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

ABSTRACT

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, e.g., MapReduce. DSGD can handle 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.

Categories and Subject Descriptors

G.4 [**Mathematics of Computing**]: Mathematical Software—*Parallel and vector implementations*

General Terms

Algorithms, Experimentation, Performance

Keywords

distributed matrix factorization, stochastic gradient descent, MapReduce, recommendation system

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 [8, 12, 13, 15, 16]. 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 (see Sec. 2), 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 massive scales, distributed algorithms for matrix factorization are essential to achieving reasonable performance [8, 9, 16, 20]. 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. 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 that has been shown, in a sequential setting, to be very effective for matrix factorization [13]. Although the generic SGD algorithm (Sec. 3) is not embarrassingly parallel and hence cannot directly scale to very large data, 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 (Sec. 4) 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. Our proof rests on stochastic approximation theory and regenerative process theory.

We then specialize SSGD to obtain a novel distributed matrix-factorization algorithm, called DSGD (Sec. 5). 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. The DSGD algorithm repeatedly selects a stratum according to the general SSGD procedure and processes the stratum in a distributed fashion. Importantly, both matrix and factors are fully distributed, so that DSGD has low memory requirements and scales to matrices with millions of rows, millions of columns, and billions of nonzero elements. When DSGD is implemented in MapReduce (Sec. 6) and compared to state-of-the-art distributed algorithms for matrix factorization, our experiments (Sec. 7) 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. See [10] for a treatment of other loss functions.

2. EXAMPLE AND PRIOR WORK

To gain understanding about applications of matrix factorizations, consider the "Netflix problem" [3] 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 (denoted by "?"); entries with a high predicted rating are then recommended to users for viewing. This matrix-factorization approach to recommender systems has been successfully applied in practice; see [13] for an excellent discussion of the underlying intuition.

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

$$L = \sum_{(i,j) \in Z} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}) \tag{1}$$

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 [19]. 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 $Z = \{(i,j) : \boldsymbol{V}_{ij} \neq 0\}$. As mentioned above, such decompositions naturally arise when zeros represent missing data. Our algorithms can handle other decompositions as well; see [10].

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 as *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. [8] provide an EM-based algorithm, and Liu et al. [16] provide a multiplicative-update method. In [16], the latter MULT approach is also applied to squared loss and to nonnegative matrix factorization with an "exponential" loss function. Each of these algorithms in essence takes an embarrassingly parallel matrix factorization algorithm developed previously and directly distributes it across the MapReduce cluster. Zhou et al. [20] 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. See [10] for details on the foregoing algorithms.

Generic algorithms are able to handle all differentiable loss functions that decompose into summation form. A simple approach is distributed gradient descent (DGD) [9, 11, 17], 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 [6]. Partitioned SGD 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 [11, 18]) or once at the end (ISGD [17, 18, 21]). 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 SGD. We also establish basic properties of this stochastic procedure.

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

$$\theta_{n+1} = \theta_n - \epsilon_n \hat{L}'(\theta_n), \tag{2}$$

where n denotes the step number and $\{\epsilon_n\}$ is a sequence of decreasing step sizes. Since $-L'(\theta_n)$ is the direction of steepest descent, (2) constitutes a noisy version of gradient descent.

Stochastic approximation theory can be used to show that, under certain regularity conditions [14], 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 variety of 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 [15], which corresponds to setting $H = \{\, \theta : \theta \geq 0 \,\}$. The projected algorithm takes the form

$$\theta_{n+1} = \Pi_H\big[\theta_n - \epsilon_n \hat{L}'(\theta_n)\big]. \tag{3}$$

In addition to the set of stationary points, the projected process may converge to the set of "chain recurrent" points [14], which are influenced by the boundary of the constraint set H.

3.2 SGD for Matrix Factorization

To apply SGD to matrix factorization, we set $\theta = (\boldsymbol{W}, \boldsymbol{H})$ and decompose the loss L as in (1) for an appropriate training set Z and local loss function l. Denote by $L_z(\theta) = L_{ij}(\theta) = l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j})$ the local loss at position $z = (i,j)$. Then $L'(\theta) = \sum_z L'_z(\theta)$ by the sum rule for differentiation. DGD methods (see Sec. 2) exploit the summation form of $L'(\theta)$ at each iteration by computing the local gradients $L'_z(\theta)$ in parallel and summing them. In contrast to this exact computation of the overall gradient, SGD obtains noisy gradient estimates by scaling up *just one* of the local gradients, i.e., $\hat{L}'(\theta) = NL'_z(\theta)$, 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 selecting a random training point $(i,j) \in Z$, we need to update only $\boldsymbol{W}_{i*}$ and $\boldsymbol{H}_{*j}$, 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

$$\frac{\partial}{\partial \boldsymbol{W}_{i'k}} L_{ij}(\boldsymbol{W}, \boldsymbol{H}) = \begin{cases} 0 & \text{if } i \neq i' \\ \frac{\partial}{\partial \boldsymbol{W}_{ik}} l(\boldsymbol{V}_{ij}, \boldsymbol{W}_{i*}, \boldsymbol{H}_{*j}) & \text{otherwise} \end{cases} \tag{4}$$

and

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

for $1 \leq k \leq r$. SGD is sometimes referred to as *online learning* or *sequential gradient descent* [4]. 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 [5], and it has been observed in the case of large-scale matrix factorization [13].

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 q (> 1) "local" loss functions:

$$L(\theta) = w_1 L_1(\theta) + w_2 L_2(\theta) + \ldots + w_q L_q(\theta), \tag{6}$$

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 the w_s; they may be altered to arbitrary values (subject to the constraints above) by appropriately rescaling 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 $\theta_{n+1} = \Pi_H\big[\theta_n - \epsilon_n \hat{L}'_{\gamma_n}(\theta_n)\big]$. 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.

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 [14, Sec. 5.6]. We distinguish step-size conditions, loss conditions, stratification conditions, and stratum-sequence conditions. Step-size conditions involve the sequence $\{\epsilon_n\}$: It has to approach 0 at the "right speed" in that $\epsilon_n \to 0$, $\sum_{i=1}^n \epsilon_i \to \infty$, and $\sum_{i=0}^n \epsilon_i^2 < \infty$ as $n \to \infty$. I.e., ϵ_n approaches 0 slowly enough so that the algorithm does not get stuck far away from the optimum, but fast enough to ensure convergence. The simplest valid choice is $\epsilon_n = 1/n$. A sufficient set of loss conditions is that the constraint set H in which L is defined is a hyperrectangle and that L is bounded and twice continuously differentiable on H.[1] Regarding stratification, we require that the estimates $\hat{L}'_s(\theta)$ of the gradient $L'_s(\theta)$ of stratum s are unbiased, have bounded second moment for $\theta \in H$, and do not depend on the past. See [10] for a more precise statement of these conditions, which are satisfied in most matrix factorization problems. Finally, we give a sufficient condition on the stratum sequence.

CONDITION 1. *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 $\theta \in H$,*

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

almost surely.

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

If all conditions 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 [14, Sec. 4.3]. As shown in [14], the limit points consist of the set of stationary points of L in H ($z = 0$), as well as a set of chain-recurrent points on the boundary of H ($z \neq 0$). In our setting, the limit point to which SSGD converges is typically a good local minimum of the loss function (Sec. 7), and the crux of showing that SSGD converges is showing that Condition 1 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 1 holds. Our key assumption is that the sequence $\{\gamma_n\}$ is *regenerative* [2, 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 $C_k = \{\gamma_{\beta(k-1)}, \gamma_{\beta(k-1)+1}, \ldots, \gamma_{\beta(k)-1}\}$ 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 1. *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 1 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 [2, Sec. VI.3], this condition—in the presence of the finite-moment condition on τ_1—is equivalent to requiring 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 $R_{2,n} = n - \beta\big(c(n)\big) + 1$.

[1]Points at which the loss is non-differentiable may arise in matrix factorization, e.g., when l_1-regularization is used. SSGD can handle with this phenomenon using "subgradients" [14, Sec. 5.6].

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

Thus

$$\begin{aligned}\frac{\sum_{i=0}^{n}(I_{\gamma_i=s}-w_s)}{n^\alpha} &= \frac{\sum_{k=1}^{c(n)} X_k(s)+R_{1,n}}{\left(\sum_{k=1}^{c(n)}\tau_k+R_{2,n}\right)^\alpha} \\ &= \frac{\sum_{k=1}^{c(n)} X_k(s)}{c(n)^\alpha}\left(\frac{\sum_{k=1}^{c(n)}\tau_k}{c(n)}+\frac{R_{2,n}}{c(n)}\right)^{-\alpha} \\ &\quad+\frac{R_{1,n}/c(n)^\alpha}{\left(\sum_{k=1}^{c(n)}\tau_k/c(n)+R_{2,n}/c(n)\right)^\alpha}.\end{aligned} \tag{7}$$

By assumption, the random variables $\{X_k(s)\}$ are i.i.d. with common mean 0. Moreover, $|X_k(s)| \le (1+w_s)\tau_k$, which implies that $\mathrm{E}\left[\,|X_1(s)|^{1/\alpha}\,\right] \le (1+w_s)^{1/\alpha}\,\mathrm{E}\left[\,\tau_1^{1/\alpha}\,\right] < \infty$. It then follows from the Marcinkiewicz-Zygmund strong law of large numbers [7, 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 $\sum_{k=1}^{c(n)}\tau_k/c(n) \xrightarrow{\text{a.s.}} \mathrm{E}\left[\,\tau_1\,\right] > 0$. Next, note that $|R_{1,n}| \le (1+w_s)\tau_{c(n)+1}$, so that $R_{1,n}/c(n)^\alpha \xrightarrow{\text{a.s.}} 0$ provided that $\tau_k/k^\alpha \xrightarrow{\text{a.s.}} 0$. This latter limit result follows from a Borel-Cantelli argument; see [10] for details. 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 (7). $\square$

The conditions on $\{\epsilon_n\}$ in Theorem 1 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). Similarly, a wide variety of strata-selection schemes satisfy the conditions of Theorem 1. 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. Certain schemes in which the number of steps per stratum is random are also covered by Theorem 1; see [10]. In Sec. 6, we discuss variants on these schemes that are particularly suitable for practical implementation in the context of DSGD.

5. THE DSGD ALGORITHM

Classic, sequential SGD as in Sec. 3 cannot be used directly for rank-r factorization of web-scale matrices. We can, however, exploit the structure of the matrix factorization problem to derive a scalable distributed SGD algorithm. 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 (3), 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 throughout 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 *with respect to a loss function L having summation form if for all $\theta \in H$, and $\epsilon > 0$,*

$$\begin{aligned} &L'_{z_1}(\theta) = L'_{z_1}(\theta - \epsilon L'_{z_2}(\theta)) \\ \textit{and}\quad &L'_{z_2}(\theta) = L'_{z_2}(\theta - \epsilon L'_{z_1}(\theta)). \end{aligned} \tag{8}$$

Two disjoint sets of training points $Z_1, Z_2 \subset Z$ are interchangeable with respect to L 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, and follows directly from (4) and (5); see [10] for more details.

THEOREM 2. *Two training points $z_1 = (i_1, j_1) \in Z$ and $z_2 = (i_2, j_2) \in Z$ are interchangeable with respect to any loss function L having summation form if they share neither row nor column, i.e., $i_1 \ne i_2$ and $j_1 \ne j_2$.*

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

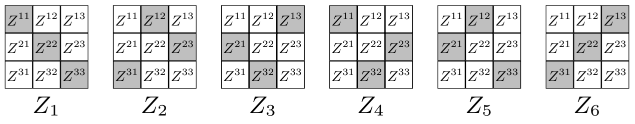
We introduce the DSGD algorithm by considering a simple case that essentially corresponds to running DSGD using a single "d-monomial" stratum (see Sec. 5.3). The goal is to highlight the technique by which DSGD runs the SGD algorithm in a distributed manner within a stratum. For a given training set 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:

$$\begin{array}{c} \\ \boldsymbol{W}^1 \\ \boldsymbol{W}^2 \\ \vdots \\ \boldsymbol{W}^d \end{array} \begin{array}{c} \begin{array}{cccc} \boldsymbol{H}^1 & \boldsymbol{H}^2 & \cdots & \boldsymbol{H}^d \end{array} \\ \begin{pmatrix} \boldsymbol{Z}^1 & \boldsymbol{0} & \cdots & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Z}^2 & \cdots & \vdots \\ \vdots & \vdots & \ddots & \boldsymbol{0} \\ \boldsymbol{0} & \cdots & \boldsymbol{0} & \boldsymbol{Z}^d \end{pmatrix} \end{array}, \tag{9}$$

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 Th. 2, sets Z^i and Z^j are interchangeable for $i \ne j$. 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. 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

$$\theta_T(\omega) = \theta_0 + \epsilon\sum_{n=0}^{T-1} Y_n(\omega). \tag{10}$$

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.



Z^{11} Z^{12} Z^{13}
Z^{21} Z^{22} Z^{23}
Z^{31} Z^{32} Z^{33}
Z_1
Z_2
Z_3
Z_4
Z_5
Z_6

Figure 1: Strata for a 3×3 blocking of training matrix $\boldsymbol{Z}$

THEOREM 3. *Using the definitions above,*

$$\theta_T(\omega) = \theta_0 + \epsilon \sum_{b=1}^{d} \sum_{k=0}^{T_b(\omega)-1} Y_k(\sigma_b(\omega)). \tag{11}$$

See [10] for a complete proof. The idea is to repeatedly invoke Th. 2 and Def. 1 to show that, if the mth training point in the combined sequence corresponds to the kth training point from block $\boldsymbol{Z}^b$, then the update terms coincide: $Y_m(\omega) = Y_k(\sigma_b(\omega))$. Applying this result to all of the blocks establishes a one-to-one correspondence between the update terms in (10) and in (11).

We now describe how to exploit Theorem 3 for distributed processing on MapReduce. We block $\boldsymbol{W}$ and $\boldsymbol{H}$ conformingly to $\boldsymbol{Z}$—as in (9)—and divide processing into d independent map 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[3] factor matrices $\boldsymbol{W}^b_{\text{new}}$ and $\boldsymbol{H}^b_{\text{new}}$. By Theorem 3, we have $\boldsymbol{W}' = (\boldsymbol{W}^1_{\text{new}} \cdots \boldsymbol{W}^d_{\text{new}})^{\mathrm{T}}$ and $\boldsymbol{H}' = (\boldsymbol{H}^1_{\text{new}} \cdots \boldsymbol{H}^d_{\text{new}})$, 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 the foregoing development, we used the fact that $\boldsymbol{Z}$ is block-diagonal only to establish interchangeability between blocks. This means that Theorem 3 and the resulting distributed SGD scheme 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 exploit this fact to obtain the overall DSGD algorithm.

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 columns 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 $\boldsymbol{Z}^{ij}$. Thus a stratum corresponds to a set of blocks; Fig. 1 shows the set of possible strata when $d = 3$. In general, 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 $\boldsymbol{Z}$.

Given a set of strata and associated weights $\{w_s\}$, we decompose the loss into a weighted sum of per-stratum losses as in (6): $L(\boldsymbol{W}, \boldsymbol{H}) = \sum_{s=1}^{q} w_s L_s(\boldsymbol{W}, \boldsymbol{H})$. (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

$$L_s(\boldsymbol{W}, \boldsymbol{H}) = c_s \sum_{(i,j) \in Z_s} L_{ij}(\boldsymbol{W}, \boldsymbol{H}), \tag{12}$$

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

$$\hat{L}'_s(\boldsymbol{W}, \boldsymbol{H}) = N_s c_s L'_{ij}(\boldsymbol{W}, \boldsymbol{H}) \tag{13}$$

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; e.g., strata Z_1, Z_2, and Z_3 in Fig. 1. Then any given loss function L of the form (1) 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 DSGD on d nodes in a shared-nothing environment such as MapReduce, we only distribute the input matrix once. Then the only data that are transmitted between nodes during subsequent processing are (small) blocks of factor matrices. In our implementation, node i stores blocks $\boldsymbol{W}^i, \boldsymbol{Z}^{i1}, \boldsymbol{Z}^{i2}, \ldots, \boldsymbol{Z}^{id}$ for $1 \leq i \leq d$; thus 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.)

[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.

Algorithm 2 DSGD for Matrix Factorization

```
Require: $\boldsymbol{Z}, \boldsymbol{W}_0, \boldsymbol{H}_0$, cluster size $d$
  $\boldsymbol{W} \leftarrow \boldsymbol{W}_0$ and $\boldsymbol{H} \leftarrow \boldsymbol{H}_0$
  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 $\epsilon$
    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
```

Since, by construction, parallel processing within the *k*th 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.

6. DSGD IMPLEMENTATION

In this section, we discuss practical methods for choosing the training sequence for the parallel SGD step, selecting strata, and picking the step size ϵ. 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.

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. To process a block $\boldsymbol{Z}^{ij}$, we randomly select training points from $\boldsymbol{Z}^{ij}$ such that each point is selected precisely once. This approach ensures that many different training points are selected while at the same time maximizing randomness; see [10] for further discussion. Note that Theorem 1 implicitly assumes sampling with replacement, but can be extended to cover the foregoing strategy 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; thus $c_s = N/N_s$ in (13). 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 (12).

Stratum selection. Recall that the stratum sequence (ξ_k, T_k) determines which stratum is chosen in each subepoch and how many steps are run on the stratum. We choose training sequences such that $T_k = N_{\xi_k} = |Z_{\xi_k}|$; i.e., we make use of all the training points in each selected stratum. Moreover, we pick a sequence of d strata to visit during an epoch such that the strata jointly cover the entire training set; the sequence is picked uniformly and at random from all such sequences of d strata. This strategy is analogous to that for intra-block training-point selection. Taking the scaling constant c_s in (12) as N/N_s, it can be seen that this strategy is covered by Theorem 1, where each epoch corresponds to a regenerative cycle. We argue informally as follows. Recall that if Theorem 1 is to apply, then w_s must correspond to the long-term fraction of steps run on stratum Z_s. This means that all but d of the weights are zero, and the remaining weights satisfy $w_s = N_s/N$. The question is then whether this choice of w_s leads to a legitimate representation of L as in (6). One can show that $\{w_s\}$ satisfies (6) for all $\boldsymbol{Z}$ and L of form (1) if and only if $\sum_{s:Z_s \supseteq Z^{ij}} w_s c_s = 1$ for each substratum Z^{ij}. This equality holds for the above choices of w_s and c_s.

Step sizes. The stochastic approximation literature often works with step size sequences roughly of form $\epsilon_n = 1/n^\alpha$ with $\alpha \in (0.5, 1]$; Theorem 1 guarantees asymptotic convergence for such 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, and thus can check whether the loss has decreased or increased from the previous epoch. We then 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 the loss decreases over an epoch, and (2) drastically decrease the step size (say, by 50%) if the loss increases. 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 many compute nodes are available, replicating a small sample of $\boldsymbol{Z}$ (say, 0.1%) to each node and trying different step sizes in parallel. Specifically, we try step sizes $1, 1/2, 1/4, \ldots, 1/2^{d-1}$; the step size that gives the best result is selected as ϵ_0. As long as the loss decreases, we repeat a variation of this process after every epoch, trying step sizes within a factor of $[1/2, 2]$ of the current step size. Eventually, the step size will become too large and the loss will increase. Intuitively, this happens when the iterate has moved closer to the global solution than to the local solution. At this point, we switch to the bold-driver method for the rest of the process.

7. EXPERIMENTS

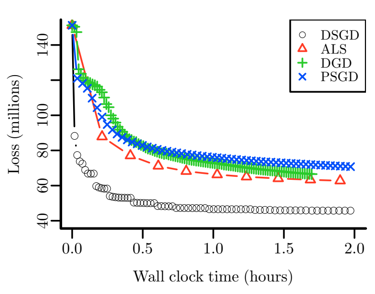
We compared various factorization algorithms with respect to convergence, runtime efficiency, and scalability. Overall, the convergence speed and result quality of DSGD was on par or better than alternative methods, even when these methods are specialized to the loss function. Due to space limitations we report on only a few representative experiments; see [10] for detailed results.

7.1 Setup

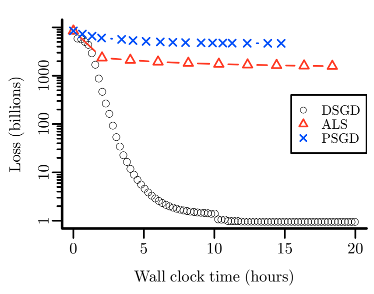
We implemented DSGD on top of MapReduce, along with the best-of-breed PSGD, DGD, and ALS methods; see Sec. 2. The DGD algorithm uses the L-BFGS quasi-Newton method as in [9]. DSGD, PSGD, and L-BFGS are generic methods that work with a wide variety of loss functions, whereas ALS is restricted to quadratic loss. We used two different implementations and compute clusters: one for in-memory experiments and one for large scaling experiments on very large datasets using Hadoop.

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 (as described at the end of Sec. 5.3). The factor matrices are communicated via a centralized file system. The R cluster consists of 8 nodes, each running two Intel Xeon E5530 processors with 4 cores at 2.4GHz each. Every node has 48GB of memory.

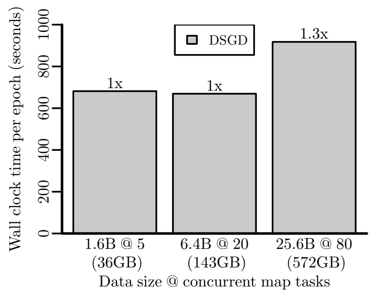
The second implementation is based on Hadoop [1], an open-source MapReduce implementation. The Hadoop cluster is equipped with 40 nodes, each with two Intel Xeon E5440 processors and 4



(a) Netflix data (NZSL, R cluster @ 64)



(b) Synth. data (L2, $\lambda = 0.1$, R cluster @ 64)



(c) Scalability (Hadoop cluster)

Figure 2: Experimental results

cores at 2.8GHz and 32GB of memory. We employ a couple of Hadoop-specific optimizations; see [10].

For our experiments with PSGD and DSGD, we used adaptive step-size computation based on a sample of roughly 1M data points, eventually switching to the bold driver. The time for step-size selection is included in all performance plots.

We used the Netflix competition dataset [3] for our experiments on real data. This 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 performance experiments, we used a much larger 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 $N(0, 10)$ distribution. We then sampled 1B entries from the product $\boldsymbol{W}^*\boldsymbol{H}^*$ and added $N(0, 1)$ noise to each sample, ensuring that there existed a reasonable low-rank factorization. We always 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. Finally, for our scalability experiments, we used the Netflix competition dataset and scaled up the data in a way that keeps the sparsity of the matrix constant. Specifically, at each successive scale-up step, we duplicated the number of customers and movies while quadrupling the number of ratings (nonzero entries). We repeat this procedure to obtain matrices between 36GB and 572GB in size. Unless stated otherwise, we use rank $r = 50$.

We focus here on two common loss functions: plain nonzero squared loss $L_{\text{NZSL}} = \sum_{(i,j)\in Z}(\boldsymbol{V}_{ij} - [\boldsymbol{W}\boldsymbol{H}]_{ij})^2$ and nonzero squared loss with L2 regularization $L_{\text{L2}} = L_{\text{NZSL}} + \lambda\big(\|\boldsymbol{W}\|_{\text{F}}^2 + \|\boldsymbol{H}\|_{\text{F}}^2\big)$. For our experiment on synthetic data and L_{L2}, we use a "principled" value of $\lambda = 0.1$; this choice of λ 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. Results for other loss functions (including GKL loss) are given in [10].

All of our reported experiments focus on *training* loss, i.e., the loss over the training data, since our emphasis is on how to compute a high quality factorization of a given input matrix as efficiently as possible. The *test* loss, i.e., how well the resulting factorized matrix predicts user ratings of unseen movies, is an orthogonal issue that depends upon, e.g., the choice of loss function and regularization term. In this regard, we re-emphasize that the DSGD algorithm can handle a wide variety of loss functions and regularization schemes. (In fact, we found experimentally that the loss performance of DSGD relative to other factorization algorithms looked similar for test loss and training loss.)

7.2 Relative Performance

We 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 training loss after every epoch. We used 64-way distributed processing on 8 nodes (with 8 concurrent map tasks per node).

Representative results are given in Figs. 2a and 2b. In all our experiments, DSGD converges about as fast as—and, in most cases, faster than–alternative methods. After DSGD, the fastest-converging algorithms are ALS, then DGD, then PSGD. Note that each algorithm has a different cost per epoch: DSGD ran 43 epochs, ALS ran 10 epochs, DGD ran 61 epochs, and PSGD ran 30 epochs in the first hour of the Netflix experiment. These 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). We omit results for DGD in Fig. 2b because its centralized parameter-update step ran out of memory.

Besides the rate of convergence, the ultimate training-loss value achieved is also of interest. DGD will converge to a good local minimum, similarly to DSGD, but the convergence is slow; e.g., in Fig. 2a, DGD was still a long way from convergence after several hours. With respect to PSGD, we note that 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 good partition-local factors as computed by PSGD do not correspond to good global factors, which explains why the algorithm converged to suboptimal solutions in the experiments. Finally, ALS, which is a specialized method, is outperformed by DSGD in the experiments shown here, but came close in performance to DSGD in some of our other experiments [10]. Unlike the other algorithms, we are unaware of any theoretical guarantees of convergence for ALS when it is applied to nonconvex optimization problems such as matrix factorization. This lack of theoretical support is perhaps related to ALS's erratic behavior. In summary, the overall performance of DSGD was consistently more stable than that of the other two algorithms, and the speed of convergence was comparable or faster.

We also assessed the impact of communication overheads on DSGD by comparing its performance with standard, sequential SGD. Note that we could not perform such a comparison on massive data, because SGD simply does not scale to very large datasets, e.g., our 572GB synthetic dataset. Indeed, even if SGD were run without any data shuffling, so that data could be read sequentially, merely reading the data once from disk would take hours. We therefore compared SGD to 64-way DSGD on the smaller Netflix dataset.

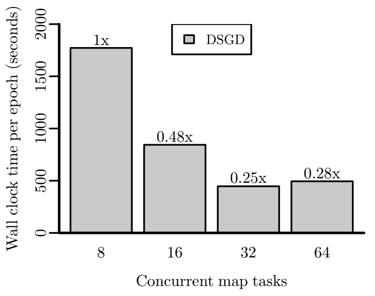


Figure 3: Speed-up experiment (Hadoop cluster, 143GB data)

Here, SGD required slightly fewer epochs than DSGD to converge. This discrepancy is a consequence of different randomizations of the training sequence: SGD shuffles the entire dataset, whereas DSGD shuffles only strata and blocks. The situation was reversed with respect to wall-clock time, and DSGD converged slightly faster than SGD. Most of DSGD's processing time was spent on communication of intermediate results over the (slow) centralized file system. Recent distributed processing platforms have the potential to reduce this latency and improve performance for moderately-sized data; we are currently experimenting with such platforms.

7.3 Scalability of DSGD

We next studied the scalability of DSGD in our Hadoop environment, which allowed us to process much larger matrices than on the in-memory R cluster. In general, we found that DSGD has good scalability properties on Hadoop, provided that the amount of data processed per map task does not become so small that system overheads start to dominate.

Figure 2c shows the wall-clock time per DSGD epoch for different dataset sizes (measured in number of nonzero entries of $\boldsymbol{V}$) and appropriately scaled numbers of concurrent map tasks (after @-sign). The processing time initially remains constant as the dataset size and number of concurrent tasks are each scaled up by a factor of 4. As we scale to very large datasets (572GB) on large clusters (80 parallel tasks), the overall runtime increases by a modest 30%. This latter overhead can potentially be ameliorated by improving Hadoop's scheduling mechanism, which was a major bottleneck.

A similar observation is made in Figure 3, where we depict the speedup performance when the number of ratings (nonzero entries) is fixed at 6.4B (143GB) and the number of concurrent map tasks is repeatedly doubled. DSGD initially achieves roughly linear speed-up up to 32 concurrent tasks. After this point, speed-up performance starts to degrade. The reason for this behavior is that, when the number of map tasks becomes large, the amount of data processed per task becomes small—e.g., 64-way DSGD uses 64^2 blocks so that the amount of data per block is only $\approx$ 35MB. The actual time to execute DSGD on the data is swamped by Hadoop overheads, especially the time required to spawn tasks.

8. CONCLUSION

We have developed a stratified version of the classic SGD algorithm and then refined this SSGD algorithm to obtain DSGD, a distributed matrix-factorization algorithm that can efficiently handle web-scale matrices. Experiments indicate its superior performance. In future work, we plan to investigate alternative loss functions, such as GKL, as well as alternative regularizations. We also plan to investigate both alternative stratification schemes and emerging distributed-processing platforms. We will also extend our techniques to other applications, such as computing Kohonen maps.

9. REFERENCES

[1] Apache Hadoop. `https://hadoop.apache.org`.

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

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

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

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

[6] 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.

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

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

[9] 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.

[10] R. Gemulla, P. J. Haas, E. Nijkamp, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. Technical Report RJ10481, IBM Almaden Research Center, San Jose, CA, 2011. Available at www.almaden.ibm.com/cs/people/peterh/dsgdTechRep.pdf.

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

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

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

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

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

[16] 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.

[17] 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.

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

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

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

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