Z}$, $\boldsymbol{W}_0$, $\boldsymbol{H}_0$, degree of parallelism d
  $\boldsymbol{W} \leftarrow \boldsymbol{W}_0$
  $\boldsymbol{H} \leftarrow \boldsymbol{H}_0$
  Randomly divide $\boldsymbol{Z}$ into d partitions $Z_1, \ldots, Z_d$
  while not converged **do** */* epoch */*
    Pick step size ϵ
    Distribute $\boldsymbol{W}$ and $\boldsymbol{H}$ */* ISGD: only in first iteration */*
    for $b = 1, \ldots, d$ **do** */* in parallel */*
      Run SGD on the training points in $\boldsymbol{Z}^b$ (step size $= \epsilon$)
    end for
    Collect $\boldsymbol{W}$ and $\boldsymbol{H}$ from all nodes and average */* ISGD: only in last iteration */*
  end while

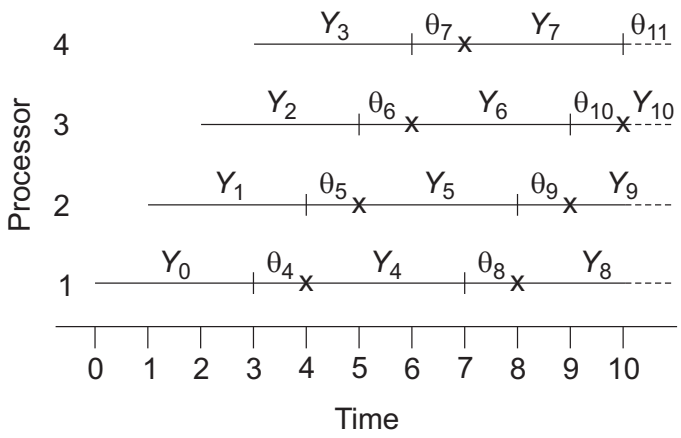


Figure 8: Pipelined SGD ($d = 3$, $t_g = 3$, $t_u = 1$)

Partitioned stochastic gradient descent (PSGD, ISGD). PSGD and ISGD both are recent approaches to distribute SGD without using stratification; see Algorithm 3. The idea is to partition the data randomly into d partitions, and run SGD independently and in parallel on each partition. Results are averaged in a *parameter mixing* step after either each epoch (PSGD [16, 27]) or once after convergence on each partition (ISGD [26, 27, 33]); observe that PSGD requires periodic synchronization between the partition-processing tasks whereas, for ISGD, processing on the different partitions can proceed in a mutually independent fashion. Both approaches can be implemented naturally on MapReduce. Compared to DSGD, ISGD is slightly more efficient (since there is no synchronization) whereas PSGD is slightly less efficient (since there are additional averaging steps). In the setting of matrix factorization, it is possible to reorder a set of rows of $\boldsymbol{W}$ and corresponding columns of $\boldsymbol{H}$ without affecting the value of the loss function. It follows that the loss function has many global minima that correspond to many different values of the factor matrices. In ISGD, the SGD processes on different partitions tend to converge to different solutions; the average of these local solutions is usually not a global loss minimizer. PSGD performs better and does converge, but it is outperformed by DSGD and, in some cases, even L-BFGS; see Sec. 7.

B. Parallelization Techniques for Stochastic Approximation

For completeness, we summarize standard approaches for parallel stochastic approximation [21, Ch. 12]. These approaches do not map naturally to MapReduce, and are designed for the case in which the computation of the update term is expensive (e.g., requires a simulation) and communication is rather cheap (e.g., few parameters). Neither assumption holds for matrix factorization, but our approaches are inspired by the techniques below.

Pipelined stochastic gradient descent. The *pipelined computation model* is based on a "delayed update" scheme in which the gradient estimate used in the nth step is based on the

parameter value from the $(n-d)$th step, where d is the number of available processors, e.g.,

$$\theta_{n+1} = \theta_n - \epsilon Y_{n-d},$$

where $Y_k = \hat{L}'(\theta_k)$ for $k \geq 0$. (We fix the step size ϵ for ease of notation.) This permits a scheme in which $d+1$ processors compute gradient estimates and parameter values in a pipelined fashion. Figure 8 illustrates the technique for the case $d = 3$, assuming that it takes $t_g = 3$ time units to compute a gradient and $t_u = 1$ time unit to update a parameter value. The scheme is initiated by choosing θ_0 and setting $\theta_3 = \theta_2 = \theta_1 = \theta_0$. At time $n = 0$, processor 1 begins to compute the update term Y_0. This computation completes at time $n = 3$ (marked with a "|"), at which point processor 1 begins to compute $\theta_4 = \theta_3 - \epsilon Y_0$, finishing at time $n = 4$ (marked with an "x"). Similarly, processor 2 begins to compute the update term Y_1 at time $n = 1$. At time $n = 4$, this computation completes, and processor 2 uses the value of θ_4 that has just been computed by processor 1 to compute $\theta_5 = \theta_4 - \epsilon Y_1$, finishing this computation at time $n = 5$. The other processors behave similarly. This scheme can be extended to handle variable updating delays.

Decentralized stochastic gradient descent In the *distributed and decentralized network model*, both parameter updates and computation of update terms take place in a distributed and decentralized fashion. This powerful model is a generalization of the pipelined computation model. Both the components of the parameter vector and the loss function are distributed across the set of d processors. Processor p is responsible for component θ^p and partial loss $L_p(\theta)$ such that:

$$\theta = \begin{pmatrix} \theta^1 \\ \theta^2 \\ \vdots \\ \theta^d \end{pmatrix} \quad \text{and} \quad L(\theta) = \sum_{p=1}^{d} L_p(\theta).$$

The processors operate in parallel and (potentially) at different speeds; communication is asynchronous. Thus at any given point in "real time" (wall clock time), each processor resides at a different time point in "iterate time" (step number). More specifically, the $(n+1)$st iteration at processor p involves the following steps:

1. *Estimate* $\hat{\theta}_{n,p}$. Obtain an estimate $\hat{\theta}_{n,p}$ of the entire parameter vector by using θ_n^p for the pth component (the current value of the component managed by p) and the most recently received value for all other components (from step 5).

2. *Compute update terms.* For each processor q (including p), compute (or estimate) update term

$$Y_{n,p}^q = -\frac{\partial L_p(\hat{\theta}_{n,p})}{\partial \theta^q}.$$

3. *Communicate update terms.* For each processor q, send $Y_{n,p}^q$ to q.

4. *Compute* θ^p_{n+1}. Add all unprocessed update terms Y^p_* to θ^p_n. This includes update terms received from other nodes as well as the update term $Y^p_{n,p}$ computed in the previous step. Do not wait for any "missing" update terms, and if multiple update terms have been received from a single processor q, process them all.

$$\theta^p_{n+1} = \theta^p_n + \epsilon \sum \{ \text{ unprocessed update terms } Y^p_* \} .$$

5. *Broadcast* θ^p_{n+1}. Broadcast the new parameter component to all other nodes.

Weak convergence results for this process model are discussed in [21, Ch. 12]. The proofs proceed by a careful treatment of "iterate time", and then use appropriate time scale changes to obtain "real time" results.

C. Example Loss Functions and Derivatives

Table 2 displays the definitions of several commonly used loss functions as mentioned in Section 7: nonzero squared loss (L_{NZSL}), non-zero squared loss with L_2 regularization (L_{L2}), nonzero squared loss with a nonzero-weighted L_2 term (L_{NZL2}), KL divergence (L_{KL}), and generalized KL divergence (L_{GKL}). In the table, $\|\boldsymbol{A}\|_{\text{F}}$ denotes the Frobenius norm of a matrix $\boldsymbol{A}$: $\|\boldsymbol{A}\|_{\text{F}} = \left(\sum_i \sum_j a_{ij}^2\right)^{1/2}$.

Our methods for obtaining a rank-r approximate factorization $\boldsymbol{V} \approx \boldsymbol{W}\boldsymbol{H}$ of an $m \times n$ input matrix $\boldsymbol{V}$ require that we represent each loss function L as a sum of local losses over points in the training set Z, i.e., $L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{(i,j)\in Z} L_{ij}(\boldsymbol{W}, \boldsymbol{H})$, where $L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$ for an appropriate function l, so that the gradient of L can be decomposed as a sum of local-loss gradients: $L'(\boldsymbol{W}, \boldsymbol{H}) = \sum_{(i,j)\in Z} L'_{ij}(\boldsymbol{W}, \boldsymbol{H})$. For each loss function L considered, Table 2 gives formulas for the components of the local-loss gradient L'_{ij}. In these formulas, N_{i*} and N_{*j} denote the number of nonzero elements in row i and column j of the matrix $\boldsymbol{V}$. Moreover, $J = \{1, \ldots, m\} \times \{1, \ldots, n\}$ and $B = \{1, \ldots, d\} \times \{1, \ldots, d\}$. Finally, for a $u \times v$ matrix $\boldsymbol{A}$, the quantities rowSums($\boldsymbol{A}$) and colSums($\boldsymbol{A}$) denote the $u \times 1$ column vector containing the row sums of $\boldsymbol{A}$ and the $1 \times v$ row vector containing the column sums of $\boldsymbol{A}$; thus, for example, colSums($\boldsymbol{W}$) and rowSums($\boldsymbol{H}$) are each vectors of length r.

As can be seen, special care has to be taken with regularization terms when representing L as a sum of local losses. In the case of L_{L2}, for example, we proceed as follows. Recall from Sec. 2 our running assumption that there is at least one training point in every row and in every column of $\boldsymbol{V}$. Since $Z = \{(i,j) : \boldsymbol{V}_{ij} > 0\}$, this means that $N_{i*} > 0$ and $N_{*j} > 0$ for all i and j. Then we have

$$\begin{aligned} \|\boldsymbol{W}\|_{\text{F}}^2 = \sum_{i=1}^m \|\boldsymbol{W}_{i*}\|_{\text{F}}^2 &= \sum_{i=1}^m N_{i*} \frac{\|\boldsymbol{W}_{i*}\|_{\text{F}}^2}{N_{i*}} \\ &= \sum_{i=1}^m \sum_{j=1}^n I[(i,j) \in Z] \frac{\|\boldsymbol{W}_{i*}\|_{\text{F}}^2}{N_{i*}} = \sum_{(i,j)\in Z} \frac{\|\boldsymbol{W}_{i*}\|_{\text{F}}^2}{N_{i*}} \end{aligned} \tag{28}$$

and, similarly, $\|\boldsymbol{H}\|_{\mathrm{F}}^2 = \sum_{(i,j)\in Z} \|\boldsymbol{H}_{*j}\|_{\mathrm{F}}^2 / N_{*j}$. (Here $I[A]$ denotes the indicator function of A.) These results lead directly to the representation of L_{L2} in Table 2. A similar algebraic manipulation is used to represent L_{NZL2}.

The case of L_{GKL} merits additional discussion. First recall that, after randomly permuting the rows and column of the training matrix $\boldsymbol{Z}$, we partition $\boldsymbol{Z}$ into d^2 blocks, and partition each of the factor matrices conformingly into d blocks as $\boldsymbol{W} = (\boldsymbol{W}^1, \ldots, \boldsymbol{W}^d)^{\mathrm{T}}$ and $\boldsymbol{H} = (\boldsymbol{H}^1, \ldots, \boldsymbol{H}^d)$. As before, we denote by Z^b the set of training points that lie in block $b \in B$. A stratum comprises d blocks, selected such that each pair of blocks has no row or column indices in common. We originally required that the local loss at a point $(i,j) \in Z$ be of the form $l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$ mentioned above, which ensures that SGD can be run independently within each block in the stratum. Observe, however, that we need only require that the local losses be of the form $l(\boldsymbol{V}_{ij}, \boldsymbol{W}^{b(i*)}, \boldsymbol{H}^{b(*j)})$, where $b(i*)$ denotes the block of $\boldsymbol{W}$ that contains $\boldsymbol{W}_{i*}$ and $b(*j)$ denotes the block of $\boldsymbol{H}$ that contains $\boldsymbol{H}_{*j}$. This looser definition preserves the interchangeability structure, so that updates to parameter values for a given block will not affect parameter values corresponding to other blocks, and SGD can still be executed in a distributed manner within the stratum. We represent L_{GKL} as a sum of losses in this looser sense. In addition to allowing a more general representation of each local loss, we also use two different decompositions of L_{GKL}. The first (resp., second) representation is more amenable to calculating derivatives with respect to the $\boldsymbol{W}_{ik}$ (resp., $\boldsymbol{H}_{kj}$) factors.

Denote by N_{i*}^b (resp., N_{*j}^b) the number of training points in substratum Z^b that appear in row i (resp., column j) of the training matrix $\boldsymbol{Z}$:

$$N_{i*}^b = \left|\{\, j : (i,j) \in Z^b \,\}\right| \qquad \text{and} \qquad N_{*j}^b = \left|\{\, i : (i,j) \in Z^b \,\}\right|.$$

We assume that $N_{i*}^b > 0$ for each row i that intersects block $\boldsymbol{Z}^b$ and that $N_{*j}^b > 0$ for each column j that intersects $\boldsymbol{Z}^b$; otherwise, we can always add additional (zero-valued) training points to the training set Z.[10] For $b \in B$, denote by $J_1^b = \{\, i : (i,j) \in Z^b \,\}$ and $J_2^b = \{\, j : (i,j) \in Z^b \,\}$ the sets of first indices and second indices of points—both zero and nonzero—in block b. Set $Q^b = \sum_{i\in J_1^b} \sum_{j \in J_2^b} [\boldsymbol{W}\boldsymbol{H}]_{ij}$ and note that

$$Q^b = \sum_{i\in J_1^b} \sum_{j\in J_2^b} \boldsymbol{W}_{i*}\boldsymbol{H}_{*j} = \sum_{i\in J_1^b} \boldsymbol{W}_{i*} \cdot \mathrm{rowSums}(\boldsymbol{H}^b) = \sum_{(i,j)\in Z^b} \frac{\boldsymbol{W}_{i*}}{N_{i*}^b} \cdot \mathrm{rowSums}(\boldsymbol{H}^b),$$

[10] For each such added point (i,j), we define the quantity $\boldsymbol{V}_{ij} \log\big(\boldsymbol{V}_{ij}/[\boldsymbol{W}\boldsymbol{H}]_{ij}\big)$ that appears in the definitions of L_{ij}^W and L_{ij}^H below—as well as the quantity $\boldsymbol{V}_{ij}/[\boldsymbol{W}\boldsymbol{H}]_{ij}$ that appears in the definitions of $\partial L_{ij}^W/\partial \boldsymbol{W}_{ik}$ and $\partial L_{ij}^H/\partial \boldsymbol{H}_{kj}$—to be equal to 0 for all $\boldsymbol{W}$ and $\boldsymbol{H}$.

where the final equality follows from a manipulation as in (28). We then have

$$\begin{aligned} L_{\text{GKL}} &= \sum_{(i,j)\in Z} (\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij}) + \sum_{(i,j)\in J} [\boldsymbol{W}\boldsymbol{H}]_{ij} \\ &= \sum_{b\in B} \Bigg(\sum_{(i,j)\in Z^b} (\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij}) + Q^b \Bigg) \\ &= \sum_{b\in B} \sum_{(i,j)\in Z^b} \Big(\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij} + \frac{\boldsymbol{W}_{i*}}{N^b_{i*}} \cdot \text{rowSums}(\boldsymbol{H}^b) \Big). \end{aligned}$$

Thus we obtain the representation $L_{\text{GKL}} = \sum_{(i,j)\in Z} L^W_{ij}$, where

$$L^W_{ij} = \boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij} + \frac{\boldsymbol{W}_{i*}}{N^b_{i*}} \cdot \text{rowSums}(\boldsymbol{H}^{b(*j)}).$$

This is a local loss of the "loose" form discussed above. This representation of L_{GKL} is convenient for computing derivatives with respect to the $\boldsymbol{W}$ factors. Observe that for each $(i,j) \in Z$, we have $\partial L^W_{ij} / \partial \boldsymbol{W}_{i'k} = 0$ for $i' \neq i$, so that—if we are running SGD on a block $\boldsymbol{Z}^b$ and are estimating the gradient based on a sampled training point $(i,j) \in Z^b$—only the elements of $\boldsymbol{W}_{i*}$ need to be updated, and we retain computational efficiency as in Algorithm 1. In a similar manner, we can derive an alternate representation $L_{\text{GKL}} = \sum_{(i,j)\in Z} L^H_{ij}$, where

$$L^H_{ij} = \boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij} + \frac{\boldsymbol{H}_{*j}}{N^b_{*j}} \cdot \text{colSums}(\boldsymbol{W}^{b(i*)}).$$

This decomposition is useful for computing derivatives with respect to the $\boldsymbol{H}$ factors. With an abuse of notation, we write $\partial L_{ij} / \partial \boldsymbol{W}_{ik}$ for $\partial L^W_{ij} / \partial \boldsymbol{W}_{ik}$ and $\partial L_{ij} / \partial \boldsymbol{H}_{kj}$ for $\partial L^H_{ij} / \partial \boldsymbol{H}_{kj}$. Although the "gradient" L'_{ij} that we have just defined is not actually the gradient of some well defined local loss function L_{ij}, it nonetheless holds that $\sum_{(i,j)\in Z^b} L'_{ij}$ is equal to the gradient of L_{GKL} on Z^b, which is all that we need for DSGD. The final derivative formulas are given in Table 2.

Table 2: Examples of loss functions and derivatives

Loss Function	**Definition and Derivatives**
L_{NZSL}	$L_{\text{NZSL}} = \sum_{(i,j)\in Z} (\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2$
	$\frac{\partial}{\partial \boldsymbol{W}_{ik}} L_{ij} = -2(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij}) \boldsymbol{H}_{kj}$
	$\frac{\partial}{\partial \boldsymbol{H}_{kj}} L_{ij} = -2(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij}) \boldsymbol{W}_{ik}$

Table 2: Examples of loss functions and derivatives (continued)

Loss Function	Definition and Derivatives
L_{L2}	$L_{\text{L2}} = L_{\text{NZSL}} + \lambda \left(\|\boldsymbol{W}\|_{\text{F}}^2 + \|\boldsymbol{H}\|_{\text{F}}^2 \right)$ $= \sum_{(i,j)\in Z} \left[(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2 + \lambda \left(\frac{\|\boldsymbol{W}_{i*}\|_{\text{F}}^2}{N_{i*}} + \frac{\|\boldsymbol{H}_{*j}\|_{\text{F}}^2}{N_{*j}} \right) \right]$ $\frac{\partial}{\partial \boldsymbol{W}_{ik}} L_{ij} = -2(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})\boldsymbol{H}_{kj} + 2\lambda \frac{\boldsymbol{W}_{ik}}{N_{i*}}$ $\frac{\partial}{\partial \boldsymbol{H}_{kj}} L_{ij} = -2(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})\boldsymbol{W}_{ik} + 2\lambda \frac{\boldsymbol{H}_{kj}}{N_{*j}}$
L_{NZL2}	$L_{\text{NZL2}} = L_{\text{NZSL}} + \lambda \left(\|\boldsymbol{N}_1\boldsymbol{W}\|_{\text{F}}^2 + \|\boldsymbol{H}\boldsymbol{N}_2\|_{\text{F}}^2 \right)$ $= \sum_{(i,j)\in Z} \left[(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2 + \lambda \left(\|\boldsymbol{W}_{i*}\|_{\text{F}}^2 + \|\boldsymbol{H}_{*j}\|_{\text{F}}^2 \right) \right]$ $\frac{\partial}{\partial \boldsymbol{W}_{ik}} L_{ij} = -2(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})\boldsymbol{H}_{kj} + 2\lambda \boldsymbol{W}_{ik}$ $\frac{\partial}{\partial \boldsymbol{H}_{kj}} L_{ij} = -2(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})\boldsymbol{W}_{ik} + 2\lambda \boldsymbol{H}_{kj}$
L_{KL}	$L_{\text{KL}} = \sum_{(i,j)\in Z} (\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}})$ $\frac{\partial}{\partial \boldsymbol{W}_{ik}} L_{ij} = -\boldsymbol{H}_{kj} \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}}$ $\frac{\partial}{\partial \boldsymbol{H}_{kj}} L_{ij} = -\boldsymbol{W}_{ik} \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}}$

Table 2: Examples of loss functions and derivatives (continued)

Loss Function	**Definition and Derivatives**
L_{GKL}	$L_{\text{GKL}} = \sum_{(i,j)\in Z} (\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij}) + \sum_{(i,j)\in J} [\boldsymbol{W}\boldsymbol{H}]_{ij}$ $= \sum_{b\in B} \sum_{(i,j)\in Z^b} \Big(\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij} + \frac{W_{i*}}{N^b_{i*}} \cdot \text{rowSums}(\boldsymbol{H}^b) \Big)$ $= \sum_{b\in B} \sum_{(i,j)\in Z^b} \Big(\boldsymbol{V}_{ij} \log \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} - \boldsymbol{V}_{ij} + \frac{H_{*j}}{N^b_{*j}} \cdot \text{colSums}(\boldsymbol{W}^b) \Big)$ $\frac{\partial}{\partial \boldsymbol{W}_{ik}} L_{ij} \stackrel{\text{def}}{=} \frac{\partial}{\partial \boldsymbol{W}_{ik}} L^W_{ij} = -\boldsymbol{H}_{kj} \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} + \frac{\text{rowSums}(\boldsymbol{H}^{b(*j)})_k}{N^b_{i*}}$ $\frac{\partial}{\partial \boldsymbol{H}_{kj}} L_{ij} \stackrel{\text{def}}{=} \frac{\partial}{\partial \boldsymbol{H}_{kj}} L^H_{ij} = -\boldsymbol{W}_{ik} \frac{\boldsymbol{V}_{ij}}{[\boldsymbol{W}\boldsymbol{H}]_{ij}} + \frac{\text{colSums}(\boldsymbol{W}^{b(i*)})_k}{N^b_{*j}}$