Abstract—CANDECOMP/PARAFAC (CP) has found numerous applications in a 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., $N \geq 10$). 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 $N \geq 5$ and high rank.

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

I. INTRODUCTION

CANDECOMP/PARAFAC (CP), also known as 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, Mørup [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], [13], [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], [17], [18].

Since the alternating least squares (ALS) algorithm was proposed [2], [1], 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], [20], [21], [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 [26], [21] and fast dGN [27], [28], [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], [31], [32].

The above mentioned CP algorithms can speed-up convergence rate, or cope with difficult problems. However, in most 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 alstep 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 Krat-Rao products to avoid rearranging entries of a tensor in a computer as much as possible. Moreover, progressive computation of allmode CP gradients has been further improved by exploiting common factors between CP gradients. It not only has a lower computational cost of $2K \sum_{n=1}^{N} I_n$, but also reduces memory requirement. For example, for hypercube tensors of size $I_n = I$ for all $n$, the reduction factor is approximately of $\frac{1}{2} I^{N/2-1}$ (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 ($N \geq 10$). 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 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] with the proposed algorithm. In Section VI we provide examples illustrating the validity and high performance of the proposed algorithm. Section VII concludes the paper.

II. NOTATION AND CANDECOMP/PARAFAC (CP) MODEL

We shall denote tensors by bold calligraphic letters, e.g., $\mathcal{A} \in \mathbb{R}^{I \times J \times K}$, matrices by bold capital letters, e.g., $\mathbf{A} = \{a_{ijk}\} \in \mathbb{R}^{I \times J \times K}$, and vectors by bold italic letters, e.g., $\mathbf{a}$ or $I = \{I_1, I_2, \ldots, I_N\}$. An index $i = (i_1, i_2, \ldots, i_n)$-th entry $y_{i} = y(i_1, i_2, \ldots, i_n)$ with $1 \leq i_n \leq I_n$, $n = 1, 2, \ldots, N$, is alternatively denoted by $y_i$ with the index $i = \text{ivect}(i, I)$ defined as

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

where $J_n = \prod_{j=1}^{n} I_j$. We also denote $K_n = \prod_{k=n+1}^{N} I_k$, $K_0 = J_N$ and $K_N = 1$. A vector of integer numbers is denoted by colon notation such as $k = i:j = \{i, i+1, \ldots, j-1, j\}$. For example, we denote $1:n = \{1, 2, \ldots, n\}$.

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 $\otimes, \odot, \odot$ [39], [38].

Notation 2.1 (Hadamard and Khatri-Rao products): Given a set of $N$ matrices $\mathbf{A}^{(n)} = \mathbb{R}^{I \times J}$, $n = 1, 2, \ldots, N$,

Hadamard and Khatri-Rao products among them are denoted by

$$\otimes_{k=1}^{N} \mathbf{A}^{(k)} = \mathbf{A}^{(N)} \otimes \cdots \otimes \mathbf{A}^{(n+1)} \otimes \mathbf{A}^{(n-1)} \otimes \cdots \otimes \mathbf{A}^{(1)},$$

$$I_n = I$$ for all $n$.

$$\odot_{n=1}^{N} \mathbf{A}^{(n)} = \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \cdots \odot \mathbf{A}^{(1)}$$

for all $n$.

$$\odot_{k=1}^{N} \mathbf{A}^{(k)} = \mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \cdots \odot \mathbf{A}^{(1)}.$$


Definition 2.1 (Reshaping): The reshape operator for a tensor $\mathbf{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ to a size specified by a vector $\mathbf{L} = \{L_1, L_2, \ldots, L_M\}$ with $\prod_{m=1}^{M} I_m = \prod_{n=1}^{N} I_n$ returns an order-$M$ tensor $\mathbf{X}$, such that $\text{vec}(\mathbf{Y}) = \text{vec}(\mathbf{X})$, and is expressed as

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

Reshape does not permute entries in its vectorization.

Definition 2.2 (Tensor unfolding [35], [36], [40]):

Unfolding a tensor $\mathbf{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ along modes $r = [r_1, r_2, \ldots, r_M]$ and $c = [c_1, c_2, \ldots, c_N-M]$ where $[r,c]$ is a permutation of $[1,2,\ldots,N]$ aims to rearrange this tensor to be a matrix $\mathbf{Y}_{rc}$ of size

$$\prod_{k=1}^{M} I_{r_k} \times \prod_{l=1}^{N-M} I_{c_l},$$

whose entries $(j_1, j_2)$ are given by $\mathbf{Y}_{rc}(j_1, j_2) = y(i)$, where $i = [i_1, \ldots, i_N]$, $I_r = \{i_1, \ldots, i_{N-M}\}$, $I_c = \{i_{N-M+1}, \ldots, i_N\}$.

Remark 2.1:

1) If $c = [c_1 < c_2 < \cdots < c_{N-M}]$, $\mathbf{Y}_{rc}$ is simplified to $\mathbf{Y}_{r}$. 
2) If $r = n$ and $c = [1,2,\ldots,n-1,n+1,\ldots,N]$, we have mode-$n$ matricization $\mathbf{Y}_{rc} = \mathbf{Y}_{r}$.  
3) $\mathbf{Y}_{rc} = \mathbf{Y}_{r}^T_{cr}$. 
4) For $r = [1,2,\ldots,n]$, $c = [n+1,n+2,\ldots,N]$, for all $n$, $\mathbf{Y}_{rc} = \mathbf{Y}_{r} = \mathbf{Y}_{(n)} \in \mathbb{R}^{I_{(n)} \times I_{(n)}}$ can be expressed and efficiently performed by reshape, that is

$$\mathbf{Y}_{r} \triangleq \text{reshape}(\mathbf{Y}_{[I_{(n)}, I_{(n)}]}).$$

Definition 2.3 (mode-$n$ tensor-vector product [39]): The mode-$n$ multiplication of a tensor $\mathbf{Y} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ by a vector $\mathbf{a} \in \mathbb{R}^{I_n}$ returns an order-($N-1$) tensor $\mathbf{Z}$ defined as

$$\text{vec}(\mathbf{Z}) = \mathbf{Y}_{r}^{T} \mathbf{a}.$$ 

Symbolically, the product is denoted by

$$\mathbf{Z} = \mathbf{Y} \bar{\otimes} \mathbf{a} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_{n-1} \times I_{n+1} \times \cdots \times I_N}.$$  

Tensor-vector product of a tensor $\mathbf{Y}$ with a set of $N$ column vectors $\{\mathbf{a}^1, \mathbf{a}^2, \ldots, \mathbf{a}^N\}$ is denoted by

$$\mathbf{Y} \bar{\otimes} \{\mathbf{a}\} = \mathbf{Y} \bar{\otimes} \mathbf{a}^1 \odot \mathbf{a}^2 \odot \cdots \odot \mathbf{a}^N = \hat{\mathbf{Y}}.$$ 

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

$$\mathbf{Y} \approx \sum_{r=1}^{R} \mathbf{a}^{(1)} \odot \mathbf{a}^{(2)} \odot \cdots \odot \mathbf{a}^{(N)} = \hat{\mathbf{Y}}.$$  

\footnote{See the “reshape” Matlab subroutine.}
where symbol “⊙” denotes the outer product, component matrices (factors): $A^{(n)} = [a_1^{(n)}, a_2^{(n)}, \ldots, a_N^{(n)}] \in \mathbb{R}^{I_n \times K_n}$, $(n = 1, 2, \ldots, N)$ represent the common (loading) factors [41], [1], [2].

### A. Complexity of Tensor Unfolding

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 view point, 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.,

$$Y_{(1)} = \begin{bmatrix}
Y_{(1):1} & Y_{(1):2} & \cdots & Y_{(1):I_1}
\end{bmatrix},$$

$Y_{(1):i} = [y_{i,1}, y_{i,2}, \ldots, y_{i,I_1}]$. The mode-$n$ unfoldings $Y_{(n)}$ are defined as

$$Y_{(n)} = \left[ Y_{(1:J_{n-1})} Y_{(J_{n-1}+1:2J_{n-1})} \cdots Y_{((I_n-1)J_{n-1}+1:J_n)} \right].$$

By taking into account that $Y_{(N)} = Y_{(1:N-1)^T}$ in practice, we compute $Y_{(1:N-1)}$ instead of $Y_{(N)}$. $Y_{(1:N-1)}$ consists of $I_N$ vectors each of which comprises $I_{N-1}$ contiguous entries given by

$$Y_{(1:N-1)} = \left[ Y_{(1:J_{N-1})} Y_{(J_{N-1}+1:2J_{N-1})} \cdots Y_{((I_N-1)J_{N-1}+1:J_N)} \right].$$

In general, unfoldings $Y_{(n)}$ $(n = 1, 2, \ldots, N)$ do not change the order of entries of $Y$.

**B. Gradients in CP Algorithms**

We consider the cost function

$$D = \frac{1}{2} \| \mathbf{Y} - \overline{\mathbf{Y}} \|^2_F,$$

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

$$G^{(n)} = \mathbf{E}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)} \right) = \mathbf{Y}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)^T} \right) A^{(n)} \in \mathbb{R}^{I_n \times K_n},$$

where $\mathbf{E}_{(n)}$ denote the mode-$n$ unfoldings of the error tensor $E = Y - \overline{Y}$. The products $\mathbf{E}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)} \right)$ or $\mathbf{Y}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)^T} \right)$ have a computational cost of $O(RJ_N)$ (see details in Table I), and are the most expensive steps in CP algorithms. Indeed, the mode-$n$ unfoldings $Y_{(n)}$, for $n > 1$, are time-consuming. The latter products $\mathbf{Y}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)} \right)$ are more efficient than $\mathbf{E}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)} \right)$ in the sense of computation because we don’t need to construct the error tensor $\overline{E}$. However, since both products involve the same mathematical expression, we also call $\mathbf{Y}_{(n)} \left( \bigotimes_{k \neq n} A^{(k)} \right)$ the CP gradient in which $\mathbf{Y}$ is considered as an error tensor. The product is referred to as MTTKR 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[2], [1], [19], [5], [42] alternatively minimizes the cost function (7) with an update rule given by

$$A^{(n)} \leftarrow Y_{(n)} \left( \bigotimes_{k \neq n} A^{(k)} \right) \left( \bigotimes_{k \neq n} A^{(k)^T} \right) \left( A^{(n)^T} \right)^T, \quad (n = 1, 2, \ldots, N),$$

where $Q_n$ are commutation matrices of size $J_N \times J_N$, whose explicit expression is given by $Q_n = I_{K_n} \otimes P_{I_n \times I_n}$, where $P_{I_n \times I_n}$ is a permutation matrix for any $I \times J$ matrix $X$ such that vec($X$) = $P_{I_n \times I_n}$ 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-$n$ matricization which arranges $Y$ to be a matrix $Y_{(n)}$ of size $I_n \times (I_{n+1}I_{n+2} \cdots I_{n+K_n-1})$, since vec($Y_{(n)}^T$) = vec($Y_{(n+1)}$), one only needs a transposition and simple reshaping of the data entries [19], [21].
Algorithm 1: Direct Computation of $G(n)$ \( \bigotimes_{k \in \mathbb{K}} A(k) \) - MTTKRP \[33], \[34]  

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

**Output:** $G(n) = Y(n) \bigotimes_{k \in \mathbb{K}} A(k) : I_n \times R$  

begin  
1. $Y \leftarrow \text{permute}(Y,[n,1:n-1,n+1:N])$  
2. $Y(n) \leftarrow \text{reshape}(Y,[I_n,J_{n-1}K_{n}])$  
3. $G(n) = Y(n) \bigotimes_{k \in \mathbb{K}} A(k)$  

end  

where “\( \bigotimes \)” denotes the pseudo-inverse. A fast implementation of ALS for 3-way tensor, the ALS3 algorithm \[21\] reduces the expensive computation of $Y(n) \bigotimes_{k \in \mathbb{K}} A(k)$. The method was later extended to higher order decomposition in the PLS toolbox \[37\]. See section V for the ALS3 algorithm.

The all-at-once algorithms such as OPT \[24\], PMF3, the damped Gauss-Newton (dGN) or fast Levenberg-Marquardt algorithms \[26\], \[21\], \[27\], \[28\], \[29\], \[43\], \[44\], the well-known multiplicative algorithm \[45\], \[38\] also compute gradients in their update rules.

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

### III. Fast Computation of CP Gradient

#### A. Order of Dimensions

The CP gradient $G(n) = \{g^{(n)}, g_2^{(n)}, \ldots, g_R^{(n)}\} \in \mathbb{R}_{I_k \times R}$ given by

\[
G(n) = Y(n) \bigotimes_{k \in \mathbb{K}} A(k)
\]

involves $R$ products

\[
g^{(n)} = Y(n) \bigotimes_{k \in \mathbb{K}} a^{(k)}_r
\]

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

\[
t \leftarrow a^{(2)} \otimes a^{(1)}, \quad t \leftarrow a^{(3)} \otimes t, \quad \ldots, \quad t \leftarrow a^{(n-1)} \otimes t,
\]

\[
t \leftarrow a^{(n)} \otimes t, \quad \ldots, \quad t \leftarrow a^{(N)} \otimes t,
\]

with a computational cost of $\sum_{k=2}^{n} J_k + \frac{1}{I_n} \sum_{k=1}^{N} J_k$. When $n = 1$, the cost is given by $\frac{1}{I_n} \sum_{k=3}^{N} J_k$.

For the least cost to compute $G^{(n)}$, the dimensions of $Y$ should be in ascending order, i.e., $I_1 \leq I_2 \leq \cdots \leq I_N$. Note that $t = \bigotimes_{k \in \mathbb{K}} a^{(k)}$ can be computed from left-to-right, i.e., first computing $t \leftarrow a^{(N)} \otimes a^{(N-1)}$, then $t \leftarrow t \otimes a^{(N-2)}$, $\ldots$. 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 $G^{(n)}$ in (10) requires the following number of multiplications

\[
M_{\text{Alg.1}}(n) = R \left( J_N + \sum_{k=2}^{n-1} J_k + \frac{1}{I_n} \sum_{k=1}^{N} J_k \right).
\]

#### B. Fast Gradient with Respect to A Specific Factor

The direct computation of $G^{(n)}$ in (10) involves the tensor unfolding $Y(n)$ which is relatively slow to obtain for $1 < n < N$, due to permutation of entries. We note that vectors $g^{(n)}_r (r = 1, 2, \ldots, R)$ can be expressed in an equivalent form consisting of tensor-vector products $Y(N) \tilde{x}^{-1} a^{(k)}_r$ and $Y(N) \tilde{x}^{-1} a^{(n-1)}_r$ on the left side and right side of $n$, that is

\[
Y(n) \bigotimes_{k \in \mathbb{K}} a^{(k)}_r = (Y(N) \tilde{x}^{-1} a^{(k)}_r) \tilde{x}^{-1} a^{(n-1)}_r, \quad \text{“left-to-right”, (14)}
\]

or

\[
Y(n) \bigotimes_{k \in \mathbb{K}} a^{(k)}_r = (Y(N) \tilde{x}^{-1} a^{(n-1)}_r) \tilde{x}^{-1} a^{(k)}_r, \quad \text{“right-to-left”. (15)}
\]

The outer tensor-vector products in (14) have been re-indexed since $Y(N) \tilde{x}^{-1} a^{(k)}_r$ is of order $(N-1) + 1$. The inner tensor-vector products $L^{(n)} = Y(N) \tilde{x}^{-1} a^{(k)}_r$ in (14) and $R^{(n)} = Y(N) \tilde{x}^{-1} a^{(n-1)}_r$ in (15) can be efficiently computed through $Y(n-1)$ and $Y(n)$

\[
L^{(n)} = \text{reshape} \left( Y_F(1:n-1) \bigotimes_{k \in \mathbb{K}} a^{(k)}_r, [I_1, \ldots, I_n] \right),
\]

\[
R^{(n)} = \text{reshape} \left( Y_F(n) \bigotimes_{k \in \mathbb{K}} a^{(k)}_r, [I_1, \ldots, I_n] \right),
\]

for $r = 1, 2, \ldots, R$. It means that the unfoldings $Y(n)$ which are time-consuming for $1 < n < N$ are avoided.

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

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

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

**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 $G^{(n)}$ for $n = 1, 2, \ldots, N$, either sequentially (in alternating algorithms \[2\], \[1\], \[19\], \[5\], \[42\], \[45\], \[46\]) or simultaneously (as in all-at-once algorithms \[24\], \[26\], \[21\], \[27\], \[28\], \[29\], linesearch \[20\], \[21\]). This section will present a fast method to compute the gradients recursively for $n = 1, \ldots, N$. 

---

\[ (12) \]
Note that
\[ L^{(r,n)} = \sum_{i=1}^{n-1} a_i^n = L^{(r,n-1)} \cdot a_i^{n-1}, \] (20)
or
\[ \text{vec}(L^{(r,n)}) = \mathbf{1}_{(r+1)}(a_i^{n-1}). \] (21)
Similarly,
\[ R^{(r,n)} = \sum_{k=n+1}^{N} a_i^k = R^{(r,n+1)} \cdot a_i^{n+1}, \] (22)
or
\[ \text{vec}(R^{(r,n)}) = \mathbf{R}_{(1n)}(a_i^{n+1}). \] (23)
By exploiting relations in (23) and (21), we can quickly derive \( R^{(r,n)} \) from \( R^{(r,n+1)} \), or \( L^{(r,n)} \) from \( L^{(r,n-1)} \) 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 \( G^{(n^*)} \) (or \( G^{(n^*+1)} \)) is first computed where
\[ n^* = \max(n; J_n \leq K_n-1). \]
Other gradients \( G^{(n)} \) are then sequentially computed in the following order \( n = n^* - 1, n^* - 2, \ldots, 1, n^* + 1, n^* + 2, \ldots, N \). Computation of \( G^{(n^*)} \) and \( G^{(n^*+1)} \) is the most expensive step. It approximately requires the following number of multiplications
\[ R \left( 2J_N + 2J_n + K_n + 2 \cdot \sum_{j=2}^{n^*} J_j + \frac{1}{n^*} \sum_{k=n^*+1}^{N} J_k \right) \approx R(2J_N + 2J_n + 2K_n) \] 
It is straightforward to verify that \( \frac{N}{n^*} \leq N < N \). For \( N = 3 \), the algorithm first computes \( G^{(2)} \), then \( G^{(3)} \) and \( G^{(4)} \), sequentially, with a total cost of \( R(2J_N + 2J_2) \). 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 \( Y_{(m)} \) (1 \( < n < N \)) which are time consuming. Therefore, 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 \( \left( a_i^{(N)} \otimes \cdots \otimes a_i^{n+2} \right) \) when computing \( g_r^{(n)} \) and \( g_r^{(n+1)} \) with \( n > n^* \) in the “left-to-right” strategy (14), and \( \left( a_i^{(n-2)} \otimes \cdots \otimes a_i^{(1)} \right) \) when computing \( g_r^{(n)} \) and \( g_r^{(n-1)} \) with \( n < n^* \) in the “right-to-left” strategy (15). The ALS algorithm which operates two modes at a time (ALSO2) in [36], [37] computes \( \left( A_i^{(3)} \otimes \cdots \otimes A_i^{(N)} \right) \) before computing \( G^{(1)} \) and \( G^{(2)} \) (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 \( \left( a_i^{(N)} \otimes \cdots \otimes a_i^{(n+2)} \right) \) for both \( g_r^{(n)} \) and \( g_r^{(n+1)} \), and \( \left( a_i^{(N)} \otimes \cdots \otimes a_i^{(n+4)} \right) \) for both \( g_r^{(n+2)} \) and \( g_r^{(n+3)} \).

Remark 3.2: The order of dimensions \( I_1 \leq I_2 \leq \cdots \leq I_N \) can be arranged to \( I_{p_1} \leq I_{p_2} \leq \cdots \leq I_{p_n} \) and \( I_{p_{n+1}} \leq \cdots \leq I_{p_N} \) 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 \( T_{n^*, N}(p^*) = \sum_{k=1}^{N} M_{ALS}(k) \) is minimum. Note that \( J_n \) and \( K_n \) in \( M_{ALS}(k) \) are replaced with \( J_{n^*}^{(p^*)} \) and \( K_{n^*}^{(p^*)} \) 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 \( p \).

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 \( T_{n^*, N}(p) = \sum_{n=1}^{N} M_{ALS}(n) \) are given by
\[ T_{2,4}(p) = R(2J_4 + 2J_2^{(p^*)} + 2K_2^{(p^*)} + \min(J_2^{(p^*)}, K_1^{(p^*)}) + \min(J_2^{(p^*)}, K_2^{(p^*)})) \]
\[ T_{3,4}(p) = R(2J_4 + 3J_2^{(p^*)} + 2J_3^{(p^*)} + \min(J_3^{(p^*)}, K_2^{(p^*)})). \]

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 \( I_4 < I_1 \), then \( n^* = 2 \) and \( p^* = [1, 4, 2, 3] \).
- Otherwise, \( n^* = 3 \) and \( p^* = [1, 2, 3, 4] \).

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

Remark 3.5: [1], [2], ..., [N] is the optimal order for \( T_{N-1,N}(p) \).
Proof: The remark is deduced from \( T_{N-1,N}([1:k+1:N,k]) \) for \( k = 1, \ldots, N-1 \).

We do not investigate details on the optimal \( n^* \) and \( p^* \) for higher order \( N \geq 5 \) because these parameters can be quickly found by a simple code to loop over all possible combinations which are not more than \( \sum_{n=1}^{N-1} \frac{N}{n} = 2^{N-1} - 1 + \frac{N}{2(N/2)} \) 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 with 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 \( G^{(n)}T^{(n)} \) where \( T_n = \bigotimes_{k=n} A^{(k)}T^{(k)} \). Evaluation of the cost function (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 \( \overline{Y} \) [33], [47]
\[ \|\overline{Y} - \overline{Y}^\ast\|_F^2 = \|\overline{Y} - \overline{Y}^\ast\|_F^2 - 2 \sum_{r=1}^{R} \sum_{n=1}^{N} a_i^{(n)} \overline{Y} \overline{Y}^\ast = \|\overline{Y} - \overline{Y}^\ast\|_F^2 + \left( \bigotimes_{n=1}^{N} A^{(N)} \right) 1_r - 2 \text{tr}(A^{(N)T} G^{(N)}), \]
(25)
Algorithm 2: Fast Computation of \( Y_{\alpha}^{(n)} (\bigotimes_{k=n}^{(k)} A_{k}) \) over all modes

Input: Data tensor \( Y \colon (I_1 \times I_2 \times \cdots \times I_N) \),
\( N \) factor matrices \( A_{k}^{(n)} \in \mathbb{R}^{I_{k} \times R_{k}} \)

Output: \( G^{(n)} = Y_{\alpha}^{(n)} (\bigotimes_{k=n}^{(k)} A_{k}) : I_{k} \times R, \quad n \in \{1, 2, \ldots, N\} \)

begin

1. \( n^* = \max(n : J_{n} \leq K_{n-1}) \) where \( J_{n} = I_{1} \times I_{2} \times \cdots \times I_{n} \), \( K_{n} = I_{n+1} \times \cdots \times I_{N} \)

for \( n = n^*, n^* - 1, \ldots, n^{*} + 1, n^{*} + 2, \ldots, N \) do

if (\( n = n^* \)) then

\[ R^{(n)}_{\alpha} = \text{reshape}[Y, [J_{n}, K_{n}]] \]
\[ G^{(n)} = \text{cp\_gradient}(R^{(n)}_{\alpha}, [A_{k}]) \]

else if (\( n \neq n^* + 1 \)) then

\[ I^{(n)}_{\alpha} = \text{reshape}[Y, [J_{n-1}, K_{n-1}]] \]
\[ G^{(n)} = \text{cp\_gradient}(I^{(n)}_{\alpha}, [A_{k}]) \]

else if (\( n < n^* \)) then

for \( r = 1, 2, \ldots, R \) do \( \% \) Compute \( R^{(n)}_{\alpha} \) as in (23)

\[ \text{vec}(R^{(n)}_{\alpha}) \leftarrow \text{reshape}(R^{(n)}_{\alpha+1}, [J_{n}, I_{n+1}])a_{r}^{(n+1)} \]
\[ G^{(n)} = \text{cp\_gradient}(R^{(n)}_{\alpha}, [A_{k}]) \]

else

for \( r = 1, 2, \ldots, R \) do \( \% \) Compute \( L^{(n)}_{\alpha} \) as in (21)

\[ \text{vec}(L^{(n)}_{\alpha}) \leftarrow \text{reshape}(L^{(n)}_{\alpha+1}, [J_{n-1}, K_{n-1}])a_{r}^{(n+1)} \]

\[ G^{(n)} = \text{cp\_gradient}(L^{(n)}_{\alpha}, [A_{k}]) \]
end for
end if
end for
end if

where \( I_{R} \) is a vector of ones, and \( \|Y\|_{F}^{2} \) is computed only once or can be neglected. Hence, when \( n^{*} < N - 1 \), the complexity per iteration of FastALS is given by

1. Computing \( G^{(n)} = \sum_{n=1}^{N} M_{\text{Alg}_{n}} \approx R(2J_{N} + 3J_{N^{*}} + 3K_{N^{*}}) \)
2. Computing \( \mathbf{\Gamma}_{n} \approx N(N - 2)R^{2} + R^{2}T, \quad T = \sum_{n=1}^{N} I_{n} \)
3. Computing \( G^{(n)}\mathbf{\Gamma}_{n}^{T} \approx NR^{3} + R^{2}T \)
4. Evaluating (25) \( \approx R^{2} + R_{N} \)

\[ \text{Total cost} \approx R(2J_{N} + 3J_{N^{*}} + 3K_{N^{*}} + R^{2}(2T + N^{2}) + NR^{3}) \]

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

\[ \sum_{n=1}^{N} M_{\text{Alg}_{n}}(n) \approx R \left( N \frac{1}{I_{n}} \right) \]

Concerning the memory consumption, in order to update the factor \( A^{(n*)} \), 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 \( A^{(n)}, \quad n = n^{*} + 1, \ldots, 3, 2 \), 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 \( \text{vec}(R^{(1,n^{*}+1)}) \ldots \text{vec}(R^{(R,n^{*}+1)}) \) of size \( (J_{n^{*}+1} \times R) \). Moreover, the FastALS algorithm computes only the left-side Khatri-Rao product of size \( (J_{n^{*}+1} \times R) \). Hence, FastALS requires a temporary \( O(R(J_{n^{*}+1} + J_{n^{*}})) \) 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 \( \text{vec}(L^{(1,n^{*}+1)}) \ldots \text{vec}(L^{(R,n^{*}+1)}) \) of size \( (K_{n} \times R) \). It requires an extra \( O(R(J_{n} + K_{n})) \) memory cells. Updating factors \( A^{(n)}, \quad n = n^{*} + 2, \ldots, N - 1 \), 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 \( Y_{\alpha}^{(n)} \) of size \( I_{n} \times J_{n-1} \times K_{n} \), the ordinary CP-ALS algorithm requires an extra \( O(RJ_{n-1}K_{n}) \) memory cells for the Khatri-Rao products.

\[ \delta_{ij} \approx \text{Kronecker delta.} \]
of size \((J_{n-1} K_n \times R)\), \((n = 1, 2, \ldots, N)\). Note that
\[
\begin{align*}
J_{n-1} K_{n+1} I_{n+1} &> (J_{n-1} + K_{n+1}) I_{n+1} \geq J_n + K_n, \\
J_n &< K_n, \quad n < n^*, \\
J_n &> K_n, \quad n > n^*.
\end{align*}
\]
Therefore, the FastALS algorithm requires much smaller extra temporary memory than CP\_ALS. The maximum extra space of FastALS is of \(O(R I_n + K_r)\) 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 [48], [38], [46], [45], [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 (ALS02), and is implemented as the subroutine alsstep in the commercial PLS toolbox [37]. The ALS02 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 \(Z^{(r,n)} = L^{(r,n)} \otimes \mathbb{X}^{(r,n)}_{K_r} a_r^{(r,n)} \in \mathbb{R}^{L_r \times I_{n+1}}\), ALS02 [36], [37] operates two modes \(n\) and \((n+1)\) at a time
\[
g^{(n)}_r = L^{(r,n)} a^{(n)}_r, \quad g^{(n+1)}_r = L^{(r,n)} a^{(n)}_r , \quad (r = 1, 2, \ldots, R) \tag{27}
\]
through the left-side projection as in (16) with
\[
L^{(r,n)} = L^{(r,n-2)} \mathbb{X}_1 a^{(n-2)}_r \mathbb{X}_2 a^{(n-1)}_r \in \mathbb{R}^{L_r \times I_{n+1} \times I_{n+2}}, \tag{28}
\]
where \(r = 1, 2, \ldots, R\), \(n = 3, 5, 7, \ldots\) and \(L^{(r,1)} = \mathbb{Y}\). The ALS02 algorithm first computes \(G^{(1)}\) when \(N\) is odd or computes \(G^{(1)}\) and \(G^{(2)}\) for even order \(N\).

For example, when \(N = 3\), ALS02 first computes \(G^{(1)} = Y^{(1)} (A^{(3)} \otimes A^{(2)})\) with a cost of \((RJ_N + RK_1)\), then computes \(Y^{(1)^T} A^{(1)}\) with a cost of \(RJ_N\), and \(G^{(2)}\) and \(G^{(3)}\) with a cost of \(2RK_1\). The total cost of ALS02 is of \(R(2J_N + 3K_1)\). A minimum cost is achieved when \(I_2\) and \(I_3\) are the two shortest dimensions. That is, ALS02 [36], [37] may need a tensor permutation so that \(I_1 \geq I_2 \geq I_3\).

For order-4 tensors, ALS02 sequentially updates factors \(A^{(2)}\) and \(A^{(1)}\), then \(A^{(3)}\) and \(A^{(4)}\), and requires a total cost of \(R(2J_N + 3J_3 + 3K_2)\). The optimal order of dimensions for ALS02 is \([I_1, I_3, I_2, I_4]\) which is similar to that for FastALS when \(n^* = 2\) in Remark 3.3. However, when \(n^* = 3\), e.g., \(I_1I_2 \leq I_4\), FastALS updates factors from right to left, and has a lower cost of \(R(2J_N + 3J_3 + 3J_2)\) than ALS02 (see decomposition of \(10 \times 10 \times 10 \times 4000\) dimensional tensors in Table III).

For higher order tensors, in general, the matrices \(Z^{(r,n)}\) cost \(R\left(K_{n-1} + \frac{1}{J_{n+1}} \sum_{k=0}^{N} J_k\right)\) multiplications, and \(L^{(r,n)}\) cost \(R(I_{n-1} + I_{n-2}J_{n-1})\) multiplications. Hence, for even orders \(N\) \((6, 8, \ldots)\), the largest workloads are involved in the computations of \(Z^{(r,1)}, L^{(r,3)}, Z^{(r,3)}, L^{(r,5)}, Z^{(r,5)}\) and \(L^{(r,7)}\) with a total cost of
\[
R\left(K_0 + \frac{1}{J_{k=0}^{N}} J_k + K_0 + I_1I_2 + K_2 + \frac{1}{J_{k=0}^{N}} J_k + K_2 + I_3I_4 + \cdots \right) \approx R(2N + 3K_1 + 3K_4). \tag{29}
\]

For odd orders \(N (5, 7, \ldots)\), the computation cost of ALSo2 [36],[37] is approximately of \(R(2N + 3K_1 + 3K_4)\). The 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 \(K_1 + K_3\) 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 [2], [1], [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
\[
\rho = \frac{\text{Execution time}_{\text{CPALS}}}{\text{Execution time}_{\text{FastALS}}}. \tag{30}
\]

The final results were averaged over at least 20 iterations \(\times\) 10 runs. Figure 1 shows how the speed-up ratio per itera-
TABLE III

COMPARISON OF EXECUTION TIMES (SECOND) PER ITERATION BETWEEN CP-ALS, FastALS and ALSo2[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.

<table>
<thead>
<tr>
<th>Tensor size</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>30×100×1000</td>
<td>1</td>
</tr>
<tr>
<td>0.115</td>
<td>0.01</td>
</tr>
<tr>
<td>0.009</td>
<td>0.052</td>
</tr>
<tr>
<td>0.374</td>
<td>0.397</td>
</tr>
<tr>
<td>0.668</td>
<td>1.095</td>
</tr>
<tr>
<td>1.017</td>
<td>1.653</td>
</tr>
<tr>
<td>18</td>
<td>4.4</td>
</tr>
<tr>
<td>1.6</td>
<td>1.235</td>
</tr>
<tr>
<td>0.282</td>
<td>0.328</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>5×10×500×1000</td>
<td>1</td>
</tr>
<tr>
<td>2.83</td>
<td>0.24</td>
</tr>
<tr>
<td>6.001</td>
<td>1.04</td>
</tr>
<tr>
<td>7.477</td>
<td>0.93</td>
</tr>
<tr>
<td>27.4</td>
<td>1.28</td>
</tr>
<tr>
<td>196</td>
<td>2.53</td>
</tr>
<tr>
<td>40</td>
<td>28</td>
</tr>
<tr>
<td>1×10×1000</td>
<td>20</td>
</tr>
<tr>
<td>0.07</td>
<td>0.217</td>
</tr>
<tr>
<td>0.25</td>
<td>0.29</td>
</tr>
<tr>
<td>0.52</td>
<td>0.73</td>
</tr>
<tr>
<td>20</td>
<td>30</td>
</tr>
<tr>
<td>4×10×4000</td>
<td>20</td>
</tr>
<tr>
<td>5.06</td>
<td>0.177</td>
</tr>
<tr>
<td>9.257</td>
<td>2.36</td>
</tr>
<tr>
<td>12.7</td>
<td>3.37</td>
</tr>
<tr>
<td>28.9</td>
<td>6.25</td>
</tr>
<tr>
<td>52.1</td>
<td>38.4</td>
</tr>
<tr>
<td>83</td>
<td>2.8</td>
</tr>
<tr>
<td>2×10×500×40</td>
<td>20</td>
</tr>
<tr>
<td>4.42</td>
<td>1.161</td>
</tr>
<tr>
<td>37.5</td>
<td>96.8</td>
</tr>
<tr>
<td>4.74</td>
<td>3.74</td>
</tr>
<tr>
<td>49</td>
<td>36</td>
</tr>
<tr>
<td>30×40×500×80</td>
<td>20</td>
</tr>
<tr>
<td>17.5</td>
<td>11.5</td>
</tr>
<tr>
<td>33.1</td>
<td>19.5</td>
</tr>
<tr>
<td>49.7</td>
<td>31.0</td>
</tr>
<tr>
<td>60.2</td>
<td>38.6</td>
</tr>
<tr>
<td>85.3</td>
<td>56.6</td>
</tr>
<tr>
<td>69</td>
<td>11</td>
</tr>
<tr>
<td>50×60×1000</td>
<td>50</td>
</tr>
<tr>
<td>2.56</td>
<td>0.42</td>
</tr>
<tr>
<td>6.3</td>
<td>0.59</td>
</tr>
<tr>
<td>10.1</td>
<td>0.68</td>
</tr>
<tr>
<td>12.3</td>
<td>0.99</td>
</tr>
<tr>
<td>18.2</td>
<td>18.4</td>
</tr>
</tbody>
</table>

(b) Experiment results on computer PC2.

<table>
<thead>
<tr>
<th>Tensor size</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>30×40×500×1000</td>
<td>1</td>
</tr>
<tr>
<td>17.5</td>
<td>11.5</td>
</tr>
<tr>
<td>33.1</td>
<td>19.5</td>
</tr>
<tr>
<td>49.7</td>
<td>31.0</td>
</tr>
<tr>
<td>60.2</td>
<td>38.6</td>
</tr>
<tr>
<td>85.3</td>
<td>56.6</td>
</tr>
<tr>
<td>69</td>
<td>11</td>
</tr>
<tr>
<td>50×60×1000</td>
<td>50</td>
</tr>
<tr>
<td>2.56</td>
<td>0.42</td>
</tr>
<tr>
<td>6.3</td>
<td>0.59</td>
</tr>
<tr>
<td>10.1</td>
<td>0.68</td>
</tr>
<tr>
<td>12.3</td>
<td>0.99</td>
</tr>
<tr>
<td>18.2</td>
<td>18.4</td>
</tr>
</tbody>
</table>

(1): Execution time of CP-ALS.
(2) and (4): Execution time of ALSo2[36, 37] without and with optimal tensor permutation.
(3): Execution time of FastALS.

Fig. 1. Speed-up ratios per iteration (in logarithmic scale) for the FastALS algorithm in comparison with the standard CP-ALS algorithm for factorizations of order-3 and order-4 tensors with various sizes $I_n = I$ for all $n$, and ranks $R$.

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 30×100×1000 and 5×10×500×1000, 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 FastALS. Table III shows the cases when ALSo2 and FastALS are identical, i.e., the speed ratio is one.

For order-4 tensors of size 10×10×10×10, since $n^* = 3$, FastALS updated factors in the order 3, 2, 1, 4. Table III shows execution times of ALSo2 [37] without and with optimal tensor permutation $p = [1, 4, 2, 3]$. The results indicates that FastALS was 2 times faster than ALSo2[37] on average when $R \in [20, 200]$. For order-5 tensors of size $I_n = 40$, FastALS was approximately 3-7 times faster than ALSo2[37] on average for $R \in [1, 40]$. However, when $R = 60$, ALSo2[37] became very time-consuming. It took 38.25 seconds per iteration on average, and allocated a total 3.6 GB of memory with a peak memory of 1.2 GB. FastALS allocated only a total 121.9 MB 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).
ALS and FastALS. Both algorithms used ALS was nearly impossible which might consume 1.82 GB of RAM. It means that PC1 had comprised I cells which in double precision format consumed only 12 MB to factorize these tensors with rank R. ALSo2 was faster than CP ALS, but more time consuming than FastALS. However, for higher ranks R ≥ 50, execution times of the ALSo2 dramatically increased because PC1 did not have sufficient memory. 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, ALSo2 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 Iν = 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 ALSo2/FastALS, the total allocated and peak memory requirements of algorithms are provided. ALSo2 was faster than CP_ALS, but more time consuming than FastALS.

Since n* = 5, 6 or 7, the products \( \bigotimes_{k=1}^{n^*-1} A^{(k)} \) and \( \bigotimes_{k=n^*+1}^{N} A^{(k)} \) 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 \( \bigotimes_{k=n^*}^{N} A^{(k)} \) comprised \( N^{N-1} \times R = 5^{12} \approx 24410^6 \) double-precision numbers which might consume 1.82 GB of RAM. It means that PC1 had insufficient memory for CP_ALS and ALSo2 for N ≥ 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-

<table>
<thead>
<tr>
<th>Execution time per iteration (seconds)</th>
<th>Ratio (2)</th>
<th>Allocated Memory (MB)</th>
<th>Peak Memory (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CP_ALS</td>
<td>FastALS</td>
<td>ALSo2</td>
<td>CP_ALS</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>5</td>
<td>0.018 (0.043)</td>
<td>0.12</td>
<td>1.9 (4.1)</td>
</tr>
<tr>
<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>0.021</td>
<td>0.38</td>
<td>8.7</td>
</tr>
<tr>
<td>5</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>0.089 (0.245)</td>
<td>4.39 (5.37)</td>
<td>17.5 (30.8)</td>
</tr>
<tr>
<td>20</td>
<td>0.098</td>
<td>10.1</td>
<td>43.9</td>
</tr>
<tr>
<td>5</td>
<td>0.534 (1)</td>
<td>3.04 (8.91)</td>
<td>59.4 (2257)</td>
</tr>
<tr>
<td>12</td>
<td>0.427 (1.71)</td>
<td>5.09 (40)</td>
<td>96.4</td>
</tr>
<tr>
<td>20</td>
<td>0.456</td>
<td>14.6</td>
<td>250</td>
</tr>
<tr>
<td>5</td>
<td>3.71</td>
<td>59.4</td>
<td>331</td>
</tr>
<tr>
<td>13</td>
<td>2.62</td>
<td>123</td>
<td>535</td>
</tr>
<tr>
<td>20</td>
<td>2.76</td>
<td>1533</td>
<td>2491</td>
</tr>
</tbody>
</table>

TABLE IV

Comparison of execution time per iteration and allocated memory per iteration between CP_ALS, ALSo2 in [37] and FastALS in factorizations of order-N tensors of size I1 = I2 = ... = IN = 5, N = 10, 11, 12, 13. The results were measured on a computer which had 2 × X5690@3.47 GHz CPUs and 96 GB RAM. Values in parentheses show execution times on PC1.

Fig. 2. Execution times (in seconds) of the FastALS algorithm and the standard CP_ALS algorithm in factorization of the order-4 ITPC tensor in Example 2 with various ranks R.

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 [50] 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 × 61 frequency bins × 64 channels × 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. Mørup 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 \( n^* = 2 \), ALSo2 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 $\varepsilon = \frac{\|y - \tilde{y}\|_F}{\|y\|_F}$ 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.

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 $R \geq 30$ components on PC2, FastALS completed the factorizations only in several seconds, and achieved a high speed-up ratio $\rho \approx 25$ - 36 times. The speed ratio on PC1 was around 10-15 times for $R \geq 20$ 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 $2 \times 3.47$ 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 longer 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 dataset 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 $f_0 = 6$ Hz, and the wavelet center frequency $f_c = 1$ Hz [51], giving an order-5 tensor with modes 2 dictionaries $\times 23$ frequency bins (8-30 Hz) $\times 50$ time frames $\times 62$ channels $\times 200$ trials.

The order-5 tensor was factorized by CP models with various ranks $R = 1, 2, \ldots, 80$. Algorithms were initialized with the same values, and stopped when their differences of successive relative errors $\varepsilon = \frac{\|y - \tilde{y}\|_F}{\|y\|_F}$ 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 $A^{(5)} \odot A^{(4)}$ comprising $200 \times 62 \times 20 = 248000$ entries in the double precision floating-point type. Both CP_ALS and ALSo2 without optimal tensor permutation computed the Khatri-Rao product $\left(\bigotimes_{n=2}^{5} A^{(n)}\right)$ which consumed $23 \times 50 \times 62 \times 200 \times 20 \times 8 \approx 2.12$ 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 only comprised $62 \times 50 \times 23 \times 2 \times 20$ entries, and used 21.8 MB RAM approximately. Figure 3 indicates that FastALS was approximately 8 times faster than ALSo2 [36],[37] with $p = [5, 4, 3, 2, 1]$ when $R \to 80$. The FastALS with optimal tensor permutation $p = [1, 3, 4, 2, 5]$ was slightly faster than FastALS with dimensions in the ascending order for high $R$.

Running on PC2 with a large amount of memory, the CP_ALS took at least 40 minutes to several hours to complete the factorizations, while the FastALS algorithm quickly returned the factors after 1-2 minutes. The big difference in execution times reveals the substantial advantage of the proposed algorithm. The speed up ratio in comparison to CP_ALS was around 100-130 times on PC2, and 300-400 times on PC1 as increasing $R$ to 80. A classification study of MI movements was performed for the same order-5 tensors in [51], [52]. Acceleration of speed in BCI is a key factor because BCI needs to work on-line.

**VII. Conclusions**

The fast computation of one mode and all mode CP gradients (MTPKRP) has been introduced together with the fast ALS algorithm (FastALS) for the CP decomposition. The proposed method can efficiently and straightforwardly accelerate other alternating algorithms fitting CP in the similar
Hence, the total number of multiplications is given as in (19). For order-4 tensor when optimal tensor permutation is equivalent to the FastALS up to 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/~phal/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 \( \mathbf{g}_r^{(n)} \) from mode-\( n \) unfolding \( \mathbf{R}_{(r)}^{(n)} \) of \( \mathbf{R}^{(n)} \).

\[
\mathbf{g}_r^{(n)} = \mathbf{R}_{(r)}^{(n)} \left( \bigotimes_{k=1}^{n} a_r^{(k)} \right) \quad (31)
\]

The two steps require the number of multiplications

\[
M_{\text{BL1}} = R \left( J_N + \frac{1}{J_n} \sum_{k=2}^{N} J_k \right), \quad M_{\text{BL2}} = R \left( J_n + \sum_{k=2}^{n-1} J_k \right).
\]

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 \( I_n \times I_{n+1} \times \cdots \times I_N \) defined in (16), and compute \( \mathbf{g}_r \) from mode-1 unfolding \( \mathbf{L}_{(r)}^{(n)} \) of \( \mathbf{L}^{(n)} \).

\[
\mathbf{g}_r = \mathbf{L}_{(r)}^{(n)} \left( \bigotimes_{k=1}^{n} a_r^{(k)} \right) \quad (32)
\]

The two steps respectively require the following number of multiplications

\[
M_{\text{LR1}} = R \left( J_N + \sum_{k=2}^{n-1} J_k \right), \quad M_{\text{LR2}} = R \left( K_{n-1} + \frac{1}{J_n} \sum_{k=2}^{n} J_k \right).
\]

APPENDIX B

PROOF OF REMARK 3.1

Since \( I_n \leq I_{n+1} \) and \( I_n \leq J_n \), we easily have

\[
M_{\text{BL1}}(n) = R \left( J_N + \frac{1}{J_n} \sum_{k=2}^{N} J_k + \sum_{k=2}^{n} J_k \right) \leq R \left( J_N + \frac{1}{J_n} \sum_{k=2}^{N} J_k + J_{n-1}I_{n+1} + \sum_{k=2}^{n-1} J_k \right) = R \left( J_N + \frac{1}{I_{n+1}} \sum_{k=2}^{n} J_k + \sum_{k=2}^{n} J_k \right) = M_{\text{Alg. 1}}(n),
\]

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

ACKNOWLEDGMENTS

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


Anh-Huy Phan received the master degree from Hochiminh University of Technology, Vietnam in 2005, and the Ph.D degree from the Kyushu Institute of Technology, Japan in 2011. He worked as Deputy head of Research and Development Department, Broadcast Research and Application Center, Vietnam Television, and is currently Research Scientist at the Laboratory for Advanced Brain Signal Processing, and a visiting research scientist in TOYOTA Collaboration Center, Brain Science Institute, RIKEN. His research interests include multilinear algebra, tensor computation, blind source separation and brain computer interface.

Petr Tichavský (M’98, SM’04) received the M.S. degree in mathematics in 1987 from the Czech Technical University, Prague, Czechoslovakia and the Ph.D. degree in theoretical cybernetics from the Czechoslovak Academy of Sciences in 1992. Since that time he is with the Institute of Information Theory and Automation, Academy of Sciences of the Czech Republic in Prague. In 1994 he received the Fulbright grant for a 10 month fellowship at Yale University, Department of Electrical Engineering, in New Haven, U.S.A. In 2002 he received the Otto Wittcherle Award from Academy of Sciences of the Czech Republic.

He is author and co-author of research papers in the area of sinusoidal frequency/frequency-rate estimation, adaptive filtering and tracking of time varying signal parameters, algorithm-independent bounds on achievable performances, sensor array processing, independent component analysis and blind source separation.

Petr Tichavský served as associate editor of the IEEE Signal Processing Letters from 2002 to 2004, and as associate editor of the IEEE Transactions on Signal Processing from 2005 to 2009 and from 2011 to now. Petr Tichavský has also served as a general co-chair of the 36th IEEE Int. Conference on Acoustics, Speech and Signal Processing ICASSP 2011 in Prague, Czech Republic.
Andrzej Cichocki received the M.Sc. (with honors), Ph.D. and Dr.Sc. (Habilitation) degrees, all in electrical engineering from Warsaw University of Technology (Poland). He spent several years at University Erlangen (Germany) as an Alexander-von-Humboldt Research Fellow and Guest Professor. In 1995-1997 he was a team leader of the Laboratory for Artificial Brain Systems, at Frontier Research Program RIKEN (Japan), in the Brain Information Processing Group. He is currently a Senior Team Leader and Head of the laboratory for Advanced Brain Signal Processing, at RIKEN Brain Science Institute (JAPAN), and Professor in IBS, PAN (POLAND).

He has served as an Associated Editor of IEEE Transactions on Neural Networks, IEEE Transactions on Signals Processing, Journal of Neuroscience Methods and as founding Editor in Chief for Journal Computational Intelligence and Neuroscience. He is (co)author of more than 400 technical journal papers and 4 monographs in English (two of them translated to Chinese). His publications currently report over 19,000 citations according to Google Scholar, with an h-index of 59. His researches focus on tensor decompositions, brain machine interface, human robot interactions, EEG hyper-scanning, brain to brain interface and their practical applications.