FLEXIFACT: Scalable Flexible Factorization of Coupled Tensors on Hadoop

Alex Beutel
Department of Computer Science
Carnegie Mellon University
abeutel@cs.cmu.edu

Abhimanu Kumar
Language Technology Institute
Carnegie Mellon University
abhimank@cs.cmu.edu

Evangelos Papalexakis
Department of Computer Science
Carnegie Mellon University
epapalex@cs.cmu.edu

Partha Pratim Talukdar
Machine Learning Department
Carnegie Mellon University
ppt@cs.cmu.edu

Christos Faloutsos
Department of Computer Science
Carnegie Mellon University
christos@cs.cmu.edu

Eric P. Xing
Machine Learning Department
Carnegie Mellon University
epxing@cs.cmu.edu

Abstract

Given multiple data sets of relational data that share a number of dimensions, how can we efficiently decompose our data into the latent factors? Factorization of a single matrix or tensor has attracted much attention, as, e.g., in the Netflix challenge, with users rating movies. However, we often have additional, side, information, like, e.g., demographic data about the users, in the Netflix example above. Incorporating the additional information leads to the *coupled factorization* problem. So far, it has been solved for relatively small datasets.

We provide a distributed, scalable method for decomposing matrices, tensors, and coupled data sets through stochastic gradient descent on a variety of objective functions. We offer the following contributions: (1) *Versatility:* Our algorithm can perform matrix, tensor, and coupled factorization, with flexible objective functions including the Frobenius norm, Frobenius norm with an ℓ_1 induced sparsity, and non-negative factorization. (2) *Scalability:* FLEXIFACT scales to unprecedented sizes in both the data and model, with up to *billions* of parameters. FLEXIFACT runs on standard Hadoop. (3) *Convergence proofs* showing that FLEXIFACT converges on the variety of objective functions, even with projections.

Note: This work is currently under review at other, non-machine learning conferences.

1 Introduction

How can we efficiently mine data that capture relations between different entities? Suppose, for instance, that we are given a time-evolving social network, such as Facebook, and we have information about who messages whom, or who becomes friends with whom, and when. This data may be formulated as a three mode tensor. Suppose now that we also have some side information pertaining to the users, e.g. demographic information. This problem can be formulated as an instance of a so-called *coupled factorization*, where the two pieces of data, a three-mode (user, user, time) tensor and a (user, demographic) matrix share a common dimension. Even without the presence of the

	FlexiFaCT	DSGD [6]	PSGD [9]	Matlab	GigaTensor [7]
Data/Model					
Matrix	✓	✓	$\sim$	✓	
Tensor	✓		$\sim$	✓[4]	✓
Coupled Tensor/Matrix	✓		$\sim$	✓[2]	
Obj. Function					
Frobenius norm	✓	✓	✓	✓	✓
Frobenius norm + ℓ_1 penalty	✓		✓	✓	
Non-negativity constraints	✓		✓	✓	
Handles missing data	✓	✓	✓	✓	
Scalability					
in number of non-zeros	✓	$\sim$	✓		✓
in data dimensions	✓	$\sim$			✓
in decomposition rank	✓	$\sim$	✓		$\sim$
Proof of convergence					
Matrix Factorization	✓	✓			
Tensor/Coupled Factorization	✓	$\sim$			✓
Projections (ℓ_1 & non-negativity)	✓		$\sim$		

Table 1: Feature Comparison of proposed FlexiFaCT vs state of the art. ($\sim$ represents unknown or not directly applicable.) FlexiFaCT contains existing state of the art as special cases.

(user, demographic) matrix, efficient tensor decomposition of truly large datasets can be challenging, attracting increasing interest.

Most prior work has either focused on a specific type of factorization or a specific loss function (e.g. Frobenius norm), thus having a limited range of potential applications. Here we propose FlexiFaCT, a flexible and highly scalable distributed factorization algorithm which attacks a very broad spectrum of problems: FlexiFaCT can handle matrices, tensors, coupled tensor-matrix settings, *cross product* a variety of loss functions, including Frobenius norm, KL divergence, ℓ_1 regularization, and non-negativity constraints.

Moreover, FlexiFaCT is very fast and scalable; we show how to implement it on Hadoop, and we show how to achieve high speeds, by distributing both the data as well as the parameters. In Table 1 , we provide a comprehensive overview of the state of the art. In short, FlexiFaCT reigns, combining both scalability, as well as versatility.

In summary, our main contributions are:

1. **Versatility**: FlexiFaCT can operate under a wide spectrum of settings, including plain matrix factorization, tensor factorization, as well as coupled decompositions. Thus, FlexiFaCT includes several recent methods [6], [7], as *special cases*.
2. **Scalability**: FlexiFaCT scales very well both with the input size, as well as with the number of model parameters.
3. **Proof of convergence**: We *prove* that FlexiFaCT converges, even with constraints like non-negativity. Moreover, we demonstrate this empirically.
4. **Usability and Reproducibility**: Our implementation runs on *stock* Hadoop, as opposed to other recent methods [6]. We also open-source our code.

2 FlexiFaCT Approach

As mentioned previously, we take on the problem of matrix, tensor and coupled factorization. For brevity, we do not include here the mathematical details of the loss functions and SGD updates. The complete documentation can be found in Appendix A. Building off of recent work in stochastic learning theory [**?**] and matrix factorization [6], we develop a block scheme for the model and data to parallelize the computation across a cluster. The details of our blocking scheme can be found in Appendex B. As a quick summary, we see an example of blocking scheme in Figure 1 and the general algorithm we follow in Algorithm 1.

Additionally, in order to support non-negative factorizations as well as sparsity constraints, we prove using stochastic learning theory that our process is regenerative, even under projections, and thus our algorithm converges. The proof can be found in Appendix C.

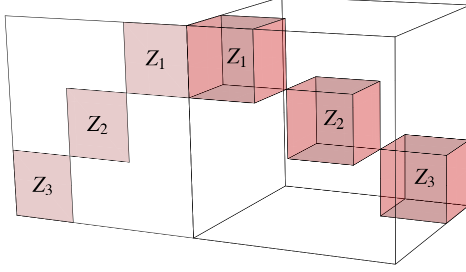


Figure 1: Dividing a paired matrix and tensor into blocks such that no two of them share any row, column, or third dimension.

Algorithm 1: FLEXIFACT for tensors

Input : $\mathcal{X}, \mathbf{U}_0, \mathbf{V}_0, \mathbf{W}_0$,sub-epoch size d
$\mathbf{U} \leftarrow \mathbf{U}_0, \mathbf{V} \leftarrow \mathbf{V}_0, \mathbf{W} \leftarrow \mathbf{W}_0$
Block $\mathcal{X}, \mathbf{U}, \mathbf{V}, \mathbf{W}$ into corresponding d blocks
while *not converged* **do**
  Pick step size η
  for $s = 0, ..., d^2 - 1$ **do**
    Pick d blocks($Z_1^{(s)}, ..., Z_d^{(s)}$) to form stratum $Z^{(s)}$
    for $b = 0, \ldots, d-1$ ***in parallel*** **do**
      Run SGD on the training points $Z_b^{(s)}$
    end
  end
end

3 Experiments

3.1 Performance Evaluation

In order to assess how scalable and fast FLEXIFACT is, we conducted a series of experiments in order to measure the running time of FLEXIFACT with respect to 1) increasing number of data points, 2) increasing dimensions of the data and thus model, and 3) increasing rank of the factorization. The first aspect has to do with scalability in terms of data size, whereas the two latter aspects refer to scalability with respect to parameter space size; FLEXIFACT is able, as we demonstrate in the following experiments, to scale easily in all three aspects. As a baseline for tensor decomposition, we use GigaTensor [7]. We also compared against PSGD [9], however, the solutions obtained achieved much worse RMSE, and the algorithm was not able to scale for very large number of parameters (either rank or dimensions).

FLEXIFACT was implemented in Java, with Hadoop 0.20.1 [5, 1]. We ran the experiments on the OCC-Y cluster[1]. For the sake of experimentation, we created a series of synthetic datasets wherein we were able to control the three aspects we were testing: data size, data dimensions, and rank. In all cases, number of reducers was constant and equal to 24.

Synthetic Data Generation To generate data we first generate randomly matrix factors $\mathbf{U}, \mathbf{V}, \mathbf{W}$ of the specified dimension D (where $I = J = K = D$) and rank $R = 30$. We then randomly select data points (i, j, k) and add their value $\mathbf{U}_{i,*} \circ \mathbf{V}_{j,*} \circ \mathbf{W}_{k,*}$ to the dataset. We do this until we have the desired number of data points for each dataset. Unless otherwise stated, we set $D = 1$ million and the number of data points to 10 million.

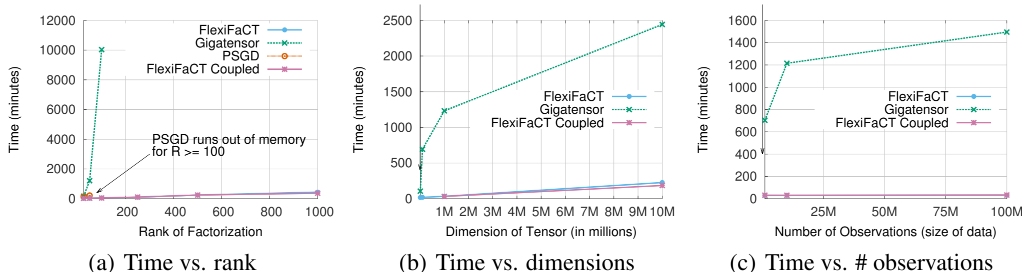
3.2 Scalability

We now test FLEXIFACT on all three types of scalability to demonstrate that it scales in all dimensions and to unprecedented sizes.

Rank Scalability In testing the scalability with respect to rank, we ran FLEXIFACT, GigaTensor, and PSGD on a tensor and varied the rank from 25 up to 1000. Figure 2(a) shows the running time for FLEXIFACT, both for tensor and coupled factorizations, as the rank (i.e. one of the parameter dimensions) increases. We can see that FLEXIFACT scales linearly as the rank of the factorization increases, having similar timing behaviour both for plain tensor and coupled factorizations. Note that with $R = 1000$ and $D = 1$ million, the coupled FLEXIFACT factorization scales to a total parameter space of *4 billion parameters*.

Data Dimensions Scalability To test more directly the scalability as the dimension D of the data tensor grows, we created a variety of tensors with varying dimension from 10,000 to 10 million. We

[1] http://opencloudconsortium.org/tag/occ-y/



(a) Time vs. rank (b) Time vs. dimensions (c) Time vs. # observations

Figure 2: Scalability of FLEXIFACT in terms of: a) rank, b) data dimensions, and c) number of observations. We observe that FLEXIFACT scales very well with respect to all aspects. PSGD can be seen in sub-figure (a) before it runs out of memory. FLEXIFACT was applied to both a tensor, and a matrix-tensor couple, whereas GigaTensor was only applied to a tensor.

decompose each tensor with $R = 50$. When testing the coupled FLEXIFACT decomposition, we add an additional coupled matrix with 100,000 data points and the same dimensionality as the main tensor.

In Figure 2(b) we show how coupled factorization using FLEXIFACT scales, as the dimensions of the data increase. We observe that FLEXIFACT runs much faster than the baseline, GigaTensor. A likely explanation for the degree to which FLEXIFACT is faster than GigaTensor is that FLEXIFACT only focuses on the observed data points, where GigaTensor has to convert unobserved data points to zeros, thus slowing down the computation. Again, for the coupled case, note that our total parameter space reaches 2 billion parameters.

Data Size Scalability Last, for data scalability, we vary the number of observed data points from 1 million to 10 million. Figure 2(c) shows FLEXIFACT's running time as a function of the data size, i.e. the number of observations. We can see that FLEXIFACT has, again, very smooth behaviour, and scales linearly with the number of observed elements. Again, we are significantly faster than GigaTensor, though the degree of difference is likely because it is must make unobserved points zeros for it to run.

3.3 Correctness & Monotone Convergence

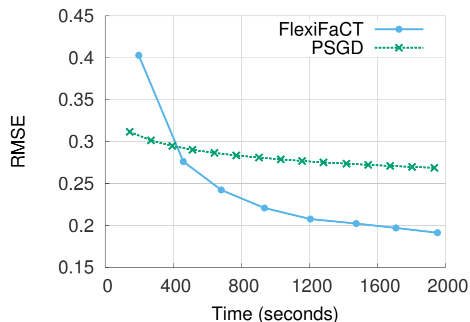


Figure 3: Convergence: RMSE vs. time, for tensor factorization comparing FLEXIFACT and PSGD [9]. Note, Zinkevich et al. [9] do not claim to work on this problem because it is not convex.

Besides speedup, we experimentally validate that FLEXIFACT indeed decreases monotonically the objective function that it is minimizing. To test this we run on a small synthetic data set with $D = 10,000$ and 10 million data points, making the dataset very dense. We run a factorization with $R = 50$ using our implementation of PSGD and FLEXIFACT with both ℓ_1 sparsity and non-negativity constraints. We then monitor the root mean squared error (RMSE).

Figure 3 shows that FLEXIFACT decreases the RMSE as expected, and at a much quicker pace than PSGD. It is important to note that the slow convergence of PSGD is because the problem (tensor factorization) is not convex, and thus Zinkevich et al. do not claim that their method works on such problems. However, we use PSGD as a comparison because it is not possible to track the RMSE with GigaTensor and thus PSGD is the closest competitor.

4 Conclusion

In this work we have introduced FLEXIFACT, a highly flexible, efficient, and scalable factorization tool. Our main contributions are: 1) Versatility: since FLEXIFACT can operate under numerous factorization scenarios and constraints, 2) Scalability, both with respect to the input size, and the number of parameters, 3) Proof of convergence: FLEXIFACT is provably correct, even in the presence of constraints, and 4) Reproducibility and usability: we use *stock* Hadoop, and make our implementation publicly available.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`, 2012.

[2] The matlab cmtf toolbox. `http://www.models.life.ku.dk/joda/CMTF_Toolbox`, 2013.

[3] S. Asmussen. *Applied Probability and Queues*. Wiley, 1987.

[4] B.W. Bader and T.G. Kolda. Matlab tensor toolbox version 2.2. *Albuquerque, NM, USA: Sandia National Laboratories*, 2007.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, December 2004.

[6] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *ACM SIGKDD*, pages 69–77, New York, NY, USA, 2011. ACM.

[7] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *ACM SIGKDD*, pages 316–324. ACM, 2012.

[8] H. Kushner and G. Yin. *Stochastic Approximation and Recursive Algorithms and Applications*. Springer, 2003.

[9] Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. Parallelized stochastic gradient descent. In *NIPS*, pages 2595–2603, 2010.

A Objectives and Updates

In this section we will explain the variety of loss functions used in these tasks, the Stochastic Gradient Descent (SGD) update rules, and our partitioning scheme allowing for distribution of the SGD work. Although much of our description of the *matrix* factorizaiton work is similar to [6], we will explain it here for completeness and clarity.

Before we begin, it is important to clarify our notiation. We will use capital boldface script characters like $\mathbf{\mathcal{X}}$ to denote a tensor, capital boldface non-script characters e.g. $\mathbf{Y}$ to denote a matrix, and lowercase boldface character, e.g. $\mathbf{y}$, to denote a vector. $\mathbf{\mathcal{X}}_{i,j,k}$ denotes the scalar in the (i, j, k) position of the tensor $\mathbf{\mathcal{X}}$, $\mathbf{Y}_{i,j}$ denotes the scalar in the (i, j) position of matrix $\mathbf{Y}$, and $\mathbf{y}_i$ denotes the scalar in the ith position of vector $\mathbf{y}$. We use $\mathbf{Y}_{i,*}$ to denote the vector of scalars $\mathbf{Y}_{i,j}$ for all j. Additionally, with a slight overloading of notation for simplicity and because our matrices and tensors may only have a small percentage of observed values, we say that $(i, j, k) \in \mathbf{\mathcal{X}}$ if $\mathbf{\mathcal{X}}_{i,j,k}$ is observed.

A.1 Optimization Objectives

We begin by explaining how stochastic gradient descent works for our variety of objective functions. We will briefly go over the objective functions for simpler cases like the Frobenius norm of matrices before expanding to more complex objectives.

Matrix Factorization For matrix factorization we would like to approximate our $I \times J$ data matrix $\mathbf{X}$ by $\mathbf{U}\mathbf{V}^T$, where $\mathbf{U}$ is of size $I \times R$ and $\mathbf{V}$ is of size $J \times R$. Therefore, we can have a loss function

Objective ($\mathcal{L}$)	Formulation
Frobenius	$\sum_{(i,j,k)\in\mathcal{X}} \mathcal{L}_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W}) + \sum_{(i,j)\in\mathbf{Y}} \mathcal{L}_{\mathbf{Y}_{i,j}}(\mathbf{U},\mathbf{A})$
Frobenius + $\ell 1$	$\sum_{(i,j,k)\in\mathcal{X}} \mathcal{L}_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W}) + \sum_{(i,j)\in\mathbf{Y}} \mathcal{L}_{\mathbf{Y}_{i,j}}(\mathbf{U},\mathbf{A})$ $+ \lambda(\|U\|_1 + \|V\|_1 + \|W\|_1 + \|A\|_1)$
Frobenius + $\ell 1$ + NN	$\sum_{(i,j,k)\in\mathcal{X}} \mathcal{L}_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W}) + \sum_{(i,j)\in\mathbf{Y}} \mathcal{L}_{\mathbf{Y}_{i,j}}(\mathbf{U},\mathbf{A})$ $+ \lambda(\|U\|_1 + \|V\|_1 + \|W\|_1 + \|A\|_1)\ s.t.\ \Theta_{i,r} \geq 0$

Table 2: Table of Objective Functions. $\mathbf{\Theta}$ denotes any of the factors U, V, W or A.

using the Frobenius norm as follows:

$$L(\mathbf{U},\mathbf{V}) = \|\mathbf{X} - \mathbf{U}\mathbf{V}^T\|_F^2 = \sum_{i,j\in\mathbf{X}} L_{\mathbf{X}_{i,j}}(\mathbf{U},\mathbf{V}) \tag{1}$$

where $L_{\mathbf{X}_{i,j}}(\mathbf{U},\mathbf{V}) = (\mathbf{X}_{i,j} - \sum_{r=1}^{R} \mathbf{U}_{i,r}\mathbf{V}_{j,r})^2$. As seen above, we divide our loss function into its component pieces $L_{\mathbf{X}_{i,j}}$ based on each observed point $\mathbf{X}_{i,j}$. This is necessary to use stochastic gradient descent.

Tensor Factorization For tensor factorization we would like to approximate our $I \times J \times K$ tensor $\mathcal{X}$ by a Khatri-Rao product $\sum_{r=1}^{R} \mathbf{U}_{*,\mathbf{r}} \circ \mathbf{V}_{*,\mathbf{r}} \circ \mathbf{W}_{*,\mathbf{r}}$ where $\mathbf{U}$ is of size $I \times R$, $\mathbf{V}$ is of size $J \times R$ and $\mathbf{W}$ is of size $K \times R$ and we are performing an Khatri Rao product between these three matrices. We can analyze the loss in a few different ways. Following the standard Frobenius norm, as is common in PARAFAC, the loss is:

$$L(\mathbf{U},\mathbf{V},\mathbf{W}) = \|\mathcal{X} - \sum_{r=1}^{R} \mathbf{U}_{*,\mathbf{r}} \circ \mathbf{V}_{*,\mathbf{r}} \circ \mathbf{W}_{*,\mathbf{r}}\|_F^2 = \sum_{(i,j,k)\in\mathcal{X}} L_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W})$$

where

$$L_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W}) = (\mathcal{X}_{i,j,k} - \sum_{r=1}^{R} \mathbf{U}_{i,r}\mathbf{V}_{j,r}\mathbf{W}_{k,r})^2.$$

Similarly we can induce sparsity in our parameter space with an ℓ_1 penalty:

$$\mathcal{L}(\mathbf{U},\mathbf{V},\mathbf{W}) = \sum_{(i,j,k)\in\mathcal{X}} L_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W}) + \lambda(\|\mathbf{U}\|_1 + \|\mathbf{V}\|_1 + \|\mathbf{W}\|_1) \tag{2}$$

or add a constraint that $\mathbf{U},\mathbf{V},\mathbf{W} \geq 0$ as is common in non-negative matrix factorization (NNMF). These terms are not as clearly separable in the loss function, but as we will see the update rules are still separable as is necessary for SGD. We make a distinction here between $\mathcal{L}$ and L: the objective $\mathcal{L}$ is obtained by adding $\ell 1$ or non-negativity constraints to the loss L.

Coupled Matrix-Tensor Factorization In this case our data tensor $\mathcal{X}$ is approximated by $\sum_{r=1}^{R} \mathbf{U}_{*,\mathbf{r}} \circ \mathbf{V}_{*,\mathbf{r}} \circ \mathbf{W}_{*,\mathbf{r}}$ and our data matrix $\mathbf{Y}$ is simultaneously approximated by $\mathbf{U}\mathbf{A}^T$. Note here we use the same component $\mathbf{U}$ in both approximations. As such, our objective function is merely a sum of the losses on each data set:

$$\mathcal{L}(\mathbf{U},\mathbf{V},\mathbf{W},\mathbf{A}) = \sum_{(i,j,k)\in\mathcal{X}} \mathcal{L}_{\mathcal{X}_{i,j,k}}(\mathbf{U},\mathbf{V},\mathbf{W}) + \sum_{(i,j)\in\mathbf{Y}} \mathcal{L}_{\mathbf{Y}_{i,j}}(\mathbf{U},\mathbf{A})$$

Table 2 denotes the loss objectives for different coupled cases. For each of these we use SGD to minimize our loss and thus approximate our data.

A.2 SGD Updates

For SGD we perform updates to our parameters $\mathbf{U},\mathbf{V},\mathbf{W},\mathbf{A}$, which we will collectively refer to as $\mathbf{\Theta}$ matrix whereas θ are the individual components of the matrix. This definition of $\mathbf{\Theta}$ and θ will come in handy for parameter updates based on the gradient at individual data points. E.g. the update for tensor $\mathcal{X}$ are:

$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \mathcal{L}_{\mathcal{X}_{i,j,k}}(\theta^{(t)}) \tag{3}$$

For these update rules, we list below the differentials for each component σ of θ where $(\nabla L_{X_{i,j,k}}(\theta))_\sigma = \frac{\partial L_{X_{i,j,k}}}{\partial \sigma}$:

$$(\nabla L_{\mathcal{X}_{i,j,k}}(\theta))_\sigma = \begin{cases} -2(\mathcal{X}_{i,j,k} - \sum_r \mathbf{U}_{i,r}\mathbf{V}_{j,r}\mathbf{W}_{k,r})\mathbf{V}_{j,\ell}\mathbf{W}_{k,\ell} & \text{if} \sigma = U_{i,\ell} \\ 0 \quad \text{if} \sigma = U_{i',\ell},\ i \neq i' \end{cases}$$

similarly for $\sigma = V_{j,l}$ or $\sigma = W_{k,l}$. From this we observe that SGD update for $U_{i,l}$ at a particular entry $X_{i,j,k}$ (for a tensor X) depends only on previous $U_{i,r}, A_{j,r}, B_{k,r}$ where $r \in 1, \ldots, R$ and R is the rank we chose. The updates for each component are similar for the paired cases.

In the case of additional components such as an ℓ_1 penalty or a non-negativity constraint on our parameters, we add a projection to our update rule. For example, for an ℓ_1 penalty, the update rule is

$$\theta^{(t+1)} = S_\lambda(\theta^{(t)} - \eta_t \nabla \mathcal{L}_{\mathcal{X}_{i,j,k}}(\theta^{(t)})) \tag{4}$$

$$S_\lambda(x) = \begin{cases} x - \lambda & \text{if } x > \lambda \\ x + \lambda & \text{if } x < -\lambda \\ 0 & \text{if } -\lambda \leq x \leq \lambda \end{cases} \tag{5}$$

Here we see that S_λ is the soft thresholding operator. We can similarly use the following projection for the non-negativity constraint:

$$NN(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases} \tag{6}$$

B Blocking for Parallelization

Given this understanding of our optimization objective and SGD update rules, we would like to segment our data in such a way that certain blocks Z_b can be run in parallel, where we define $Z_b \subseteq \mathcal{X}$. Figure 1 is a pictorial representation of the way we segment our simple matrix or a coupled tensor/matrix to enable parallelization. In order to run SGD on our blocks in parallel, we divide them such that no two blocks share common rows or columns. To be more precise, we say that a point $x \in Z_b$ is the coordinates in the data, such as $x = (x_i, x_j, x_k) \in \mathcal{X}$. Two blocks Z_b and $Z_{b'}$ are non-overlapping if for all $x \in Z_b$ and $x' \in Z_{b'}$, $x_i \neq x'_i$ and $x_j \neq x'_j$ and $x_k \neq x'_k$. (We will prove later that this allows us to run the blocks in parallel.) We see that in the division shown in Figure 1 no two blocks share common rows or columns. More interestingly, we note that in Figure 1(c) blocks in the tensor $\mathcal{X}$ and the matrix $\mathbf{Y}$ share coordinates in the i dimension, and as a result, data points in the same i range must be in the same block across both data sets.

Given this intuition, we provide a detailed description of our partition function. We call one set of independent blocks a stratum, and we denote the number of blocks in each stratum by d. In order to cover all regions of $\mathcal{X}$, we need multiple strata. For a matrix we require d strata, and for tensors we require d^2 strata. For a stratum s we have blocks $Z_i^{(s)}$ for $i = 0 \ldots d-1$. Each block $Z = (b_i, b_j, b_k)$ where b_i, b_j, b_k are ranges in I, J, and K: $b_i = (i\lceil I/d \rceil, (i+1)\lceil I/d \rceil), b_j = (j\lceil J/d \rceil, (j+1)\lceil J/d \rceil), b_k = (k\lceil K/d \rceil, (k+1)\lceil K/d \rceil)$. With this we define the blocks for stratum s as

$$Z_i^{(s)} = (b_i, b_{j_{s,i}}, b_{k_{s,i}}) \tag{7}$$

$$j_{s,i} = (j + s) \bmod d \tag{8}$$

$$k_{s,i} = \lfloor (j+s)/d \rfloor \bmod d \tag{9}$$

for $i = 0 \ldots d-1$.

In our algorithm, we run the strata sequentially, but for each stratum we run SGD on the blocks in parallel. We consider running SGD on one stratum to be a subepoch in our algorithm, and running it on all strata an epoch. (Note, the order in which you run the strata does not matter, as long as they are each run once per epoch.) We can do this repeatedly, iteratively updating our parameters θ, until the algorithm converges. A more formal write up of the distributed stochastic gradient algorithm for a tensor (which can easily be generalized to matrices and coupled factorizations) is shown in Algorithm 1. We next offer a proof that this converges appropriately.

C Proof of convergence with projections

The FLEXIFACT approach is described in Algorithm 1. We first prove that two blocks in a stratum are interchangeable. We use this to prove that sequence of strata are a regenerative process, defined later in this section. We use this to prove that our FLEXIFACT approach for Tensor and coupled case converges.

Our generic constrained loss function for a tensor case is

$$\mathcal{L} = L(U,V,W) + \lambda_u \|U\|_1 + \lambda_v \|V\|_1 + \lambda_w \|W\|_1 \\ s.t.\ U_{i,r}, V_{j,r}, W_{k,r} \geq 0. \tag{10}$$

In the above projected loss equation 10 the parameter is always in a set, $\mathcal{P}$, constrained by the $\ell 1$ and non-negativity constraints. The set $\mathcal{P}$ is a hyperrectangle defined as $\exists\, a_i < b_i, i = 1 \dots r$, such that $\mathcal{P} = \{\theta : a_i \leq \theta_i \leq b_i\}$ where $a_i, b_i \in (-\infty, \infty)$. Here θ is a parameter to be updated as defined in previous section (equation 3). The gradient based on equation 10 is:

$$\nabla_\theta \mathcal{L} = \nabla_\theta L + p(\theta), \quad p(\theta) \in C(\theta) \tag{11}$$

where θ is defined in Table **??** and $\mathcal{L}$ and L are defined in equation 2. Function $p()$ is the projection or constraint term of the gradient. The set $C(\theta)$ is the union of the subgradients at θ. When $\theta \in$ interior of $\mathcal{P}, C(\theta)$ contains only the zero elements and contains the convex cone generated by the subgradients at θ when $\theta \in \partial\mathcal{P}$, boundary of $\mathcal{P}$.

Definition 1 *Two blocks Z_i and $Z_{i'}$ in a given stratum are independent if for each $x \in Z_i$ and $x' \in Z_{i'}$ we have*

$$\nabla L_x(\theta) = \nabla L_x(\theta - \eta \nabla L_{x'}(\theta)) \\ \text{and } \nabla L_{x'}(\theta) = \nabla L_{x'}(\theta - \eta \nabla L_x(\theta)) \tag{12}$$

where $\nabla L_x(\theta)$ is the partial differential of L_x w.r.t. θ and θ is the parameter we are updating.

Theorem 1 *If blocks Z_i and $Z_{i'}$ in a given stratum S_n are non-overlapping then they are independent (as defined previously in Definition 1).*

Proof. For any two points $x \in Z_i$ and $x' \in Z_{i'}$ their rows or columns or any other coordinate do not overlap. From equation (4) we see that x does not modify θ in positions for which $i \neq x_i$, $j \neq x_j$ and $k \neq x_k$. Therefore, because x and x' are not equal in any dimension, an update from $\nabla L_{x'}$ will update different values than ∇L_x, where ∇L_x is the gradeint at point x

Additionally from equation (4) we see that any updates on ∇L_x only use values from $U_{x_i,*}$, $A_{x_j,*}$ and $B_{x_k,*}$, and thus do not use any values that would be updated by $\nabla L_{x'}$. Because updates from x only effect parameters in x's coordinates, updates from x are only based on parameters in x's coordinates, and x and x' have no overlapping coordinates, we know that

$$\nabla L_x(\theta) = \nabla L_x(\theta - \eta \nabla L_{x'}(\theta)) \\ \text{and } \nabla L_{x'}(\theta) = \nabla L_{x'}(\theta - \eta \nabla L_x(\theta))$$

Therefore, Z_i and $Z_{i'}$ are independent. By a similar argument

$$\nabla \mathcal{L}_x(\theta) = \mathcal{L}_x^{'}(\theta - \eta \nabla \mathcal{L}_{x'}(\theta)) \\ \text{and } \nabla \mathcal{L}_{x'}(\theta) = \nabla \mathcal{L}_{x'}(\theta - \eta \nabla \mathcal{L}_x(\theta))$$

□

Definition 2 *a process $P(t), t \geq 0$ is regenerative if there exist time points $0 \leq T_0 < T_1 < T_2 < \dots$ such that the the remainder of the process after T_k $P(T_k + t) : t \geq 0$, for $k \geq 1$: (a) has the same distribution as the remainder of the process after T_0, and (b) process $P(T_0 + t) : t \geq 0$ is independent of the process prior to T_k $P(t) : 0 \leq t < T_k$.*

In other words a stochastic process with certain time points such that from a probabilistic view the process restarts itself at these time points is called a regenerative process. Intuitively this means a regenerative process can be split in to i.i.d. cycles [3].

Based on equation 10 the projected sgd updates can be written as:

$$\theta^{t+1} = \Pi_{\mathcal{P}}(\theta^{(t)} + \eta_t \nabla L_x(\theta^{(t)})) \tag{13}$$

where $\Pi_{\mathcal{P}}()$ is the projection of the updated gradient with respect to the original loss $L(U, V, W)$. The projection step can be further broken up into

$$\begin{aligned} \theta^{t+1} &= \theta^{(t)} + \eta_t \nabla L_x^{s_i}(\theta^{(t)}) + \eta_t p(\theta^{(t)}) \ \ (using\ \ (11)) \\ &= \theta^{(t)} + \eta_t \nabla L^0(\theta^{(t)}) + \eta_t \delta M_t + \eta_t \beta_t + \eta_t p(\theta^{(t)}) \end{aligned} \tag{14}$$

where $L_x^{s_i}(\theta^{(t)})$ is the loss function at stratum s_i at a point x in iteration t given parameter value in previous iteration $\theta^{(t)}$. $\nabla L^0(\theta^{(t)})$ is the exact gradient in iteration t given previous parameter value $\theta^{(t)}$. And

$$\delta M_t = \nabla L_x^{s_i}(\theta^{(t)}) - \nabla L^0(\theta^{(t)}) - \beta_t \tag{15}$$

where β_t is the "error" before projection i.e. the error by which the update is outside $\mathcal{P}$.

To prove the convergence of the method we define the following conditions, similar to the ones defined in [8, 6]:
Condition 1. $\nabla L^0(\theta)$ is continuous.
Condition 2. $\nabla L^0(\theta^{(t)})$ is bounded in second moment: $E[(\nabla L^0(\theta^{(t)}))^2] < \infty$ for all θ.
Condition 3. The squared sum of the step sizes η_t is bounded i.e. $\sum_t \eta_t^2 < \infty$.
Condition 4. The noise is martinagale difference: $E[\delta M_{(t+1)} | \delta M_i, i \leq t] = \delta M_t$.
Condition 5. $E[\eta_t \beta_t] < \infty$ with probability 1. Note that this is a condition on step-size. It implicitly says that the projection must not wander off infinitely outside the set $\mathcal{P}$ over the iterations.

Theorem 2 *The distributed SGD algorithm for tensor decomposition with projections, as presented in algorithm 1, converges.*

Proof. The primary equations being updated each time in our iterations is equation 14. Rewriting it here we have:

$$\theta^{t+1} = \theta^{(t)} + \eta_t \nabla L^0(\theta^{(t)}) + \eta_t \delta M_t + \eta_t \beta_t + \eta_t p(\theta^{(t)}) \tag{16}$$

From theorem 1 we can see that the individual blocks in a given stratum are independent of each other's updates and are interchangeable. We can also observe from Algorithm 1 that every stratum out of d strata is picked exactly once in one cycle i.e. one epoch (outer while loop). Moreover two different cycles of strata i.e. iterations of the while loop are identical and independent. In other words the while loop forms an i.i.d cycle, and thus a regenerative process. The time-period of cycles is finite and bounded consequently that of the regenerative process too. Besides given all the conditions 1 to 5 as defined above, we have

$$\left[\sum_{i=0}^{\kappa(t+t_0)-1} (\eta_i \delta M_i + \eta_i \beta_i) \right] \to 0 \tag{17}$$

for any arbitrary κ. The proof is similar to [8] and is valid due to the fact that noise is a martingale difference sequence and $\eta_i \delta M_i$ and $\eta_i \beta_i$ are an equicontinuous sequence ([8] Theorem 2.1, part 1, chapter 5; [6] follows a similar proof up to this point). We can now use this to analyze the updated with a projected loss. We find that equation 14 has the same set of stable points as

$$\theta^{t+1} = \theta^{(t)} + \eta_t \nabla L^0(\theta^{(t)}) + \eta_t p(\theta^{(t)}) \tag{18}$$

Now we show that equation 18 converges. Through few algebraic manipulations it can be verified that the projection functions $p(\cdot)$ we have, $\ell 1$ soft threshold and non-negativity constraint project, are lipschitz continuous. Following the arguments of [8] (theorem 2.1, part 2) and with the assumption that updates $\theta^{(t)}$ are bounded (follow from the conditions 1 to 5 assumed earlier), equation 18 converges to a set of stationary points. □

D MapReduce Implementation of FLEXIFACT

Along with our theoretical analysis, we implemented our algorithm within the MapReduce framework [5]. To do this we used the open source Hadoop [1] version of MapReduce. The challenge is to turn the factorization problem into `map()` and `reduce()` functions, that Hadoop is designed to handle.

In our implementation we pass the data matrix or tensor as input to the mappers in the form $(i, j, k, \mathcal{X}_{i,j,k})$. We also store our current parameters $\theta^{(s)}$, which could include $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$, and $\mathbf{A}$ on the Hadoop File System (HDFS).

Algorithm 2: FLEXIFACT Mapper (for tensor)

Input: I, J, K, d
1: **for all** $(i, j, k, \mathcal{X}_{i,j,k})$ **do**
2: $b_i = \lfloor \frac{i}{\lceil \frac{I}{d} \rceil} \rfloor$, $b_j = \lfloor \frac{j}{\lceil \frac{J}{d} \rceil} \rfloor$, $b_k = \lfloor \frac{k}{\lceil \frac{K}{d} \rceil} \rfloor$ {Get block index}
3: *subepoch* $= d \times ((b_k - b_i + d) \bmod d) + ((b_j - b_i + d) \bmod d)$
4: **emit** $\langle (b_i, b_j, b_k, \textit{subepoch}), (i, j, k, \mathcal{X}_{i,j,k}) \rangle$
5: **end for**

Algorithm 3: FLEXIFACT Reducer (for tensor)

Given: $\mathbf{U}, \mathbf{V}, \mathbf{W}, I, J, K, d, \eta$
Input: Values $\mathcal{V}$
s_{old} =Pointer to final parameters from last epoch
for $(i, j, k, \mathcal{X}_{i,j,k}) \in \mathcal{V}$ **do**
$b_i = \lfloor \frac{i}{\lceil \frac{I}{d} \rceil} \rfloor$, $b_j = \lfloor \frac{j}{\lceil \frac{J}{d} \rceil} \rfloor$, $b_k = \lfloor \frac{k}{\lceil \frac{K}{d} \rceil} \rfloor$
$t = d \times ((b_k - b_i + d) \bmod d) + ((b_j - b_i + d) \bmod d)$
if $t \neq t_{\text{old}}$ **then**
Write $\mathbf{V}^{(s_{\text{old}})}_{b_{j_{\text{old}}}}$ and $\mathbf{W}^{(s_{\text{old}})}_{b_{k_{\text{old}}}}$ to HDFS
Wait for $\mathbf{V}^{(s_{\text{old}})}_{b_j}$ and $\mathbf{W}^{(s_{\text{old}})}_{b_k}$ to be available on HDFS
Save $\mathbf{U}^{(s)}_{b_i} = \mathbf{U}^{(s_{\text{old}})}_{b_i}$ Save $\mathbf{V}^{(s)}_{b_j} = \mathbf{V}^{(s_{\text{old}})}_{b_j}$ and $\mathbf{W}^{(s)}_{b_k} = \mathbf{W}^{(s_{\text{old}})}_{b_k}$ from HDFS $b_{i_{\text{old}}} = b_i$, $b_{j_{\text{old}}} = b_j$, $b_{k_{\text{old}}} = b_k$, $t = t_{\text{old}}$
end
$\theta^{(s)} = \theta^{(s)} - \eta \nabla \mathcal{L}_{\mathcal{X}_{i,j,k}}(\theta^{(s)})$ (where $\theta^{(s)}$ is the concatenation of $\mathbf{U}^{(s)}_{b_i}, \mathbf{V}^{(s)}_{b_j}, \mathbf{W}^{(s)}_{b_k}$)
end
Write $\mathbf{U}^{(s_{\text{old}})}_{b_{i_{\text{old}}}}$, $\mathbf{V}^{(s_{\text{old}})}_{b_{j_{\text{old}}}}$, and $\mathbf{W}^{(s_{\text{old}})}_{b_{k_{\text{old}}}}$ to HDFS

FLEXIFACT Mapper: Our Mapper function, is shown in Algorithm 2. It splits the data into the appropriate blocks and determines the order they should be processed in within each reducer. We also overload the default Hadoop partitioner, which typically just partitions on unique KEY values, and now partition only on b_i so that each reducer represents a unique set of i in I or a unique set of rows in $\mathbf{U}$. We additionally override the default Comparator, allowing us to sort our (KEY,VALUE) pairs within each reducer by the *subepoch* term calculated in the Mapper. We see here while the Mapper is quite simple, the calculation of the blocks and the order within each reducer captures our partition function that allows us to perform SGD in this distributed fashion.

FLEXIFACT Reducer: Our Reducer function is shown in Algorithm 3. As we explained before, each reducer gets all points for a given range of values i ordered by the subepochs. (As before we use s to denote the subepochs.) The reducer iterates over the points in $\mathcal{V}$ in order, each time updating $\theta^{(s)}$. Each reducer only stores the components of θ that correspond to its current block in the current stratum. As such, when a new subepoch is reached, it must write its updated θ values to disk (for another reducer to retrieve) and read the most current θ values for its next block in the subsequent subepoch.

We run the MapReduce jobs iteratively. Each MapReduce job is one epoch using all points in $\mathcal{X}$ to update the full parameter space θ and ultimately to save it to HDFS. We then use the updated

parameters θ in the subsequent epoch (another run of the MapReduce algorithm). We do this for a constant number of steps or until the algorithm converges.

Reproducibility and Usability: This is a high level overview of our implementation, but captures the general method we use to both distribute our work and optimize our speed within the MapReduce framework. While this is not the typical way Hadoop is programmed, it requires no modification of the Hadoop framework and can be run on *any* standard Hadoop cluster. Our code is open-sourced, and available at `http://alexbeutel.com/l/flexifact`. It can run for all of the data types and loss functions described in this paper.