NUMERICAL STABILITY OF TREE TENSOR NETWORK OPERATIONS, AND A STABLE ROUNDING ALGORITHM*

MARC BABOULIN†, OGUZ KAYA‡, THEO MARY§, AND MATTHIEU ROBEYNS†,¶

Abstract. We propose a general framework for analyzing the stability of computations on tree tensor networks in finite precision arithmetic. We identify a key property that is fundamental for stability: the norm of the network must be tightly concentrated in a single node, such as is the case when all nodes except one are semi-orthogonal. We provide a detailed error analysis proving that computations on such tensor networks are stable. This establishes the stability of some commonly used tensor algorithms. We also investigate tensor rounding, which is the problem of finding the optimal dimensions of a given tensor network topology. The most widely used rounding algorithm based on Gram SVD is unstable. We propose a rounding algorithm that, in view of our error analysis, is careful to maintain the semi-orthogonality of all nodes except one throughout the computation; it is therefore guaranteed stable. We experimentally confirm the stability of this algorithm and compare it to Gram SVD, showing that it is much more resilient to the use of low precision arithmetics, and is thus better suited to take advantage of all their performance benefits on modern hardware.

Key words. Tensor computations, tree tensor networks, low-rank approximations, finite precision arithmetic, low precision, numerical stability, error analysis, Gram SVD, orthonormalization

AMS subject classifications. 15A69, 65G50, 65F55, 65F25

1. Introduction.

Tensors, the higher dimensional generalization of matrices, are widely used in fields such as artificial intelligence, signal processing, and quantum computing to easily process large datasets. Unfortunately, the size of tensors is exponential in their number of dimensions (their order), which makes it impossible to store them exactly. One solution to tackle this "curse of dimensionality" is to reduce the order of the tensor via tensor low-rank approximation methods [11, 6, 16, 7]. A key operation in this context is tensor rounding [5, 1], which is the problem of finding the optimal ranks for a prescribed tensor topology and a prescribed accuracy.

This paper is concerned with tensor computations in finite precision arithmetic. While the stability of approximate tensor computations subject to low-rank truncation errors is well understood in exact arithmetic, it remains a completely open problem when a finite precision is used. This lack of analysis can be explained by mainly two reasons: first, the truncation errors tend to dominate the rounding errors when high precision is used, so that the latter have been traditionally neglected; and second, there is a wide range of different tensor formats, and their associated algorithms can be very complex to analyze. Importantly, the first reason is becoming less and less valid due to the growing prevalence of low precision arithmetics, which offer significant performance benefits on modern hardware. And the second reason makes it even more critical to develop a rounding error analysis precisely to identify which tensor algorithms are stable and which will become problematic in low precision.

Our contribution towards tackling these questions is two-fold. First, we propose a general framework based on a tree tensor network format that encompasses the most important tensor formats (Tensor-Train, Tucker, and hierarchical Tucker) [4] and we define the essential operations

*Version of March 18, 2025.

†Université Paris-Saclay, CNRS, ENS Paris-Saclay, LMF, Gif-sur-Yvette, France, (baboulin@lmf.cnrs.fr)

‡University Paris-Saclay, CNRS, LISN, Gif-sur-Yvette, France, (oguz.kaya, matthieu.robeyns@universite-paris-saclay.fr)

§Sorbonne Université, CNRS, LIP6, Paris, France, (theo.mary@lip6.fr)

¶Corresponding author.

that are needed to express some of the most common computations of interest. We then carry out an error analysis of an abstract computation in this framework and identify conditions to guarantee its stability. In particular, a key condition is that the norm of the tensor should be tightly concentrated around a single node of the network, a property that we will formalize in Definition 2.2 of our framework. This property is notably satisfied when all nodes of the network except one are semi-orthogonal. Our analysis shows that if this property is maintained throughout the computation then the error introduced by each operation is controlled in terms of the norm of the global tensor. This first contribution can thus be used to establish the stability of a wide range of tensor computations that can be expressed in our framework.

Our second contribution is to use our framework to propose a general rounding algorithm. The algorithm is designed based on the conclusions of our error analysis: throughout the computation, it is careful to maintain the semi-orthogonality of all nodes except one and is thus guaranteed to be stable. The algorithm works for any tree topology of tensor and can thus be applied to a wide range of tensor formats. When applied to specific tree topologies, it can be shown to be equivalent to existing algorithms such as TTSVD [16] or HODSVD [17], which we have thus proven to be stable in finite precision arithmetic. Our algorithm also shares similarities with existing algorithms for general topologies [12], [8]. We compare this stable rounding algorithm with the Gram SVD-based rounding, a popular choice for hierarchical Tucker [5] and tensor train [1] topologies, despite being unstable [14]. We show that our stable algorithm can significantly improve the accuracy of the rounding in finite precision arithmetic, and can thus more reliably exploit the low precision arithmetics available on modern hardware.

The rest of this paper is structured as follows. In Section 2 we present our general framework for tensor computations. Then, in Section 3, we carry out the error analysis of tensor computations based on our framework and identify conditions to guarantee their stability. We then exploit our framework to establish the stability of some common tensor computations in Section 4. We propose a tensor rounding algorithm in Section 5 that is guaranteed to be stable. In Section 6, we perform some numerical experiments that illustrate the stability of our stable rounding algorithm and compare it to the unstable Gram SVD-based one. Finally, we provide our concluding remarks in Section 7.

2. The framework.

In this section we present the framework that we will use to develop our analysis and algorithms. First, in Subsection 2.1, we introduce some basic definitions and notations. Then, in Subsection 2.2, we define the four key kernels that operate locally on a tree tensor network. Finally, in Subsection 2.3, we define a key property of the network that our analysis will require, and in Subsection 2.4, we give practical examples of networks where this property is satisfied.

2.1. Definitions and notations.

Our framework is based on tree tensor networks [15, 3, 4], illustrated in Figure 2.1. A tensor network $\boldsymbol{\mathcal{X}}$ is a tree, a connected acyclic undirected graph. The nodes represent tensors and the edges represent the dimensions of the tensor; thus the degree of a node is the order of the associated tensor. There are two types of edges: inner edges that connect two nodes, and outer (or "dangling") edges that are only connected to one node. Two nodes connected by an inner edge represent the contraction of the corresponding tensors along the dimension that connects them. Therefore, the whole network represents a full tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \ldots \times n_d}$ of order d, the number of dangling edges of $\boldsymbol{\mathcal{X}}$.

In the following, to clearly distinguish the objects under consideration, we will use the following notation.

- scalars are denoted with lower case letters (in particular, we will denote the inner edges as

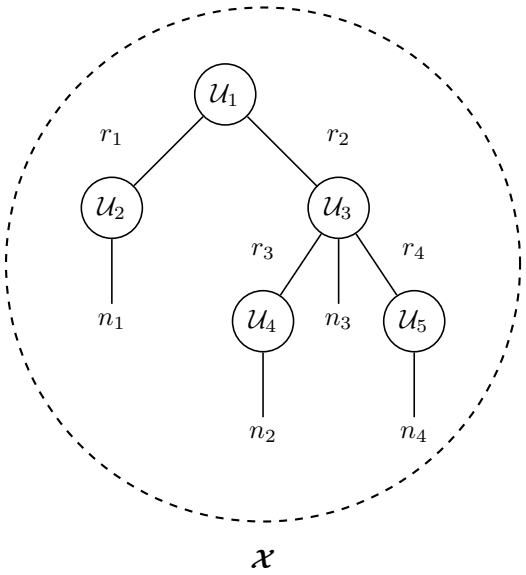


Fig. 2.1: Tree tensor network representation

r_i and the outer edges as n_i);

- matrices are denoted with upper case letters (e.g., X, U);
- tensors (nodes of a tensor network) are denoted with calligraphic upper case letters (e.g., $\mathcal{X}, \mathcal{U}$);
- and tensor networks are denoted with bold calligraphic upper case letters (e.g., $\boldsymbol{\mathcal{X}}$).

Throughout the article, the unsubscripted norm $\|\cdot\|$ denotes the Frobenius norm, whose standard definition for matrices extends to higher order tensors: $\|\mathcal{X}\|^2$ is the sum of the squares of all elements of $\mathcal{X}$. The Frobenius norm is invariant under unitary transformations, that is, $\|UA\| = \|AV\| = \|A\|$ for any matrix $A \in \mathbb{R}^{m\times n}$ and any unitary matrices $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{n\times n}$. This property is partially retained for semi-orthogonal matrices, that is, rectangular $p \times q$ matrices which have either orthonormal rows (when $p < q$) or orthonormal columns (when $p > q$). Specifically, for any $U \in \mathbb{R}^{p\times m}$ with orthonormal columns and any $V \in \mathbb{R}^{n\times q}$ with orthonormal rows, we have

$$\|UA\| = \|AV\| = \|A\|. \tag{2.1}$$

Given two tensor networks $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, we define the network $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}}\boldsymbol{\mathcal{B}}$ as the contraction of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ along one or multiple specified dimensions (which are left implicit as they will be apparent from the context).

We define $\|\boldsymbol{\mathcal{A}}\| = \|\mathcal{A}\|$, that is, the norm of the network is the norm of the full tensor it represents. As a result the usual submultiplicativity of the Frobenius norm is preserved for tensor networks:

$$\|\boldsymbol{\mathcal{A}}\boldsymbol{\mathcal{B}}\| = \|\mathcal{A}\mathcal{B}\| \le \|\mathcal{A}\|\|\mathcal{B}\| = \|\boldsymbol{\mathcal{A}}\|\|\boldsymbol{\mathcal{B}}\|.$$

Moreover, we extend the definition of semi-orthogonality to tensors as follows: given two adjacent nodes $\mathcal{A}$ and $\mathcal{B}$, we call $\mathcal{A}$ semi-orthogonal *in the direction of* $\mathcal{B}$ if the matricization A of $\mathcal{A}$

in the direction of $\mathcal{B}$ is a matrix with orthonormal columns (when considering the contraction $\mathcal{AB}$) or with orthonormal rows (when considering the contraction $\mathcal{BA}$). The invariance of the Frobenius norm is preserved with this extended definition: if $\mathcal{A}$ is semi-orthogonal in the direction of $\mathcal{B}$ then

$$\|\mathcal{AB}\| = \|\mathcal{B}\|.$$

Finally, we define $\boldsymbol{\mathcal{C}} = \boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}}$ as the network that represents the tensor $\mathcal{C} = \mathcal{A} + \mathcal{B}$; note that its nodes are *not* the sum of the nodes of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$, but rather their diagonal concatenation along the dimensions corresponding to inner edges. Concretely, if $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ are two compatible networks having an identical tree topology with the same outer edge dimensions $(n_1, \ldots, n_d)$, but potentially different inner dimensions ($r_1, \ldots, r_e$ for $\boldsymbol{\mathcal{A}}$ and $q_1, \ldots, q_e$ for $\boldsymbol{\mathcal{B}}$), then the resulting network $\boldsymbol{\mathcal{C}}$ also comprises the same tree topology and outer edge dimensions, but its inner dimensions become the sum of corresponding inner dimensions of $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ (that is, $p_i = r_i + q_i$). After this operation, two corresponding nodes $\mathcal{D} \in \mathbb{R}^{n_{i_1} \times \cdots \times n_{i_k} \times r_{j_1} \times \cdots \times r_{j_\ell}}$ of $\boldsymbol{\mathcal{A}}$ and $\mathcal{E} \in \mathbb{R}^{n_{i_1} \times \cdots \times n_{i_k} \times q_{j_1} \times \cdots \times q_{j_\ell}}$ of $\boldsymbol{\mathcal{B}}$, both with ℓ inner and k identical outer edges, get diagonally concatenated along the inner edges to yield the node $\mathcal{F} \in \mathbb{R}^{n_{i_1} \times \cdots \times n_{i_k} \times p_{j_1} \times \cdots \times p_{j_\ell}}$ of $\boldsymbol{\mathcal{C}}$, with $p_{j_t} = r_{j_t} + q_{j_t}$ and elements

$$\mathcal{F}(x_{i_1}, \ldots, x_{i_k}, y_{j_1}, \ldots, y_{j_\ell}) = \begin{cases} \mathcal{D}(x_{i_1}, \ldots, x_{i_k}, y_{j_1}, \ldots, y_{j_\ell}), & \text{if } y_{j_t} \le r_{j_t}, t = 1\colon \ell, \\ \mathcal{E}(x_{i_1}, \ldots, x_{i_k}, y_{j_1} - r_{j_1}, \ldots, y_{j_\ell} - r_{j_l}), & \text{if } y_{j_t} > r_{j_t}, t = 1\colon \ell, \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

An important case where this simplifies is when $\boldsymbol{\mathcal{A}}$ and $\boldsymbol{\mathcal{B}}$ only differ by one node: if $\boldsymbol{\mathcal{A}} = \boldsymbol{\mathcal{X}}_1 \mathcal{C} \boldsymbol{\mathcal{X}}_2$ and $\boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}}_1 \mathcal{D} \boldsymbol{\mathcal{X}}_2$, then

$$\boldsymbol{\mathcal{A}} + \boldsymbol{\mathcal{B}} = \boldsymbol{\mathcal{X}}_1 \mathcal{C} \boldsymbol{\mathcal{X}}_2 + \boldsymbol{\mathcal{X}}_1 \mathcal{D} \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1 (\mathcal{C} + \mathcal{D}) \boldsymbol{\mathcal{X}}_2.$$

2.2. Operations on tree tensor networks. In order to express various computations of interest in our framework, we define four key kernels that operate locally on a tree tensor network. These four kernels are Matricize, Tensorize, Split, and Merge.

- Matricize$(\mathcal{U}, r)$ matricizes the tensor $\mathcal{U}$ along the dimension r (which may be an inner or outer edge). This operation is illustrated in Figure 2.2, where $\mathcal{U}_3$ is a tensor of dimensions $r_1 \times r_2 \times r_3$ that is matricized along the dimension r_2. This results in matrix U_3 of dimensions $r_2 \times r_1 r_3$. To leave the representation of the entire network unchanged, we add a "phantom node" representing the identity tensor $\mathcal{I}$; note that this node is purely conceptual and never actually stored. In the case where r is an inner edge, we will also write Matricize$(\mathcal{U}, \mathcal{V})$, where $\mathcal{V}$ is the node connected to $\mathcal{U}$ by edge r.
- Tensorize(U) is the inverse operation of Matricize: it reshapes the matrix U into its tensor form $\mathcal{U}$ by contracting it with the phantom node $\mathcal{I}$ (note that this contraction is conceptual and does not require any operation). This operation is also illustrated in Figure 2.2 (going from right to left).
- Split(A) decomposes a matrix node A into the product of two matrices B and C. This operation is illustrated in Figure 2.3, where the node $U_2 \in \mathbb{R}^{n_1 \times r_1}$ is split into the product BC of $B \in \mathbb{R}^{n_1 \times r_0}$ and $C \in \mathbb{R}^{r_0 \times r_1}$.
- Merge$(\mathcal{B}, \mathcal{C})$ is the inverse operation of Split: given two tensors $\mathcal{B}$ and $\mathcal{C}$, it merges them into a single node $\mathcal{A}$ representing their contraction $\mathcal{BC}$ along the dimension that connects them. This operation is also illustrated in Figure 2.3 (going from right to left).

Note that since these kernels modify the network, formally we should pass $\boldsymbol{\mathcal{X}}$ as input/output, that is, we should for example write $\boldsymbol{\mathcal{X}} = \textsc{Matricize}(\mathcal{U}, r, \boldsymbol{\mathcal{X}})$. To keep the notation light, we instead consider these kernels to be member functions of the network, so that it is implicit that they modify the network.

The Merge kernel allows us to express contractions and, more specifically, matrix–matrix products when its input are matrix nodes. The Split kernel allows us to express various types of matrix decompositions, such as QR or truncated SVD decompositions. Together with Matricize and Tensorize, these four kernels are sufficient to express a wide range of tensor computations of interest. We will provide some examples in Section 4.

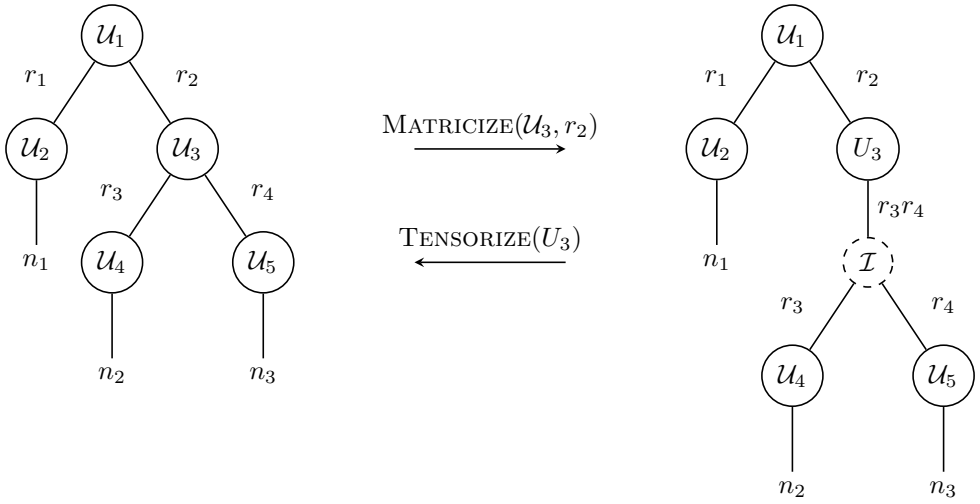


Fig. 2.2: Illustration of the Matricize and Tensorize kernels.

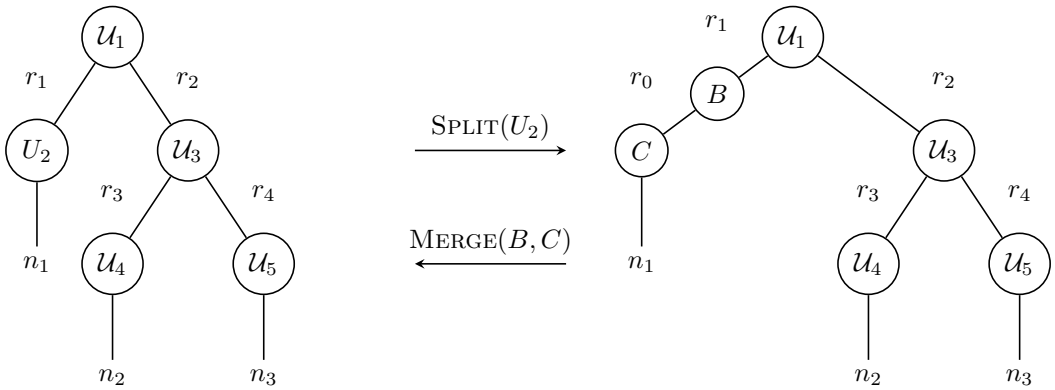


Fig. 2.3: Illustration of the Split and Merge kernels.

2.3. α-normalized networks. Bounding the error introduced by each of the kernels in terms of the local node to which they are applied is not sufficient to obtain an error bound in terms of the global network. To do so, we will need some assumptions on the network itself.

DEFINITION 2.1. *Given two nodes $\mathcal{U}_i, \mathcal{U}_j$ of a network $\boldsymbol{\mathcal{X}}$, $\mathcal{U}_i$ is said to be α_i-normalized in the direction of $\mathcal{U}_j$ if there exists a constant $\alpha_i > 0$ such that the contraction of $\mathcal{U}_i$ along the dimension that connects $\mathcal{U}_i$ with $\mathcal{U}_j$ satisfies, for any compatible tensor $\mathcal{M}$, $\|\mathcal{U}_i\mathcal{M}\| \leq \alpha_i\|\mathcal{M}\|$.*

DEFINITION 2.2. *Consider a network $\boldsymbol{\mathcal{X}}$ composed of n nodes $\mathcal{U}_1, \ldots, \mathcal{U}_n$. $\boldsymbol{\mathcal{X}}$ is said to be α-normalized with respect to a node $\mathcal{U}_j$ if there exists a vector of constants $\alpha = (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ such that $\|\mathcal{U}_j\| \leq \alpha_j\|\boldsymbol{\mathcal{X}}\|$ and all the other nodes $\mathcal{U}_i$, $i \neq j$, are α_i-normalized in the direction of $\mathcal{U}_j$.*

This abstract definition reflects the fact that in several contexts the norm of the network $\boldsymbol{\mathcal{X}}$ is tightly concentrated in a single node $\mathcal{U}_j$, while the remaining nodes satisfy some Lipschitz-like condition meaning that their contraction approximately preserves the norm. In particular, there are two important cases where the network $\boldsymbol{\mathcal{X}}$ is α-normalized with small α_i values, which we discuss in the next section (Subsection 2.4).

We can derive the following useful properties of normalized networks.

LEMMA 2.3. *Let the network $\boldsymbol{\mathcal{X}}$ with nodes $\mathcal{U}_1, \ldots, \mathcal{U}_n$ be α-normalized with respect to a node $\mathcal{U}_j$. Let the network $\boldsymbol{\mathcal{Y}}$ with nodes $\mathcal{V}_1, \ldots, \mathcal{V}_{n-1}$ be obtained by replacing a pair of adjacent nodes $\mathcal{U}_k$ and $\mathcal{U}_\ell$ (with $k < \ell$) by their contraction, so that the nodes of $\boldsymbol{\mathcal{Y}}$ are*

$$\mathcal{V}_1 \equiv \mathcal{U}_1, \ldots, \mathcal{V}_{k-1} \equiv \mathcal{U}_{k-1}, \mathcal{V}_k \equiv \mathcal{U}_k\mathcal{U}_\ell, \mathcal{V}_{k+1} \equiv \mathcal{U}_{k+1}, \ldots, \mathcal{V}_{\ell-1} \equiv \mathcal{U}_{\ell-1}, \mathcal{V}_\ell \equiv \mathcal{U}_{\ell+1}, \ldots, \mathcal{V}_{n-1} \equiv \mathcal{U}_n,$$

that is, the kth node is replaced by the contraction of the kth and ℓth nodes, and the ℓth node is removed. Then $\boldsymbol{\mathcal{Y}}$ is β-normalized with respect to $\mathcal{V}_m$, with $m = j$ if $j \neq \ell$ and $m = k$ if $j = \ell$, and with

$$\beta = [\alpha_1, \ldots, \alpha_{k-1}, \alpha_k\alpha_\ell, \alpha_{k+1}, \ldots \alpha_{\ell-1}, \alpha_{\ell+1}, \ldots, \alpha_n].$$

Proof. Since $\boldsymbol{\mathcal{X}}$ is a tree, either $\mathcal{U}_k$ is α_k-normalized in the direction of $\mathcal{U}_\ell$ or $\mathcal{U}_\ell$ is α_ℓ-normalized in the direction of $\mathcal{U}_k$. Let us assume the former without loss of generality. If $\ell \neq j$, for any compatible tensor $\mathcal{M}$, the contraction $\mathcal{V}_k\mathcal{M}$ along the dimension that connects $\mathcal{U}_\ell$ to $\mathcal{U}_j$ satisfies

$$\|\mathcal{V}_k\mathcal{M}\| = \|\mathcal{U}_k\mathcal{U}_\ell\mathcal{M}\| \leq \alpha_k\|\mathcal{U}_\ell\mathcal{M}\| \leq \alpha_k\alpha_\ell\|\mathcal{M}\|.$$

and so $\boldsymbol{\mathcal{Y}}$ is β-normalized with respect to $\mathcal{V}_m = \mathcal{V}_j$. In the special case $\ell = j$, the node $\mathcal{V}_k$ satisfies

$$\|\mathcal{V}_k\| = \|\mathcal{U}_k\mathcal{U}_j\| \leq \alpha_k\|\mathcal{U}_j\| \leq \alpha_k\alpha_j\|\boldsymbol{\mathcal{X}}\|$$

and so $\boldsymbol{\mathcal{Y}}$ is β-normalized with respect to $\mathcal{V}_m = \mathcal{V}_k$. ∎

LEMMA 2.4. *If a network $\boldsymbol{\mathcal{X}}$ with n nodes is α-normalized with respect to a node $\mathcal{U}_j$, then*

$$\|\boldsymbol{\mathcal{X}}\| \leq \prod_{\substack{i=1,\\ i\neq j}}^{n} \alpha_i\|\mathcal{U}_j\|. \tag{2.3}$$

Proof. This is a direct consequence of Lemma 2.3 by considering the full contraction of the entire network. ∎

LEMMA 2.5. *If a network $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to a node $\mathcal{U}_j$, then for any node $\mathcal{U}_i$, $i \neq j$, we have*

$$\|\mathcal{U}_i\| \leq \alpha_i \sqrt{r_i}, \tag{2.4}$$

where r_i is the dimension of $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$.

Proof. Let U_i be the matricization of $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$; U_i is a matrix with r_i columns. We have

$$\|\mathcal{U}_i\| = \|U_i\| = \|U_i I_{r_i}\| \leq \alpha_i \|I_{r_i}\| = \alpha_i \sqrt{r_i},$$

where I_{r_i} is the $r_i \times r_i$ identity matrix. ∎

2.4. e-normalized networks (and their sum). There are two important cases where the network $\boldsymbol{\mathcal{X}}$ is α-normalized with small α_i values.

The first case is when the matricization of any node $\mathcal{U}_i$ in the direction of $\mathcal{U}_j$ is semi-orthogonal. Indeed, in this case, $\|\boldsymbol{\mathcal{X}}\| = \|\mathcal{U}_j\|$ and the network is e-normalized where e is the vector of all ones, that is, $\alpha_i = 1$ for $i = 1\colon n$. Note that in this case, e-normality is equivalent to other definitions discussed in the literature, under the name of r-orthogonality [2, Definition 24] or t-frame [5, Definition 3.5].

The second case is when the network is obtained as the sum of two e-normalized networks. Indeed, in this case, we will prove that the nodes with at least two inner dimensions remain semi-orthogonal (1-normalized), whereas the nodes with only one inner dimension (which are the leaves) become $\sqrt{2}$-normalized.

Let us first analyze the case of nodes with at least two inner dimensions with the following result.

LEMMA 2.6. *Let $\mathcal{A} \in \mathbb{R}^{n_{i_1} \times \cdots \times n_{i_k} \times r_{j_1} \times \cdots \times r_{j_\ell}}$ and $\mathcal{B} \in \mathbb{R}^{n_{i_1} \times \cdots \times n_{i_k} \times q_{j_1} \times \cdots \times q_{j_\ell}}$ be nodes with ℓ inner and k identical outer edges, and let both be semi-orthogonal in the direction of r_{j_s} and q_{j_s}, respectively, for an inner edge index j_s. Then, if $\ell \geq 2$, their diagonal concatenation along the inner edges $\mathcal{C} \in \mathbb{R}^{n_{i_1} \times \cdots \times n_{i_k} \times p_{j_1} \times \cdots \times p_{j_\ell}}$, where $p_{j_t} = r_{j_t} + q_{j_t}$ for $t = 1\colon \ell$, is also semi-orthogonal in the direction of p_{j_s}.*

Proof. Without loss of generality, assume that $s = 1$ and let $\mathcal{M} \in \mathbb{R}^{p_{j_1} \times m_1 \times \cdots \times m_h}$ be an $(h+1)$-dimensional tensor for $h \geq 0$ whose first dimension is amenable to contraction with $\mathcal{C}$ along the dimension p_{j_1}. Consider the split of $\mathcal{M}$ along the first dimension $\mathcal{M} = [\mathcal{M}_1 | \mathcal{M}_2]$ with $\mathcal{M}_1 \in \mathbb{R}^{r_{j_1} \times m_1 \times \cdots \times m_h}$ and $\mathcal{M}_2 \in \mathbb{R}^{q_{j_1} \times m_1 \times \cdots \times m_h}$. Then, the result of the contraction $\mathcal{D} = \mathcal{C}\mathcal{M}$ along p_{j_1} has the following the elements according to (2.2):

$$\mathcal{D}(x_{i_1}, \ldots, x_{i_k}, y_{j_2}, \ldots, y_{j_\ell}, z_1, \ldots, z_h) = \begin{cases} \mathcal{A}(x_{i_1}, \ldots, x_{i_k}, :, y_{j_2}, \ldots, y_{j_\ell})^T \mathcal{M}_1(:, z_1, \ldots, z_h), & \text{if } y_{j_t} \leq r_{j_t}, t = 2\colon \ell, \\ \mathcal{B}(x_{i_1}, \ldots, x_{i_k}, :, y_{j_2}, \ldots, y_{j_\ell})^T \mathcal{M}_2(:, z_1, \ldots, z_h), & \text{if } y_{j_t} > r_{j_t}, t = 2\colon \ell, \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

By setting $\mathcal{D}_1 = \mathcal{A}\mathcal{M}_1$ and $\mathcal{D}_2 = \mathcal{B}\mathcal{M}_2$, we obtain $\mathcal{D}$ as the diagonal concatenation of $\mathcal{D}_1$ and $\mathcal{D}_2$

across dimensions $p_{j_2}, \ldots, p_{j_l}$ according to (2.2)–(2.5). This yields

$$\begin{aligned}\|\mathcal{C}\mathcal{M}\| = \|\mathcal{D}\| &= \sqrt{\|\mathcal{D}_1\| + \|\mathcal{D}_2\|} \\ &= \sqrt{\|\mathcal{A}\mathcal{M}_1\| + \|\mathcal{B}\mathcal{M}_2\|} \\ &= \sqrt{\|\mathcal{M}_1\| + \|\mathcal{M}_2\|} \\ &= \|\mathcal{M}\|. \end{aligned} \tag{2.6}$$

This concludes the proof because a tensor $\mathcal{C}$ whose contraction with *any* tensor $\mathcal{M}$ is norm-preserving must necessarily be semi-orthogonal. ∎

Note that for (2.6) to hold, it is imperative to have $\ell \geq 2$ to have the subtensors $\mathcal{D}_1$ and $\mathcal{D}_2$ in a block diagonal structure. If $\ell = 1$, this is no longer valid since $\mathcal{D}_1$ and $\mathcal{D}_2$ overlap and hence need to be added together, that is, $\mathcal{D} = \mathcal{D}_1 + \mathcal{D}_2$.

Let us thus now turn to the case of nodes with only one inner dimension. These nodes are the leaves of the network, and moreover they are matrices since they are of degree two: one inner dimension and one outer dimension (since any node can be reshaped to only have one outer dimension). These nodes become the concatenation of two semi-orthogonal matrices, which is $\sqrt{2}$-normalized, as the next result shows.

LEMMA 2.7. *Let $U_1 \in \mathbb{R}^{n \times r_1}$ and $U_2 \in \mathbb{R}^{n \times r_2}$ be semi-orthogonal nodes in the directions r_1 and r_2, respectively. Then, their concatenation $U = [U_1\ U_2] \in \mathbb{R}^{n \times r_1 + r_2}$ is $\sqrt{2}$-normalized in the direction $r_1 + r_2$.*

Proof. Let $M \in \mathbb{R}^{(r_1+r_2)\times m}$ be a matrix and let $M = \begin{bmatrix} M_1 \\ M_2 \end{bmatrix}$ be the partition of M corresponding to the partition of $U = [U_1\ U_2]$. We have

$$\|UM\| = \|U_1 M_1 + U_2 M_2\| \leq \|U_1 M_1\| + \|U_2 M_2\| = \|M_1\| + \|M_2\|.$$

Thus

$$\|UM\|^2 \leq (\|M_1\| + \|M_2\|)^2 = \|M_1\|^2 + \|M_2\|^2 + 2\|M_1\|\|M_2\| = \|M\|^2 + 2\|M_1\|\|M_2\|.$$

The term $\|M_1\|\|M_2\| = \|M_1\|\sqrt{\|M\|^2 - \|M_1\|^2}$ is maximized when $\|M_1\|^2 = \|M\|^2/2$, in which case it attains its maximum value

$$\frac{\|M\|}{\sqrt{2}}\sqrt{\frac{\|M\|^2}{2}} = \frac{\|M\|^2}{2}.$$

Hence overall

$$\|UM\|^2 \leq \|M\|^2 + 2\|M_1\|\|M_2\| \leq \|M\|^2 + 2\frac{\|M\|^2}{2} = 2\|M\|^2$$

which proves that $\|UM\| \leq \sqrt{2}\|M\|$. ∎

In conclusion, by combining Lemma 2.6 and Lemma 2.7, we obtain the following result on the sum of e-normalized networks.

THEOREM 2.8. *Let $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ be e-normalized networks, as defined in Definition 2.2, and where e is the vector of all ones. Then $\boldsymbol{\mathcal{X}}_1 + \boldsymbol{\mathcal{X}}_2$ is α-normalized, where $\alpha_i = \sqrt{2}$ for the leaf nodes and $\alpha_i = 1$ for the other nodes.*

3. Stability analysis. In this section, we are interested in analyzing the effect of a network $\boldsymbol{\mathcal{X}}$ undergoing a sequence of inexact operations in finite precision arithmetic. Denoting $\widehat{\boldsymbol{\mathcal{X}}}$ the resulting network after such a sequence of operations, we ask how different is $\widehat{\boldsymbol{\mathcal{X}}}$ from $\boldsymbol{\mathcal{X}}$? To answer this question, we must first define a measure of distance between tree tensor networks. The natural definition is to measure the norm of the difference of the tensors that these networks represent, that is:

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}.\textsc{Full}() - \widehat{\boldsymbol{\mathcal{X}}}.\textsc{Full}()\| = \|\mathcal{X} - \widehat{\mathcal{X}}\|. \tag{3.1}$$

In order to upper bound $\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\|$, we divide our analysis into three parts. First, in Subsection 3.1, we make some assumptions on the accuracy of the four kernels. Then, in Subsection 3.2, we analyze how a single local operation affects the accuracy of the global network. Finally, in Subsection 3.3, we show that the global error resulting from a sequence of such operations is bounded by the sum of the local errors and conclude our analysis.

3.1. Model. First, since the Matricize and Tensorize kernels only reshape data without performing any numerical computations, it is natural to assume that they do not incur any error. In other words, we have

$$\textsc{Tensorize}(\textsc{Matricize}(\mathcal{A}, r)) = \mathcal{A}.$$

Consider now three matrices A, B, and C such that $A = BC$. Let $\widehat{B}\widehat{C}$ be the decomposition computed by the $\textsc{Split}(A)$ kernel. We assume that it satisfies,

$$\widehat{B}\widehat{C} = \textsc{Split}(A) = A + E, \quad \|E\| \leq c\varepsilon\|A\|, \tag{3.2}$$

where c is a constant depending only on the dimensions of the matrices. This amounts to assuming that the decomposition computed by Split is backward stable with respect to some precision parameter ε. This assumption is certainly satisfied for standard matrix decompositions (QR, SVD, ...) [9] that are typically used in the computations of interest.

Finally, consider three tensors $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ such that $\mathcal{A} = \mathcal{B}\mathcal{C}$ where the product $\mathcal{B}\mathcal{C}$ corresponds to their contraction along a common dimension, and let $\widehat{\mathcal{A}}$ be the contraction computed by the $\textsc{Merge}(\mathcal{B}, \mathcal{C})$ kernel. We assume that it satisfies

$$\widehat{\mathcal{A}} = \textsc{Merge}(\mathcal{B}, \mathcal{C}) = \mathcal{B}\mathcal{C} + \mathcal{E}, \quad \|\mathcal{E}\| \leq c\varepsilon\|\mathcal{B}\|\|\mathcal{C}\|. \tag{3.3}$$

This assumption is certainly satisfied if A, B, and C are matrices and A is computed via a standard matrix–matrix product [9, (3.13)]. It is not hard to see that the assumption is also satisfied for tensors if the contraction is performed by matricizing $\mathcal{B}$ and $\mathcal{C}$ along their common dimension and computing their product via a standard matrix–matrix product.

3.2. Local errors. Let us now consider two networks $\boldsymbol{\mathcal{X}}$ and $\widehat{\boldsymbol{\mathcal{X}}}$, where $\widehat{\boldsymbol{\mathcal{X}}}$ is obtained by applying a single instance of one of the four kernels of our framework. Our goal is to bound $\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\|$. In fact, we only need to focus on the Split and Merge kernels since, as mentioned in the previous section, the Matricize and Tensorize kernels do not incur any error.

Let us first consider the case where $\widehat{\boldsymbol{\mathcal{X}}}$ is obtained by applying Split on a given node $\mathcal{U}_k \equiv A$ of $\boldsymbol{\mathcal{X}}$. We define $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ the partial networks corresponding to all nodes along either dimension of A, so that $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}_1 A \boldsymbol{\mathcal{X}}_2$. Then by (3.2) we have

$$\widehat{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{X}}_1 \widehat{B}\widehat{C}\boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1 (A + E)\boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2$$

and so

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2\| \leq \|\boldsymbol{\mathcal{X}}_1\|\|E\|\|\boldsymbol{\mathcal{X}}_2\| \leq c\varepsilon\|\boldsymbol{\mathcal{X}}_1\|\|A\|\|\boldsymbol{\mathcal{X}}_2\|. \tag{3.4}$$

This shows that even a single operation can lead to instability if $\|\boldsymbol{\mathcal{X}}_1\|\|A\|\|\boldsymbol{\mathcal{X}}_2\| \gg \|\boldsymbol{\mathcal{X}}\|$.

Stability can fortunately be guaranteed when the network is α-normalized as defined in Definition 2.2. Indeed, assume that $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to some node $\mathcal{U}_j$, and let n be the number of nodes of $\boldsymbol{\mathcal{X}}$.

If $j = k$ (that is, $\mathcal{U}_j = A$), then the network $\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2$ is β-normalized with respect to node E where $\beta_i = \alpha_i$ for all $i \neq j$ (the value of β_j is irrelevant). Therefore by Lemma 2.4 we obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2\| \leq \prod_{\substack{i=1 \\ i\neq j}}^{n} \alpha_i \|E\| \leq \prod_{\substack{i=1 \\ i\neq j}}^{n} \alpha_i c\varepsilon \|A\| \leq \prod_{i=1}^{n} \alpha_i c\varepsilon \|\boldsymbol{\mathcal{X}}\| \tag{3.5}$$

since $\|A\| \leq \alpha_j \|\boldsymbol{\mathcal{X}}\|$.

If $k \neq j$ (that is, $\mathcal{U}_j \neq A$), assume that $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_2$ (the case $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_1$ is analogous). Then the network $\boldsymbol{\mathcal{X}}_1 E$ is β-normalized with respect to node E with $\beta_i = \alpha_i$ for all nodes $i \in \boldsymbol{\mathcal{X}}_1$ (the value of β_k is irrelevant). Therefore by Lemma 2.4 we have

$$\|\boldsymbol{\mathcal{X}}_1 E\| \leq \prod_{i\in\boldsymbol{\mathcal{X}}_1} \alpha_i \|E\| \leq \prod_{i\in\boldsymbol{\mathcal{X}}_1} \alpha_i c\varepsilon \|A\| \leq \prod_{i\in\boldsymbol{\mathcal{X}}_1} \alpha_i \alpha_k \sqrt{r_k} c\varepsilon \tag{3.6}$$

where r_k is the dimension of $A = \mathcal{U}_k$ in the direction of $\mathcal{U}_j$ by Lemma 2.5. Moreover the network $\boldsymbol{\mathcal{X}}_2$ is $\bar{\alpha}$-normalized where $\bar{\alpha}$ is a subset of α restricted to the nodes in $\boldsymbol{\mathcal{X}}_2$, and therefore by Lemma 2.4

$$\|\boldsymbol{\mathcal{X}}_2\| \leq \prod_{\substack{i\in\boldsymbol{\mathcal{X}}_2, \\ i\neq j}} \alpha_i \|\mathcal{U}_j\| \leq \prod_{i\in\boldsymbol{\mathcal{X}}_2} \alpha_i \|\boldsymbol{\mathcal{X}}\|. \tag{3.7}$$

By combining (3.6) and (3.7) we finally obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 E \boldsymbol{\mathcal{X}}_2\| \leq \|\boldsymbol{\mathcal{X}}_1 E\|\|\boldsymbol{\mathcal{X}}_2\| \leq \prod_{i=1}^{n} \alpha_i \sqrt{r_k} c\varepsilon \|\boldsymbol{\mathcal{X}}\|. \tag{3.8}$$

For the MERGE kernel, the analysis is very similar. Let MERGE$(\mathcal{B}, \mathcal{C})$ be applied to $\boldsymbol{\mathcal{X}}$ where $\mathcal{B} \equiv \mathcal{U}_k$ and $\mathcal{C} \equiv \mathcal{U}_\ell$. Defining $\boldsymbol{\mathcal{X}}_1$ and $\boldsymbol{\mathcal{X}}_2$ such that $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}_1 \mathcal{B}\mathcal{C} \boldsymbol{\mathcal{X}}_2$, by (3.3) we have

$$\widehat{\boldsymbol{\mathcal{X}}} = \boldsymbol{\mathcal{X}}_1 \widehat{\mathcal{A}} \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}}_1 (\mathcal{B}\mathcal{C} + \mathcal{E}) \boldsymbol{\mathcal{X}}_2 = \boldsymbol{\mathcal{X}} + \boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2.$$

The goal is thus to bound $\|\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2\|$ assuming that $\boldsymbol{\mathcal{X}}$ is α-normalized with respect to some node $\mathcal{U}_j$.

If $j = \ell$ (that is, $\mathcal{U}_j = \mathcal{C}$) then $\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2$ is β-normalized with $\beta_i = \alpha_i$ for all $i \neq j, k$. By

Lemma 2.4 we have

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2\| \tag{3.9}$$

$$\leq \prod_{\substack{i=1 \\ i \neq j,k}}^{n} \alpha_i \|\mathcal{E}\| \tag{3.10}$$

$$\leq c\varepsilon \prod_{\substack{i=1 \\ i \neq j,k}}^{n} \alpha_i \|\mathcal{U}_k\| \|\mathcal{U}_j\| \tag{3.11}$$

$$\leq c\varepsilon \prod_{\substack{i=1 \\ i \neq k}}^{n} \alpha_i \|\mathcal{U}_k\| \|\boldsymbol{\mathcal{X}}\| \tag{3.12}$$

$$\leq c\varepsilon \sqrt{r_k} \prod_{i=1}^{n} \alpha_i \|\boldsymbol{\mathcal{X}}\| \tag{3.13}$$

since $\|\mathcal{U}_k\| \leq \alpha_k \sqrt{r_k}$ by Lemma 2.5.

If $j = k$ (that is, $\mathcal{U}_j = \mathcal{B}$) then we similarly have

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq \prod_{i=1}^{n} \alpha_i \sqrt{r_\ell} c\varepsilon \|\boldsymbol{\mathcal{X}}\|.$$

Finally, if $j \neq k, \ell$ (that is, $\mathcal{U}_j$ is neither $\mathcal{B}$ nor $\mathcal{C}$), assume that $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_2$ (the case $\mathcal{U}_j \in \boldsymbol{\mathcal{X}}_1$ is analogous). Then $\boldsymbol{\mathcal{X}}_1 \mathcal{E}$ is β-normalized with respect to node $\mathcal{E}$ with $\beta_i = \alpha_i$ for all nodes $i \in \boldsymbol{\mathcal{X}}_1$. Therefore by Lemma 2.4 we have

$$\|\boldsymbol{\mathcal{X}}_1 \mathcal{E}\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_1} \alpha_i \|\mathcal{E}\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_1} \alpha_i c\varepsilon \|\mathcal{U}_k\| \|\mathcal{U}_\ell\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_1} \alpha_i \alpha_k \alpha_\ell \sqrt{r_k r_\ell} c\varepsilon \tag{3.14}$$

by Lemma 2.5. Moreover by the same argument as before we also have

$$\|\boldsymbol{\mathcal{X}}_2\| \leq \prod_{i \in \boldsymbol{\mathcal{X}}_2} \alpha_i \|\boldsymbol{\mathcal{X}}\|. \tag{3.15}$$

By combining (3.14) and (3.15) we finally obtain

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| = \|\boldsymbol{\mathcal{X}}_1 \mathcal{E} \boldsymbol{\mathcal{X}}_2\| \leq \prod_{i=1}^{n} \alpha_i \sqrt{r_k r_\ell} c\varepsilon \|\boldsymbol{\mathcal{X}}\|. \tag{3.16}$$

We have thus bounded the error introduced by SPLIT(A) and MERGE$(\mathcal{B}, \mathcal{C})$ assuming that $\boldsymbol{\mathcal{X}}$ is α-normalized. The bounds (3.8) and (3.16) are proportional to $\prod_{i=1}^{n} \alpha_i$, which must therefore be small to guarantee stability. They also depend on at most $\sqrt{r_k r_\ell}$, which is certainly bounded by the largest inner dimension r of the network.

We summarize the conclusions of this local error analysis in the following theorem.

THEOREM 3.1. *Let $\widehat{\boldsymbol{\mathcal{X}}}$ be obtained by applying either* SPLIT(A) *or* MERGE$(\mathcal{B}, \mathcal{C})$ *to $\boldsymbol{\mathcal{X}}$. Under the assumptions (3.2) and (3.3), and assuming that $\boldsymbol{\mathcal{X}}$ is α-normalized as defined in Definition 2.2, we have*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq rc \prod_{i=1}^{n} \alpha_i \varepsilon \|\boldsymbol{\mathcal{X}}\|,$$

where r is the largest inner dimension of $\boldsymbol{\mathcal{X}}$ and c is the constant in (3.2) and (3.3).

3.3. Global error. We now consider a network $\widehat{\boldsymbol{\mathcal{X}}}$ which is the result of a sequence of p local operations (SPLIT or MERGE) applied to $\boldsymbol{\mathcal{X}}$. Let $\boldsymbol{\mathcal{X}}_k$ be the network after having performed k operations, so that $\boldsymbol{\mathcal{X}}_0 = \boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{X}}_p = \widehat{\boldsymbol{\mathcal{X}}}$.

One technical difficulty is that the local errors incurred by each operation lead to second-order error terms: the error incurred by the kth operation depends on the errors incurred by the previous $k-1$ operations. This effect, however, only introduces an $O(\varepsilon^2)$ error term which is not particularly significant but quite tedious to compute precisely. Therefore, in the following we will not track explicitly these $O(\varepsilon^2)$ terms.

To bound the global error $\widehat{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{X}}$, we observe that it is bounded by the sum of the local errors. Indeed, we have the telescopic sum

$$\begin{aligned}
\|\widehat{\boldsymbol{\mathcal{X}}} - \boldsymbol{\mathcal{X}}\| &= \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_p\| \\
&= \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_1 + \boldsymbol{\mathcal{X}}_1 - \boldsymbol{\mathcal{X}}_p\| \\
&= \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_1 + \boldsymbol{\mathcal{X}}_1 - \ldots + \boldsymbol{\mathcal{X}}_{p-1} - \boldsymbol{\mathcal{X}}_p\| \\
&\leq \|\boldsymbol{\mathcal{X}}_0 - \boldsymbol{\mathcal{X}}_1\| + \ldots + \|\boldsymbol{\mathcal{X}}_{p-1} - \boldsymbol{\mathcal{X}}_p\| \\
&= \sum_{k=0}^{p-1} \|\boldsymbol{\mathcal{X}}_k - \boldsymbol{\mathcal{X}}_{k+1}\|.
\end{aligned}$$

Let n_k be the number of nodes of $\boldsymbol{\mathcal{X}}_k$. Assume that at each step k the network $\boldsymbol{\mathcal{X}}_k$ is $\alpha^{(k)}$-normalized, where $\alpha^{(k)} \in \mathbb{R}^{n_k}$ is a vector with elements $\alpha_i^{(k)}$, $i = 1\colon n_k$. Define

$$\zeta_k = r_k c \prod_{i=1}^{n_k} \alpha_i^{(k)},$$

where r_k is the largest inner dimension of $\boldsymbol{\mathcal{X}}_k$. Then by Theorem 3.1 we have

$$\|\boldsymbol{\mathcal{X}}_k - \boldsymbol{\mathcal{X}}_{k+1}\| \leq \zeta_k \varepsilon \|\boldsymbol{\mathcal{X}}_k\| = \zeta_k \varepsilon \|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2).$$

We therefore obtain

$$\begin{aligned}
\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| &= \sum_{k=0}^{p-1} \|\boldsymbol{\mathcal{X}}_k - \boldsymbol{\mathcal{X}}_{k+1}\| \\
&\leq \sum_{k=0}^{p-1} \zeta_k \varepsilon \|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2).
\end{aligned}$$

We summarize our conclusions in the following theorem, which determines sufficient conditions for an abstract computation on $\boldsymbol{\mathcal{X}}$ to be stable.

THEOREM 3.2. *Let $\widehat{\boldsymbol{\mathcal{X}}}$ be obtained by applying a sequence of p operations of the form* SPLIT(A) *or* MERGE$(\mathcal{B}, \mathcal{C})$ *to $\boldsymbol{\mathcal{X}}$. Under the assumptions (3.2) and (3.3), and assuming that at each step k of the computation, the intermediate network $\boldsymbol{\mathcal{X}}_k$ is $\alpha^{(k)}$-normalized as defined in Definition 2.2 with $\alpha^{(k)} \in \mathbb{R}^{n_k}$, we have*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon \|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{3.17}$$

with

$$s = \sum_{k=0}^{p-1} cr_k \prod_{i=1}^{n_k} \alpha_i^{(k)},$$

where r_k *is the largest inner dimension of* $\boldsymbol{\mathcal{X}}_k$ *and* c *is the constant in* (3.2) *and* (3.3).

The bound (3.17) can be simplified to

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \lesssim pr\alpha^n c\varepsilon \|\boldsymbol{\mathcal{X}}\|,$$

where $r = \max_k r_k$, $\alpha = \max_{i,k} \alpha_i^{(k)}$, and $n \leq \max_k n_k$ is the maximum number of nodes for which $\alpha_i^{(k)} \neq 1$ at any given step k. In other words, Theorem 3.2 shows that the error grows linearly with the kernel error $c\varepsilon$, the largest inner dimension r of the network, the number of operations p, and the term α^n, which is exponential in the number of non-orthogonal nodes n. Therefore, if the kernels are stable (small c), the main condition for the overall network computation to also be stable is that α^n is small.

As discussed in Subsection 2.4, there are important practical cases where we know α^n will be small. For e-normalized networks, all nodes are semi-orthogonal and thus $n = 0$ and $\alpha^n = 1$. For the sum of two e-normalized networks, we have proven in Theorem 2.8 that n is the number of leaves and $\alpha = \sqrt{2}$. In these cases, stability is thus guaranteed. We discuss concrete examples in the next section (Section 4). We also develop in the section after that (Section 5) a rounding algorithm that exploits the conclusion of our analysis to guarantee stability.

4. Examples of stable tensor computations. Our analysis in the previous section has identified conditions for an abstract computation in our framework to be stable. Here, we aim to provide some concrete examples of tensor computations that are proven stable by our analysis. To do so, we need to achieve two things: 1) show that the computation of interest can be expressed in our framework; and 2) show that throughout the computation the network remains α-normalized (ideally with small values of α).

4.1. Full. We begin with the simple example of the FULL operator, which contracts a tree tensor network into the full tensor it represents as illustrated in Figure 4.1 (going from left to right).

This operation can simply be expressed as a sequence of MERGE$(\mathcal{U}, \mathcal{V})$ of adjacent nodes $\mathcal{U}$ and $\mathcal{V}$, until only one node $\mathcal{X}$ remains, as shown in Algorithm 4.1.

Algorithm 4.1 FULL()

1: **while** there is more than one node left **do**
2: Choose any pair of adjacent nodes $\mathcal{U}$, $\mathcal{V}$.
3: $\mathcal{U} =$ MERGE$(\mathcal{U}, \mathcal{V})$
4: **end while**

Mathematically, the contractions may be performed in any order since all orders lead to the same result $\mathcal{X}$. However, some orders may be computationally more efficient depending on the dimensions. Moreover, in finite precision arithmetic, different orders are not numerically equivalent. The following result provides an error bound for α-normalized networks that is valid for any order of contraction.

COROLLARY 4.1. *Let Algorithm 4.1 be applied to a network* $\boldsymbol{\mathcal{X}}$ *α-normalized with respect to node* $\mathcal{U}_j$*. Regardless of the order of contractions, the computed* $\widehat{\mathcal{X}}$ *satisfies*

$$\|\mathcal{X} - \widehat{\mathcal{X}}\| \leq s\varepsilon\|\boldsymbol{\mathcal{X}}\| \tag{4.1}$$

with $s = (n-1)cr\prod_{i=1}^{n}\alpha_i$*, where* n *is the number of nodes of* $\boldsymbol{\mathcal{X}}$*,* r *is its largest inner dimension, and* c *is the constant in* (3.2) *and* (3.3)*.*

Proof. Algorithm 4.1 consists of $n-1$ MERGE operations where the intermediate networks $\boldsymbol{\mathcal{X}}_k$ at each step k are progressively more and more contracted: $\boldsymbol{\mathcal{X}}_k$ has $n_k = n-k$ nodes and its largest inner dimension r_k is certainly bounded by the largest inner dimension r of the original network $\boldsymbol{\mathcal{X}} = \boldsymbol{\mathcal{X}}_0$. By Lemma 2.3, at step k $\boldsymbol{\mathcal{X}}_k$ is $\beta^{(k)}$-normalized where $\beta^{(k)} \in \mathbb{R}^{n_k}$ satisfies $\prod_{i=1}^{n_k}\beta_i^{(k)} = \prod_{i=1}^{n}\alpha_i + O(\varepsilon)$. The $O(\varepsilon)$ term accounts for the inexactness of the operations which may slightly alter the α constants, and which only contribute to the $O(\varepsilon^2)$ term in the final error bound. Therefore, by Theorem 3.2 we obtain the bound (3.17) with

$$s = \sum_{k=0}^{n-2} cr_k \prod_{i=1}^{n_k}\beta_i^{(k)} + O(\varepsilon^2) \leq \sum_{k=0}^{n-2} cr\prod_{i=1}^{n}\alpha_i + O(\varepsilon^2).$$

∎

Corollary 4.1 shows that if a network $\boldsymbol{\mathcal{X}}$ is α-normalized with small values of α_i, it can be stably contracted into a full tensor $\mathcal{X}$.

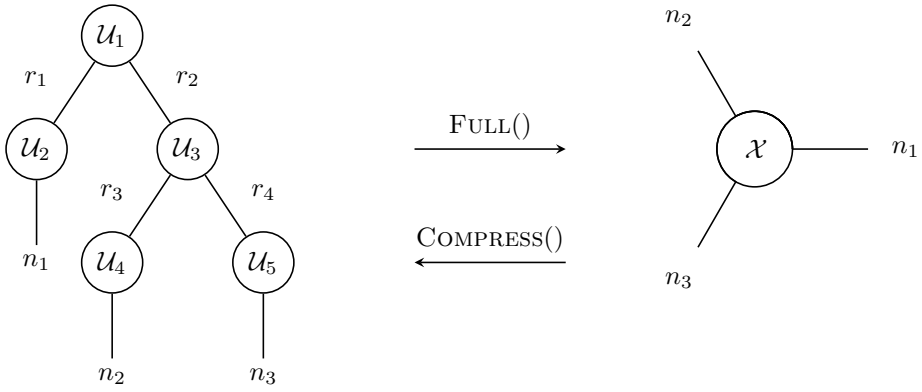


Fig. 4.1: Illustration of FULL and COMPRESS.

4.2. Compress. We can also express the inverse operation of the FULL operation, denoted as COMPRESS: given a full tensor $\mathcal{X}$, we seek to compress it into a tree tensor network $\boldsymbol{\mathcal{X}}$ with a prescribed topology, as illustrated in Figure 4.1 (going from right to left).

To minimize the inner dimensions of the network, we perform successive low-rank approximations, which are computed via truncated SVDs. Thus, we can express COMPRESS as a sequence of MATRICIZE, SPLIT, and TENSORIZE kernels, as described in Algorithm 4.2. Note that Algorithm 4.2 computes the nodes of the network in the order $\mathcal{U}_1$, ..., $\mathcal{U}_n$; this is done without loss of generality since the nodes can be relabelled. Note also that the "suitable dimension r" on Line 2 strongly depends on the network topology. For example, for a Tucker tensor, the $n-1 = d$ matricizations are

Algorithm 4.2 COMPRESS()

1: **for** each node $\mathcal{U}_i = \mathcal{U}_1, \dots \mathcal{U}_{n-1}$ of the prescribed topology **do**
2: $X = \text{MATRICIZE}(\mathcal{X}, r)$ for a suitable dimension r
3: $UY = \text{SPLIT}(X)$ {Truncated SVD; U is semi-orthogonal}
4: $\mathcal{U}_i = \text{TENSORIZE}(U)$
5: $\mathcal{X} = \text{TENSORIZE}(Y)$
6: **end for**
7: $\mathcal{U}_n = \mathcal{X}$ {The remaining $\mathcal{X}$ is the root node.}

done along the d outer dimensions of the original full tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \dots \times n_d}$. For more general tree networks, the leaf nodes similarly lead to matricizations along the outer dimensions; the nodes in the upper levels lead to matricizations along the inner dimensions computed in the previous levels.

Importantly, the truncated SVD yields a semi-orthogonal factor U and a non-orthogonal factor Y that we recursively compress. Therefore, Algorithm 4.2 yields a network $\boldsymbol{\mathcal{X}}$ where all nodes except the root are semi-orthogonal in the direction of the root. Therefore, $\boldsymbol{\mathcal{X}}$ is e-normalized and we obtain the following error bound.

COROLLARY 4.2. *Let Algorithm 4.2 be applied to a tensor $\mathcal{X}$ to compress it into a prescribed network topology $\boldsymbol{\mathcal{X}}$. The computed $\widehat{\boldsymbol{\mathcal{X}}}$ satisfies*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq (n-1)cr\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2), \tag{4.2}$$

where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, r is its largest inner dimension, and c is the constant in (3.2) and (3.3).

Proof. Algorithm 4.2 consists of $n-1$ truncated SVD (SPLIT) operations where the networks $\boldsymbol{\mathcal{X}}_k$ at each step k are progressively more and more compressed: $\boldsymbol{\mathcal{X}}_k$ has $k+1$ nodes and its largest inner dimension r_k is certainly bounded by the largest inner dimension r of the final network $\boldsymbol{\mathcal{X}}_{n-1} = \boldsymbol{\mathcal{X}}$. Since each node $\mathcal{U}_k$ computed at step k is semi-orthogonal in the direction of the root node $\mathcal{X}$, the network $\boldsymbol{\mathcal{X}}_k$ at each step k is β-normalized with respect to $\mathcal{X}$ (the only non-orthogonal node), where $\beta = e_{k+1} + O(\varepsilon)$ is equal to the vector of all ones of length $k+1$, up to $O(\varepsilon)$ inexactness due to the previous operations. Therefore, by Theorem 3.2 we have the bound (3.17) with $s = (n-1)cr$. ∎

4.3. Orthogonalize. Finally, we consider the ORTHOGONALIZE operation, which consists in orthogonalizing all nodes of a network except one, usually its root. As described in Algorithm 4.3, this can be achieved by performing a QR factorization of each node (Line 7), matricized in the direction of its parent (Line 6); the node is then replaced by the tensorized semi-orthogonal factor Q (Line 8), while the non-orthogonal factor R is merged with the parent node (Line 9). At the end of this process, all the nodes will be semi-orthogonal in the direction of the root $\mathcal{R}$, which will be the only node that is not semi-orthogonal and which satisfies $\|\mathcal{R}\| = \|\boldsymbol{\mathcal{X}}\|$.

The following result provides an error bound for orthogonalizing α-normalized networks.

COROLLARY 4.3. *Let Algorithm 4.3 be applied to a network $\boldsymbol{\mathcal{X}}$ α-normalized with respect to its root. Then the computed $\widehat{\boldsymbol{\mathcal{X}}}$ satisfies*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{4.3}$$

with $s = 2(n-1)cr\prod_{i=1}^{n}\alpha_i$, where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, r is its largest inner dimension, and c is the constant in (3.2) and (3.3).

Algorithm 4.3 ORTHOGONALIZE()

for each node $\mathcal{U}$ from leaves to root **do**
  if $\mathcal{U}$ is the root **then**
    return
  else
    $\mathcal{P} = \text{PARENT}(\mathcal{U})$
    $U = \text{MATRICIZE}(\mathcal{U}, \mathcal{P})$ {Matricize in the direction of the parent}
    $QR = \text{SPLIT}(U)$ {QR factorization; Q is semi-orthogonal}
    TENSORIZE(Q)
    MERGE($R, \mathcal{P}$)
  end if
end for

Proof. Algorithm 4.3 consists of $2(n-1)$ operations: $n-1$ QR factorizations (SPLIT) and $n-1$ contractions (MERGE). Let us denote $\boldsymbol{\mathcal{X}}_{2k}$ the network after k SPLIT and k MERGE, and $\boldsymbol{\mathcal{X}}_{2k+1}$ the network after $k+1$ SPLIT and k MERGE, for $k = 0\colon n-1$. Then $\boldsymbol{\mathcal{X}}_{2k}$ has n nodes and $\boldsymbol{\mathcal{X}}_{2k+1}$ has $n+1$ nodes; the largest inner dimension of either of them is bounded by the largest inner dimension r of the original network. Let us now prove by induction that, for any ℓ, $\boldsymbol{\mathcal{X}}_\ell$ is $\beta^{(\ell)}$-normalized such that

$$\prod_{i=1}^{m_\ell} \beta_i^{(\ell)} = \prod_{i=1}^{n} \alpha_i + O(\varepsilon), \tag{4.4}$$

with $\beta^{(\ell)} \in \mathbb{R}^{m_\ell}$, $m_\ell = n$ if $\ell = 2k$ and $m_\ell = n+1$ if $\ell = 2k+1$. For $\ell = 0$, $\boldsymbol{\mathcal{X}}_0 = \boldsymbol{\mathcal{X}}$ is α-normalized and so (4.4) holds. Assume it holds for a given ℓ. If $\ell = 2k$, $\boldsymbol{\mathcal{X}}_{2k+1}$ is obtained from $\boldsymbol{\mathcal{X}}_{2k}$ by replacing a given node $\mathcal{U}$ (say, node i) by two nodes corresponding to its approximate QR factors $\mathcal{U} = \mathcal{Q}\mathcal{R} + O(\varepsilon)$ (say, node i for the $\mathcal{R}$ factor and node $m = n+1$ for the $\mathcal{Q}$ factor). Since by assumption $\mathcal{U}$ is $\beta_i^{(\ell)}$-normalized and since $\|\mathcal{U}\mathcal{M}\| = \|\mathcal{R}\mathcal{M}\| + O(\varepsilon)$ for any tensor $\mathcal{M}$, $\mathcal{R}$ is $(\beta_i^{(\ell)} + O(\varepsilon))$-normalized. Moreover $\mathcal{Q}$ is semi-orthogonal and so is 1-normalized. Thus $\boldsymbol{\mathcal{X}}_{2k+1}$ is $\beta^{(\ell+1)}$-normalized with $\beta^{(\ell+1)} = [\beta^{(\ell)}\ 1] + O(\varepsilon)$ and hence (4.4) holds for $\ell+1$. If $\ell = 2k+1$ instead, then $\boldsymbol{\mathcal{X}}_{2k+2}$ is obtained from $\boldsymbol{\mathcal{X}}_{2k+1}$ by merging two nodes, say, nodes i and j which are $\beta_i^{(\ell)}$- and $\beta_j^{(\ell)}$-normalized, respectively. By Lemma 2.3 their contraction is $\beta_i^{(\ell)}\beta_j^{(\ell)}$-normalized, so that we have

$$\beta^{(\ell+1)} = [\beta_1^{(\ell)}, \dots, \beta_{i-1}^{(\ell)}, \beta_i^{(\ell)}\beta_j^{(\ell)}, \beta_{i+1}^{(\ell)}, \dots, \beta_{j-1}^{(\ell)}, \beta_{j+1}^{(\ell)}, \dots, \beta_n^{(\ell)}] + O(\varepsilon)$$

and (4.4) holds. By Theorem 3.2 we thus obtain (4.3). ∎

ORTHOGONALIZE is particularly of interest when $\boldsymbol{\mathcal{X}}$ is obtained as the sum of two e-normalized networks. As explained in Subsection 2.4, in this case $\boldsymbol{\mathcal{X}}$ is α-normalized with $\alpha_i = 1$ for non-leaf nodes and $\alpha_i \leq \sqrt{2}$ for leaf nodes. We may use ORTHOGONALIZE to make the network e-normalized again. Corollary 4.3 shows that the error incurred by this process is proportional to $2^{\ell/2}$, where ℓ is the number of leaves, and therefore stability is guaranteed as long as the number of leaves is small. For example, we have $\ell = 2$ for the TT topology and so orthonormalization of TT tensors is proven stable. Moreover, we note that this $2^{\ell/2}$ constant can be pessimistic in practice, because it does not take into account the structure of the nodes. Indeed, Definition 2.1 defines α_i as the largest amplification constant for *any* tensor $\mathcal{M}$, but the actual error tensor $\mathcal{E}$ may not attain

this value. In particular, let $\mathcal{U}_i$ be a $\sqrt{2}$-normalized leaf node and $\mathcal{P}$ be its parent node, which is orthogonal and has block-diagonal structure as described in Subsection 2.4. Then, if the error tensor $\mathcal{E}$ associated with an inexact computation on $\mathcal{P}$ matches the same block-diagonal structure, then we have $\|\mathcal{U}_i\mathcal{E}\| = \|\mathcal{E}\|$ and so there is no error amplification at all. We leave for future work formalizing this observation to rigorously establish in which cases the $2^{\ell/2}$ constant is provably pessimistic.

5. A general stable rounding algorithm. Tensor rounding is a fundamental task that consists in finding the optimal dimensions for each edge of the network while satisfying a prescribed accuracy ε. In this section, we exploit the conclusions of our error analysis in Section 3 to design a tensor rounding algorithm, that is both general and stable. It is *general*, because it can handle any tree tensor network and only relies on the four kernels defined by our framework; it is *stable*, because it takes care of preserving the semi-orthogonality of all nodes except one throughout the computation.

We first describe the proposed algorithm in Subsection 5.1, and then discuss in Subsection 5.2 how it relates to existing tensor rounding algorithms.

5.1. The proposed algorithm. Our rounding algorithm is outlined in Algorithm 5.1. It proceeds in two phases. The first phase (ORTHOGONALIZE) orthogonalizes all nodes of the network except its root as described and analyzed in Subsection 4.3. At the end of this first phase, the root of the network is therefore the only node that is not semi-orthogonal.

Algorithm 5.1 ROUND

1: ORTHOGONALIZE() (Algorithm 4.3)
2: Let $\mathcal{U}$ be the root (which is the only node not semi-orthogonal).
3: TRUNCATE($\mathcal{U}$) (Algorithm 5.2)

The second phase (TRUNCATE) then proceeds from the root down to the leaves and is implemented through the recursive function TRUNCATE outlined in Algorithm 5.2. It is initially called on the root node $\mathcal{U}$, which is the only node that is not semi-orthogonal. Then it loops on each of its children $\mathcal{C}$ (Line 2), and we tighten the dimension connecting $\mathcal{U}$ and $\mathcal{C}$ by computing a truncated SVD YV of the matricized $\mathcal{U}$ (Line 5). The semi-orthogonal factor V is tensorized to replace $\mathcal{U}$ (Line 6), while the non-orthogonal factor Y is merged with the children $\mathcal{C}$ (Line 7), which newly becomes the only node of the network that is not semi-orthogonal. We can therefore recursively call TRUNCATE on $\mathcal{C}$ (Line 8), which will tighten all the dimensions in the subtree rooted at $\mathcal{C}$. The recursion terminates when the child $\mathcal{C}$ is a leaf node (Lines 14 to 17): in this case, we compute a truncated SVD VY (Line 15) where the semi-orthogonal factor V is on the left and is merged with $\mathcal{C}$ (Line 16), whereas Y is tensorized (Line 17) to replace $\mathcal{U}$ which therefore remains non-orthogonal. Once the recursion on $\mathcal{C}$ has terminated, all that remains to do is to transfer the non-orthogonality back to the parent node $\mathcal{U}$. This is achieved by performing a QR factorization (Line 10) of $\mathcal{C}$ matricized in the direction of its parent $\mathcal{U}$ (Line 9). The semi-orthogonal factor Q is tensorized (Line 11) to replace $\mathcal{C}$ while the non-orthogonal factor R is merged (Line 12) with the parent $\mathcal{U}$, which is therefore once again the only node that is not semi-orthogonal.

The key idea of this algorithm is to concentrate the non-orthogonality in a single node of the network, which therefore carries the entire norm of the network, and this information is transferred through the network as the computation advances while keeping all the other nodes semi-orthogonal. In other words, the orthogonalization phase makes the network e-normalized, and then the network

Algorithm 5.2 TRUNCATE($\mathcal{U}$)

1: {On input, $\mathcal{U}$ is the only node not semi-orthogonal}
2: **for** all children $\mathcal{C}$ of $\mathcal{U}$ **do**
3: $U = \text{MATRICIZE}(\mathcal{U}, \mathcal{C})$ {Matricize $\mathcal{U}$ in the direction of its child}
4: **if** $\mathcal{C}$ is not a leaf **then**
5: $YV = \text{SPLIT}(U)$ {Truncated SVD; V is semi-orthogonal}
6: $\mathcal{U} = \text{TENSORIZE}(V)$
7: $\mathcal{C} = \text{MERGE}(Y, \mathcal{C})$ {$\mathcal{C}$ is now the only node not semi-orthogonal}
8: TRUNCATE($\mathcal{C}$) {Recursive call}
9: $C = \text{MATRICIZE}(\mathcal{C}, \mathcal{U})$ {Matricize $\mathcal{C}$ in the direction of its parent}
10: $QR = \text{SPLIT}(C)$ {QR factorization}
11: $\mathcal{C} = \text{TENSORIZE}(Q)$
12: $\mathcal{U} = \text{MERGE}(R, \mathcal{U})$
13: {$\mathcal{U}$ is back to being the only node not semi-orthogonal}
14: **else**
15: $VY = \text{SPLIT}(U)$ {Truncated SVD; V is semi-orthogonal}
16: $\mathcal{C} = \text{MERGE}(V, \mathcal{C})$
17: $\mathcal{U} = \text{TENSORIZE}(Y)$ {$\mathcal{U}$ still is the only node not semi-orthogonal}
18: **end if**
19: **end for**

is maintained e-normalized throughout the entire truncation phase. We therefore have the following key result.

COROLLARY 5.1. *Let Algorithm 5.1 be applied to a network $\boldsymbol{\mathcal{X}}$ α-normalized with respect to its root. Then the computed $\widehat{\boldsymbol{\mathcal{X}}}$ satisfies*

$$\|\boldsymbol{\mathcal{X}} - \widehat{\boldsymbol{\mathcal{X}}}\| \leq s\varepsilon\|\boldsymbol{\mathcal{X}}\| + O(\varepsilon^2) \tag{5.1}$$

with

$$s = cr\left(2(n-1)\prod_{i=1}^{n}\alpha_i + 4(n-1) - 2\ell\right),$$

where n is the number of nodes of $\boldsymbol{\mathcal{X}}$, ℓ is its number of leaves, r is its largest inner dimension, and c is the constant in (3.2) and (3.3).

Proof. The result is a direct consequence of Corollary 4.3, which bounds the error for the orthogonalization phase, and of the fact that the network remains e-normalized throughout the truncation phase. The $2(n-1)\prod_{i=1}^{n}\alpha_i$ accounts for the orthogonalization phase and comes from (4.3). The truncation phase leads to an error pcr where p is the number of operations. TRUNCATE requires four operations (two SPLIT and two MERGE) for each internal node (excluding the leaves and the root), and two operations (one SPLIT and one MERGE) for the leaves. Therefore we have $p = 4(n-1-\ell) + 2\ell = 4(n-1) - 2\ell$. ∎

Corollary 5.1 shows in particular that the sum of two e-normalized tensors can be stably rounded with Algorithm 5.1 as long as the number of leaves is small, since in this case $\prod_{i=1}^{n}\alpha_i = 2^{\ell/2}$ (see Subsection 2.4).

5.2. Comparison with existing tensor rounding algorithms. Various tensor rounding algorithms have been proposed in the literature, depending on the network topology.

For the tensor train format (which corresponds to a chain of nodes), a standard implementation of the algorithm is provided in [16, Alg. 2]. The algorithm is divided in two steps. First, all the nodes are orthogonalized from right to left except for the leftmost one. The second step starts from this leftmost non-orthogonal node, tightens its inner dimension to the prescribed accuracy ε via a truncated SVD, and merges the non-orthogonal factor with the node to its right. The process is then recursively repeated on this second node until all inner dimensions have been tightened. It is not hard to see that our Algorithm 5.1, when applied to a tensor train topology, reduces precisely to this very algorithm. In particular, this proves that the tensor train rounding algorithm from [16] is stable in finite precision arithmetic.

For the Tucker format, the HOSVD algorithm described in [17], which implements the Compress kernel (computation of the Tucker tensor directly from the full tensor), can be used to round a given Tucker tensor as follows. First, we make all nodes semi-orthogonal except for the root (Orthogonalize); then, we use HOSVD to compute a Tucker approximation of the root (Compress); finally, we contract the leaves of this Tucker root tensor with the leaves of the original tensor (Merge) to obtain its tightened Tucker approximation. This method performs exactly the same operations than our Round algorithm, but in a different order: our algorithm would interlace the truncations (Split) and contractions (Merge) along each successive dimension of the root, instead of first performing all the truncations in Compress with HOSVD and then all the contractions. However, since these operations are independent the two orders produce numerically equivalent results, and so we conclude that using this HOSVD-based method is also stable.

A widely used algorithm is the Gram SVD, which was originally proposed for the hierarchical Tucker format [5] (which corresponds to a complete binary tree topology) and notably implemented in the MATLAB Htucker toolbox [13]. Gram SVD has also been recently extended to the tensor train format [1]. For each node, Gram SVD computes its Gramian, a small square matrix whose eigenvalue decomposition can be used to recover the desired truncated SVD of the node in exact arithmetic. This algorithm is quite efficient and is in particular very suitable for parallelization. However, in finite precision arithmetic, this algorithm is unstable due to the ill-conditioning of the Gramians: the error analysis in [14] proves that an accuracy of the order of the square root of the machine precision can be expected, which is quite limiting when considering the use of low precision arithmetics. We will perform a detailed experimental comparison between this algorithm and our Algorithm 5.1 in the next section.

Finally, we also mention the PhD thesis [12], which contains a rounding algorithm for general tree topologies [12, Alg. 6]. This algorithm shares some similarities with our Algorithm 5.1, but also some differences. In particular, it performs the truncation from leaves to root instead of from root to leaves, which is costlier since the inner dimensions are truncated later than with Algorithm 5.1. Nevertheless, this algorithm also has the property of preserving the semi-orthogonality of all nodes except one throughout the computation, and so based on our error analysis, we expect it to be numerically stable.

6. Numerical experiments. We now present some numerical experiments to validate the numerical stability of our rounding algorithm, Algorithm 5.1, and compare its accuracy to the Gram SVD approach.

6.1. Experimental setting. All experiments were performed with MATLAB R2019a.

We use a tensor $\boldsymbol{\mathcal{Z}}$ obtained as the sum of two fourth-order tensors $\boldsymbol{\mathcal{X}}, \boldsymbol{\mathcal{Y}} \in \mathbb{R}^{100\times100\times100\times100}$, where the nodes of $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ are randomly generated with exponentially decaying elements. For these experiments, we use the hierarchical Tucker format (binary tree network), although we also tested other network topologies and obtained similar results.

We evaluate the accuracy of the algorithms with the normwise relative error

$$\eta = \|\boldsymbol{\mathcal{Z}} - \widehat{\boldsymbol{\mathcal{Z}}}\|/\|\boldsymbol{\mathcal{Z}}\|, \tag{6.1}$$

where $\widehat{\boldsymbol{\mathcal{Z}}}$ is the computed rounded tensor.

We test the use of various floating-point arithmetics: double, single, and half precisions, which correspond to a machine precision of $u = 2^{-53} \approx 1 \times 10^{-16}$, $u = 2^{-24} = 6 \times 10^{-8}$, and $u = 2^{-11} = 5 \times 10^{-4}$, respectively. We simulate the use of lower precisions using the `chop` library [10].

For the orthonormalization operations (Split without truncation), we use MATLAB's `qr` function, which implements Householder QR factorization. Since this is a stable operation, it satisfies assumption (3.2) with $\varepsilon = u$. For the truncated SVD operations (Split with truncation), we use MATLAB's `svd` to compute the full SVD and then truncate the decomposition based on the specified truncation threshold τ. Since the SVD is also stable, this operation satisfies assumption (3.2) with $\varepsilon = u + \tau$. Finally, for the contraction operations (Merge), we simply use standard matrix–matrix products, and so assumption (3.3) is satisfied with $\varepsilon = u$. Overall, we therefore expect our Round algorithm to achieve an accuracy of order $\varepsilon = u + \tau$, which we will validate in Subsection 6.2.

We will also compare with Gram SVD in Subsection 6.3. For these experiments, we use the MATLAB Htucker toolbox [13], with some modifications of our own for the purpose of the experiments (such as plugging the `chop` function in order to simulate low precision arithmetic).

6.2. Validating the stability of Algorithm 5.1. In Figure 6.1 we evaluate the accuracy of Algorithm 5.1 for various values of the truncation threshold τ and for various precisions u. The numbered labels next to the markers indicate the ratio between the size of the network rounded at accuracy τ using double precision and the size of the network rounded for the same τ but using a lower precision (single or half); when this ratio is equal to one, we omit it for the sake of readability. This ratio measures the potential loss of compression incurred when using low precision: indeed, the numerical noise introduced by the use of low precision may negatively affect the ability of the algorithm to tighten the dimensions to their optimal value if the requested accuracy τ is too close to the machine precision u.

Figure 6.1 confirms the numerical stability of Algorithm 5.1: the relative error (6.1) is of order $\varepsilon = \max(\tau, u)$, which means that if an accuracy of order ε is requested, using a truncation threshold $\tau = \varepsilon$ and any precision $u \lesssim \tau$ will deliver an error of order ε. Using a precision u that is too close to (or larger than) τ is not desirable since it leads to a significant loss of compression; however, taking u safely smaller than τ leads to exactly the same compression as if using double precision. To be precise, the figure shows that single precision arithmetic can be used without any loss of compression for $\tau \geq 10^{-7}$, whereas half precision arithmetic can be used for $\tau \geq 10^{-3}$. In conclusion, Algorithm 5.1 can be safely run in lower precision arithmetic, without degrading neither the accuracy nor the compression. This is a very attractive property that suggests that

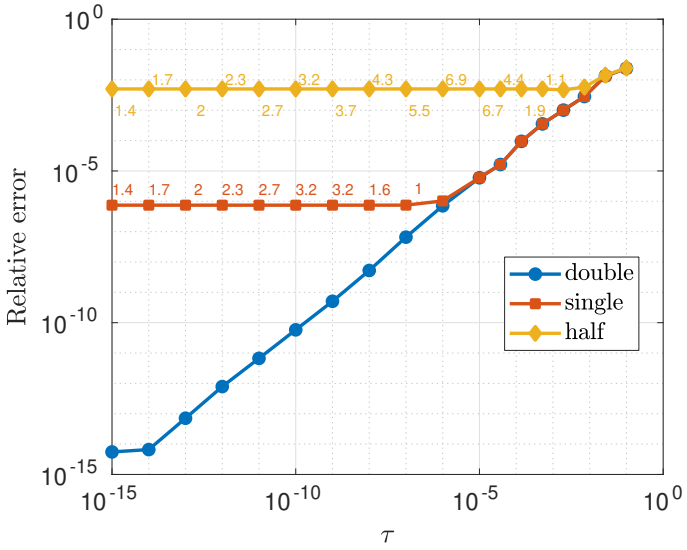


Fig. 6.1: Accuracy of Algorithm 5.1 depending on the truncation threshold τ and the floating-point precision u. The text labels next to the markers indicate the compression ratio between a given variant in lower precision and the reference compression obtained in double precision, for the same value of τ (if this ratio is equal to 1 we omit the label).

our algorithm should be able to take advantage of all the performance benefits of lower precision arithmetics on modern hardware.

6.3. Comparison with Gram SVD. Next, we compare the numerical behavior of Algorithm 5.1 with that of the widely used Gram SVD.

As mentioned in Subsection 5.2, Gram SVD is computationally efficient and in particular very suitable for parallelization. However, it is also numerically unstable: the error analysis in [14] proves an error bound of order $\sqrt{u}$ where u is the machine precision.

Figure 6.2 compares the accuracy of our Algorithm 5.1 and of Gram SVD when using double precision (Figure 6.2a), single precision (Figure 6.2b), or half precision (Figure 6.2c). We compare two variants of Gram SVD: one that uses a prescribed truncation threshold τ, and one that tightens the network to prescribed dimensions that correspond to the dimensions obtained via our Algorithm 5.1 with the corresponding τ.

The first Gram SVD variant with prescribed τ is unfortunately unable to correctly tighten the dimensions of the tensor except for very large values of τ, as indicated by the numbered labels that are greater than 1. This is due to the numerical noise of size $\sqrt{u}$ introduced by the instability of the algorithm, which therefore requires τ to be sufficiently larger than $\sqrt{u}$. For example, even in double precision, Gram SVD achieves a correct compression only when $\tau \geq 10^{-6}$. This effect is even worse when using single of half precisions, for which Gram SVD is effective only when $\tau \geq 10^{-3}$ and $\tau \geq 10^{-1}$, respectively.

The second Gram SVD variant circumvents this issue by enforcing a truncation to some predetermined dimensions; this is not a realistic scenario in general since the tightened dimensions may

not be known in advance (in fact, here, we are using our Algorithm 5.1 to determine the correct dimensions). However, in some cases, it may be acceptable to heuristically truncate to some prescribed dimensions. In any case, Figure 6.2 shows that this variant of Gram SVD achieves an accuracy of order $\sqrt{u}$ as expected; therefore, it achieves a comparable accuracy to that of Algorithm 5.1 only when $\tau \geq \sqrt{u}$ and and is much less accurate when $\tau \ll \sqrt{u}$.

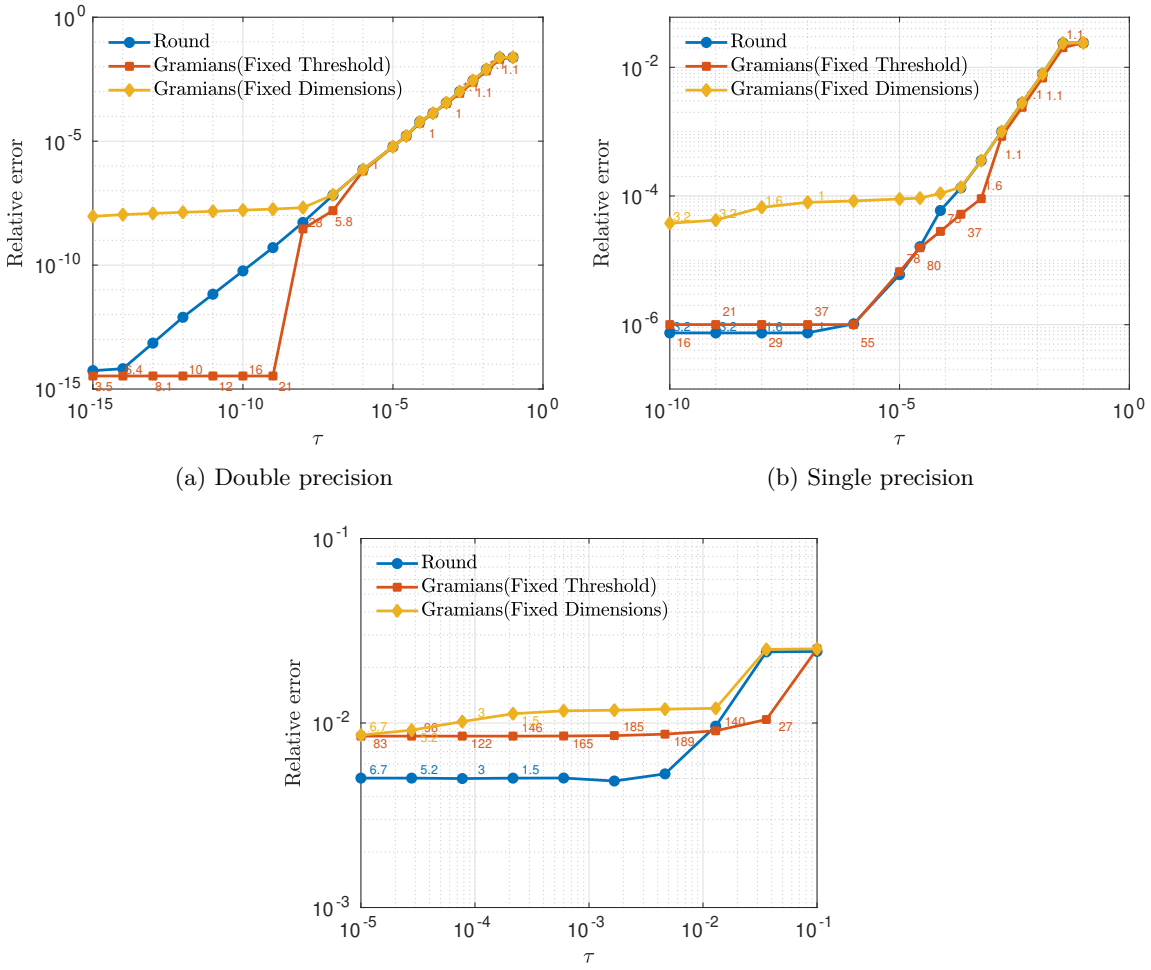
These results confirm that our Algorithm 5.1 is much more stable than Gram SVD and is in particular more resilient to the use of low precisions; therefore, it is better suited to exploit low precision arithmetics available on modern hardware.

7. Conclusion. We have proposed a general framework for analyzing the stability of tensor computations in finite precision arithmetic. Our framework is based on tree tensor networks, which encompass a wide range of tensor formats. It defines four key kernel operations that, when combined together, allow for expressing a wide range of tensor operations, such as compression, contractions, orthogonalization, and rounding. Our framework formalizes a key property, defined in Definition 2.2 and called α-normalization, that we have identified as fundamental for stability. An example of an α-normalized network is when all the nodes except one are semi-orthogonal (in this case $\alpha = e$, the vector of all ones).

We have performed an error analysis which leads to Theorem 3.2, our main theoretical result. It proves that an arbitrary computation composed of a sequence of local operations is stable provided that each local operation is stable, and that the network remains α-normalized throughout the computation (with small values for α). We use this key result to establish the stability of some common tensor computations in Corollaries 4.1 to 4.3.

Then, in Section 5, we have turned to the fundamental problem of tensor rounding. One of the most widely used tensor rounding algorithms is based on the Gram SVD, which is unstable in finite precision arithmetic. We exploit the conclusions of our error analysis to design a rounding algorithm, Algorithm 5.1, that is careful to maintain the semi-orthogonality of all nodes except one throughout the computation, which makes it stable (Corollary 5.1). Our numerical experiments have confirmed the stability of the algorithm in practice, and shown that it is significantly more resilient to the use of low precisions than Gram SVD. Thus our algorithm is better suited to take advantage of all the performance benefits of lower precision arithmetics on modern hardware. Since Gram SVD is however more parallel, this motivates a separate study, which we leave for future work, to compare the practical performance of both algorithms on various computer architectures depending on their parallel and arithmetic environment.

Acknowledgments. This work was supported by the NumPEx Exa-SofT (ANR-22-EXNU-0003), MixHPC (ANR-23-CE46-0005-01), and SELESTE (ANR-20-CE46-0008-01) projects of the French National Research Agency (ANR), and Paris Ile-de-France Region (DIM RFSI RC-TENSOR No 2021-05).



(a) Double precision

(b) Single precision

(c) Half precision

Fig. 6.2: Comparison between Algorithm 5.1 and Gram SVD depending on the truncation threshold τ and the floating-point precision u. The text labels next to the markers indicate the compression ratio between a given variant and the reference compression obtained with Algorithm 5.1 in double precision, for the same value of τ (if this ratio is equal to 1 we omit the label).

REFERENCES

[1] H. Al Daas, G. Ballard, and L. Manning, *Parallel tensor train rounding using Gram SVD*, in 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2022, pp. 930–940.

[2] S. Etter, *Parallel ALS algorithm for solving linear systems in the hierarchical Tucker representation*, SIAM J. Sci. Comput., 38 (2016), pp. A2585–A2609.

[3] G. Evenbly and G. Vidal, *Tensor network states and geometry*, Journal of Statistical Physics, 145 (2011), pp. 891–918.

[4] A. Falcó, W. Hackbusch, and A. Nouy, *Tree-based tensor formats*, SeMA Journal, 78 (2021), pp. 159–173.

[5] L. Grasedyck, *Hierarchical singular value decomposition of tensors*, SIAM J. Matrix Anal. Appl., 31 (2010), pp. 2029–2054.

[6] L. Grasedyck, D. Kressner, and C. Tobler, *A literature survey of low-rank tensor approximation techniques*, GAMM-Mitteilungen, 36 (2013), pp. 53–78.

[7] L. Grasedyck and C. Löbbert, *Distributed hierarchical SVD in the hierarchical Tucker format*, Numerical Linear Algebra with Applications, 25 (2018), p. e2174.

[8] E. Grelier, A. Nouy, and M. Chevreuil, *Learning with tree-based tensor formats*, arXiv preprint arXiv:1811.04455, (2018).

[9] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second ed., 2002, https://doi.org/10.1137/1.9780898718027.

[10] N. J. Higham and S. Pranesh, *Simulating low precision floating-point arithmetic*, SIAM J. Sci. Comput., 41 (2019), pp. C585–C602, https://doi.org/10.1137/19M1251308.

[11] T. G. Kolda and B. W. Bader, *Tensor decompositions and applications*, SIAM Rev., 51 (2009), pp. 455–500, https://doi.org/10.1137/07070111X.

[12] S. Krämer, *Tree tensor networks, associated singular values and high-dimensional approximation*, PhD thesis, 2020.

[13] D. Kressner and C. Tobler, *htucker—A MATLAB toolbox for tensors in hierarchical Tucker format*, Mathicse, EPF Lausanne, (2012).

[14] T. Mary, *Error analysis of the Gram low-rank approximation (and why it is not as unstable as one may think)*. To appear in SIAM J. Matrix Anal. Appl., Apr. 2024, https://hal.science/hal-04554516.

[15] R. Orús, *A practical introduction to tensor networks: Matrix product states and projected entangled pair states*, Annals of physics, 349 (2014), pp. 117–158.

[16] I. V. Oseledets, *Tensor-train decomposition*, SIAM J. Sci. Comput., 33 (2011), pp. 2295–2317.

[17] N. Vannieuwenhoven, R. Vandebril, and K. Meerbergen, *A new truncation strategy for the higher-order singular value decomposition*, SIAM J. Sci. Comput., 34 (2012), pp. A1027–A1052, https://doi.org/10.1137/110836067, https://doi.org/10.1137/110836067, https://arxiv.org/abs/https://doi.org/10.1137/110836067.