Static and Streaming Tucker Decomposition for Dense Tensors

JUN-GI JANG, Seoul National University, Republic of Korea
U KANG*, Seoul National University, Republic of Korea

Given a dense tensor, how can we efficiently discover hidden relations and patterns in static and online streaming settings? Tucker decomposition is a fundamental tool to analyze multidimensional arrays in the form of tensors. However, existing Tucker decomposition methods in both static and online streaming settings have limitations of efficiency since they directly deal with large dense tensors for the result of Tucker decomposition. In a static setting, although few static methods have tried to reduce their time cost by sampling tensors, sketching tensors, and efficient matrix operations, there remains a need for an efficient method. Moreover, streaming versions of Tucker decomposition are still time-consuming to deal with newly arrived tensors.

We propose D-Tucker and D-TuckerO, efficient Tucker decomposition methods for large dense tensors in static and online streaming settings, respectively. By decomposing a given large dense tensor with randomized singular value decomposition, avoiding the reconstruction from SVD results, and carefully determining the order of operations, D-Tucker and D-TuckerO efficiently obtain factor matrices and core tensor. Experimental results show that D-Tucker achieves up to 38.4× faster running times, and requires up to 17.2× less space than existing methods while having similar accuracy. Furthermore, D-TuckerO is up to 6.1× faster than existing streaming methods for each newly arrived tensor while its running time is proportional to the size of the newly arrived tensor, not the accumulated tensor.

Additional Key Words and Phrases: Dense tensor, Tucker decomposition, static setting, online streaming setting, efficiency

ACM Reference Format:
Jun-Gi Jang and U Kang. 2022. Static and Streaming Tucker Decomposition for Dense Tensors. *ACM Trans. Knowl. Discov. Data.* 37, 4, Article 111 (August 2022), 35 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

How can we efficiently discover hidden concepts and patterns of large dense tensors? Many real-world data including video, music, and air quality, can be represented as dense tensors. Tucker decomposition is a fundamental tool for factorizing a given tensor into factor matrices and a core tensor to find hidden concepts and latent patterns. Tucker decomposition has spurred much interests with various applications including dimensionality reduction [27, 47], recommendation [40, 44], and clustering [9, 20].

Alternating Least Square (ALS) is the most widely used method for Tucker decomposition. Existing ALS based methods, however, fail to satisfy all the desired properties for dense tensor decompositions: fast running time, low memory requirement, and high accuracy. Tucker-ALS

*Corresponding author.

Authors' addresses: Jun-Gi Jang, Seoul National University, Seoul, Republic of Korea, elnino4@snu.ac.kr; U Kang, Seoul National University, Seoul, Republic of Korea, ukang@snu.ac.kr.

1556-4681/2022/8-ART111 $15.00
https://doi.org/10.1145/1122445.1122456

which updates factor matrices iteratively is slow when the number of iterations is large. Moreover, Tucker-ALS has a memory problem to obtain final factor matrices and a core tensor since it directly handles large dense tensors in order to update the factor matrices and the core tensor at each iteration. A few static Tucker decomposition methods reduce the computational cost using efficient matrix operations [5, 33] or applying randomized algorithms [11, 35]. In addition, other Tucker decomposition methods [34, 56] reduce the computational time and the memory requirement by approximating large dense tensor. However, none of them provide both fast running time and accuracy. The major challenges to deal with large dense tensors are 1) how to efficiently approximate a large dense tensor with low error, and 2) how to update factor matrices by using approximated results.

In this paper, we propose D-Tucker and D-TuckerO, efficient Tucker decomposition methods on large dense tensors. D-Tucker and D-TuckerO run in static and online streaming settings, respectively. The main ideas of D-Tucker are as follows: 1) slice an input tensor into matrices and compress each matrix by exploiting randomized singular value decomposition (SVD), 2) initialize and update factor matrices and a core tensor using the SVD results, and 3) carefully determine the ordering of computations for efficiency. Similar to D-Tucker, D-TuckerO tackles Tucker decomposition for an online streaming setting with the following ideas: 1) avoid direct computations related to previous time-steps, 2) approximate each new incoming tensor, and then 3) carefully update factor matrices by determining the ordering of computations.

D-Tucker has three main phases: approximation, initialization, and iteration (see Fig. 1). The approximation phase of D-Tucker slices an input tensor into matrices, and then performs randomized SVD [19] of each sliced matrix. It allows us to reduce the size of the input tensor for updating the factor matrices and the core tensor. The initialization phase of D-Tucker initializes factor matrices by computing orthogonal factor matrices using the SVD results of sliced matrices. The iteration phase of D-Tucker updates the factor matrices and the core tensor by carefully exploiting the SVD results. D-Tucker achieves better time and space efficiency by carefully dealing with SVD results. Experimental results show that D-Tucker is faster and more memory-efficient than existing methods.

In an online streaming setting, D-TuckerO efficiently deals with each new incoming tensor by updating the temporal factor matrix, and then updating factor matrices of non-temporal modes. To update the temporal factor matrix, we leverage only the new incoming tensor and factor matrices of non-temporal modes obtained at the previous time step. For factor matrices of non-temporal modes, we avoid direct computations related to the entire tensor and the temporal factor matrix obtained at previous time-steps. It enables that computational cost and memory requirements are proportional to the size of a new incoming tensor, not the entire tensor. In addition, at each time-step, we approximate a new incoming tensor using the approximation phase of D-Tucker, and then update the factor matrices by carefully using the approximation results. Exploiting the approximation phase gives D-TuckerO the same benefit as D-Tucker: it allows us to use a smaller size of the approximated results than that of a new incoming tensor in updating the factor matrices and the core tensor, to achieve better time and space efficiency. Through comprehensive experiments, we show that D-TuckerO is more efficient than existing streaming methods, and the running time of D-TuckerO is proportional to the size of a newly arrived tensor, not the accumulated tensor.

The contributions of this paper are as follows.

- **Algorithm.** We propose D-Tucker and D-TuckerO, efficient methods for decomposing dense tensors in static and online streaming settings.
- **Analysis.** We provide analysis for the time and the space complexities of our proposed methods D-Tucker and D-TuckerO.

Table 1. Symbol description.

Symbol	Description
$\mathcal{X}_r$	Reordered tensor ($\in I_1 \times I_2 \times K_3 \times ... \times K_N$)
$\mathcal{G}$	Core tensor ($\in J_1 \times J_2 \times ... \times J_N$)
$\mathbf{A}^{(n)}$	Factor matrix of the n-th mode
I_n	Dimensionality of the n-th mode of $\mathcal{X}_r$ for modes $n = 1$ and 2
K_n	Dimensionality of the n-th mode of $\mathcal{X}_r$ for mode $n = 3, 4, ..., N$
J_n	Dimensionality of the n-th mode of core tensor
$[\mathbf{X}\ ;\ \mathbf{Y}]$	Horizontal concatenation of two matrices $\mathbf{X}$ and $\mathbf{Y}$
$\mathbf{X}_{::k_3...k_N}$	$(k_3, ..., k_N)$-th sliced matrix of size $I_1 \times I_2$
$\mathbf{U}_{::k_3...k_N}$	Left singular vector matrix of $\mathbf{X}_{::k_3...k_N}$
$\mathbf{\Sigma}_{::k_3...k_N}$	Singular value matrix of $\mathbf{X}_{::k_3...k_N}$
$\mathbf{V}_{::k_3...k_N}$	Right singular vector matrix of $\mathbf{X}_{::k_3...k_N}$
L	Number of sliced matrices ($= K_3 \times \cdots K_N$)
r	Number of singular values for SVD
N	Order of the given tensor
ϵ	Error tolerance in the iteration phase
t_{new}	New time-step in an online streaming setting
$\mathcal{X}_{old}$	Accumulated tensor
$\mathcal{X}_{new}$	New time slice at a time step t_{new}
T_{old}	Dimensionality of the temporal mode of an accumulated tensor ($\in I_1 \times I_2 \times K_3 \times ... \times T_{old}$)
T_{new}	Dimensionality of the temporal mode of a new time slice ($\in I_1 \times I_2 \times K_3 \times ... \times T_{new}$)
$blkdiag(\{\mathbf{A}_l\}_{l=1}^{L})$	Block diagonal matrix consisting of $\mathbf{A}_l$ for $l = 1, ...L$ (see Equation (10))
$\mathbf{A}_{old}^{(n)}$	Pre-existing factor matrix of the n-th mode in an online streaming setting
$\otimes$	Kronecker product
$\dagger$	Pseudoinverse

- **Experiment.** We experimentally show that D-Tucker 1) is up to 38.4× faster and requires up to 17.2× less space than competitors (see Fig. 3), and 2) provides good starting points to minimize the running time. Moreover, D-Tucker is scalable in handling dense tensors in terms of dimensionality, rank, order, and number of iterations. D-TuckerO is up to 6.1× faster than competitors in an online streaming setting (see Fig. 7).

In the rest of this paper, we describe the preliminaries in Section 2, propose our methods D-Tucker and D-TuckerO in Sections 3 and 4, respectively, present experimental results in Section 5, discuss related works in Section 6, and conclude in Section 7. The code and datasets are available at https://datalab.snu.ac.kr/dtucker.

2 PRELIMINARIES

We describe the preliminaries for tensor, SVD, and Tucker decomposition. Table 1 shows the symbols used.

2.1 Tensor

Each 'dimension' of a tensor (i.e., a multi-dimensional array) is denoted by *mode* or *way*. 'dimensionality' of a mode denotes the length of it. An N-mode or N-way tensor is represented as a boldface Euler script capital (e.g. $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$) letter, and matrices are denoted by boldface capitals (e.g. $\mathbf{A}$). A mode-n fiber is a vector having fixed indices except for the n-th index in a tensor. A sliced matrix is a matrix having fixed all indices except for two indices in a tensor.

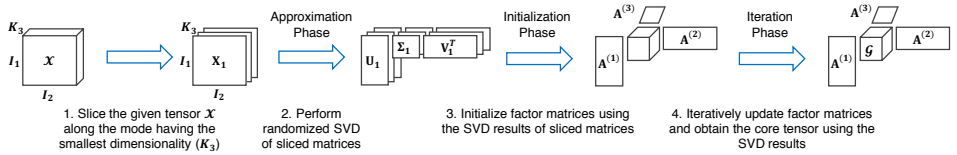


Fig. 1. Overview of D-Tucker. We first slice the given 3-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3}$ along the mode having the smallest dimensionality (K_3), and approximate sliced matrices using singular value decomposition (SVD). Then, we compute factor matrices using SVD results of sliced matrices in initialization step. We iteratively update factor matrices using SVD results of sliced matrices. After that, we obtain the core tensor using the updated factor matrices and SVD results of sliced matrices.

2.2 Tensor Operation

We use the following tensor operations in this paper: Frobenius norm, matricization, n-mode product, Kronecker product, and slicing.

Frobenius Norm. The Frobenius norm of $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times \ldots \times I_N}$) is denoted by $\|\mathcal{X}\|_F$ and defined as follows:

$$\|\mathcal{X}\|_F = \sqrt{\sum_{\forall (i_1,\ldots,i_N) \in \mathcal{X}} \mathcal{X}^2_{(i_1,\ldots,i_N)}}.$$

Matricization. Mode-n matricization converts a given tensor into a matrix form along n-th mode. We denote the mode-n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ as $\mathbf{X}_{(n)}$. Each element $(i_1, \ldots, i_N)$ of $\mathcal{X}$ is mapped to an element (i_n, j) of $\mathbf{X}_{(n)}$ such that

$$j = 1 + \sum_{\substack{k=1 \\ k \neq n}}^{N} \left((i_k - 1) \prod_{\substack{m=1 \\ m \neq n}}^{k-1} I_m \right),$$

where all indices start from 1.

n-mode product. The n-mode product $\mathcal{X} \times_n \mathbf{A}$ of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with a matrix $\mathbf{A} \in \mathbb{R}^{J_n \times I_n}$ has the size of $I_1 \times \cdots I_{n-1} \times J_n \times I_{n+1} \cdots \times I_N$, and defined by

$$(\mathcal{X} \times_n \mathbf{A})_{i_1 \ldots i_{n-1} j_n i_{n+1} \ldots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \ldots i_N} a_{j_n i_n}$$

where $a_{j_n i_n}$ is the (j_n, i_n)-th entry of $\mathbf{A}$. The result of n-mode product of a tensor $\mathcal{X}$ with a matrix $\mathbf{A}$ is identical to that of the following three operations: 1) performing mode-n matricization $\mathbf{X}_{(n)}$, 2) computing $\mathbf{Y}_{(n)} = \mathbf{A}\mathbf{X}_{(n)}$, and 3) reshaping the result $\mathbf{Y}_{(n)}$ as a tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times \cdots I_{n-1} \times J_n \times I_{n+1} \cdots \times I_N}$.

Kronecker product. Kronecker product of a matrix $\mathbf{A} \in \mathbb{R}^{p \times q}$ with a matrix $\mathbf{B} \in \mathbb{R}^{r \times s}$ produces the output $\mathbf{C} = \mathbf{A} \otimes \mathbf{B}$ of the size $pr \times qs$. Each element of the output is defined as follows:

$$\mathbf{C}_{r(t-1)+u, s(v-1)+w} = a_{t,v} \times b_{u,w} \tag{1}$$

where $a_{t,v}$ is (t, v)-th element of the matrix $\mathbf{A}$ and $b_{u,w}$ is (u, w)-th element of the matrix $\mathbf{B}$.

Slicing a tensor. Slicing an N-order tensor $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times \ldots \times I_N}$) along modes not in $\{m, n\}$ decomposes $\mathcal{X}$ into L sliced matrices of size $I_m \times I_n$, where $L = I_1 \times \ldots \times I_{m-1} \times I_{m+1} \times \ldots \times I_{n-1} \times I_{n+1} \times \ldots \times I_N$. For example, consider a 3-order tensor $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times I_2 \times K_3}$) in Fig. 1. Slicing $\mathcal{X}$ along mode 3 leads to K_3 sliced matrices of size $I_1 \times I_2$.

Algorithm 1: Randomized SVD [59]

Input: matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$, target rank k, and sampling parameters p and l
Output: SVD results $\mathbf{U} \in \mathbb{R}^{m \times k}$, $\Sigma \in \mathbb{R}^{k \times k}$, $\mathbf{V} \in \mathbb{R}^{n \times k}$

1: draw random matrices $\mathbf{\Omega} \in \mathbb{R}^{p \times m}$ and $\mathbf{\Psi} \in \mathbb{R}^{l \times n}$
2: form matrices $\mathbf{Y} = \mathbf{\Omega A}$ and $\mathbf{Z} = \mathbf{\Psi A}^T$
3: obtain column orthogonal matrices $\mathbf{Q}$ and $\mathbf{P}$ by QR factorization of $\mathbf{Y}^T$ and $\mathbf{Z}^T$.
4: form matrices $\mathbf{W} = \mathbf{\Omega P}$ and $\mathbf{B} = \mathbf{YQ}$.
5: obtain a matrix $\mathbf{X}$ which minimizes $\|\mathbf{WX} - \mathbf{B}\|$
6: compute SVD of $\mathbf{X} = \tilde{\mathbf{U}}\Sigma\tilde{\mathbf{V}}^T$
7: $\mathbf{U} \leftarrow \mathbf{P}\tilde{\mathbf{U}}_k$, $\Sigma \leftarrow \tilde{\Sigma}_k$, $\mathbf{V} \leftarrow \mathbf{Q}\tilde{\mathbf{V}}_k$

2.3 Singular Value Decomposition (SVD)

Given a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$, Singular Value Decomposition (SVD) decomposes it into the three matrices $\mathbf{U} \in \mathbb{R}^{m \times r}$, $\Sigma \in \mathbb{R}^{r \times r}$, and $\mathbf{V} \in \mathbb{R}^{n \times r}$ where $\mathbf{X}$ is equal to $\mathbf{U\Sigma V}^{\mathrm{T}}$. $\mathbf{U}$ is a column orthogonal matrix (i.e., $\mathbf{U}^T\mathbf{U} = \mathbf{I}$) consisting of left singular vectors of $\mathbf{X}$; Σ is an $r \times r$ diagonal matrix consisting of singular values σ_r where $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$. $\mathbf{V} \in \mathbb{R}^{n \times r}$ is a column orthogonal matrix (i.e., $\mathbf{V}^T\mathbf{V} = \mathbf{I}$) consisting of right singular vectors of $\mathbf{X}$.

SVD with randomized algorithm. Randomized SVD efficiently approximates a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with a low rank using randomization techniques (See Algorithm 1). The main idea of randomized SVD is 1) to generate random matrices $\mathbf{\Omega} \in \mathbb{R}^{p \times m}$ and $\mathbf{\Psi} \in \mathbb{R}^{l \times n}$ where p and l are sampling parameters, and find column orthogonal matrices $\mathbf{Q} \in \mathbb{R}^{n \times p}$ and $\mathbf{P} \in \mathbb{R}^{m \times l}$ of sketches $\mathbf{Y}^T = (\mathbf{\Omega A})^T \in \mathbb{R}^{n \times p}$ and $\mathbf{Z}^T = (\mathbf{\Psi A}^T)^T \in \mathbb{R}^{m \times l}$, respectively, 2) to construct a smaller matrices $\mathbf{W} = \mathbf{\Omega P} \in \mathbb{R}^{p \times l}$ and $\mathbf{B} = \mathbf{YQ} \in \mathbb{R}^{p \times p}$, and find $\mathbf{X} \in \mathbb{R}^{l \times p}$ that minimizes $\|\mathbf{WX} - \mathbf{B}\|$, 3) to compute $\mathbf{X} \approx \tilde{\mathbf{U}}_k\Sigma_k\tilde{\mathbf{V}}_k^T$ by truncated SVD at target rank k, and 4) to compute $\mathbf{U} = \mathbf{P}\tilde{\mathbf{U}}_k \in \mathbb{R}^{m \times k}$ and $\mathbf{V} = \mathbf{Q}\tilde{\mathbf{V}}_k \in \mathbb{R}^{n \times k}$. The dominant terms to compute randomized SVD are to form sketches $\mathbf{Y}$ and $\mathbf{Z}$. Recent works [14, 34] require $O(mn)$ time to construct random matrix and form matrix $\mathbf{Y}$ using sparse embedding matrix $\mathbf{\Omega} \in \mathbb{R}^{p \times m} = \mathbf{\Phi D}$.

- h: $[m] \rightarrow [p]$ is a random map so that $h(m') = p'$ for $p' \in [p]$ with probability $1/p$ for each $m' \in [m]$, where $[m] = \{1, 2, ..., m\}$ and $[p] = \{1, 2, ..., p\}$.
- $\mathbf{\Phi} \in \{0, 1\}^{p \times m}$: for each m'-th column of $\mathbf{\Phi}$, all the entries are 0 except that $h(m')$-th entry is 1; each column vector is a one-hot encoding vector whose only one entry is 1 and remaining entries are 0.
- Diagonal matrix $\mathbf{D} \in \mathbb{R}^{m \times m}$: diagonal entries are randomly chosen to be 1 or -1 with equal probability.

Due to the special form of $\mathbf{\Phi}$ and $\mathbf{D}$, the complexity of multiplying $\mathbf{\Omega}$ to $\mathbf{A}$ is $O(mn)$ (see [14] for details). $\mathbf{Z}$ is also constructed like $\mathbf{Y}$ using sparse embedding matrix. Therefore, the time complexity of randomized SVD is $O(mn)$ when we use sparse embedding matrices. In this paper, we use randomized SVD to efficiently deal with large dense matrices in the approximation phase. We use standard SVD [7] with time complexity $O(mnk)$ to stably deal with relatively small matrices in the initialization and iteration phases.

2.4 Tucker Decomposition

Definition 1 (Tucker Decomposition). *Given an N-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$, Tucker decomposition decomposes $\mathcal{X}$ into the core tensor $\mathcal{G} \in \mathbb{R}^{J_1 \times \ldots \times J_N}$ and factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ for $n = 1 \ldots N$.* □

Algorithm 2: Tucker-ALS (HOOI) [31]

Input: tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ and core tensor dimensionality $J_1, ..., J_N$
Output: core tensor $\mathcal{G} \in \mathbb{R}^{J_1,\ldots,J_N}$ and factor matrices $\mathbf{A}^{(i)}$ $(i = 1, ..., N)$

1: **initialize:** factor matrices $\mathbf{A}^{(i)}$ $(i = 1, ..., N)$
2: **repeat**
3: **for** $i = 1, ..., N$ **do**
4: $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{i-1} \mathbf{A}^{(i-1)T} \times_{i+1} \mathbf{A}^{(i+1)T} \cdots \times_N \mathbf{A}^{(N)T}$
5: $\mathbf{A}^{(i)} \leftarrow \mathbf{J}_i$ leading left singular vectors of $\mathbf{Y}_{(i)}$
6: **end for**
7: **until** the maximum iteration is reached, or the error ceases to decrease;
8: $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)T} \times_2 \mathbf{A}^{(2)T} \cdots \times_N \mathbf{A}^{(N)T}$

Note that $\mathbf{A}^{(n)}$ is a column orthogonal matrix, i.e. $\mathbf{A}^{(n)T}\ \mathbf{A}^{(n)} = \mathbf{I}$ where $\mathbf{I}$ is the identity matrix, and core tensor $\mathcal{G}$ is small and dense. The objective function of Tucker decomposition is given as follows.

$$\min_{\mathcal{G},\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}} ||\mathcal{X} - \mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)}|| \tag{2}$$

where we represent the given tensor $\mathcal{X}$ using the core tensor $\mathcal{G}$ and factor matrices $\mathbf{A}^{(n)}$:

$$\mathcal{X} \approx \mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)} \tag{3}$$

In addition, we re-express Equation (3) with matricization and Kronecker product as follows:

$$\mathbf{X}_{(n)} \approx \mathbf{A}^{(n)} \mathbf{G}_{(n)} (\otimes_{k \neq n}^{N} \mathbf{A}^{(k)T}) \tag{4}$$

where $(\otimes_{k \neq n}^{N} \mathbf{A}^{(k)T})$ indicates Kronecker product of $\mathbf{A}^{(k)T}$ for $k = N, N-1, ..., n+1, n-1, ..., 2, 1$.

Computing the Tucker decomposition. A common approach to minimize Equation (2) is ALS (Alternating Least Square). ALS approach iteratively updates the factor matrix of a mode while fixing all factor matrices of other modes. Algorithm 2 describes Tucker decomposition based on ALS approach, which is called higher-order orthogonal iteration (HOOI). A bottleneck of ALS approach for a dense tensor is to compute Equation (5) (line 4 in Algorithm 2) which requires $O(\prod_{m=1}^{N} I_m)$ space and $O(J_1 \times \prod_{m=1}^{N} I_m)$ computational time even to compute the first n-mode product between an input tensor $\mathcal{X}$ and the factor matrix $\mathbf{A}^{(1)}$.

$$\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)T} \cdots \times_{i-1} \mathbf{A}^{(i-1)T} \times_{i+1} \mathbf{A}^{(i+1)T} \cdots \times_N \mathbf{A}^{(N)T} \Leftrightarrow \mathbf{Y}_{(i)} \leftarrow \mathbf{X}_{(i)} \left(\otimes_{k \neq i}^{N} \mathbf{A}^{(k)}\right) \tag{5}$$

Note that Equation (5) re-expresses line 4 of Algorithm 2 with mode-i matricization and Kronecker product (see details in [28]). Moreover, the computational time grows as the number of iterations increases. Applying the naive Tucker-ALS is impractical in terms of time and space.

Recent works [34, 56] propose efficient tensor decomposition methods by approximating a large dense tensor to a small tensor, and updating the factor matrices and the core tensor using the small approximated tensor. MACH [56] updates them after randomly choosing elements of an input tensor, but an accuracy issue remains. Besides, it requires high time and space costs in an update phase. Malik et al. [34] proposed Tucker-ts which uses a sketching technique in updating factor matrices and a core tensor. Tucker-ts generates small sketching tensors for each mode, and uses them in the update phase. However, approximating a tensor with sketching requires a heavy computational cost since it performs sketching for all modes of the input tensor. In addition, scalability issues remain in the update phase since Tucker-ts generates large intermediate data when order N and rank J are large.

2.5 Streaming Tucker Decomposition

We formally define the problem of Tucker decomposition in an online streaming setting as follows:

Definition 2. (Tucker decomposition in a streaming fashion)
Given: *a time slice* $\mathcal{X}_{new} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times T_{new}}$ *at a time-step* t_{new}*, a pre-existing set of factor matrix* $\mathbf{A}^{(n)}_{old}$ *for* $n = 1, 2, ..., N$*, and a pre-existing core tensor* $\mathcal{G}_{old}$ *where* $\mathbf{A}^{(n)}_{old}$ *and* $\mathcal{G}_{old}$ *approximate* $\mathcal{X}_{old} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times T_{old}}$,
Update: *the factor matrix* $\mathbf{A}^{(n)}_{new}$ *for* $n = 1, 2, ..., N$ *and the core tensor* $\mathcal{G}_{new}$ *to approximate the accumulated tensor* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times T_{total}}$ *where* $T_{total} = T_{old} + T_{new}$.

$\mathbf{A}^{(n)}_{new} \in \mathbb{R}^{I_n \times J_n}$ (or $\mathbb{R}^{K_n \times J_n}$) for $n = 1, 2, ..., N-1$ is a factor matrix updated at t_{new}, $\mathbf{A}^{(N)}_{inc} \in \mathbb{R}^{T_{new} \times J_N}$ is the temporal factor matrix corresponding to t_{new}, and $\mathbf{A}^{(N)}_{new} = \begin{bmatrix} \mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc} \end{bmatrix} \in \mathbb{R}^{T_{total} \times J_N}$ is the temporal factor matrix corresponding to $t_{total} = t_{old} + t_{new}$ where $\mathbf{A}^{(N)}_{old} \in \mathbb{R}^{T_{old} \times J_N}$ is the pre-existing temporal factor matrix.

Computing the Tucker decomposition in an online streaming setting. We can deal with a newly arrived tensor using a static version of Tucker decomposition. However, it is inevitable that running times and memory requirements increase over time. Recent works have tried to update factor matrices and core tensor without the growth of the costs. DTA [52] updates factor matrices and core tensor by efficiently updating covariance matrices $\mathbf{X}^T_{(n)}\mathbf{X}_{(n)}$. STA [52] is an approximate version of DTA by exploiting SPIRIT [41] which efficiently deals with newly arrived vectors. Tucker-ts and Tucker-ttmts can be adapted to an online streaming setting: 1) approximating each newly arrived tensor using a sketching technique, and 2) updating factor matrices and core tensor using the approximated results of the whole tensor. Although they avoid increasing running time and memory requirements over time, there remains a need for accelerating the update process since computations involved with a large dense incoming tensor are still time-consuming. To efficiently update factor matrices and core tensor in an online streaming setting, we need to 1) prevent the increase of cost over time, 2) reduce the cost of approximating a newly arrived tensor, and 3) update them using the approximated results of the newly arrived tensor.

3 PROPOSED METHOD FOR STATIC TENSORS: D-TUCKER

We propose D-Tucker, a fast and memory-efficient Tucker decomposition method for large-scale dense tensors. We first give an overview of D-Tucker in Section 3.1. We describe details of D-Tucker in Sections 3.2 to 3.4. Finally, we analyze D-Tucker's complexities in Section 3.5.

3.1 Overview

D-Tucker efficiently computes Tucker decomposition of large dense tensors. The main challenges are as follows:

(1) **Exploiting the characteristics of real-world tensors.** Many real-world tensors are dense, provoking time and space problems. Furthermore, many real-world tensors are skewed (i.e., one of the dimensionality is much smaller than the others) and have low dimensional structures. How can we exploit such characteristics of real-world tensors to compress a dense input tensor with low computational cost and error?
(2) **Minimizing intermediate data.** Existing methods require heavy computations and large space while updating factor matrices and core tensor in the iteration phase. How can we minimize the size of intermediate data when updating the factor matrices and the core tensor?

Algorithm 3: D-Tucker

Input: tensor $\mathcal{X}$
Output: factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$, and core tensor $\mathcal{G}$
Parameters: rank J_i $(i = 1, 2, ..., N)$, and error tolerance ϵ
1: approximate slices of $\mathcal{X}$ by Algorithm 4
2: initialize factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$ by Algorithm 5
3: **repeat**
4: update factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$ and core tensor $\mathcal{G}$ by Algorithm 6
5: **until** the maximum iteration is reached, or the error difference is smaller than the error tolerance ϵ

(3) **Reducing numerical computation.** Tucker decomposition deals with a large number of tensor computations. How can we reduce the computational time of Tucker decomposition?

We address the above challenges with the following ideas:

(1) **Slicing an input tensor into matrices and computing randomized SVD of sliced matrices** minimize the computational cost and error, by utilizing the low dimensional structure of sliced matrices (Section 3.2).
(2) **Avoiding the reconstruction from SVD results** reduces the computational time as well as memory usage. By replacing a dense input tensor with SVD results of sliced matrices, we overcome a bottleneck of Tucker decomposition, n-mode product with a dense input tensor (Sections 3.3 and 3.4).
(3) **Careful ordering for matrix operations** reduces the memory usage and minimizes the computations. (Sections 3.3 and 3.4)

As shown in Fig. 1 and Algorithm 3, D-Tucker comprises three phases: approximation (Algorithm 4), initialization (Algorithm 5), and iteration (Algorithm 6). In the approximation phase, D-Tucker reorders modes of the input tensor in descending order for efficiency, extracts matrices of size $I_1 \times I_2$ by slicing the reordered tensor where I_1 and I_2 are the two largest dimensionalities, and performs randomized SVD of sliced matrices in order to support fast and memory-efficient Tucker decomposition (line 1 in Algorithm 3). In the initialization phase, we obtain initial factor matrices using the SVD results of sliced matrices (line 2 in Algorithm 3). This phase provides a good starting point for the iteration phase, reducing the number of iterations. In the iteration phase, we obtain the factor matrices and the core tensor using the initial factor matrices and the SVD results of sliced matrices (line 4 in Algorithm 3).

3.2 Approximation Phase

The main goal of the approximation phase is to compress the input tensor with low error; it enables the iteration phase to reduce the memory requirements and the number of flops. Given a large-scale dense tensor, previous works based on ALS approach require heavy computations and memory usage in updating a factor matrix at each iteration step since they directly process the given tensor. Although a few methods tried to solve the above problem by approximating the input tensor, they give high errors, or require heavy computations. The approximation phase of D-Tucker enables efficiently updating the factor matrices and the core tensor in the iteration phase based on two characteristics of real-world tensors: 1) skewed shape, and 2) low dimensional structure in sliced matrices. We reorder modes of a given tensor based on the first characteristic, and compress the sliced matrices of the reordered tensor using a fast dimensionality reduction technique, randomized SVD.

Algorithm 4: Approximation phase of D-Tucker

Input: tensor $\mathcal{X}$
Output: sets of SVD result $\mathbf{U}_{::k_3,\ldots,k_N}\Sigma_{::k_3,\ldots,k_N}\mathbf{V}^{\mathrm{T}}_{::k_3,\ldots,k_N}$ of sliced matrix $\mathbf{X}_{::k_3,\ldots,k_N}$
Parameters: rank r
1: reorder modes of the input tensor by dimensionality in descending order
2: extract the matrix $\mathbf{X}_{::k_3,\ldots,k_N} \in \mathbb{R}^{I_1 \times I_2}$ by slicing the reordered tensor where I_1 and I_2 are the two largest dimensionalities
3: **for each** $(k_3, \ldots, k_N)$ **do**
4: perform randomized SVD of $\mathbf{X}_{::k_3,\ldots,k_N} \simeq \mathbf{U}_{::k_3,\ldots,k_N}\Sigma_{::k_3,\ldots,k_N}\mathbf{V}^{\mathrm{T}}_{::k_3,\ldots,k_N}$
5: **end for**

Skewed shape of real-world tensors. A skewed shape, where there are gaps between the dimensionalities of modes, exists in many real-world tensors. For example, a 3-order Air Quality tensor (see Table 3) of size $(30648, 376, 6)$ in the form of (timestamp in second, location, atmospheric pollutants; measurement) has a skewed shape where the dimensionality of the last mode is much smaller than those of others. We reorder modes in descending order of dimensionality (line 1 in Algorithm 4). Reordered tensor is defined as follows:

Definition 3 (Reordered tensor $\mathcal{X}_r$). *Given an N-mode input tensor $\mathcal{X}$, we reorder the input tensor by dimensionality in descending order. We represent the reordered tensor as $\mathcal{X}_r \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_N}$ where I_1 and I_2 are the two largest dimensionalities, K_n for $n = 3, 4, \ldots, N$ are the remaining dimensionalities, and $I_1 \geq I_2 \geq K_3 \geq \cdots \geq K_N$.* □

This reordering helps minimize the output size of the approximation phase, which is described in the analysis of space complexity in Section 3.5.

Low dimensional structure in sliced matrices. Many real-world data represented as a matrix have a low dimensional structure since they have redundant and correlated components. Similarly, sliced matrices of a given real-world tensor for any two modes often have a low dimensional structure. For example, consider the 3-order Air Quality tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3}$ of size $(30648, 376, 6)$ in Table 3 containing (timestamp in second, location, atmospheric pollutants; measurement), sliced along modes 3. Out of the 6 sliced matrices, the ith sliced matrix $\mathbf{X}_{::i} \in \mathbb{R}^{I_1 \times I_2}$ indicates the matrix containing (timestamp in second, location; measurement) for the ith atmospheric pollutant. We observe that the number r of singular values to keep 90% energy of each sliced matrix is $(28, 8, 6, 7, 6, 18)$, which is much smaller than $I_1 = 30648$ and $I_2 = 360$. Note that the energy of a matrix $\mathbf{X}_{::i} \in \mathbb{R}^{I_1 \times I_2}$ is defined as $\sum_{r=1}^{min(I_1,I_2)} \sigma_r^2$ where σ_r is the rth singular value of $\mathbf{X}_{::i}$. This result indicates that the sliced matrices have low dimensional structures. D-Tucker compresses the given tensor by exploiting the low dimensional structure, achieving low errors. Moreover, this structure provides the following computational benefit: the approximation phase of D-Tucker yields faster performance by leveraging the randomized SVD [59] of sliced matrices. It enables us to avoid performing tensor decomposition methods for sub-tensors, which makes D-Tucker efficient since the tensor-based methods iteratively perform expensive operations such as n-mode product. Therefore, D-Tucker achieves high efficiency and low errors even on single-core systems.

We express a reordered tensor $\mathcal{X}_r \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_N}$ as a collection of sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$. We formally define the sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$ in Definition 4.

Definition 4 (Sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$). *Given a reordered tensor $\mathcal{X}_r \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_N}$, each sliced matrix of size $I_1 \times I_2$ is extracted by slicing the reordered tensor $\mathcal{X}_r$. The size of a sliced matrix $\mathbf{X}_{::k_3 \ldots k_N}$ is $I_1 \times I_2$ where I_1 is the number of rows and I_2 is the number of columns of the sliced matrix.* □

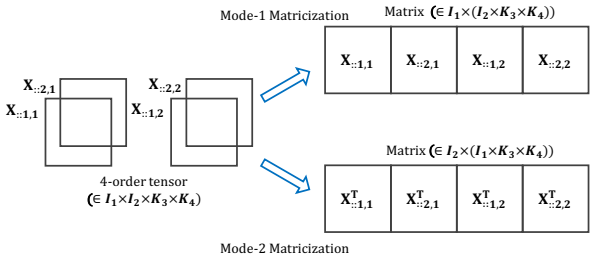


Fig. 2. Example of matricizing a 4-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times K_4}$ using sliced matrices for the first and the second mode when $K_3 = 2$ and $K_4 = 2$.

After slicing the tensor $\mathcal{X}_r$ into the matrices $\mathbf{X}_{::k_3 \ldots k_N}$, we decompose the sliced matrix using randomized SVD [59] with sparse embedding matrix [14, 34] (line 4 in Algorithm 4).

$$\mathbf{X}_{::k_3 \ldots k_N} \simeq \mathbf{U}_{::k_3 \ldots k_N} \mathbf{\Sigma}_{::k_3 \ldots k_N} \mathbf{V}^{\mathrm{T}}_{::k_3 \ldots k_N} \tag{6}$$

where $\mathbf{U}_{::k_3 \ldots k_N}$ $(\in \mathbb{R}^{I_1 \times r})$ is a left singular vector matrix, $\mathbf{\Sigma}_{::k_3 \ldots k_N}$ $(\in \mathbb{R}^{r \times r})$ is a singular value matrix, and $\mathbf{V}_{::k_3 \ldots k_N}$ $(\in \mathbb{R}^{I_2 \times r})$ is a right singular vector matrix. Note that the number r of singular values is much smaller than the dimensionalities I_1 and I_2. By computing Equation (6) for all sliced matrices, we achieve high efficiency in terms of time and space, to obtain factor matrices and core tensor. In the following initialization and iteration phases, we describe how to perform Tucker decomposition efficiently with the SVD results of sliced matrices rather than the raw input tensor.

3.3 Initialization Phase

The initialization phase, which initializes factor matrices of a given tensor $\mathcal{X}$, enables the iteration phase to reduce the number of iterations by providing a good starting point of the ALS algorithm. The main challenge is how to handle the SVD results for efficient initialization of the factor matrices. Truncated HOSVD has provided good initial factor matrices to compute Tucker decomposition based on ALS approach [31]. However, truncated HOSVD has limitations in efficiently initializing factor matrices using the SVD results because it cannot avoid reconstructing the reordered tensor for the mode $i = 3, 4, \ldots, N$ using the SVD results. To avoid reconstructing the reordered tensor using the SVD results, we apply Sequentially Truncated Higher-Order SVD (ST-HOSVD) [57], which is a variant of HOSVD. Note that ST-HOSVD obtains factor matrix $\mathbf{A}^{(i)}$ which contains left singular vectors of mode-i matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}} \cdots \times_{i-1} \mathbf{A}^{(i-1)\mathrm{T}}$. Applying ST-SHOVD allows us to efficiently initialize factor matrices using the results of the approximation phase, in contrast to HOSVD. The detail is described in initializing factor matrices for the remaining modes.

D-Tucker initializes factor matrices by obtaining the left singular vectors efficiently using the SVD results. For the first mode, we efficiently obtain the factor matrix by reusing the SVD results from the approximation phase. For the second mode, we efficiently compute mode-1 product between the first factor matrix and the SVD results by carefully ordering matrix multiplications, and then obtain the initial factor matrix. For the remaining modes, we process a small tensor computed by n-mode products between the SVD results and the factor matrices of the first and the second modes. These enable D-Tucker to achieve high efficiency in term of time and space. Note that we use standard SVD [7] in the initialization phase since the randomized SVD can decrease the effectiveness of the initialization. We describe the initialization of the first two modes corresponding to the dimensionalities I_1 and I_2, and then describe those of remaining modes [35].

Algorithm 5: Initialization phase of D-Tucker

Input: SVD results $\mathbf{U}_l$, Σ_l, and $\mathbf{V}_l$ for $l = 1, 2, ..., L$
where L is the number of sliced matrices
Output: initialized factor matrices $\mathbf{A}^{(i)}$ $(i = 1, 2, ..., N)$
Parameters: rank J_i $(i = 1, 2, ..., N)$

perform SVD of $\left[\mathbf{U}_1\Sigma_1; \cdots ; \mathbf{U}_L\Sigma_L\right] \simeq \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}$
$\mathbf{A}^{(1)} \leftarrow \mathbf{U}$
compute $\mathbf{Y}_{(2),inter} = \mathbf{A}^{(1)\mathrm{T}}\left[\mathbf{U}_1; \cdots ; \mathbf{U}_L\right]$
$\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{(2),inter}$
$\mathbf{Y}_{(2),inter} \leftarrow \mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$
$\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{(2),inter}, [J_1, I_2, K_3, \cdots, K_N])$
$\mathbf{A}^{(2)} \leftarrow J_2$ leading singular vectors of $\mathbf{Y}_{(2)}$
for $i \leftarrow 3$ to N **do**
  if $i = 3$ **then**
    $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{reuse} blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L})$
    $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{reuse}, [J_1, J_2, K_3, \cdots, K_N])$
  else
    $\mathcal{Y} \leftarrow \mathcal{Y}_{reuse} \times_{i-1} \mathbf{A}^{(i-1)\mathrm{T}}$
  end if
  $\mathbf{A}^{(i)} \leftarrow J_i$ leading singular vectors of $\mathbf{Y}_{(i)}$
  $\mathcal{Y}_{reuse} \leftarrow \mathcal{Y}$
end for

First mode. Our goal is to initialize the factor matrix of the first mode as left singular vectors of mode-1 matricization of $\mathcal{X}$. A naive approach would compute SVD of mode-1 matricization of $\mathcal{X}$. However, this approach requires heavy computation and high memory usage since it directly deals with large-scale dense tensor. Our idea is to avoid reconstructing $\mathcal{X}$ from the SVD results of sliced matrices, initializing the factor matrix of the first mode. Without the reconstruction, we reduce the computational cost and memory usage.

As shown in Fig. 2, we represent mode-1 matricized matrix $\mathbf{X}_{(1)}$ of the reordered tensor $\mathcal{X}_r$ as follows:

$$\mathbf{X}_{(1)} = \left[\mathbf{X}_{::1,\ldots,1}; \cdots ; \mathbf{X}_{::K_3,\ldots,K_N}\right] = \left[\mathbf{X}_1; \cdots ; \mathbf{X}_l; \cdots ; \mathbf{X}_L\right]$$

where L is equal to $K_3 \times \cdots \times K_N$, and the index l is defined as in Equation (7).

$$l = 1 + \sum_{i=3}^{N} \left((k_i - 1) \prod_{m=3}^{i-1} K_m \right) \tag{7}$$

where K_m is the dimensionality of mode-m, N is the order of the input tensor, and $\prod_{m=3}^{i-1} K_m$ is equal to 1 if $i - 1 < m$. Note that we represent a sliced matrix as $\mathbf{X}_l$ with the index l instead of $\mathbf{X}_{::k_3 \ldots k_N}$ for brevity. Using the SVD of sliced matrices, the mode-1 matricized matrix $\mathbf{X}_{(1)}$ is expressed as follows:

$$\mathbf{X}_{(1)} = \left[\mathbf{X}_1; \cdots ; \mathbf{X}_l; \cdots ; \mathbf{X}_L\right] \simeq \left[\tilde{\mathbf{X}}_1; \cdots ; \tilde{\mathbf{X}}_l; \cdots ; \tilde{\mathbf{X}}_L\right] \tag{8}$$

where $\tilde{\mathbf{X}}_l$ is a representation of $\mathbf{U}_l\Sigma_l\mathbf{V}_l^{\mathrm{T}}$. The computational cost to explicitly reconstruct the matrices $\tilde{\mathbf{X}}_l$ for $l = 1..L$ from SVD results and to obtain left singular vectors of $\mathbf{X}_{(1)}$ is expensive in terms of time and space. D-Tucker obtains left singular vectors of the first mode without the reconstruction of $\tilde{\mathbf{X}}_l$. The main idea is to carefully decouple $\mathbf{U}_l\Sigma_l$ and $\mathbf{V}_l^{\mathrm{T}}$, and perform SVD of a

concatenated matrix consisting of $\mathbf{U}_l\Sigma_l$ for $l = 1..L$. The above idea allows us to efficiently obtain left singular vectors of the concatenated matrix based on block structure [21, 23]. Performing SVD of the concatenated matrix ($\in \mathbb{R}^{I_1 \times (r K_3 \times \cdots \times K_N)}$) consisting of $\mathbf{U}_l\Sigma_l$ for $l = 1..L$ is more efficient than SVD of the mode-1 matricized matrix $\mathbf{X}_{(1)}$ ($\in \mathbb{R}^{I_1 \times (I_2 K_3 \times \cdots \times K_N)}$) (line 1 in Algorithm 5).

$$\mathbf{X}_{(1)} \simeq \left[\mathbf{U}_1\Sigma_1; \cdots ; \mathbf{U}_L\Sigma_L\right] \times (blkdiag(\{\mathbf{V_l}\}_{l=1}^{L}))^{\mathrm{T}} \simeq \mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}(blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}))^{\mathrm{T}} \tag{9}$$

where $\mathbf{U}\Sigma\mathbf{V}^{\mathrm{T}}$ is the SVD result of the concatenated matrix $\left[\mathbf{U}_1\Sigma_1; \cdots ; \mathbf{U}_L\Sigma_L\right]$, and the number L of sliced matrices is equal to $K_3 \times \cdots \times K_N$. $blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}) \in \mathbb{R}^{I_2 L \times rL}$ is a block diagonal matrix consisting of $\mathbf{V}_l \in \mathbb{R}^{I_2 \times r}$ for $l = 1, ..., L$:

$$blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}) = \begin{bmatrix} \mathbf{V}_1 & \mathbf{O} & \cdots & \mathbf{O} \\ \mathbf{O} & \mathbf{V}_2 & \cdots & \mathbf{O} \\ \vdots & \mathbf{O} & \ddots & \vdots \\ \mathbf{O} & \mathbf{O} & \cdots & \mathbf{V}_L \end{bmatrix} \tag{10}$$

$\mathbf{U}$ and $\mathbf{V}^{\mathrm{T}}(blkdiag(\{\mathbf{V}_l\}_{l=1}^{L}))^{\mathrm{T}}$ are column orthogonal and Σ has the property of singular value matrix, and thus the last term of Equation (9) has the SVD form. Therefore we obtain the initial factor matrix $\mathbf{A}^{(1)} = \mathbf{U}$ (line 2 in Algorithm 5).

Second mode. Our goal is to initialize the factor matrix of the second mode as left singular vectors of mode-2 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}}$ like ST-HOSVD. As in the first mode, a naive approach would compute SVD of mode-2 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}}$, but it has the same problems of heavy computational cost and high memory requirement. Our idea is to compute $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}}$ without reconstructing $\mathcal{X}$ from a set of SVD results of sliced matrices, and then compute SVD of mode-2 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}}$. By avoiding the reconstruction, we reduce the computational cost and memory usage to compute $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}}$.

To compute n-mode product for mode-1, we exploit SVD results computed from the approximation phase, instead of the given tensor, and then obtain left singular vectors for the second mode. In detail, we perform matrix multiplication between $\mathbf{A}^{(1)\mathrm{T}}$ and mode-1 matricized matrix described in Equation (8) as follows:

$$\begin{aligned} &\mathbf{A}^{(1)\mathrm{T}}\mathbf{X}_{(1)} \simeq \mathbf{A}^{(1)\mathrm{T}}\left[\mathbf{U}_1\Sigma_1\mathbf{V}_1^{\mathrm{T}}; \cdots ; \mathbf{U}_L\Sigma_L\mathbf{V}_L^{\mathrm{T}}\right] = \left(\mathbf{A}^{(1)\mathrm{T}}\left[\mathbf{U}_1; \cdots ; \mathbf{U}_L\right]\right) blkdiag(\{\Sigma_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L}) \\ &= \mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L}) \end{aligned} \tag{11}$$

where $\mathbf{Y}_{(2),inter} = \mathbf{A}^{(1)\mathrm{T}}\left[\mathbf{U}_1; \cdots ; \mathbf{U}_L\right]$, L is equal to $K_3 \times \cdots \times K_N$, and $blkdiag(\{\Sigma_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ is a block diagonal matrix consisting of $\Sigma_l\mathbf{V}_l^{\mathrm{T}}$. In Equation (11), $\mathbf{Y}_{(2),inter}$ is computed, and then multiplied with the block diagonal matrix. After reshaping the result of $\mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l\ \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ as a tensor $\mathcal{Y}$ of the size $J_1 \times I_2 \times K_3 \times \cdots \times K_N$, we compute left singular vectors of mode-2 matricized matrix $\mathbf{Y}_{(2)}$ to initialize $\mathbf{A}^{(2)}$ (lines 3 to 7 in Algorithm 5).

Remaining modes. For mode $i = 3, ..., N$, our goal is to initialize $\mathbf{A}^{(i)}$ as left singular vectors of mode-i matricization $\mathbf{Y}_{(i)}$ of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}} \cdots \times_{i-1} \mathbf{A}^{(i-1)\mathrm{T}}$. For a mode-$i$, explicitly computing $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}} \cdots \times_{i-2} \mathbf{A}^{(i-2)\mathrm{T}}$ is inefficient since it is computed for the previous mode-$(i-1)$. Our idea is to reuse the result of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}} \cdots \times_{i-2} \mathbf{A}^{(i-2)\mathrm{T}}$ to initialize the factor matrix of the mode-i.

Now, we describe how to obtain the factor matrix of the third mode, and then the factor matrix of the modes $i = 4, 5, ..., N$. For mode-3, the goal is to obtain $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}}$, and perform SVD.

Algorithm 6: Iteration phase of D-Tucker

Input: SVD results $\mathbf{U}_l$, Σ_l, and $\mathbf{V}_l$ ($l = 1, 2, ..., L$),
factor matrices $\mathbf{A}^{(i)}$ $(i = 1, ..., N)$, and core tensor $\mathcal{G}$
Output: updated factor matrices $\mathbf{A}^{(i)}$ $(i = 1, ..., N)$, and core tensor $\mathcal{G}$
Parameters: Rank J_i $(i = 1, ..., N)$

1: **for** $i \leftarrow 1$ to 2 **do**
2: **if** $i = 1$ **then**
3: $\mathbf{Y}_{(1),inter} \leftarrow \mathbf{A}^{(2)\mathrm{T}} \left[\mathbf{V}_1; \cdots; \mathbf{V}_L\right]$
4: $\mathbf{Y}_{(1),inter} \leftarrow \mathbf{Y}_{(1),inter} blkdiag(\{\Sigma_l \mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$
5: $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{(1),inter}, [I_1, J_2, K_3, \cdots, K_N])$
6: **else**
7: $\mathbf{Y}_{(2),inter} \leftarrow \mathbf{A}^{(1)\mathrm{T}} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]$
8: $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{(2),inter}$
9: $\mathbf{Y}_{(2),inter} \leftarrow \mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$
10: $\mathcal{Y} \leftarrow reshape(\mathbf{Y}_{(2),inter}, [J_1, I_2, K_3, \cdots, K_N])$
11: **end if**
12: $\mathcal{Y} \leftarrow \mathcal{Y} \times_3 \mathbf{A}^{(3)\mathrm{T}} \cdots \times_N \mathbf{A}^{(N)\mathrm{T}}$
13: $\mathbf{A}^{(i)} \leftarrow J_i$ leading singular vectors of $\mathbf{Y}_{(i)}$
14: **end for**
15: $\mathbf{Y}_{reuse} \leftarrow \mathbf{Y}_{reuse} blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L})$
16: $\mathcal{Y}_{reuse} \leftarrow reshape(\mathbf{Y}_{reuse}, [J_1, J_2, K_3, \cdots, K_N])$
17: **for** $i \leftarrow 3$ to N **do**
18: $\mathcal{Y} \leftarrow \mathcal{Y}_{reuse} \times_3 \mathbf{A}^{(3)\mathrm{T}} \cdots \times_{i-1} \mathbf{A}^{(i-1)\mathrm{T}} \times_{i+1} \mathbf{A}^{(i+1)\mathrm{T}} \cdots \times_N \mathbf{A}^{(N)\mathrm{T}}$
19: $\mathbf{A}^{(i)} \leftarrow J_i$ leading singular vectors of $\mathbf{Y}_{(i)}$
20: **end for**
21: $\mathcal{G} \leftarrow \mathcal{Y}_{reuse} \times_3 \mathbf{A}^{(3)\mathrm{T}} \cdots \times_N \mathbf{A}^{(N)\mathrm{T}}$

The following equation re-expresses the mode-1 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}}$.

$$\begin{aligned} &\mathbf{A}^{(1)\mathrm{T}} \mathbf{X}_{(1)} blkdiag(\{\mathbf{A}^{(2)}\}_{l=1}^{L}) \simeq \left(\mathbf{A}^{(1)\mathrm{T}} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]\right) blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L}) \\ &= \mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}} \mathbf{A}^{(2)}\}_{l=1}^{L}) \end{aligned} \tag{12}$$

Note that we save $\mathbf{Y}_{reuse} = \left(\mathbf{A}^{(1)\mathrm{T}} \left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]\right)$ to reuse when computing Equation (12) (line 4 in Algorithm 5). After computing Equation (12), we 1) reshape the result as a tensor $\mathcal{Y}$ of size $J_1 \times J_2 \times K_3 \times \cdots \times K_N$, 2) perform SVD of $\mathbf{Y}_{(3)}$, and 3) store $\mathcal{Y}$ as $\mathcal{Y}_{reuse}$ for remaining modes (lines 10, 11, 15, and 16 in Algorithm 5).

Next, factor matrices for mode $i = 4, 5, ..., N$ are initialized by using the result of the previous mode. For mode i, we compute $\mathcal{Y}_{reuse} \times_{i-1} \mathbf{A}^{(i-1)T}$, and then perform SVD of mode-i matricization of $\mathcal{Y}_{reuse} \times_{i-1} \mathbf{A}^{(i-1)T}$. Since $\mathbf{Y}_{(i)}$ is much smaller than an input tensor $\mathcal{X}$, we efficiently initialize factor matrices $\mathbf{A}^{(i)}$ for $i = 3, ..., N$. This is the reason why we apply ST-HOSVD, not HOSVD which requires high computational costs to compute left singular vectors for the remaining modes $i = 3, ..., N$. HOSVD needs to perform SVD of mode-i matricization of $\mathcal{X}$. Then, we initialize the factor matrix $\mathbf{A}^{(i)}$ as the left singular vectors of the SVD result (line 15 in Algorithm 5).

3.4 Iteration Phase

The goal of the iteration phase is to alternately update factor matrices and compute core tensor by efficiently computing n-mode products in lines 4 and 8 of Algorithm 2. As described in Section 2.4, a naive ALS approach is much inefficient in terms of time and space due to large intermediate tensor

including the input tensor. Furthermore, increasing the number of iterations affect the overall running time. Therefore, the main challenge of the iteration phase is how to reduce the number of flops by minimizing the intermediate data. Our ideas to tackle the challenge are to 1) exploit the special structure of SVD results, 2) careful ordering of matrix multiplications, and 3) avoid redundant computations for the first and second modes.

Our ideas allow D-Tucker to be less affected by the number of iterations, and to avoid rapid growth of computational time as the number of iterations increases, due to the small amount of computations. We describe how to update 1) the factor matrices of the first two modes corresponding to the dimensionalities I_1 and I_2, and 2) those of other modes and the core tensor. Note that we use standard SVD [7] for stable convergence in the iteration phase.

First mode. Consider updating the first factor matrix $\mathbf{A}^{(1)}$. We use the initialized factor matrices and SVD results of the sliced matrices for $\mathbf{A}^{(1)}$. Following line 4 of Algorithm 2, we efficiently compute n-mode product for mode-2 using SVD results obtained in the approximation phase instead of the given tensor, and then perform products for remaining modes 3, 4, ..., N. We matricize the tensor along mode-2 with the sliced matrices as follows:

$$\mathbf{X}_{(2)} = \left[\mathbf{X}_1^\mathrm{T}; \cdots; \mathbf{X}_l^\mathrm{T}; \cdots; \mathbf{X}_L^\mathrm{T}\right] \simeq \left[\tilde{\mathbf{X}}_1^\mathrm{T}; \cdots; \tilde{\mathbf{X}}_l^\mathrm{T}; \cdots; \tilde{\mathbf{X}}_L^\mathrm{T}\right]$$

After that, we perform matrix multiplication between $\mathbf{A}^{(2)\mathrm{T}}$ and the mode-2 matricized matrix as follows:

$$\begin{aligned}\mathbf{A}^{(2)\mathrm{T}}\mathbf{X}_{(2)} &\simeq \mathbf{A}^{(2)\mathrm{T}}\left[\mathbf{V}_1\Sigma_1\mathbf{U}_1^\mathrm{T}; \cdots; \mathbf{V}_L\Sigma_L\mathbf{U}_L^\mathrm{T}\right] = \left(\mathbf{A}^{(2)\mathrm{T}}\left[\mathbf{V}_1; \cdots; \mathbf{V}_L\right]\right) blkdiag(\{\Sigma_l\mathbf{U}_l^\mathrm{T}\}_{l=1}^L) \\ &= \mathbf{Y}_{(1),inter}\, blkdiag(\{\Sigma_l\mathbf{U}_l^\mathrm{T}\}_{l=1}^L)\end{aligned} \tag{13}$$

where $\mathbf{Y}_{(1),inter} = \mathbf{A}^{(2)\mathrm{T}}\left[\mathbf{V}_1; \cdots; \mathbf{V}_L\right]$, L is equal to $K_3 \times \cdots \times K_N$, and $blkdiag(\{\Sigma_l\mathbf{U}_l^\mathrm{T}\}_{l=1}^L)$ is a block diagonal matrix consisting of $\Sigma_l\mathbf{U}_l^\mathrm{T}$. In Equation (13), we compute $\mathbf{Y}_{(1),inter}$, and multiply it with the block diagonal matrix. Then, we reshape the result of $\mathbf{Y}_{(1),inter}\, blkdiag(\{\Sigma_l\, \mathbf{U}_l^\mathrm{T}\}_{l=1}^L)$ as $\mathcal{Y}$ of size $I_1 \times J_2 \times K_3 \times \cdots \times K_N$ (lines 3 to 5 in Algorithm 6). After that, we perform the remaining n-mode products with $\mathcal{Y}$ for $n = 3, 4, ..., N$, and then update the factor matrix $\mathbf{A}^{(1)}$ by computing SVD of mode-1 matricized matrix $\mathbf{Y}_{(1)}$ (lines 12 and 13 in Algorithm 6).

Second mode. Next, to update $\mathbf{A}^{(2)}$, we compute n-mode product for mode-1 using SVD results obtained in the approximation phase instead of the given tensor. Then, we perform n-mode products for remaining modes 3, 4, ..., N. As in Equation (11), we perform matrix multiplication between $\mathbf{A}^{(1)\mathrm{T}}$ and the mode-1 matricized matrix which is the matricization of the tensor along mode-1 with the sliced matrices in Equation (8). For efficiency, we compute Equation (11) with the following order: 1) $\mathbf{Y}_{(2),inter} = \mathbf{A}^{(1)\mathrm{T}}\left[\mathbf{U}_1; \cdots; \mathbf{U}_L\right]$, 2) multiply it with the block diagonal matrix $blkdiag(\{\Sigma_l\mathbf{V}_l^\mathrm{T}\}_{l=1}^L)$, and 3) reshape the result of $\mathbf{Y}_{(2),inter} blkdiag\ (\{\Sigma_l\ \mathbf{V}_l^\mathrm{T}\}_{l=1}^L)$ as $\mathcal{Y}$ ($\in \mathbb{R}^{J_1 \times I_2 \times K_3 \times \cdots \times K_N}$) (lines 7, 9, and 10 in Algorithm 6). Note that $\mathbf{Y}_{(2),inter}$ is reused when computing $\mathbf{A}^{(i)}$ for $i = 3, 4, ..., N$ and the core tensor (line 8 in Algorithm 6). We update $\mathbf{A}^{(2)}$ by performing the remaining n-mode products with $\mathcal{Y}$ for $n = 3, 4, ..., N$, and computing SVD of mode-2 matricized matrix $\mathbf{Y}_{(2)}$ (lines 12 and 13 in Algorithm 6).

Remaining modes and core tensor. Consider updating factor matrices $\mathbf{A}^{(i)}$ for all $i = 3, 4, ..., N$, and the core tensor $\mathcal{G}$. The mode-1 matricization of $\mathcal{X} \times_1 \mathbf{A}^{(1)\mathrm{T}} \times_2 \mathbf{A}^{(2)\mathrm{T}}$ is given by Equation (12). In computing Equation (12), reusing the saved $\mathbf{Y}_{(2),inter}$ at line 8 of Algorithm 6 allows us to avoid redundant computation, sufficiently reducing computational costs; the reason is that $\mathbf{Y}_{(2),inter}$ is much smaller than the input tensor $\mathcal{X}$ and the SVD results of sliced matrices. We compute $\mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l\mathbf{V}_l^\mathrm{T}\mathbf{A}^{(2)}\}_{l=1}^L)$ and reshape the result $\mathcal{Y}_{reuse}$ of size $J_1 \times J_2 \times K_3 \cdots \times K_N$ once, which is reused to compute factor matrices $\mathbf{A}^{(i)}$ for $i = 3, 4, ..., N$ and the core tensor $\mathcal{G}$ (lines 15

Table 2. Time and space costs of D-Tucker and competitors. Space cost indicates the requirement for updating factor matrices and the core tensor. Boldface denotes the optimal complexities. I denotes the two largest dimensionalities, K is the remaining dimensionalities, M is the number of iterations, J is the dimensionality of the core tensor, and N is the order of the given tensor.

Algorithm	Time	Space
D-Tucker	$\mathbf{O(I^2K^{N-2} + MNIK^{N-2}J^2)}$	$O(IK^{N-2}J)$
Tucker-ALS [30]	$O(MNI^2K^{N-2}J)$	$O(I^2K^{N-2})$
MACH [56]	$O(MNI^2K^{N-2}J)$	$O(I^2K^{N-2})$
RTD [11]	$O(MNI^2K^{N-2})$	$O(I^2K^{N-2})$
Tucker-ts [34]	$O(NI^2K^{N-2} + MN(IJ^N + J^{2N}))$	$O(NIJ^N + J^{2N})$
Tucker-ttmts [34]	$O(NI^2K^{N-2} + MN(IJ^{2N-2} + J^{2N-2}))$	$O(NIJ^N + J^{2N-1})$

and 16 in Algorithm 6). For $i = 3, ..., N$, we update $\mathbf{A}^{(i)}$ by performing the remaining n-mode products, and SVD of $\mathbf{Y}_{(i)}$ (lines 17 to 20 in Algorithm 6). In addition, we update the core tensor by performing n-mode products between the reshaped tensor $\mathcal{Y}_{reuse}$ ($\in \mathbb{R}^{J_1 \times J_2 \times K_3 \cdots \times K_N}$) and $\mathbf{A}^{(n)\mathbf{T}}$ for all $n = 3, 4, ..., N$ (line 21 in Algorithm 6).

3.5 Theoretical Analysis

We theoretically analyze the time complexity, the space complexity, and the error of D-Tucker, as summarized in Table 2. For brevity, we assume $I_1 = I_2 = I$, $K_1 = K_2 = ... = K_N = K$, $r = J_1 = J_2 = ... = J_N = J$.

Time complexity. We analyze the time complexities of D-Tucker in Theorem 1.

Lemma 1. *The approximation phase of D-Tucker takes $O(I^2K^{N-2})$ where I is the largest dimensionality, and K is the remaining dimensionality.* □

Proof. See the proof in Appendix A.1. □

Lemma 2. *The initialization phase of D-Tucker takes $O(IK^{N-2}J^2)$ where I is the largest dimensionality, K is the remaining dimensionality, and J is the rank.* □

Proof. See the proof in Appendix A.2. □

Lemma 3. *The time complexity of an iteration at the iteration phase is $O(NIK^{N-2}J^2)$ where N is the order of a given tensor, I is the largest dimensionality, K is the remaining dimensionality, and J is the rank.* □

Proof. See the proof in Appendix A.3. □

Theorem 1. *The total time complexity of D-Tucker is $O(I^2K^{N-2}J + MNIK^{N-2}J^2)$ where M is the number of iteration, N is the order of a given tensor, I is the largest dimensionality, K is the remaining dimensionality, and J is the rank.* □

Proof. See the proof in Appendix B.1. □

Note that the time complexity of the approximation phase of D-Tucker is proportional only to the size I^2K^{N-2} of the input tensor without any parameters such as rank J and order N. Also, D-Tucker is less affected by the number of iterations because the time complexity $O(NIK^{N-2}J^2)$ per iteration of the iteration phase is much smaller than the time complexity $O(I^2K^{N-2})$ of the approximation phase: I is much larger than NJ^2 since $I \gg J$ and $I \gg N$. Thus, D-Tucker avoids rapid growth of computational time as the number of iterations increases.

Space complexity. We analyze space requirements of D-Tucker for initializing and updating factor matrices.

THEOREM 2. *D-Tucker requires $O(IK^{N-2}J)$ space for initializing and updating factor matrices.* □

PROOF. See the proof in Appendix B.2. □

Note that the original input tensor requires $O(I^2K^{N-2})$ space. Thanks to the reordering in the approximation phase, the space complexity of D-Tucker is I/J times smaller than directly using the raw input tensor. Without the reordering, the compression rate would worsen; e.g., if we have decomposed sliced matrices of size $I \times K_n$, the compression rate would have decreased to K_n/J, which is worse than I/J since $I > K_n > J$.

4 PROPOSED METHOD FOR ONLINE TENSORS: D-TUCKERO

4.1 Overview

We propose D-TuckerO, an efficient Tucker decomposition method in an online streaming setting. Our goal is to design D-TuckerO to efficiently update factor matrices and core tensor for a new incoming tensor slice. The main challenges that need to be tackled for an efficient Tucker decomposition method in an online streaming setting are as follows:

(1) **Preventing the increase of costs over time.** How can we prevent increasing the computational cost and space cost as tensors continuously arrive over time?
(2) **Accelerating updates.** How can we accelerate the update process for each incoming time slice?

We address the challenges with the following main ideas:

(1) **Avoiding explicit computations with $\mathcal{X}_{old}$ and $\mathbf{A}^{(N)}_{old}$** enables D-Tucker to update factor matrices and core tensor without increasing the costs where $\mathcal{X}_{old}$ and $\mathbf{A}^{(N)}_{old}$ are a pre-existing tensor and a pre-existing temporal factor matrix, respectively.
(2) **Applying the approximation phase for an incoming time slice** accelerates the update procedure for factor matrices and core tensor.

As shown in Algorithm 7, D-TuckerO efficiently updates factor matrices when a new incoming tensor is given. We present an efficient update procedure for each new incoming tensor in Section 4.2, and then describe how to apply the approximation phase to the update procedure in Section 4.3. Lastly, we analyze the time and space complexities of D-TuckerO. For brevity, we set the last mode N as the temporal mode when an N-order tensor repeatedly comes.

4.2 Efficient Update for Time Slice

Our goal is to update factor matrices and the core tensor for a new incoming tensor slice $\mathcal{X}_{new}$. D-TuckerO alternately updates factor matrices, and core tensor as in ALS algorithm; D-TuckerO updates the n-th factor matrix while fixing the other factor matrices and core tensor. We present how to update the temporal factor matrix $\mathbf{A}^{(N)}$ and then factor matrices of non-temporal modes.

Temporal Mode. Consider updating the temporal factor matrix $\mathbf{A}^{(N)}$. A naive approach is to update it by computing lines 4 and 5 in Tucker-ALS. However, dealing with an accumulated tensor $\mathcal{X}$ is impractical since the size of the tensor $\mathcal{X}$ increases over time. To efficiently update the factor without dealing with the accumulated tensor, we only update a part of the temporal factor matrix, i.e., $\mathbf{A}^{(N)}_{inc}$, corresponding to t_{new}. Lemma 4 describe an update rule for $\mathbf{A}^{(N)}_{inc}$.

Algorithm 7: Update phase of D-TuckerO

Input: a time slice $\mathcal{X}_{new} \in \mathbb{R}^{I_1 \times I_2 \times K_3 \times \cdots \times K_{N-1} \times t_{new}}$, a pre-existing set $\mathcal{A}$ of factor matrix $\mathbf{A}^{(n)}_{old}$ $(n = 1, ..., N)$, and core tensor $\mathcal{G}_{old}$, a set of $\mathbf{P}^{(n)}_{old}$, $\mathbf{Q}^{(n)}_{old}$ for $n = 1, ..., N-1$, $\mathbf{P}^{(N+1)}_{old}$, and $\mathbf{Q}^{(N+1)}_{old}$
Output: updated factor matrices $\mathbf{A}^{(n)}_{new}$ $(n = 1, ..., N)$ and core tensor $\mathcal{G}_{new}$
Parameters: Rank J_i $(i = 1, ..., N)$

1: obtain SVD results $\mathbf{U}_l$, Σ_l, and $\mathbf{V}_l$ ($l = 1, 2, ..., L$) of $\mathcal{X}_{new}$ using the approximation phase.
2: obtain $\mathbf{A}^{(N)}_{new}$ by computing Equation (14) with the SVD results of $\mathcal{X}_{new}$
3: **for** $n \leftarrow 1$ to $N-1$ **do**
4: obtain $\mathbf{A}^{(n)}_{new}$ by computing Equation (15) with the SVD results of $\mathcal{X}_{new}$
5: update $\mathbf{P}^{(n)}_{old} \leftarrow \mathbf{P}^{(n)}_{old} + \mathbf{P}^{(n)}_{new}$ by Equation (19)
6: update $\mathbf{Q}^{(n)}_{old} \leftarrow \mathbf{Q}^{(n)}_{old} + \mathbf{Q}^{(n)}_{new}$ by Equation (23)
7: **end for**
8: obtain $\mathcal{G}_{new}$ by computing Equation (24) with the SVD results of $\mathcal{X}_{new}$
9: update $\mathbf{P}^{(N+1)}_{old} \leftarrow \mathbf{P}^{(N+1)}_{old} + \mathbf{P}^{(N+1)}_{new}$ by Equation (25)
10: update $\mathbf{Q}^{(N+1)}_{old} \leftarrow \mathbf{Q}^{(N+1)}_{old} + \mathbf{Q}^{(N+1)}_{new}$ by Equation (26)

Lemma 4 (Update rule for temporal mode). *When fixing all non-temporal factor matrices, $\mathbf{A}^{(N)}_{inc}$ is updated as follows:*

$$\mathbf{A}^{(N)}_{inc} \leftarrow \mathbf{X}_{(N),new} \left(\otimes_{k=1}^{N-1} \left(\mathbf{A}^{(k)T} \right)^{\dagger} \right) \mathbf{G}^{\dagger}_{(N)} \tag{14}$$

where $\dagger$ indicates a pseudo-inverse of a matrix, and $(\otimes_{k=1}^{N-1} \left(\mathbf{A}^{(k)T}\right)^{\dagger})$ indicates the entire Kronecker product of $\left(\mathbf{A}^{(k)T}\right)^{\dagger}$ for $k = N-1, N-2, ..., 2, 1$. □

Proof. See the proof in Appendix A.4. □

Since $\mathbf{A}^{(N)}_{old}$ is already computed at the previous step, we only compute $\mathbf{A}^{(N)}_{inc}$ using $\mathcal{X}_{new}$, $\mathbf{G}_{(N)}$, and $\mathbf{A}^{(n)}$ for $n = 1, 2, ..., N-1$. In updating the temporal factor matrix $\mathbf{A}^{(N)}$, we exploit $\mathbf{G}_{(N),old}$ and $\mathbf{A}^{(n)}_{old}$ for $n = 1, 2, ..., N-1$ to compute $\mathbf{G}_{(N)}$ and $(\otimes_{k=1}^{N-1} \left(\mathbf{A}^{(k)T}\right)^{\dagger})$, respectively. In Equation (14), we compute 1) $\left(\mathbf{A}^{(k)T}\right)^{\dagger}$ for $k = 1, ..., N-1$, 2) the Kronecker product, and 3) matrix multiplication between $\mathbf{X}_{(N),new}$, the result of the Kronecker product, and $\mathbf{G}^{\dagger}_{(N),old}$ in Equation (14).

Non-temporal Modes. Our goal is to update $\mathbf{A}^{(n)}$ when a new incoming tensor $\mathcal{X}_{new}$ is given. By avoiding explicit computations with $\mathcal{X}_{old}$ and $\mathbf{A}^{(N)}_{old}$ whose size increases over time, we efficiently update $\mathbf{A}^{(n)}$. We first introduce an update rule for $\mathbf{A}^{(n)}$, and then provide details on efficiently computing $\mathbf{A}^{(n)}$ based on the rule.

Lemma 5 (Update rule for non-temporal mode). *When fixing $\mathbf{A}^{(k)}$ for $k = 1, ..., n-1, n+1, ..., N$, $\mathbf{A}^{(n)}_{new}$ is updated as follows:*

$$\mathbf{A}^{(n)}_{new} \leftarrow \mathbf{P}^{(n)} \left(\mathbf{Q}^{(n)} \right)^{-1} \tag{15}$$

where $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$ are equal to $\mathbf{X}_{(n)}(\otimes^{N}_{k \neq n} \mathbf{A}^{(k)})\mathbf{G}^{T}_{(n)}$ and $\left(\mathbf{G}_{(n)}(\otimes^{N}_{k \neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}\right)$, respectively. □

Proof. See the proof in Appendix A.5. □

To update $\mathbf{A}^{(n)}$, we compute $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$ in Equation (15). However, a naive computation for Equation (15) is impractical since the size of $\mathbf{X}_{(n)}$ and $\mathbf{A}^{(N)}$ increases over time. To achieve the efficiency, we avoid explicit computations with $\mathbf{X}_{(n),old}$ and $\mathbf{A}^{(N)}_{old}$ decoupled from $\mathbf{X}_{(n)}$ and $\mathbf{A}^{(N)}$, respectively, in $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$.

We now describe details on efficient computations of $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$. Given $\mathbf{P}^{(n)}$, we divide it into $\mathbf{P}^{(n)}_{old}$ and $\mathbf{P}^{(n)}_{new}$ where $\mathbf{P}^{(n)}_{old}$ and $\mathbf{P}^{(n)}_{new}$ are equal to Equations (17) and (18), respectively.

$$\mathbf{P}^{(n)} = \begin{bmatrix}\mathbf{X}_{(n),old} & \mathbf{X}_{(n),new}\end{bmatrix} \times \left(\begin{bmatrix}\mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc}\end{bmatrix} \otimes (\otimes^{N-1}_{k\neq n}\mathbf{A}^{(k)}) \right) \mathbf{G}^{T}_{(n)} \tag{16}$$

$$= \left(\mathbf{X}_{(n),old}(\mathbf{A}^{(N)}_{old} \otimes (\otimes^{N-1}_{k\neq n}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}\right) \tag{17}$$

$$+ \left(\mathbf{X}_{(n),new}(\mathbf{A}^{(N)}_{inc} \otimes (\otimes^{N-1}_{k\neq n}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}\right) \tag{18}$$

$$= \mathbf{P}^{(n)}_{old} + \mathbf{P}^{(n)}_{new} \tag{19}$$

We only compute $\mathbf{P}^{(n)}_{new}$ for Equation (19) as $\mathbf{P}^{(n)}_{old}$ is computed and stored at the previous step. $\mathbf{P}^{(n)}$ is used as $\mathbf{P}^{(n)}_{old}$ at the next step.

Next, we efficiently compute $\mathbf{Q}^{(n)}$; we divide $\mathbf{Q}^{(n)}$ into $\mathbf{Q}^{(n)}_{old}$ and $\mathbf{Q}^{(n)}_{new}$ which are equal to Equations (21) and (22), respectively.

$$\mathbf{Q}^{(n)} = \mathbf{G}_{(n)} \left(\begin{bmatrix}\mathbf{A}^{(N)T}_{old} & \mathbf{A}^{(N)T}_{inc}\end{bmatrix} \begin{bmatrix}\mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc}\end{bmatrix} \right) \otimes \left(\otimes^{N-1}_{k\neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})\right) \mathbf{G}^{T}_{(n)} \tag{20}$$

$$= \mathbf{G}_{(n)} \left((\mathbf{A}^{(N)T}_{old}\mathbf{A}^{(N)}_{old}) \otimes (\otimes^{N-1}_{k\neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})) \right) \mathbf{G}^{T}_{(n)} \tag{21}$$

$$+ \mathbf{G}_{(n)} \left((\mathbf{A}^{(N)T}_{inc}\mathbf{A}^{(N)}_{inc})(\otimes^{N-1}_{k\neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})) \right) \mathbf{G}^{T}_{(n)} \tag{22}$$

$$= \mathbf{Q}^{(n)}_{old} + \mathbf{Q}^{(n)}_{new} \tag{23}$$

Similar to $\mathbf{P}^{(n)}$, $\mathbf{Q}^{(n)}_{old}$ is computed and stored at the previous step. We only compute $\mathbf{Q}^{(n)}_{new}$ for Equation (23). $\mathbf{Q}^{(n)}$ is also used as $\mathbf{Q}^{(n)}_{old}$ at the next step.

Core tensor. After updating factor matrices, we update the core tensor with Lemma 6. By avoiding explicit computations with $\mathcal{X}_{old}$ and $\mathbf{A}^{(N)}_{old}$, we efficiently update $\mathcal{G}$. We first derive an equation for updating the core tensor, and then describe how to efficiently update it.

Lemma 6 (Update rule for core tensor). *When fixing all factor matrices, we update the core tensor with the following equation:*

$$\mathbf{G}_{(N)} = \left(\mathbf{Q}^{(N+1)}\right)^{-1} \mathbf{P}^{(N+1)} \tag{24}$$

where $\mathbf{P}^{(N+1)}$ *and* $\mathbf{Q}^{(N+1)}$ *are equal to* $\mathbf{A}^{(N)T}\mathbf{X}_{(N)}(\otimes^{N-1}_{k=1}\mathbf{A}^{(k)}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})^{-1})$ *and* $\left(\mathbf{A}^{(N)T}\mathbf{A}^{(N)}\right)$, *respectively. Note that* $(N+1)$ *in* $\mathbf{P}^{(N+1)}$ *and* $\mathbf{Q}^{(N+1)}$ *corresponds to the core tensor.* □

Proof. See the proof in Appendix A.6. □

A naive computation for Equation (24) is expensive due to $\mathcal{X}$ and $\mathbf{A}^{(N)}$ corresponding to t_{total}. Therefore, we precisely divide $\mathbf{P}^{(N+1)}$ and $\mathbf{Q}^{(N+1)}$ to avoid computing the terms related to $\mathcal{X}_{old}$ and

$\mathbf{A}^{(N)}_{old}$. $\mathbf{P}^{(N+1)}$ is divided as follows:

$$\begin{aligned}\mathbf{P}^{(N+1)} &= \mathbf{A}^{(N)T}\mathbf{X}_{(N)}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right) = \left(\begin{bmatrix}\mathbf{A}^{(N)T}_{old} & \mathbf{A}^{(N)T}_{inc}\end{bmatrix}\begin{bmatrix}\mathbf{X}_{(N),old}\\ \mathbf{X}_{(N),new}\end{bmatrix}\right)\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right)\\ &= \mathbf{A}^{(N)T}_{old}\mathbf{X}_{(N),old}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right) + \mathbf{A}^{(N)T}_{inc}\mathbf{X}_{(N),new}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right)\\ &= \mathbf{P}^{(N+1)}_{old} + \mathbf{P}^{(N+1)}_{new}\end{aligned} \tag{25}$$

Similar to updating the factor matrices of the non-temporal modes, we only compute $\mathbf{P}^{(N+1)}_{new}$ for updating the core tensor since $\mathbf{P}^{(N+1)}_{old}$ is already computed at the previous step.

Next, we divide $\mathbf{Q}^{(N+1)}$ into $\mathbf{Q}^{(N+1)}_{old}$ and $\mathbf{Q}^{(N+1)}_{new}$.

$$\begin{aligned}\mathbf{Q}^{(N+1)} &= \mathbf{A}^{(N)T}\mathbf{A}^{(N)} = \begin{bmatrix}\mathbf{A}^{(N)T}_{old} & \mathbf{A}^{(N)T}_{inc}\end{bmatrix}\begin{bmatrix}\mathbf{A}^{(N)}_{old}\\ \mathbf{A}^{(N)}_{inc}\end{bmatrix} = \left(\mathbf{A}^{(N)T}_{old}\mathbf{A}^{(N)}_{old}\right) + \left(\mathbf{A}^{(N)T}_{inc}\mathbf{A}^{(N)}_{inc}\right)\\ &= \mathbf{Q}^{(N+1)}_{old} + \mathbf{Q}^{(N+1)}_{new}\end{aligned} \tag{26}$$

Then, we compute $\mathbf{Q}^{(N+1)}_{new}$, and obtain $\mathbf{Q}^{(N+1)}$; note that $\mathbf{Q}^{(N+1)}_{old}$ is already computed at the previous step.

4.3 Applying Approximation Phase

The objective of applying the approximation phase is to accelerate the update process for each incoming time slice. The main ideas are to 1) approximate a time slice by performing randomized SVD of each sliced matrix of a time slice and 2) update factor matrices and a core tensor with the SVD results of the time slice instead of the time slice $\mathcal{X}_{new}$. We accelerate computing Equation (14) for the temporal mode, the computation of $\mathbf{P}^{(n)}$ for non-temporal modes, and $\mathbf{P}^{(N+1)}$ for the core tensor.

Temporal Mode. To obtain the factor matrix $\mathbf{A}^{(N)}_{inc}$ of the temporal mode, we first apply the approximation phase for a new incoming tensor $\mathcal{X}_{new}$ and then efficiently compute Equation (14). With the temporal mode fixed to the last mode, we assume that dimensionalities of an incoming tensor $\mathcal{X}_{new} \in \mathbb{R}^{I_1\times I_2\times\cdots\times T}$ are sorted in descending order.

To apply the approximation phase to Equation (14), we start from re-expressing the term $\mathbf{X}_{(N),new}$ $(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger})$ in tensor form. Referring to Equation (5), we can rewrite the term as follows:

$$\mathcal{X}_{new}\times_1\left(\mathbf{A}^{(1)}\right)^{\dagger}\cdots\times_{N-1}\left(\mathbf{A}^{(N-1)}\right)^{\dagger}$$

Since the above equation has the same form as line 4 of Algorithm 2 for the N-th mode, computing it is the same as computing the N-th factor matrix in the iteration phase of D-Tucker. D-TuckerO first performs randomized SVD of each sliced matrix of a time slice $\mathcal{X}_{new}$ where the size of a sliced matrix is $I_1 \times I_2$. Then, we compute the term $\mathcal{X}_{new}\times_1\left(\mathbf{A}^{(1)}\right)^{\dagger}\times_2\left(\mathbf{A}^{(2)}\right)^{\dagger}$. The mode-1 matricization of the term is given by the following equation.

$$\mathbf{Y}_{inter} = \left(\left(\mathbf{A}^{(1)}\right)^{\dagger}\left[\mathbf{U}_1;\cdots;\mathbf{U}_L\right]\right)\times blkdiag\left(\{\Sigma_l\mathbf{V}_l^{\mathrm{T}}\left(\mathbf{A}^{(2)T}\right)^{\dagger}\}_{l=1}^{L}\right) \tag{27}$$

We compute $\left(\mathbf{A}^{(1)}\right)^{\dagger}\left[\mathbf{U}_1;\cdots;\mathbf{U}_L\right]$ and $blkdiag(\{\Sigma_l\mathbf{V}_l^{\mathrm{T}}\left(\mathbf{A}^{(2)T}\right)^{\dagger}\}_{l=1}^{L})$, respectively; then, $\mathbf{Y}_{inter}$ is obtained by multiplying the two results. After that, we perform n-mode products between $\mathcal{Y}_{inter}$ and $\left(\mathbf{A}^{(n)}\right)^{\dagger}$ for $n = 3, 4, ..., N-1$. Lastly, $\mathbf{A}^{(N)}_{inc}$ is updated by multiplying the mode-N matricization of the result of the n-mode products and $\mathbf{G}^{\dagger}_{(N)}$.

Non-temporal Modes. For a mode n except for N, the goal is to efficiently compute $\mathbf{P}_{new}^{(n)} = (\mathbf{X}_{(n),new}\ (\mathbf{A}_{inc}^{(N)} \otimes (\otimes_{k\neq n}^{N-1}\mathbf{A}^{(k)}))\mathbf{G}_{(n)}^{T})$ for updating the n-th factor matrix. Due to the expensive computations with $\mathcal{X}_{new}$ as in Tucker-ALS, we apply the approximation phase to reduce the computational cost of computing $\mathbf{P}_{new}^{(n)}$. We perform randomized SVD of each sliced matrix of a new incoming time slice $\mathcal{X}_{new}$ where the size of a sliced matrix is $I_1 \times I_2$, and then compute $\mathbf{P}_{new}^{(n)}$ using the SVD results.

To obtain $\mathbf{P}_{new}^{(n)}$, we compute $(\mathbf{X}_{(n),new}\ (\mathbf{A}_{inc}^{(N)} \otimes (\otimes_{k\neq n}^{N-1}\mathbf{A}^{(k)}))$, and then multiply it with $\mathbf{G}_{(n)}^{T}$. Before applying the approximation phase, we re-express $(\mathbf{X}_{(n),new}\ (\mathbf{A}_{inc}^{(N)} \otimes (\otimes_{k\neq n}^{N-1}\mathbf{A}^{(k)}))$ in tensor form as follows:

$$\mathcal{X}_{new} \times_1 \mathbf{A}^{(1)T} \cdots \times_{n-1} \mathbf{A}^{(n-1)T} \times_{n+1} \mathbf{A}^{(n+1)T} \cdots \times_N \mathbf{A}_{inc}^{(N)T} \tag{28}$$

Since the above equation has the same form as line 4 of Algorithm 2 for the n-th mode, computing it is the same as computing the n-th factor matrix in the iteration phase of D-Tucker. By using the SVD results of each sliced matrix of the time slice $\mathcal{X}_{new}$, we compute Equation (28) in the same way as computing an n-th factor matrix in the iteration phase of D-Tucker. Then, we obtain $\mathbf{P}_{new}^{(n)}$ by performing matrix multiplication between the mode-n matricized version of the result of Equation (28) and $\mathbf{G}_{(n)}^{T}$. After that, we update the n-th factor matrix using $\mathbf{P}_{new}^{(n)}$.

Core tensor. To efficiently update the core tensor $\mathcal{G}$, we focus on accelerating the computation for the matrix $\mathbf{P}_{new}^{(N+1)}$ since the matrices $\mathbf{P}_{old}^{(N+1)}$ and $\mathbf{Q}_{old}^{(N+1)}$ are already computed and the computational cost of $\mathbf{Q}_{new}^{(N+1)}$ is relatively low. For $\mathbf{P}_{new}^{(N+1)} = \mathbf{A}_{inc}^{(N)T}\mathbf{X}_{(N),new}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)T}\right)^{\dagger}\right)$, directly using $\mathbf{X}_{(N),new}$ is inefficient, so we apply the approximation phase. We first re-express $\mathbf{P}_{new}^{(N+1)}$ in tensor form:

$$\mathcal{X}_{new} \times_1 \mathbf{A}^{(1)\dagger} \cdots \times_{N-1} \mathbf{A}^{(N-1)\dagger} \times_N \mathbf{A}_{inc}^{(N)T} \tag{29}$$

$\mathbf{P}_{new}^{(N+1)}$ is obtained by computing the above equation in the following order: computing 1) Equation (27) for $\mathcal{X}_{new} \times_1 \left(\mathbf{A}^{(1)}\right)^{\dagger} \times_2 \left(\mathbf{A}^{(2)}\right)^{\dagger}$, 2) n-mode products with $\mathbf{A}^{(n)\dagger}$ for $n = 3, ..., N-1$, and 3) n-mode product with $\mathbf{A}_{inc}^{(N)T}$. Note that we use the SVD results of $\mathcal{X}_{new}$ in Equation (27), thereby we reduce the computational cost to update the core tensor compared to using $\mathcal{X}_{new}$. After that, we compute Equation (24) using $\mathbf{P}_{new}^{(N+1)}$.

4.4 Theoretical Analysis

Theorem 3. *Given a time slice $\mathcal{X}_{new}$ of size $I^2 \times K^{N-3} \times T_{new}$, the total time complexity of D-TuckerO to update factor matrices and core tensor is $O(I^2K^{N-3}T_{new} + NIK^{N-3}T_{new}J^2)$ where N is the order of a given tensor, I is the largest dimensionality, K is the remaining dimensionality, and J is the rank.*

Proof. See the proof in Appendix B.3. □

Theorem 4. *D-TuckerO requires $O(IK^{N-3}T_{new}J)$ space for updating factor matrices when a new incoming tensor $\mathcal{X}$ of the size $I_1 \times I_2 \times K^{N-3} \times T_{new}$ is given.*

Proof. See the proof in Appendix B.4. □

5 EXPERIMENT

In this section, we experimentally evaluate the performance of D-Tucker and D-TuckerO. We answer the following questions:

Table 3. Description of real-world tensor datasets for evaluating the performance of static Tucker decomposition methods.

Dataset	Order	Dimensionality	Rank
Brainq[1] [36]	3	(360, 21764, 9)	(10, 10, 5)
Boats[2] [58]	3	(320, 240, 7000)	(10, 10, 10)
Air Quality[3]	3	(30648, 376, 6)	(10, 10, 5)
HSI[4] [17]	4	(1021, 1340, 33, 8)	(10, 10, 10, 5)

Table 4. Description of real-world tensor datasets for evaluating the performance of streaming Tucker decomposition methods.

Dataset	Order	Dimensionality	Rank
Stock	3	(3028, 4, 200)	(10, 4, 10)
FMA[5] [15, 16]	3	(7994, 1025, 200)	(10, 10, 10)
Traffic[6] [45]	3	(1084, 96, 200)	(10, 10, 10)
Absorb[7]	4	(192, 288, 30, 200)	(10, 10, 10, 10)

- **Q1. Time cost and reconstruction error (Section 5.2).** How quickly does D-Tucker obtain factor matrices and core tensor compared to other competitors, while having low reconstruction error?
- **Q2. Effectiveness of the initialization phase (Section 5.3).** How much does the initialization phase reduce the number of iterations in D-Tucker?
- **Q3. Efficiency of the iteration phase (Section 5.4).** How efficient is the iteration phase of D-Tucker compared to other methods?
- **Q4. Space cost (Section 5.5).** How much space does D-Tucker require to obtain factor matrices and core tensor compared to other competitors?
- **Q5. Scalability (Section 5.6).** How well does D-Tucker scale up with regard to dimensionality, rank, order, and a number of iterations?
- **Q6. Running time and error in online streaming setting (Section 5.7).** For each new incoming tensor, how efficiently does D-TuckerO update factor matrices and core tensor?
- **Q7. Size of a time slice in an online streaming setting (Section 5.8).** How efficiently does D-TuckerO handle an incoming tensor slice of various sizes?

5.1 Experimental Settings

We describe experimental settings for the datasets, competitors, and environments.

Machine. We use a workstation with a single CPU (Intel Xeon E5-2630 v4 @ 2.2GHz), and 512GB memory.

Dataset. For static Tucker decomposition, we use four real-world tensors in Table 3 for evaluating the performance. Brainq dataset[1] [36] contains fMRI information consisting of (word, voxel, person; measurement). Boats dataset[2] [58] contains gray scale videos in the form of (height, width, frame; value). Air quality dataset[3] contains air pollutant information in Korea, in the form of (timestamp in second, location, atmospheric pollutants; measurement). HSI dataset[4] [17] contains hyperspectral

[1] http://www.cs.cmu.edu/afs/cs/project/theo-73/www/science2008/data.html
[2] http://changedetection.net/
[3] https://www.airkorea.or.kr
[4] https://personalpages.manchester.ac.uk/staff/d.h.foster/Hyperspectral_images_of_natural_scenes_04.html

images of natural scenes in the form of (spatial dimension (x), spatial dimension (y), spectral dimension, scene index; value).

For online streaming decomposition, we use four real-world tensors described in Table 4. Stock dataset contains features of stocks over 200 days in South Korea. The features consist of (adjusted opening price / previous day's adjusted closing price), (adjusted highest price / previous day's adjusted closing price), (adjusted lowest price / previous day's adjusted closing price), and (adjusted closing price / previous day's adjusted closing price). FMA dataset[5] [15, 16] is a song dataset whose form is (song, frequency, time; value). Each song is represented as an image of a log-power spectrogram. Traffic dataset[6] [45] contains traffic volume measurements from 1,084 sensors over 200 days, and each sensor yields 96 observations per day. Absorb dataset[7] is a 4-order tensor containing aerosol absorption; the form is (longitudes, latitudes, altitude, time; measurement). Note that the original values in this data are so small that we use a tensor multiplied by 10.

Competitors. We compare D-Tucker with static Tucker decomposition methods based on ALS approach. All the methods including D-Tucker are implemented in MATLAB (R2019b).

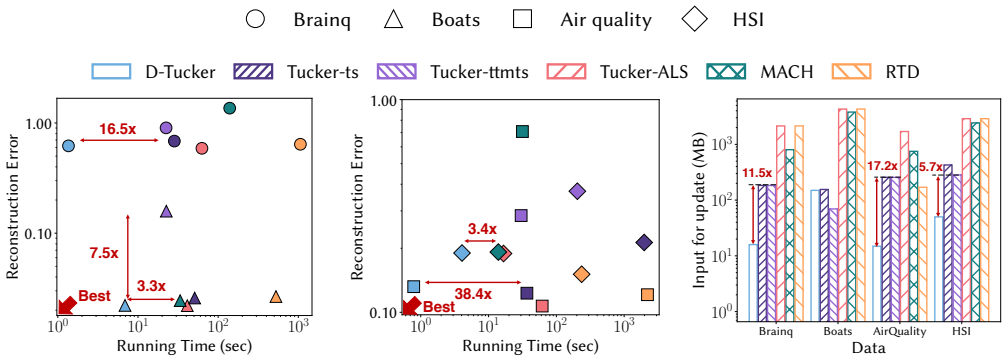
- **D-Tucker** [24]: we use randomized SVD [14] in the approximation phase using the implementation of Malik and Becker [34], standard SVD (*svds()* function in MATLAB) in the initialization and iteration phases, and Tensor Toolbox [6] for tensor operations such as n-mode product and matricization.
- **Tucker-ALS**: Tucker decomposition method based on ALS. We use the implementation in Tensor Toolbox [6].
- **MACH** [56]: Tucker decomposition method which samples entries of an input tensor and runs Tucker-ALS for the sampled tensor. We run Tucker-ALS in Tensor Toolbox [6] after sampling elements of a tensor.
- **Randomized Tucker Decomposition (RTD)** [11]: Tucker decomposition using a randomized algorithm. We use the Matlab code provided by authors.
- **Tucker-ts**, **Tucker-ttmts** [34]: Tucker-ts is a Tucker decomposition method using tensor sketch designed to approximate the solution of a large least-squares problem. Tucker-ttmts is a variant of Tucker-ts for better efficiency. We use the Matlab code[8] provided by authors.

We also compare D-TuckerO with the following streaming Tucker decomposition methods in an online streaming setting:

- **D-TuckerO**: We leverage Tensor Toolbox [6] for tensor operations such as n-mode product and matricization.
- **Tucker-ALS**: Tucker decomposition method based on ALS. We use the implementation in Tensor Toolbox [6].
- **Tucker-ts**, **Tucker-ttmts** [34]: Tucker-ts and Tucker-ttmts are easily adapted to online streaming settings.
- **DTA** (Dynamic Tensor Analysis): DTA finds factor matrices and core tensor to fit to newly arrived tensors. We use the Matlab code[9] provided by authors.
- **STA** (Streaming Tensor Analysis): STA is an approximation version of DTA that finds factor matrices and core tensor to fit to newly arrived tensors. We use the Matlab code[9] provided by authors.

Parameters. We use the following parameters.

[5] https://github.com/mdeff/fma
[6] https://github.com/florinsch/BigTrafficData
[7] https://www.earthsystemgrid.org/
[8] https://github.com/OsmanMalik/tucker-tensorsketch
[9] http://www.cs.cmu.edu/~jimeng/code/tensorCode.zip



(a) Running time vs. error for Brainq and Boats datasets

(b) Running time vs. error for Air quality and HSI datasets

(c) Memory Usage

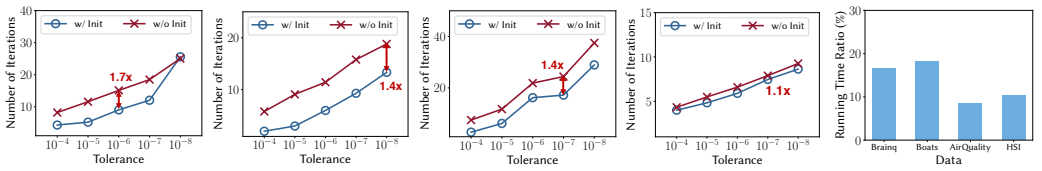
Fig. 3. D-Tucker achieves the best performance in terms of error, running time, and memory usage. (a) (b) Comparison for the tradeoff between running time and error; D-Tucker is up to 38.4× faster than the second-fastest competitor while having a similar error. (c) Space cost of D-Tucker. D-Tucker initializes and updates factor matrices and core tensor by using up to 17.2× smaller space than competitors except for Boats dataset. Note that, for Boats dataset, D-Tucker requires 2× higher space than Tucker-ttmts which has 7.5× higher error than our method.

- **Number of threads:** we use a single thread.
- **Max number of iterations:** we set the maximum number of iterations to 50.
- **Rank:** the dimensionality J_n of the nth mode of a core tensor is set to 10. We set it to 4 and 5, respectively, when the dimensionality is smaller than 5 and 10, respectively. We also set the rank J of randomized SVD to 10 which is the same as the dimensionality J_n of core tensor.
- **Tolerance:** the iteration stops when the variation of the error $\frac{\sqrt{\|\mathcal{X}\|_F^2 - \|\mathcal{G}\|_F^2}}{\|\mathcal{X}\|_F}$ [28] is less than $\epsilon = 10^{-4}$ except in Section 5.3 where we vary it.

We set other parameters of competitors based on their original papers. To compare running time, we run each method 10 times for D-Tucker and D-TuckerO, and report the average.

Reconstruction error. In a static setting, we evaluate the accuracy in terms of reconstruction error defined as $\frac{\|\mathcal{X} - \hat{\mathcal{X}}\|_F^2}{\|\mathcal{X}\|_F^2}$ where $\mathcal{X}$ is an input tensor and $\hat{\mathcal{X}}$ is the reconstruction of the output of Tucker decomposition.

In an online streaming setting, we measure the two kinds of errors, global and local reconstruction errors. The global reconstruction error is defined as $\sqrt{\frac{\sum_{i=1}^{T} \|\mathcal{X}_i - \hat{\mathcal{X}}_i\|_F^2}{\sum_{i=1}^{T} \|\mathcal{X}_i\|_F^2}}$ where $\mathcal{X}_i$ is a tensor obtained at time i and $\hat{\mathcal{X}}_i$ is a reconstructed tensor from factor matrices and core tensor of D-TuckerO. The global error indicates how well the results of a Tucker decomposition method represent an accumulated tensor over time. The local reconstruction error is defined as $\sqrt{\frac{\|\mathcal{X}_{new} - \hat{\mathcal{X}}_{new}\|_F}{\|\mathcal{X}_{new}\|_F}}$. In contrast to the global error, the local error indicates how well the results of a Tucker decomposition method represent a new incoming tensor.



(a) Brainq (b) Boats (c) Air quality (d) HSI (e) Running time ratio of the initialization phase

Fig. 4. The initialization phase of D-Tucker helps reduce the number of iterations and thus the total running time. (a-d) The number of iterations with the initialization phase is up to 1.7×, 1.4×, 1.4×, and 1.1× smaller than those without the initialization phase for Brainq, Boats, Air quality, and HSI datasets, respectively. (e) The average ratio of the running time in the initialization phase compared to the total running time does not exceed 20% for all the datasets.

5.2 Time Cost and Reconstruction Error (Q1)

We measure the running time and the reconstruction error of D-Tucker and competitors. As shown in Fig. 3(a) and 3(b), D-Tucker achieves the best trade-offs between the time and error, achieving up to 38.4× faster running time than Tucker-ts, Tucker-ttmts, and MACH with smaller or similar reconstruction errors. Tucker-ALS and RTD have smaller reconstruction errors for Air quality and HSI datasets, but they are at least 3.4× and 42× slower than D-Tucker, respectively.

5.3 Effectiveness of the Initialization Phase (Q2)

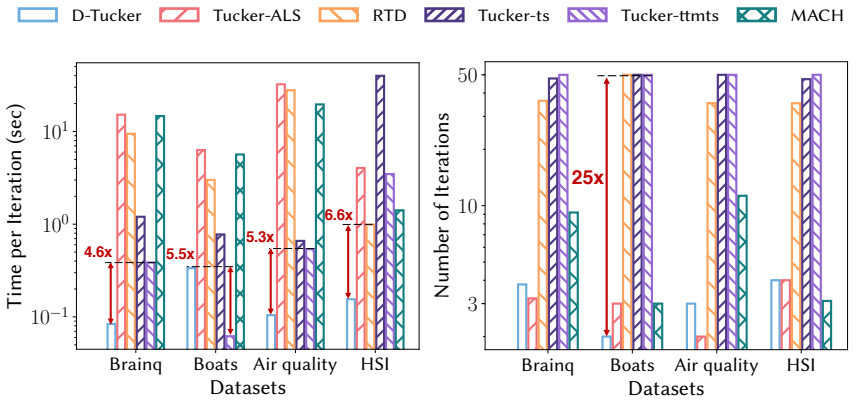
We show that the initialization phase of D-Tucker provides a good starting point for the iteration step, by measuring the number of iterations in the iteration phase. We vary the error tolerance ϵ in the iteration phase from 10^{-4} to 10^{-8}. As shown in Fig. 4, the number of iterations with the initialization phase is up to 1.7× smaller than that without the initialization phase. The initialization phase allows D-Tucker to reduce the total running time since the running time of the initialization phase is less than the reduction time of the iteration phase. Moreover, the average ratio of the initialization phase's running time to the total running time in D-Tucker does not exceed 20%. This indicates that the initialization phase of D-Tucker reduces the number of iterations significantly with little additional overhead on the total running time.

5.4 Efficiency of the Iteration Phase (Q3)

We investigate the number of iterations and the running time per iterations. In Fig. 5, For each iteration, D-Tucker is at least 4.6× faster than competitors on all datasets except for Boat dataset, and consumes smaller number of iterations than the competitors. Although Tucker-ttmts is faster than D-Tucker at each iteration, it requires larger number of iterations than D-Tucker; hence, the total running time of D-Tucker is 4.5× longer than that of Tucker-ttmts at iteration phase. For the number of iterations, Fig. 5(b) shows that D-Tucker requires smaller number of iteration than all the competitors except for Tucker-ALS on 3-order datasaets; however, the difference is quite small considering the running time per iteration.

5.5 Space Cost (Q4)

We investigate the memory requirements of D-Tucker and competitors for initializing and updating factor matrices and core tensor. Fig. 3(c) shows that D-Tucker requires up to 17.2× smaller space than the second best methods Tucker-ts and Tucker-ttmts in terms of memory usage. For Boats



(a) Time per iteration (b) Number of iterations

Fig. 5. In the iteration phase, D-Tucker is the most efficient compared to competitors. (a) The running time of each iteration of D-Tucker is up to 6.6× faster than those of competitors except for the Boats dataset. For the Boats dataset, Tucker-ttmts achieves the fastest running time per iteration, but requires much larger number of iterations, and has much higher error than D-Tucker. (b) The number of iterations of D-Tucker is in general smaller than others; while there are cases D-Tucker requires more number of iterations, the difference is negligible considering the running time per iteration.

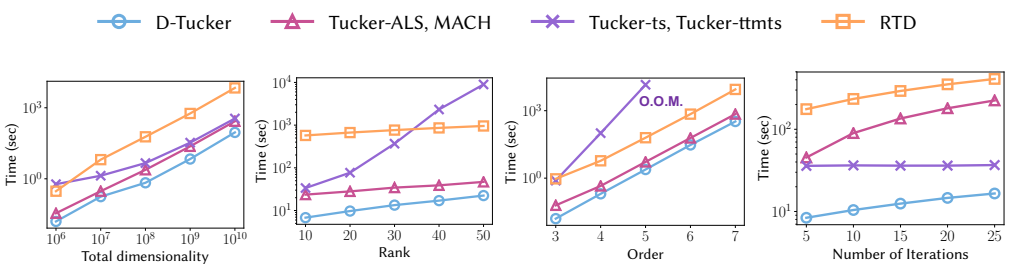
dataset, Tucker-ts, and Tucker-ttmts require small space since this dataset has the following setting where the two methods operate well: 1) order N and rank J are very small, and 2) dimensionalities I and K are very large. Note that D-Tucker has 7.5× less error than Tucker-ttmts while requiring 2.1× more space than Tucker-ttmts.

5.6 Scalability (Q5)

We investigate the scalability of D-Tucker and competitors with regard to dimensionality, target rank, order, and number of iterations in Fig. 6. In sum, D-Tucker is the most scalable with the smallest running time. Since the time complexities of Tucker-ts and Tucker-ttmts are proportional to J^N, they are not scalable for the target rank, and order of an input tensor. RTD operates for all the given experimental settings, but RTD is much slower than D-Tucker. MACH and Tucker-ALS also operate for all the given experimental settings, but they are at least 2× slower than D-Tucker. Furthermore, they become much slower than D-Tucker as the number of iteration increases (e.g., when setting smaller tolerance ϵ or when converging slowly in real-world datasets). The details of scalability experiments are as follows.

Dimensionality. For investigating the scalability related to dimensionality, we generate synthetic 3-order tensors of true rank $J_{true} = 10$, while increasing the total dimensionality $I_1 I_2 K_3$ from 10^6 to 10^{10} (dimensionality list: $\{(10^2, 10^2, 10^2), (10^3, 10^2, 10^2), (10^3, 10^3, 10^2), (10^3, 10^3, 10^3), (10^4, 10^3, 10^3)\}$). As shown in Fig. 6(a), D-Tucker is the fastest for various dimensionalities, and runs at least 2.7× faster than all competitors.

Target rank. For investigating the scalability related to target rank, we generate synthetic 3-order tensors of size $I_1 = I_2 = K_3 = 10^3$ and true rank $J_{true} = 10$, while varying the target rank from 10 to 50. As shown in Fig. 6(b), D-Tucker is the fastest for various target ranks. Tucker-ts and Tucker-ttmts provide the worst scalabilities since their time complexities are proportional to J^N.



(a) Total dimensionality (b) Rank (c) Order (d) Number of Iterations

Fig. 6. Scalability of D-Tucker compared to other Tucker decomposition methods. O.O.M.: out of memory. For clarity, we show 4 groups of methods having similar tendencies. Note that D-Tucker is the most scalable with the smallest running time: for all settings, D-Tucker is at least 2.1× faster than competitors. Tucker-ts and Tucker-ttmts have limited scalability with respect to the target rank and the order. RTD has good scalability for all aspects, but it is up to 76× slower than D-Tucker. MACH and Tucker-ALS are also scalable for all aspects, but they are at least 2× slower than D-Tucker. Furthermore, their performance gaps compared to D-Tucker become even worse when the number of iteration increases.

The running times of all competitors except Tucker-ts and Tucker-ttmts scale with regard to target ranks, but they are at least 2.1× slower than D-Tucker.

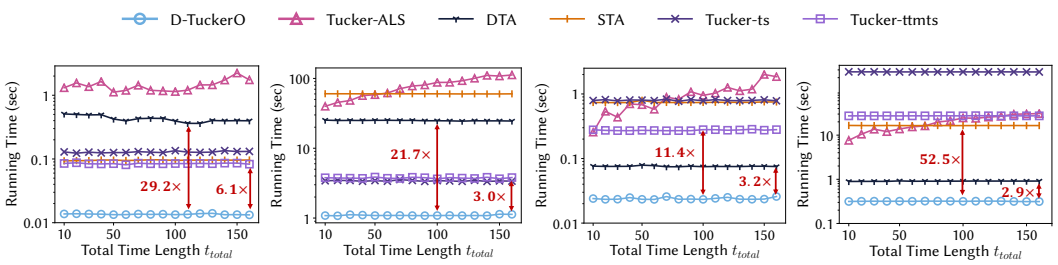
Order. For investigating the scalability related to order N, we generate synthetic N-order tensors of true rank $J_{true} = 10$, while varying the order from 3 to 7. We set dimensionalities of synthetic tensors to $I_1 = 10^3$, $I_2 = 10^2$, and $K_i = 10$ for $i = 3, 4, ..., 7$. In Fig. 6(c), D-Tucker is the fastest for various orders of input tensors. Since the time and memory complexities of Tucker-ts and Tucker-ttmts are proportional to J^{2N}, they are 5883× slower than D-Tucker, and cannot deal with 6 and 7-order tensors. Although all competitors except Tucker-ts and Tucker-ttmts can process higher order tensors, they are at least 2.1× slower than D-Tucker.

Number of iterations. We generate synthetic 3-order tensors of size $I_1 = I_2 = K_3 = 10^3$ with true rank $J_{true} = 10$. Then we evaluate the running time varying the number of iterations from 5 to 25. As shown in Fig. 6(d), D-Tucker is the fastest for varying numbers of iterations. In addition, the running time of D-Tucker is not affected much by the number of iterations while those of all competitors except Tucker-ts and Tucker-ttmts are affected much by the number of iterations. Note that the running time of Tucker-ts and Tucker-ttmts are 3.6× slower than that of D-Tucker although those are less affected by the number of iterations than D-Tucker.

5.7 Streaming Setting (Q6)

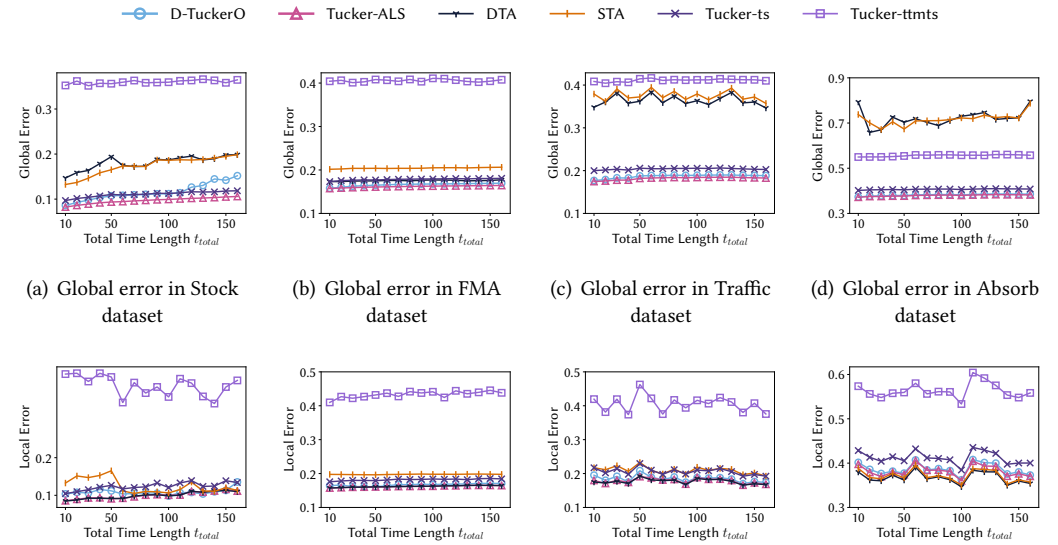
We compare D-TuckerO with streaming Tucker decomposition methods. We initially construct factor matrices and a core tensor using the first 20% of a whole tensor, and then measure the running time of updating a new incoming tensor at each time point. In addition, we set t_{new} of each time slice to 10.

Running Time. As shown in Fig. 7, we compare the running time of D-TuckerO with those of competitors. For the 3-order datasets, D-TuckerO is up to 6.1× faster than the second-fastest competitor Tucker-ttmts as shown in Fig. 7(a) to 7(c). Also, D-TuckerO is at least 2.9× faster than the competitors for Absorb dataset which is a 4-order tensor. In addition, the running time of D-TuckerO does not increase over time since it is proportional to the size of a new incoming tensor, not the accumulated tensor.



(a) Update time in Stock dataset (b) Update time in FMA dataset (c) Update time in Traffic dataset (d) Update time in Absorb dataset

Fig. 7. Running time of D-TuckerO and competitors over time. D-TuckerO outperforms competitors when we compare the running time of updating factor matrices and core tensor for each new incoming tensor. D-TuckerO is up to 6.1× faster than the second fastest method, and the running time does not increase over time.

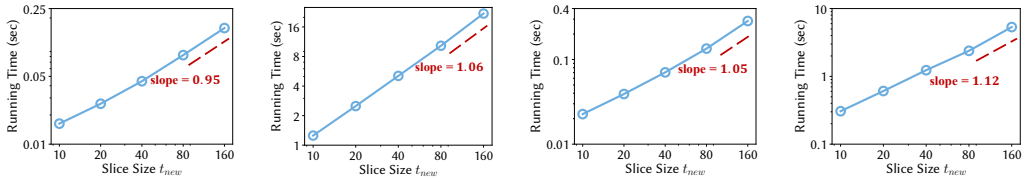


(a) Global error in Stock dataset (b) Global error in FMA dataset (c) Global error in Traffic dataset (d) Global error in Absorb dataset

(e) Local error in Stock dataset (f) Local error in FMA dataset (g) Local error in Traffic dataset (h) Local error in Absorb dataset

Fig. 8. Global and local errors in an online streaming setting. D-TuckerO achieves comparable global and local errors with Tucker-ALS which is a static version of Tucker decomposition.

Error. We measure global and local reconstruction errors of D-TuckerO and competitors. Fig. 8(a) to 8(d) show the results for global reconstruction errors, and Fig. 8(e) to 8(h) show the results for local reconstruction errors. As shown in Fig. 8(a) to 8(d), D-TuckerO has comparable global errors with Tucker-ALS which performs Tucker decomposition for accumulated tensors, while DTA and STA have higher global errors than D-TuckerO. These results indicate that updated results of D-TuckerO sufficiently contain global patterns of an accumulated tensor. As shown in Fig. 8(e) to 8(h), the local errors of D-TuckerO are close to those of Tucker-ALS which is a static version of



(a) Stock dataset (b) FMA dataset (c) Traffic dataset (d) Absorb dataset

Fig. 9. We measure the running time of D-TuckerO, varying the size of a time slice. The running time of D-TuckerO increases near-linearly as the size of a time slice increases. Note that a slope equal to 1 indicates linear scalability.

Tucker decomposition since updated results of D-TuckerO sufficiently contains information of a new incoming tensor. In addition, the approximation phase of D-TuckerO does not hurt accuracy much since a time slice of real-world datasets has a low-rank structure.

5.8 Size of Time Slice (Q7)

We evaluate the performance of D-TuckerO, varying the size t_{new} of a time slice: 10, 20, 40, 80, and 160. Fig. 9 shows that there are near-linear relationships between t_{new} and the running time of D-TuckerO in an online streaming setting; for all the four datasets, the slopes are close to 1. This is because the running time of D-TuckerO is proportional to the size of a new incoming tensor.

6 RELATED WORK

We describe related works for Tucker decomposition methods and their applications.

Tensor decomposition. De Lathauwer et al. [31] proposed Tucker-ALS (Algorithm 2) which alternately updates factor matrices and obtains core tensor. A few Tucker decomposition methods slightly reduce the computational time using efficient matrix operations [5, 33]. Che et al. [11] applied randomized algorithms for Tucker decomposition. The main challenges of Tucker decomposition are heavy computational time and large memory requirements due to large-scale dense tensors. To overcome the challenges, MACH [56] is designed to reduce the computational time and the memory requirement by sampling input tensors. Also, Malik et al. [34] used a sketch of input tensors to overcome the challenges. However, there is still plenty of room for improvement in terms of efficiency. Several Tucker decomposition algorithms [4, 10, 13, 26, 38, 46, 60] have been developed in parallel and distributed systems as well. Several works [4, 8, 10, 13] optimize n-mode product for dense tensors in distributed systems. Other works present Tucker decomposition methods exploiting the characteristic of sparse tensors in parallel systems [38, 39, 49] and distributed systems [26, 38]. Contrary to the above methods, D-Tucker efficiently runs on a single machine.

Streaming Tensor Decomposition. Many works [3, 18, 37, 48, 51, 62, 63] have developed CP decomposition methods in an online streaming setting. RLST (Recursive Least Squares Tracking) and SDT (Simultaneous Diagonalization Tracking) [37] are adaptive PARAFAC decomposition methods of a third-order tensor in an online streaming setting. Zhou et al. [63] developed onlineCP, a streaming CP decomposition method, while Zhou et al. [62] extend onlineCP for sparse tensors. Gujral et al. [18] and Smith et al. [48] proposed streaming CP decomposition methods in parallel systems. Lee et al. [32] proposed a robust tensor factorization that leverages two temporal characteristics: graduality and seasonality. Ahn et al. [2, 3] proposed tensor factorization methods by capturing temporal locality patterns. Son et al. [50] proposed a n online tensor factorization

method by capturing sudden change in data. The main difference between the above methods and our proposed method is that they focus on developing online versions of CP decomposition while D-TuckerO is based on Tucker decomposition.

Sun et al. [52] incrementally analyzed temporal tensors over time: they proposed two algorithms, DTA (dynamic tensor analysis) and STA (streaming tensor analysis). However, the above methods update factor matrices and core tensor by naively using a new incoming tensor without compression, thereby efficiency improvement is limited when a new incoming tensor is sufficiently large. In addition, tucker-ts and tucker-ttmts [34] can be applied to online streaming settings. However, they fail to avoid increasing the running time over time. Sun et al. [53] proposed a streaming Tucker decomposition method with a sketching technique in distributed systems, assuming that time slices are stored in several machines. MAST [51] deals with the scenario in which a given tensor grows in multiple modes while D-TuckerO runs on the setting where only one mode increases.

Applications of Tensor decomposition. Tucker decomposition has been widely used for several applications including dimensionality reduction [27, 47], recommendation [12, 40, 44], clustering [9, 20], image tag refinement [54, 55], phenotype discovery [1, 43, 61], and many others [22, 29, 42]. Oh et al. [40] analyzed movie rating data and discovered relations between movie and time attributes by considering only observable entries. Kim et al. [27] used Tucker decomposition for compressing a deep convolutional neural network. Jang et al. [25] proposed a Tucker decomposition-based method to efficiently analyze a given time range.

7 CONCLUSIONS

We propose D-Tucker and D-TuckerO, efficient Tucker decomposition methods for large-scale dense tensors in static and online streaming settings. D-Tucker and D-TuckerO accelerate computing Tucker decomposition by approximating a given dense tensor, and carefully computing Tucker results from the approximated tensor. We show D-Tucker provides the fastest running time and the smallest memory usage. Furthermore, D-TuckerO is also the fastest method to update factor matrices and core tensor for new incoming tensors. We also provide theoretical analysis for the time and space complexities of D-Tucker and D-TuckerO. Extensive experiments show that D-Tucker is up to 38.4× faster, and requires up to 17.2× less space than existing methods with little sacrifice in accuracy. D-Tucker is also scalable with regard to dimensionality, rank, order, and the number of iterations. D-TuckerO is up to 6.1× faster than existing methods running in an online streaming setting, while not increasing the running time over time.

ACKNOWLEDGEMENT

This work was partly supported by the National Research Foundation of Korea(NRF) funded by MSIT(2022R1A2C3007921), and Institute of Information & communications Technology Planning & Evaluation(IITP) grant funded by MSIT [No.2021-0-01343, Artificial Intelligence Graduate School Program (Seoul National University)] and [No.2021-0-02068, Artificial Intelligence Innovation Hub (Artificial Intelligence Institute, Seoul National University)]. The Institute of Engineering Research and ICT at Seoul National University provided research facilities for this work.

REFERENCES

[1] Ardavan Afshar, Ioakeim Perros, Evangelos E Papalexakis, Elizabeth Searles, Joyce Ho, and Jimeng Sun. 2018. COPA: Constrained PARAFAC2 for sparse & large datasets. In *CIKM*. 793–802.

[2] Dawon Ahn, Jun-Gi Jang, and U Kang. 2022. Time-aware tensor decomposition for sparse tensors. *Mach. Learn.* 111, 4 (2022), 1409–1430.

[3] Dawon Ahn, Seyun Kim, and U Kang. 2021. Accurate Online Tensor Factorization for Temporal Tensor Streams with Missing Values. In *CIKM*. ACM, 2822–2826.

[4] Woody Austin, Grey Ballard, and Tamara G. Kolda. 2016. Parallel Tensor Compression for Large-Scale Scientific Data. In *IPDPS*. 912–922.

[5] Brett W. Bader and Tamara G. Kolda. 2006. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Trans. Math. Software* 32, 4 (Dec. 2006), 635–653. https://doi.org/10.1145/1186785.1186794

[6] Brett W. Bader, Tamara G. Kolda, et al. 2017. MATLAB Tensor Toolbox Version 3.0-dev. Available online. https://www.tensortoolbox.org

[7] James Baglama and Lothar Reichel. 2005. Augmented Implicitly Restarted Lanczos Bidiagonalization Methods. *SIAM J. Scientific Computing* 27, 1 (2005), 19–42.

[8] Grey Ballard, Alicia Klinvex, and Tamara G. Kolda. 2019. TuckerMPI: A Parallel C++/MPI Software Package for Large-scale Data Compression via the Tucker Tensor Decomposition. *CoRR* abs/1901.06043 (2019).

[9] Xiaochun Cao, Xingxing Wei, Yahong Han, and Dongdai Lin. 2015. Robust Face Clustering Via Tensor Decomposition. *IEEE Trans. Cybernetics* 45, 11 (2015), 2546–2557.

[10] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Xing Liu, Prakash Murali, Yogish Sabharwal, and Dheeraj Sreedhar. 2017. On Optimizing Distributed Tucker Decomposition for Dense Tensors. In *IPDPS*. 1038–1047.

[11] Maolin Che and Yimin Wei. 2019. Randomized algorithms for the approximations of Tucker and the tensor train decompositions. *Adv. Comput. Math.* 45, 1 (2019), 395–428.

[12] Dongjin Choi, Jun-Gi Jang, and U Kang. 2019. S3CMTF: Fast, accurate, and scalable method for incomplete coupled matrix-tensor factorization. *PloS one* 14, 6 (2019), e0217316.

[13] Jee W. Choi, Xing Liu, and Venkatesan T. Chakaravarthy. [n. d.]. High-performance dense tucker decomposition on GPU clusters. In *SC*. 42:1–42:11.

[14] Kenneth L Clarkson and David P Woodruff. 2017. Low-rank approximation and regression in input sparsity time. *JACM* 63, 6 (2017), 54.

[15] Michaël Defferrard, Kirell Benzi, Pierre Vandergheynst, and Xavier Bresson. 2017. FMA: A Dataset for Music Analysis. In *18th International Society for Music Information Retrieval Conference (ISMIR)*. arXiv:1612.01840 https://arxiv.org/abs/1612.01840

[16] Michaël Defferrard, Sharada P. Mohanty, Sean F. Carroll, and Marcel Salathé. 2018. Learning to Recognize Musical Genre from Audio. In *The 2018 Web Conference Companion*. ACM Press. https://doi.org/10.1145/3184558.3192310 arXiv:1803.05337

[17] David H. Foster, Kinjiro Amano, Sérgio M C Nascimento, and Michael J. Foster. 2006. Frequency of metamerism in natural scenes. *Optical Society of America. Journal A: Optics, Image Science, and Vision* 23, 10 (10 2006), 2359–2372. https://doi.org/10.1364/JOSAA.23.002359

[18] Ekta Gujral, Ravdeep Pasricha, and Evangelos E. Papalexakis. 2018. SamBaTen: Sampling-based Batch Incremental Tensor Decomposition. In *SDM*. SIAM, 387–395.

[19] Nathan Halko, Per-Gunnar Martinsson, and Joel A. Tropp. 2011. Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions. *SIAM Rev.* 53, 2 (2011), 217–288.

[20] Heng Huang, Chris H. Q. Ding, Dijun Luo, and Tao Li. 2008. Simultaneous tensor subspace selection and clustering: the equivalence of high order svd and k-means clustering. In *SIGKDD*. 327–335.

[21] MA Iwen and BW Ong. 2016. A distributed and incremental SVD algorithm for agglomerative data analysis on large networks. *SIAM J. Matrix Anal. Appl.* 37, 4 (2016), 1699–1718.

[22] Jun-Gi Jang and U Kang. 2022. DPar2: Fast and Scalable PARAFAC2 Decomposition for Irregular Dense Tensors. In *ICDE*. IEEE, 2454–2467.

[23] Jun-Gi Jang, Dongjin Choi, Jinhong Jung, and U Kang. 2018. Zoom-SVD: Fast and Memory Efficient Method for Extracting Key Patterns in an Arbitrary Time Range. In *CIKM*. ACM, 1083–1092.

[24] Jun-Gi Jang and U Kang. 2020. D-tucker: Fast and memory-efficient tucker decomposition for dense tensors. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1850–1853.

[25] Jun-Gi Jang and U Kang. 2021. Fast and Memory-Efficient Tucker Decomposition for Answering Diverse Time Range Queries. In *KDD*. 725–735.

[26] Inah Jeon, Evangelos E. Papalexakis, U. Kang, and Christos Faloutsos. 2015. HaTen2: Billion-scale tensor decompositions. In *ICDE*. 1047–1058.

[27] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. 2015. Compression of Deep Convolutional Neural Networks for Fast and Low Power Mobile Applications. *CoRR* abs/1511.06530 (2015). arXiv:1511.06530 http://arxiv.org/abs/1511.06530

[28] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (2009), 455–500.

[29] Tamara G. Kolda and Jimeng Sun. 2008. Scalable Tensor Decompositions for Multi-aspect Data Mining. In *ICDM*. 363–372.

[30] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. A Multilinear Singular Value Decomposition. *SIAM J. Matrix Analysis Applications* 21, 4 (2000), 1253–1278.

[31] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. 2000. On the Best Rank-1 and Rank-(R_1 , R_2, ... , R_N) Approximation of Higher-Order Tensors. *SIAM J. Matrix Analysis Applications* 21, 4 (2000), 1324–1342.

[32] Dongjin Lee and Kijung Shin. 2021. Robust Factorization of Real-world Tensor Streams with Patterns, Missing Values, and Outliers. In *ICDE*. IEEE, 840–851.

[33] Jiajia Li, Casey Battaglino, Ioakeim Perros, Jimeng Sun, and Richard W. Vuduc. 2015. An input-adaptive and in-place approach to dense tensor-times-matrix multiply. In *SC*. 76:1–76:12.

[34] Osman Asif Malik and Stephen Becker. 2018. Low-Rank Tucker Decomposition of Large Tensors Using TensorSketch. In *NeurIPS*. 10117–10127.

[35] Rachel Minster, Arvind K. Saibaba, and Misha E. Kilmer. 2019. Randomized algorithms for low-rank tensor decompositions in the Tucker format. *CoRR* abs/1905.07311 (2019).

[36] Tom M. Mitchell, Svetlana V. Shinkareva, Andrew Carlson, Kai-Min Chang, Vicente L. Malave, Robert A. Mason, and Marcel Adam Just. 2008. Predicting human brain activity associated with the meanings of nouns. *Science* 320 (May 2008), 1191–1195.

[37] Dimitri Nion and Nicholas D. Sidiropoulos. 2009. Adaptive algorithms to track the PARAFAC decomposition of a third-order tensor. *IEEE Trans. Signal Process.* 57, 6 (2009), 2299–2310.

[38] Jinoh Oh, Kijung Shin, Evangelos E. Papalexakis, Christos Faloutsos, and Hwanjo Yu. 2017. S-HOT: Scalable High-Order Tucker Decomposition. In *WSDM*. 761–770.

[39] Sejoon Oh, Namyong Park, Jun-Gi Jang, Lee Sael, and U Kang. 2019. High-Performance Tucker Factorization on Heterogeneous Platforms. *IEEE Trans. Parallel Distrib. Syst.* 30, 10 (2019), 2237–2248.

[40] Sejoon Oh, Namyong Park, Lee Sael, and U. Kang. 2018. Scalable Tucker Factorization for Sparse Tensors - Algorithms and Discoveries. In *ICDE*. 1120–1131.

[41] Spiros Papadimitriou, Jimeng Sun, and Christos Faloutsos. 2005. Streaming Pattern Discovery in Multiple Time-Series. In *VLDB*. ACM, 697–708.

[42] Ioakeim Perros, Robert Chen, Richard W. Vuduc, and Jimeng Sun. 2015. Sparse Hierarchical Tucker Factorization and Its Application to Healthcare. In *ICDM*. 943–948.

[43] Ioakeim Perros, Evangelos E Papalexakis, Fei Wang, Richard Vuduc, Elizabeth Searles, Michael Thompson, and Jimeng Sun. 2017. SPARTan: Scalable PARAFAC2 for large & sparse data. In *KDD*. 375–384.

[44] Steffen Rendle and Lars Schmidt-Thieme. 2010. Pairwise interaction tensor factorization for personalized tag recommendation. In *WSDM*. 81–90.

[45] Florin Schimbinschi, Xuan Vinh Nguyen, James Bailey, Chris Leckie, Hai Vu, and Rao Kotagiri. 2015. Traffic forecasting in complex urban networks: Leveraging big data and machine learning. In *Big Data*. IEEE, 1019–1024.

[46] Kijung Shin, Lee Sael, and U. Kang. 2017. Fully Scalable Methods for Distributed Tensor Factorization. *IEEE Trans. Knowl. Data Eng.* 29, 1 (2017), 100–113.

[47] Nicholas D. Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E. Papalexakis, and Christos Faloutsos. 2017. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Trans. Signal Processing* 65, 13 (2017), 3551–3582.

[48] Shaden Smith, Kejun Huang, Nicholas D. Sidiropoulos, and George Karypis. 2018. Streaming Tensor Factorization for Infinite Data Sources. In *SDM*. SIAM, 81–89.

[49] Shaden Smith and George Karypis. 2017. Accelerating the Tucker Decomposition with Compressed Sparse Tensors. In *Euro-Par 2017 (Lecture Notes in Computer Science, Vol. 10417)*. Springer, 653–668.

[50] Sangjun Son, Yong-chan Park, Minyong Cho, and U. Kang. 2022. DAO-CP: Data-Adaptive Online CP decomposition for tensor stream. *PLOS ONE* 17, 4 (04 2022), 1–18.

[51] Qingquan Song, Xiao Huang, Hancheng Ge, James Caverlee, and Xia Hu. 2017. Multi-Aspect Streaming Tensor Completion. In *KDD*. ACM, 435–443.

[52] Jimeng Sun, Dacheng Tao, and Christos Faloutsos. 2006. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Philadelphia, PA, USA, August 20-23, 2006*, Tina Eliassi-Rad, Lyle H. Ungar, Mark Craven, and Dimitrios Gunopulos (Eds.). ACM, 374–383.

[53] Yiming Sun, Yang Guo, Charlene Luo, Joel A. Tropp, and Madeleine Udell. 2019. Low-Rank Tucker Approximation of a Tensor From Streaming Data. *CoRR* abs/1904.10951 (2019).

[54] Jinhui Tang, Xiangbo Shu, Zechao Li, Yu-Gang Jiang, and Qi Tian. 2019. Social Anchor-Unit Graph Regularized Tensor Completion for Large-Scale Image Retagging. *IEEE Trans. Pattern Anal. Mach. Intell.* 41, 8 (2019), 2027–2034.

[55] Jinhui Tang, Xiangbo Shu, Guo-Jun Qi, Zechao Li, Meng Wang, Shuicheng Yan, and Ramesh C. Jain. 2017. Tri-Clustered Tensor Completion for Social-Aware Image Tag Refinement. *IEEE Trans. Pattern Anal. Mach. Intell.* 39, 8 (2017), 1662–1674.

[56] Charalampos E. Tsourakakis. 2010. MACH: Fast Randomized Tensor Decompositions. In *SDM*. 689–700.

[57] Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. 2012. A New Truncation Strategy for the Higher-Order Singular Value Decomposition. *SIAM J. Scientific Computing* 34, 2 (2012).
[58] Yi Wang, Pierre-Marc Jodoin, Fatih Murat Porikli, Janusz Konrad, Yannick Benezeth, and Prakash Ishwar. 2014. CDnet 2014: An Expanded Change Detection Benchmark Dataset. In *IEEE Conference on Computer Vision and Pattern Recognition CVPR Workshops*. 393–400.
[59] Franco Woolfe, Edo Liberty, Vladimir Rokhlin, and Mark Tygert. 2008. A fast randomized algorithm for the approximation of matrices. *Applied and Computational Harmonic Analysis* 25, 3 (2008), 335–366.
[60] Fan Yang, Fanhua Shang, Yuzhen Huang, James Cheng, Jinfeng Li, Yunjian Zhao, and Ruihao Zhao. 2017. LFTF: A Framework for Efficient Tensor Analytics at Scale. *PVLDB* 10, 7 (2017), 745–756.
[61] Kejing Yin, William K Cheung, Benjamin CM Fung, and Jonathan Poon. 2021. Tedpar: Temporally dependent parafac2 factorization for phenotype-based disease progression modeling. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 594–602.
[62] Shuo Zhou, Sarah M. Erfani, and James Bailey. 2018. Online CP Decomposition for Sparse Tensors. In *ICDM*. IEEE Computer Society, 1458–1463.
[63] Shuo Zhou, Xuan Vinh Nguyen, James Bailey, Yunzhe Jia, and Ian Davidson. 2016. Accelerating Online CP Decompositions for Higher Order Tensors. In *KDD*. ACM, 1375–1384.

APPENDIX

A PROOFS OF LEMMAS

We provide proof for lemmas described in the proposed method sections (Sections 3 and 4).

A.1 Proof of Lemma 1

Proof. Performing randomized SVD of each sliced matrix takes $O(I^2)$ (Algorithm 1). Since the number of sliced matrices is K^{N-2}, the time complexity of the approximation phase is $O(I^2K^{N-2})$. □

A.2 Proof of Lemma 2

Proof. For the first mode, size of $\left[\mathbf{U}_1\Sigma_1; \cdots ; \mathbf{U}_l\Sigma_l\right]$ is $I \times K^{N-2}J$. Then, performing SVD [7] takes $O(IK^{N-2}J^2)$ for the first mode. For the second mode, it takes $O(IK^{N-2}J^2)$ to compute $\mathbf{Y}_{(2),inter}$ (line 3 of Algorithm 5), $O(IK^{N-2}J^2)$ to compute $\mathbf{Y}_{(2),inter} blkdiag(\{\Sigma_l \mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$, and $O(IK^{N-2}J^2)$ to perform SVD of $\mathbf{Y}_{(2)}$. Then, it takes $O(IK^{N-2}J^2)$ to initialize the factor matrix of the second mode. For the remaining modes, it takes $O(IK^{N-2}J^2 + \sum_{k=0}^{N-3} K^{N-2-k}J^{3+k})$ to compute the remaining n-mode products for all $n = 2, 3, ..., N$, and $O(J\sum_{k=0}^{N-3} K^{N-2-k}J^{2+k})$ to compute SVD for all $n = 3, 4, ..., N$. For all modes, the dominant term is $O(IK^{N-2}J^2)$ since $I > K > J$, thus we simplify the time complexity of the initialization phase as $O(IK^{N-2}J^2)$. □

A.3 Proof of Lemma 3

Proof. For the first mode, it takes $O(IK^{N-2}J^2)$ to compute $\mathbf{Y}_{(1),inter}$ and $\mathbf{Y}_{(1),inter} blkdiag(\{\Sigma_l \mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$ in Equation (13), and $O(I\sum_{k=0}^{N-3} K^{N-2-k}J^{2+k})$ for computing the remaining n-mode products for all $n = 3, 4, ..., N$. We simplify $O(I\sum_{k=0}^{N-3} K^{N-2-k}J^{2+k})$ as $O(NIK^{N-2}J^2)$ since $J < K$. Computational time of the second mode is the same as that of the first mode. Before computing for remaining modes, it takes $O(IK^{N-2}J^2 + K^{N-2}J^3)$ to compute $\mathbf{Y}_{reuse}$ in line 15 of Algorithm 6. For mode-i, it takes $O(-K^{N-(i-1)}J^i + \sum_{k=0}^{N-3} K^{N-2-k}J^{3+k})$ to perform n-mode products for all $n = 3, 4, i-1, i+1, ..., N$. For core tensor, it takes $O(\sum_{k=0}^{N-3} K^{N-2-k}J^{3+k})$ to perform n-mode products for all $n = 3, 4, ..., N$. We simplify the complexity $O(-K^{N-(i-1)}J^i + \sum_{k=0}^{N-3} K^{N-2-k}J^{3+k})$ and $O(\sum_{k=0}^{N-3} K^{N-2-k}J^{3+k})$ to $O(NK^{N-2}J^3)$ since $K > J$. Therefore, the time complexity of one iteration in the iteration phase is $O(IK^{N-2}J^2 + NIK^{N-2}J^2 + IK^{N-2}J^2 + K^{N-2}J^3 + NK^{N-2}J^3)$. Without loss of generality, we express the time complexity of the iteration phase as $O(NIK^{N-2}J^2)$ since $I > K > J$. □

A.4 Proof of Lemma 4

Proof. The following equation represents the mode-N matricized version of Equation (4) by replacing $\mathcal{X}$ with $\mathcal{X}_{old}$ and $\mathcal{X}_{new}$:

$$\mathbf{X}_{(N)} = \begin{bmatrix} \mathbf{X}_{(N),old} \\ \mathbf{X}_{(N),new} \end{bmatrix} \approx \begin{bmatrix} \mathbf{A}^{(N)}_{old}\mathbf{G}_{(N)}(\otimes^{N-1}\mathbf{A}^{(n)T}) \\ \mathbf{A}^{(N)}_{inc}\mathbf{G}_{(N)}(\otimes^{N-1}\mathbf{A}^{(n)T}) \end{bmatrix}$$

where $\mathbf{X}_{(N)}$ is the mode-N matricized matrix of an accumulated tensor $\mathcal{X}$, and $\mathbf{X}_{(N),old}$ and $\mathbf{X}_{(N),new}$ are the mode-N matricization of a pre-existing tensor $\mathcal{X}_{old}$ and a new incoming tensor slice $\mathcal{X}_{new}$, respectively. By fixing the factor matrix $\mathbf{A}^{(n)}$ for $n = 1, 2, ..., N-1$, we update the factor matrix $\mathbf{A}^{(N)}$ of the temporal mode as follows:

$$\begin{bmatrix} \mathbf{A}^{(N)}_{old} \\ \mathbf{A}^{(N)}_{inc} \end{bmatrix} = \begin{bmatrix} \mathbf{X}_{(N),old}\left(\mathbf{G}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)T})\right)^{\dagger} \\ \mathbf{X}_{(N),new}\left(\mathbf{G}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)T})\right)^{\dagger} \end{bmatrix}$$

By adapting the properties, $(\mathbf{AB})^{\dagger} = \mathbf{B}^{\dagger}\mathbf{A}^{\dagger}$ and $(\mathbf{C} \otimes \mathbf{D})^{\dagger} = \mathbf{C}^{\dagger} \otimes \mathbf{D}^{\dagger}$ to the above equation, we obtain the following equation:

$$\mathbf{A}^{(N)}_{inc} \leftarrow \mathbf{X}_{(N),new}\left(\mathbf{G}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)T})\right)^{\dagger} = \mathbf{X}_{(N),new}\left(\otimes_{k=1}^{N-1}\left(\mathbf{A}^{(k)}\left(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}\right)^{-1}\right)\right)\mathbf{G}^{\dagger}_{(N)}$$

□

A.5 Proof of Lemma 5

Proof. For mode n, we formulate the loss function $L_{(n)}$ as follows:

$$L_{(n)} = \frac{1}{2}\|\mathbf{X}_{(n)} - \mathbf{A}^{(n)}\mathbf{G}_{(n)}(\otimes^{N}_{k\neq n}\mathbf{A}^{(k)})^{T}\|^2 \tag{30}$$

where $(\otimes^{N}_{k\neq n}\mathbf{A}^{(k)})$ indicates Kronecker products of $\mathbf{A}^{(k)}$ for $k = N, N-1, ..., n+1, n-1, ..., 1$. When fixing $\mathbf{A}^{(k)}$ for $k = 1, ..., n-1, n+1, ..., N$, the partial derivative of the function $L_{(n)}$ with respect to $\mathbf{A}^{(n)}$ is as follows:

$$\frac{\partial L_{(n)}}{\partial \mathbf{A}^{(n)}} = -\mathbf{X}_{(n)}(\otimes^{N}_{k\neq n}\mathbf{A}^{(k)})\mathbf{G}^{T}_{(n)} + \mathbf{A}^{(n)}\mathbf{G}_{(n)}(\otimes^{N}_{k\neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}$$

To minimize $\frac{\partial L_{(n)}}{\partial \mathbf{A}^{(n)}}$, we set it to zero and compute $\mathbf{A}^{(n)}$ as follows:

$$\begin{aligned} \mathbf{A}^{(n)} &= \mathbf{X}_{(n)}(\otimes^{N}_{k\neq n}\mathbf{A}^{(k)})\mathbf{G}^{T}_{(n)} \times \left(\mathbf{G}_{(n)}(\otimes^{N}_{k\neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}\right)^{-1} \\ &= \mathbf{P}^{(n)}\left(\mathbf{Q}^{(n)}\right)^{-1} \end{aligned}$$

where $\mathbf{P}^{(n)}$ and $\mathbf{Q}^{(n)}$ are equal to $\mathbf{X}_{(n)}(\otimes^{N}_{k\neq n}\mathbf{A}^{(k)})\mathbf{G}^{T}_{(n)}$ and $\left(\mathbf{G}_{(n)}(\otimes^{N}_{k\neq n}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)}))\mathbf{G}^{T}_{(n)}\right)$, respectively. □

A.6 Proof of Lemma 6

Proof. To update core tensor, we start from the following equation:

$$\mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)} = \mathcal{X}$$

For each mode n, we multiply $\mathbf{A}^{(n)\dagger} = (\mathbf{A}^{(n)T}\mathbf{A}^{(n)})^{-1}\mathbf{A}^{(1)T}$ on both left and right terms. Then, we obtain the core tensor by computing the following equation:

$$\mathcal{G} = \mathcal{X} \times_1 \mathbf{A}^{(1)\dagger} \cdots \times_N \mathbf{A}^{(N)\dagger}$$

For brevity, we compute the core tensor with mode-N matricization. We carefully decouple the computations for $\mathbf{A}^{(N)}_{old}$ and $\mathbf{A}^{(N)}_{new}$. It leads to avoiding explicit computations related to $\mathbf{A}^{(N)}_{old}$ and $\mathbf{X}_{(N),new}$.

$$\begin{aligned}\mathbf{G}_{(N)} &= (\mathbf{A}^{(N)T}\mathbf{A}^{(N)})^{-1} \times \mathbf{A}^{(N)T}\mathbf{X}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})^{-1}) \\ &= \left(\mathbf{Q}^{(N+1)}\right)^{-1}\mathbf{P}^{(N+1)}\end{aligned} \tag{31}$$

where $\mathbf{P}^{(N+1)}$ and $\mathbf{Q}^{(N+1)}$ are equal to $\mathbf{A}^{(N)T}\mathbf{X}_{(N)}(\otimes_{k=1}^{N-1}\mathbf{A}^{(k)}(\mathbf{A}^{(k)T}\mathbf{A}^{(k)})^{-1})$ and $\left(\mathbf{A}^{(N)T}\mathbf{A}^{(N)}\right)$, respectively. □

B PROOFS OF THEOREMS

We provide proof for theorems described in the proposed method sections (Sections 3 and 4).

B.1 Proof of Theorem 1

Proof. The total time complexity of D-Tucker is the summation of time complexities for the three phases: approximation, initialization, and iteration. By Lemmas 1 to 3, the time complexity is $O(I^2K^{N-2}J + MNIK^{N-2}J^2)$, which is simplified from $O(I^2K^{N-2}J + IK^{N-2}J^2 + MNIK^{N-2}J^2)$ without loss of generality. □

B.2 Proof of Theorem 2

Proof. In the initialization phase, initializing for the first two modes requires $O(IK^{N-2}J)$ space to deal with $\left[\mathbf{U}_1\Sigma_1;\cdots;\mathbf{U}_l\Sigma_l\right]$, mode-2 matricization matrix $\mathbf{Y}_{(2)}$, and related tensors in lines 1 to 7 of Algorithm 5. Initializing for the remaining modes requires $O(K^{N-2}J^2)$ to store $\mathbf{Y}_{(i)}$, $\mathcal{Y}_{reuse}$, and related tensors in lines 8 to 17 of Algorithm 5.

The iteration phase requires $O(IK^{N-2}J)$ space for matrices $\mathbf{Y}_{(2),inter}blkdiag(\{\Sigma_i\mathbf{U}_l^{\mathrm{T}}\}_{l=1}^{L})$ and $\mathbf{Y}_{(1),inter}blkdiag(\{\Sigma_l\mathbf{V}_l^{\mathrm{T}}\}_{l=1}^{L})$ in lines 4 and 9 of Algorithm 6. The dominant term for the remaining modes is $O(K^{N-2}J^2)$ to store $\mathcal{Y}_{reuse}$ in line 16 of Algorithm 6.

Considering $I > K > J$, the total space complexity is $O(IK^{N-2}J)$. □

B.3 Proof of Theorem 3

Proof. There are two dominant terms in the time complexity of D-TuckerO: 1) the approximation of a new time slice $\mathcal{X}_{new}$, and 2) n-mode products between the approximation result and factor matrices $\mathbf{A}^{(k)}$ (or $\left(\mathbf{A}^{(k)T}\right)^{\dagger}$). Approximating a new time slice $\mathcal{X}_{new}$ require $O(I^2K^{N-3}T_{new})$ by Lemma 1. In addition, the time complexity of updating all factor matrices is $O(NIK^{N-3}\ T_{new}J^2)$ since updating them includes n-mode products between the approximation of $\mathcal{X}_{new}$ and $\mathbf{A}^{(k)}$ (or $\left(\mathbf{A}^{(k)T}\right)^{\dagger}$) whose complexity is analyzed in Lemma 3. Therefore, the total time complexity of D-TuckerO for each time slice is $O(I^2K^{N-3}T_{new} + NIK^{N-3}T_{new}J^2)$. □

B.4 Proof of Theorem 4

Proof. The space of of D-TuckerO is determined by storing $\mathbf{P}_{(n),old}$ and $\mathbf{Q}_{(n),old}$, and computing $\mathbf{P}_{(n),new}$ and $\mathbf{Q}_{(n),new}$. Space costs of $\mathbf{P}_{(n),old}$ and $\mathbf{Q}_{(n),old}$ are $O((I_1+I_2+(N-3)K)J)$ and $O((N-1)J^2)$ for all $n = 1, ..., N-1$, respectively. We perform n-mode product between $\mathcal{G}$ of the size J^N and $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ for $\mathbf{Q}_{(n),new}$ of the size $J \times J$. Since the intermediate data are always smaller than $\mathcal{G}$,

the space cost of $\mathbf{Q}_{(n),new}$ is $O(J^N)$ which is the size of $\mathcal{G}$. Additionally, the space cost of $\mathbf{P}_{(n),new}$ is $O(IK^{N-3}T_{new}J)$ since the size of the SVD results of $\mathcal{X}_{new}$ is $O(IK^{N-3}T_{new}J)$, and the size of intermediate data of $\mathbf{P}_{(n),new}$ is always smaller than $O(IK^{N-3}T_{new}J)$. The total space cost to update factor matrices and core tensor for $\mathcal{X}_{new}$ is $O((I_1+I_2+(N-3)K)J+(N-1)J^2+J^N+IK^{N-3}T_{new}J)$. We simplify the space cost as $O(IK^{N-3}T_{new}J)$ since the dominant term is to compute $\mathbf{P}_{(n),new}$. □