Scalable Tensor Decompositions for Multi-aspect Data Mining

Tamara G. Kolda
Sandia National Laboratories
tgkolda@sandia.gov

Jimeng Sun
IBM TJ Watson Research Center
jimeng@us.ibm.com

Abstract

Modern applications such as Internet traffic, telecommunication records, and large-scale social networks generate massive amounts of data with multiple aspects and high dimensionalities. Tensors (i.e., multi-way arrays) provide a natural representation for such data. Consequently, tensor decompositions such as Tucker become important tools for summarization and analysis.

One major challenge is how to deal with high-dimensional, sparse data. In other words, how do we compute decompositions of tensors where most of the entries of the tensor are zero. Specialized techniques are needed for computing the Tucker decompositions for sparse tensors because standard algorithms do not account for the sparsity of the data. As a result, a surprising phenomenon is observed by practitioners: Despite the fact that there is enough memory to store both the input tensors and the factorized output tensors, memory overflows occur during the tensor factorization process. To address this intermediate blowup *problem, we propose Memory-Efficient Tucker (MET). Based on the available memory, MET adaptively selects the right execution strategy during the decomposition. We provide quantitative and qualitative evaluation of MET on real tensors. It achieves over 1000X space reduction without sacrificing speed; it also allows us to work with much larger tensors that were too big to handle before. Finally, we demonstrate a data mining case-study using MET.*

1 Introduction

Many applications generate large amounts high dimensional data with multiple aspects, which are naturally represented as tensors, or multi-arrays. Some examples include (a) email exchanges, (b) bibliographic data, (c) hyperlinks on the web, and (d) Internet network traffic flows. These examples can be naturally represented as tensors as follows: (a) can be modeled as a 4D array, or a fourth-order tensor, with sender, recipeient, keyword, and datestamp as the four modes; (b) can viewed as a author-conference-keyword third-order tensor [19]; (c) hyperlinks on the web yield a third-order tensor of sources by desinations by anchor text [15]; and (d) network traffic is a fourth-order tensor with source IP, destination IP, port number, and time [18, 19].

Beyond being multi-aspect and high-dimensional, another key characteristic associated with these examples is *sparsity*, meaning that most of entries in the tensor are zeros. Sparsity is a common property in tensor data, e.g., Table 1 illustrates the density of datasets used in this paper. Therefore, it is much more efficient to store the nonzeros only; in fact, very large sparse tensors can be stored using only moderate hardware. For example, a 10K-by-10K-by-10K tensor with 1 million nonzeros out of 1 trillion elements can be formed in a personal laptop with only 32MB memory[1]. However, the challenge is, given a large sparse tensor, how can it be efficiently analyzed with fixed memory?

Name	Enron	DBLP	Web	Network
density	.0025%	.0187%	.0008%	.0286%

Table 1. Sparsity in the real datasets

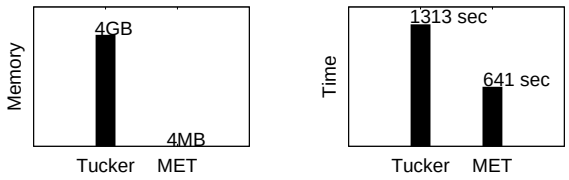
To analyze such tensor data, various tensor decompositions are proposed in the literature (see [14]). The Tucker decomposition [21] is has been applied in many different domains such as web search mining [20], network forensics [19] and social network [8] (see Section 6 for an overview).

Despite its popularity, how to apply Tucker on a large sparse tensor is still an open problem. One surprising phenomenon is observed by practitioners: Despite that both the input (huge, sparse) tensor and the output (small, dense, factorized) tensor can be stored in memory, memory overflows may occur during the

[1]This is measured in the Tensor Toolbox using the sparse tensor object [5].

Tucker decomposition process. We call this the *intermediate blowup* problem. Due to this problem, up to now most of work on Tucker has been focused on relatively small tensors (e.g., of the order of 100-by-100-by-100 or smaller).

To address the intermediate blowup problem, we propose a Memory-Efficient Tucker decomposition (MET), which maximizes the computation speed while optimally utilizing the available memory. MET avoids constructing large intermediate results by handling the computation in a piecemeal fashion, adaptively selecting the order of operations. Not only does MET eliminate the intermediate blowup problem, it also achieves significant memory savings without any loss of accuracy. In many cases, this can be done without compromising on speed and is sometimes even faster. For example, Figure 1 is an example comparison on 100K-by-100K-by-100K random tensor with 1M nonzeros. Our proposed method MET yields a 1000X saving on memory and is twice as fast as the standard Tucker.



(a) Space consumption (b) CPU time

Figure 1. Comparison of Tucker versus MET on a 10K-by-10K-by-10K tensor with 1M nonzeros

In summary, the contributions of this paper are as follows.

- We demonstrate the severity of the *intermediate blowup* problem in Tucker decomposition for sparse tensors;
- we propose an adaptive algorithm called MET for computing the Tucker decomposition for sparse tensors; and
- we introduce a novel heuristics to prioritize the computation across tensor modes in order to reduce memory consumption.

2 Background

In this section, we introduce the notation, define the key tensor operations required in this paper. All the definitions presented here come from [14].

2.1 Tensor

Definition 2.1 (Tensor) *A tensor is a multi-way array. The* order *of a tensor is the number of dimensions, also known as ways or modes.*

Higher-order (N-way with $N \geq 3$) tensors are denoted by boldface Euler script letters, e.g., $\mathcal{X}$. Matrices (tensors of order two) are denoted by boldface capital letters, e.g., $\mathbf{A}$; vectors (tensors of order one) are denoted by boldface lowercase letters, e.g., $\mathbf{a}$; and scalars are denoted by lowercase letters, e.g., a. The ith entry of a vector $\mathbf{a}$ is denoted by a_i, element (i, j) of a matrix $\mathbf{A}$ by a_{ij}, and element (i, j, k) element of a third-order tensor $\mathcal{X}$ by x_{ijk}. Indices typically range from 1 to their capital version, e.g., $i = 1, \ldots, I$. The nth element in a sequence is denoted by a superscript in parentheses. For example, $\mathbf{v}^{(n)}$ denotes the nth vector in a sequence and $v_i^{(n)}$ denotes the ith element on the nth vector $\mathbf{v}^{(n)}$. Likewise, $\mathbf{A}^{(n)}$ denotes the nth matrix in a sequence and $\mathbf{a}_r^{(n)}$ denotes the rth column of the matrix $\mathbf{A}^{(n)}$.

Definition 2.2 (Norm of a Tensor) *The norm of an N-way tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is*

$$\|\mathcal{X}\| = \sqrt{\sum_{i_1=1}^{I_1} \sum_{i_2=1}^{I_2} \cdots \sum_{i_N=1}^{I_N} x_{i_1 i_2 \cdots i_N}^2}.$$

Definition 2.3 (Tensor Fiber) *A tensor fiber is a one-dimensional fragment of a tensor, obtained by fixing all indices but one.*

Tensor fibers are the higher-order analogue of matrix rows and columns. Third-order tensors have column, row, and tube fibers, denoted by $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$, and $\mathbf{x}_{ij:}$, respectively.

Definition 2.4 (Tensor Slice) *A tensor slice is a two-dimensional fragment of a tensor, obtained by fixing all indices but two.*

For example, the horizontal, lateral, and frontal slides of a third-order tensor $\mathcal{X}$ are denoted by $\mathbf{X}_{i::}$, $\mathbf{X}_{:j:}$, and $\mathbf{X}_{::k}$, respectively.

2.2 Basic Tensor Operations

Definition 2.5 (Matricization) Matricization*, also known as* unfolding *or* flattening*, is the process of reordering the elements of an N-way array into a matrix.*

Definition 2.6 (Mode-n matrix product) *The n-mode matrix product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{U}$ and is of size $I_1 \times \cdots \times I_{n-1} \times J \times I_{n+1} \times \cdots \times I_N$. Element-wise, we have*

$$(\mathcal{X} \times_n \mathbf{U})_{i_1 \cdots i_{n-1} j \, i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \cdots i_N} \, u_{j i_n}.$$

Multiple mode-n matrix products can be performed in any order (see, e.g., [13]), i.e.,

$$(\mathcal{X} \times_n \mathbf{A}) \times_m \mathbf{B} = (\mathcal{X} \times_m \mathbf{B}) \times_n \mathbf{A}$$

for $m \neq n$. From [4], we adopt the following shorthand notation for multiplication in every mode:

$$\mathcal{X} \times \{\mathbf{A}\} \equiv \mathcal{X} \times_1 \mathbf{A}^{(1)} \times_2 \mathbf{A}^{(2)} \cdots \times_N \mathbf{A}^{(N)},$$

and for multiplication in every mode but one:

$$\mathcal{X} \times_{-n} \{\mathbf{A}\} \equiv \\ \mathcal{X} \times_1 \mathbf{A}^{(1)} \cdots \times_{n-1} \mathbf{A}^{(n-1)} \times_{n+1} \mathbf{A}^{(n+1)} \cdots \times_M \mathbf{A}^{(M)}.$$

Definition 2.7 (Mode-n vector product) *The n-mode vector product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with a vector $\mathbf{v} \in \mathbb{R}^{I_n}$ is denoted by $\mathcal{X} \,\bar{\times}_n\, \mathbf{v}$ and is of size $I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N$. Element-wise, we have*

$$(\mathcal{X} \,\bar{\times}_n\, \mathbf{v})_{i_1 \cdots i_{n-1} i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \cdots i_N} \, v_{i_n}.$$

Note that the order of the result is reduced by one. Multiplying a three-way tensor by a vector in one mode results in a two-way tensor (a matrix). It is possible to multiple a tensor by a vector in more than one mode as well. Multiplying a three-way tensor by vectors in two modes results in a one-way tensor (a vector); multiplying in all modes results in a scalar.

2.3 Tensor Decompositions

Many tensor decompositions have been proposed (see [14] for a detailed survey), among which, CANDECOMP/PARAFAC (CP) [7, 11] and Tucker decomposition [21] are the two most popular. Here we focus on the Tucker decomposition.

Definition 2.8 (Tucker Decomposition) *Let $\mathcal{X}$ be a tensor of size $I_1 \times I_2 \times \cdots \times I_N$. A Tucker decomposition of $\mathcal{X}$ yields a core tensor $\mathcal{G}$ of specified size $J_1 \times J_2 \times \cdots \times J_N$ and factor matrices $\mathbf{A}^{(n)}$ of size $I_n \times J_n$ for $n = 1, \ldots, N$ such that*

$$\mathcal{X} \approx \mathcal{G} \times \{\mathbf{A}\}. \tag{1}$$

The Tucker decomposition approximates a tensor as a smaller core tensor (i.e., a compressed version of the original tensor) times the product of matrices that span appropriate subspaces in each mode. Typically, the factor matrices $\mathbf{A}^{(n)}$ are assumed to be orthogonal.

3 Tensor storage and access

There are many options for storing sparse tensors. Here, we use coordinate format as proposed in [6]. Assume $\mathcal{X}$ is a sparse tensor of size $I_1 \times I_2 \times \cdots \times I_N$ with $P = \text{nnz}(\mathcal{X})$ non-zeros. We store the tensor as

$$\mathbf{v} \in \mathbb{R}^P \quad \text{and} \quad \mathbf{S} \in \mathbb{N}^{P \times N}.$$

Here, the pth nonzero value is given by v_p and its subscript is given by the pth row of $\mathbf{S}$, i.e., $\mathbf{s}_{p:}$. In other words,

$$x_{s_{p1}, s_{p2}, \ldots, s_{pN}} = v_p.$$

The total storage for a sparse tensor with P non-zeros is the $(N+1)P$ elements. This is the format implemented in the MATLAB Tensor Toolbox, version 2.0 [5], which is the framework in which we do all our testing.

Here, we summarize the main point. Let $\mathcal{X}$ be an N-way tensor of size $I_1 \times I_2 \times \cdots \times I_N$ and assume that we are doing mode-n vector products in the modes specified by

$$\mathcal{M} = \{n_1, \ldots, n_M\} \subseteq \{1, \ldots, N\}.$$

It is possible to calculate

$$\mathcal{Z} = \mathcal{X} \,\bar{\times}_{n_1}\, \mathbf{v}^{(n_1)} \cdots \,\bar{\times}_{n_M}\, \mathbf{v}^{(n_M)},$$

using only P additional memory elements and the storage required for $\mathcal{Z}$, which is

$$\max\{\prod_{n \notin \mathcal{M}} I_n, 1\}.$$

This calculation can be done with $O(\text{nnz}(\mathcal{X}))$ memory and is described in detail in [6].

4 MET: Memory-Efficient Tucker Decomposition

We are interested in the case where $\mathcal{X}$ is a large, sparse tensor and we wish to find a Tucker approximation as in Equation (1) such that the (dense) core tensor $\mathcal{G}$ has much smaller dimensions than the original sparse tensor $\mathcal{X}$; in other words, we assume

$$J_n \ll I_n \quad \text{for all} \quad n = 1, \ldots, N.$$

Moreover, we assume that the dimensions J_n are small enough that the dense tensor $\mathcal{G}$ (with $\prod J_n$ elements) can easily fit in memory.

In fitting Tucker, the goal is to find a core tensor $\mathcal{G}$ of specified size $J_1 \times J_2 \times \cdots \times J_N$ and factor matrices $\mathbf{A}^{(n)}$ of size $I_n \times J_n$ for $n = 1, \ldots, N$ that minimize:

$$\min_{\mathcal{G}, \mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}} \| \mathcal{X} - \mathcal{G} \times \{\mathbf{A}\} \|^2. \tag{2}$$

The core tensor $\mathcal{G}$ is determined uniquely by the factor matrices. That is, given a fixed set of factor matrices, the optimal core is

$$\mathcal{G} = \mathcal{X} \times \{\mathbf{A}^\mathsf{T}\}. \tag{3}$$

Here we assume $\{\mathbf{A}\}$ is orthogonal.

The naive way of computing the error $e = \| \mathcal{X} - \mathcal{G} \times \{\mathbf{A}\} \|^2$ is very expensive, since $\mathcal{X} - \mathcal{G} \times \{\mathbf{A}\}$ is a large dense tensor. Fortunately, it can be simplified as the follows:

$$\begin{aligned} e &= \| \mathcal{X} - \mathcal{G} \times \{\mathbf{A}\} \|^2 \\ &= \| \mathcal{X} \|^2 + \| \mathcal{G} \|^2 - 2\langle \mathcal{X}, \mathcal{G} \times \{\mathbf{A}\} \rangle \\ &= \| \mathcal{X} \|^2 + \| \mathcal{G} \|^2 - 2\langle \mathcal{X} \times \{\mathbf{A}^\mathsf{T}\}, \mathcal{G} \rangle \\ &= \| \mathcal{X} \|^2 - \| \mathcal{G} \|^2 \end{aligned}$$

Through the above transformation, the objective function in Equation (2) can be rewritten as $\|\mathcal{X}\|^2 - \|\mathcal{G}\|^2$.

Since $\|\mathcal{X}\|$ is fixed, Equation (2) is equivalent to

$$\max_{\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}} \| \mathcal{X} \times \{\mathbf{A}^\mathsf{T}\} \|. \tag{4}$$

A common approach for solving Equation (4) is to use an alternating least squares (ALS) method, solving for one factor matrix at a time while holding the others fixed. In other words, we fix all the factor matrices except $\mathbf{A}^{(n)}$ and solve

$$\max_{\mathbf{A}^{(n)}} \| \mathcal{X} \times \{\mathbf{A}^\mathsf{T}\} \|. \tag{5}$$

Setting

$$\mathcal{Y} = \mathcal{X} \times_{-n} \{\mathbf{A}^\mathsf{T}\}, \tag{6}$$

Equation (5) can be rewritten as

$$\max_{\mathbf{A}^{(n)}} \|\mathbf{A}^{(n)\mathsf{T}} \mathbf{Y}_{(n)}\|,$$

which is solved via the SVD of $\mathbf{Y}_{(n)}$. See [9, 13] for details.

This full Tucker-ALS algorithm is shown in Algorithm 1. The initialization procedure is standard (see [14] and reference therein). Note that the first factor matrix does not need to be initialized since it is the one solved for in the first inner iteration. The problem boils down to calculating the leading singular vectors of a large, sparse matrix, which is straightforward. We discuss the inner loops in more detail below. Also observe that $\mathcal{G}$, calculated once per outer loop, is calculated using the $\mathcal{Y}$ from the last inner iteration. This is straightforward to compute, as is its norm.

Algorithm 1 Tucker-ALS for N-mode tensors

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \cdots \times I_N$.
in: Desired core size: $J_1 \times J_2 \times \cdots \times J_N$.
for $n = 2, \ldots, N$ **do** {initialize factor matrices}
  $\mathbf{A}^{(n)} \leftarrow J_n$ leading left singular vectors of $\mathbf{X}_{(n)}$.
end for
repeat {outer loop}
  for $n = 1, \ldots, N$ **do** {inner loop}
    $\mathcal{Y} \leftarrow \mathcal{X} \times_{-n} \{\mathbf{A}^\mathsf{T}\}$.
    $\mathbf{A}^{(n)} \leftarrow J_n$ leading left singular vecs. of $\mathbf{Y}_{(n)}$.
  end for
  $\mathcal{G} \leftarrow \mathcal{Y} \times_N \mathbf{A}^{(N)}$ $(= \mathcal{X} \times \{\mathbf{A}\})$.
until $\|\mathcal{G}\|$ ceases to increase or the maximum number of outer iterations is exceeded.
out: Core tensor $\mathcal{G}$ of size $J_1 \times J_2 \times \cdots \times J_N$ and orthogonal factor matrices $\mathbf{A}^{(n)}$ of size $I_n \times J_n$ for $n = 1, \ldots, N$.

The bottleneck in this method is in the inner loop in the calculation of $\mathcal{Y}$; see Equation (6). Here we need to compute the product of a sparse tensor times a series of dense matrices. However, the intermediate products are dense and may be too large to fit in memory even if the final tensor $\mathcal{Y}$ does fit. We assume that there is enough memory to fit $\mathcal{Y}$, i.e., that there is enough memory to store

$$\max_n \left(I_n \prod_{m \neq n} J_m \right)$$

elements and focus entirely on the problem of how to compute $\mathcal{Y}$. Once $\mathcal{Y}$ is obtained, finding its leading left singular vectors is straightforward.

4.1 Three-way tensors

To simplify the discussion, we first focus on the problem of calculating $\mathcal{Y}$ in Equation (6) for $N = 3$. Consider the following calculation (the size of each object is listed above it), which we encounter in the first inner iteration:

$$\overset{I_1 \times J_2 \times J_3}{\mathcal{Y}} = \overset{I_1 \times I_2 \times I_3}{\mathcal{X}} \times_2 \overset{J_2 \times I_2}{\mathbf{A}^{(2)}} \times_3 \overset{J_3 \times I_3}{\mathbf{A}^{(3)}}. \tag{7}$$

We assume that I_1, I_2, I_3 are relatively large and J_1, J_2, J_3 are relatively small; e.g., $I_n = 1000$ and $J_n = 10$ for $n = 1, 2, 3$.

MET(0): Standard calculation (no modes element-wise) The standard way to calculate Equation (7) in Tucker-ALS is as follows, where the brackets indicate the order of operations:

$$\mathcal{Y} = \overbrace{\left(\overbrace{\left(\mathcal{X} \times_3 \mathbf{A}^{(3)}\right)}^{I_1 \times I_2 \times J_3} \times_2 \mathbf{A}^{(2)}\right)}^{I_1 \times J_2 \times J_3}$$

The first intermediate result,

$$\mathcal{X} \times_3 \mathbf{A}^{(3)}$$

is a *dense* tensor of size $I_1 \times I_2 \times J_3$ and may easily be too big to fit in memory. Depending on the amount of memory available, there are two alternatives for calculating $\mathcal{Y}$ with less memory.

MET(1): Slice updates (one mode element-wise). We can calculate $\mathcal{Y}$ from Equation (7) one slice at a time using less memory. Specifically, we handle the third mode element-wise and have

$$\mathbf{Y}_{::j_3} = \overbrace{\left(\overbrace{\left(\mathcal{X} \;\bar{\times}_3\; \mathbf{a}^{(3)}_{j_3}\right)}^{I_1 \times I_2} \times_2 \mathbf{A}^{(2)}\right)}^{I_1 \times J_2} \text{ for } j_3 = 1, \ldots, J_3, \tag{8}$$

where $\mathbf{a}^{(3)}_{j_3}$ is the j_3-th column of $\mathbf{A}^{(3)}$. Here, the largest intermediate result is a matrix of size $I_1 \times I_2$ and we need to do J_3 calculations. Of course, it is possible to instead handle the second mode element-wise:

$$\mathbf{Y}_{:j_2:} = \overbrace{\left(\overbrace{\left(\mathcal{X} \;\bar{\times}_2\; \mathbf{a}^{(2)}_{j_2}\right)}^{I_1 \times I_3} \times_3 \mathbf{A}^{(3)}\right)}^{I_1 \times J_3} \quad \text{for } j_2 = 1, \ldots, J_2. \tag{9}$$

MET(2): Fiber updates (two modes element-wise). We can calculate $\mathcal{Y}$ from Equation (7) one fiber at a time with even less memory. Specifically, two modes are handled element-wise:

$$\mathbf{y}_{:j_2j_3} = \overbrace{\mathcal{X} \;\bar{\times}_2\; \mathbf{a}^{(2)}_{j_2} \;\bar{\times}_3\; \mathbf{a}^{(3)}_{j_3}}^{I_1} \quad \text{for } j_2 = 1, \ldots, J_2,\ j_3 = 1, \ldots, J_3.$$

Here the largest intermediate result is a vector of size I_1 and we need to do J_2J_3 calculations.

4.2 N-way tensors

To generalize to the N-way case, two questions need to be answered:

- Given fixed memory, how can we quickly compute
$$\mathcal{Y} = \mathcal{X} \times_{-n} \{\mathbf{A}\}?$$
- Which modes should be treated element-wise?

Memory-Efficient Tensor Times Matrices (METTM): Assume that we know which subset modes should be treated element-wise, denoted by

$$\mathcal{E} \subseteq \{1, \ldots, N\} \setminus \{n\}.$$

Then the order of the intermediate sub-tensors that are formed is $N - |\mathcal{E}|$. For example, if $N = 4$ and $\mathcal{E} = \{3, 4\}$, then the intermediate results are two-way tensors (matrices). Moreover, if $\mathcal{E}$ is non-empty, the size of the largest intermediate result is

$$\prod_{m \notin \mathcal{E}} I_m,$$

and the number of element-wise calculations that needs to be performed is

$$\prod_{m \in \mathcal{E}} J_m.$$

Selecting the modes to handle element-wise: There is some choice in *which* modes to handle element-wise. For example, trivially we can minimize the memory by having all modes handled element-wise, i.e., $\mathcal{E} = \{1, 2, \ldots, N\}$. The other extreme, which standard Tucker-ALS uses, is to have empty $\mathcal{E}$, leading to the intermediate blow-up problem for large-scale tensors.

In order to balance the number of computations and the size of the intermediate result, we consider the *reduction ratios* defined as

$$K_m \equiv \frac{I_m}{J_m} \quad \text{for } m \in \{1, \ldots, N\} \setminus \{n\}.$$

Those modes with the largest values for K_m are handled element-wise. For example, if $N = 3$ and $|\mathcal{E}| = 1$, we would choose Equation (8) if $J_3/I_3 > J_2/I_2$ and Equation (9) otherwise. The intuition is that the matrix $\mathbf{A}^{(m)}$ with large K_m typically has two nice properties: 1) fewer columns (J_m is small) need to be handled element-wise, which implies fast computation; and 2) more elements in a single column, which implies big space reduction on tensor size when multiplying this matrix.

How many modes are handled element-wise depends on the available memory. Typically, we will choose the

minimal number of modes in $\mathcal{E}$ that guarantees that the memory does not overflow. Given the order based on reduction ratio, this selection can be done easily.

In summary, we have implemented a general-purpose function for this calculation in which the only parameter is the available memory and all the others are handled automatically. Algorithm 2 illustrates the pseudo-code.

Algorithm 2 MET for N-mode tensor

in: Tensor $\mathcal{X}$ of size $I_1 \times I_2 \times \cdots \times I_N$.
in: Desired core size: $J_1 \times J_2 \times \cdots \times J_N$.
in: Available memory M
initialize $\mathbf{A}^{(n)}$, for $n = 1$ to N.
repeat {outer loop}
  for $n = 1, \ldots, N$ **do** {inner loop}
    $\mathcal{Y} \leftarrow \text{METTM}(\mathcal{X}, \{\mathbf{A}^{(i)\mathsf{T}}\}_{i \neq n})$.
    $\mathbf{A}^{(n)} \leftarrow J_n$ leading left singular vecs. of $\mathbf{Y}_{(n)}$.
  end for
  $\mathcal{G} \leftarrow \mathcal{Y} \times_N \mathbf{A}^{(N)}$.
until $\|\mathcal{G}\|$ ceases to increase or the maximum number of outer iterations is exceeded.
out: Core tensor $\mathcal{G}$ of size $J_1 \times J_2 \times \cdots \times J_N$ and orthogonal factor matrices $\mathbf{A}^{(n)}$ of size $I_n \times J_n$ for $n = 1, \ldots, N$.

5 Experiment Evaluation

Now we evaluate our method on a number of large sparse tensors. First, we study their performance on synthetic tensors that we generate by varying several data dependent parameters. Second, we report the results on various of real tensors. We consider three performance metrics:

- **Intermediate space consumption** is the memory needed for storing the intermediate results in the computation. This determines the scale of the problems that we can solve.
- **CPU time** is the execution elapsed time of the computation. This determines how efficient we can solve the problem.
- **Relative error** quantifies the quality of the tensor decomposition. It is computed as $\|\mathcal{T} - \mathcal{X}\|^2 / \|\mathcal{X}\|^2$ where $\mathcal{X}$ is the input tensor, and $\mathcal{T}$ the factorized result from Tucker.

All experiments are performed using MATLAB 2007b with the Tensor Toolbox [4, 6, 5] on a workstation with Intel Xeon 3GHz processor and 16GB RAM. For both Tucker-ALS and MET, we set the maximum number of iterations to 50 and the threshold for the change in $\|\mathcal{G}\|$ to 1e-4.

5.1 Tucker on synthetic data

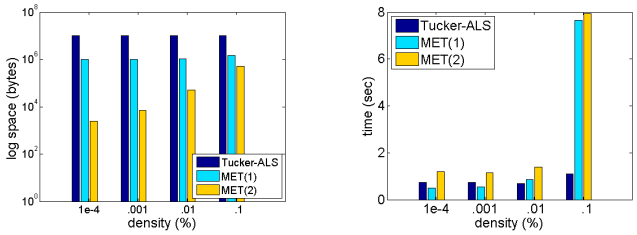
To understand how different tensor properties affect the memory and speed, we test MET on a set of low-rank sparse tensors by varying different parameters. There are four sets of data dependent parameters:

- **Density** D is the percentage of nonzero elements in the tensor, e.g., Enron dataset has the density .0025%.
- **Mode** N is the number of modes or ways in the tensors, e.g., Enron dataset has 4 modes.
- **Dimensionality** I is the size on a mode, e.g., the dimensionality on mode 1 of Enron dataset is 1,000.
- **core size** J is the dimensionality in the core tensor.

Given parameters D, N, I, and J, we first construct a random core tensor of size $\underbrace{J \times \cdots \times J}_{N \text{ times}}$ and N random matrices $\mathbf{A}^{(1)} \ldots \mathbf{A}^{(N)}$ of size $I \times J$; we then construct the full tensor $\mathcal{X} = \mathcal{G} \times \{\mathbf{A}\}$; finally we keep only the top D percent of large elements in $\mathcal{X}$ to construct a sparse tensor. Note that this data generation operation is not memory-efficient, since it requires formatting the full tensor and sorting all the elements. Because of this, the synthetic tensors are smaller than the real tensor datasets we experimented with.

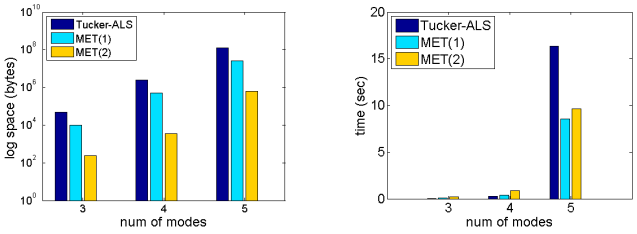
To acquire deeper insights into Tucker, we first present how the space consumption and computational time vary when varying different data-dependent parameters. Note that in all cases, the relative error is extremely small and consistent across all methods (the error varies from $1e-9$ to $1e-19$), so we omitted that part in the paper. Here we compare four variants of Tucker as described in Section 4. Figure 2-5 shows the space and time (y-axis) when varying each parameter while setting the other ones to default values. All figures plot the y-axis in the logarithm scale (base 10).

Density: Figure 2 shows space and time (y-axis) versus density (x-axis). Compared to the standard Tucker-ALS, MET achieves a 100X to 1,000X space reduction. As the density drops, the gap between Tucker-ALS and MET becomes bigger. CPU times are comparable between MET and Tucker-ALS, except for the high density one, where MET takes much longer than Tucker-ALS. The rule of thumb is if the tensor is sparse,



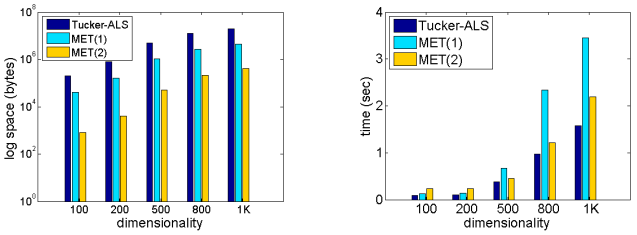
(a) space consumption (b) CPU time

Figure 2. Varying tensor density: mode $N = 3$, dimensionality $n = 500$, core size 10^3



(a) space consumption (b) CPU time

Figure 3. Varying tensor modes: density .001%, dimensionality $n = 500$, core size 10^3



(a) space consumption (b) CPU time

Figure 4. Varying tensor dimensionality: density .001%, mode $N = 3$, core size 10^3

(a) space consumption (b) CPU time

Figure 5. Varying core size:density .001%, mode $N = 3$, dimensionality $n = 500$

MET can perform almost as fast as Tucker-ALS but needs much less memory.

Mode: In Figure 3, space and time increase with the number of modes of the tensor. In all cases, MET require much less space while taking comparable time to Tucker-ALS. It is interesting to see that for the mode-5 tensor, Tucker-ALS is actually slower than MET. Note that the per-element CPU time is actually sub-linear in the number of modes.

Dimensionality: In Figure 4, space and time increase with the dimensionality of the tensor. Again MET outperforms Tucker-ALS significantly in space, and achieves that with comparable CPU-time.

core size: Similarly, in Figure 5, as we increase the result size (core tensor size), both space consumption and CPU time increase. It may be hard to notice the upward trend in Figure 5(a), since the y-axis is in logarithm scale. But actually, the increase for both space and time are proportional in this case. Again, MET is the best in all cases.

5.2 Performance on real data

5.2.1 Description

We select a diverse set of data from different domains including Email communication (Enron), bibliography data (DBLP), web links (Web), and network flow traffic (Network).

The **Enron dataset** comprises 4 modes: sender, recipient, date, and keyword, with size 1K×1K×1.1K×200 and over 5.39 million nonzero entries. We select the top 1000 users and the 200 most frequent keywords (after simple stop-word removal and stemming) from the email messages of 1,126 days in the Enron MYSQL database [22]. The element (i,j,k,l) is 1 if there exists an email that is sent from sender i to recipient j with keyword k on day l; otherwise, it is 0. Note that the tensor is very sparse with only .0025% nonzeros out of all **220 billion** potential elements.

The **DBLP dataset** comprises 3 modes: author, date, and keyword, with size 5K×1K×1K and 0.5 million nonzero entries, which is constructed using a similar procedure to the Enron dataset. By parsing the DBLP XML file [23], we select the 5000 most prolific authors, the 1000 most common keywords, and the 1000 must popular conferences up to 2005. Each element in the tensor is the corresponding paper count. Again the tensor is very sparse with .0536% nonzeros out of 1 billion potential elements.

The **web dataset** comprises 3 modes: source URL, outgoing URL and anchor text, with size 5K×5K×200 and 0.1 million nonzero entries. We con-

struct this tensor from the well-known TREC Web Corpus WT10G [24], which has about 1.7 million webpages (10GB raw text) Based on the hyperlinks in the WT10G data, we select the top 3000 most popular words in the anchor texts to construct a source-destination-word tensor. Elements in this tensor are the word counts.

The **network dataset** comprises 3 modes: source IP, destination IP and destination port number with the size 3K×5K×200 and .5 million nonzero entries. The data are aggregated from anonymized intranet traffic flows from a university for over a month. We selected the top 3000 IP address and top 200 ports based on the number of distinct IP connections. The element (i,j,k) in the tensor is 1 if there exists a connection from source i to destination j on port k, and otherwise 0.

Choosing the "right" size of core tensors often depends on the applications, which is beyond the scope of this paper. In this paper, we will present the effect of core size on space and time for synthetic datasets. For these real datasets, we fix the core size across all experiments to be 10. For example, the Enron core tensor is 10×10×10×10.

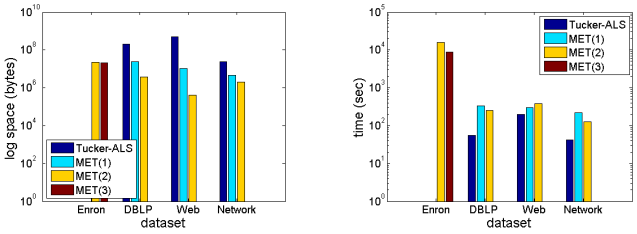
5.2.2 Performance

Figure 6 shows the space and time required for all four datasets. Note that for Enron data, only the MET(2) and MET(3) can run on that server, and Tucker-ALS and MET(1) failed due to memory overflow. For the other datasets, all methods completed, with a significant gap (1000X difference) between Tucker-ALS and our MET(1)-(2). Besides that, the similar trends are observed as in synthetic data: 1) space-wise, MET achieves order of magnitudes saving to the standard Tucker-ALS; and 2) time-wise, MET is comparable to Tucker-ALS.

5.2.3 Data mining case-study

As a higher-order generalization of SVD and PCA, Tucker decomposition has widely been used as a core technique for analyzing sparse tensors [19, 20, 8]. The typical pattern is to use the factorized tensor for other mining tasks such as clustering, trend identification and anomaly detection. Here we first give a quick overview of how Tucker can help with different mining tasks. Then we demonstrate some results on the clustering task.

Recall, given an input tensor $\mathcal{X}$, Tucker approximates $\mathcal{X}$ as $\mathcal{G} \times \{\mathbf{A}\}$ such that the error $e = \| \mathcal{X} - \mathcal{G} \times \{\mathbf{A}\} \|$ is small. With the result of Tucker, several mining tasks can be approached as the follows:



(a) space consumption (b) CPU time

Figure 6. Performance of Tucker on real datasets: Tucker-ALS and MET(1) failed to complete on Enron due to memory overflow. Note that only 4-way tensor, Enron, has the result on MET(3), while the others are all 4-way tensors, therefore, only results up to MET(2) are possible.

Clustering: Since $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ forms the basis for mode n, any clustering algorithm can be applied on $\mathbf{A}^{(n)}$ by treating each row of $\mathbf{A}^{(n)}$ as a J_n-dimensional point. The benefit of doing clustering on these subspaces is that they are jointly chosen across all modes $(1 \leq n \leq N)$ to minimize the global reconstruction error, which cannot be achieved by looking at an individual mode.

Trend detection: The core tensor $\mathcal{G}$ captures the main trend in the tensor. The trend across mode i is encoded in $\mathcal{G} \times_n \mathbf{A}^{(n)}$, e.g., PCA is the special case in the second order case.

Anomaly detection: The reconstruction error gives a global measure of the quality of the approximation. However, the error can be calculated on an arbitrary scale, e.g., on the element level or the sub-tensor level such as slices or fibers. Higher error on a specific sub-tensor indicates a deviation from the main trend in that specific region, and the sub-tensor is therefore an anomaly.

Now let us see one example of clustering on DBLP dataset using Tucker decomposition with k-means. Similar clustering has been proposed in [19]; however, due to the huge memory overhead in original Tucker-ALS, only a small set of data are involved in that work. In particular, only publications from two conferences are considered. However, with MET, now we can easily analyze much larger tensors with 1,000 conferences. More specifically, we apply Tucker powered by MET on $5K \times 1K \times 1K$ DBLP to obtain three matrices: the author matrix $\mathbf{A}^{(1)} \in \mathbb{R}^{5K \times 10}$, the conference matrix $\mathbf{A}^{(2)} \in \mathbb{R}^{1K \times 10}$, and the keyword matrix $\mathbf{A}^{(3)} \in \mathbb{R}^{1K \times 10}$. Then we apply k-means on the row vectors of each factor matrix. We set k to be 200, 100, and 100 on $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$, and $\mathbf{A}^{(3)}$, respectively.

The sample clusters on each mode are shown in Table 2. In the conference mode, it clearly clusters different conferences into their corresponding domains, such as KDD, ICDM, PAKDD for data mining, and ICDE, VLDB, SIGMOD for databases. In the keyword mode, it again identifies interesting group of keywords, such as database, data, query for databases; 3D, video, motion, images for graphics. Similarly, we are able to group authors into clusters based on their research interests. A more detailed analysis can be done to identify the correlation across different modes, but due to the space limitations, we omit that part.

In short, we believe with the help of MET, Tucker can be more effectively applied to many large scale data mining problems.

Conf.
1. KDD,ICDM,PAKDD,SIGIR
2. ICDE, VLDB,SIGMOD
3. UAI,ICML,NIPS,Agents
4. ICNP,ICDCS, SIGCOMM, SIGMETRICS
Word
1. database, data, query
2. 3d, video, motion, images, segmentation
3. knowledge, learning, reasoning
Author
1. H. V. Jagadish, Hans-Jrg Schek, David Maier, Michael Stonebraker, Surajit Chaudhuri, Carlo Zaniolo, Umeshwar Dayal, Rakesh Agrawal, Divesh Srivastava, Jeffrey F. Naughton, Raghu Ramakrishnan, C. Mohan, Yannis E. Ioannidis, David J. DeWitt, Jennifer Widom, Laks V. S. Lakshmanan, Nick Koudas
2. Philip S. Yu, Ming-Syan Chen, Hector Garcia-Molina, Abraham Silberschatz, Rajeev Rastogi
3. Elisa Bertino, Divyakant Agrawal, Amr El Abbadi, Shamkant B. Navathe, Clement T. Yu, Jiawei Han, Elke A. Rundensteiner, Hongjun Lu, Gerhard Weikum, Masaru Kitsuregawa, Hans-Peter Kriegel, Kian-Lee Tan, Beng Chin Ooi, Christos Faloutsos

Table 2. Sample clustering results on DBLP dataset

6 Related Work

Computing tensor decompositions The ALS algorithm for computing Tucker was proposed by Kroonenberg an De Leeuw [16] for the 3-way case and later extended to the N-way case by Kapteyn, Neudecker, and Wansbeel [12]. Andersson and Bro [3] suggest several ideas for improving the computation in MATLAB. Later, De Lathauwer, De Moor, and Vandewalle [9] suggested futher improvements to improve the numerical accuracy of the method, which they renamed to be the "higher order orthogonal iteration."

There has been no previous work on computing Tucker-ALS for large-scale sparse tensors. In their discussion of n-mode multiplication of a tensor by a matrix $\mathbf{A}$, Bader and Kolda [6] observe that "Unless $\mathbf{A}$ has special structure (e.g., diagonal) the result is dense."

Recently, other methods for computing Tucker factorizations have been developed. Of note, Eldén and Savas [10] have proposed a Newton-Grassmann optimization approach. The procedures developed here would also be important for applying their approach to large-scale sparse tensors.

Tensor applications Tucker decompositions have been used in a variety of applications in data mining, and we highlight several distinctive examples here. Savas and Eldén [17] applied the HO-SVD (a version of the Tucker decomposition) to identifying handwritten digits. Acar et al. [1, 2] applied Tucker and other tensor decompositions to the problem of separating conversations in online chatrooms, and Chi et al. [8] have applied it to the blogosphere. J.-T. Sun et al. [20] used Tucker for analyzing clickthrough data. J. Sun et al. [19, 18] have written a pair of papers on dynamically updating a Tucker approximation, with applications ranging from text analysis to environmental and network modeling.

7 Conclusions

In conclusion, tensors, especially sparse tensors, are important data models for many different applications. Tensor decompositions such as Tucker are becoming standard tools for analyzing multiway data. However, the "intermediate blowup" issue in Tucker is a severe problem for even moderately sized real problems. To address this problem, we propose memory-efficient Tucker decomposition (MET) for decomposing large sparse tensors. MET is adaptive to the available system resources with automatic parameter tuning. Compared to the previous state-of-the-art Tucker-ALS algorithm, MET can achieve significant space reduction (over 1000X) without sacrificing much speed. We demonstrate the performance on both synthetic and real datasets and illustrate the potential data mining tasks with the help of MET.

In the future, we plan to evaluate other types of tensor decomposition in the context of sparse data. Since tensor decompositions such as Tucker may in turn depend on matrix decompositions such as the SVD, an in-

teresting extension is to leverage memory-efficient matrix operations in the tensor decomposition to further decrease memory consumption. Another different direction is to study the parallel version of the tensor algorithms.

References

[1] E. Acar, S. A. Çamtepe, M. S. Krishnamoorthy, and B. Yener. Modeling and multiway analysis of chatroom tensors. In *ISI 2005*, pages 256–268, 2005.

[2] E. Acar, S. A. Çamtepe, and B. Yener. Collective sampling and analysis of high order tensors for chatroom communications. In *ISI 2006*, pages 213–224, 2006.

[3] C. A. Andersson and R. Bro. Improving the speed of multi-way algorithms: Part I. Tucker3. *Chemometrics and Intelligent Laboratory Systems*, 42(1–2):93–103, 1998.

[4] B. W. Bader and T. G. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, 2006.

[5] B. W. Bader and T. G. Kolda. MATLAB Tensor Toolbox Version 2.1. 2006.

[6] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[7] J. D. Carroll and J. J. Chang. Analysis of individual differences in multidimensional scaling via an N-way generalization of 'Eckart-Young' decomposition. *Psychometrika*, 35:283–319, 1970.

[8] Y. Chi, B. L. Tseng, and J. Tatemura. Eigen-trend: trend analysis in the blogosphere based on singular value decompositions. In *CIKM '06*, pages 68–77, 2006.

[9] L. De Lathauwer, B. De Moor, and J. Vandewalle. On the best rank-1 and rank-$(R_1, R_2, \ldots, R_N)$ approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[10] L. Eldén and B. Savas. A Newton–Grassmann method for computing the best multi-linear rank-(r_1, r_2, r_3) approximation of a tensor. Technical Report LITH-MAT-R-2007-6-SE, Linköpings University, 2007.

[11] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.

[12] A. Kapteyn, H. Neudecker, and T. Wansbeek. An approach to n-mode components analysis. *Psychometrika*, 51(2):269–275, 1986.

[13] T. G. Kolda. Multilinear operators for higher-order decompositions. Technical Report SAND2006-2081, Sandia National Laboratories, 2006.

[14] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*. To appear.

[15] T. G. Kolda, B. W. Bader, and J. P. Kenny. Higher-order web link analysis using multilinear algebra. In *ICDM 2005*, pages 242–249, 2005.

[16] P. M. Kroonenberg and J. De Leeuw. Principal component analysis of three-mode data by means of alternating least squares algorithms. *Psychometrika*, 45(1):69–97, 1980.

[17] B. Savas and L. Eldén. Handwritten digit classification using higher order singular value decomposition. *Pattern Recognition*, 40(3):993–1003, 2007.

[18] J. Sun, S. Papadimitriou, and P. Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *ICDM 2006*, pages 1076–1080, 2006.

[19] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: Dynamic tensor analysis. In *KDD'06*, pages 374–383, 2006.

[20] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen. CubeSVD: a novel approach to personalized web search. In *WWW 2005*, pages 382–390, 2005.

[21] L. R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3), 1966.

[22] Enron database. `http://bailando.sims.berkeley.edu/enron_email.html`.

[23] `http://www.informatik.uni-trier.de/~ley/db/`.

[24] WT10G collection. `http://trec.nist.gov/data.html`.