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 real number is denoted as $\mathbb{R}$ and its restrictions to 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.

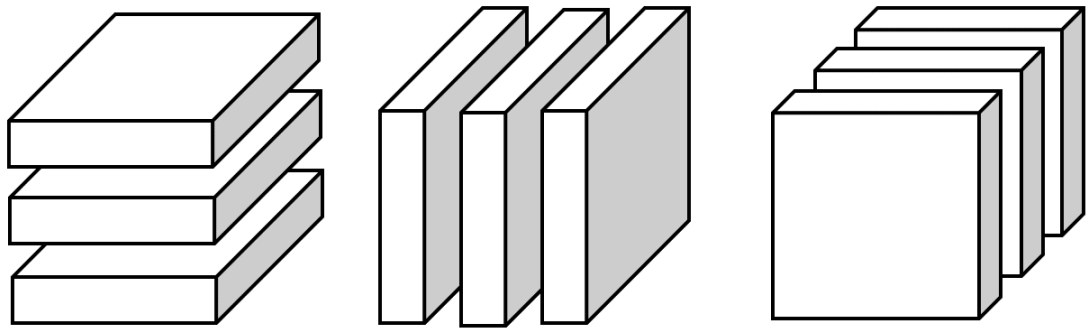
2.1.b Vectors, Matrices, and Tensors

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 transpose $A^\top \in \mathbb{R}^{J \times I}$ of a matrix $A \in \mathbb{R}^{I \times J}$ flips entries along the main diagonal: $A^\top[j, i] = A[i, j]$. In Julia, the transpose of a matrix is typed with a single apostrophe `A'`.

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)`.

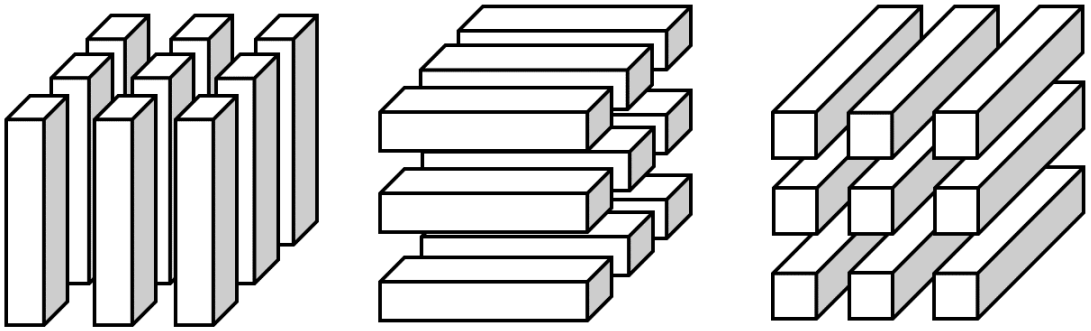


(a) horizontal slices *A[i, :, :]* (b) lateral slices *A[:, j, :]* (c) frontal slices *A[:, :, k]*

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(I, ndims)` to construct id_I.

2.1.c Operations

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

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

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

The n-slice dot product $\cdot_n$ between two tensors $A \in \mathbb{R}^{K_1, \ldots, K_{n-1}, I, K_{n+1}, \ldots, K_N}$ and $B \in \mathbb{R}^{K_1, \ldots, K_{n-1}, J, K_{n+1}, \ldots, K_N}$ returns a matrix $(A \cdot_n B) \in \mathbb{R}^{I \times J}$ with entries

$$(A \cdot_n B)[i, j] = \sum_{k_1 \ldots k_{n-1} k_{n+1} \ldots k_N} A[k_1, \ldots, k_{n-1}, i, k_{n+1}, \ldots, k_N] B[k_1, \ldots, k_{n-1}, j, k_{n+1}, \ldots, k_N].$$

This product can also be thought of as taking the dot product $(A \cdot_n B)[i, j] = A_i \cdot B_j$ between all pairs of nth order slices of A and B, which exactly how BlockTensorDecomposition.jl defines the operation.

```
function slicewise_dot(A::AbstractArray, B::AbstractArray; dims=1)
    C = zeros(size(A, dims), size(B, dims))
    if A === B # use faster routine if they are the same
        return _slicewise_self_dot!(C, A; dims)
    end

    for (i, A_slice) in enumerate(eachslice(A; dims))
        for (j, B_slice) in enumerate(eachslice(B; dims))
            C[i, j] = A_slice ⋅ B_slice
        end
    end
    return C
end

function _slicewise_self_dot!(C, A; dims=1)
    enumerated_A_slices = enumerate(eachslice(A; dims))
    for (i, Ai_slice) in enumerated_A_slices
        for (j, Aj_slice) in enumerated_A_slices
            if i > j
                continue
            else # only compute the upper triangle entries of C
                C[i, j] = Ai_slice ⋅ Aj_slice
            end
        end
    end
    return Symmetric(C) # indexing C[2,1] points to the entry in C[1,2]
end
```

BlockTensorDecomposition.jl defines this operation with `slicewise_dot(A, B, n)`. In the special case where $A = B$, a more efficient method that only computes entries where $i \leq j$ is defined since $A \cdot_n A$ is a symmetric matrix.

The n-mode product $\times_n$ between a tensor $A \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ and matrix $B \in \mathbb{R}^{I_n \times J}$, returns a tensor $(A \times_n B) \in \mathbb{R}^{I_1 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N}$ with entries

$$(A \times_n B)[i_1, \ldots, i_{n-1}, j, i_{n+1}, \ldots, i_N] = \sum_{i_n=1}^{I_n} A[i_1, \ldots, i_{n-1}, i_n, i_{n+1}, \ldots, i_N] B[i_n, j].$$

BlockTensorDecomposition.jl defines this operation with `nmode_product(A, B, n)`.

```julia
function nmode_product(A::AbstractArray, B::AbstractMatrix, n::Integer)
    # convert the problem to the mode-1 product
    Aperm = swapdims(A, n)
    Cperm = Aperm ×₁ B
    return swapdims(Cperm, n) # swap back
end

function ×₁(A::AbstractArray, B::AbstractMatrix)
    # Turn the 1-mode product into matrix-matrix multiplication
    sizeA = size(A)
    Amat = reshape(A, sizeA[1], :)

    # Initialize the output tensor
    C = zeros(size(B, 1), sizeA[2:end]...)
    Cmat = reshape(C, size(B, 1), prod(sizeA[2:end]))

    # Perform matrix-matrix multiplication Cmat = B*Amat
    mul!(Cmat, B, Amat)

    return C # Output entries of Cmat in tensor form
end

function swapdims(A::AbstractArray, a::Integer, b::Integer=1)
    # Construct a permutation where a and b are swapped
    # e.g. [4, 2, 3, 1, 5, 6] when a=4 and b=1
    dims = collect(1:ndims(A))
    dims[a] = b; dims[b] = a
    return permutedims(A, dims)
end
```

> **Note**
>
> If we were only working with a fixed order of tensors, we could have defined `×₁` entry-wise with `Tullio.jl`. The function definition `tullio×₁` below gives an example for order three tensors.
>
> ```
> function tullio×₁(A::AbstractArray{_,3}, B::AbstractMatrix)
>   @tullio C[i, j, k] := A[r, j, k] * B[i, r]
>   return C
> end
> ```
>
> But we would need a new definition for each ordered tensor, or use Julia's meta programming to write a method for each order at runtime.

The Frobenius norm of a tensor A is the square root of its dot product with itself

$$\|A\|_F = \sqrt{\langle A, A \rangle}.$$

For vectors v, this is equivalent to the (Euclidean) 2-norm

$$\|v\|_F = \|v\|_2 = \sqrt{\langle v, v \rangle}.$$

For matrices M, the (Operator) 2-norm is defined as

$$\|M\|_2 = \arg\max_{\|v\|_2=1} \|Mv\|_2 = \sigma_1(M)$$

where $\sigma_1(M)$ is the largest singular value of M.

For tensors T, the (Operator) 2-norm needs to be defined in terms of how we treat them as function on other tensors. There is a canonical way to do this for vectors $x \to v^\top x$ and matrices $x \to Mx$, but not tensors. This is relevant to Section 3.2.c where the Lipschitz step-size is computed in terms of the Operator norm of the Hessian of our objective function.

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)

A tensor decomposition is a factorization of a tensor into multiple (usually smaller) tensors, that can be recombined into the original tensor. To make a common interface for decompositions, we make an abstract subtype of Julia's `AbstractArray`, and subtype `AbstractDecomposition` for our concrete tensor decompositions.

```
abstract type AbstractDecomposition{T, N} <: AbstractArray{T, N} end
```

Computationally, we can think of a generic decomposition as storing factors $(A, B, C, ...)$ and operations $(\times_a, \times_b, ...)$ for combining them. This is what we do in BlockTensorDecomposition.jl.

```
struct GenericDecomposition{T, N} <: AbstractDecomposition{T, N}
    factors::Tuple{Vararg{AbstractArray{T}}} # e.g. (A, B, C)
    contractions::Tuple{Vararg{Function}} # e.g. (×₁, ×₂)
end
# Y = A ×₁ B ×₂ C
array(G::GenericDecomposition) = multifoldl(contractions(G), factors(G))
```

The function `multifoldl` applies the given operations between each factor, from left to right.

```
function multifoldl(ops, args)
    @assert (length(ops) + 1) == length(args)
    x, xs... = args
    for (op, arg) in zip(ops, xs)
        x = op(x, arg)
    end
    return x
end
```

Different types of decompositions define different operations, and different "ranks" of the same decomposition specific the sizes of the factors used.

A commonly used family of decompositions can be derived from the Tucker decomposition.

Definition 2.1: A rank-$(R_1, ..., R_N)$ Tucker decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times ... \times I_N}$ produces N matrices $A_n \in \mathbb{R}^{I_n \times R_n}$, $n \in [N]$, and core tensor $B \in \mathbb{R}^{R_1 \times ... \times R_N}$ such that

$$Y[i_1, ..., i_N] = \sum_{r_1=1}^{R_1} ... \sum_{r_N=1}^{R_N} A_1[i_1, r_1] \cdots A_r[i_N, r_N] B[r_1, ..., r_N] \tag{1}$$

entry-wise. More compactly, this decomposition can be written using the n-mode product, or with double brackets

$$Y = B \times_1 A_1 \times_2 ... \times_N A_N = B \bigtimes_n A_n = [\![B; A_1, ..., A_N]\!].$$

Sometimes we write $A_0 = B$ to ease notation, and suggest the "zeroth" factor of the tucker decomposition is the core tensor B. In the special case when $N = 3$, we can visualize Tucker decomposition as multiplying the core tensor by matrices on all three sides as shown in Figure 3.

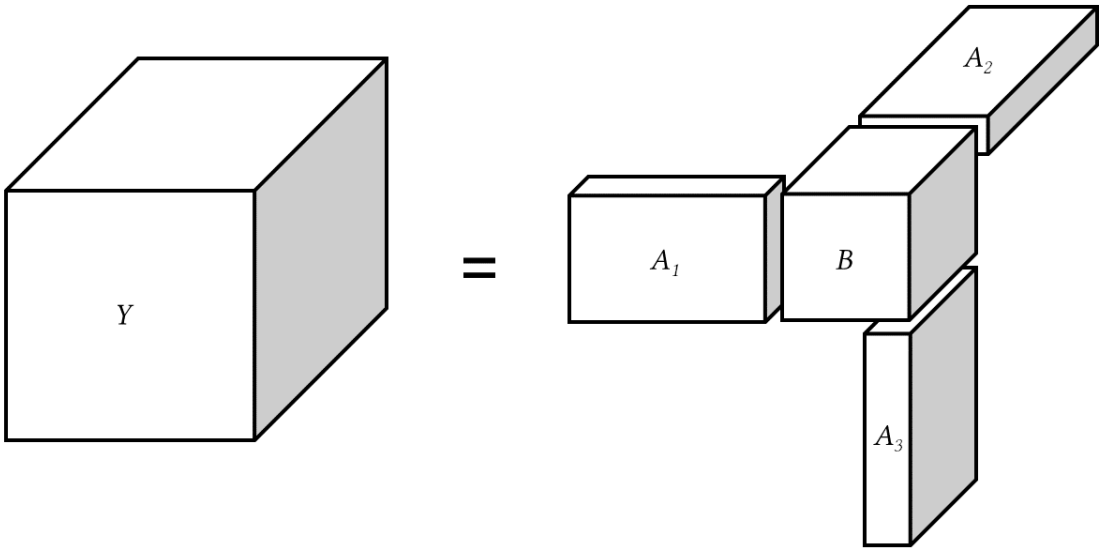


Figure 3: Tucker factorization of a 3rd order tensor Y.

Setting all the matrices of a Tucker decomposition to the identity matrix but the first gives the Tucker-1 decomposition.

Definition 2.2: A rank-R Tucker-1 decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ produces a matrix $A \in \mathbb{R}^{I_1 \times R}$, and core tensor $B \in \mathbb{R}^{R \times I_2 \times \ldots \times I_N}$ such that

$$Y[i_1, ..., i_N] = \sum_{r=1}^{R} A[i_1, r]B[r, i_2, ..., i_N] \tag{2}$$

entry-wise or more compactly,

$$Y = AB = B \times_1 A = [\![B; A]\!].$$

Note we extend the usual definition of matrix-matrix multiplication

$$(AB)[i, j] = \sum_{r=1}^{R} A[i, r]B[r, j]$$

to tensors B in the compact notation for Tucker-1 decomposition $Y = AB$.

More generally, any number of matrices can be set to the identity matrix giving the Tucker-n decomposition.

Definition 2.3: A rank-$(R_1, ..., R_n)$ Tucker-n decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times ... \times I_N}$ produces n matrices $A_1, ..., A_n$, and core tensor $B \in \mathbb{R}^{R_1 \times ... \times R_n \times I_{n+1} \times ... \times I_N}$ such that

$$Y[i_1, ..., i_N] = \sum_{r_1=1}^{R_1} ... \sum_{r_N=1}^{R_n} A_1[i_1, r_1] \cdots A_n[i_N, r_n] B[r_1, ..., r_n, i_{n+1}, ..., i_N] \tag{3}$$

entry-wise, or compactly written in the following three ways,

$$\begin{aligned} Y &= B \times_1 A_1 \times_2 ... \times_n A_n \times_{n+1} \mathrm{id}_{I_{n+1}} \times_{n+2} ... \times_N \mathrm{id}_{I_N} \\ Y &= B \times_1 A_1 \times_2 ... \times_n A_n \\ Y &= [\![B; A_1, ..., A_n]\!]. \end{aligned}$$

Lastly, if we set the core tensor B to the identity tensor id_R, we obtain the **can**onical **decomp**osition/**para**llel **fac**tors model (CANDECOMP/PARAFAC or CP for short).

Definition 2.4: A rank-R CP decomposition of a tensor $Y \in \mathbb{R}^{I_1 \times ... \times I_N}$ produces N matrices $A_n \in \mathbb{R}^{I_n \times R}$, such that

$$Y[i_1, ..., i_N] = \sum_{r=1}^{R} A_1[i_1, r] \cdots A_r[i_N, r] \tag{4}$$

entry-wise. More compactly, this decomposition can be written using the n-mode product, or with double brackets

$$Y = \mathrm{id}_R \times_1 A_1 \times_2 ... \times_N A_N = \mathrm{id}_R \bigtimes_n A_n = [\![A_1, ..., A_N]\!].$$

Note CP decomposition is sometimes referred to as Kruskal decomposition, and requires the core only be diagonal (and not necessarily identity) and the factors A_n have normalized columns $\|A_n[:, r]\|_2 = 1$.

Other factorization models are used that combine aspects of CP and Tucker decomposition [13], are specialized for order 3 tensors [14, 15], or provide alternate decomposition models entirely like tensor-trains [16]. But the (full) Tucker, and its special cases Tucker-n, and CP decomposition are most commonly used extensions of the low-rank matrix factorization to tensors. These factorizations are summarized in Table 1.

Table 1: Summary of common tensor factorizations. Here, N is the order of the factorized tensor.

Name	Bracket Notation	n-mode Product	Entry-wise
Tucker	$[\![A_0; A_1, ..., A_N]\!]$	$A_0 \times_1 A_1 \times_2 ... \times_N A_N$	Equation 1
Tucker-1	$[\![A_0; A_1]\!]$	$A_0 \times_1 A_1$	Equation 2
Tucker-n	$[\![A_0; A_1, ..., A_n]\!]$	$A_0 \times_1 A_1 \times_2 ... \times_n A_n$	Equation 3
CP	$[\![A_1, ..., A_N]\!]$	$\mathrm{id}_R \times_1 A_1 \times_2 ... \times_N A_N$	Equation 4

TODO add discussion on other decompositions - high order SVD (see Kolda and Bader) - HOSVD (see Kolda, Shifted power method for computing tensor eigenpairs)

Tensor decompositions are not nessisarily unique. It should be clear that scaling one factor by $x \neq 0$ and dividing another by x yields the same original tensor. Furthermore, fibres and slices can be permuted without affecting the the original tensor. Up to these manipulations, for a fixed rank, there exist criteria that ensures their decompositions are unique [13, 17, 18].

2.2.a Representing Tucker Decompositions

There are implemented in BlockTensorDecomposition.jl and can be called, for a third order tensor, with `Tucker((B, A₁, A₂, A₃))`, `Tucker1((B, A₁))`, and `CPDecomposition((A₁, A₂, A₃))`. These Julia `structs` store the tensor in its factored form. We could define the contractions for these types and use the common interface provided by `array`, but it turns out we can reconstruct the whole tensor more efficiently. If the recombined tensor or particular entries are requested, Julia dispatches on the type of decomposition and calls a particular method of `array` or `getindex`. The implementations for efficient array construction and index access are provided below.

```
array(T::Tucker) = multifoldl(tucker_contractions(ndims(T)), factors(T))
tucker_contractions(N) = Tuple((G, A) -> nmode_product(G, A, n) for n in 1:N)
```

TODO add getindex method for Tucker type

```
function array(T::Tucker1)
    B, A = factors(T)
    return B ×₁ A
end

function getindex(T::Tucker1, I::Vararg{Int})
    B, A = factors(T)
    i, J... = I # (i, J) = (I[1], I[begin+1:end])
    return (@view A[i, :]) · view(B, :, J...)
end
```

```
array(CPD::CPDecomposition) =
  mapreduce(vector_outer, +, zip((eachcol.(factors(CPD)))...))
```

```
vector_outer(v) = reshape(kron(reverse(v)...),length.(v))

getindex(CPD::CPDecomposition, I::Vararg{Int}) =
  sum(reduce(.*, (@view f[i,:]) for (f,i) in zip(factors(CPD), I)))
```

2.3 Tensor rank

- tensor rank
- constrained rank (nonnegative etc.)

The rank of a matrix $Y \in \mathbb{R}^{I \times J}$ can be defined as the smallest $R \in \mathbb{Z}_+$ such that there exists an exact factorization $Y = AB$ for some $A \in \mathbb{R}^{I \times R}$ and $B \in \mathbb{R}^{R \times J}$.

Although this can be extended to higher order tensors, we must specify under which factorization model we are using. For example, the *CP-rank* R of a tensor Y is the smallest such R that omits an exact CP decomposition of Y.

Definition 2.5: The CP rank of a tensor $Y \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ is the smallest R such that there exist factors $A_n \in \mathbb{R}^{I_n \times R}$ and $Y = [\![A_1, \ldots, A_N]\!]$,

$$\operatorname{rank}_{\text{CP}}(Y) = \min\{R \,|\, \exists A_n \in \mathbb{R}^{I_n \times R}, n \in [N] \quad \text{s.t.} \quad Y = [\![A_1, \ldots, A_N]\!]\}.$$

In a similar way, we can define the *Tucker-1-rank* R.

Definition 2.6: The Tucker-1 rank of a tensor $Y \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ is the smallest R such that there exist factors $A \in \mathbb{R}^{I_1 \times R}$ and $B \in \mathbb{R}^{R \times I_2 \times \ldots \times I_N}$ where $Y = AB$

$$\operatorname{rank}_{\text{Tucker-1}}(Y) = \min\{R \,|\, \exists A_n \in \mathbb{R}^{I_n \times R}, B \in \mathbb{R}^{R \times I_2 \times \ldots \times I_N} \quad \text{s.t.} \quad Y = AB\}$$

For the Tucker and Tucker-n decompositions, we instead call a particular factorization **a** rank-$(R_1, \ldots, R_N)$ Tucker factorization or **a** rank-$(R_1, \ldots, R_n)$ Tucker-n factorization, rather than **the** CP- or Tucker-1-rank of a tensor or **the** rank of a matrix.

One reason CP and Tucker-1 only need a single rank R can be explained by considering the case when the order of the tensor $N = 2$ (matrices). The two factorizations become equivalent and are equal to low-rank R matrix factorization $Y = AB$. In fact, Tucker-1 is always equivalent to a low-rank matrix factorization, if you consider a flattening of the tensor to arrange the entries as a matrix.

The idea of tensor rank can be generalized further to constrained rank. These are the smallest rank R such that the factors in the decomposition obey the given set of constraints.

For example, the nonnegative Tucker-1 rank is defined as

$$\text{rank}^{+}_{\text{Tucker-1}}(Y) = \min\Big\{R \;\Big|\; \exists A_n \in \mathbb{R}_{+}^{I_n \times R}, B \in \mathbb{R}_{+}^{R \times I_2 \times \ldots \times I_N} \quad \text{s.t.} \quad Y = AB\Big\}.$$

More restrictive constraints increase the rank of the tensor since there is less freedom in selecting the factors.

Most tensor decomposition algorithms require the rank as input [CITE] since calculating the rank of the tensor can be NP-hard in general [19]. For applications where the rank is not known a priori, a common strategy is to attempt a decomposition for a variety of ranks, and select the model with smallest rank that still achieves good fit between the factorization and the original tensor.

3 Computing Decompositions

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

Many tensor decompositions algorithms exist in the literature. Usually, they cyclically (or in a random order) update factors until their reconstruction satisfies some convergence criterion. The base algorithm described in Section 3.2 provides flexible framework for wide class of constrained tensor factorization problems. This framework was selected based on empirical observations where it outperforms other similar algorithms, and has also been observed in the literature [10].

3.1 Optimization Problem

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

Ideally, we would be given a data tensor Y and decomposition model, and compute an exact factorization of Y into its factors. Because there is often measurement, numerical, or modeling error, an exact factorization of Y for a particular rank may not exist. To over come this, we instead try to fit the model to the data. Let X be the reconstruction of factors $A_1, ..., A_N$ according to some decomposition for a fixed rank. We assume we know the size of the factors $A_1, ..., A_N$ and how they are combined to produce a tensor the same size of Y, i.e. the map $g : (A_1, ..., A_N) \mapsto X$.

There are many loss functions that can be used to determine how close the model X is to the data Y. In principle, any distance or divergence $d(Y, X)$ could be used. We use the L_2 loss or least-squares distance between the tensors $\|X - Y\|_F^2$, but other losses are used for tensor decomposition in practice such as the KL divergence [CITE].

The main optimization we must solve is now given.

Definition 3.1: The constrained least-squares tensor factorization problem is to solve

$$\min_{A_1,...,A_N} \frac{1}{2}\|g(A_1, ..., A_N) - Y\|_F^2 \quad \text{s.t.} \quad (A_1, ..., A_N) \in \mathcal{C}_1 \times ... \times \mathcal{C}_N \tag{5}$$

for a given data tensor Y, constraints $\mathcal{C}_1, ..., \mathcal{C}_N$, and decomposition model g with fixed rank.

Note the problem would have the same solutions as simply using the objective $\|g(A_1, ..., A_N) - Y\|$ without squaring and dividing by 2. We define the objective in Equation 5 to make computing the function value and gradients faster.

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)

Let $f(A_1, ..., A_N) := \frac{1}{2}\|g(A_1, ..., A_N) - Y\|_F^2$ be the objective function we wish to minimize in Equation 5. Following Xu and Yin [10], the general approach we take to minimize f is to apply block coordinate descent using each factor as a different block. Let A_n^t be the tth iteration of the nth factor, and let

$$f_n^t(A_n) := \frac{1}{2}\left\|g\left(A_1^{t+1}, ..., A_{n-1}^{t+1}, A_n, A_{n+1}^t, ..., A_N^t\right) - Y\right\|_F^2$$

be the (partially updated) objective function at iteration t for factor n.

Given initial factors $A_1^0, ..., A_N^0$, we cycle through the factors $n \in [N]$ and perform the update

$$A_n^{t+1} \leftarrow \arg\min_{A_n \in \mathcal{C}_n} \left\langle \nabla f_n^t(A_n^t), A_n - A_n^t \right\rangle + \frac{L_n^t}{2}\left\|A_n - A_n^t\right\|_F^2,$$

for $t = 1, 2, ...$ until some convergence criterion is satisfied (see Section 4.2.a).

This implicit update has the *projected gradient descent* closed form solution for convex constraints $\mathcal{C}_n$,

$$A_n^{t+1} \leftarrow P_{\mathcal{C}_n}\left(A_n^t - \frac{1}{L_n^t}\nabla f_n^t(A_n^t)\right). \tag{6}$$

We typically choose L_n^t to be the Lipschitz constant of ∇f_n^t, since it is a sufficient condition to guarantee $f_n^t(A_n^{t+1}) \leq f_n^t(A_n^t)$, but other step sizes can be used in theory [20 (Sec. 1.2.3)].

?ASIDE? To write ∇f_n^t, we have assumed (block) differentiability of the decomposition model g. In practice, most decompositions are "block-linear" (freeze all factors but one and you have a linear function) and in rare cases are "block-affine". "block-affine" is enough to ensure f_n^t is convex (i.e. f is "block-convex") so the updates Equation 6 converge to a Nash equilibrium (block minimizer).

3.2.a High level code

To ensure the code stays flexible, the main algorithm of BlockTensorDecomposition.jl, `factorize`, is defined at a very high level.

```
factorize(Y; kwargs...) =
    _factorize(Y; (default_kwargs(Y; kwargs...))...)
```

```
"""
Inner level function once keyword arguments are set
"""
function _factorize(Y; kwargs...)
    decomposition, previous, updateprevious!, parameters, updateparameters!,
    update!, stats_data, getstats, converged, kwargs = initialize(Y, kwargs)

    while !converged(stats_data; kwargs...)
        # Usually one cycle of updates through each factor in the decomposition
        update!(decomposition; parameters...)

        # This could be the next stepsize or other info used by update!
        updateparameters!(parameters, decomposition, previous)

        push!(stats_data,
            getstats(decomposition, Y, previous, parameters, stats_data))

        # Update one or two previous iterates. For example, used for momentum
        updateprevious!(previous, parameters, decomposition)
    end

     kwargs = postprocess!(decomposition, Y, previous, parameters, stats_data,
updateparameters!, getstats, kwargs)

    return decomposition, stats_data, kwargs
end
```

The magic of the code is in defining the functions at runtime for a particular decomposition requested, from a reasonable set of default keyword arguments. This is discussed further in Section 4.2.

3.2.b Computing Gradients

- Use Auto diff generally
- But hand-crafted gradients and Lipschitz calculations *can* be faster (e.g. symmetrized slicewise dot product)

Generally, we can use automatic differentiation on f to compute gradients. Some care needs to be taken otherwise the forward or backwards pass will have to be recompiled every iteration since the factors are updated every iteration.

But for Tucker decompositions, we can compute gradients faster than what an automatic differentiation scheme would give, by taking advantage of symmetry and other computational shortcuts.

Starting with the Tucker-1 decomposition (Definition 2.2), we would like to compute $\nabla_B f(B, A)$ and $\nabla_A f(B, A)$ for $f(B, A) = \frac{1}{2}\|AB - Y\|_F^2$ for a given input Y. We have the gradient

$$\nabla_B f(B, A) = A^\top (AB - Y) = (B \times_1 A - Y) \times_1 A^\top \tag{7}$$

by chain rule, but it is more efficient to calculate the gradient as

$$\nabla_B f(B, A) = (A^\top A)B - A^\top Y = B \times_1 (A^\top A) - Y \times_1 A^\top. \tag{8}$$

[1]For $A \in \mathbb{R}^{I \times R}$, $B \in \mathbb{R}^{R \times J \times K}$, and $Y \in \mathbb{R}^{I \times J \times K}$, Equation 7 requires

$$\underbrace{2IJKR}_{AB-Y} + \underbrace{IJK(2I-1)}_{A^\top(AB-Y)} \sim 2IJKR + 2I^2JK$$

floating point operations (FLOPS) whereas Equation 8 only uses

$$\underbrace{\frac{R(R+1)}{2}(2I-1)}_{A^\top A} + \underbrace{RJK(2I-1)}_{A^\top Y} + \underbrace{2R^2JK}_{(A^\top A)B-(A^\top Y)} \sim 2IJKR + 2R^2JK + IR^2$$

FLOPS[2]. So for small ranks $R \ll I$, Equation 8 is cheaper.

A similar story can be said about $\nabla_A f(B, A)$ which is most efficiently computed as

$$\nabla_A f(B, A) = A(B \cdot_1 B) - Y \cdot_1 B.$$

The associated implementation with BlockTensorDecomposition.jl is shown below. We define a `make_gradient` which takes the decomposition, factor index `n`, and data tensor `Y`, and creates a function that computes the gradient for the same type of decomposition. This lets us manipulate the function that computes the gradient, rather than just the computed gradient.

```
function make_gradient(T::Tucker1, n::Integer, Y::AbstractArray; objective::L2,
kwargs...)
    if n==0 # the core is the zeroth factor
        function gradient0(T::Tucker1; kwargs...)
            (B, A) = factors(T)
            AA = A'A
            YA = Y×₁A'
            grad = B×₁AA - YA
            return grad
        end
        return gradient0
    elseif n==1 # the matrix is the first factor
        function gradient1(T::Tucker1; kwargs...)
            (B, A) = factors(T)
            BB = slicewise_dot(B, B)
            YB = slicewise_dot(Y, B)
            grad = A*BB - YB
            return grad
        end
```

[1]Seeing Equation 7 and Equation 8 written using the 1-mode product shows how it is "backwards" to normal matrix-matrix multiplication.

[2]Note we have the smaller factor $R(R+1)/2$ and not the expected R^2 number of entries needed to compute $A^\top A$. The product is a symmetric matrix so only the upper or lower triangle of entries needs to be computed.

```
        return gradient1
    else
        error("No $(n)th factor in Tucker1")
    end
end
```

Similarly, we also have special methods for the Tucker and CP Decomposition.

The gradient with respect to the core for a full Tucker factorization is

$$\nabla_B f(B, A_1, ..., A_N) = B \bigtimes_n A_n^\top A_n - Y \bigtimes_n A_n^\top,$$

and the gradient with respect to the matrix factor A_m is

$$\nabla_{A_m} f(B, A_1, ..., A_N) = A_m \left(B \bigtimes_{n \neq m} A_n \right) \cdot_m \left(B \bigtimes_{n \neq m} A_n \right) - Y \cdot_m \left(B \bigtimes_{n \neq m} A_n \right).$$

```
function make_gradient(T::Tucker, n::Integer, Y::AbstractArray; objective::L2,
kwargs...)
    N = ndims(T)
    if n==0 # the core is the zeroth factor
        function gradient_core(T::AbstractTucker; kwargs...)
            C = core(T)
            matrices = matrix_factors(T)
             gram_matrices = map(A -> A'A, matrices) # gram matrices AA = A'A,
BB = B'B...
            YAB = tuckerproduct(Y, adjoint.(matrices)) # Y ×₁ A' ×₂ B' ...
            grad = tuckerproduct(C, gram_matrices) - YAB
            return grad
        end
        return gradient_core

    elseif n in 1:N # the matrix factors start at m=1
        function gradient_matrix(T::AbstractTucker; kwargs...)
            matrices = matrix_factors(T)
            TExcludeAn = tuckerproduct(core(T), matrices; exclude=n)
            An = factor(T, n)
                  grad = An*slicewise_dot(TExcludeAn, TExcludeAn; dims=n) -
slicewise_dot(Y, TExcludeAn; dims=n)
            return grad
        end
        return gradient_matrix

    else
        error("No $(n)th factor in Tucker")
    end
end
```

```
function make_gradient(T::CPDecomposition, n::Integer, Y::AbstractArray;
objective::L2, kwargs...)
    N = ndims(T)
    if n in 1:N # the matrix factors start at m=1
        function gradient_matrix(T::AbstractTucker; kwargs...)
            matrices = matrix_factors(T)
            TExcludeAn = tuckerproduct(core(T), matrices; exclude=n)
            An = factor(T, n)
                  grad = An*slicewise_dot(TExcludeAn, TExcludeAn; dims=n) -
slicewise_dot(Y, TExcludeAn; dims=n)
            return grad
        end
        return gradient_matrix

    else
        error("No $(n)th factor in Tucker")
    end
end
```

The function `tuckerproduct(B, (A₁, ..., Aₙ))` computes

$$B \bigtimes_n A_n = [\![B; A_1, ..., A_N]\!],$$

and can optionally "exclude" one of the matrix factors `tuckerproduct(B, (A₁, ..., Aₙ); exclude=m)` to compute

$$B \bigtimes_{n \neq m} A_n = \left[\!\!\left[B; A_1, ..., A_{n-1}, \mathrm{id}_{I_n}, A_{n+1}, ..., A_N\right]\!\!\right]$$

where $B \in \mathbb{R}^{I_1 \times ... \times I_N}$.

3.2.c Computing Lipschitz Step-sizes

Similar to automatic differentiation, there exist "automatic Lipschitz" calculations to upper bound the Lipschitz constant of a function [21].

4 Computational Techniques

4.1 For Improving Convergence Speed

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

4.1.a 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"

4.1.b Momentum

- This one is standard

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

4.2 For Flexibility

- there are a number of software engineering techniques used
- these help flexibility for hot swapping and a language for making custom…
  - convergence criterion (and having multiple stopping conditions)
  - probing info during the iterations (stats collected at the end)
  - having multiple constraints and ways to enforce them
  - cyclically or partially randomly or fully randomly update factors
- smart enough to apply these in a reasonable order

4.2.a Convergence Criteria and Stats

- Can request info about any factor at each outer iteration
- any subset of stats can be the convergence criteria

4.2.b `BlockUpdate` Language

- construct the updates as a list of updates
- very functional programming
- can apply them in sequence or in a random order (or partially random)

4.2.c Constraints

- one type of update (other than the typical GD update)
- can combine them with composition
  - which is different than projecting onto their intersection!
- Constraint updates combine the constraint with how they are enforced
  - need to go together since there are multiple ways to enforce them e.g. simplex (see next section)

5 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)

6 Multi-scale

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

7 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.

[13] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, Aug. 2009, doi: 10.1137/07070111X.

[14] L. Qi, Y. Chen, M. Bakshi, and X. Zhang, "Triple Decomposition and Tensor Recovery of Third Order Tensors," Mar. 01, 2020. http://arxiv.org/abs/2002.02259 (accessed Aug. 01, 2023).

[15] F. Wu, C. Li, and Y. Li, "Manifold Regularization Nonnegative Triple Decomposition of Tensor Sets for Image Compression and Representation," *Journal of Optimization Theory and Applications*, vol. 192, no. 3, pp. 979–1000, Mar. 2022, doi: 10.1007/s10957-022-02001-6.

[16] I. V. Oseledets, "Tensor-Train Decomposition," *SIAM Journal on Scientific Computing*, vol. 33, no. 5, pp. 2295–2317, Jan. 2011, doi: 10.1137/090752286.

[17] J. B. Kruskal, "Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics," *Linear Algebra and its Applications*, vol. 18, no. 2, pp. 95–138, Jan. 1977, doi: 10.1016/0024-3795(77)90069-6.

[18] A. Bhaskara, M. Charikar, and A. Vijayaraghavan, "Uniqueness of Tensor Decompositions with Applications to Polynomial Identifiability," in *Proceedings of The 27th Conference on Learning Theory*, May 2014, pp. 742–778. Accessed: Jan. 08, 2025. [Online]. Available: https://proceedings.mlr.press/v35/bhaskara14a.html

[19] S. A. Vavasis, "On the Complexity of Nonnegative Matrix Factorization," *SIAM Journal on Optimization*, vol. 20, no. 3, pp. 1364–1377, Jan. 2010, doi: 10.1137/070709967.

[20] Y. Nesterov, "Nonlinear Optimization," *Lectures on Convex Optimization*. Springer International Publishing, Cham, pp. 3–58, 2018. doi: 10.1007/978-3-319-91578-4_1.

[21] A. Virmaux and K. Scaman, "Lipschitz regularity of deep neural networks: analysis and efficient estimation," in *Advances in Neural Information Processing Systems*, 2018, vol. 31. Accessed: Jan. 11, 2025. [Online]. Available: https://proceedings.neurips.cc/paper/2018/hash/d54e99a6c03704e95e6965532dec148b-Abstract.html