Scalable matrix factorization for data fusion

A dissertation presented
by

Andrej Čopar

to
The Faculty of Computer and Information Science
in partial fulfilment of the requirements for the degree of
Doctor of Science
in
Computer and Information Science

Ljubljana, 2019
I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgement has been made in the text.

— Andrej Čopar —
October 2019

The submission has been approved by

dr. Blaž Zupan
Professor of Computer and Information Science
advisor

dr. Marinka Žitnik
Assistant Professor of Computer and Information Science
advisor

dr. Uroš Lotrič
Associate Professor of Computer and Information Science
examiner

dr. Polona Oblak
Associate Professor of Computer and Information Science
examiner

Riccardo Bellazzi, PhD
Professor of Bioengineering
external examiner
University of Pavia, Italy
I hereby declare that the research reported herein was previously published/submitted for publication in peer reviewed journals or publicly presented at the following occasions:


I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Ljubljana.
Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Andrej Čopar
Skalabilna matrična faktorizacija za zlivanje podatkov

POVZETEK


Sodobne metode za zlivanje podatkov temeljijo na matrični faktorizaciji. Matrična faktorizacija se nauči skritega podatkovnega modela, ki omogoča posplošitev modela, odstrani šum ter odkrije nove značilke. Matrična tri-faktorizacija je pogosto uporabljena oblika faktorizacije, ki ni omejena s predpostavko, da podatki ležijo v enem samem prostoru. Matrična tri-faktorizacija izlušči ločen skriti prostor za vsako dimenzijo posebej in se uporablja kot osnovni gradnik metod zlivanja podatkov. Algoritmi za faktorizacijo so računsko zahtevni, zato je njihova prilagoditev za velike podatke ključnega pomena za razvoj hitrih metod zlivanja podatkov.

Trenutne metode nenegativne matrične tri-factorizacije se naučijo predstavitve modela z uporabo optimizacijskih postopkov, ki temeljijo na multiplikativenih pravilih. Ta postopek omogoča počasna konvergenca. Razvili smo tri alternativne načine za matrično tri-faktorizacijo, ki temeljijo na postopku izmenjujočih najmanjših kvadratov, postopku projiciranih gradientov in postopku koordinatnega spusta. Naredili smo empirično analizo, s katero smo primerjali postopek multiplikativenih pravil z ostalimi alternativnimi tehnikami. Pokazali smo, da postopek projiciranih gradientov konvergira tri-krat hitreje, postopek koordinatnega spusta pa tudi do 20-krat hitreje v primerjavi z multiplikativenimi pravili.


Ključne besede Strojno učenje, bioinformatika, matrična faktorizacija, zlivanje podatkov
ABSTRACT

Data collection technologies are advancing quickly and are producing larger amounts of data than ever before. Biomedical data analysis, text analysis and recommender systems rely on machine learning to perform tasks such as modeling gene-disease associations, clustering documents, and user recommendations. The analysis of such data is particularly challenging due to the large dimensionality and multitude of different object types. Data fusion methods can accurately deal with such heterogeneous datasets by integrating them into a single model. Existing data fusion approaches were not designed for speed on huge datasets and can be prohibitively slow for practical use. Our main goal is to develop new methods that increase the speed of data fusion using efficient optimization techniques and modern parallel systems.

Contemporary data fusion methods are based on matrix factorization as its core component. Matrix factorization learns a latent data model that transforms the data into a latent feature space enabling generalization, noise removal and feature discovery. Matrix tri-factorization is a popular method that is not limited by the assumption of standard matrix factorization about data residing in one latent space. Matrix tri-factorization infers separate latent space for each dimension, making the approach ideal for data fusion. Factorization algorithms are numerically intensive, hence scaling current algorithms to work with large datasets is crucial for development of fast data fusion approaches.

We developed a block-wise approach for latent factor learning in matrix tri-factorization. The approach partitions a data matrix into disjoint submatrices that are treated independently and fed into a parallel factorization system. We show that our approach scales well on multi-processor and multi-GPU architectures. Our approach on four GPU devices is more than a hundred times faster than its single-processor counterpart.

Currently, non-negative matrix tri-factorization learns a representation of a dataset through an optimization procedure that typically uses multiplicative update rules.
procedure has had limited success due to its slow convergence. We develop three alternative optimization techniques for non-negative matrix tri-factorization based on alternating least squares, projected gradients, and coordinate descent. We perform an empirical study comparing multiplicative update rules with the three alternative techniques and show that coordinate descent-based techniques converges up to twenty times faster compared to multiplicative updates.

Finally, we employ block-wise techniques together with coordinate descent to speed up data fusion. With block-wise parallelization we accelerate an existing data fusion approach over 30 times. We derive a new coordinate descent-based data fusion approach that converges over 15 times faster compared to existing approach. Coordinate-descent data fusion accelerated on GPU devices performs over 100 times faster compared to an existing approach on 16 processes.

**Keywords** Machine learning, bioinformatics, matrix factorization, data fusion
ACKNOWLEDGEMENTS

I wish to thank my advisor Blaž Zupan for guiding me through my years as a doctoral student and as a researcher. I would also like to thank my co-advisor Marinka Žitnik for priceless advice and feedback. Without them this thesis would not be possible.

I would like to thank my co-workers in Biolab for additional motivation and pleasant working environment.

Finally, I would like to thank my family for support and patience.

— Andrej Čopar, Ljubljana, October 2019.
CONTENTS

Povzetek vii
Abstract ix
Acknowledgements xi
1 Introduction 1
1.1 Scientific contributions 3
2 Parallelization of non-negative matrix tri-factorization 5
  2.1 Related work 7
     2.1.1 Objective functions 7
     2.1.2 Orthogonality constraints 8
     2.1.3 Tensor factorization 9
  2.2 Methods 9
     2.2.1 Block-wise multiplicative update rules 10
     2.2.2 Matrix partitioning 12
     2.2.3 Overview of block-wise matrix tri-factorization 12
     2.2.4 Parenthesization of update rules 13
  2.3 Data and experimental setup 18
     2.3.1 Data 18
     2.3.2 Experimental setup 19
     2.3.3 Time complexity 20
     2.3.4 Multi-core parallelization 21
     2.3.5 GPU and multi-GPU parallelization 22
2.4 Results .......................................................... 24
2.5 Discussion ....................................................... 27
  2.5.1 Speedup ................................................... 27
  2.5.2 Efficiency effects of block configuration .............. 27
  2.5.3 Factorization rank ......................................... 28
  2.5.4 Communication overhead ................................. 29
  2.5.5 Scalability ................................................ 30
  2.5.6 Memory requirements ..................................... 33
  2.5.7 Speedups of sparse data format ......................... 34
  2.5.8 Interpretation of factorization results ................. 36
2.6 Conclusion ...................................................... 39

3 Convergence of non-negative matrix tri-factorization  41
  3.1 Related work ................................................ 43
  3.2 Methods ....................................................... 44
    3.2.1 Multiplicative update rules for NMTF ................. 44
    3.2.2 Alternating least squares for NMTF ................... 45
    3.2.3 Projected gradients for NMTF ........................... 45
    3.2.4 Coordinate descent for NMTF ......................... 47
    3.2.5 Optimization algorithms for NMTF ..................... 48
  3.3 Data and preprocessing ...................................... 49
  3.4 Results ........................................................ 50
    3.4.1 Experimental setup ................................... 50
    3.4.2 Convergence of NMTF optimization methods .......... 51
    3.4.3 Analysis of matrix tri-factorization runtime ........ 52
    3.4.4 Impact of factorization rank on optimization runtime 52
    3.4.5 Stochastic mini-batch approach to NMTF ............. 54
    3.4.6 Impact of initialization on convergence ............. 55
  3.5 Discussion ..................................................... 55
  3.6 Conclusion .................................................... 58

4 Speedup of data fusion ........................................ 65
  4.1 Related work ................................................. 67
    4.1.1 Collective matrix factorization ...................... 69
A. Čopar
Scalable matrix factorization for data fusion

A.4 Computational complexity of objective value 116

Bibliography 117

Razširjeni povzetek 125
Introduction
Biomedical data are becoming increasingly challenging to analyze due to their sheer volume and complexity. Dimensionality reduction approaches address challenges in modern biomedical data analytics by learning useful projections of data into a smaller, compact and pattern-rich latent space. An especially popular dimensionality reduction approach uses matrix factorization [1]. Numerous non-negative matrix factorization methods have successfully been used for gene expression analysis [2–4], patient stratification [5], drug-target interaction discovery [6], gene phenotyping [7], and magnetic resonance image analysis [8–10].

Matrix factorization deals with interactions of up to two object types. Heterogeneous data sources contain interactions between more than two types of objects, and potentially contain millions or even billions of elements. Data fusion enables us to infer a single data model from heterogeneous datasets that encompass many object types and relation types. Existing approaches for data fusion have been tried in various studies [11–13], including systems biology and biomedicine, and have shown their benefits in terms of prediction accuracy and reasoning across different domains [14]. They were not designed for scalability on large datasets, and their concurrent implementations have yet to be invented.

Speeding up data fusion for larger datasets can be approached in two different ways. First, increased speed is commonly achieved by adding more computational resources. In order to benefit from highly parallel environments, the methods need to be designed such that additional computational resources decrease the runtime of the method. In other words, the method needs to be scalable, which is a challenging task. Second, increased speed can be achieved by lowering the computational requirements. The total runtime of factorization can be decreased, if the optimization technique converges faster.

Recent data fusion methods are based on matrix tri-factorization [12, 13, 15]. Non-negative matrix tri-factorization (NMTF) [16, 17] is a more general form of two-factor non-negative matrix factorization, a widely used dimensionality reduction approach with many applications [18–20]. A large number of studies work on improving the speed of the related two-factorization. Classical two-way matrix factorization has been successfully ported to highly parallel systems [21–23] and utilized block-formulation to improve efficiency on MapReduce cluster [24]. Studies also show that faster optimization techniques successfully improved the convergence of two-factorization [25–27].
The corresponding approaches to matrix tri-factorization and data fusion have yet to be invented, which is the main focus of our research.

We developed a block-wise approach for latent factor learning in matrix tri-factorization that allows for better parallelization on multi-processor and multi-GPU systems. The approach partitions a data matrix into disjoint submatrices that are treated independently and fed into a parallel factorization system. In a study on large biomedical datasets we show that our approach scales well on multi-processor and multi-GPU architectures. On a four-GPU system we demonstrate that our approach can be more than 100-times faster than its single-processor counterpart.

We increase convergence of the matrix tri-factorization using alternative optimization techniques. We develop three training algorithms for non-negative matrix tri-factorization that use projected gradients, coordinate descent, and alternating least squares optimization. The success of these three methods for various tasks in machine learning [28–30] encouraged us to adapt them for non-negative matrix tri-factorization. We show convergence and runtime improvements of the new training algorithms over traditional multiplicative update rules on six datasets. We show that coordinate descent converges more than 20 times faster compared to existing approach with more than 15 times faster runtime.

We developed a parallel block-wise variants of existing data fusion approach [13]. We show more than 100 times faster factorization time compared to existing approach. We derived an alternative data fusion approach based on coordinate descent optimization technique. We show on a collection of datasets that the proposed approach converges faster than related approach using multiplicative updates.

1.1 Scientific contributions

The three main scientific contributions can be summarized in the following list:

1. formulation of NMTF method with block-wise update rules,
2. derivation of coordinate descent rules for NMTF,
3. scalable method for data fusion based on block-wise update rules and coordinate descent.
Chapter 2 presents our efficient block-wise formulation of NMTF which we also present in our paper [31]. Chapter 3 presents convergence improvements of NMTF by alternative optimization techniques such as coordinate descent. Chapter 4 shows the improvements of existing data fusion approach with parallelization, improvements in convergence of data fusion and finally combines these two approaches in a scalable data fusion approach.
Parallelization of non-negative matrix tri-factorization
Matrix tri-factorization is a popular method that is not limited by the assumption of standard matrix factorization about data residing in one latent space. Factorization algorithms are numerically intensive, and hence there is a pressing challenge to scale current algorithms to work with large datasets.

We develop a principled mathematical approach and an algorithmic solution to latent factor learning for non-negative matrix tri-factorization. While there exists an initial solution to speed up the latent factor learning procedure using accelerated matrix operations on a MapReduce cluster [32], this approach is not optimal because it requires a specialized architecture [33]. In the case of two-factor non-negative matrix factorization, it was shown that the MapReduce based approach was outperformed by block-wise approaches by two orders of magnitude [24]. Block-wise approaches also provide the means for load balancing. These related studies thus encourage the development of a block-wise approach for matrix tri-factorization.

We make the following contributions. We develop a block-wise approach for matrix tri-factorization. The new approach enables fast factorization on concurrent systems, such as multi-processor and multi-GPU architectures. We report on two variants of the approach: one variant for orthogonal matrix factorization [17] and the other for non-orthogonal matrix factorization [16]. We provide implementation of the new approach for both multi-processor and multi-GPU architectures. We evaluate the proposed approach with respect to dataset shape and size, parallelization degree, factorization rank, and data sparsity. In experiments on several biomedical datasets, we demonstrate that the new approach provides substantial speedups. The speedup is most pronounced on multi-GPU architectures, where matrix tri-factorization can be more than 100-times faster than its serial counterpart.
2.1 Related work

Non-negative (two-factor) matrix factorization considers as input a data matrix $X$ and learns two latent factors, $U$ and $V$, such that their product $UV$ approximates $X$, $X = UV$, under some criterion of approximation error. One class of non-negative matrix factorization approaches is non-negative matrix tri-factorization, which extends the two-factor model by introducing a third latent factor $S$, such that $X \approx USV^T$ [17]. This representation is more appropriate for non-square data because it explicitly models data interactions through a latent factor $S$ [34].

Several parallel non-negative (two-factor) matrix factorization have been proposed [21–23]. These techniques first partition matrix $X$ into blocks and then exploit the block-matrix multiplication when learning $U$ and $V$. However, such a straightforward approach does not apply to matrix tri-factorization because, the learning of any block of $U$ and $V$ depends on factor $S$.

2.1.1 Objective functions

Consider a non-negative data matrix $X \in \mathbb{R}_{+}^{n \times m}$, where $n$ rows typically describe data instances and $m$ columns provide their features. Non-negative matrix tri-factorization (NMTF) learns a decomposition of $X$ into three latent factors $U \in \mathbb{R}_{+}^{n \times k_1}$, $S \in \mathbb{R}_{+}^{k_1 \times k_2}$, and $V \in \mathbb{R}_{+}^{m \times k_2}$. Discrepancy between input data matrix $X$ and its reconstruction $X' = USV^T$ is measured through a loss function that aims to minimize the following Frobenius distance [35, 36] $D_{\text{Fro}}$:

$$D_{\text{Fro}}(X||USV^T) = \|X - USV^T\|_{\text{Fro}}^2.$$  \hspace{1cm} (2.1)

Columns in factors $U$ and $V$ are latent vectors and provide the basis of the vector space into which the data (columns and rows of $X$, respectively) are projected. Factorization ranks $k_1, k_2 \ll \min(m, n)$ are model parameters that specify the number of latent vectors.

Reconstruction error is typically minimized using the multiplicative update rules [16]. The rules are derived by computing the gradient of the reconstruction error $F$ with respect to model parameters $U$, $S$, and $V$ and by solving the gradient equations for the model parameters. The full derivations are given in A.2.1 and result in the following set
of update rules:

\[ U \leftarrow U \odot XVS^T \odot USV^T V^T, \tag{2.2} \]
\[ V \leftarrow V \odot X^T US \odot VS^T U^T US, \tag{2.3} \]
\[ S \leftarrow S \odot U^T XV \odot U^T USV^T V, \tag{2.4} \]

where \( \odot \) represents the Hadamard product and the division is performed entry-wise.

The matrix tri-factorization algorithm starts by initializing latent factors using small random values and then iteratively applies the update rules in Eq. (2.2–2.4) until convergence [1].

In addition to Frobenius norm, matrix factorization often uses the divergence between the data and approximation defined as:

\[ D_{KL}(X||USV^T) = X \odot \log(X \odot USV^T) - X + USV^T. \tag{2.5} \]

This measure can be reduced to KL divergence [1, 37], when the data and its approximation are normalized (\(|X|_1 = 1\) and \(|USV^T|_1 = 1\)). The logarithm is performed entry-wise.

The Frobenius norm-based update rules are more efficient, because we can avoid explicit computation of \( USV^T \), which is the most time consuming operation. Because of our focus on efficiency, Frobenius-norm based rules are used in the rest of this work.

Several other loss functions exist, such as the Alpha divergence [38], and Beta divergence [39], however their corresponding tri-factorization update rules have not yet been developed.

### 2.1.2 Orthogonality constraints

In addition to non-negativity, we can promote other structural properties by including additional regularization terms in the loss function \( D_{Fro} \). In particular, in clustering applications, we might want to impose orthogonality on \( U \) and \( V \) [17], such that \( U \)'s and \( V \)'s latent vectors indicate memberships of row and column objects in distinct clusters [40]. For example, adding \( U^T U \) regularization term to the loss function will make latent vectors in \( U \) to be orthogonal to each other. Another popular approach is to impose sparsity constraints on latent matrices by including \(|U|_1 \) and \(|V|_1 \) regularization in the loss function [41].
Orthogonal matrix tri-factorization needs to satisfy two additional constraints: $U^T U = I$ and $V^T V = I$, where $I$ is an identity matrix. Following a similar procedure of gradient computation as described above, we arrive at the following update rules for orthogonal matrix tri-factorization:\[17]:

$$U \leftarrow U \odot \sqrt{XVS^T} \odot UU^T XVS^T,$$  
$$V \leftarrow V \odot \sqrt{X^T US} \odot VV^T X^T US,$$  
$$S \leftarrow S \odot \sqrt{U^T XV} \odot USV^T V.$$  
\[2.6\]  \[2.7\]  \[2.8\]

2.1.3 Tensor factorization

Non-negative matrix two-factorization and three-factorization both deal with two-dimensional arrays. Data often appears in the form of multi-dimensional tensors, for example in signal processing [18] and image analysis [42]. Adding additional dimension, such as temporal data, extends the matrix dataset into a multi-dimensional tensor dataset. Tensor factorization approaches are capable of modeling three-way or multi-way datasets, with applications in neuroscience for analysis of EEG and fMRI data [43].

Data is commonly very sparse in high-dimensional datasets, which can lead to poor reconstruction due to higher number of variables [44]. Furthermore, the time complexity and memory consumption for tensor factorization are substantially higher compared to two-factorization. Memory-efficient approach has been developed to lower memory consumption [45], where parallel GPU implementations have been proposed to lower the computational time of tensor factorization [46, 47]. Similar parallel approaches for three-factorization are needed, which is our focus in the rest of this chapter.

2.2 Methods

We start by describing the notation, factorization model, and matrix tri-factorization algorithm. The algorithm starts by initializing the latent factors, which are then iteratively revised until convergence. We then introduce a block data representation and provide an algorithm for partitioning data matrices into blocks. We use block data representation to derive block-wise latent factor update rules and implement the block-wise tri-factorization algorithm. Finally, we describe matrix parenthesization which is used to optimize matrix operations in the update rules.
2.2.1 Block-wise multiplicative update rules

We present a block-wise formulation of multiplicative update rules for NMTF. We partition the input data $\mathbf{X}$ into $N \times M$ blocks, $\mathbf{X}^{(I,J)}$, where $I \in \{0, 1, \ldots, N - 1\}$ and $J \in \{0, 1, \ldots, M - 1\}$. Conversely, latent factor $\mathbf{U}$ is row-partitioned into $N$ blocks, and $\mathbf{V}$ is column-partitioned into $M$ blocks. Fig. 2.1 shows an example where matrix $\mathbf{X}$ is row-partitioned into $N = 3$ blocks and column-partitioned into $M = 2$ blocks.

![Figure 2.1](image_url)

Using this block-wise data representation we reformulate the multiplicative update rules from Eq. (2.2–2.4) as follows:

$$
\mathbf{U}^{(I)} \leftarrow \mathbf{U}^{(I)} \odot \sum_j \mathbf{X}^{(I,J)} (\mathbf{V}^{(J)T} \mathbf{S}^T) \odot \\
\mathbf{U}^{(I)} \sum_j \left( \mathbf{S} \left( \mathbf{V}^{(J)T} \right) \left( \mathbf{V}^{(J)} \mathbf{S}^T \right) \right),
$$

(2.9)

$$
\mathbf{V}^{(J)} \leftarrow \mathbf{V}^{(J)} \odot \sum_i \left( \left( \mathbf{X}^{(I,J)} \right)^T \mathbf{U}^{(I)} \right) \mathbf{S} \odot \\
\mathbf{V}^{(J)} \mathbf{S}^T \sum_i \left( \left( \mathbf{U}^{(I)} \right)^T \mathbf{U}^{(I)} \right) \mathbf{S},
$$

(2.10)

$$
\mathbf{S} \leftarrow \mathbf{S} \odot \sum_j \sum_i \left( \left( \mathbf{U}^{(I)} \right)^T \mathbf{X}^{(I,J)} \right) \mathbf{V}^{(J)} \odot \\
\sum_i \left( \left( \mathbf{U}^{(I)} \right)^T \mathbf{U}^{(I)} \right) \mathbf{S} \sum_j \left( \left( \mathbf{V}^{(J)} \right)^T \mathbf{V}^{(J)} \right),
$$

(2.11)

where $I$ and $J$ denote $I$-th row and $J$-th column matrix block, respectively. The parenthesization is determined according to procedures described in section 2.2.4. Notice that our block partitioning scheme and update rules in Eq. (2.9–2.11) preserve all properties.
of factorizing a non-partitioned matrix $X$. That is, the result of block-wise matrix tri-factorization is identical to the result returned by non-partitioned matrix tri-factorization as proposed by Long et al. [16]. For example, consider an update for factor $U$ in Eq. (2.2) and its block-wise variant in Eq. (2.9). To show that these two update rules are equivalent, we need to check that the values in $U^{(i)}$ are identical to the values of $U$ at corresponding positions. Notice that division in both updates is element-wise; hence, we can independently check equivalency of numerator and denominator. For example, the numerator in Eq. (2.2) is expressed as $XVS^T$. An $I$-th row of this expression can be written in a block-wise manner as $\sum_j X^{(I,J)}V^{(J)}S^T$, which is exactly the corresponding numerator in Eq. (2.9). The equivalency of other terms of non-partitioned and block-wise updated rules are further shown in proof of equivalence of block-wise and non-block-wise formulation in appendix A.1.

Next, we propose update rules for block-wise orthogonal matrix tri-factorization:

$$
U^{(I)} \leftarrow U^{(I)} \odot \left( \sum_j X^{(I,J)} (V^{(J)}S^T) \right) \odot \left( U^{(I)} \sum_i (U^{(I)})^T \sum_j (X^{(I,J)} (V^{(J)}S^T))) \right)^{1/2}, \tag{2.12}
$$

$$
V^{(J)} \leftarrow V^{(J)} \odot \left( \sum_j (X^{(I,J)})^T U^{(I)} \right) S \odot \left( V^{(J)} \sum_i (V^{(J)})^T V^{(J)}) \sum_j (X^{(I,J)} (V^{(J)}S^T)) \right)^{1/2}, \tag{2.13}
$$

$$
S \leftarrow S \odot \left( \sum_j \sum_i ((U^{(I)})^T U^{(I)}) (V^{(J)}V^{(J)}) \sum_j (X^{(I,J)} (V^{(J)}S^T))) \right)^{1/2}. \tag{2.14}
$$

The formulation is identical to the non-block-wise formulation, originally proposed in Ding et al. [17] and shown in Eq. (2.6–2.8). As before, this property is important because it indicates the proposed block-wise update rules yield latent factors that are identical to the non-block-wise update rules in Ding et al. [17].
2.2.2 Matrix partitioning

To partition data matrix $X$ and latent factors $U$, $V$, and $S$ into blocks, we distinguish between sparse and dense data matrices. In general, most elements of sparse data matrices are zero, whereas most elements of dense matrices are nonzero \[48\]. In the case of dense matrix $X$, our matrix partitioning procedure splits $X$ into contiguous blocks of approximately equal size. In the case of sparse matrix $X$, we adapt the block size such that each block contains approximately equal number of nonzero elements. Such partitioning leads to workload balancing when factorization is carried out in parallel.

The details of matrix partitioning are provided in Algorithm 1. The algorithm takes as input a data matrix $X$ and a desired block-wise configuration and returns an appropriate partitioning of $X$. Additional parameters are the number of row blocks $N$ and column blocks $M$. Partitioning of latent factors $U$, $V$, and $S$ is determined by the partitioning of matrix $X$ (for example, see Fig 2.2). Slices of matrix $X$ are shown with $X(i:j,\cdot)$, where $i:j$ denotes range between $i$-th and $j$-th row and $\cdot$ represents all columns.

![Figure 2.2](image)

Block-wise partitioning of data matrix $X$ using a $2 \times 2$ configuration. Latent factors $U$ and $V$ are each partitioned into two blocks. The remaining latent factor $S$ is not partitioned into blocks.

2.2.3 Overview of block-wise matrix tri-factorization

A complete algorithm for matrix tri-factorization is given as Algorithm 2. The algorithm starts with matrix partitioning, followed by initialization of latent factors. Initial latent factors are then iteratively refined until convergence using the proposed block-wise multiplicative update rules. Convergence is heuristically determined by observing the value of the objective function or the quality of latent factors and corresponding reconstruction error \[16, 17\]. The algorithm is stopped when the relative difference in objective function becomes sufficiently small \[49\].
Algorithm 1 Algorithm for partitioning data matrix $X$ into $N \times M$ block matrices.

Input: Data matrix $X \in \mathbb{R}^{n \times m}$, Number of row blocks $N$, Number of column blocks $M$.

1: $z = \text{nonzero}(X)$
2: $r_0 = 0$
3: for $I \in \{1, \ldots, N\}$ do
4: \hspace{1em} $r_I = \min(k; \text{nonzero}(X(r_{I-1}, \cdot)) \geq I z/N)$
5: end for
6: $c_0 = 0$
7: for $J \in \{1, \ldots, M\}$ do
8: \hspace{1em} $c_J = \min(k; \text{nonzero}(X(\cdot, c_{J-1}) \geq J z/M))$
9: end for
10: for $I \in \{0, 1, \ldots, N-1\}$ do
11: \hspace{1em} for $J \in \{0, 1, \ldots, M-1\}$ do
12: \hspace{2em} $X^{(i,j)} = X(r_i : r_{i+1}, c_j : c_{j+1})$
13: \hspace{1em} end for
14: end for
15: return $X^{(i,j)}$ for all $I, J$

2.2.4 Parenthesization of update rules

Update rules contain several chains of matrix multiplications with three or more matrices. These multiplications can be computed in many different orders, while the result remains the same due to associative property of matrix multiplication. Parenthesization addresses the priority between operations of the same type to achieve better efficiency. Existing approaches focus on product-only matrix chains [50, 51]. They are not suitable for factorization rules, which are composed of heterogeneous operators, with different precedence levels. Changing the order of multiplications can significantly improve the efficiency of matrix factorization, because the factors are very wide or very tall.

Existing methods are very efficient at parenthesization of individual chains [51, 52], but do not consider the duplication of results between disjoint chains. Factorization rules also contain many duplicate sequences, in which case the independent optimiza-
### Algorithm 2 Algorithm for learning latent factors in block-wise matrix tri-factorization

Description on how to reuse the calculated latent factors is in Implementation section.

**Input:** Data matrix $X \in \mathbb{R}^{n \times m}$, Factorization ranks $k_1$, $k_2$, Number of row blocks $N$, Number of column blocks $M$, Factorization type $F \in \{\text{“orthogonal”, “non-orthogonal”}\}$

1: Partition $X$ into $N \times M$ block matrices using Algorithm 1
2: Initialize $U^{(I)} \sim \mathcal{U}(0,1)$ in parallel for $I \in \{0, 1, \ldots, N - 1\}$
3: Initialize $V^{(J)} \sim \mathcal{U}(0,1)$ in parallel for $J \in \{0, 1, \ldots, M - 1\}$
4: Initialize $S \sim \mathcal{U}(0,1)$
5: repeat
6: Update $U^{(I)}$ using Eq. (2.9) if non-orthogonal or Eq. (2.12) if orthogonal.
7: Update $V^{(J)}$ using Eq. (2.10) if non-orthogonal or Eq. (2.13) if orthogonal.
8: Update $S$ using Eq. (2.11) if non-orthogonal or Eq. (2.14) if orthogonal.
9: until $U$, $V$ and $S$ converge or maximum number of iterations is exceeded
10: return $U$, $V$ and $S$

Partition of chains often leads to sub-optimal efficiency. Consider, for example, the following update rule $B$:

$$B = A_{0}A_{1}A_{2}A_{3} + A_{2}A_{1},$$

(2.15)

containing two chains of matrices, with dimensions $k$ and $n$, where $k \ll n$. The first chain contains four matrices $A_{0}, A_{1}, A_{2}, A_{3}$, where the second chain contains two matrices $A_{2}, A_{1}$. A naive left-to-right parenthesization $(A_{0}A_{1})A_{2}A_{3}$ has a computational complexity $O(n^{2}k)$. If we optimize the first chain without considering the second, the combined time complexity of both terms is $O(nk^{2} + k^{3})$ and the parenthesization is $A_{0}(A_{1}A_{2})A_{3}$. Considering that we can reuse $A_{2}A_{3}$ term, the optimal parenthesization of the first term is $A_{0}(A_{1}(A_{2}A_{3}))$, with time complexity $O(nk^{2})$. 
We have developed a parenthesization approach tailored for the task of matrix factorization rules that considers duplication within update rules. Our method is based on a branch and bound algorithm [53], which skips over solutions with higher cost compared to current best solution. Additionally, our approach is guided by heuristics that prioritize operations with potentially lower overall complexity. Among the same-cost operations, the priority is given to the operations that produce a smaller output matrix. For example, the heuristic function favors multiplication of matrices with dimensions $k \times n$ and $n \times k$, compared to matrices with dimensions $n \times k$ and $k \times k$. The output matrix of the first case is $k \times k$, where the second case results in matrix with dimensions $n \times k$.

Algorithm 3 shows the proposed search algorithm, which optimizes parenthesization of operations. The algorithm is similar to depth first search algorithm with bounding step $S + oper < best$, which skips branches of solutions that would result in computational cost worse than the current best solution. Algorithm 4 describes generate function, which returns an ordered list of operations such that lower cost operations take priority. Move function performs the multiplication by joining two matrices into a partial result, where undo function reverts the action taken by move.

Each multiplication is associated with a specific time and space complexity. Since the dataset shape and factorization ranks are unknown at this stage, we use symbolic notation of dimensions 1, $k$, and $n$, where 1 denotes single column, $k$ any of the factorization ranks, and $n$ any of the dataset dimensions. We make some assumptions about the dimensions, one important assumption is that factorization rank is much lower than dataset size and therefore $k^2 \ll n$.

The heuristic procedure that prioritizes multiplications with smaller matrix is shown in algorithm 4. The generate function iterates over all possible multiplications, and then orders them based on the computational complexity. Among operations with the same computational complexity, operation with lower space complexity takes precedence. For example, let us optimize the first term in Eq. (2.15). We can perform one of the following three operations: $A_0 A_1$, $A_1 A_2$, and $A_2 A_3$, and their respective time complexities $O(kn^2)$, $O(k^2 n)$, and $O(k^2 n)$. The sorting function in algorithm 4 orders the operations based on the following list of priorities, shown from highest to lowest:

$$1, k, k^2, n, k^3, kn, k^2 n, n^2, kn^2, n^3.$$
Considering these complexity priorities, multiplications $A_1 A_2$ and $A_2 A_3$ are denoted with $k^2 n$ and multiplication $A_0 A_1$ is denoted with $kn^2$ which correspond to their time complexities. The algorithm will first explore the operations with lower time complexity. Given two multiplications with the same complexity, priority is given to the multiplication that forms a smaller output matrix. For example, consider the following two multiplications: $A_1 A_2$ and $A_2 A_3$. The first multiplication forms an output matrix with dimensions $k \times k$ and the second forms an output matrix with dimensions $n \times k$. The same-complexity operations are ordered based on the output size in the following order:

$$1 \times 1, 1 \times k, k \times k, 1 \times n, k \times n, n \times n.$$ 

Using this order, the multiplication $A_1 A_2$ will have higher priority than $A_2 A_3$ in the search algorithm, since $k \times k$ is ordered before $k \times n$. The prioritization heuristic relies on the idea that multiplications with smaller output will generally result in a lower overall complexity. Algorithm will then iterate over the rest of the operations to find duplicate
Algorithm 4 Generation of possible moves.

function \text{generate}(\text{Factorization rules } T) \n\begin{align*} 
1: & \quad \text{if } T \text{ is empty then} \\
2: & \quad \text{return empty} \\
3: & \quad \text{end if} \\
4: & \quad \text{possibleMoves} = \text{list}() \\
5: & \quad \text{complexities} = \text{dict}() \\
6: & \quad \text{outputSizes} = \text{dict}() \\
7: & \quad \text{for oper in } T.\text{nextOperation}() \text{ do} \\
8: & \quad \quad \text{possibleMoves} \leftarrow \text{possibleMoves} + \text{oper} \\
9: & \quad \quad \text{complexities}[\text{oper}] = \text{oper.timeComplexity} \\
10: & \quad \quad \text{outputSizes}[\text{oper}] = \text{oper.outputSize} \\
11: & \quad \text{end for} \\
12: & \quad \text{possibleMoves} \leftarrow \text{sort(possibleMoves, orderby=[complexities, outputSizes])} \\
13: & \quad \text{return possibleMoves} \\
\end{align*}

operations, which will reduce the overall computational complexity, for example in the update rule in Eq. (2.15).

The overall speed of the resulting parenthesization of our approach is approximately 1.5 times faster compared to the parenthesization that individually optimize chains. This is mostly the result of finding the duplicate $X^T U = U^T X$ in Eq. 2.9 and Eq. 2.12. Our parenthesization is several orders of magnitude faster compared to the naive left-to-right approach.

In order to detect convergence we need to calculate the objective value in each iteration, therefore its efficient calculation is crucial for a speedy approach. The additional number of operations to compute objective value is $O(nk^2)$, which is a significant improvement over naive approach $O(n^2 k)$, which also requires $O(n^2)$ memory. Appendix A.4 contains derivations of efficient objective value calculation.
2.3 Data and experimental setup

To test the benefits of the block-wise tri-factorization approach, we implemented the approach on multi-processor and multi-GPU architecture. We then tested the implementation on several biomedical datasets. Here, we describe the datasets, evaluation approach and implementation details.

2.3.1 Data

We considered the following six datasets (Table 2.1). Density is defined as the number of nonzero values divided by number of all entries in the data.

- **TCGA-BRCA** is an RNA-Seq gene expression dataset from the GDC databases [54]. The dataset contains expression measurements [55] of genes and gene variants from almost 1,300 human samples.

- **E-TABM-185** is a microarray gene expression dataset [56] available at ArrayExpress database with accession number E-TABM-185 [57]. It contains gene expression measurements from almost 6,000 human samples representing different cell and tissue types.

- **Fetus** denotes the fetus-specific functional interaction network from the GIANT database [58]. This is a network on human genes where two genes are connected if they are specifically co-expressed in fetal tissue. The fetus-specific gene interaction network [59] has 30 million interactions and is the sparsest network dataset in the GIANT database.

- **Retina** denotes the retina-specific functional interaction network from the GIANT database [58]. This is a network on human genes where two genes are connected if they are specifically co-expressed in retinal tissue. The retina-specific gene interaction network [60] has 147 million interactions.

- **Cochlea** denotes the cochlea-specific functional interaction network from the GIANT database [58]. This is a network on human genes where two genes are connected if they are specifically co-expressed in cochlear tissue. The cochlea-specific gene interaction network [61] has 280 million interactions and is the densest network dataset in the GIANT database.
Parallelization of non-negative matrix tri-factorization

- **TCGA-Methyl** denotes a DNA methylation dataset from the GDC database [54], which contains 10,181 samples from Illumina Human Methylation 450 platform [62]. Each sample contains methylation beta values for over 485,577 CpG sites.

Table 2.1
Summary of datasets. We manually categorized each data matrix into three shapes: tall datasets have substantially more rows than columns, wide datasets vice versa, and square datasets have a comparable number of rows and columns. Density denotes the fraction of nonzero matrix elements. The number of nonzero elements in each matrix is given in the last column (in millions of elements).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Rows</th>
<th>Columns</th>
<th>Shape</th>
<th>Density</th>
<th>Nonzero</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetus</td>
<td>25,569</td>
<td>25,608</td>
<td>square</td>
<td>4.7%</td>
<td>31M</td>
</tr>
<tr>
<td>TCGA-BRCA</td>
<td>1,222</td>
<td>60,483</td>
<td>wide</td>
<td>100.0%</td>
<td>74M</td>
</tr>
<tr>
<td>E-TABM-185</td>
<td>5,896</td>
<td>22,283</td>
<td>wide</td>
<td>100.0%</td>
<td>131M</td>
</tr>
<tr>
<td>Retina</td>
<td>25,823</td>
<td>25,822</td>
<td>square</td>
<td>22.0%</td>
<td>147M</td>
</tr>
<tr>
<td>Cochlea</td>
<td>25,824</td>
<td>25,824</td>
<td>square</td>
<td>42.0%</td>
<td>280M</td>
</tr>
<tr>
<td>TCGA-Methyl</td>
<td>10,181</td>
<td>485,577</td>
<td>wide</td>
<td>81.4%</td>
<td>3841M</td>
</tr>
</tbody>
</table>

2.3.2 Experimental setup

We factorized each dataset on multi-processor and on multi-GPU architectures. To asses the runtime statistics for a single iteration of factorization, factorization was run for 100 iterations, and measurements were averaged across ten runs. To test relationship between scalability and factorization rank, we varied parameter $k$, such that $k \in \{10, 20, ..., 100\}$. For a given dataset and a given value of factorization rank, factor matrices were initialized to the same values across different platforms.

We considered the following runtime metrics:

- **Speedup** was expressed as the ratio between processing time $t_{CPU-1}$ on a single CPU, and processing time $t_{CPU-p}$ on a multi-processor architecture using $p$ processes: $s_{CPU-p} = \frac{t_{CPU-1}}{t_{CPU-p}}$. Speedup on GPU is defined as the ratio between iteration time of single CPU and multi-GPU: $s_{GPU-p} = \frac{t_{CPU-1}}{t_{GPU-p}}$.

- **Efficiency** was expressed as speedup divided by the number of processing units $p$: $E_{CPU-p} = \frac{s_{CPU-p}}{p}$. On multi-GPU architecture the efficiency is defined as a ratio
between multi-GPU speedup: \( S_{\text{mGPU}→p} = \frac{t_{\text{GPU}→1}}{t_{\text{GPU}→p}} \) and the number of GPU units:
\[
E_{\text{GPU}→p} = \frac{t_{\text{mGPU}→p}}{p}.
\]

- **Scalability** is the ability of a system to accommodate an increased workload [61], in this case increasing the dataset size. Scalability is shown as a function of dataset size and parallelization degree, while maintaining the same efficiency.

### 2.3.3 Time complexity

The theoretical time complexity of serial approach is influenced by parenthesization. In this analysis, we treat the data as square \( n = m \), and the factorization rank for both object types is the same \( k_1 \approx k_2 \). We show the time complexity of serial approach using the parenthesization from Eqs. 2.9 to 2.11. Table 2.2 counts the number of multiplications denoted with \( \text{mult} \), and data transfers denoted with \( \text{transfer} \). \( N \) and \( M \) represent the partitioning scheme, where \( N \) is the number of horizontal and \( M \) is the number of vertical blocks. These results already account for the reuse of duplicated factors. For example, there are only two large multiplications with the \( X \) matrix instead of three, since \( X^T U \) from update in Eq. (2.10) can be reused in Eq. (2.11).

Although parallel implementation operates on smaller blocks of data, the combined number of multiplications is the same, therefore only the number of data transfers increases by increasing the number of blocks. Factors have been changed since last synchronization, they need to be synchronized to all nodes before they are used. Reduce operations are performed after the parallel version of multiplication, which is responsible for summing together partial results from different processes. Fig 2.5 shows when such transfers occur for the update procedure of factor \( U \).

Table 2.2

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Serial</th>
<th>Parallel</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{mult}(n^2k) )</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>( \text{mult}(nk^2) )</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>( \text{mult}(k^3) )</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>( \text{transfer}(nk) )</td>
<td>0</td>
<td>( N + M )</td>
</tr>
<tr>
<td>( \text{transfer}(k^2) )</td>
<td>0</td>
<td>( 8 \cdot N \cdot M )</td>
</tr>
</tbody>
</table>
We performed a series of experiments to determine the relation between the runtime and theoretical time complexity shown in Fig. 2.3 for multi-processor and 2.4 for multi-GPU architecture. We show the measured runtime with points, and line is corresponding function that was fit according to the following model:

\[ T = \lambda_{\text{mul}} (2n^2k + 7nk^2 + 3k^3) + \chi_{\text{tr}}pn + \psi_{\text{tr}}n \log(p) + c, \]  

(2.16)

where \( p \) denotes number of parallel processes. We use tall partitioning \( N \times 1 \), therefore \( N \) equals \( p \). \( \lambda_{\text{mul}} \) represents the time for multiplying one element. Constant \( \chi_{\text{tr}} \) represents the time to transfer one byte of data for synchronizations that scale linearly with the number of blocks. Constant \( \psi_{\text{tr}}n \log(p) \) represents synchronizations that use parallel data transfers, see 2.5 for details.

2.3.4 Multi-core parallelization

We implemented the block-wise matrix factorization in a Python module. Communication between processing units uses OpenMPI [64] with Mpi4py Python interface [65]. Matrix operations are accelerated with OpenBLAS [66] on multi-processor architectures. On multi-processor architectures we use NumPy for dense matrices and SciPy for operations on sparse matrices.

Multi-processing experiments were run on a computational server with dual-socket Xeon Silver 4110 system with 16 physical cores. Given \( p \) processing units, we split input
data matrix $X$ into $p$ blocks, testing various block configurations. Each block was passed to a processing unit that communicated the block with other units, when data for next computational steps were required. Fig. 2.5 shows an example of this computational and data transfer workflow for one update of matrix $U$ on a $2 \times 2$-block configuration. Notice this workflow applies to both 4-GPU and 4-processor architecture. We distribute blocks of data across devices, where no single device needs to store complete data or complete factors. An exception is block-value matrix $S$ in tri-factorization, which is not partitioned, but it is small enough to be stored on all devices.

2.3.5 GPU and multi-GPU parallelization

To support multi-GPU architecture we use PyCUDA [67]. Matrix operations are accelerated with CuBLAS on GPUs. On GPUs we use Scikit-cuda [68] for dense matrix operations and CuSPARSE with Python-cuda-cffi [69] for operations on sparse matrices.

Reducing the data transfers and memory usage is vital in GPU implementations, because transfers between GPU devices are very slow compared to communications between processes. For better efficiency, we transfer data and factors to GPU before the first iteration and perform all subsequent iterations on GPU directly. Additional transfers are only performed when needed, according to Fig. 2.5.

GPU experiments were run on a computational server with Intel Xeon E5-1650 processor and on four NVIDIA Titan X (Maxwell) GPUs, each with 12 GB of memory. The data between GPU devices is transferred through PCI bus without NVLINK.
Parallelization of non-negative matrix tri-factorization

Figure 2.5
Computational and data transfer workflow for block-wise update of factor matrix \( \mathbf{U} \) on architecture with four processing units and data matrix \( \mathbf{X} \) partitioned into \( 2 \times 2 \) blocks. Each vertical band represents a processing unit (PU0 to PU3). Stages where all data are available for the next wave of asynchronous operations are horizontally aligned and are marked with \( t_i \), \( i \in \{0,1,...,11\} \).
2.4 Results

We here present results for non-orthogonal block-wise matrix tri-factorization. Results for orthogonal block-wise matrix tri-factorization are qualitatively the same.

Figures 2.6 and 2.7 show speedups achieved on multi-processing and multi-GPU architectures, for each of six considered biomedical datasets. Runtime performance was tested on architectures with one, two or four processing units. Data matrices were partitioned according to block configurations in Table 2.3.

![Speedup Graph](image)

**Figure 2.6**
Computational speedups on multi-processing architectures. Speedup using 1, 2 and 4 processes compared to a configuration with one processor. Datasets are ordered from smallest to largest based on the number of non-zero values.

**Table 2.3**
Block configurations used in experiments where we tested architectures with two or four processing units (PUs). The 2 PU configuration are only included in CPU results, because the dataset is too large for GPU memory.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Data type</th>
<th>2 PUs</th>
<th>4 PUs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetus</td>
<td>sparse</td>
<td>2 × 1</td>
<td>2 × 2</td>
</tr>
<tr>
<td>TCGA-BRCA</td>
<td>dense</td>
<td>1 × 2</td>
<td>1 × 4</td>
</tr>
<tr>
<td>E-TABM-185</td>
<td>dense</td>
<td>2 × 1</td>
<td>4 × 1</td>
</tr>
<tr>
<td>Retina</td>
<td>dense</td>
<td>2 × 1</td>
<td>2 × 2</td>
</tr>
<tr>
<td>Cochlea</td>
<td>dense</td>
<td>2 × 1</td>
<td>2 × 2</td>
</tr>
<tr>
<td>TCGA-Methyl</td>
<td>dense</td>
<td>1 × 2</td>
<td>1 × 4</td>
</tr>
</tbody>
</table>
Efficiency of parallel implementation depends on dataset shape and on chosen block configuration. Measurements illustrating this dependency are shown in Fig 2.8 for multi-processor architectures and in Fig 2.9 for multi-GPU architectures.

Figure 2.7
Computational speedups on multi-GPU architectures. Speedup on 1-, 2- and 4-GPU devices is compared to a single-processor configuration. Datasets are ordered from smallest to largest based on the number of non-zero values.

Figure 2.8
Efficiency of multi-processor architectures for different block configurations. Efficiency is represented by the fraction of linear speedup. The number of parallel processes is equal to the number of partitions.
One bottleneck of GPU-based architectures is communication overhead that occurs when copying data between GPU devices. This overhead was also observed in our experiments. For example, up to 50% of time needed to factorize TCGA-BRCA dataset in a 4-GPU environment was spent for communication. On larger datasets, however, this overhead was less pronounced. A detailed analysis is provided in Fig 2.13. The communication overhead on multi-processor architectures is negligible as shown in Fig 2.12.

We also studied algorithm scalability with respect to factorization rank. Fig 2.11 shows runtime of one iteration as a function of factorization rank value on a four-GPU architecture using a $2 \times 2$-block configuration. Fig 2.10 shows the results on a four-processor architecture.

Using matrix partitioning approach presented in Algorithm 1, we can increase speedup on sparse datasets that have imbalanced distribution of nonzero elements. The approach adapts matrix block size based on the number of nonzero elements. In Fig 2.21 we show factorization speedup attributed to the adaptive nature of Algorithm 1, and compared to non-adaptive partitioning of data matrix into equally sized blocks. We observe a speedup of up to 1.4-times on multi-processor architecture, and up to 1.2-times on multi-GPU architecture.
2.5 Discussion

2.5.1 Speedup

Speedup on GPU-architectures is substantial, and pronounced with the dataset size and number of GPUs. For example, factorization on a retina dataset was 150-times faster than that on a single processor. Datasets in Fig 2.7 are ordered by their number of nonzero elements, and we can observe a steady increase in speedup. Similar trends can also be observed on multi-processor architectures (Fig 2.6), but the speedups are substantially lower than those on the GPUs.

For TCGA-Methyl dataset, the complete data matrix occupies about 19 GBytes of GPU’s memory (Table 2.4). With a 12 GBytes of total memory on each GPU, and considering the overhead of libraries and temporary data matrices for inter-GPU communication, the data does not fit to the working memory in $1 \times 1$ and $1 \times 2$ block configurations. Running the factorization with $1 \times 4$ block configurations on 1-GPU or 2-GPU is feasible, but due to insufficient memory to store all necessary blocks in a single GPU requires a transfer of data between main memory and GPUs which severely impacts the runtime and prohibits any speedup. On this large dataset, a configuration with 4-GPUs has sufficient memory and provides for excellent speed-up (Fig 2.7). This case also demonstrates that for large datasets the proposed approach requires setups with the adequate number of GPUs that can keep all the data in working GPU memory.

2.5.2 Efficiency effects of block configuration

Block configuration plays a significant role in minimizing the impact of data transfers and balancing the load across devices (Figures 2.8 and 2.9). Wide datasets (E-TABM-185, TCGA-BRCA, TCGA-Methyl) favor column-wise partitioning ($1 \times 2$ and $1 \times 4$). Row-wise partitioning (e.g., $2 \times 1$ and $4 \times 1$) would have been more suitable for tall datasets. The effect of the block configuration on efficiency was most pronounced in TCGA-BRCA dataset, because it has very disproportional shape. This observation highlights that suitable block configuration is data dependent, and also indicates that the selection of block configuration can be automated.

The drop in efficiency under a particular choice of block configuration can be explained by increased communication overhead (Fig 2.13). As we increase the number of devices that run in parallel, we need to perform additional data transfers that are not needed on setups with one matrix block. For example, in the case of wide dataset TCGA-
BRCA and row-wise partitioning (4 × 1), over half of factorization runtime was spent for data transfer between GPUs.

2.5.3 Factorization rank

Next, we evaluate the performance of our approach when varying the value of the matrix factorization rank. Factorization rank is a vital parameter of all matrix factorization methods because it determines the number of latent vectors. A larger factorization rank means the inferred latent model has a larger degree of freedom and can thus better approximate the input data matrix [70]. However, increasing factorization rank demands more computational resources and can result in poorer generalization performance [71]. Instead of determining the optimal factorization rank for a given dataset, our goal here is to investigate how the scalability of the proposed block-wise matrix factorization algorithm depends on the value of the factorization rank and on the sparsity of the input data matrix.

Fig 2.10 shows the iteration time of NMTF as a function of factorization rank on a 4-processor architecture. We can observe that by increasing the factorization rank, the time of iteration increases linearly. For this analysis, both parameters $k_1$ and $k_2$, were set with equal values and shown as a single factorization rank parameter. Partitioning was done according to Table 2.3. When using a single process, the iteration time is proportionally slower according to the speedup shown in Fig 2.6.

Fig 2.11 shows results that correspond to iteration time on 4-GPU architecture. We can see step-wise increases in iteration time, which is a result of the way the multiplication kernel utilizes the physical resources of the GPU [72]. The multiplication on the GPU is done on tiles of data which are processed by several threads in parallel. If the matrix shape is not aligned to the tile size, the border tiles will not make full use of the resources [73].

When comparing the factorization time of a sparse dataset (Fetus) and dense datasets (Retina, Cochlea) of similar size, the benefits of using sparse data structure are substantial. On a GPU, the factorization time on a sparse dataset (Fetus) is slower than on comparable dense datasets (Retina, Cochlea). This is because multiplication with sparse structures requires slower non-sequential memory access [74]. Note that the points on Figures 2.10 and 2.11 are connected for better visualization.
2.5.4 Communication overhead

The main reason for the drop in efficiency of multi-GPU environments is communication overhead. The comparison of efficiency (darker colors) compared to the efficiency that would be achieved without data transfers (bright colors) for non-orthogonal NMTF is shown on Figures 2.12 for multi-processor and 2.13 for multi-GPU architectures. We can see that the communication overhead has almost no impact on multi-processor architecture since the different processor cores share the same global
memory. However, the impact of communication on multi-GPU architectures is substantial, because it requires transferring the data between GPU devices. Larger datasets that are shown towards the right side require more computation, which reduces the impact of communication overhead on efficiency.

![Figure 2.12](image)

**Figure 2.12** Communication overhead in multi-processor environment for non-orthogonal NMTF. Impact of communication on efficiency is shown with bright colors.

### 2.5.5 Scalability

We study the effect of data size on efficiency. As we increase the number of processors or GPU devices, the efficiency decreases due to additional communication overhead. However, as we increase the data size, the efficiency increases. Scalability describes how fast the data needs grow so the system maintains the same efficiency at higher parallelization degree.

We have generated a series of square $n \times n$ datasets, where each dimension is $n \in \{5000, 10000, \ldots, 55000\}$. The number of iterations is set to 100 and factorization rank is set to $k = 20$. Fig 2.14 shows the increase in efficiency with respect to dataset size, where each line shows the efficiency for a number of processes $p \in \{1, 2, 4, 8, 16\}$. Data size is shown in millions of entries.

Figures 2.14 and 2.12 show that on two and four processes, the efficiency can be higher than one, meaning that the speedup on two processes will be more than double com-
pared to a single process. Because two processes work on isolated partitions of the data, greater portion of it will fit into cache memory. As we increase the data size the efficiency increases as the percentage of communication overhead drops due to lower time complexity.

Fig 2.15 shows the efficiency with relation to the dataset size. Dataset with dimensions $60,000 \times 60,000$ exceeds the 12 gigabyte memory limit of a single GPU, therefore it is not possible to compute the efficiency on GPU over this limit. We use single precision for each data entry, which occupies 4 bytes of memory space.

Fig 2.16 shows the scalability on multi-processor architecture. The horizontal line shows the efficiency thresholds $E_{CPU-p} \in \{0.6, 0.8, 0.9, 0.95\}$, where $p$ is number of parallel processes. For each threshold we find the minimum data size at which point the efficiency is higher than this threshold. The results for higher parallelization degrees and higher efficiency thresholds are missing, because the system never reached that level of efficiency. The system has lower than linear scalability. For example, to increase the number of processes from 4 to 8 and maintain the efficiency 0.9, we need to increase the data size by more than five times.

Fig 2.17 shows a similar experiment on multi-GPU architecture. Here, the horizontal axis shows efficiency thresholds $E_{GPU-p} \in \{0.6, 0.8, 0.9, 0.95\}$ where $p$ is the number
Figure 2.14
Efficiency depending on dataset size. Each line shows the function of efficiency on one of the 1, 2, 4, 8 or 16-processor architectures.

Figure 2.15
Efficiency depending on dataset size. Lines show the efficiency on 1-, 2-, 3- and 4-GPU architecture.

of GPU devices. We can see that the system on GPU has substantially better scalability compared to multi-processor counterpart. However, we the increase in efficiency is much easier to achieve because the starting point is much lower on multi-GPU architecture.
2.5.6 Memory requirements

Table 2.4 shows an example of CPU and GPU usage on setup with four processing units. The memory requirements are proportional to the data size, in particular, each value needs 4 bytes of memory; hence the total memory usage can be approximated as four times the size of the data. Note that in addition to the reported numbers, GPU implementation requires approximately 300MB per GPU for software libraries. When
running on multiple GPUs, data is distributed, and requirements per device are proportionally smaller.

Table 2.4
Total CPU and GPU memory usage in megabytes (MB).

<table>
<thead>
<tr>
<th>Dataset</th>
<th>4-CPU</th>
<th>4-GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetus</td>
<td>290</td>
<td>505</td>
</tr>
<tr>
<td>TCGA-BRCA</td>
<td>298</td>
<td>300</td>
</tr>
<tr>
<td>E-TABM-185</td>
<td>509</td>
<td>515</td>
</tr>
<tr>
<td>Retina</td>
<td>2595</td>
<td>2566</td>
</tr>
<tr>
<td>Cochlea</td>
<td>2595</td>
<td>2566</td>
</tr>
<tr>
<td>TCGA-Methyl</td>
<td>18937</td>
<td>18949</td>
</tr>
</tbody>
</table>

When selecting the block size, we must consider smaller memory space on GPU especially for larger problems. Ideally, the number of blocks is the same as the number of parallel devices, but if the data is too big, some blocks need to be exchanged between CPU and GPU memory, which has significant drawbacks in efficiency. If the data is sparse, then block-wise partitioning can be used to balance the number of nonzero values and consequently computational load across devices.

2.5.7 *Speedups of sparse data format*

The data is often composed of many zero entries. Specialized data structure that only stores non-zero entries can compress the data and substantially reduce the required memory space and often leading to faster computation speed compared to dense data format. As we increase the density, the speedup of dense data format compared to sparse increases. At some point it is faster to use the dense format, however that point depends on the underlying hardware, implementation data size.

We perform a series of experiments on square datasets with dimensions $n$ in \{2000, 4000, 8000, 16000\}. We select the density from the following set of values \{0.001, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.3\}. We use compressed sparse row format (CSR), which is optimized for matrix multiplications. Fig 2.18 shows the speedup of dense format compared to sparse. We observe that for densities below 15% it is beneficial to use sparse data format on processor implementations.
Fig 2.18 shows the speedup of dense data format compared to sparse data format on CPU architecture. Dense format is faster where the colored line is over the threshold, shown with black line.

Fig 2.19 shows the speedup of dense data format compared to sparse data format on 1-GPU architecture. GPU devices are optimized for dense matrix multiplications and is often much faster compared to the sparse format with exception of datasets with very low density (lower that 1%). Hence, the most important benefit of sparse data format on GPU is lower memory consumption on sparse datasets that would otherwise not run on GPU device because of limited memory.

Figure 2.18
Speedup with of dense compared to sparse data type on CPU architecture. Dense format is faster where the colored line is over the threshold, shown with black line.

Figure 2.19
Speedup with of dense compared to sparse data type on GPU architecture. Dense format is faster where the colored line is over the threshold, shown with black line.
On sparse datasets, the partitioning can be done either by splitting the data into blocks of equal size or with regards to the number of nonzero elements. Using sparse data type, the computational complexity depends on the number of nonzero elements. Balancing the number of non-zero elements in each block better distributes the workload across different devices.

With the proposed balanced partitioning algorithm we can increase the speedup up to 1.4 times on multi-processor architectures, as shown in Figures 2.20 for non-orthogonal NMTF. Similarly, we can observe speedup of up to 1.2 times on multi-GPU architectures, shown on Figures 2.21 for non-orthogonal NMTF model. These observations suggest that the GPU architecture is more robust to the distribution of non-zero elements in the data.

2.5.8 Interpretation of factorization results

Matrix factorization methods can be used to gain a better understanding of the data and their relationships as the methods identify cluster structures and detect potential new associations. The latent factors learned by NMTF reveal clusters in each of the two dimensions of the input data (matrices $U$ and $V$) and encode cluster interactions (matrix $S$). The analysis of the latent factors can then lead to data interpretation, cluster discovery, and to prediction of new interactions.

We here demonstrate that tri-factorization can lead to the reconstruction of biologically meaningful interactions. We have used a DNA methylation dataset (TCGA-Methyl,
Table 2.1) consisting of 10,181 tissue samples from 33 cancer types. Tissue samples are profiled using methylation beta values for 485,577 CpG sites of the DNA. From these, we have considered only the sites that are related to 567 genes with known cancer interactions as listed in the Sanger cancer catalog [75]. Of those, 491 genes were included in our dataset and altogether involved 14,299 methylation sites. The resulting matrix had 10,181 rows and 14,299 columns. We factorized the matrix using factorization ranks $k_1 = 25, k_2 = 30$, which yielded an optimal data compression with respect to the accuracy evaluated on a validation dataset, as shown in Fig 2.22.

Table 2.5 lists five resulting cluster pairs that relate clusters of genes (from matrix $V$) and clusters of cancer types (from matrix $U$) with highest interaction scores in matrix $S$. First, we note that factorization revealed related cancer types, with, for example, colon, stomach and rectum adenocarcinoma (Table 2.5, first row) forming its own group. Also, we found several common Gene Ontology annotations for the clustered genes (Table 2.6). Most importantly, we found evidence in published literature for a majority of interactions between genes and cancer types inferred through matrix tri-factorization. For example, $GATA2$ was suggested as a prospective indicator for poor prognosis in patients with colorectal cancer [76], and $FAT4$ functions as a tumor suppressor for stomach cancer [77]. Other supporting publications are listed in Table 2.5.

Transcriptional silencing by DNA methylation plays an important role in the onset of cancer [84, 85]. It is thus encouraging that some of the critical interactions between
methylated genes and diseases can be inferred, as demonstrated by this analysis, by non-negative matrix factorization of methylation cancer data alone.
Table 2.6
Common GO terms for each gene group in Table 2.5.

<table>
<thead>
<tr>
<th>Gene cluster</th>
<th>Common GO term</th>
</tr>
</thead>
<tbody>
<tr>
<td>GATA2, FAT4</td>
<td>protein binding</td>
</tr>
<tr>
<td>CXCR4, BIRC3, WWTR1</td>
<td>cytoplasm, protein binding</td>
</tr>
<tr>
<td>CCND2, FEV, WT1</td>
<td>nucleus</td>
</tr>
<tr>
<td>HLA-A, HOXA9, VHL</td>
<td>protein binding</td>
</tr>
</tbody>
</table>

2.6 Conclusion

Non-negative matrix tri-factorization is a successful modeling approach that can reveal hidden patterns in biomedical datasets. Current serial factorization approaches require substantial runtime, particularly for larger datasets. We proposed a block-wise approach to speed up matrix tri-factorization through parallel execution. Experiments show the approach easily scales to very large datasets, and can achieve speedups of up to two orders of magnitude on current GPU-based architectures.
Convergence of non-negative matrix tri-factorization
Non-negative matrix tri-factorization (NMTF) is a general technique that takes a data matrix and compresses, or embeds, the matrix into a compact latent space. The learned embedding space can be used to identify clusters [17, 18], reveal interesting patterns [86, 87], and generate feature representations for downstream analytics [13, 88]. NMTF has been used to discover disease-disease associations [14], identify cancer driver genes from patient data [89], and to model topics in text data [19]. However, despite numerous applications, training NMTF models on large datasets can be slow and has remained computationally challenging [90].

Currently, NMTF learns a representation of a dataset through an optimization procedure that typically uses multiplicative updates rules, which suffer from slow convergence. We here perform an empirical study involving six large datasets comparing multiplicative update rules including three alternative optimization methods: alternating least squares, projected gradients, and coordinate descent.

First, projected gradient method uses a step-size parameter to maximize the learning rate without compromising non-negative constraints on latent matrices in NMTF [25]. Second, coordinate descent method uses partial computation result of latent matrices to successively adjust the update step, decomposing the update of latent matrices into a series of coordinate-specific, or latent factor-specific, updates. Third, alternating least squares method alternates between updating one latent matrix while fixing the other two [91]. The success of these three methods for various tasks in machine learning [28–30] encouraged us to adapt them for non-negative matrix tri-factorization.

We find that methods based on projected gradients and coordinate descent converge up to twenty-four times faster than multiplicative update rules. Furthermore, alternating least squares method can quickly train NMTF models on sparse datasets but often fails on dense datasets. Coordinate descent-based NMTF converges up to sixteen times faster compared to well-established methods.
3.1 Related work

Non-negative matrix tri-factorization (NMTF) aims to represent the data $X \in \mathbb{R}^{n \times m}$ with a product of three non-negative latent matrices $U \in \mathbb{R}^{n \times k_1}$, $S \in \mathbb{R}^{k_1 \times k_2}$, and $V \in \mathbb{R}^{m \times k_2}$ [17]. Here, parameters $k_1$ and $k_2$ represent factorization ranks and describe the number of latent vectors that form the row and column space, respectively. Matrix factorization can reduce dimensionality and noise of the original input data matrix $X$ [92], and provide an understanding of the latent structure present in the data. In contrast to classic non-negative matrix factorization [1], which decomposes the input matrix into two latent matrices, NMTF decomposes the input matrix into three latent matrices. Here, latent matrix $U$ approximates the row vector space of $X$ with a $k_1$-dimensional vector space. Similarly, $V$ describes a column-space with $k_2$ vectors, and $S$ describes interactions between the two latent vector spaces.

A distinguishing property of non-negative matrix tri-factorization is that it factorizes a given data matrix and represents it with a product of three non-negative low-dimensional matrices, often called latent matrices [93] (Fig 3.1). The product gives a reconstruction of the original data matrix. The reconstructed matrix has all of its elements completed, which can be leveraged for prediction. The goal of training an NMTF model is to find the latent matrices that produce a high-quality reconstruction of the input matrix [16]. While these latent matrices are key to matrix tri-factorization, finding the factorization of a given matrix is an NP-hard problem [94]. We thus use optimization methods to find latent matrices that approximately factorize the matrix.

A traditional approach uses multiplicative update rules [17], a method, which iteratively revises latent matrices to minimize the approximation error. Such an iterative update involves multiplying the current approximation with the gradient of the objective function, which captures the discrepancy between the input data matrix and its latent-based reconstruction. Several studies improved the performance of multiplicative update rules, for example, by using parallelization [31, 32]. A significant limitation of multiplicative update rules is that the method is slow to converge [90]. For this reason, classic non-negative matrix factorization [1] has been studied using alternative training algorithms, including alternating least squares [26, 91], projected gradients [25, 27], and coordinate descent [26]; however, these methods have not been investigated for non-negative matrix tri-factorization.
Figure 3.1
Non-negative matrix tri-factorization (NMTF). An input $5 \times 4$ sparse data matrix (left) is approximated by a product of three non-negative low-dimensional latent matrices ($U$, $S$, and $V$).

$$\begin{array}{c|c|c|c}
\text{Input} & U & S & V \\
\hline
2 & 2 & 0.5 & 0.0 & 0.8 & 0.9 & 1.8 & 0.1 \\
3 & 5 & 2 & 3 & 0.2 & 2.8 & 0.1 & 1.3 & 1.0 \\
3 & 3 & 14 & 0.1 & 11 & 0.1 & 1.0 & 16 & 0.0 \\
1 & 1 & 0.2 & 0.1 & 0.2 & 0.4 & 1.3 & 0.0 & 0.1 \\
\end{array}$$

$\approx$ Reconstruction

$$\begin{array}{c|c|c}
\text{Input} & U & S \\
\hline
2 & 2 & 0.5 & 0.0 & 0.8 & 0.9 & 1.8 & 0.1 \\
3 & 5 & 2 & 3 & 0.2 & 2.8 & 0.1 & 1.3 & 1.0 \\
3 & 3 & 14 & 0.1 & 11 & 0.1 & 1.0 & 16 & 0.0 \\
1 & 1 & 0.2 & 0.1 & 0.2 & 0.4 & 1.3 & 0.0 & 0.1 \\
\end{array}$$

$\approx$ Reconstruction

### 3.2 Methods

We develop optimization algorithms, which optimize an objective function that consists of the reconstruction error and does not include any additional constraints or regularization terms other than non-negativity of latent matrices.

#### 3.2.1 Multiplicative update rules for NMTF

The objective function of NMTF is non-convex; however when we fix all but one latent matrix, the function becomes convex [17]. Minimization of the objective function with respect to each of the three latent matrices $U$, $V$, and $S$, allows the algorithm to converge to a local stationary point [90]. Multiplicative update rules start by initializing latent matrices with random values and then iteratively update the matrices in the direction of the gradient until convergence. Convergence criteria is often measured as difference in the value of the following objective function between two or more successive iterations of the algorithm:

$$D_{\text{Fro}}(X||USV^T) = ||X - USV^T||^2_{\text{Fro}}.$$  \hfill (3.1)

Next, we give a summary derivation of existing multiplicative update rules [16]. Karush Kuhn-Tucker condition takes the partial derivative of $U$ and calculates the updated $U$ matrix at $i$-th row and $k$-th column. The resulting update rule for $U$ is as follows:

$$U \leftarrow U \odot (XVS^T \odot USV^T VS^T),$$  \hfill (3.2)
where symbol $\odot$ denotes Hadamard product and symbol $\oslash$ denotes Hadamard division. Similarly, the update rule for $V$ is derived:

$$V \leftarrow V \odot (X^TUS \odot VS^TUS^T).$$

(3.3)

Finally, to obtain the update rule for latent matrix $S$, we take derivative of the objective function with respect to $S$ and use the Karush Kuhn-Tucker conditions for $S$. This procedure gives the following update rule for $S$:

$$S \leftarrow S \odot (U^T XV \odot U^T USV^T V).$$

(3.4)

Full derivations are shown shown in A.2.1.

### 3.2.2 Alternating least squares for NMTF

Alternating least squares method [95] iteratively updates the latent matrices and each update involves solving a least-squares problem. Here, we obtain the update rules by deriving the objective function in Eq. 3.1 for each latent matrix and then enforcing non-negativity on the latent matrix using a heuristic. This derivation procedure gives the following update rules:

$$U \leftarrow \left[ (XVS^T)(SV^T VS^T)^{-1} \right]_+, \quad V \leftarrow \left[ (X^T US)(S^T U^T US)^{-1} \right]_+,$$

$$S \leftarrow \left[ (U^T U)^{-1}(U^T XV)(V^T V)^{-1} \right]_+,$$

(3.5)

where $[A]_+$ is projection to non-negative space, calculated as $A_{ij} = 0$ if $A_{ij} < 0$ else $A_{ij}$. The full derivation is given in A.2.2. Alternating least squares approach is equivalent to the second-order quasi-Newton approach [18]. The derivation of quasi-Newton NMTF and its equivalence to alternating least squares is given in A.2.5. Efficient implementations of alternating least squares method is as fast as multiplicative update rules but has unstable convergence. This is because alternating least squares method transforms current approximation of the latent matrices into non-negative matrices by simply replacing all negative values with zero values [91].

### 3.2.3 Projected gradients for NMTF

Optimization of matrix factorization models that use gradient descent [1] repeatedly apply additive updates to model parameters in the direction specified by the gradient of the
objective function and using a particular step size. The selection of the step size is not trivial [96]. When using a large fixed step size, we risk accidentally increasing the value of objective function. When the step size is too small, it can significantly slow down the convergence speed.

Projected gradients method is a gradient-based optimization method intended for solving constrained convex problems [97]. In the case of non-negative matrix tri-factorization, the method realizes the non-negativity constraints by projecting negative values in a latent matrix to a non-negative space [98]. The method is similar to multiplicative update rules. In particular, it uses an adaptive learning rate (i.e., step-size parameter) that is automatically determined in order to perform a maximum possible step in the gradient direction while staying in the non-negative space. In contrast to alternating least squares, projected gradients method is able to handle the non-negativity constraint of latent matrices in a more principled way [99]. Note that by setting the step-size parameter to 1, the update rules become equivalent to multiplicative update rule.

We derive projected gradients for NMTF and obtain the following update rule for latent matrix $U$:

$$\begin{align*}
P_u &= U - U \odot (USV^TVS^T) \odot (XVS^T), \\
\eta_u &= \frac{\sum(P_u \odot (USV^TVS^T - XVS^T))}{Tr((SV^TV)(S^TP_u^TP_u))}, \\
U &\leftarrow [U - \eta_u P_u]_+ 
\end{align*}$$

(3.6)

where $P_u$ is a projection matrix, and $\eta_u$ is step-size parameter. The update rule for latent matrix $V$ is as follows:

$$\begin{align*}
P_v &= V - V \odot (VS^TU^TUS) \odot (X^TUS), \\
\eta_v &= \frac{\sum(P_v \odot (VS^TU^TUS - X^TUS))}{Tr((SP_v^TP_v)(S^TU^TU))}, \\
V &\leftarrow [V - \eta_v P_v]_+ 
\end{align*}$$

(3.7)

where $P_v$ is a projection matrix, and $\eta_v$ is a step-size value. The update rule for latent matrix $S$ is as follows:

$$\begin{align*}
P_s &= S - S \odot (U^TU^VS^TV) \odot (U^T XV), \\
\eta_s &= \frac{\sum(P_s \odot (U^TU^VS^TV - U^T XV))}{Tr((U^TU^P_s)(V^TVP_s^T))}, \\
S &\leftarrow [S - \eta_s P_s]_+ 
\end{align*}$$

(3.8)
where $P_s$ is a projection matrix, and $\eta_s$ is a step-size value. The full derivation is given in A.2.3.

3.2.4 Coordinate descent for NMTF

Coordinate descent is an optimization method widely used in machine learning, including in support vector machines [100], and non-negative matrix factorization (NMF) [26, 101]. Coordinate descent has been proposed as an alternative approach for NMF methods, and its advantages for two-factor NMF and multiplicative updates have been already reported [102–104]. In contrast to the multiplicative and gradient-based method, which update latent matrices in a joint gradient direction, coordinate descent separately computes the gradient of each vector in each latent matrix.

Coordinate descent is a first-order method, similar to multiplicative update rules, alternating least squares, and projected gradients. While other methods use derivatives of entire latent matrices, coordinate descent computes derivatives concerning scalars or one-dimensional vectors of latent matrices and re-use partially computed results as soon as possible [105]. For example, updates to the first vector in a latent matrix are included in computing the second one, and the values from the first two vectors are then used to compute the third vector. Coordinate descent can use different ordering of vector updates, which gives rise to different variants of the method [106]: cyclic coordinate descent, stochastic coordinate descent, and greedy coordinate descent. The cyclic approach uses the same ordering of updates in each iteration of the algorithm, whereas a stochastic approach uses a random order of updates. Finally, a greedy approach [104] selects to update the vector that reduce objective function the most.

We present NMTF update rules implementing cyclic coordinate descent:

$$
\begin{align*}
    u_{i} &\leftarrow u_{i} + [(XVS^T)i - (USV^TVS^T)i] \odot (s_{i} V^T V s^T i)_{+}, \\
v_{j} &\leftarrow v_{j} + [(X^T U S)_j - (V S^T U^T U S)_j] \odot (s^T j U^T U s_j)_{+}, \\
s_{ij} &\leftarrow s_{ij} + [(U^T X V)_{ij} - (U^T U S V^T V)_{ij}] \odot (u^T i u_{i} v^T j v_{j})_{+}.
\end{align*}
$$

(3.9)

Here, $u_{i}$ represents $i$-th column of $U$, and $u_{i}$ represents $i$-th row of $U$. Update rules for $U$ and $V$ successively applied to every column in $U$ and $V$, where $s_{ij}$ update is applied to each element in latent matrix $S$. Full derivations are given in A.2.4.
3.2.5 Optimization algorithms for NMTF

Considered optimization methods use the same overall algorithmic approach shown in Algorithm 5. The main difference between these methods is the use of different update rules for latent matrices $U$, $S$, and $V$. The algorithm takes as input a data matrix $X$, and factorization rank parameters $k_1$ and $k_2$, which define the number of latent vectors for each dimension of the input matrix. Parameter $\epsilon$ defines the stopping criterion. First, the algorithm initializes latent matrices and fills them with values from uniform distribution $\mathcal{U}(0, 1)$. It then performs a series of iterations, during which it iteratively improves $U$, $V$, and $S$ using appropriate equations.

Algorithm 5 Algorithm for non-negative matrix tri-factorization of $X$ into latent matrices $U$, $S$, and $V$. MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent.

Input: Data matrix $X \in \mathbb{R}^{n \times m}$, Factorization ranks $k_1$, $k_2$ and optimization technique OPT.

1: Initialize $U^{n \times k_1} \sim \mathcal{U}(0, 1)$
2: Initialize $V^{m \times k_2} \sim \mathcal{U}(0, 1)$
3: Initialize $S^{k_1 \times k_2} \sim \mathcal{U}(0, 1)$
4: repeat
5: switch OPT do
6: case MUR
7: Update $U$, $V$, $S$ using Eqs. 3.2, 3.3, 3.4
8: case ALS
9: Update $U$, $V$, $S$ using Eqs. 3.5
10: case PG
11: Update $U$, $V$, $S$ using Eqs. 3.6, 3.7, 3.8
12: case COD
13: Update $U$, $V$, $S$ using Eqs. 3.9
14: until $U$, $V$ and $S$ converge or maximum number of iterations is exceeded
15: return $U$, $V$ and $S$
Multiplicative update rules method uses Eqs. 3.2-3.4, alternating least squares method uses Eq. 3.5, projected gradients method uses Eqs. 3.6-3.8, and coordinate descent method uses Eq. 3.9.

3.3 Data and preprocessing

We first describe the datasets and preprocessing. We then continue with a formal presentation of optimization methods, focusing on the derivation of three optimization methods that are new for non-negative matrix tri-factorization. In experiments, we considered six datasets of varying size and density, shown in Table 3.1.

Table 3.1
Datasets considered in this study. Datasets are ordered by their density.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Rows</th>
<th>Columns</th>
<th>Density (%)</th>
<th>Nonzero</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaDigits</td>
<td>1404</td>
<td>320</td>
<td>100.0</td>
<td>0.45M</td>
</tr>
<tr>
<td>Coil20</td>
<td>1440</td>
<td>16,384</td>
<td>100.0</td>
<td>23M</td>
</tr>
<tr>
<td>STRING</td>
<td>19,576</td>
<td>19,576</td>
<td>2.9</td>
<td>11.3M</td>
</tr>
<tr>
<td>MovieLens</td>
<td>69,878</td>
<td>10,677</td>
<td>1.3</td>
<td>9.7M</td>
</tr>
<tr>
<td>Mutations</td>
<td>4,790</td>
<td>25,169</td>
<td>0.8</td>
<td>1M</td>
</tr>
<tr>
<td>Newsgroups</td>
<td>18,821</td>
<td>70,066</td>
<td>0.1</td>
<td>1.4M</td>
</tr>
</tbody>
</table>

These datasets are popular benchmark datasets in the analysis of relational data and matrix factorization. (1) AlphaDigits [107] is a binary dataset of 1404 hand-drawn images of numbers and letters with dimensions of 16x20. (2) Coil20 [108] is a dataset of 1440 images each of size 128x128. Images from both datasets were flattened into a single 16,384-column vector and each pixel is represented with a value in range 0-255. (3) Mutations [109] contains a sparse binary matrix of almost five thousand patient samples with 19 different types of tumors and somatic mutations in 25 thousand genes. (4) MovieLens [110] is a sparse dataset of 10 million ratings given to ten thousand movies from 70 thousand different users. Each rating is represented by a discrete value between 0 and 5. (5) Newsgroups [111] is a real-valued sparse document-term dataset containing over 10 thousand documents with 73 thousand terms. Stop words are removed from the text and TF-IDF is used to generate feature vectors. (6) Finally, STRING dataset [112]
contains binary and undirected protein-protein interaction network for Homo Sapiens, which we obtain from the STRING database.

### 3.4 Results

We empirically study the convergence of the algorithms on six datasets of varying size and density. We find that traditional multiplicative update rules method has the worst performance. In contrast, coordinate descent converges 5 to 24 times faster than multiplicative update rules (Table 3.2) and up to 16 times faster when comparing the runtime (Table 3.3). Multiplicative update rules method outperforms alternating least squares on dense datasets, whereas alternating least squares achieves most promising results on sparse datasets.

#### 3.4.1 Experimental setup

We quantify convergence of an NMTF optimization algorithm by recording the number of algorithm iterations and the optimization runtime. We run each NMTF optimization algorithm until the relative difference of approximation error between two successive iterations is below a user-specified threshold. In particular, in iteration $i$, we calculate the value of objective function $D_i$, which is defined as the Frobenius distance between input data matrix $X$ and its approximation $X' = USV^T$ [113]:

$$D_i = \|X - X'\|_F^2 = \|X - USV^T\|_F^2,$$

(3.10)

where $U$, $V$, $S$ are the latent matrices returned in $i$-th iteration of the algorithm. Optimization is then terminated when the relative difference in objective function becomes sufficiently small [49]:

$$|D_{i+1} - D_i| / D_i < \varepsilon,$$

(3.11)

where $\varepsilon = 10^{-6}$ is used in our experiments. Optimization method that needs fewer iterations to satisfy this stopping criterion is considered to represent a faster NMTF training algorithm under the assumption that the amount of computation required to execute one iteration is similar across different optimization methods. To avoid this assumption, we also measured the optimization runtime, i.e., the total amount of computation time needed to train the NMTF model until convergence.

We also qualitatively check convergence of NMTF training by tracing the value of the objective function (Fig 3.2) and we mark the training as diverging if the objective
function oscillates or is at convergence point substantially higher than those of other optimization methods. In our experiments, we observed that alternating least squares method diverged on dense datasets. If the algorithm does not converge within a maximum number of iterations ($n_{\text{STOP}} = 50,000$), the optimization is terminated. If the algorithm does not reach the stopping criterion in $n_{\text{STOP}}$ iteration, its results are excluded from reporting to avoid potential bias in results caused by selection of $n_{\text{STOP}}$ parameter. Finally, in the case of multiplicative update rules methods, convergence in early iterations of training algorithm can be slow, which can accidentally trigger the stopping criterion. To address this issue, we additionally specify a minimum number of iterations ($n_{\text{START}} = 100$).

Non-negative matrix tri-factorization has two parameters, $k_1$ and $k_2$, that determine the size of latent matrices. We set these parameters to 20 in our analysis of convergence and we vary them ($k_1 = k_2; k_i \in \{10, 20, ..., 100\}$) in order to study the impact of factorization rank on optimization runtime. We repeat all our experiments ten times and initialize latent matrices to values between 0 and 1 that are sampled uniformly at random [114].

### 3.4.2 Convergence of NMTF optimization methods

Table 3.2 and Fig 3.2 show convergence of four NMTF optimization methods across six datasets. Table 3.2 reports the number of iterations needed by each optimization method to converge, averaged across ten independent runs of the method and omitting the runs in which the method does not converge. We see that alternating least squares and coordinate descent converge fastest and have a clear advantage over multiplicative update rules, a traditional NMTF optimization method. Additionally, our results suggest that coordinate descent might be most suitable for dense datasets, whereas alternating least squares method has poor convergence on dense datasets. Overall, considering optimization traces in Fig 3.2, coordinate descent converges fast and does not suffer from unstable training, which hampers alternative least squares. These results indicate that multiplicative update rules, which is the default NMTF optimization method in many applications, perform substantially worse than alternative optimization methods described in the present study.
Table 3.2
Number of iterations needed by NMTF training algorithms to converge (Eq. 3.11, \( \epsilon = 10^{-6} \)). Symbol \( \infty \) denotes no convergence. MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent. The MUR/COD column shows a speed-up of coordinate descent relative to multiplicative update rules, i.e., the number of iterations of MUR divided by the number of iterations of COD.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Sparsity</th>
<th>MUR</th>
<th>ALS</th>
<th>PG</th>
<th>COD</th>
<th>MUR/COD</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaDigit</td>
<td>dense</td>
<td>3641</td>
<td>( \infty )</td>
<td>1444</td>
<td>332</td>
<td>10.97</td>
</tr>
<tr>
<td>Coil20</td>
<td>dense</td>
<td>13598</td>
<td>( \infty )</td>
<td>6348</td>
<td>566</td>
<td>24.03</td>
</tr>
<tr>
<td>STRING</td>
<td>sparse</td>
<td>1516</td>
<td>67</td>
<td>579</td>
<td>114</td>
<td>13.30</td>
</tr>
<tr>
<td>MovieLens</td>
<td>sparse</td>
<td>2165</td>
<td>319</td>
<td>1029</td>
<td>154</td>
<td>14.06</td>
</tr>
<tr>
<td>Mutations</td>
<td>sparse</td>
<td>1293</td>
<td>86</td>
<td>486</td>
<td>149</td>
<td>8.68</td>
</tr>
<tr>
<td>Newsgroups</td>
<td>sparse</td>
<td>432</td>
<td>70</td>
<td>148</td>
<td>86</td>
<td>5.02</td>
</tr>
</tbody>
</table>

3.4.3 Analysis of matrix tri-factorization runtime

So far, we investigated convergence of NMTF optimization methods by studying the number of iterations needed by each method to converge. However, comparing methods solely based on the number of algorithm iterations is sufficient only if all methods perform an equal number of computations in each iteration. That is not true when training NMTF models (see Materials and methods). In particular, computational complexity of a single iteration of the algorithm varies substantially across optimization methods. It is thus essential to investigate and compare different methods by studying their optimization runtime.

Table 3.3 shows optimization runtime of four NMTF optimization methods. Results are qualitatively consistent with results in Table 3.2. Specifically, we find that coordinate descent excels on dense datasets, whereas alternating least squares method is the fastest method on sparse datasets.

3.4.4 Impact of factorization rank on optimization runtime

Factorization rank is a crucial parameter of non-negative matrix tri-factorization (see Experimental setup) as it determines the size of latent matrices and, indirectly, the learning capacity of a factorized model. By increasing the number of latent vectors, i.e., increasing
Table 3.3
Runtime of NMTF training algorithms. Shown is time in seconds until convergence of each optimization method, averaged across ten independent runs of the method. Runs that did not converge are excluded from reporting. MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent. The MUR/COD column shows a speed-up of coordinate descent relative to multiplicative update rules, i.e., the runtime of MUR divided by the runtime of COD.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>MUR</th>
<th>ALS</th>
<th>PG</th>
<th>COD</th>
<th>MUR/COD</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaDigit</td>
<td>7.1</td>
<td>∞</td>
<td>4.5</td>
<td>1.8</td>
<td>4.0</td>
</tr>
<tr>
<td>Coil20</td>
<td>295.4</td>
<td>∞</td>
<td>170.8</td>
<td>18.0</td>
<td>16.4</td>
</tr>
<tr>
<td>STRING</td>
<td>236.0</td>
<td>10.1</td>
<td>92.5</td>
<td>19.8</td>
<td>11.9</td>
</tr>
<tr>
<td>MovieLens</td>
<td>839.6</td>
<td>106.6</td>
<td>349.6</td>
<td>51.7</td>
<td>16.2</td>
</tr>
<tr>
<td>Mutations</td>
<td>67.5</td>
<td>4.5</td>
<td>29.5</td>
<td>10.5</td>
<td>6.4</td>
</tr>
<tr>
<td>Newsgroups</td>
<td>39.4</td>
<td>6.7</td>
<td>15.9</td>
<td>11.4</td>
<td>3.5</td>
</tr>
</tbody>
</table>

the values of $k_1$ and $k_2$, we can typically reduce the approximation error $D_i$ (Eq. 3.10); however larger factorization rank increases the runtime.

We studied how an increase in factorization rank affects the runtime of each of four NMTF optimization algorithms. Results in Fig 3.3 indicate that the runtime of multiplicative update rules and projected gradients increase much faster than the runtime for coordinate descent. Thus, we conclude that coordinate descent method might be the preferred optimization method in applications when large factorization rank is needed.

By increasing factorization rank, more latent vectors are added to the model. Larger factorization rank can lead to overfitting and with it to poorer generalization and can potentially affect performance on held-out data. To study the effects of factorization rank on objective value, we have varied the factorization rank in the range $k \in \{10, 20, ..., 100\}$ and assessed the objective value on the held-out data transformed into the same latent space.

We split the data $X$ row-wise into two parts: $X_{train}$ is composed of the first 80 percent and $X_{new}$ composed of the remaining 20 percent of the data. Then, we run the factorization on $X_{train}$, resulting in $U_{train}$, $S$ and $V$. We transform the remaining data $X_{new}$ into the latent space defined by $S$ and $V$, such that the following objective function is
minimized:
\[ D_{\text{Fro}}(X_{\text{new}} \mid U_{\text{new}} SV^T) = \|X_{\text{new}} - U_{\text{new}} SV^T\|_F^2. \] (3.12)

We keep the S and V factors from the training step and initialize \( U_{\text{new}} \) with random values. We iteratively apply updates on \( U_{\text{new}} \), while S and V remain unchanged. Fig 3.4 shows the objective value of four optimization techniques according to Eq. 3.12 at factorization rank \( k_1, k_2 = 20 \). Each experiment is repeated ten times until the convergence criterion with parameter \( \epsilon = 6 \) is reached.

The convergence using pre-trained model is orders of magnitude faster than training from scratch. Consistent with the optimization on training data, we observe that Alternating least squares and Coordinate descent are fastest, Projected gradient is slower and Multiplicative updates is the slowest. ALS approach did not converge on Coil20 training data and was therefore not used on testing data.

Fig 3.5 shows the lowest objective value from multiple runs (solid lines). We compare the optimization function at convergence for factorization rank \( k \in \{10, 20, ..., 100\} \). We can observe that for majority of datasets, all methods converge to a similar solution (Coil20, STRING, Mutations, Newsgroups). Coordinate descent and Alternating least squares appear more sensitive to random initialization on AlphaDigit and MovieLens datasets. The difference between best and worst solution increases as we increase factorization rank. In such cases, Coordinate descent needs to be repeated a few times in order to ensure results comparable to Projected gradients and Multiplicative update rules. Alternatively, using a different initialization technique \[114\] we may overcome this drawback.

3.4.5 Stochastic mini-batch approach to NMTF

Serizel et al. \[102\] show that two-factor NMF converges faster using a stochastic mini-batch approach, where the dataset is split into blocks and updates are in each iteration performed on each individual block. We have developed stochastic mini-batch versions of each of the four presented NMTF optimization techniques.

We show the convergence of mini-batch versions together with its non-batch counterparts in Fig 3.6. While the mini-batch variants do improve the convergence speed of multiplicative updates and projected gradient, they are highly unstable, and the resulting value of the objective function is worse compared to the non-mini-batch variants. Mini-batch variants of alternating least squares and coordinate descent did not converge.
In each iteration we randomly permute the dataset into \( b \) batches such that each row is included in exactly one batch. We iterate over all batches and update the dataset using a subset of the data \( X_i \) and a subset of \( U \) factor: \( U_i \), where \( i \in \{1, 2, \ldots, b\} \). We split the data into \( b = 20 \) batches. The objective function is evaluated over the entire dataset. Each experiment was run for a maximum of 1000 iterations. Mini-batch versions of ALS (dashed yellow line) and COD (dashed red line) do not converge. Projected gradient mini-batch (dashed green line) and MUR mini-batch (dashed blue line) approaches show faster convergence at the beginning of some experiments (AlphaDigit, Coil20, MovieLens). The final solution is in all cases worse than the non-batch counterpart and large oscillations (larger than \( \epsilon = 0.01 \)) make it difficult to determine stopping criteria.

### 3.4.6 Impact of initialization on convergence

Initialization is another parameter of non-negative matrix tri-factorization and it determines the values of \( U, S, \) and \( V \) at the start of iterative optimization. The default approach is random initialization, but we also explored initialization by pre-training using a different technique. Algorithm 6 shows coordinate descent approach using multiplicative-based pre-training. We run a single iteration of multiplicative updates and then use the result as input to coordinate descent optimizer. Instead of multiplicative pre-training, projected gradient or alternating least squares can be used. Fig. 3.7 shows the convergence of four different initialization techniques. We can see that pre-training initializations lead to faster and more robust convergence compared to random initialization. Random initialization in Fig. 3.7 is equivalent to COD approach in Fig. 3.2.

### 3.5 Discussion

Currently, multiplicative update rules represent a popular off-the-shelf optimization approach for non-negative matrix tri-factorization (NMTF) that is used in diverse applications, ranging from bioinformatics to natural language processing (e.g., [13, 14, 17–19, 86–89]). We derived three new optimization methods for NMTF and demonstrated their convergence and scalability on six datasets of varying size and density. Importantly, we observe that coordinate descent, the newly derived method, converges fast and is stable on datasets of varying size and density. Our results suggest that coordinate descent might
Algorithm 6 Coordinate descent NMTF approach with pre-training initialization. We show pre-training of $U$, $S$, and $V$ based on multiplicative updates.

**Input:** Data matrix $X \in \mathbb{R}^{n \times m}$, Factorization ranks $k_1$, $k_2$ and optimization technique OPT.

1: Initialize $U^{n \times k_1} \sim \mathcal{U}(0, 1)$
2: Initialize $V^{m \times k_2} \sim \mathcal{U}(0, 1)$
3: Initialize $S^{k_1 \times k_2} \sim \mathcal{U}(0, 1)$
4: Update $U$ using Eq. 3.2
5: Update $V$ using Eq. 3.3
6: Update $S$ using Eq. 3.4
7: repeat
8: Update $U$, $V$, $S$ using Eqs. 3.9
9: until $U$, $V$ and $S$ converge or maximum number of iterations is exceeded

**return** $U$, $V$ and $S$

be a preferred off-the-shelf optimization method to train NMTF models. These findings together with complete mathematical derivations (see Appendix A.2) and a public implementation of the algorithms are our primary contributions.

Coordinate descent offers a good compromise between factorization quality and the number of iterations of the algorithm needed for convergence. We find that coordinate descent is the fastest approach that often requires fewer than 100 iterations to converge, even on large datasets. Furthermore, the final value of the NMTF objective function attained by coordinate descent is comparable to that of multiplicative update rules. One drawback of coordinate descent is a higher computational cost per iteration at larger factorization ranks. Coordinate descent also exhibits higher sensitivity to initialization of the latent matrices, as indicated by the larger span of the objective function in Fig 3.2, especially in the case of small and sparse datasets.

The alternating least squares method performs well on sparse datasets but fails to converge to a high-quality solution on dense datasets. The method is thus sensitive to the properties of the dataset and, despite its performance on sparse data, we would advise using coordinate descent as a stable off-the-shelf NMTF optimization method. We note
that the observed instabilities of alternating least squares and notable convergence issues are due to the heuristic enforcement of non-negativity in the learned latent matrices. In particular, as a final step in each iteration of the algorithm, alternating least squares method sets negative values in each latent matrix ($U, S, \text{and } V$) to zero values [91, 95]. The use of this heuristic generates non-negative latent matrices. However, the alternating least squares method cannot guarantee that the objective function value will decrease with each iteration of the algorithm, which can lead to instability of NMTF model training.

Our results suggest that multiplicative update rules method the most robust approach, as the method is not sensitive to initialization of latent matrices (Fig 3.2, see the width of the span of the NMTF objective function) and its final solution is at least as good as that of projected gradients or coordinate descent. However, multiplicative update rules method has the slowest convergence among the considered optimization methods. This finding is especially important as multiplicative update rules are currently favored NMTF optimization method. Multiplicative update rules have slow convergence during the first hundred iterations; that is, the method appears to have reached a local stationary point during which the algorithm gives no improvements and returns latent matrices of low-quality if exited prematurely. We avoid this by setting a minimum number of iteration for multiplicative updates to one hundred iterations. A good alternative to multiplicative update rules are projected gradients. Similar to multiplicative update rules, projected gradients are robust and can learn a high-quality NMTF model, however, the methods needs an order of magnitude fewer iterations than multiplicative update rules.

Sparsity is an important aspect of the data which greatly impacts the objective value. Fig 3.2 shows that the objective value on sparse datasets (STRING, MovieLens, Mutations, Newsgroups) is consistently worse compared to dense datasets (AlphaDigit, Coil20). The worst performance can be observed on the sparsest dataset (Newsgroups), where the objective value error is 0.95. Since only a small number of non-zero values are present in the data, the optimal approximation is achieved by fitting mostly to zeros and the non-zero entries will be wrongly approximated. This can be reduced by using a different objective function that masks missing entries. For example, weighted matrix factorization factorization \[ \text{constraints the model by adding greater weights to specific entries in the data.} \]
There are many interesting avenues of future work. For example, the use of heuristics could further improve performance of NMTF optimization methods [104]. Applying multiple updates to a particular latent matrix before moving on to updating the next latent matrix is a fruitful direction, as such approach could reduce the number of expensive matrix multiplications. Another idea is to use heuristics to determine the ordering of updates in the case of coordinate descent algorithm. We studied how initialization technique affects the convergence of coordinate descent NMTF. We observed in Fig 3.7 that initialization with pre-training increases the robustness of coordinate descent compared to random initialization and substantially increases the convergence rate in four out of six benchmark datasets.

Non-negative matrix tri-factorization is a core component of joint matrix factorization [13] that has been successfully used for fusion of heterogeneous data [15, 116, 117]. Such matrix factorization-based data integration can fuse many large datasets [14], however it can require substantial computational resources for inference. A speed-up of non-negative matrix tri-factorization by coordinate descent thus provides a fruitful research direction towards a computationally-effective data fusion and large-scale data integration.

3.6 Conclusion

An established approach to non-negative matrix tri-factorization is based on multiplicative update rules. We have derived three alternative non-negative matrix tri-factorization techniques based on alternating least squares, projected gradient, and coordinate descent. Comparison of convergence and runtime of these approaches on six large data sets shows that alternative approaches converge faster. The best average performance is achieved by coordinate descent.
Figure 3.2
Optimization traces for six datasets and four NMTF optimization methods. Graphs show the value of NMTF cost (objective) function \([13, 17, 93]\) at each iteration of the NMTF training algorithm. Shown is the optimization trace of the algorithm run with the smallest approximation error (solid lines). The highlighted area shows the span of the NMTF objective values across ten independent runs; each started from a different random initialization (see Experimental setup). MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent.
Figure 3.3
Impact of factorization rank on factorization time across six datasets and four NMTF optimization methods. The total runtime in seconds needed for convergence of the NMTF training algorithm is shown as a function of factorization rank \((k_1 = k_2; k_i \in \{10, 20, ..., 100\})\), averaged across ten independent runs of the algorithm. Runs that did not converge are excluded from reporting. MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent.
Figure 3.4
Graphs show convergence on testing data $X_{\text{test}}$. Solid lines show the optimization trace of initialization that results in the lowest objective value. The highlighted area shows the span of objective value across ten runs. MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent.
Figure 3.5
Reconstruction of testing data $\mathbf{X}_{\text{new}}$ with respect to factorization rank. The lowest objective function at convergence is shown as a function of factorization rank ($k_1 = k_2; k_i \in \{10, 20, ..., 100\}$). The highlighted area shows the span from lowest to highest objective value across ten repeated runs. MUR, multiplicative update rules; ALS, alternating least squares; PG, projected gradients; COD, coordinate descent.
Convergence of non-negative matrix tri-factorization

Figure 3.6
Convergence of mini-batch optimization techniques on six datasets using four NMTF optimization methods (continuous lines) and four mini-batch variants (dashed transparent lines). MUR_MB: mini-batch multiplicative updates; PG_MB: mini-batch projected gradients; COD_MB: mini-batch coordinate descent; ALS_MB: mini-batch ALS.
Figure 3.7
Convergence using coordinate descent with different initialization approaches. MUR init, PG init and ALS init all use a single iteration of one of three alternative approaches. The highlighted area shows the objective value span across ten independent runs of the algorithm.
Speedup of data fusion
Most algorithms are designed to deal with homogeneous data, where all objects are of the same type. However, in real-world applications, data is often heterogeneous, containing many types of objects that interact with each other \[118\]. To accurately model interacting data, we need to use specialized methods. For example, say we want to find clusters of genes and clusters of chemicals. Traditional clustering algorithms are designed to find one-way clusters and will not consider the interactions between clusters of genes and chemicals. Matrix tri-factorization is capable of modeling row and column latent space simultaneously, for example in co-clustering, modeling interactions between row and column clusters can improve accuracy of the model \[17\]. As we include additional object types and datasets, building a model that incorporates all object types becomes increasingly more difficult. Data fusion addresses the challenge of modeling heterogeneous data with large number of object types.

Existing data fusion methods can have excellent accuracy and operate on heterogeneous datasets with multiple object types and relation types \[11–13\]. These methods were not designed to run in highly parallel environments and the convergence of the methods is slow.

We here increase the speed of data fusion by increasing the scalability on multi-processor and GPU systems, where GPU implementation converges over 30 times faster compared to existing solution. We use block-wise partitioning on each dataset to efficiently distribute the data across devices and lower the communication requirements between different processes.

Further, we develop a coordinate descent-based data fusion approach which shows up to 17 times faster convergence compared to existing multiplicative-update based approach. We observe that total runtime until convergence of coordinate descent-based data fusion performs up to 80 times faster compared to existing approach running on 16 processes.
4.1 Related work

Data fusion methods are capable of incorporating information from multiple datasets. The ability to integrate data from multiple domains makes it superior in terms of accuracy [13], and a preferred method in biomedical data analysis [119–121]. Biomedical data is often very heterogeneous, composed of many object types, such as genes, diseases, chemicals, pathways, and patients. Many recent studies leverage the power of data fusion for gene prioritization [122], drug-cancer interaction mining [116], disease classification [14], protein function prediction [15, 123], microarray data classification [124], disease-chemical association [125], predicting protein-protein interactions [126], and modeling interactions between proteins and non-coding ribonucleic acids [127, 128].

Data fusion techniques are also successful in multi-domain recommendation systems [129], such as movie, music and book recommendations [130], predicting food choices in restaurants [131], and social network recommendations [132, 133]. Furthermore, multi-matrix factorization can be used for data imputation in multi-sensor systems [134, 135].

We can categorize data fusion methods into three main groups [136] that are shown in Fig 4.1. Stage-based methods use different datasets at different stages of data fusion, such that each dataset is analyzed using a different algorithm. For example, using geographical map data we can find locations that are physically close, and then combine them with road network data to find the fastest path.

The feature based methods learn a new representation of the original features, either by simple concatenation of features, or using more advanced methods for example generating a feature representation using deep neural networks. Because the features are converted into a different space, we are unable to interpret the hidden structure of the data. Another major drawback of deep neural networks are long runtime, where the complexity of parameter tuning further slows down the process.

Semantic meaning-based data fusion retains the structure of the data, which gives insight into each dataset. Semantic meaning data fusion can be further categorized into similarity, multi-view, probabilistic, and transfer learning sub-categories [136]. Similarity-based methods leverage the underlying similarity between different objects, examples of such methods are collective matrix factorization and collaborative filtering models. Multi-view based methods treat different datasets as different views of an object, for example multiple kernel learning algorithms [137] find an optimal linear
Collective matrix factorization was first introduced in spectral relational clustering [118]. It builds upon non-orthogonal NMTF [16] to factorize multi-type inter-related data, where one of the factors is a cluster indicator matrix. Later, tri-SPMF approach was proposed, which uses semi-supervised model and supports multi-type datasets [11]. It uses additional constraints to incorporate known information. Information on intra-type relations and data for negative relations (cannot-link constraints) is stored as regularization parameter. Symmetric non-negative matrix tri-factorization [12] does not use regularization, but rather combines intra-type with inter-type relations in a single symmetric matrix.

Recently, a matrix factorization-based data fusion method called DFMF was proposed [13]. Instead of modeling the entire data collection as a single large matrix, as
in the previous approaches [11, 12], DFMF treats the data collection as a set of smaller matrices. Instead, it uses a series of factor updates over each dataset, where certain relations between objects are missing. It is capable of modeling asymmetric relations.

### 4.1.1 Collective matrix factorization

Similarity-based relations can be described as intra-type relations and inter-type relations. Intra-type relation can describe the similarities between object of the same type, where inter-type relations contain information on interactions or similarities between two different object types. On Fig 4.2 we show an example with three object and two inter-type relations $U^{(1)} \rightarrow U^{(2)}$ and $U^{(1)} \rightarrow U^{(3)}$.

Each data matrix is composed of one or two object types, represented with a single matrix. Modeling of the single-type relational data, such as pairwise similarity matrices, can be done using symmetric NMTF [17]. Dual-type relational data can be modeled using NMTF, such that rows and columns reside in different latent space. NMTF is capable of modeling two object types simultaneously. By introducing third object type in a form of an additional dataset (for example $U^{(1)} \rightarrow U^{(3)}$), the standard NMTF approach needs to be adapted before the data between datasets can be shared. Fig 4.2 shows a representation of two datasets, where $U^{(1)}$ object type is shared.

A natural way to generalize the NMTF to multi-type data is to block-wise concatenate the datasets into a single large data matrix $X$, as shown in Eq. (4.1). $X^{(I,J)}$ represents pairwise relations between instances of the I-th and J-th object type, for each combination of $I \in \{1, 2, ..., N\}$ and $J \in \{1, 2, ..., N\}$, where $N$ is the number of object types.

$$
X = \begin{bmatrix}
X^{(1,1)} & X^{(1,2)} & \cdots & X^{(1,N)} \\
X^{(2,1)} & X^{(2,2)} & \cdots & X^{(2,N)} \\
\vdots & \vdots & \ddots & \vdots \\
X^{(N,1)} & X^{(N,2)} & \cdots & X^{(N,N)}
\end{bmatrix}
$$

Each dataset has a different internal structure and can not be represented by a single $S$ matrix, therefore running NMTF on the concatenated data $X$ can not properly model different datasets. We concatenate the set of matrices $U^{(1)}, U^{(2)}, \ldots, U^{(N)}$ into a block-diagonal matrix $U$. $S^{(I,J)}$ matrices are concatenated into a block matrix $S$: 
Figure 4.2
Factorization of two datasets containing three object types. Three object types $U^{(1)}$, $U^{(2)}$, $U^{(3)}$ are represented with individual matrices, where interactions between objects is contained in $S^{(1,2)}$ and $S^{(1,3)}$ matrices.

$$U = \begin{bmatrix} U^{(1)} & 0 & \cdots & 0 \\ 0 & U^{(2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & U^{(N)} \end{bmatrix}, \quad S = \begin{bmatrix} S^{(1,1)} & S^{(1,2)} & \cdots & S^{(1,N)} \\ S^{(2,1)} & S^{(2,2)} & \cdots & S^{(2,N)} \\ \vdots & \vdots & \ddots & \vdots \\ S^{(N,1)} & S^{(N,2)} & \cdots & S^{(N,N)} \end{bmatrix}$$  \hspace{1cm} (4.2)

After that, we minimize the following objective function

$$D_{Frø}(X||USU^T) = ||X - USU^T||_{Frø}^2,$$  \hspace{1cm} (4.3)

using a form of symmetric NMTF method which is the basis of simultaneous matrix factorization \cite{12}. Such approach is not very efficient as it includes a lot of zeros. Further, this may encourage the method to fit to the missing data to further reduce the objective function. Even though the objective function is lower, such approach can result in worse reconstruction of non-missing datasets \cite{13}.
4.2 Methods

In this section we describe the notation and present an existing data fusion approach based on multiplicative updates (DFMF) \cite{13}. Further, we present coordinate descent data fusion approach (DFCOD) and evaluate the convergence speed on two collections of datasets. We formulate block-wise versions of both presented data fusion techniques and apply them in efficient parallel implementations on systems with multiple processors and multiple GPU devices.

4.2.1 Multiplicative update-based data fusion

Recently proposed data fusion by matrix factorization (DFMF) \cite{13} tries to minimize the following objective function:

\[
D_{\text{Fro}}(X||USU^T) = \sum_{I,J \in X} ||X^{(I,J)} - U^{(I)}S^{(I,J)}U^{(J)^T}||^2_{\text{Fro}},
\]

where \(X^{(I,J)}\) represents the relational matrix between \(I\)-th object and \(J\)-th object. \(U^{(I)}\) represents \(I\)-th factor with dimensions \(n_I \times k_I\) and \(S^{(I,J)}\) represents matrix \(S\) associated with relation between \(I\)-th and \(J\)-th object. \(\sum_{I,J \in X}\) iterates only through relations that are present in the data \(X\). This approach is substantially more efficient than 4.3, because it can ignore missing relations and has a much more compact representation.

Contrary to matrix tri-factorization approach presented in the previous chapters, the middle factor \(S\) is not limited to non-negative values. Without non-negative constraint is possible to successfully apply alternating least squares approach without the instabilities that are caused by replacing negative values with zero. Alternating least squares exhibits very fast convergence as shown in the Chapter 3, but it is not suitable for cases where we must ensure non-negative values in factors. The derivations are shown in Appendix A.3.1 and the resulting \(S\) update rule for individual dataset between \(I\)-th and \(J\)-th object type is:

\[
S^{(I,J)} \leftarrow \left( U^{(I)^T}U^{(I)} \right)^{-1} U^{(I)^T}X^{(I,J)}U^{(J)} \left( U^{(J)^T}U^{(J)} \right)^{-1}.
\]

Note that DFMF and DFCOD both use this ALS-based rules for the \(S\) factor. Note the similarity between the update rule 4.5 and the update rules in section 3.2.2. We have used multiplicative and coordinate descent naming approach, to indicate, which optimization technique is used to update factor \(U\), where DFMF uses multiplicative and DFCOD uses coordinate descent based update rules for \(U\).
Iteratively, for each dataset, update $U^{(I)}$ factor. The update rule for $I$-th block of factor $U$ is as follows:

$$
E^{(I)} = \sum_{IJ} \left( X^{(I,J)} U^{(I)} S^{(I,J)T} + X^{(I,J)T} U^{(I)T} S^{(I,J)} \right),
$$

$$
D^{(I)} = \sum_{IJ} \left( U^{(I)} S^{(I,J)} U^{(I)T} S^{(I,J)T} + U^{(I)T} S^{(I,J)} U^{(I)} S^{(I,J)} \right),
$$

$$
U^{(I)} \leftarrow U^{(I)} \odot \sqrt{E^{(I)} \odot D^{(I)}}. \tag{4.6}
$$

Note that this update rule is equivalent to formulation, where $U$ is block-diagonal matrix, composed of $U^{(1)}, U^{(2)}, \ldots, U^{(N)}$. Full derivations of this update rule are shown in Appendix A.3.1.

The multiplicative data fusion algorithm is shown in Algorithm 7. Note that changes in factor $U$ include information from all datasets. Iteration over $X$ and $U$ can be performed in any order as the changes are applied at the end of iteration.

---

**Algorithm 7** Algorithm for data fusion based on multiplicative updates. $\mathcal{U}(0, 1)$ represents uniform distribution.

**Input:** Data matrix $X^{(I,J)} \in \mathbb{R}^{n \times m}$, Factorization ranks $k_1, k_2, \ldots, k_N$.

1: Initialize $U^{(I)} \sim \mathcal{U}(0, 1)$ for each $U^{(I)} \in U$

2: Initialize $S^{(I,J)} \sim \mathcal{U}(0, 1)$ for each $S^{(I,J)} \in S$

3: repeat

4: for $X^{(I,J)} \in X$ do

5: Update $S^{(I,J)}$ using Eq. 4.5

6: end for

7: for $U^{(I)} \in U$ do

8: Update $U^{(I)}$ using Eq. 4.6

9: end for

10: until $U$ and $S$ converge or maximum number of iterations is exceeded

return $U$, $S$
4.2.2 Coordinate-descent based data fusion

Coordinate descent is an optimization approach that we successfully used to accelerate convergence of matrix tri-factorization. Convergence of coordinate descent is faster compared to approaches based on multiplicative update rules or projected gradients, as well as more stable than alternating least squares approach. These findings inspired the development of coordinate descent-based collective matrix factorization, which we present in this section.

We use the same update rules for optimization of $S$ factor as multiplicative-based approach, shown in Eq. (4.5). This update rule is derived in the same way as alternating least squares (ALS) technique, which means DFCOD is in fact a hybrid approach of two fast optimization techniques, where $S$ is updated using ALS and $U$ is updated using coordinate descent. The alternating least squares approach did not appear to suffer from the same instabilities as in NMTF, because we allow negative values in the factor $S$. The values in $U$ are still required to be non-negative, therefore ALS approach will not be suitable for update of factor $U$.

In each iteration, we iterate over all available data relations $I,J \in X$ and apply the update rules for $S$. Then, we iterate over all object types in $U^{(l)}$, where $I,J \in \{1, \ldots, N\}$ and $N$ is the number of object types. We update each individual vector in $U^{(l)}$ according to the following rules. We use the following notation: $U_{i,}$ represents $i$-th column of $U$. We define coordinate descent update rule for $U$ for $i$-th column in $U$ as:

$$
U_{i}^{(l)} \leftarrow \left[ E_{i}^{(l)} \odot D_{i}^{(l)} \right]_{+},
$$

where $E_{i}^{(l)}$ denotes the following expression:

$$
E_{i}^{(l)} = (X_{iJ}^{(l)T} U_{j}^{(l)T} S_{ij}) - (U_{i}^{(l)T} S_{ij} U_{j}^{(l)T} U_{j}^{(l)T} S_{ij}) + 
+ U_{i}^{(l)T} (S_{ij} U_{j}^{(l)T})_{j} + (X_{jI}^{(l)T} U_{i}^{(l)T} U_{i}^{(l)T} S_{ij})_{j} - 
- (U_{i}^{(l)} S_{ij} U_{j}^{(l)T})_{j} + U_{i}^{(l)} (S_{ij} U_{j}^{(l)T})_{j} + (X_{jI}^{(l)T} U_{i}^{(l)T} U_{i}^{(l)T} S_{ij})_{j}.
$$

and $D_{i}^{(l)}$ denotes the normalization term:

$$
D_{i}^{(l)} = (S_{ij} U_{j}^{(l)T})_{j} + (S_{ij} U_{j}^{(l)T})_{j} + (S_{ij} U_{j}^{(l)T})_{j}.
$$

Note that we enforced non-negativity constraint on the $U$ factor. Full derivations for DFCOD rules are shown in Appendix A.3.2.
Algorithm 8 Algorithm for data fusion based on coordinate descent.

Input: Data collection $X^{(I,J)}$ for $I, J \in X$, Factorization ranks $k_1, k_2, \ldots, k_N$.

1: Initialize $U^{(I)} \in \mathbb{R}^{n_i \times k_i} \sim \mathcal{U}(0, 1)$ for each $I$ in $U$
2: Initialize $S^{(I,J)} \in \mathbb{R}^{k_i \times k_j} \sim \mathcal{U}(0, 1)$ for each $I, J$ in $X$
3: repeat
4: for $X^{(I,J)} \in X$ do
5: Update $S^{(I,J)}$ using Eq. 4.5
6: end for
7: for $U^{(I)} \in U$ do
8: for each $i$-th column in $\{0, 1, \ldots, k_i\}$ do
9: Update $U_i^{(I)}$ using Eqs. 4.7-4.9
10: end for
11: end for
12: until $U$ and $S$ converge or maximum number of iterations is exceeded
return $U, S$

4.2.3 Data partitioning

We have used block-wise parallelization on both collective matrix factorization approaches. Each of the available datasets is further sliced into $p$ blocks, where $p$ is the number of parallel devices using tall partitioning. We have already shown that tall $N \times 1$ or wide $1 \times N$ partitioning performs superior on sparse datasets (Fig. 2.20) compared to two-dimensional $N \times M$ partitioning, because we can evenly partition the data based on the number of nonzero elements. Collective matrix factorization is based on symmetric matrix factorization, meaning that both row- and column-space contain the same factor $U$. The consequence is that wide and tall partitioning are equal. Fig. 4.3 shows example partitioning, where each dataset is split into $p$ blocks, where $p$ denotes the parallelization degree.

We can see that $U$ matrix needs to be stored using two distinct block representations. First representation stores complete factor, which is used in $XU$ operation. For transposed operations, for example $X^T U$, we need to use a partitioned representation of $U$ to match the row partitioning in $X$. The synchronization between single-block and
Speedup of data fusion

Figure 4.3
Block-wise partitioning of data matrix $X$ using a 3-way parallelization. Latent factor $U$ is partitioned into three blocks, but operations with transposed data require the use of full $U$ factor.

multi-block representation of factor $U$ introduces communication overhead and has negative impact on the efficiency of parallel approaches, which we show in section 4.4.4.

4.3 Data and experimental setup

For evaluating the performance of data fusion, we considered three data collections. Toxicogenomics data collection is retrieved from Comparative Toxicogenomics Database [141]. The collection contains five object types: genes, chemicals, diseases, pathways and phenotypes, present in the following eight relations:

- gene-chemical: binary relation between genes and chemicals
- gene-disease: real-value inference score between genes and diseases
- gene-pathway: binary relation between genes and pathways
- chemical-disease: real-value inference score between chemicals and diseases
- chemical-pathway: binary relations between chemicals and enriched pathways
- disease-pathway: binary relation between disease and pathways
- chemical-GO term: binary relation between chemicals and GO enriched associations
- chemical-phenotype: binary interactions between chemicals and phenotype
Toxicogenomics dataset is sparse and has many relations that share many object types. The size and density of Toxicogenomics relations is summarized in Table 4.1. Density is defined as the number of nonzero values divided by number of all entries in the data.

Table 4.1
Size and density of Toxicogenomics data collection relations.

<table>
<thead>
<tr>
<th>Object 1</th>
<th>Object 2</th>
<th>Dimensions</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gene</td>
<td>Disease</td>
<td>47k × 7k</td>
<td>20.7%</td>
</tr>
<tr>
<td>Chemical</td>
<td>Gene</td>
<td>47k × 13k</td>
<td>0.3%</td>
</tr>
<tr>
<td>Chemical</td>
<td>Disease</td>
<td>15k × 7k</td>
<td>5.3%</td>
</tr>
<tr>
<td>Gene</td>
<td>Pathway</td>
<td>11k × 2k</td>
<td>0.5%</td>
</tr>
<tr>
<td>Chemical</td>
<td>Pathway</td>
<td>9k × 2k</td>
<td>5.7%</td>
</tr>
<tr>
<td>Disease</td>
<td>Pathway</td>
<td>5k × 2k</td>
<td>4.8%</td>
</tr>
<tr>
<td>Chemical</td>
<td>GO term</td>
<td>9k × 11k</td>
<td>4.7%</td>
</tr>
<tr>
<td>Chemical</td>
<td>Phenotype</td>
<td>7k × 4k</td>
<td>0.5%</td>
</tr>
</tbody>
</table>

Hetrec-movies is a data collection from movie recommendation domain, provided by Hetrec 2011 international workshop [142]. Data collection is composed of eight object types containing the following seven relations:

- movie-rating: real-value ratings given to movies by users
- movie-genre: binary dataset representing movie genres
- movie-actors: binary relation between movies and actors
- movie-tags: integer matrix containing movies and weighted tag assignment
- movie-directors: binary matrix of movies and directors
- movie-countries: binary relation of movies and country of origin
- movie-locations: filming locations of movies

Hetrec-movies is a very sparse data collection and contains a single shared object type: movie and are summarized in Table 4.2. Other relations are not shared between datasets.
Table 4.2
List of relations, dimensions and density of Hetrec-movies dataset.

<table>
<thead>
<tr>
<th>Object 1</th>
<th>Object 2</th>
<th>Dimensions</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movie</td>
<td>User rating</td>
<td>10k × 2k</td>
<td>0.04%</td>
</tr>
<tr>
<td>Movie</td>
<td>Genre</td>
<td>10k × 20</td>
<td>10.2%</td>
</tr>
<tr>
<td>Movie</td>
<td>Actor</td>
<td>10k × 95k</td>
<td>0.02%</td>
</tr>
<tr>
<td>Movie</td>
<td>Tag</td>
<td>7k × 5k</td>
<td>0.1%</td>
</tr>
<tr>
<td>Movie</td>
<td>Director</td>
<td>10k × 4k</td>
<td>0.02%</td>
</tr>
<tr>
<td>Movie</td>
<td>Country of origin</td>
<td>10k × 72</td>
<td>1.4%</td>
</tr>
<tr>
<td>Movie</td>
<td>Filming location</td>
<td>9k × 1k</td>
<td>0.4%</td>
</tr>
</tbody>
</table>

We also constructed a third data collection Gene-Chem-Disease, which is a subset of Toxicogenomics dataset, where matrices are converted into dense format to measuring speed on very large and dense datasets. In data fusion, this is important when if the workflow requires the missing values to be imputed. Summary of all three data collections is shown in Table 4.3.

Table 4.3
Data collections, number of relations, total number of elements, nonzero number of elements and average density.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Relations</th>
<th>Nonzero el.</th>
<th>Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toxicogenomics</td>
<td>8</td>
<td>55M</td>
<td>2.76%</td>
</tr>
<tr>
<td>Hetrec-movies</td>
<td>7</td>
<td>1.4M</td>
<td>0.13%</td>
</tr>
<tr>
<td>Gene-Chem-Disease</td>
<td>3</td>
<td>1053M</td>
<td>100%</td>
</tr>
</tbody>
</table>

4.3.1 Experimental setup

Convergence criterion is met when the difference in two consecutive iterations is less than $\epsilon = 10^{-5}$. The objective value is measured as normalized reconstruction error of all datasets in a collection as shown in Eq. (4.4).
Factorization rank for all convergence and runtime experiments was set to $k_f = 20$ for all object types. The factors are initialized using random values with predetermined seed value to ensure reproducible results.

We considered the following runtime metrics:

- **Speedup** is defined as the ratio between runtime $t_{CPU-1}$ on a single CPU, and runtime $t_{CPU-p}$ on a multi-processor architecture using $p$ processes: $s_{CPU-p} = \frac{t_{CPU-1}}{t_{CPU-p}}$. Runtime is defined as the processing time until convergence. Speedup on GPU is defined as the ratio between iteration time of single CPU and multi-GPU: $s_{GPU-p} = \frac{t_{CPU-1}}{t_{GPU-p}}$.

- **Iteration speedup** is defined as the ratio between the average iteration time of single-core and multi-core implementation. $IS_{CPU-p} = \frac{t_{CPU-1}}{t_{CPU-p}}$, where $p$ defines number of parallel processes. On multi-GPU architecture, the iteration speedup is defined as $IS_{CPU-p} = \frac{t_{CPU-1}}{t_{GPU-p}}$, where $p$ is number of GPU devices.

- **Scalability** is the ability of a system to accommodate an increased workload, in this case larger datasets. We define scalability as a function of dataset size and parallelization degree.

The experiments were run on a dual-socket Intel Xeon Silver 4110 CPU system, with 16 physical cores. Multi-processor implementations are run on 16 cores using Intel Math Kernel Library (MKL) back-end, which reflects the number of physical cores. GPU and multi-GPU experiments were performed on four NVIDIA Titan X (Maxwell) GPUs, each with 12 GB of memory. The implementation is built using PyCUDA and scikit-cuda python libraries. The software is available on https://github.com/acopar/fast-fusion.

### 4.4 Results

We empirically compare the convergence of multiplicative update (DFMF) and coordinate descent-based (DFCOD) versions of collective matrix factorization. The results suggest that data fusion with coordinate descent converges 4 to 17 times faster compared to DFMF. Further, we measure the runtime and speedup of our parallel implementations compared to reference scikit-fusion implementation. Note that the DFMF implementation in scikit-fusion and our parallel implementation of DFMF give numerically equi-
valent results. We measure the total time until convergence and time per iteration. We observed that single iteration of coordinate descent approach is much slower than multiplicative approach, but the overall runtime is over ten times faster as a result of faster convergence.

### 4.4.1 Convergence of data fusion methods

Fig 4.4 shows the convergence of data fusion methods on two large data collections. We evaluate the objective value with respect to iteration, until convergence criterion $\epsilon = 10^{-5}$ is triggered. DFMF represents multiplicative update technique and scikit-fusion, where DFCOD represents coordinate descent-based technique.

![Optimization traces of different collective matrix factorization techniques. Graphs show the normalized cost function function at each iteration of the training algorithm. DFMF shows the convergence of existing approach, DFCOD shows convergence of the proposed coordinate descent-based rules.](image)

Multiplicative-based approach converges in 2400 iterations on Toxicogenomics data collection and 815 iterations on Hetrec-movies data collection. Coordinate descent-based approach converges 141 iterations on Toxicogenomics and in 220 iterations on Hetrec-movies data collection, resulting in 17 times faster convergence on Toxicogenomics dataset.

### 4.4.2 Data fusion speedup

We have evaluated our parallel data fusion approach by measuring the runtime of multicore and GPU implementations until convergence. Fig 4.5 shows speedups achieved on
multi-processor system. We can observe almost ten times greater speedup using 16 processes compared to the existing scikit-fusion implementation on 16 processes. In comparison, coordinate-descent based data fusion runs up to 80 times faster compared to scikit-fusion, which can be attributed to faster convergence speed (see Fig 4.4).

![Figure 4.5](image)

**Figure 4.5**  
Speedup of total runtime until convergence on multi-processing architectures compared to scikit-fusion.

Fig 4.6 shows speedups on a single GPU. We can observe over 30 times greater speed over existing 16-process implementation using the same number of iterations, where coordinate descent performs over 100 times faster on GPU systems. Runtime of multiplicative update-based data fusion on GPU is on Toxicogenomics and Gene-chem-disease data over 100 times faster compared to reference scikit-fusion implementation on 16 processes. Our GPU implementation of DFMF is on Hetrec-movies over 30 times faster.

### 4.4.3 Efficiency of data fusion implementations

We measure speedup of a single iteration in DFMF and DFCOD with respect to different parallelization degrees to evaluate the efficiency regardless of convergence speed.

Fig 4.7 shows the speedup of a single iteration of DFMF with respect to 16-processor, single-GPU, and 4-GPU architectures. We can observe over ten times increase in speed on sparse datasets (Toxicogenomics, Hetrec-movies) using GPU architecture and up to

---

**Toxicogenomics**  
**Hetrec-movies**  
**Gene-chem-disease**

<table>
<thead>
<tr>
<th>Dataset</th>
<th>DFMF speedup</th>
<th>DFCOD speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Toxicogenomics</td>
<td>Over 100</td>
<td>Over 100</td>
</tr>
<tr>
<td>Hetrec-movies</td>
<td>Over 30</td>
<td>Over 30</td>
</tr>
<tr>
<td>Gene-chem-disease</td>
<td>Over 30</td>
<td>Over 30</td>
</tr>
</tbody>
</table>

---

**Method**
- scikit-fusion  
- DFMF  
- DFCOD

---

**Speedup of 16-thread implementations**

![Graph showing speedup of 16-thread implementations](image)
100 times increase in speed using multi-GPU architecture if the data is dense (Gene-chem-disease).

**Figure 4.6**
Speedup of total runtime until convergence on 4-GPU architecture compared to scikit-fusion.

**Figure 4.7**
Iteration speedup on 16-processor, 1-GPU and 4-GPU DFMF implementations compared to a single-core approach.
Fig 4.8 shows speedups using 16-processor, single-GPU and 4-GPU architectures for a single sparse iteration of DFCOD. We observe less than five times improvements in speed on sparse datasets. However, speedup on dense datasets is ten times on multi-processor architecture and over 30 times using GPU architectures.

### 4.4.4 Scalability

Increasing parallelization degree reduces the efficiency due to increased communication overhead between parallel processes. Parallel systems can perform more efficiently on larger problems, when the time complexity of the problem increases faster than time complexity of the communication overhead. Scalability describes how many times we need to increase the data in order to achieve the same efficiency.

We study the scalability of DFMF and DFCOD algorithms. We have generated a series of square synthetic datasets with dimensions $n \in \{5000, 10000, \ldots, 55000\}$. Larger datasets will not fit into GPU memory. In each experiment we use a single dataset, the number of iterations is set to 100, and factorization rank is set to $k = 20$.

Fig 4.9 shows the increase in efficiency for DFMF as we increase the data size on multi-processor architecture. The efficiency is substantially lower compared to NMTF shown...
in Fig 2.14. The symmetry of the factor requires increased number of factor synchronizations compared to NMTF.

The efficiency on GPU is lower compared to multi-processing architecture, since GPUs are faster at computation and slower at data transfers. Fig 4.10 shows the relation between efficiency and data size on multi-GPU architecture.

Fig 4.11 shows the efficiency on multi-processor architectures. DFCOD requires many additional synchronizations compared to DFMF. DFMF synchronizations of
factor $U$ are performed once per iteration, however, the number of synchronizations in DFCOD is proportional to the number of vectors in $U$. Data transfers on GPUs are very slow, consequently the efficiency is much lower on multi-GPU architecture compared to multi-processor architecture. For example, in Figs 4.10 and 4.12 at 3 billion entries we can see that 2-GPU DFCOD system has efficiency around 0.6 and DFMF has efficiency over 0.8.

Data transfers on CPU are faster compared to GPU. The percentage of data transfers in DFCOD is lower compared to DFMF because of the additional computational overhead. As a result, we can observe better efficiency in DFCOD (Fig 4.12) compared to DFMF (Fig 4.10).

Fig 4.13 shows the scalability on multi-processor architecture for DFMF method. The horizontal line shows the efficiency thresholds $E_{CPU-p} \in \{0.6, 0.8, 0.9, 0.95\}$, where $p$ is the number of parallel processes. We determine the minimum data size where the efficiency crosses over each of these thresholds. Results for higher parallelization degrees and higher efficiency thresholds are missing in experiments when the system never reached that level of efficiency.

Fig 4.14 shows a similar experiment on multi-GPU architecture. Here, the horizontal axis shows efficiency thresholds lowered to $E_{GPU-p} \in \{0.4, 0.5, 0.6\}$ where $p$ is the number of GPU devices. We lowered thresholds in order to make the results comparable with DFCOD on multi-GPU, where the efficiency was much lower. We can see
that the system on GPU has substantially better scalability compared to multi-processor counterpart.

Fig 4.15 shows the scalability on multi-processor architecture for DFCOD method. We can see that DFCOD has much better scalability compared to DFMF on multi-processor architecture. For example, at efficiency 0.6 and 0.8, DFCOD system needs around three times increase in data when doubling the number of processes. This is much lower compared to DFMF (Fig 4.13) and much closer to linear scalability. DF-
Figure 4.14
Scalability on 1-, 2-, 3- and 4-GPU architecture for DFMF. Each bar shows the minimum data size at which the system achieves the efficiency shown on horizontal axis.

COD is more computationally intensive, which reduces the percentage of time needed for communication and improves the scalability.

To evaluate scalability of DFCOD on multi-GPU architecture, we reduced the efficiency thresholds $E_{GPU-P} \in \{0.4, 0.5, 0.6\}$. Fig. 4.16 shows the scalability on multi-GPU architecture for DFCOD method. We observe that for increasing the number of GPU devices from 2 to 4, we need six times increase in data size to achieve the same effi-
Speedup of data fusion

ciency, for example increase from 500 to 3000 millions of entries for efficiency 0.4. These
data increases are similar to DFMF, where the increase is from 100 to 600 for efficiency
0.4 and from 200 to 1200 millions of entries for efficiency 0.5. We conclude that the effi-
ciency of DFCOD is lower than DFMF on multi-GPU architecture, but the scalability
remains the same.

Figure 4.16
Scalability on multi-GPU ar-
chitecture for
DFCOD. Each
bar corresponds
to the minimum
data size when
the method
reaches effi-
ciency threshold
shown on hori-
zontal axis.

4.4.5 Prediction accuracy on testing data

Data fusion by coordinate descent converges faster compared to existing DFMF ap-
proach. Convergence experiments in section 4.4.1 show how accurately each technique
approximates the training data. However, accuracy on training data does not tell
whether the method will be able to predict values on testing data, not present in the
model. For example, increasing factorization rank often leads to better approximation,
but decrease accuracy on testing data.

We have performed a series on experiments, where we test the model on a subset that
was held-out from the original data. From each dataset $X^{(i,j)}$ we randomly remove
20 percent of all entries which form a subset $T^{(i,j)}$. The remaining 80 percent of the
data is then used to generate factors $U^{(1)}$, $U^{(l)}$ and $S^{(1,1)}$, $S^{(l,l)}$. Each method
is run until convergence, triggered by criterion $\epsilon = 10^{-5}$. We repeat the experiments
for different factorization ranks $k \in \{10, 20, \ldots, 100\}$. Then, we reconstruct each of the
datasets using \( X^{(i,j)} = U^{(i)} S^{(i,j)} U^{(j)\top} \) and calculate the RMSE between matrices \( X^{(i,j)} \) and \( X^{*(i,j)} \) at positions in subset \( T^{(i,j)} \):

\[
\text{RMSE}^{(i,j)} = \sqrt{\frac{1}{T} \sum_{a,b} \left( X^{(i,j)}_{ab} - X^{*(i,j)}_{ab} \right)^2}.
\] (4.10)

Then, we combine RMSE from each datasets, such that each RMSE value is weighted by \( t^{(i,j)} \), which denotes the number of elements in testing subset \( T^{(i,j)} \):

\[
\text{RMSE} = \frac{\sum_{i,j} \left( t^{(i,j)} \ast \text{RMSE}^{(i,j)} \right)}{\sum_{i,j} t^{(i,j)}}.
\] (4.11)

Fig 4.17 shows RMSE on Toxicogenomics data collection. We see that DFCOD is at least as accurate as existing approach on held-out data. We also clearly see that increasing the factorization rank leads to overfitting and hence reduced accuracy on testing data.

DFMF and DFCOD have very similar performance on Toxicogenomics and Hetrec-movies data collections. As we increase factorization rank, the error increases on Toxicogenomics and Gene-chem-disease datasets, which suggests that they can be described with less latent vectors. In the case of Hetrec-movies data collection we see that the optimum factorization rank is between 20 and 30, but higher factorization rank reduces the accuracy on testing data.

### 4.5 Discussion

#### 4.5.1 Block-wise approach

Block-wise approach was implemented to increase the speed on sparse data and allow for multi-GPU parallelization. Datasets are not evenly sized, therefore treating each dataset as a separate block is not efficient and suffers from balancing problems. Processing unit that is assigned the largest dataset will be the bottleneck. Using block-wise partitioning we partition each dataset into the number of blocks that correspond to the number of processes. Then, in each iteration we block-wise apply update rules for the specific dataset.

Fig 4.18 shows the speedup of block-wise approach compared to non-block-wise. SciPy sparse linear algebra routines using the same number of processes yield slower results than our block-wise approach. Block-wise approach compared to non-block-wise
performs 4 times faster using DFMF and over 3 times greater speed using DFCOD. Our block-wise approach is around 1.5 times faster on dense dataset (Gene-chem-disease) compared to state-of-the-art NumPy libraries using the same degree of parallelization. Our block-wise approach works on a higher level, and can partition the matrices for the complete duration of the algorithm. Non-block-wise approach can only consider currently active operation and can not perform optimization based on the context. Block-wise approach can also exploit this separation to increase the speed of element-wise operations such as matrix additions, divisions and multiplications.

Figure 4.17
Impact of factorization rank on accuracy for DFMF and DFCOD. Accuracy is shown as RMSE for factorization ranks \( k \in \{10, 20, \ldots, 100\} \).
4.5.2 Communication overhead

By comparing 4.7 and 4.8 we can observe that the speed of DFCOD approach on GPU is substantially slower compared to DFMF. The higher communication requirements of DFCOD introduce a substantial slowdown, therefore 4-way GPU implementation is not much faster than single-GPU and is on sparse datasets even slower. The reason for this is that coordinate descent update rules require much more communication within the update of factor $U$ compared to multiplicative approach. If we observe the equation 4.9 and Algorithm 8, we can see that the changes in a single column of factor $U$ can influence the outcome of several other datasets. This requires multiple synchronizations of the factors within a single iteration and is the main bottleneck. This bottleneck is even more pronounced on multi-GPU architecture, since the transfer speed between GPUs is much slower compared to processor-based transfers.

However, much faster convergence of coordinate descent out weights its drawbacks in lower scalability. We observe that the coordinate descent-based approach is overall over five times faster (see Fig 4.6) compared to the multiplicative-based approach.
4.5.3 Parameter selection

There are several parameters that need to be considered when building a data fusion model: factorization rank, optimization technique, stopping criteria, and sparse format. The appropriate choice of these parameters relies heavily on the dataset. In this section, we describe a few best practices that can simplify the selection of parameters.

Factorization rank is among the most important parameters, which controls the balance between generalization and overfitting. The appropriate factorization rank is usually selected using a linear search of several values, as shown in section 4.4.5. We stop the search process when the error on testing data starts to increase.

Optimization technique defines which type of mathematical optimization is used to find the optimal solution. Our results suggest that coordinate descent is superior in terms of convergence speed, while maintaining or improving on the accuracy compared to multiplicative updates.

Selection of stopping criteria greatly influences the runtime of the approach. The stopping criterion $\epsilon = 10^{-5}$ is a good compromise, because the threshold is strict enough to prevent stopping too early during the optimization procedure, while still maintaining reasonably fast runtime. When choosing less strict parameter, for example $\epsilon = 10^{-4}$, together with multiplicative updates, we suggest setting minimum number of iterations to at least one hundred iterations to prevent early stopping.

Relational matrices are often very sparse. We suggest using a sparse format for datasets with less than 15 percent density, when using processor architecture. On GPU architecture, we suggest using sparse format only for datasets with density of less than a few percent, unless the data is too big for the memory of the GPU device. Section 2.5.7 shows an analysis of the speedups of sparse format and density thresholds.
4.6 Conclusion

Collective matrix factorization is a successful approach with many applications in bioinformatics and biomedicine. Current approach take substantial amount of time, particularly for larger datasets. We proposed a parallel collective matrix factorization approach with an order of magnitude faster runtime compared to existing approach. We also proposed a new coordinate descent-based collective matrix factorization which converges more than 15 times faster compared to multiplicative update-based approach. The final solution that integrates both, the convergence speedup and parallelization speedup, is more than a hundred times faster compared to an existing approach.
Conclusion
Relational data often combines many different object types with large number of instances. Current data fusion approaches take substantial runtime, particularly for larger datasets. Our aim is to improve the speed of data fusion, in particular, we focus on collective matrix factorization-based data fusion. Collective matrix factorization is based on non-negative matrix tri-factorization, where the former deals with more than two object types and the latter with less than three object types.

In this work we first focus on improving non-negative matrix tri-factorization, and then apply the same techniques to speed up data fusion. We increased the speed of non-negative matrix tri-factorization by leveraging highly parallel systems with multiple cores and GPU devices. We developed a block-wise variant of the tri-factorization that can efficiently run on multi-processing and multi-GPU environments. We report on up to 200 times faster speed compared to a single-core variant on a multi-GPU system. Then, we focused on improving the mathematical convergence of non-negative matrix tri-factorization. We present there alternative approaches to existing multiplicative-based approach. The best performance was achieved with our coordinate descent-based approach which shows over 15 times shorter runtime compared to the existing approach. Finally, we applied the block-wise parallelization to existing data fusion approach. The speed on four GPU devices is over 100 times faster compared to single-core version. Further, we developed coordinate descent-based data fusion that converges up to 17 times faster compared to an existing approach for the same computational resources.

There are many avenues for future work. In our work we focused primarily on the parallelization and convergence of non-constrained NMTF and data fusion. Depending on the application, imposing additional orthogonality and sparsity constraints can further improve the results. In data fusion, regularization term that penalizes negative relations is often desired. The provided detailed mathematical derivations (Appendices A.2.1-A.3.2) should provide a good foundation to formulate constrained variants of our work.

Distributed computation containing multiple nodes can be another avenue of further work. If the dataset is orders of magnitude larger, it might be beneficial to run the setup on multiple nodes, each containing multiple CPU and GPU devices. We noticed that the communication between parallel processes is the main bottleneck and even intra-node communications can have a large impact on efficiency. Because of the slower network speeds and latency, it is challenging to design a multi-node system that would scale well. Perhaps alternative optimization techniques with asynchronous communica-
tion could be developed to reduce the data transfers in distributed environments, while still maintaining robust convergence.

Finally, our methods work with two-dimensional datasets. An interesting extension of our approach would be extension of parallelization and convergence improvements to non-negative tensor tri-factorization and related data fusion that works with data where instead of matrices, we model third-order tensors.
Mathematical derivations
A. Čopar

Scalable matrix factorization for data fusion

A.1 Equivalence of block-wise and non-block-wise formulation of NMTF

We show that our block-wise formulation of NMTF is mathematically equivalent to the non-block-wise NMTF, which does not partition a given input data matrix into blocks. That is, the block-wise version of NMTF yields the same latent factors as its non-block-wise counterpart. As a result, the proposed approach for latent factor learning in NMTF has the same predictive power as standard NMTF.

To show this mathematical equivalence we proceed as follows. For each latent matrix $U$, $V$, and $S$ we show that its updating rule in the block-wise NMTF is equivalent to the corresponding updating rule in standard NMTF. In particular, we establish a connection between the updating rules in standard NMTF (2.2)–(2.4) and the rules in block-wise NMTF in Eqs. (2.9)–(2.11). We provide the relevant proofs in Lemmas 1–3.

A.1.1 Updating rule for latent matrix $U$

Lemma 1: The updating rules in Eq. (2.2) and Eq. (2.9) are equivalent.

Proof 1: We show that every entry $u, v$ in matrix $U$ results in the same values. Note that $\odot$ and $\oslash$ represent Hadamard multiplication and division. The updating rule in left side of Eq. (2.2) can be written as:

$$U_{uv} \leftarrow U_{uv} \odot (XVS^T)_{uv} \oslash (USV^T VS^T)_{uv}.$$  

We therefore only need to show that expressions in numerators and denominators of the updating rules are equivalent. In particular, we need to show that the following holds:

$$(XVS^T)_{uv} = k \sum_{x=1}^{m} (X^{(i,j)}S^T)_{uv} = k \sum_{x=1}^{m} \left( \sum_j X^{x(i,j)} \odot V^{(j)} \right) S_{xy}. \quad (A.1)$$
Expressions in the left and the right hand side of Eq. (A.1) are equivalent when the following holds:

\[(\mathbf{XV})_{ux} = \left( \sum_j \mathbf{X}^{(i,j)}(V^{(j)}) \right)_{ux},\]

which can be simply shown by writing the product \(\mathbf{XV}\) as a sum of entry-wise operations:

\[\sum_{y=1}^{m} X_{uy} V_{yx} = \sum_j \left( \sum_{y=1}^{m_j} \mathbf{X}^{(i,j)}(V^{(j)}) \right)_{yx},\]

and observing that \(u\) is assigned to the \(i\)-th block and that \(\sum_j m_j = m\).

We use the same approach to show that expressions in denominators of the updating rules are equivalent:

\[\left(\mathbf{USV}^T\mathbf{VS}^T\right)_{uv} = \left( \mathbf{U}^{(i)} \sum_j (\mathbf{S}((V^{(j)}))^{T}(V^{(j)})^{S^T})) \right)_{uv},\]

\[\sum_x U_{ux} (\mathbf{SV}^T\mathbf{VS}^T)_{xy} = \sum_x \left( \mathbf{U}^{(i)} \right)_{ux} \left( \mathbf{S} \left( \sum_j \mathbf{V}^{(j)}V^{(j)} \right) \mathbf{S^T} \right)_{xy} .\]

Since \(\mathbf{V}^T\mathbf{V} = \sum_j \mathbf{V}^{(j)}T\mathbf{V}^{(j)}\) holds true, we can simplify the expression in Eq. (A.3) to: \(U_{ux} = \left( \mathbf{U}^{(i)} \right)_{ux}\). Finally, we conclude that Eq. (A.2) holds true for every \(u, x, i,\) where \(U_{ux} \in \mathbf{U}^{(i)}\). This concludes our proof.

### Updating rule for latent matrix \(\mathbf{V}\)

**Lemma 2:** The updating rules in Eq. (2.3) and Eq. (2.10) are equivalent.

**Proof 2:** We show that every entry \(u, v\) in matrix \(\mathbf{V}\) has the same update under both updating rules. Note that multiplication (\(\odot\)) and division (\(\oslash\)) are entry-wise operations and thus the updating rule in Eq. (2.3) can be written as:

\[V_{uv} \leftarrow V_{uv} \odot (X^TUS)_{uv} \odot (VS^TUS)_{uv}.\]

We therefore only need to show that expressions in numerators and denominators of the updating rules are equivalent. In particular, we need
to show that the following holds:

\[(X^TUS)_{uv} = \left( \sum_i ((X^{(i,j)})^T U^{(i)}) S \right)_{uv}, \]

\[(VS^T U^T US)_{uv} = \left( V^{(j)} S^T \sum_i (U^{(i)})^T U^{(i)} S \right)_{uv}. \]

Using definition of the matrix product \((AB)_{ij} = \sum_k A_{ik} B_{kj}\) and the right-associativity rule of matrix product we get:

\[
\sum_{x=1}^k (X^T U)_{ux} S_{vx} = \sum_{x=1}^k \left( \sum_i X^{(i,j)} U^{(i)} \right)_{ux} S_{vx}. \tag{A.4}
\]

Expressions in the left and the right hand side of Eq. \((A.4)\) are equivalent when the following holds:

\[(X^T U)_{ux} = \sum_i X^{(i,j)} U^{(i)}_{ux}, \]

which can be simply shown by writing the product \(X^T U\) as a sum of entry-wise operations:

\[
\sum_{y=1}^n X_{yu} U_{yx} = \sum_i \sum_{y=1}^{n_j} \left( X^{(i,j)} \right)_{yu} \left( U^{(i)} \right)_{yx},
\]

and observing that \(u\) is assigned to the \(i\)-th block and that \(\sum_j m_j = m\).

We use the same approach to show that expressions in denominators of the updating rules are equivalent:

\[(VS^T U^T US)_{uv} = \left( V^{(j)} S^T \sum_i (U^{(i)})^T U^{(i)} S \right)_{uv}, \tag{A.5}\]

\[
\sum_x V_{ux} (S^T U^T US)_{vx} = \sum_x \left( V^{(j)} \right)_{ux} \left( S^T \sum_i (U^{(i)})^T U^{(i)} S \right)_{vx}. \tag{A.6}
\]

Since \(U^T U = \sum U^{(i)} U^{(i)}\) holds true, we can simplify the expression in Eq. \((A.6)\) to: \(V_{ux} = \left( V^{(j)} \right)_{ux}\). Finally, we conclude that Eq. \((A.5)\) holds true for every \(u, x, j\), where \(V_{ux} \in V^{(j)}\). This concludes our proof.
Lemma 3: The updating rules in Eq. (2.4) and Eq. (2.11) are equivalent.

Proof 3: We show that every entry $u, v$ in matrix $S$ has the same update under both updating rules. Note that multiplication ($\odot$) and division ($\oslash$) are entry-wise operations and thus the updating rule in Eq. (2.4) can be written as:

$$S_{uv} \leftarrow S_{uv} \odot (U^T X V)_{uv} \odot (U^T U S V^T V)_{uv}.$$ 

We therefore only need to show that expressions in numerators and denominators of the updating rules are equivalent. In particular, we need to show that the following holds:

$$\left(\sum_j \sum_i (U^{(i)})^T X^{(i,j)} V^{(j)}\right)_{uv} = \left(\sum_i \left( ((U^{(i)})^T U^{(i)}) S \sum_j \left( (V^{(j)})^T V^{(j)} \right) \right) \right)_{uv}.$$ 

(A.8)

Expressions in the left and the right hand side of Eq. (A.7) are equivalent when $U_{ux}$ is a member of $U^{(i)}$ and the following holds:

$$(X V)_{xy} = \sum_j \sum_i (X^{(i,j)} V^{(j)})_{xy},$$

which can be written as a sum of entry-wise operations:

$$\sum_{y=1}^m X_{xy} V_{yp} = \sum_j \sum_i \left( \sum_{y=1}^m X_{xy}^{(i,j)} V_{yp}^{(j)} \right).$$ 

(A.8)

Eq. (A.8) holds true for every $x, y, i, j$, where $X_{xy} \in X^{(i,j)}$ and for every $y, v, j$, where $V_{yp} \in V^{(j)}$. We use the same approach to show that expressions
in denominators of the updating rules are equivalent:

\[
(U^TUSV^T)_{uv} = \left( \sum_i \left( (U^{(i)})^T U^{(i)} \right) S \sum_j \left( (V^{(j)})^T V^{(j)} \right) \right)_{uv}. \tag{A.9}
\]

Since \( U^T U = \sum_i U^{(i)} U^{(i)} \) and \( V^T V = \sum_j V^{(j)} V^{(j)} \) hold true, we can simplify the expression on the right hand side of Eq. (A.9) to \((U^TUSV^T)_{uv}\), which is equivalent to the left hand side. This concludes our proof.

\section*{A.2 Derivation of NMTF optimization techniques}

\subsection*{A.2.1 Multiplicative Update Rules}

Multiplicative update rules for non-negative matrix tri-factorization and an orthogonal variant were introduced \cite{16,17}. We focus on the non-orthogonal variant \cite{16} to make the approach comparable with alternative optimization techniques without added constraints. Both methods and all derivations in the following sections are based on the squared Frobenius norm objective function:

\[
D_{Fro}(X||USV^T) = \|X - USV^T\|_F^2, \tag{A.10}
\]

where \(X\) denotes the input data and \(U, S\) and \(V\) are latent factors. Squared Frobenius norm can be written as \(\|A\|^2 = Tr(A^TA)\), therefore \(D_{Fro}\) equals:

\[
F = Tr(X^TX - 2X^TUSV^T + VS^TUSV^T). \tag{A.11}
\]

Let \(\lambda_1, \lambda_2\) and \(\lambda_3\) be the Lagrange multipliers for the non-negative constraints \(U \geq 0\), \(V \geq 0\) and \(S \geq 0\). We construct the Lagrange function:

\[
L = F - \lambda_1 U - \lambda_2 V - \lambda_3 S.
\]
Using the Karush-Kuhn Tucker (KKT) complementary conditions, we can find a static point. The KKT conditions are:

\[
\frac{\partial L}{\partial U} = 0, \quad (A.12)
\]
\[
\frac{\partial L}{\partial V} = 0, \quad (A.13)
\]
\[
\frac{\partial L}{\partial S} = 0, \quad (A.14)
\]
\[
\lambda_1 \odot U = 0, \quad (A.15)
\]
\[
\lambda_2 \odot V = 0, \quad (A.16)
\]
\[
\lambda_3 \odot S = 0. \quad (A.17)
\]

The partial derivatives of the Lagrange functions with respect to \(U, V,\) and \(S\) are:

\[
\frac{\partial L}{\partial U} = -2XVS^T + 2USV^TVS^T - \lambda_1, \quad (A.18)
\]
\[
\frac{\partial L}{\partial V} = -2X^TUS + 2VS^TU^TUS - \lambda_2, \quad (A.19)
\]
\[
\frac{\partial L}{\partial S} = -2U^TXV + 2U^TUSV^TV - \lambda_3. \quad (A.20)
\]

We apply Hadamard multiplication with \(U\) and condition \((A.12)\) on Eq. \((A.18)\):

\[-2XVS^T \odot U + (2USV^TVS^T) \odot U - \lambda_1 \odot U = 0.\]

Using the condition \((A.15)\), we remove the third term:

\[-2XVS^T \odot U + (2USV^TVS^T) \odot U = 0.\]

We explicitly define \(U\), where the division is performed entry-wise:

\[U = U \odot (XVS^T) \odot (USV^TVS^T).\]

Similarly, we use the partial derivative of \((A.19)\), apply condition \((A.13)\) and multiply with \(V\):

\[-2X^TUS \odot V + (2VS^TU^TUS) \odot V - \lambda_2 \odot V = 0.\]

Using the condition \((A.16)\), we remove the third term:

\[-2X^TUS \odot V + (2VS^TU^TUS) \odot V = 0.\]
The explicit update for $V$ is:

$$V = V \odot (X^TUS) \odot (VS^TUS).$$

Finally, we use the partial derivative with respect to $S$ \((A.20)\) and apply condition \((A.14)\). We multiply the equation with $S$:

$$(-2U^TXV) \odot S + (2U^TUSV^TV) \odot S - \lambda_3 \odot S = 0.$$

Using the condition \((A.17)\), we remove the third term:

$$(-2U^TXV) \odot S + (2U^TUSV^TV) \odot S = 0$$

Using explicit form and Hadamard division, we get the update rule for factor $S$:

$$S = S \odot (U^TXV) \odot (U^TUSV^TV)$$

### A.2.2 Alternating Least Squares

Let us calculate the gradient of function \((A.11)\) with respect to each factor matrix $U$, $V$ and $S$:

\[
\begin{align*}
\frac{\partial F}{\partial U} &= 2XVS^T - 2USV^TVS^T, \quad \text{(A.21)} \\
\frac{\partial F}{\partial V} &= 2X^TUS - 2VS^TUS, \quad \text{(A.22)} \\
\frac{\partial F}{\partial S} &= 2U^TXV - 2U^TUSV^TV. \quad \text{(A.23)}
\end{align*}
\]

We equate the gradient for each factor to zero. After this step, we calculate the inverse to get the update rule for $U$:

$$USV^TVS^T = XV^TV$$

$$U = XVS^T(SV^TVS^T)^{-1}.$$

This step can introduce negative values, so we must force non-negativity by assigning all negative entries with zero. Note that this update rule will not work unless $SV^TVS^T$ is invertible. If we use random initialization, this matrix is invertible, however during the projection step potentially too many zeroes is introduced to the model, so this condition can no longer hold. If this happens, the algorithm is stopped and its results are discarded.

$$U = [XVS^T(SV^TVS^T)^{-1}]_+.$$
Following the derivative for $V$ from Eq. (A.22), we can similarly formulate update rule for $V$. We equate the gradient with zero $\frac{\partial F}{\partial V} = 0$, calculate the inverse and force non-negativity:

$$
STU^TUS^T = STU^TX,
$$

$$V^T = (S^TU^TUS)^{-1}S^TU^TX,
$$

$$V = X^TUS(S^TU^TUS)^{-1},
$$

$$V = [X^TUS(S^TU^TUS)^{-1}]_+.
$$

Derivative for $S$ is shown in equation (A.23). We set the gradient $\frac{\partial F}{\partial S}$ to zero and force non-negativity constraints:

$$U^TUSV^TV = U^TXV,
$$

$$S = (U^TU)^{-1}(U^TXV)(V^TV)^{-1},
$$

$$S = [(U^TU)^{-1}(U^TXV)(V^TV)^{-1}]_+.
$$

### A.2.3 Projected Gradients

Projected gradient methods are based on the general gradient descent scheme, where we take the variable $Y$ and create a step towards the descent direction $P$ scaled with the learning rate $\eta$:

$$Y \leftarrow Y - \eta P.
$$

We derive projected gradient algorithm for NMTF, for the squared Frobenius norm objective function (A.10). We add projection to the non-negative values $[\ ]_+$ to enforce non-negativity of the factors in case in crosses into negative values. Projected gradient methods follow this form:

$$U \leftarrow [U - \eta_u P_U]_+,
$$

$$S \leftarrow [S - \eta_s P_S]_+, \quad (A.24)
$$

$$V \leftarrow [V - \eta_v P_V]_+.$$
where $P_U$, $P_V$ and $P_S$ are descent directions, $\eta_u$, $\eta_v$ and $\eta_s$ are learning rates. Descent directions are defined with the following form:

\[
\begin{align*}
P_U &= D_U \odot \frac{\partial F}{\partial U}, \\
P_V &= D_V \odot \frac{\partial F}{\partial V}, \\
P_S &= D_S \odot \frac{\partial F}{\partial S}.
\end{align*}
\]

(A.25)

The scaling factors $D_U$, $D_V$ and $D_S$ are set the following way:

\[
\begin{align*}
D_U &= U \odot (USV^T VS^T), \\
D_V &= V \odot (VS^T U^T US), \\
D_S &= S \odot (U^T USV^T V).
\end{align*}
\]

The scaling factor was chosen based on a study which compares convergence analysis of four different scaling factors [96]. The choice of scaling factor is inspired by existing projected gradient algorithm for non-negative matrix factorization [25]. The learning rate or step size parameter is dynamically chosen using the same form as a related study for classical non-negative matrix factorization [25]:

\[
\eta_u = Tr((SV^T V)(S^T P_U^T P_U)).
\]

We calculate the descent direction for each factor matrix $U$ using the descent direction for $U$ and the partial derivative (A.21):

\[
\begin{align*}
P_U &= (U \odot USV^T VS^T) \odot (X - USV^T) VS^T, \\
P_U &= U - U \odot USV^T VS^T \odot XV^T S^T.
\end{align*}
\]

We define learning rate for $V$ as follows:

\[
\eta_v = Tr((SP_V^T P_V)(SU^T U)).
\]

We calculate the descent direction for $V$ by inserting the $D_S$ and derivative in Eq. (A.22) into the Eq. (A.26):

\[
\begin{align*}
P_V &= (V \odot VS^T U^T US) \odot (X - USV^T)^T US, \\
P_V &= V - V \odot VS^T U^T US \odot X^T US.
\end{align*}
\]
We define learning rate for $S$ as follows:

$$\eta_s = Tr((U^TUP_s)(V^TVP_s^T)).$$

The descent direction for $S$ is computed by inserting the $D_S$ and derivative in Eq. (A.23) into the Eq. (A.26):

$$P_S = (S \odot U^TUSV^TV) \odot U^T(X - USV^TV),$$

$$P_S = S - S \odot U^TUSV^TV \odot U^TXV.$$

We insert the descent directions and learning rates in Eq. (A.24). The resulting update rules for $U$ are:

$$P_u = U - U \odot (USV^TVS^T) \odot (XVS^T),$$

$$\eta_u = \frac{\sum (P_u \odot (USV^TVS^T - XVS^T))}{Tr((SV^TV)(S^TP_u^TP_u))},$$

$$U \leftarrow [U - \eta_u P_u]_+.\]$$

The following update rules define the procedure for updating factor $V$:

$$P_v = V - V \odot (VS^TUS^T) \odot (X^TUS),$$

$$\eta_v = \frac{\sum (P_v \odot (VS^TUS^T - X^TUS))}{Tr((SP_v^TP_v)(S^TU^TU))},$$

$$V \leftarrow [V - \eta_v P_v]_+.\]$$

Resulting update rules for factor $S$ are:

$$P_s = S - S \odot (U^TUSV^TV) \odot (U^TXV),$$

$$\eta_v = \frac{\sum (P_s \odot (U^TUSV^TV - U^TXV))}{Tr((U^TUP_s)(V^TVP_s^T))},$$

$$S \leftarrow [S - \eta_v P_s]_+.\]$$

### A.2.4 Coordinate Descent

We describe factors $U$, $V$, and $S$ as a set of column vectors, where $U$ is a set of column vectors $\{u_1, u_2, ..., u_{k_1}\}$, where $u_i$ is $i$-th column of $U$. Similarly $V$ is a set of column vectors $\{v_1, v_2, ..., v_{k_2}\}$ and factor $S$ as a set of entries $\{s_{11}, s_{12}, ..., s_{1k_2}, s_{21}, ..., s_{k_1k_2}\}$.
Updating factor $U$ consists of as series of $k_1$ number of updates, where we update each column at a time. Similarly, we update factor $V$ as a series of $k_2$ number of column updates. Updates for factor $S$ is done as a series $k_1 \cdot k_2$ number of updates, where each element in $S$ is updated individually. We define the following residual matrices:

$$R_u = X - USV^T + u_i (SV^T)_i,$$  \hspace{1cm} (A.26)

$$R_v = X - USV^T + (US)_j v_j^T,$$  \hspace{1cm} (A.27)

$$R_s = X - USV^T + (u_i s_i v_j^T).$$  \hspace{1cm} (A.28)

The idea is to iterate over the following set of functions for each $i \in \{1 \ldots k_1\}$ and $j \in \{1 \ldots k_2\}$:

$$F_U = ||R_u - u_i (SV^T)_i||^2,$$

$$F_V = ||R_v - (US)_j v_j^T||^2,$$

$$F_S = ||R_s - u_i s_i v_j^T||^2.$$

First, we derive the update rule for $F_U$, where we use $||A||_2 = Tr(A^T A)$ and $(AB)_i = Ab_i$.

$$F_U = Tr(R_u^T R_u - 2R_u^T u_i s_i V^T + V s_i^T u_i s_i V^T),$$

$$\frac{\partial F_U}{\partial u_i} = -2R_u V s_i^T + 2u_i s_i V^T V s_i^T.$$

Then, we equate the derivative $\frac{\partial F_U}{\partial u_i}$ to zero and divide by scalar $s_i V^T V s_i^T$:

$$R_u V s_i^T = u_i s_i V^T V s_i^T,$$

$$u_i = (R_u V s_i^T) \odot (s_i V^T V s_i^T).$$

Update for the function $F_V$:

$$F_V = Tr(R_v^T R_v - 2R_v^T U s_j v_j^T + v_j s_j^T U^T U s_j v_j^T),$$

$$\frac{\partial F_V}{\partial v_j} = -2R_v^T U s_j + 2v_j s_j^T U^T U s_j.$$

We equate the derivative $\frac{\partial F_V}{\partial v_j}$ to zero and divide by scalar $s_j^T U^T U s_j$:

$$R_v^T U s_j = v_j s_j^T U^T U s_j,$$

$$v_j = (R_v^T U s_j) \odot (s_j^T U^T U s_j).$$
Finally, we find the static point for the $F_S$ subproblem:

$$F_S = \text{Tr}(R_s^T R_s - 2R_s^T u_i s_j v_j^T + v_j s_j u_i s_j v_j^T),$$

and its derivative with respect to $s_{ij}$:

$$\frac{\partial F_S}{\partial s_{ij}} = -2u_i^T R_s v_{.j} + 2u_i^T R_s s_{ij} v_{.j} v_{.j}.$$ 

We equate the derivative $\frac{\partial F_S}{\partial s_{ij}}$ to zero and then divide by scalars $u_i^T u_i$ and $v_{.j}^T v_{.j}$:

$$u_i^T R_s v_{.j} = u_i^T u_i s_{ij} v_{.j} v_{.j},$$

$$s_{ij} = \left( u_i^T R_s v_{.j} \right) \odot \left( u_i^T u_i v_{.j} v_{.j} \right).$$

For efficient implementation, we replace the residual matrices $R_u, R_v$ and $R_s$ with the definitions in equations (A.26), (A.27) and (A.28). We also enforce non-negativity with projection to non-negative values to each update rule. The resulting update rules are:

$$u_{.i} \leftarrow u_{.i} + \left[ \left( (XVS^T)_{.i} - (USV^T V S_i^T)_{.i} \right) \odot \left( s_{i} V^T V S_{i}^T \right) \right]_+,$$

$$v_{.j} \leftarrow v_{.j} + \left[ \left( (X^T U S)_{.j} - (V S^T U^T U S)_{.j} \right) \odot \left( s_{.j} U^T U S_{.j} \right) \right]_+,$$

$$s_{ij} \leftarrow s_{ij} + \left[ \left( (U^T X V)_{ij} - (U^T U S V^T V)_{ij} \right) \odot \left( u_i^T u_i v_j^T v_j \right) \right]_+.$$ 

### A.2.5 Quasi-Newton Update Rules

In this section we show the equivalence of quasi-Newton update rules and ALS. Quasi-Newton approach is a second order optimization technique \[143\] that follows the following optimization approach, where $H_k$ is the Hessian and $g(x_k)$ is the gradient of $x_k$:

$$x_{k+1} = x_k - H_k^{-1} g(x_k).$$

Applying the same optimization approach on factors $U$, $V$, and $S$ and converting to vec-operator notation, we get the following form:

$$\text{vec}(U) \leftarrow \left[ \text{vec}(U) - H_u^{-1} \text{vec} \left( \frac{\partial F}{\partial U} \right) \right]_+, \quad (A.29)$$

$$\text{vec}(V) \leftarrow \left[ \text{vec}(V) - H_v^{-1} \text{vec} \left( \frac{\partial F}{\partial V} \right) \right]_+, \quad (A.30)$$

$$\text{vec}(S) \leftarrow \left[ \text{vec}(S) - H_s^{-1} \text{vec} \left( \frac{\partial F}{\partial S} \right) \right]_+, \quad (A.31)$$
where $H_U$, $H_V$, and $H_S$ are the Hessians of factors $U$, $V$, and $S$ respectively and the vec-operator is shown with vec. Let us calculate the gradient and the Hessian of function Eq. (A.10) with respect to $U$. We calculate Hessian by deriving (A.21), we get the following:

$$H_U = SV^T V^T S^T \otimes I.$$ 

The Kronecker product is shown with $\otimes$, and $I$ is identity matrix. The Kronecker product with identity matrix is added to convert between vector and matrix form. By placing the gradient and the Hessian into the equation (A.29), we get:

$$\text{vec}(U) \leftarrow \left[ \text{vec}(U) - (SV^T V^T S^T \otimes I)^{-1} \text{vec}(USV^T V^T S^T - XV^T S^T) \right],$$

which by converting from vec-operator to matrix form, using the rule $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$ and vec$(ABC) = (C^T \otimes A) \text{vec}(B)$ simplifies to:

$$U = \left[ U - (USV^T V^T S^T - XV^T S^T)(SV^T V^T S^T)^{-1} \right].$$

Further, we derive the gradient and the Hessian matrix of the factor $V$. From (A.22) we compute

$$H_V = S^T U^T U S \otimes I.$$ 

We place the gradient and Hessian into equation (A.30) and convert to matrix form:

$$V \leftarrow \left[ V - (VS^T U^T US - X^T US)(S^T U^T US)^{-1} \right].$$

The gradient and the Hessian of the $S$ matrix are:

$$\frac{\partial F}{\partial S} = U^T X V - U^T US V^T V,$$

$$H_S = V^T V \otimes U^T U.$$

We place the gradient and the Hessian into the equation (A.31).

$$\text{vec}(S) = \left[ \text{vec}(S) - (V^T V \otimes U^T U)^{-1} \text{vec}(USV^T V^T S^T - XV^T S^T) \right],$$

We use the following rule to convert from vector to the matrix form: vec$(ABC) = (C^T \otimes A) \text{vec}(B)$, where $A = (U^T U)^{-1}$, $C = (V^T V)^{-1}$, and $B = \frac{\partial F}{\partial S}$. Note that matrix $A^T A$ is always symmetric, therefore $(V^T V)^T = V^T V$.

$$S \leftarrow \left[ S - (U^T U)^{-1} (U^T US V^T V - U^T XV)(V^T V)^{-1} \right].$$
Let us simplify the equations \((A.32)\), \((A.32)\) and \((A.32)\), using the distributive rule \((A + B)C = AC + BC\):

\[
\begin{align*}
U & \leftarrow \left[ XVS^T(SV^T VS^T)^{-1} \right]_+, \\
V & \leftarrow \left[ X^T US(S^T U^T US)^{-1} \right]_+, \\
S & \leftarrow \left[ (U^T U)^{-1}(U^T XV)(V^T V)^{-1} \right]_+.
\end{align*}
\]

These rules are equivalent to the ALS update rules. Note that similar was already shown for NMF by Cichocki. [18]

### A.3 Derivation of data fusion optimization techniques

#### A.3.1 Data fusion by matrix factorization

In this section we show the derivations for the data fusion based on multiplicative update rules, where factor is a non-negative matrix \( U \in \mathbb{R}^+ \). For a given data collection composed of \( N \) object types, we define the objective function as a sum of Frobenius norms of each dataset, where \( I \in \{1, \ldots, N\} \) and \( J \in \{1, \ldots, N\} \):

\[
D_{Fro}(X||USU^T) = \sum_{IJ} D_{Fro}(X(I,J)||U(I)S(I,J)U(J)^T).
\]

(A.32)

We concatenate the set of matrices \( U(1), U(2), \ldots, U(N) \) into a block-diagonal matrix \( U \). \( S(I,J) \) matrices are concatenated into a block matrix \( S \). Block-wise concatenation is shown in (4.2) and (4.1).

Using the block matrices \( X, U, \) and \( S \), the loss function \( A.32 \) is equal to the following function

\[
D_{Fro}(X||USU^T) = ||X - USU^T||_{Fro}^2,
\]

(A.33)

which is equal to:

\[
D_{Fro}(X||USU^T) = Tr(X^TX - 2X^TUSU^T + US^TU^TUSU^T).
\]

(A.34)

We introduce Lagrangian multiplier \( \lambda \) for the non-negative constraint \( U \geq 0 \) and construct the Lagrange function:

\[
L = D_{Fro}(X||USU^T) - \lambda U.
\]

(A.35)
The Karush-Kuhn-Tucker conditions are the following:

\[
\frac{\partial L}{\partial S} = 0, \quad (A.36)
\]

\[
\frac{\partial L}{\partial U} = 0, \quad (A.37)
\]

\[
\lambda \odot U = 0. \quad (A.38)
\]

Please note that there is no non-negativity constraint on factor $S$, hence only one Lagrange multiplier. The partial derivative of (A.35) with respect to $S$ is

\[
\frac{\partial L}{\partial S} = -2U^TXU + 2U^TUSU^TU. \quad (A.39)
\]

Using the KKT condition (A.36), the partial derivative $\frac{\partial L}{\partial S}$ equals zero:

\[-2U^TXU + 2U^TUSU^TU = 0. \]

By multiplying with the inverse $U^TU$ on both sides, the resulting update rule is:

\[S \leftarrow (U^TU)^{-1}U^TXU(U^TU)^{-1}. \quad (A.39)\]

To calculate partial derivative of $U$, we need a second order derivative and fourth order derivative of second and third term in (A.34). The partial derivative of (A.35) with respect to $U$ is

\[
\frac{\partial L}{\partial U} = -2XUS^T - 2X^TUS + 2USU^TUS^T + 2U^TUS^TUS - \lambda. \quad (A.40)
\]

Note that the first two terms in (A.40) come from the derivative of $2X^TUSU^T$ and the derivative of $US^TUU^TUSU^T$ results in the last two terms in (A.40). We multiply the equation with $U^2$, where $U^2 = U \odot U$:

\[(-2XUS^T - 2X^TUS) \odot U^2 + (2USU^TUS^T + 2US^TU^TUS) \odot U^2 - \lambda \odot U^2 = 0. \]

Using the condition Eq. (A.38) the last term equals zero:

\[(-XUS^T - X^TUS) \odot U^2 + (USU^TUS^T + US^TU^TUS) \odot U^2 = 0. \]
We explicitly expose $U$, where the division and square root are performed entry-wise:

$$U = U \odot \sqrt{(XUS^T + X^TUS) \odot (US^TUS + U^TUSUS)}.$$  \hfill (A.41)

Using the block-diagonal formulation of $U$ and block matrices $X$ and $S$ in the update rules (A.41) and (A.39) is very inefficient, because they contain zero padding. We can convert (A.39) into block-wise form:

$$S^{(I,J)} \leftarrow (U^{(I)}^TU^{(I)})^{-1}U^{(I)}^TX^{(I,J)}U^{(J)}(U^{(J)}^TU^{(J)})^{-1}.$$  

Similarly, we convert (A.41) into block-wise form:

$$U^{(I)} \leftarrow U^{(I)} \odot \sqrt{E^{(I)} \odot D^{(I)}},$$  \hfill (A.42)

where $E$ represents enumerator and $D$ denominator in Eq. A.42:

$$E^{(I)} = \sum_{I,J} \left( X^{(I,J)}U^{(I)}S^{(I,J)} + X^{(I,J)}^TU^{(I)}S^{(I,J)} \right),$$

$$D^{(I)} = \sum_{I,J} \left( U^{(I)}S^{(I,J)}U^{(J)} + U^{(I)}S^{(I,J)}U^{(I)}^TS^{(I,J)} + \right.$$  

$$\left. + U^{(J)}S^{(I,J)}U^{(I)}U^{(I)}^TS^{(I,J)} \right).$$

### A.3.2 Data fusion with coordinate descent

We derive the update rules for coordinate descent-based data fusion. Note that the update rule for factor $S$ is the same as in Appendix A.3.1, thus only $U$ derivations are presented here.

There are two occurrences of factor $U \in \mathbb{R}_+^{n \times k}$ in Eq. (A.32). To consider both the left and the right factor $U$, we define two residual matrices and their derivatives. Then we combine the two derivatives to update each column $\mu_i$ in factor $U$. First, we define the residual matrices:

$$R_{ui} = X - USU^T + \mu_i(SU^T)i, \hfill (A.43)$$

$$R_{uj} = X - USU^T + (US)i\mu_j^T,$$  \hfill (A.44)

where $R_{ui}$ denotes residual matrix for the columns in left $U$ and $R_{uj}$ denotes residual matrix for the columns of right factor $U$. Each time $U$ is updated, we iterate all columns.
function $\mathbf{u}_i$, where $i \in \{1, 2, \ldots, k\}$. The objective value $A.33$ is therefore equal to the following functions:

\[
G_{U_i} = ||\mathbf{R}_{ui} - \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i||^2,
\]
\[
G_{U_j} = ||\mathbf{R}_{uj} - (\mathbf{U}\mathbf{S})_i\mathbf{u}_j||^2,
\]

which is equal to:

\[
G_{U_i} = \operatorname{Tr}(\mathbf{R}_{ui}^T \mathbf{R}_{ui} - 2\mathbf{R}_{ui}^T \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i + \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{S}\mathbf{U}^T)_i),
\]
\[
G_{U_j} = \operatorname{Tr}(\mathbf{R}_{uj}^T \mathbf{R}_{uj} - 2\mathbf{R}_{uj}^T (\mathbf{U}\mathbf{S})_i\mathbf{u}_j + (\mathbf{U}\mathbf{S})_i\mathbf{u}_j (\mathbf{S}\mathbf{U}^T)_i).
\]

We compute the partial derivative with respect to $\mathbf{u}_i$ for functions $G_{U_i}$ and $G_{U_j}$ combine them together.

\[
\frac{\partial G_{U_i}}{\partial \mathbf{u}_i} + \frac{\partial G_{U_j}}{\partial \mathbf{u}_i} = -2\mathbf{R}_{ui} (\mathbf{U}\mathbf{S}^T)_i - 2\mathbf{R}_{uj}^T (\mathbf{U}\mathbf{S})_i +
\]
\[
+ 2\mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i + 2\mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i.
\]

Now, we equate the partial derivatives to zero:

\[
\mathbf{R}_{ui} (\mathbf{U}\mathbf{S}^T)_i + \mathbf{R}_{uj}^T (\mathbf{U}\mathbf{S})_i = \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i + (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i
\]

We replace the residual matrices $\mathbf{R}_{ui}$ and $\mathbf{R}_{uj}$ with their definitions in (A.43) and (A.44):

\[
\mathbf{X} (\mathbf{U}\mathbf{S}^T)_i - \mathbf{U}\mathbf{S}^T (\mathbf{U}\mathbf{S}^T)_i + \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S}^T)_i +
\]
\[
+ \mathbf{X}^T (\mathbf{U}\mathbf{S})_i - \mathbf{U}\mathbf{S}^T (\mathbf{U}\mathbf{S})_i + \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i = \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i + \mathbf{u}_i (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i.
\]

We divide the whole expression with the following scalar normalization term:

\[
\mathbf{D}_i = (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S}^T)_i + (\mathbf{S}\mathbf{U}^T)_i (\mathbf{U}\mathbf{S})_i.
\]

The resulting update rule for $\mathbf{u}_i$ is in the following form, where we also project values into non-negative space:

\[
\mathbf{u}_i \leftarrow \left[ \mathbf{E}_i \odot \mathbf{D}_i \right]_+.
\]
where $E$ contains the following expression:

$$
E_i = \left( XUS^T \right)_i - \left( USU^T US^T \right)_i + U_i (SU^T)_i (US^T)_i + (X^T US)_i - \left( US^T U^T US \right)_i + U_i (S^T U^T)_{i, (US)}_i
$$

Non-block-wise formulation is inefficient for collective matrix factorization, because it needs to contain zero padding for missing datasets as well as large structure for block-diagonal matrix $U$. We convert the update rule to block-wise formulation. The difference is that with block-wise formulation we only operate on a single dataset at a time, but it is numerically equivalent to non-block-wise. The resulting update rule for $U^{(I)}_i$ is in the following form, where we also project values into non-negative space:

$$
U^{(I)}_i \leftarrow \left[ E^{(I)}_i \odot D^{(I)}_i \right]_+, 
$$

and $E^{(I)}_i$ contains the following expression, converted into block-matrix form:

$$
E^{(I)}_i = \left( X^{(I,J)} U^{(I)} S^{(I,J)^T} \right)_i - \left( U^{(I)} S^{(I,J)} U^{(I)^T} U^{(I)^T} S^{(I,J)^T} \right)_i + U^{(I)}_i (S^{(I,J)^T} U^{(I)^T})_i (U^{(I)^T} S^{(I,J)^T})_i + U^{(I)}_i S^{(I,J)^T} U^{(I)^T} (U^{(I)^T} S^{(I,J)^T})_i - \left( U^{(I)} S^{(I,J)^T} U^{(I)^T} (U^{(I)^T} S^{(I,J)^T})_i \right)_i.
$$

and denominator $D^{(I)}_i$ in block-matrix form:

$$
D^{(I)}_i = (S^{(I,J)^T} U^{(I)})_{i, (U^{(I)^T} S^{(I,J)^T})_i} + (S^{(I,J)^T} U^{(I)^T})_i (U^{(I)^T} S^{(I,J)})_i.
$$
The stopping criteria of matrix factorization relies on computation of objective value (A.11), where $X \in \mathbb{R}^{n \times m}$, $U \in \mathbb{R}^{n \times k_1}$, $S \in \mathbb{R}^{k_1 \times k_2}$, and $V \in \mathbb{R}^{m \times k_2}$. Let’s say that the data matrix is square $n \approx m$ and factorization ranks are equal $k_1 \approx k_2$. The non-optimized implementation of Eq. A.10 has time complexity $O(n^2 k)$ requires an additional memory space for $O(n^2)$ numbers. Using the trace property $\text{Tr}(ABC) = \text{Tr}(BCA) = \text{Tr}(CAB)$, we can reshape the equation (A.11) to:

$$
F = \text{Tr}(X^T X) - \text{Tr}(SV^T X^T U) - \text{Tr}(S^T U^T XV) + \text{Tr}(S^T U^T USV^T V)
$$

(A.45)

Using this optimization and given that we can reuse the following result: $XV$, calculation of objective value requires only additional $O(nk^2)$ operations and $O(k^2)$ of memory. Note that the first term can be efficiently calculated in $O(n^2)$ time and does not change throughout the iterations, therefore it is only calculated once.
BIBLIOGRAPHY


Scalable matrix factorization for data fusion


[76] Xu K et al. (2016) GATA binding protein 2 overexpression is associated with poor prognosis in KRAS mutant colorectal cancer. Oncology Reports 36(3): 1672–1678.


Skalabilna matrična faktorizacija za zlivanje podatkov

Andrej Čopar

doktorska disertacija
predana
Fakulteti za računalništvo in informatiko
kot del izpolnjevanja pogojev za pridobitev naziva
doktor znanosti
s področja
računalništva in informatike

razširjeni pouzetek

Ljubljana, 2019

rabo štirih grafičnih kartic istočasno je naš način več kot stokrat hitrejši v primerjavi s sistemom na enem procesorskem jedru.

Hitrost konvergence matrične tri-faktorizacije smo dosegli z uporabo alternativnih


Paralelizacija nenegativne matrične tri-faktorizacije

Matrična tri-faktorizacija je metoda, ki v primerjavi s klasično dvo-faktorizacijo ne predpostavlja, da podatki ležijo v enem samem skritem prostoru. Faktorizacijski algoritmi so numerično zahtevni, zato je pomembno, da se obstoječe metode prilagodi za delo z velikimi podatkovnimi viri. Razvili smo postopek matrične tri-faktorizacije, ki uporablja pravila bločnega množenja za paralelizacijo na vzporednih sistemih. Iznosimo se, da ima tak postopek dve glavni prednosti: večja učinkovitost na redkih podatkih zaradi boljše uravnoteženosti dela med posameznimi računskimi enotami in zmožnost računanja na več grafičnih enotah istočasno.

Glavni prispevek v povezavi s paralelizacijo je uporaba postopka bločnega množenja na matrični tri-faktorizaciji. Ta postopek je prilagojen za sisteme z več jedri in sisteme z več grafičnimi enotami. Dva obstoječa postopka matrične tri-faktorizacije smo priredili za izvajanje v paralelnih sistemih: neortogonalni ter ortogonalni način tri-faktorizacije. Na podlagi analize na več biomedicinskih podatkovnih virih smo pokazali, da je vzpore-
dni postopek skoraj 200-krat hitrejši od serijske implementacije, kadar ga uporabimo na sistemih s štirimi grafičnimi enotami.

**Konvergenca nenegativne matrične tri-faktorizacije** Matrična tri-faktorizacija trenutno uporablja optimizacijske postopke, ki temeljijo na tehniki multiplikativnih pravil. Uporaba tega postopka je omejena zaradi njegove počasne konvergenco. V tem delu smo opravili raziskavo, ki vključuje šest velikih podatkovnih virov in na njih preverili kakovost naslednjih štirih optimizacijskih postopkov matrične tri-faktorizacije: postopek multiplikativnih pravil, postopek izmenjujočih najmanjših kvadratov, postopek projiciranih gradientov in postopek koordinatnega spusta.

Postopek projiciranih gradientov uporablja prilagojen korak, ki izvede največ pomik v smeri gradieneta, ne da bi vrednosti prešle v negativni prostor. Koordinatni spust deluje na posameznih vektorjih znotraj faktorskih matrik in delne rezultate uporabi znotraj iste iteracije. Izmenjujoči najmanjši kvadrati iterirajo preko treh faktorjev, pri čemer enega posodabljamo preko inverza matrik, druga dva pa ne spreminjamo. Uspešnost teh treh metod na obstoječih postopkih strojnega učenja, predvsem klasične matrične dvo-faktorizacije, nam je dala zamisel za uporabo teh tehnik tudi na metodi tri-faktorizacije. Ugotovili smo, da metode, ki temeljijo na projiciranem gradientu konvergirajo tri-krat hitreje, metode, ki temeljijo na koordinatnem spustu pa konvergirajo tudi do 24-krat hitreje v primerjavi s tehniko multiplikativnih pravil. Ugotovili smo tudi, da je postopek izmenjujočih najmanjših kvadratov hiter, vendar zelo nestabilen in ni praktično uporaben predvsem na manj redkih podatkih z manj ničelnimi vrednostmi. Čas izvajanja metode koordinatnega spusta je skupno 16-krat hitrejši v primerjavi z metodo multiplikativnih pravil.

**Pohitritev zlivanja podatkov** Algoritmi so povečini primerni za delo s homogenimi podatkovnimi viri, ki vsebujejo objekte istega tipa. V praksi so podatkovni viri pogosto heterogeni, kar pomeni da so sestavljeni iz številnih med seboj povezanih objektov. Za pravilno modeliranje takih podatkov potrebujevamo posebne metode. Če želimo iz podatkov o interakcijah med geni in kemikalijami najti skupine genov in kemikalij, običajne metode najdejo skupine genov ter skupine kemikalij, vendar v tem postopku ne znajo uporabiti podatkov o interakcijah različnih skupin. Matrična tri-faktorizacija je zmožna sočasnega grušenja prostora genov in kemikalij ter z uporabo njihovih interakcij izboljšati natančnost modelov. Z dodajanjem novih tipov objektov se poveča komple-
ksnost modeliranja takih podatkovnih zbirk. Za takšne probleme se uporabljajo metode zlivanja podatkov, ki so zmožne sočasnega modeliranja heterogenih podatkov.

Obstoječe metode zlivanja podatkov imajo zelo dobro napovedno točnost na heterogenih podatkovnih zbirkah z veliko različnimi relacijami. Te metode niso namenjene analizi masivnih podatkov ter uporabi na sistemih z visoko stopnjo vzporednega računanja. V tem delu smo omogočili hitro izvajanje metoda za zlivanje podatkov na več-jedrnih sistemih in sistemih z grafičnimi enotami. Izkazalo se je, da je implementacija na grafičnih enotah vsaj 30-krat hitrejša od obstoječe metode ki uporablja 16 procesorskih jader. Za porazdelitev problema na več jedrih ter več grafičnih enotah smo podatke razbili z metodo bločnega razbitja, ki smo jo predhodno že uspešno uporabili za pohitritev matrične tri-faktorizacije.

Poleg tega smo razvili novo metodo za zlivanje podatkov, ki uporablja tehniko koordinatnega spusta, zato da optimizacijski postopek hitreje konvergira v smer rešitve. Rezultati kažejo, da je tak postopek do 17-krat hitrejši v primerjavi z obstoječo metodo. Če združimo paralelizacijo z grafičnimi karticami in tehniko koordinatnega spusta je takšna metoda 80-krat hitrejša od trenutne metode ki je uporabljala 16 procesorskih jader.


**Prispevki v znanosti** Delo predstavljeno v tej disertaciji lahko povzamemo s tremi glavnimi prispevki:

- formulacija pravil NMTF z bločnimi pravili,
- izpeljava pravil NMTF na osnovi koordinatnega spusta,
- skalabilna metoda zlivanja podatkov na podlagi bločnih pravil in koordinatnega spusta.