BlockTensorDecompositions.jl: A Unified Constrained Tensor Decomposition Julia Package

Nicholas J. E. Richardson
Department of Mathematics

Noah Marusenko
Department of Computer Science

Michael P. Friedlander
Departments of Mathematics and Computer Science

Table of contents

1 Introduction

- Tenors are useful in many applications
- Need tools for fast and efficient decompositions

For the scientific user, it would be most useful for there to be a single piece of software that can take as input 1) any reasonable type of factorization model and 2) constraints on the individual factors, and produce a factorization. Details like what rank to select, how the constraints should be enforced, and convergence criteria should be handled automatically, but customizable to the knowledgable user. These are the core specification for BlockTensorDecompositions.jl.

1.1 Related tools

- Packages within Julia

- Other languages
- Hint at why I developed this

Beyond the external usefulness already mentioned, this package offers a playground for fair comparisons of different parameters and options for performing tensor factorizations across various decomposition models. There exist packages for working with tensors in languages like Python (TensorFlow [1], PyTorch [2], and TensorLy [3]), MATLAB (Tensor Toolbox [4]), R (rTensor [5]), and Julia (TensorKit.jl [6], Tullio.jl [7], OMEinsum.jl [8], and TensorDecompositions.jl [9]). But they only provide a groundwork for basic manipulation of tensors and the most common tensor decomposition models and algorithms, and are not equipped to handle arbitrary user defined constraints and factorization models.

Some progress towards building a unified framework has been made [10–12]. But these approaches don't operate on the high dimensional tensor data natively and rely on matricizations of the problem, or only consider nonnegative constraints. They also don't provide an all-in-one package for executing their frameworks.

1.2 Contributions

- Fast and flexible tensor decomposition package
- Framework for creating and performing custom
  - tensor decompositions
  - constrained factorization (the what)
  - iterative updates (the how)
- Implement new "tricks"
  - a (Lipschitz) matrix step size for efficient sub-block updates
  - multi-scaled factorization when tensor entries are discretizations of a continuous function
  - partial projection and rescaling to enforce linear constraints (rather than Euclidean projection)
- ?? rank detection ??

The main contribution is a description of a fast and flexible tensor decomposition package, along with a public implementation written in Julia: BlockTensorDecompositions.jl. This package provides a framework for creating and performing custom tensor decompositions. To the author's knowledge, it is the first package to provide automatic factorization to a large class of constrained tensor decompositions problems, as well as a framework for implementing new constraints and iterative algorithms. This paper also describes three new techniques not found in the literature that empirically convergence faster than traditional block-coordinate descent.

2 Tensor Decompositions

- the math section of the paper

This section reviews the notation used throughout the paper and commonly used tensor decompositions.

2.1 Notation

- tensor notation, use MATLAB notation for indexing so subscripts can be used for a sequence of tensors

2.1.a Sets

The set of nonnegative numbers is denoted as $\mathbb{R}_+ = \mathbb{R}_{\geq 0} = \{x \in \mathbb{R} \,|\, x \geq 0\}$.

We use $[N] = \{1, 2, ..., N\} = \{n\}_{n=1}^{N}$ to denote integers from 1 to N.

Usually, lower case symbols will be used for the running index, and the capitalized letter will be the maximum letter it runs to. This leads to the convenient shorthand $i \in [I]$, $j \in [J]$, etc.

We use a capital delta Δ to denote sets of vectors or higher order tensors where the slices or fibres along a specified dimension sum to 1 i.e. generalized simplexes.

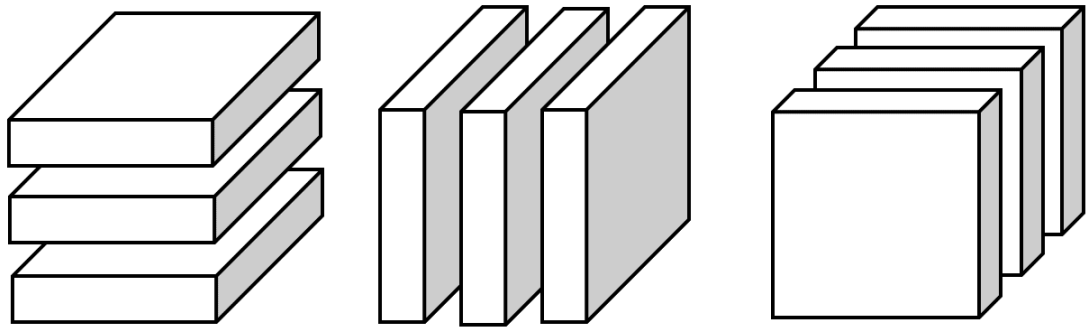
Usually, we use script letters ($\mathcal{A}, \mathcal{B}, \mathcal{C}, etc.$) for other sets.

2.1.b Vectors, Matrices, and Tensors

Vectors are denoted with lowercase letters (x, y, etc.), and matrices and higher order tensors with uppercase letters (commonly A, B, C and X, Y, Z). The order of a tensor is the number of axes it has. We would call vectors "order-1" or "1st order" tensors, and matrices "order-2" or "2nd order" tensors.

To avoid confusion between entries of a vector/matrix/tensor and indexing a list of objects, we use square brackets to denote the former, and subscripts to denote the later. For example, the entry in the ith row and jth column of a matrix $A \in \mathbb{R}$ is $A[i, j]$. This follows MATLAB/Julia notation where `A[i,j]` points to the entry $A[i, j]$. We contrast this with a list of I objects being denoted as $a_1, ..., a_I$, or more compactly, $\{a_i\}$ when it is clear the index $i \in [I]$.

The n-slices, nth mode slices, or mode n slices of an Nth order tensor A are notated with the slice $A[:, ..., :, i_n, :, ..., :]$. For a 3rd order tensor A, the 1st, 2nd, and 3rd mode slices $A[i, :, :]$, $A[:, j, :]$, and $A[:, :, k]$ have special names and are called the horizontal, lateral, and frontal slices and are displayed in Figure 1. In Julia, the 1-, 2-, and 3-slices of a third order array `A` would be `eachslice(A, dims=1)`, `eachslice(A, dims=2)`, and `eachslice(A, dims=3)`.

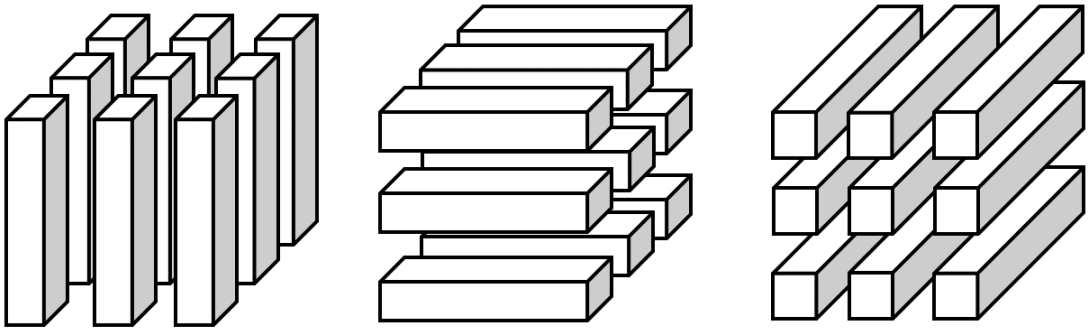


(a) horizontal slices *A[i, :, :]* (b) lateral slices *A[:, j, :]* (c) frontal slices *A[:, :, k]*

Figure 1: Slices of an order 3 tensor A.

The n-fibres, nth mode fibres, or mode n fibres of an Nth order tensor A are denoted $A[i_1, ..., i_{n-1}, :, i_{n+1}, ..., i_N]$. For example, the 1-fibres of a matrix M are the column vectors $M[:, j]$, and the 2-fibres are the row vectors $M[i, :]$. For order-3 tensors, the 1st, 2nd, and 3rd mode fibres $A[:, j, k]$, $A[i, :, :]$, and $A[i, j, :]$ are called the vertical/column, horizontal/row, and depth/tube fibres respectively and are displayed in Figure 2. Natively in Julia, the 1-, 2-, and 3-fibres of a third order array `A` would be `eachslice(A, dims=(2,3))`, `eachslice(A, dims=(1,3))`, and `eachslice(A, dims=(1,2))`. BlockTensorDecomposition.jl defines the function `eachfibre(A; n)` to do exactly this. For example, the 1-fibres of an array `A` would be `eachfibre(A, n=1)`.

For matrices, the 1-fibres are the same as the 2-slices (and vice versa), but for Nth order tensors in general, fibres are always vectors, whereas n-slices are $(N-1)$th order tensors.



(a) vertical fibres *A[:, j, k]* (b) horizontal fibres *A[i, :, k]* (c) depth fibres *A[i, j, :]*

Figure 2: Fibres of an order 3 tensor A.

Since we commonly use I as the size of a tensor's dimension, we use id_I to denote the identity tensor of size I (of the appropriate order). When the order is 2, id_I is an $I \times I$ matrix with ones along the main diagonal, and zeros elsewhere. For higher orders N, this is an $\underbrace{I \times \cdots \times I}_{N \text{ times}}$ tensor where $\mathrm{id}_I[i_1, ..., i_N] = 1$ when $i_1 = ... = i_N \in [I]$, and is zero otherwise.

BlockTensorDecomposition.jl defines `identity_tensor` ### Operations

The Frobenius inner product between two tensors $A, B \in \mathbb{R}^{I_1 \times ... \times I_N}$ is denoted

$$\langle A, B \rangle = A \cdot B = \sum_{i_1=1}^{I_1} ... \sum_{i_N=1}^{I_N} A[i_1, ..., i_N] B[i_1, ..., i_N].$$

Julia's standard library package LinearAlgebra implements the Frobenius inner product with `dot(A, B)` or `A ⋅ B`.

2.2 Common Decompositions

- Extensions of PCA/ICA/NMF to higher dimensions
- talk about the most popular Tucker, Tucker-n, CP
- other decompositions
  - high order SVD (see Kolda and Bader)

- HOSVD (see Kolda, Shifted power method for computing tensor eigenpairs)

2.3 Tensor rank

- tensor rank
- constrained rank (nonnegative etc.)

3 Computing Decompositions

- Given a data tensor and a model, how do we fit the model?

3.1 Optimization Problem

- Least squares (can use KL, 1 norm, etc.)

3.2 Base algorithm

- Use Block Coordinate Descent / Alternating Proximal Descent
  - do *not* use alternating least squares (slower for unconstrained problems, no closed form update for general constrained problems)

4 Techniques for speeding up convergences

- As stated, algorithm works
- But can be slow, especially for constrained or large problems

4.1 Sub-block Descent

- Use smaller blocks, but descent in parallel (sub-blocks don't wait for other sub-blocks)
- Can perform this efficiently with a "matrix step-size"

4.2 Momentum

- This one is standard
- Use something similar to [10]
- This is compatible with sub-block descent with appropriately defined matrix operations

4.3 Partial Projection and Rescaling

- for bounded linear constraints
  - first project
  - then rescale to enforce linear constraints
- faster to execute then a projection
- often does not loose progress because of the rescaling (decomposition dependent)

4.4 Multi-scale

- use a coarse discretization along continuous dimensions
- factorize
- linearly interpolate decomposition to warm start larger decompositions

5 Conclusion

- all-in-one package

- provide a playground to invent new decompositions
- like auto-diff for factorizations

Bibliography

[1] Martín Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015. https://www.tensorflow.org/

[2] J. Ansel *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, Apr. 2024, vol. 2, pp. 929–947. doi: 10.1145/3620665.3640366.

[3] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "TensorLy: Tensor Learning in Python," *Journal of Machine Learning Research*, vol. 20, no. 26, pp. 1–6, 2019, Accessed: Aug. 16, 2024. [Online]. Available: http://jmlr.org/papers/v20/18-277.html

[4] B. W. Bader and T. G. Kolda, "Tensor Toolbox for MATLAB." Sep. 2023.

[5] J. Li, J. Bien, and M. T. Wells, "rTensor: An R Package for Multidimensional Array (Tensor) Unfolding, Multiplication, and Decomposition," *Journal of Statistical Software*, vol. 87, pp. 1–31, Nov. 2018, doi: 10.18637/jss.v087.i10.

[6] Jutho, "Jutho/TensorKit.jl," Aug. 2024. https://github.com/Jutho/TensorKit.jl (accessed Aug. 15, 2024).

[7] M. Abbott *et al.*, "mcabbott/Tullio.jl: v0.3.7," Oct. 2023. https://doi.org/10.5281/zenodo.10035615

[8] A. Peter, "under-Peter/OMEinsum.jl," Aug. 2024. https://github.com/under-Peter/OMEinsum.jl (accessed Aug. 16, 2024).

[9] Y.-J. Wu, "yunjhongwu/TensorDecompositions.jl," Feb. 2024. https://github.com/yunjhongwu/TensorDecompositions.jl (accessed Aug. 16, 2024).

[10] Y. Xu and W. Yin, "A Block Coordinate Descent Method for Regularized Multiconvex Optimization with Applications to Nonnegative Tensor Factorization and Completion," *SIAM Journal on Imaging Sciences*, vol. 6, no. 3, pp. 1758–1789, Jan. 2013, doi: 10.1137/120887795.

[11] J. Kim, Y. He, and H. Park, "Algorithms for nonnegative matrix and tensor factorizations: a unified view based on block coordinate descent framework," *Journal of Global Optimization*, vol. 58, no. 2, pp. 285–319, Feb. 2014, doi: 10.1007/s10898-013-0035-4.

[12] Z. Yang and E. Oja, "Unified Development of Multiplicative Algorithms for Linear and Quadratic Nonnegative Matrix Factorization," *IEEE Transactions on Neural Networks*, vol. 22, no. 12, pp. 1878–1891, Dec. 2011, doi: 10.1109/TNN.2011.2170094.