Fast Alternating LS Algorithms for High Order CANDECOMP/PARAFAC Tensor Factorizations
Anh-Huy Phan, Member, IEEE, Petr Tichavský, Senior Member, IEEE, and Andrzej Cichocki, Fellow, IEEE

Abstract—CANDECOMP/PARAFAC (CP) has found numerous applications in wide variety of areas such as in chemometrics, telecommunication, data mining, neuroscience, separated representations. For an order-N tensor, most CP algorithms can be computationally demanding due to computation of gradients which are related to products between tensor unfoldings and Khatri-Rao products of all factor matrices except one. These products have the largest workload in most CP algorithms. In this paper, we propose a fast method to deal with this issue. The method also reduces the extra memory requirements of CP algorithms. As a result, we can accelerate the standard alternating CP algorithms 20–30 times for order-5 and order-6 tensors, and even higher ratios can be obtained for higher order tensors (e.g., N > 10). The proposed method is more efficient than the state-of-the-art ALS algorithm which operates two modes at a time (ALSo2) in the Eigenvector PLS toolbox, especially for tensors with order N ≥ 5 and high rank.

Index Terms—ALS, CANDECOMP/PARAFAC (CP), canonical decomposition, gradient, tensor factorization.

I. INTRODUCTION

CANDECOMP/PARAFAC (CP), also coined canonical polyadic decomposition, [1], [2] is a common tensor factorization which has found a wide range of applications. For example, CP was applied to analyze the auditory tones by Carroll and Chang [2], or to vowel-sound data by Harshman [1], or to model fluorescence excitation-emission data by hidden loading components in chemometrics [3]. Applications of CP to sensor array processing and CDMA systems in telecommunications have been developed in [4], [5]. In neuroscience, Field and Graupe [6] extracted topographic components model from event-related potentials data, Merup et al. [7] analyzed EEG data in the time-frequency domain. Deburchgraeve et al. developed an automatic method for determination of the seizure location in the neonatal brain [8]. Constantine et al. [9] modeled the pressure measurements along the combustion chamber as order-6 tensors corresponding to the flight conditions—Mach number, altitude and angle of attack, and the wall temperatures in the combustor and the turbulence mode. Hackbusch and Khoromskij [10] investigated CP approximation to operators and functions in high dimensions. Other applications of CP are in time-varying EEG spectrum [11], data mining [12]–[14], separated representations for generic functions involved in quantum mechanics or kinetic theory descriptions of materials [15]. The CP decomposition has been commonly used over the years because of its uniqueness under mild conditions [16]–[18].

Since the alternating least squares (ALS) algorithm was proposed [1], [2], there have been intensive research efforts to improve performance and accelerate convergence rate of CP algorithms. A number of particular techniques have been developed such as line search extrapolation methods [1], [19]–[22], compression [23]. Instead of alternating estimation, all-at-once algorithms such as the OPT algorithm [24], the conjugate gradient algorithm for nonnegative CP [25], the PMF3, damped Gauss-Newton (dGN) algorithms [21], [26] and fast dGN [27]–[29] have been studied to deal with problems of a slow convergence of the ALS in some cases. Another approach is to consider the CP decomposition as a joint diagonalization problem [30]–[32].

The above mentioned CP algorithms can speed-up convergence rate, or cope with difficult problems. However, in most of existing CP algorithms, the largest workload is product of tensor unfoldings and all-but-one factors which has not been adequately considered till now. If a tensor of size $I_1 \times I_2 \times \cdots \times I_N$ is an error tensor of a tensor and its CP approximation, the product expresses the gradient of a least squares cost function with respect to a factor matrix of size $I_n \times R$. Hereafter, we call this product “CP gradient” also referred to as “matricized tensor times Khatri-Rao product” (MTTKRP) [33], [34]. The CP gradients with respect to all the factors have a high computational cost of order $O(NRJ_N)$ where $J_N = \prod_{n=1}^{N} I_n$ (see detailed cost in Table I). In addition, mode-n tensor unfoldings with $n = 2, 3, \ldots, N - 1$ are also time consuming because they permute the order of data entries. For high order tensors ($N \geq 4$), the CP gradients may become very computationally demanding. Experimental results show that it might take several hours to several months on standard computers to factorize order-12 tensors consisting of million or billion entries (e.g., a tensor of size $I_1 \times I_2 \times \cdots \times I_N$, $I_n = 5$) and having rank $R \geq I_n$.

In an effort to handle with the CP gradients (MTTKRP) over all modes, Tomasi [36] proposed a computation method for order-4 tensors which operates two modes at a time (ALSo2) and reduces the largest number of multiplications from $4RJ_N$
to $2RJ_N$. The generalized method for order-$N$ tensor has been implemented as subroutine `alsstep` in the commercial PLS toolbox [37]. However, this method has not yet optimally reduced the computational cost and space cost. In particular, it becomes less efficient when the rank of the decomposition exceeds the largest dimension for odd order $N$ or the product of the two largest ones for even $N$.

In this paper, a fast computation method is proposed for one mode and all mode CP gradients (`MTTKRP`). The CP gradient (`MTTKRP`) is calculated using two smaller Khatri-Rao products to avoid rearranging entries of a tensor in a computer as much as possible. Moreover, progressive computation of all-mode CP gradients has been further improved by exploiting common factors between CP gradients. It not only has a lower computational cost, but also reduces memory requirement. For example, for hypercube tensors of size $I_n = I$ for all $n$, the reduction factor is approximately of $\frac{1}{2}I^{N/2}$ (details given in Table II). As a result, we formulated the FastALS algorithm which is 20–30 times faster than the ordinary ALS algorithm for order-5, order-6 tensors, and achieves much higher ratios for higher order tensors. The proposed method is also faster and less memory demanding than the ALSo2 algorithm [36], [37], especially in decomposition of tensors of order $N \geq 5$ and high rank.

The paper is organized as follows. Notation and basic multilinear algebra are briefly reviewed in Section II. CP model and CP gradients are shortly reviewed in this section. The fast computation method is presented in Section III. The fast implementation of the ALS algorithm utilizing the fast CP gradient is introduced in Section IV. Section V compares the related ALS algorithm, which is ALSo2, [36], [37], especially in decomposition of tensors of order $N \geq 5$ and high rank.

The paper is organized as follows. Notation and basic multilinear algebra are briefly reviewed in Section II. CP model and CP gradients are shortly reviewed in this section. The fast computation method is presented in Section III. The fast implementation of the ALS algorithm utilizing the fast CP gradient is introduced in Section IV. Section V compares the related ALS algorithm, which is ALSo2, [36], [37], especially in decomposition of tensors of order $N \geq 5$ and high rank.

#### II. NOTATION AND CANDECOMP/PARAFAC (CP) MODEL

We shall denote tensors by bold calligraphic letters, e.g., $\mathcal{A} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, matrices by bold capital letters, e.g., $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \ldots, \mathbf{a}_R] \in \mathbb{R}^{R \times K}$, and vectors by bold italic letters, e.g., $\mathbf{a}_j \in \mathbb{R}$ or $\mathbf{I} = [I_1, I_2, \ldots, I_N]$.

#### TABLE I

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Multiplications</th>
</tr>
</thead>
<tbody>
<tr>
<td>M_{Als1}(n)</td>
<td>$R\left(\sum_{j=1}^{n-1} I_j + 1\right) + \sum_{j=1}^{n-1} J_j + J_0$</td>
</tr>
<tr>
<td>M_{Als2}(n)</td>
<td>$R\left(\sum_{k=2}^{n-1} J_k(1-\delta_{nk})\min(J_{k+1}, J_{k-1}) + 1\right) + \sum_{j=1}^{n-1} J_j + J_0$</td>
</tr>
</tbody>
</table>

$\delta_{nk}$: Kronecker delta.

#### TABLE II

<table>
<thead>
<tr>
<th>Method</th>
<th>Extra Memory Requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP_ALS</td>
<td>$RJ_{n-1}K_n$</td>
</tr>
<tr>
<td>FastALS</td>
<td>$R(K_n + J_n)$</td>
</tr>
<tr>
<td>Memory cells</td>
<td>$n = n^<em>, n^</em> + 1$</td>
</tr>
<tr>
<td>Peak memory cells</td>
<td>$RJ_{n-1}K_n$</td>
</tr>
</tbody>
</table>

An $i = [i_1, i_2, \ldots, i_N]$-th entry $y_{ik} = \mathcal{Y}(i_1, i_2, \ldots, i_N)$ with $1 \leq i_n \leq I_n, n = 1, 2, \ldots, N$, is alternatively denoted by $y_{ik}$ with the index $i = \text{ivec}(i, \mathbf{I})$ defined as

$$i = \text{ivec}(i, \mathbf{I}) = i_1 + \sum_{n=2}^{N} (i_n - 1)J_{n-1},$$

where $J_n = \prod_{j=1}^{n} I_j$. We also denote $K_n = \prod_{k=n+1}^{N} I_k, K_0 = J_{N-1}$ and $K_{N-1} = 1$. A vector of integer numbers is denoted by colon notation such as $\mathbf{k} = [i, j, k]$. For example, we denote $\mathbf{k} = [1, 2, \ldots, n]$. Generally, we adopt notation used in [38], [39]. The Kronecker product, the Khatri-Rao (column-wise Kronecker) product, and the (element-wise) Hadamard product are denoted respectively by $\otimes$, $\odot$ [38], [39].

#### Definition 2.1 (Hadamard and Khatri-Rao products): Given a set of $N$ matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$, $n = 1, 2, \ldots, N$, Hadamard and Khatri-Rao products among them are denoted by

$$\bigotimes_{k \neq n} \mathbf{A}^{(k)} \triangleq \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)},$$

$I_n = I$ for all $n$.

$$\bigodot_{n=1}^{N} \mathbf{A}^{(n)} \triangleq \mathbf{A}^{(N)} \cdots \mathbf{A}^{(1)}$$

for all $n$.

$\otimes_{k \neq n} \mathbf{A}^{(k)} \triangleq \mathbf{A}^{(N)} \otimes \cdots \otimes \mathbf{A}^{(n+1)} \otimes \mathbf{A}^{(n-1)} \otimes \cdots \otimes \mathbf{A}^{(1)}$.

#### Definition 2.1 (Reshaping): The reshape$^2$ operator for a tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ to a size specified by a vector $\mathbf{L} = [I_1, I_2, \ldots, I_M]$ with $\prod_{m=1}^{M} I_m = \prod_{n=1}^{N} I_n$ returns an order-$M$ tensor $\mathbf{X}$, such that vec$(\mathcal{Y}) = \text{vec}(\mathbf{X})$, and is expressed as

$$\mathbf{X} = \text{reshape}(\mathcal{Y}, \mathbf{L}) \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_M}.$$  

Reshape does not permute entries in its vectorization.

#### Definition 2.2 (Tensor Unfolding) [35], [36], [40]): Unfolding a tensor $\mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ along modes $\mathbf{r} = [r_1, r_2, \ldots, r_M]$ and $\mathbf{c} = [c_1, c_2, \ldots, c_{N-M}]$ where $[\mathbf{r}, \mathbf{c}]$ is a permutation of $[1, 2, \ldots, N]$ aims to rearrange this tensor to be a matrix $\mathcal{Y}_{\mathbf{r} \times \mathbf{c}}$ of size $\prod_{m=1}^{M} I_m \times \prod_{n=1}^{N-M} I_n$, whose entries $(j_1, j_2)$ are given by $\mathcal{Y}_{\mathbf{r} \times \mathbf{c}}(j_1, j_2) = \mathcal{Y}(j_1, j_2)$, where

$^1$ivec is the “sub2ind” Matlab subroutine.

$^2$see the “reshape” Matlab subroutine.
Remark 2.1:
1) If $c = [c_1 < c_2 < \cdots < c_{N-M}]$, $Y_{rxr}$ is simplified to $Y_{(r)}$.
2) If $r = n$ and $c = 1, 2, \ldots, n-1, n+1, \ldots, N$, we have mode-$n$ matricization $Y_{rxr} = Y_{(n)}$.
3) $Y_{rxr} = Y^{T}_{(n)}$.
4) For $r = 1, 2, \ldots, n$, $c = [n+1, n+2, \ldots, N]$, for all $n$, $Y_{rxr} = Y_{(r)}$ can be expressed and efficiently performed by reshape, that is
\[ Y_{(r)} \triangleq \text{reshape}(Y, [J_n, K_n]). \]

Definition 2.3 (Mode-$n$ Tensor-Vector Product [39]): The mode-$n$ multiplication of a tensor $Y \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ by a vector $a \in \mathbb{R}^{I_n}$ returns an order-$(N-1)$ tensor $Z$ defined as
\[ \text{vec}(Z) = Y^{T}_{(n)}a. \]

Symbolically, the product is denoted by
\[ Z = Y \hat{\times} a \in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N}. \]

Tensor-vector product of a tensor $Y$ with a set of $N$ column vectors $\{a\} = \{a^{(1)}, a^{(2)}, \ldots, a^{(N)}\}$ is denoted by
\[ Y \bar{\times} \{a\} \triangleq Y \hat{x} a^{(1)} \otimes a^{(2)} \otimes \cdots \otimes a^{(N)}. \]

Definition 2.4 (CANDECOMP/PARAFAC (CP)): CP decomposition means an approximation of a given $N$-th order tensor by a rank-$R$ tensor of the form
\[ Y \approx \sum_{r=1}^{R} a^{(1)}_r \otimes a^{(2)}_r \otimes \cdots \otimes a^{(N)}_r = \hat{Y}, \]

where symbol $\otimes$ denotes the outer product, component matrices (factors): $A^{(n)} = [a^{(n)}_1, a^{(n)}_2, \ldots, a^{(n)}_R] \in \mathbb{R}^{I_1 \times R}$, $(n = 1, 2, \ldots, N)$ represent the common (loading) factors [1], [2], [41].

A. Complexity of Tensor Unfoldings

Tensor unfoldings are to rearrange entries of tensors to be matrices. We note that entries of the tensor $Y$ are stored as a long vector $\text{vec}(Y)$ of the size $J_N - \prod_{n=1}^{N} I_n$ in memory. From this viewpoint, tensor unfolding is to change the order to entries in its vectorization. The more the changes of entries take place, the slower the unfoldings are. Moreover, reading data (entries) stored in non-contiguous blocks will be at a slower rate than accessing data stored in a contiguous block.

The mode-1 unfolding $Y_{(1)}$ comprises $K_1 = I_2 I_3 \cdots I_N$ column vectors which consist of $I_1$ contiguous entries of $Y$, i.e.,
\[ Y_{(1)} = \begin{bmatrix} y_{11} & y_{12} & \cdots & y_{1(I_1-1)I_1+1} \\ y_{21} & y_{22} & \cdots & y_{2(I_1-1)I_1+2} \\ \vdots & \vdots & \ddots & \vdots \\ y_{I_1-1} & y_{I_1-1+1} & \cdots & y_{I_1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 3 & \cdots & 23 \\ 2 & 4 & \cdots & 24 \\ \vdots & \vdots & \ddots & \vdots \\ 6 & 12 & \cdots & 23 \end{bmatrix}, \]

where $Q_n$ are commutation matrices of size $J_N \times J_N$, whose explicit expression is given by $Q_n = I_{K_n} \otimes P_{J_{n-1}J_n}$, where $P_{I,J}$ is a permutation matrix for any $I \times J$ matrix $X$ such that $\text{vec}(X) = P_{I,J} \text{vec}(X^T)$ (see Lemma A.1 in [28]). Some other properties of tensor unfolding $Y_{(n)}$, which are useful to avoid tensor permutation, can be found in [36], and in Section II, Paper...
III in [21]. For example, when using an alternative mode-$n$ matrixization which arranges $\mathbf{Y}$ to be a matrix $\mathbf{Y}_{(n)}$ of size $I_n \times I_{n+1} I_{n+2} \cdots I_N I_1 \cdots I_{n-1}$, since $\text{vec}(\mathbf{Y}_{(n)}^T) = \text{vec}(\mathbf{Y}_{(n+1)})$, one only needs a transposition and simple reshaping of the data entries [19], [21].

B. Gradients in CP Algorithms

We consider the cost function

$$ D = \frac{1}{2} \| \mathbf{Y} - \hat{\mathbf{Y}} \|_F^2, \quad (7) $$

and the gradients of this cost function with respect to the factors $\mathbf{A}^{(n)}$, $n = 1, 2, \ldots, N$ are given by [21], [28]

$$
\begin{align*}
\mathbf{G}^{(n)} &= \mathbf{E}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right) \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)^T} \mathbf{A}^{(k)} \right) \\
&= \mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right) - \mathbf{A}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)^T} \mathbf{A}^{(k)} \right) \\
&\in \mathbb{R}^{I_n \times R}, \quad (8)
\end{align*}
$$

where $\mathbf{E}^{(n)}$ denote the mode-$n$ unfoldings of the error tensor $\mathbf{E} = \mathbf{Y} - \hat{\mathbf{Y}}$. The products $\mathbf{E}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$ or $\mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$ have a computational cost of $O(I R J_N)$ (see details in Table I), and are the most expensive steps in CP algorithms. Indeed, the mode-$n$ unfoldings $\mathbf{Y}^{(n)}$, for $n > 1$, are time-consuming. The latter products $\mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$ are more efficient than $\mathbf{E}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$ in the sense of computation because we don’t need to construct the error tensor $\mathbf{E}$. However, since both products involve the same mathematical expression, we also call $\mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$ the CP gradient in which $\mathbf{Y}$ is considered as an error tensor. The product is referred to as MTTKRP in the Matlab Tensor toolbox [33], [34].

The CP gradients are employed in almost all CP algorithms. For example, the alternating least squares (ALS) algorithm [1], [2], [5], [19], [42] alternatively minimizes the cost function (7) with an update rule given by

$$
\begin{align*}
\mathbf{A}^{(n)} &\leftarrow \mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right) \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)^T} \mathbf{A}^{(k)} \right)^{\dagger} \\
&\times \{n - 1, 2, \ldots, N\}, \quad (9)
\end{align*}
$$

where “$\dagger$” denotes the pseudo-inverse. A fast implementation of ALS for 3-way tensor, the ALSO2 algorithm, [21] reduces the expensive computation of $\mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$. The method was later extended to higher order decomposition in the PLS toolbox [37]. See Section V for the ALSO2 algorithm.

The all-at-once algorithms such as OPT [24], PMF3, the damped Gauss-Newton (dGN) or fast Levenberg-Marquardt algorithms [21], [26]-[29], [43], [44], the well-known multiplicative algorithm [38], [45] also compute gradients in their update rules.

The direct computation of $\mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$ for single mode is illustrated in Algorithm 1, and is implemented in the MTTKRP function of the Matlab Tensor toolbox [33], [34].

Algorithm 1: Direct Computation of $\mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)$—MTTKRP [33], [34]

**Input**: Data tensor $\mathbf{Y} : (I_1 \times I_2 \times \cdots \times I_N)$, and $N$ factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$

**Output**: $\mathbf{G}^{(n)} = \mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right) : I_n \times R$

**begin**

1. \begin{align*}
\mathbf{Y} &\leftarrow \text{permute}(\mathbf{Y}, \{n - 1, n + 1, 1 : N'\}) \\
\mathbf{Y}^{(n)} &\leftarrow \text{reshape}(\mathbf{Y}, \{I_n, I_{n-1}, I_{n+1}\}) \\
\mathbf{G}^{(n)} &\leftarrow \mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{A}^{(k)} \right)
\end{align*}

**end**

III. FAST COMPUTATION OF CP GRADIENT

A. Order of Dimensions

The CP gradient $\mathbf{G}^{(n)} = [g_1^{(n)}, g_2^{(n)}, \ldots, g_R^{(n)}] \in \mathbb{R}^{I_n \times R}$ given by

$$
\begin{align*}
\mathbf{G}^{(n)} &= \mathbf{Y}^{(n)} \left( \bigotimes_{k \neq n} \mathbf{a}_k^{(n)} \right) \\
&= \left[ \mathbf{Y} \times_1 \mathbf{a}_1^{(n)} \cdots \mathbf{Y} \times_n \mathbf{a}_n^{(n-1)} \mathbf{Y} \times_{n+1} \mathbf{a}_{n+1}^{(n+1)} \cdots \mathbf{Y} \times_{N-1} \mathbf{a}_{N-1}^{(N)} \right] \quad (10)
\end{align*}
$$

involves $R$ products

$$
\begin{align*}
g_r^{(n)} &= \mathbf{Y} \times_r \mathbf{a}_r^{(n)} \\
&= \mathbf{Y} \times_1 \mathbf{a}_1^{(n)} \cdots \mathbf{Y} \times_n \mathbf{a}_n^{(n-1)} \mathbf{Y} \times_{n+1} \mathbf{a}_{n+1}^{(n+1)} \cdots \mathbf{Y} \times_{N-1} \mathbf{a}_{N-1}^{(N)} \quad (11)
\end{align*}
$$

for $r = 1, 2, \ldots, R$. For $n > 1$, the Kronecker products $\mathbf{t} = \bigotimes_{k \neq n} \mathbf{a}_r^{(k)}$ can be efficiently computed via the following scheme [33], [34]

$$
\begin{align*}
t &\leftarrow \mathbf{a}_1^{(1)} \otimes \mathbf{t}, \\
t &\leftarrow \mathbf{a}_2^{(1)} \otimes \mathbf{t}, \ldots, \mathbf{t} &\leftarrow \mathbf{a}_{n-1}^{(1)} \otimes \mathbf{t}, \\
t &\leftarrow \mathbf{a}_1^{(n-1)} \otimes \mathbf{t}, \mathbf{t} &\leftarrow \mathbf{a}_2^{(n-1)} \otimes \mathbf{t}, \ldots, \mathbf{t} &\leftarrow \mathbf{a}^{(N)} \otimes \mathbf{t}, \quad (12)
\end{align*}
$$

with a computational cost of $O(\sum_{k=1}^{n-1} J_k + \frac{1}{R} \sum_{k=n+1}^{N} J_k)$. When $n = 1$, the cost is given by $O(\frac{1}{R} \sum_{k=3}^{N} J_k)$.

For the least cost to compute $\mathbf{G}^{(n)}$, the dimensions of $\mathbf{Y}$ should be in ascending order, i.e., $I_1 \leq I_2 \leq \cdots \leq I_N$. Note that $\mathbf{t} = \bigotimes_{k \neq n} \mathbf{a}_r^{(k)}$ can be computed from left-to-right, i.e., first computing $\mathbf{t} \leftarrow \mathbf{a}^{(n)} \otimes \mathbf{a}^{(N-1)}$, then $\mathbf{t} \leftarrow \mathbf{t} \otimes \mathbf{a}^{(N-2)}$, etc.

In this case, the dimensions should be in the descending order. Hereafter, we implicitly assume that the tensor has been rearranged in the ascending order of its dimensions. From (12), computation of $\mathbf{G}^{(n)}$ in (10) requires the following number of multiplications

$$
M_{\text{Alg. 1}}(n) = R \left( J_N + \sum_{k=2}^{n-1} J_k + \frac{1}{I_n} \sum_{k=\max[3, n+1]}^{N} J_k \right), \quad (13)
$$
B. Fast Gradient with Respect to a Speciﬁc Factor

The direct computation of \( \mathbf{G}^{(n)} \) in (10) involves the tensor unfolding \( \mathbf{Y}^{(n)} \), which is relatively slow to obtain for \( 1 < n < N \), due to permutation of entries. We note that vectors \( \mathbf{g}^{(n)} \) can be expressed in an equivalent form consisting of tensor-vector products \( \langle \mathbf{Y}^{\times_{k=1}^{n-1} a_r^{(k)}} \rangle \) and \( \langle \mathbf{Y}^{\times_{k=n+1}^{N} a_r^{(l)}} \rangle \) on the left side and right side of \( n \), that is

\[
\mathbf{Y}^{(n)} = \left( \bigotimes_{k=1}^{n} a_r^{(k)} \right) \quad \text{or} \quad \left( \bigotimes_{k=n+1}^{N} a_r^{(l)} \right)
\]

The outer tensor-vector products in (14) have been re-indexed since \( \mathbf{Y} \) is of order-\( (N - n + 1) \). The inner tensor-vector products \( \mathbf{L}^{(r,n)} = \mathbf{Y}^{\times_{k=1}^{n-1} a_r^{(k)}} \) in (14) and \( \mathbf{R}^{(r,n)} = \mathbf{Y}^{\times_{k=n+1}^{N} a_r^{(l)}} \) in (15) can be efﬁciently computed through (16) and (17), respectively. It means that the unfoldings \( \mathbf{Y}^{(n)} \) which are time-consuming for \( 1 < n < N \) are avoided.

Lemma 3.1: Consider an order-\( N \) tensor \( \mathbf{Y} \) with \( I_1 \leq I_2 \leq \cdots \leq I_N \), the right-to-left projections in (14) and the left-to-right projections in (15) require the following number of multiplications

\[
M_{LR}(n) = \left( J_N + \sum_{k=2}^{n-1} J_k + K_{n-1} + \frac{1}{J_n} \sum_{k=n+2}^{N} J_k \right)
\]

or

\[
M_{RL}(n) = \left( J_N + \sum_{k=2}^{n-1} J_k + J_n + \frac{1}{J_{n-1}} \sum_{k=n+2}^{N} J_k \right)
\]

Remark 3.1: (See proof in the Appendix).

- The right-to-left projections in (15) are less computationally demanding than Algorithm 1.
- For \( J_n > K_{n-1} \), the left-to-right projections in (14) are cheaper than the right-to-left projections in (15).

C. Progressive Computation of All Mode CP Gradients

CP algorithms available in the literature compute all \( \mathbf{G}^{(n)} \) for \( n = 1, 2, \ldots, N \), either sequentially (in alternating algorithms [1], [2], [5], [19], [42], [45], [46]) or simultaneously (as in all-at-once algorithms [21], [24], [26]–[29], line-search [20], [21]). This section will present a fast method to compute the gradients recursively for \( n = 1, \ldots, N \).

Note that

\[
\mathbf{L}^{(r,n)} = \mathbf{Y}^{\times_{k=1}^{n-1} a_r^{(k)}} - \mathbf{L}^{(r,n-1)} \times_{1} a_r^{(n-1)}
\]

or

\[
\text{vec} \mathbf{L}^{(r,n)} = \Gamma^{(r,n-1)} \times_{(1)} a_r^{(n-1)}
\]

Similarly,

\[
\mathbf{R}^{(r,n)} = \mathbf{Y}^{\times_{k=n+1}^{N} a_r^{(l)}} - \mathbf{R}^{(r,n+1)} \times_{n+1} a_r^{(n+1)}
\]

or

\[
\text{vec} \mathbf{R}^{(r,n)} = \Gamma^{(r,n+1)} \times_{(1)} a_r^{(n+1)}
\]

By exploiting relations in (23) and (21), we can quickly derive \( \mathbf{R}^{(n)} \) from \( \mathbf{R}^{(n+1)} \), or \( \mathbf{L}^{(n)} \) from \( \mathbf{L}^{(n-1)} \) instead of computing them as in (17) and (16), respectively. The total number of multiplications of the algorithm summarized in Table I indicates that it is lower than that of Algorithm 1.

The proposed algorithm to compute CP gradients over all modes is summarized in Algorithm 2. Gradient \( \mathbf{G}^{(n)} \) (or \( \mathbf{G}^{(n^*+1)} \)) is first computed where

\[
n^* = \max \{ n : J_n \leq K_{n-1} \}
\]

Other gradients \( \mathbf{G}^{(n)} \) are then sequentially computed in the following order \( n = n^* - 1, n^* - 2, \ldots, 1, n^* + 1, n^* + 2, \ldots, N \). Computation of \( \mathbf{G}^{(n^*)} \) and \( \mathbf{G}^{(n^*+1)} \) is the most expensive step. It approximately requires the following number of multiplications

\[
R \left( 2J_N + 2J_{n^*} + K_{n^*} + 2 \sum_{k=2}^{n^*-1} J_k + \frac{1}{J_{n^*}} \sum_{k=n^*+2}^{N} J_k \right)
\]

or

\[
R \left( 2J_N + 2J_{n^*} + 2K_{n^*} \right)
\]

It is straightforward to verify that \( \frac{N}{2} \leq n^* < N \). For \( N = 3 \), the algorithm ﬁrst computes \( \mathbf{G}^{(2)} \), then \( \mathbf{G}^{(1)} \) and \( \mathbf{G}^{(3)} \), sequentially, with a total cost of \( R(2J_N + 3J_2) \). For \( N = 4 \), \( n^* = 2 \) if \( I_1I_2 > I_4 \); otherwise, \( n^* = 3 \).

In comparison with Algorithm 1, besides the lower number of multiplications, Algorithm 2 avoids unfoldings \( \mathbf{Y}^{(n)} \) which are time-consuming for \( 1 < n < N \). The higher the tensor order is, the more signiﬁcant the computational saving of Algorithm 2 in comparison to Algorithm 1 is.

Taking into account that there are additional common parts in computing two consecutive gradients which are \( \mathbf{a}_r^{(N)} \otimes \cdots \otimes \mathbf{a}_r^{(n+2)} \), when computing \( \mathbf{g}^{(n)} \) and \( \mathbf{g}^{(n+1)} \) with \( n > n^* \) in the “left-to-right” strategy (14), and \( \mathbf{a}_r^{(n-2)} \otimes \cdots \otimes \mathbf{a}_r^{(1)} \) when computing \( \mathbf{g}^{(n)} \) and \( \mathbf{g}^{(n-2)} \) with \( n < n^* \) in the “right-to-left” strategy (15). The ALS algorithm which operates two modes at a time (ALS02) in [36], [37] computes \( \mathbf{A}^{(3)} \otimes \cdots \otimes \mathbf{A}_r^{(N)} \) before computing \( \mathbf{G}^{(1)} \) and \( \mathbf{G}^{(2)} \) (see more details in Section V). The similar scheme can be applied to FastALS when \( n^* > 3 \) or \( n^* \leq N - 3 \). For example, we can compute \( \mathbf{a}_r^{(N)} \otimes \cdots \otimes \mathbf{a}_r^{(n^*+2)} \) for both \( \mathbf{g}^{(n^*)} \) and \( \mathbf{g}^{(n^*+1)} \), and \( \mathbf{a}_r^{(N)} \otimes \cdots \otimes \mathbf{a}_r^{(n^*+4)} \) for both \( \mathbf{g}^{(n^*+2)} \) and \( \mathbf{g}^{(n^*+3)} \).
Remark 3.2: The order of dimensions $I_1 \leq I_2 \leq \cdots \leq I_N$ can be arranged to $I_{p_1} \leq I_{p_2} \leq \cdots \leq I_{p_N}$, where $p^* = [p_1, p_2, \ldots, p_N]$ is a permutation of $[1, 2, \ldots, N]$ such that total computational cost of Alg. 2 $T_{n^*} = \sum_{k=1}^{N} M_{A_{LS}}(k)$ is minimum. Note that $J_n$ and $K_n$ in $M_{A_{LS}}(k)$ are replaced by $J_n(\vec{p})$ and $K_n(\vec{p})$ which are similarly defined but calculated from $\{I_{p_1}, I_{p_2}, \ldots, I_{p_N}\}$.

The optimal $n^*$ and $p^*$ can be selected among all possible combinations of $\frac{N}{2} \leq n < N$ and at most $\binom{N}{n}$ permutations $\vec{p}$.

For $N = 3$, the optimal selection is $n^* = 2$ and $p^* = [1, 2, 3]$.

For $N = 4$, there are only two possible values $n^* = 2, 3$, and the total costs $T_{n^*} N(\vec{p}) = \sum_{n=1}^{N} M_{A_{LS}}(n)$ are given by

$$T_{2,4}(\vec{p}) = R \left( 2J_4 + 2J_2^{(p)} + 2K_2^{(p)} + \min \left( J_2^{(p)}, K_1^{(p)} \right) \right) + \min \left( J_3^{(p)}, K_2^{(p)} \right);$$

$$T_{3,4}(\vec{p}) = R \left( 2J_4 + 3J_2^{(p)} + 2K_2^{(p)} + \min \left( J_3^{(p)}, K_2^{(p)} \right) \right).$$

Remark 3.3: For $N = 4$, Alg. 2 attains the lowest cost in dependence on $I_1, I_2, I_3$ and $I_4$:

- If $I_4 < I_1 \left( I_4 + 1 - \frac{I_4}{I_1} \right)$, then $n^* = 2$ and $p^* = [1, 4, 2, 3]$. That is, dimensions should be re-arranged as $[I_1, I_4, I_2, I_3]$.
- Otherwise, $n^* = 3$ and $p^* = [1, 2, 3, 4].$

Remark 3.4: For relatively higher order $N$, the total cost can be approximated by $T_{n^*, N}(\vec{p}) \approx R(2J_4 + 3J_3 + 3K_{n-1})$ when $n^* < N - 1$ and $T_{n^*, N}(\vec{p}) \approx R(2J_4 + 3J_{n-1} + 3K_n)$.

Remark 3.5: For $[1, 2, 3, N]$, is the optimal order for $T_{N-1, N}(\vec{p})$.

Theorem: The remark is deduced from $T_{n^*, N}(\{1 : k - 1, k + 1 : N, k\}) \geq T_{n^*, N}(\{1 : k, k + 2 : N, k + 1\})$ for $k = 1, \ldots, N - 1$.

We do not investigate details on the optimal $n^*$ and $p^*$ for higher order $N \geq 5$ because parameters can be quickly found by a simple code to loop over all possible combinations which are not more than $\sum_{n=\frac{N}{2}}^{N-1} \left( \begin{array}{c} N-1 \vspace{1pt} \end{array} \right) = 2^{N-1} - 1 + \frac{1}{2} \left( \frac{N}{2} \right)$ for even $N$, and $2^{N-1} - 1$ for odd $N$, respectively. For $N = 5, 6$, there are at most 15, 41 combinations, respectively. Although the optimal order of dimensions and update order of factor matrices can further reduce computational cost of Alg. 2 compared that with $I_1 \leq I_2 \leq \cdots \leq I_N$, they do not significantly reduce execution time of Alg. 2.

IV. FAST ALS ALGORITHM

This section presents the fast CP_ALS (FastALS) algorithm in which gradients are computed using Algorithm 2. The update rules (9) are sequentially executed after computing gradients $G^{(n)}$ in Algorithm 2. The fast ALS algorithm first updates $A^{(n^*)}$ instead of $A^{(1)}$, then sequentially updates other factor matrices following the order $n = n^* - 1, n^* - 2, \ldots, 1, n^* + 1, n^* + 2, \ldots, N$.

The cost of the (fast) ALS algorithm is highly dependent on the computational cost of $G^{(n)}$ and cost of computing $G^{(n)}B_n$ where $B_n = \bigotimes_{k \neq n} A^{(k)^T} A^{(k)}$. Evaluation of the cost function

\[
|\mathbf{Y} - \mathbf{\hat{Y}}|_F^2 = |\mathbf{Y}|_F^2 + |\mathbf{\hat{Y}}|_F^2 - 2 \sum_{r} \mathbf{Y} \mathbf{X}_{n=1}^{N-1} a_r^{(n)}
\]

in Step 2

\[
|\mathbf{Y}|_F^2 + 1_R \left( \bigotimes_{n=1}^{N} A^{(k)^T} A^{(k)} \right) 1_R
\]

in Step 2

\[
|\mathbf{L}|_F^2 \left( i_1, \ldots , i_r \right) = |L^{(n)}(i_1, \ldots , i_r)| \text{ in Step 2}
\]

\[
|\mathbf{L}|_F^2 \left( i_1, \ldots , i_r \right) = |L^{(n)}(i_1, \ldots , i_N)| \text{ in Step 4}
\]

(7) is not expensive and it is quickly computed from the last computed gradient $G^{(N)}$ (or $G^{(1)}$) without construction of the approximate tensor $\hat{\mathbf{Y}}$ [33], [47]

**Algorithm 2: Fast Computation of $G^{(n)}$ Over All Modes**

**Input:** Data tensor $\mathbf{Y} : (I_1 \times I_2 \times \cdots \times I_N)$, $N$ factor matrices $A^{(n)} \in \mathbb{R}^{I_n \times K_n}$

**Output:** $G^{(n)} = Y_{(n)} (\bigotimes_{k \neq n} A^{(k)} ; I_n \times K_n, n = 1, 2, \ldots, N)$

1. $n^* = \max \{ n : J_n \leq K_{n-1} \}$ where $J_n = I_1 I_2 \cdots I_n$.
2. $R^{(n)} \leftarrow \text{reshape} \mathbf{Y} \mathbf{X}_{n=1}^{N-1} a_r^{(n)}$
3. $G^{(n)} = \text{cp.gradient}(R^{(n)}, [A] \cdot \text{right})$
4. if $(n \neq n^* + 1)$ then
5. $I^{(n)} = \left( \bigotimes_{k \neq n} A^{(k)^T} \right) \text{reshape} \mathbf{Y} \mathbf{X}_{n=1}^{N-1} $
6. $G^{(n)} = \text{cp.gradient}(\mathbf{L}^{(n)} ; [A] \cdot \text{left}\cdot\cdot\cdot$
7. else if $(n < n^*)$
8. $G^{(n)} = \text{cp.gradient}([A] \cdot \text{side})$
9. switch side
10. $g^{(n)} = \text{reshape} \mathbf{Z}^{(n)} \mathbf{X}_{n=1}^{N-1} a_r^{(n)} \% \mathbf{Z}^{(n)} = \mathbf{R}^{(n)}(n)$ in (31)
11. $g^{(n)} = \text{reshape} \mathbf{Z}^{(n)} \mathbf{X}_{n=1}^{N-1} a_r^{(n)} \% \mathbf{Z}^{(n)} = \mathbf{L}^{(n)}(n)$ in (32)
12. $g^{(n)} = \text{reshape} \mathbf{Z}^{(n)} \mathbf{X}_{n=1}^{N-1} a_r^{(n)} \% \mathbf{Z}^{(n)} = \mathbf{L}^{(n)}(i_1, \ldots , i_N)$ in Step 2
13. $g^{(n)} = \text{reshape} \mathbf{Z}^{(n)} \mathbf{X}_{n=1}^{N-1} a_r^{(n)} \% \mathbf{Z}^{(n)} = \mathbf{L}^{(n)}(i_1, \ldots , i_N)$ in Step 4

where $1_R$ is a vector of ones, and $|\mathbf{Y}|_F^2$ is computed only once or can be neglected. Hence, when $n^* < N - 1$, the complexity per iteration of FastALS is given by
1) Computing $G^{(n)} \sum_{n=1}^{N} M_{\text{alg}2}(n) \approx R(2J_{n} + 3J_{n*} + 3K_{n*})$

2) Computing $\Gamma_{n} N(N - 2)R^{2} + R^{2}T, T = \sum_{n=1}^{N} I_{n}$

3) Computing $G^{(n)} \Gamma_{n}^{\top}, NR^{2} + R^{2}T$

4) Evaluating (25) $R^{2} + RI_{N}$

Total $\approx R(2J_{n} + 3J_{n*} + 3K_{n*}) + R^{2}(2T + N^{2}) + NR^{3}$

When $n^{*} = N - 1$, the approximate total cost is $R(2J_{n} + 3J_{n*} + 3J_{n-1}) + R^{2}(2T + N^{2}) + NR^{3}$. The last three steps are common in both ordinary and fast ALS algorithms. Computation of all $G^{(N)}$ in the ordinary ALS algorithm costs

$$\sum_{n=1}^{N} M_{\text{alg}1}(n) = R \left( N J_{n} + \sum_{n=2}^{N-1} (N-n) J_{n} + \sum_{n=3}^{N} \left( \sum_{k=1}^{n-1} \frac{1}{I_{k}} \right) J_{n} \right) \approx R \left( N + \sum_{n=1}^{N} \frac{1}{I_{n}} \right) J_{n}. \quad (26)$$

Concerning the memory consumption, in order to update the factor $A^{(n-1)}$, FastALS first computes the right-side Khatri-Rao product of size $(K_{n*} \times R)$, and then the left-side Khatri-Rao product of size $(J_{n-1} \times R)$. A projection matrix $[\text{vec}(R^{(1,n)}), \ldots, \text{vec}(R^{[R,n]})]$ of size $(J_{n} \times R)$ is kept to update other factors. Hence, it needs an extra temporary $O(R(K_{n*} + J_{n*}))$ memory cells besides those of the tensor and the factors.

For updating factors $A^{(n)}$, $n = n - 1, \ldots, 3, 2$, FastALS does not need to access the raw tensor. It utilizes the projection matrix obtained when updating $A^{(n+1)}$, and constructs a new projection matrix $[\text{vec}(R^{(1,n)}), \ldots, \text{vec}(R^{[R,n]})]$ of size $(J_{n} \times R)$. Moreover, the FastALS algorithm computes only the left-side Khatri-Rao product of size $(J_{n-1} \times R)$; hence, FastALS requires a temporary $O(R(2J_{n} + K_{n*}))$ memory cells. Note that updating $A^{(1)}$ does not need to compute any Khatri-Rao product and tensor unfolding.

Similarly, to update the factor $A^{(n+1)}$, FastALS algorithm first computes the left-side Khatri-Rao product of size $(J_{n} \times R)$, then the right-side Khatri-Rao product of size $(K_{n*} \times R)$, and yields a projection matrix $[\text{vec}(L^{(1,n)}), \ldots, \text{vec}(L^{[R,n]})]$ of size $(K_{n} \times R)$. It requires $O(R(J_{n} + K_{n*}))$ memory cells. Updating factors $A^{(n)}$, for $n = n-2, \ldots, N-1$, requires a temporary memory of order $O(R(K_{n} + J_{n*}))$, while updating $A^{(N)}$ does not compute any Khatri-Rao product.

The extra temporary memory of FastALS is summarized in Table II. Without taking into account the unfoldings $Y_{(n)}$ of size $I_{n} \times J_{n-1}K_{n}$, the ordinary CP_ALS algorithm requires an extra $O(R(J_{n} + K_{n}))$ memory cells for the Khatri-Rao products of size $(J_{n} \times K_{n} \times R)$, $(n = 1, 2, \ldots, N)$. Note that $J_{n-1}K_{n+1}I_{n} > (J_{n-1} + K_{n+1})I_{n} \geq J_{n} + K_{n}$,

- $J_{n} < K_{n}, \quad n < n^{*}$
- $J_{n} > K_{n}, \quad n > n^{*}$

Therefore, the FastALS algorithm requires much smaller extra temporary memory than CP_ALS. The maximum extra space of FastALS is of $O(R(J_{n*} + K_{n*}))$ memory cells as updating $A^{(n*)}$ or $A^{(n+1)}$. This confirms that FastALS not only has a lower computational cost, but also requires less memory than CP_ALS. Other alternating algorithms for CPD with/without additional constrains such as nonnegativity, orthogonality [38], [45], [46], [48], [49] can be accelerated in a similar way.

V. RELATED WORKS

This section compares Algorithm 2 with the algorithm proposed by Tomasi in [36], which operates two modes at a time (ALSo2), and is implemented as the subroutine alstep in the commercial PLS toolbox [37].

The ALSo2 algorithm reduces the largest number of multiplications $NRJ_{N}$ to $2RJ_{N}$, but it has not yet optimally processed the order of dimensions $I_{n}$ which is also important to reduce the computation cost and space cost. Denote $Z^{(r, n)} = L^{(r, n)}x_{n}^{(n+1)}a_{n}^{n+1} \in R^{1 \times J_{n+1}}, ALSo2$ [36], [37] operates two modes $n$ and $(n+1)$ at a time

$$g_{(n)}^{(n)} = Z^{(r, n)}a_{(n+1)}^{(n)}; \quad g_{(n+1)}^{(n+1)} = Z^{(r, n)}T a_{(n)}^{(n)}; \quad (r = 1, 2, \ldots, R) \quad (27)$$

through the left-side projection as in (16) with

$$L^{(r, n)} = L^{(r, n-2)} \bar{x}_{1}a_{(n-2)}^{(n-2)} \bar{x}_{2}a_{(n-1)}^{(n-1)} \in R^{1 \times J_{n+1} \times \cdots \times J_{N}}, \quad (28)$$

where $r = 1, 2, \ldots, R$, $n = 3, 5, 7, \ldots$, and $L^{(r, 1)} = \Phi$. The ALSo2 algorithm first computes $G^{(1)}$ when $N$ is odd or computes $G^{(2)}$ and $G^{(4)}$ for even order $N$.

For example, when $N = 3$, ALSo2 first computes $G^{(1)} = Y_{(1)}(A^{(3)} \odot A^{(3)})$ with a cost of $(RJ_{N} + RK_{1})$, then computes $Y_{(1)}^{(1)}A^{(1)}$ with a cost of $RI_{N}$, and $G^{(2)}$ and $G^{(3)}$ with a cost of $2RK_{1}$. The total cost of ALSo2 is of $R(2J_{N} + 3K_{1})$. A minimum cost is achieved when $J_{2}$ and $I_{2}$ are the two shortest dimensions. That is, ALSo2 [[36], [37]] may need a tensor permutation so that $I_{1} \geq I_{2} \geq I_{3}$.

For order-4 tensors, ALSo2 sequentially updates factors $A^{(2)}$ and $A^{(1)}$, then $A^{(3)}$ and $A^{(4)}$, and requires a total cost of $R(2J_{N} + 3J_{2} + 3K_{3})$. The optimal order of dimensions for ALSo2 is $|I_{1}, I_{2}, I_{3}, I_{2}|$ which is similar to that for FastALS when $n^{*} = 2$ in Remark 3.3. However, when $n^{*} = 3$, e.g., $I_{1}I_{2} \leq I_{3}$, FastALS updates factors from right to left, and has a lower cost of $R(2J_{N} + 3J_{3} + 3J_{2})$ than ALSo2 (see decomposition of $10 \times 10 \times 10 \times 4(100)$ dimensional tensors in Table III).

For higher order tensors, in general, the matrices $Z^{(r, n)}$ cost $R \left( \sum_{J_{n-1}+1}^{J_{n}} N \sum_{J_{n-1}+1}^{J_{n}} J_{k} \right)$ multiplications, and $L^{(r, n)}$ cost $R \left( \sum_{J_{n-2}+1}^{J_{n-1}} \sum_{J_{n-2}+1}^{J_{n-1}} J_{k} \right)$ multiplications. Hence, for even orders $N (6, 8, \ldots)$, the largest workloads are involved in the computation of $Z^{(r, 3)}$, $L^{(r, 3)}$, $Z^{(r, 3)}$, $L^{(r, 5)}$, $Z^{(r, 5)}$, and $L^{(r, 7)}$ with a total cost of

$$R \left( K_{0} + \frac{1}{J_{k-4}} \sum_{k=4}^{J_{4}} J_{k} + K_{0} + I_{1}I_{2} + K_{2} \right) \approx R(2J_{N} + 3K_{3} + 3K_{4}). \quad (29)$$

For odd orders $N (5, 7, \ldots)$, the computation cost of ALSo2 [36],[37] is approximately of $R(2J_{N} + 3K_{1} + 3K_{2})$. The algo-
Algorithm requires an extra temporary $RK_1$ or $RK_2$ memory cells for odd or even order $N$, respectively. The optimal order of $I_n$ for ALSo2 can be chosen such that $K_1 + K_2$ or $K_2 + K_4$ is minimum. In general, the dimensions should be in the descending order. The optimal mode permutation is performed just once before the FastALS and ALSo2 algorithms are even started.

We note that ALSo2 [36], [37] can be considered as a particular case of FastALS with a general permutation $p$ in Remark 3.3 when $n^* - N - 1$ for $N = 3, 5, 7, \ldots$ and $n^* - N - 2$ for $N = 4, 6, 8, \ldots$. This algorithm updates factor matrices from right to left. Therefore, in general, FastALS always has lower or equal cost as ALSo2 with an optimal tensor permutation.

VI. SIMULATIONS

A. Factorization of Synthetic Tensors

Simulations in this section compare execution times of the ALS algorithms: the ordinary CP_ALS algorithm [1], [2], [34], the FastALS in Section IV, the ALSo2 algorithm, i.e., the subroutine alsstep which operates two modes at a time in the commercial PLS toolbox [36], [37].

Comparison of execution times (second) per iteration between CP ALS, FastALS and ALSo2 [35], [36] in factorization of random tensors. Execution times per iteration and speed-up ratio between algorithms are shown as indicated in the below sub-table. Computer PC1, used in all scenarios, has 1.8 GHz i7 processor and 4 GB memory. Computer PC2, used in the last scenario, had 96 GB of RAM and two six-core processors X5690@3.47 GHz.

Example 1 [Factorization of Random Tensors]: We decomposed order-$N$ tensors which were randomly generated from normal distribution with zero mean and unit variance in single-precision floating point with different sizes $I_n = I = 10, 20, \ldots$ for all $n$. Algorithms factorized the same tensors using the same initial values and ran in 20 iterations without any other stopping criterion. Execution times were measured for various ranks $R$ using the stopwatch command: “tic” “toc” of Matlab release 2011a on a computer (PC1) with a 1.8 GHz i7 processor and 4 GB RAM and the Mac OS X 10.7 operating system. In addition, memory usage of algorithms including allocated memory and peak memory was measured using the Matlab profiler.

Speed ratio between CP_ALS and FastALS is defined as the ratio between their execution times per iteration

$$\rho = \frac{\text{Execution time}_{\text{ALS}}}{\text{Execution time}_{\text{FastALS}}}. \quad (30)$$

The final results were averaged over at least 20 iterations $\times 10$ runs. Fig. 1 shows how the speed-up ratio per iteration (times) changes for factorizations of order-3 and order-4 tensors with different sizes $I_n = I$ for all $n$, and ranks $R$. Other results are summarized in Table III. The ratios were relatively high for low rank $R$, and gradually decreased when increasing $R$. The fact is that the computational cost of FastALS was also affected by rank $R$, e.g., the term $NT^3 + TR^2$ where $T = \sum_{n=1}^{N} I_n$ as analyzed in Section IV. Another reason is due to the array-based language Matlab which is relatively slow for relatively high $R$ when executing codes relying on “for” loops. Memory requirement is also an important issue for the ALS algorithms. The higher the rank-$R$ is, the more memory cells the algorithms require. Since CP_ALS and ALSo2 [37] without proper tensor permutation are much more space-consuming than FastALS, their memory requirements exceed the memory bound sooner than that of FastALS. For this case, the ratio will increase as increasing $R$. For example, in order to factorize order-4 tensors with $I_n = 120$, into $R = 50$ rank-one tensors, FastALS allocated 22 MB of physical memory on average per iteration with a peak memory of 5.5 MB for the term $(\mathbf{C}_{k=1}^{n^*} \mathbf{A}^{(k)})$ where $n^* = 2$. For the same task, CP_ALS allocated 5.1 GB of memory cells which included the tensor unfoldings $\mathbf{Y}_{(n)}$ and 691 MB of memory for each product $(\mathbf{C}_{k=1}^{n^*} \mathbf{A}^{(k)})$, while the computer had only 4 GB of memory. This explains why the speed-up ratio increased when $R \geq 50$ for tensors of size $120 \times 120 \times 120 \times 120$ as illustrated in Fig. 1(b).

Besides the comparison between FastALS and ALS, ALSo2 [37] is also compared with FastALS in Table III. For example, in decompositions of tensors of size $30 \times 100 \times 100$ and $5 \times 10 \times 500 \times 1000$, FastALS was 4–5 times faster than ALSo2 without optimal tensor permutation. Notice that with a tensor permutation $p = [3, 2, 1]$ or $p = [1, 4, 2, 3]$, ALSo2 is exactly
TABLE III

COMPARISON OF EXECUTION TIMES (SECOND) PER ITERATION BETWEEN CP_ALS, FastALS and ALS02 [36], [37] IN FACTORIZATION OF RANDOM TENSORS. EXECUTION TIMES PER ITERATION AND SPEED-UP RATIO BETWEEN ALGORITHMS ARE SHOWN AS INDICATED IN THE BELOW SUB-TABLE. (a) EXPERIMENT RESULTS ON COMPUTER PC1, (b) EXPERIMENT RESULTS ON COMPUTER PC2

<table>
<thead>
<tr>
<th>Tensor size</th>
<th>CP_ALS</th>
<th>FastALS</th>
<th>ALS02</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>30x10x1000x1000</td>
<td>0.115</td>
<td>0.087</td>
<td>0.066</td>
<td>1.362</td>
</tr>
<tr>
<td>5x10x500x1000</td>
<td>0.07</td>
<td>0.055</td>
<td>0.039</td>
<td>1.992</td>
</tr>
<tr>
<td>40x10x20x30x50</td>
<td>0.02</td>
<td>0.012</td>
<td>0.007</td>
<td>1.517</td>
</tr>
<tr>
<td>4x10x40x20x50x1000</td>
<td>0.06</td>
<td>0.04</td>
<td>0.02</td>
<td>2.77</td>
</tr>
<tr>
<td>30x10x20x30x50</td>
<td>0.02</td>
<td>0.01</td>
<td>0.007</td>
<td>1.517</td>
</tr>
<tr>
<td>40x10x40x20x50x1000</td>
<td>0.06</td>
<td>0.04</td>
<td>0.02</td>
<td>2.77</td>
</tr>
<tr>
<td>30x10x20x30x50</td>
<td>0.02</td>
<td>0.01</td>
<td>0.007</td>
<td>1.517</td>
</tr>
<tr>
<td>40x10x40x20x50x1000</td>
<td>0.06</td>
<td>0.04</td>
<td>0.02</td>
<td>2.77</td>
</tr>
<tr>
<td>30x10x20x30x50</td>
<td>0.02</td>
<td>0.01</td>
<td>0.007</td>
<td>1.517</td>
</tr>
<tr>
<td>40x10x40x20x50x1000</td>
<td>0.06</td>
<td>0.04</td>
<td>0.02</td>
<td>2.77</td>
</tr>
</tbody>
</table>

(a) EXPERIMENT RESULTS ON COMPUTER PC1

(b) EXPERIMENT RESULTS ON COMPUTER PC2

FastALS. Table III shows the cases when ALS02 and FastALS are identical, i.e., the speed ratio is one.

For order-4 tensors of size 10x10x10x1000, since $n^* = 3$, FastALS updated factors in the order 3, 2, 1, 4. Table III shows execution times of ALS02 [37] without and with optimal tensor permutation. For these order-5 tensors, FastALS required only a few seconds per iteration on PC1. Even when running on a computational server (PC2) which had 96 GB of RAM and two six-core Intel Xeon processors X5690@3.47 GHz and the Windows 7 operating system, ALS02 with optimal tensor permutation was still at least 10 times (up to 18 times) slower than FastALS.

Results for higher order tensors with $N = 10, 11, 12, 13$ and $I_n = 5$ for all $n$ are summarized in Table IV. The execution times were averaged over 30 iterations when $R = 5, 10, 20$ on two computers PC1 and PC2. In addition to speed-up ratios CP_ALS/FastALS and ALS02/FastALS, the total allocated and peak memory requirements of algorithms are provided. ALS02 was faster than CP_ALS, but more time consuming than FastALS.

For higher order tensors with $N = 5, 6, 7$, the products $A(k)$ occupied not more than 1.562.500 memory cells which in double precision format consumed only 12 MB of memory. For $N = 12$ and $R = 5$, the products $A(k)$ comprised $I^{N-1} \times R = 5^{12} \approx 24416^6$ double-precision numbers which might consume 1.82 GB of RAM. It means that PC1 had insufficient memory for CP_ALS and ALS02 for $N \geq 12$, and FastALS consumed much less memory than CP_ALS. The speed-up ratio increased as increasing $R$ from 5 to 20. Even when factorizing order-13 tensors, FastALS was still relatively fast and need approximately 3 seconds/iteration.

B. Factorizations of EEG Data

Example 2 [Factorization of Event-Related EEG Time-Frequency Representation]: This example illustrates application of the FastALS algorithm for analysis of real-world EEG data [7], [45] which consists of 28 inter-trial phase coherence (ITPC) measurements [45] of EEG signals of 14 subjects during a proprioceptive pull of the left and right hands. The whole ITPC data set has been represented as a 4-way tensor of 28 measurements $\times 61$ frequency bins $\times 64$ channels $\times 72$ time frames. The first 14 measurements are associated to a group of the left hand stimuli, while the other ones are with the right hand stimuli. Morup et al. analyzed the dataset by nonnegative CP and Tucker components and compared them with components extracted by NMF and ICA [7].

In this example, we approximated the ITPC tensor $\mathbf{Y}$ by CP tensors with various $R = 1, 2, \ldots, 50$. Our aim was to compare the factorization time of CP_ALS and FastALS over various $R$ while interpretation of the results can be found in [7], [45]. Since $n^* = 2$, ALS02 and FastALS have identical cost. We only compared CP_ALS and FastALS. Both algorithms used the same initial values and stopped when their differences of successive relative errors $\varepsilon = \left| \mathbf{Y} - \mathbf{Y}_A \right| / \left| \mathbf{Y} \right|$ were lower than $10^{-6}$, or until the maximum number of iterations (2000) was achieved. Factorizations were performed on two computers PC1 and PC2. In addition, the factor matrices in CP_ALS were updated in the same order as in FastALS. This ensured the estimated factors and the numbers of iterations of two algorithms were equivalent up to machine precision.
TABLE IV

<table>
<thead>
<tr>
<th>N</th>
<th>R</th>
<th>Execution time per iteration (seconds)</th>
<th>Allocated Memory (MB)</th>
<th>Peak Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FastALS (1)</td>
<td>ALSo2 (2)</td>
<td>CP_ALS (3)</td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>0.018 (0.043)</td>
<td>0.12</td>
<td>1.9 (4.1)</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>0.018 (0.054)</td>
<td>0.20</td>
<td>3.2 (5.5)</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>0.021</td>
<td>0.38</td>
<td>8.7</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
<td>0.085 (0.188)</td>
<td>2.25 (2.85)</td>
<td>10.8 (20.3)</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>0.089 (0.245)</td>
<td>4.39 (5.37)</td>
<td>17.5 (30.8)</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>0.098</td>
<td>10.1</td>
<td>43.9</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>0.534 (1)</td>
<td>3.04 (8.91)</td>
<td>59.4 (2257)</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>0.427 (1.171)</td>
<td>5.09 (40)</td>
<td>96.4</td>
</tr>
<tr>
<td>20</td>
<td>10</td>
<td>0.456</td>
<td>14.6</td>
<td>250</td>
</tr>
<tr>
<td>5</td>
<td>13</td>
<td>3.71</td>
<td>59.4</td>
<td>331</td>
</tr>
<tr>
<td>13</td>
<td>10</td>
<td>2.62</td>
<td>123</td>
<td>535</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>2.76</td>
<td>1533</td>
<td>2491</td>
</tr>
</tbody>
</table>

Execution times of the two algorithms illustrated in Fig. 2 indicate that FastALS was faster than CP_ALS on both machines. While CP_ALS required 110–240 seconds to factorize the tensor into R > 30 components on PC2, FastALS completed the factorizations only in several seconds, and achieved a high speed-up ratio ρ ≈ 25–36 times. The speed ratio on PC1 was around 10–15 times for R ≥ 20 and lower than that on PC2. The speed-up ratio depended on the CPU power of the system. We note that PC2 was a computational server with 2 × 3.47 GHz processors each of which had 6 cores, while PC1 was only a laptop with only one Core i7 1.8 GHz. The execution times of CP_ALS on PC1 were approximately two times lower than those of this algorithm on PC2, while the execution time ratio of FastALS on PC1/PC2 varied from 1 to 7 times as increasing R from 1 to 50. It indicates that FastALS was more efficient on PC2 than PC1.

Example 3 [Factorization of EEG Motor Imagery Data]:

In the next set of simulations, we emphasized the superior efficiency of FastALS in comparison to CP_ALS and ALSO2 for factorization of high order tensor which involves left/right motor imagery (MI) movements. We analyzed the EEG MI dataset3 for the subject 1 which was recorded from 62 channels at a sampling rate of 500 Hz in a duration of 2 seconds per trial. The total number of trials was 200 (100 trials for each class). EEG signals were transformed into the time-frequency domain using the two complex Morlet wavelets CMOR1-1 and CMOR6-1 with the bandwidth parameters f_b = 1 Hz and f_s = 6 Hz, and the wavelet center frequency f_c = 1 Hz [51], giving an order-5 tensor with modes 2 dictionaries × 23 frequency bins × 50 time frames × 62 channels × 200 trials.

The order-5 tensor was factorized by CP models with various ranks R = 1, 2, ..., 80. Algorithms were initialized with the same values, and stopped when their differences of successive relative errors ε = ∑||A(ν)| - |A(ν-1)||/||A(ν)|| were lower than 10−6, or until the maximum number of iterations (2000) was achieved. The order of factor matrices to be updated in CP_ALS was the same as in FastALS so that both algorithms took the same number of iterations. The number of iterations of ALSO2 were set to that of FastALS. In addition to FastALS with n* = 3 and dimensions in the ascending order, we executed FastALS with the optimal order p = [1, 3, 4, 2, 5], ALSO2 (without reordering dimension) and ALSO2 with a tensor permutation p = [5, 4, 3, 2, 1] were both executed.

Factorization times of algorithms on PC2 are illustrated in Fig. 3. When R = 20, FastALS demanded a total 7.6 MB of RAM per iteration with a peak memory of 1.9 MB to allocate the product A(5) ⊗ A(4) comprising 210 × 62 × 20 = 248000 entries in the double precision floating-point type. Both CP_ALS and ALSO2 without optimal tensor permutation computed the Khatri-Rao product (C_5^2 A(A^4)) which consumed 23 × 50 × 62 × 200 × 20 × 8 ≈ 2.12 GB RAM. Moreover, CP_ALS and ALSO2 [36], [37] without a proper tensor permutation allocated a total memory of 5.83 GB and 6.52 GB per iteration respectively, which exceeded the specification of PC1.

With the best tensor permutation p = [5, 4, 3, 2, 1] prior to the factorization, ALSO2 [36], [37] was significantly faster than CP_ALS, but it was still slower than FastALS. The largest Khatri-Rao product computed for the rank-20 case by ALSO2 was

3The data set for single trial EEG classification in BCI is provided by Center for Brain-Like Computing and Machine Intelligence, Shanghai Jiao Tong University, http://bcmi.sjtu.edu.cn/data1/.
FastALS makes standard PCs with relatively low physical memory applicable to factorization of huge and high order tensors. The FastALS algorithm and other alternating and all-at-once algorithms using fast CP gradient are implemented in the Matlab package TENSORBOX which is available online at: http://www.bsp.brain.riken.jp/~phan/tensorbox.php.

APPENDIX A
PROOF OF LEMMA 3.1

The projection in (15) first computes the order-n tensors $\mathbf{R}^{(r,n)}$ defined in (17) for $r = 1, \ldots, R$, then computes $\mathbf{g}^{(n)}_r$ from mode-n unfolding $\mathbf{F}^{(r,n)}_n$ of $\mathbf{R}^{(r,n)}$:

$$
\mathbf{g}^{(n)}_r = \mathbf{R}^{(r,n)}_n \left( \bigotimes_{k=1}^{n-1} \mathbf{a}^{(k)}_r \right). 
$$

The two steps require the number of multiplications

$$
M_{RL1} = R \left( J_n + \sum_{k=2}^{N} J_k \right),
M_{RL2} = R \left( J_n + \sum_{k=2}^{n-1} J_k \right).
$$

Hence, the total number of multiplications is given as in (19).

For the left-to-right projection in (14), we build up order-$(N - n + 1)$ tensors $\mathbf{L}^{(r,n)}$ of size $I_1 \times I_{n+1} \times \cdots \times I_N$ defined in (16), and compute $\mathbf{g}^{(n)}_r$ from mode-1 unfolding $\mathbf{I}^{(r,n)}_{(1)}$ of $\mathbf{L}^{(r,n)}$:

$$
\mathbf{g}^{(n)}_r = \mathbf{I}^{(r,n)}_{(1)} \left( \bigotimes_{k=2}^{n+1} \mathbf{a}^{(k)}_r \right). 
$$

The two steps respectively require the following number of multiplications

$$
M_{LR1} = R \left( J_n + \sum_{k=2}^{n-1} J_k \right),
M_{LR2} = R \left( K_{n-1} + \sum_{k=2}^{n-1} J_k \right).
$$

APPENDIX B
PROOF OF REMARK 3.1

Since $I_n < I_{n+1}$ and $I_n < J_n$, we easily have

$$
M_{RL}[n] = R \left( J_n + \sum_{k=2}^{n-1} J_k \right) \leq R \left( J_n + \sum_{k=2}^{n-1} J_k \right) - M_{A1}(n).
$$

for $1 < n < N$. It means that the right-to-left projections should be faster than Algorithm 1.

ACKNOWLEDGMENT

The authors would like to thank the referees for the very constructive comments and suggestions which helped to improve
the quality and presentation of the paper. They also thank for the referee who has provided us the Matlab code of the ALSo2 algorithm (the subroutine alsstep) [36, 37].

REFERENCES


