Incremental Hierarchical Tucker Decomposition

Doruk Aksoy DORUK@UMICH.EDU
Department of Aerospace Engineering
University of Michigan
Ann Arbor, MI, USA

Alex A. Gorodetsky GORODA@UMICH.EDU
Department of Aerospace Engineering
University of Michigan
Ann Arbor, MI, USA

Abstract

We present two new algorithms for approximating and updating the hierarchical Tucker decomposition of tensor streams. The first algorithm, *Batch Hierarchical Tucker - leaf to root* (`BHT-l2r`), proposes an alternative and more efficient way of approximating a batch of similar tensors in hierarchical Tucker format. The second algorithm, *Hierarchical Tucker - Rapid Incremental Subspace Expansion* (`HT-RISE`), updates the batch hierarchical Tucker representation of an accumulated tensor as new batches of tensors become available. The `HT-RISE` algorithm is suitable for the online setting and never requires full storage or reconstruction of all data while providing a solution to the incremental Tucker decomposition problem. We provide theoretical guarantees for both algorithms and demonstrate their effectiveness on physical and cyber-physical data. The proposed `BHT-l2r` algorithm and the batch hierarchical Tucker format offers up to 6.2× compression and 3.7× reduction in time over the hierarchical Tucker format. The proposed `HT-RISE` algorithm also offers up to 3.1× compression and 3.2× reduction in time over a state of the art incremental tensor train decomposition algorithm.

Keywords: Tensor decompositions, incremental algorithms, streaming data, scientific machine learning, data compression, latent representation, low-rank factorization

1 Introduction

Low-rank tensor decomposition formats (Oseledets, 2011; Tucker, 1966; Grasedyck, 2010; De Lathauwer et al., 2000b) provide an efficient way of representing multidimensional data arising from a large number of applications that include videos (Tian et al., 2023; Chen et al., 2024; Panagakis et al., 2021), MRI scan images (Zhang et al., 2019; Lehmann et al., 2022; Mai and Zhang, 2023), deep learning (Kossaifi et al., 2023; Luo et al., 2024; Yang et al., 2024), and solutions of scientific simulations (Marks and Gorodetsky, 2024; Pfaff et al., 2020). There does not exist a single, univerally optimal, low-rank tensor format. Instead, some well-known low-rank tensor formats are the CP format (Harshman et al., 1970; Carroll and Chang, 1970), the Tucker format (Tucker, 1966), the Tensor Train (TT) format (Oseledets, 2011), and the hierarchical Tucker (HT) format (Oseledets and Tyrtyshnikov, 2009; Grasedyck, 2010). These formats provide different forms of compression that exploit slightly different types of low-rank structure in data.

In this paper, we consider the problem decomposing data in batches. This problem is motivated by the fact that data in many applications is not available at once, and/or the

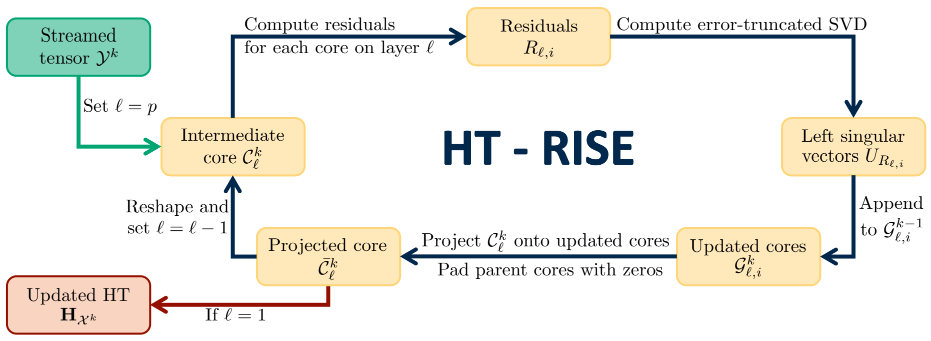

Streamed tensor $\mathcal{Y}^k$
Set $\ell = p$
Intermediate core $\mathcal{C}_\ell^k$
Compute residuals for each core on layer ℓ
Residuals $R_{\ell,i}$
Compute error-truncated SVD
HT - RISE
Left singular vectors $U_{R_{\ell,i}}$
Append to $\mathcal{G}_{\ell,i}^{k-1}$
Updated cores $\mathcal{G}_{\ell,i}^k$
Project $\mathcal{C}_\ell^k$ onto updated cores
Pad parent cores with zeros
Projected core $\bar{\mathcal{C}}_\ell^k$
Reshape and set $\ell = \ell - 1$
If $\ell = 1$
Updated HT $\mathbf{H}_{\mathcal{X}^k}$

Figure 1: *Flow of the proposed* ***`HT-RISE`*** *algorithm. The algorithm updates the hierarchical Tucker representation of an accumulated tensor in batch hierarchical Tucker form as new batches of tensors become available. Green represents the input data and red represents the output data.*

size of the data makes it infeasible or impossible to compute an approximation using a one-shot tensor decomposition algorithm due to computational issues (e.g., memory limits). For these scenarios, incremental algorithms can be effective by incrementally compress new data as it becomes available. They are particularly useful in the online setting where the data comes from a streaming process. While the literature has numerous incremental algorithms to compute the CP decomposition (Zeng and Ng, 2021; Smith et al., 2018), the Tucker decomposition (De et al., 2023; Malik and Becker, 2018), and the TT decomposition (Aksoy et al., 2024a; Liu et al., 2018; Kressner et al., 2023), we are not aware of incremental approaches for the HT format.

The main contribution of this paper is providing the first algorithm to incrementally construct a HT decomposition from streaming data.

Furthermore, many applications that leverage compressed representations use it as a latent space for performing downstream tasks. In this setting, it is important to have both an encoding and decoding procedure between the full data and the latent space. While the HT format has some advantages over other formats, it does not immediately provide an efficient latent vector that represents an encoding of a batch of data. Straight-forward applications of existing HT decomposition algorithms (Kressner and Tobler, 2014) treat the batch dimension same as an additional dimension identical to existing ones — so that when a batch of d-dimensional tensors $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$ is compressed, the batch dimension is represented as an HT leaf node with orthonormal columns.

We improve upon this approach by modifying the HT format to more efficiently represent the batch latent space by the HT root node.

Formally, we describe the problem as follows. We consider a d-way tensor to be a multidimensional array $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ with d dimensions. Parallel to the definition in Aksoy et al. (2024a), a *tensor stream* (or alternatively, *stream of tensors*), is a sequence of $d+1$-way tensors $\mathcal{Y}^1, \mathcal{Y}^2, \ldots$, where each element in the sequence $\mathcal{Y}^k \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N^k}$ is a N^k batch of d-dimensional tensors. A finite stream of tensors is called an *accumulation* and can be viewed as a $(d+1)$-way tensor $\mathcal{X}^k \in \mathbb{R}^{n_1 \times \cdots \times n_d \times \mathbf{n}^k}$ by concatenating the tensors in the stream along the last dimension, with $\mathbf{n}^k = \sum_{i=1}^{k} N^i$.

Taken together, the above setting motivates the following problem as the first one that we solve.

Problem 1 *(Approximating a batch of similar tensors in hierarchical Tucker format) Construct a scheme to compute an approximation $\hat{\mathcal{Y}}$ for a batch of d-dimensional tensors $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$ in a (batch-modified) hierarchical Tucker format. Furthermore, the computed representation should approximate the original batch of tensors with error $\|\mathcal{Y} - \hat{\mathcal{Y}}\|_F \leq \varepsilon_{des} \|\mathcal{Y}\|_F$.*

The scheme should also create individual latent representations for all tensors in a batch, and provide a means to decode a latent representation into the full tensor.

Next, we seek to perform the same task, but incrementally:

Problem 2 *(Updating an existing hierarchical Tucker representation as new batches of tensors become available) Construct a scheme to update the approximation $\hat{\mathcal{X}}^k$ of the accumulation tensor $\mathcal{X}^k$ after every increment k in hierarchical Tucker format. The constructed scheme should maintain the guaranteed bounds on the error $\|\mathcal{X}^k - \hat{\mathcal{X}}^k\|_F$ for all k. Furthermore, this approximate accumulation should represent all previous tensor increments $\mathcal{Y}^\ell$ with error $\|\mathcal{Y}^\ell - \hat{\mathcal{Y}}^\ell\|_F \leq \varepsilon_{des} \|\mathcal{Y}^\ell\|_F$ for any $\ell \leq k$, where $\hat{\mathcal{Y}}^\ell$ can be extracted from $\hat{\mathcal{X}}^k$.*

Our contributions are algorithms to solve these two problems:

1. The *Batch Hierarchical Tucker - leaf to root*, Algorithm 1, computes an approximate hierarchical Tucker representation of a batch of tensors $\mathcal{Y}$.

2. The *Hierarchical Tucker - Rapid Incremental Subspace Expansion*, Algorithm 2, updates an existing batch HT with new data.

Moreover, `HT-RISE` is suitable for the online setting and never requires full storage or reconstruction of all data while providing a solution to Problem 2. These algorithms are both rank adaptive and have provable error bounds. A high level overview of the `HT-RISE` algorithm is shown in Figure 1.

These contributions are achieved by limiting the number of new basis vectors, representing directions orthogonal to the span of the existing ones, appended to the hierarchical Tucker cores of an existing accumulation tensor after each data increment. Moreover, our theoretical results are empirically justified on both scientific applications dealing with compression of numerical solutions of partial differential equations (PDEs) and on applications arising from image-based data such as video streams and multispectral satellite images.

Our experiments show up to 1.5× compression and 1.7× reduction in time by the new Batch-HT format compared to the HT format on scientific data and up to 6.2× compression and 3.7× reduction in time for an image dataset. Furthermore, the proposed `HT-RISE` algorithm provides up to 3.1× compression and 4.5× reduction in time at physical data, and 2× compression and 5.3× reduction in time at image-based data over a state of the art incremental TT decomposition algorithm. In addition to its computational efficiency, the proposed `HT-RISE` algorithm discovers a latent representation that is more expressive and generalizable than the methods compared. This is evidenced by achieving the target approximation error on the test set with up to 15× less data than methods compared. Even for cases where the state-of-the-art TT decomposition algorithm fails to achieve the target error on the test set, `HT-RISE` still achieves the target error.

The rest of this paper is structured as follows. In section 2, we present the foundational concepts behind both the Tucker format and the HT format. In section 3, we present the

`BHT-l2r` and `HT-RISE` algorithms and prove their correctness. In section 4, we provide numerical experiments using our proposed approaches on physical and cyber-physical data. Finally, in section 5, we conclude the paper and discuss possible future extensions.

2 Background

This section presents the relevant background on tensors, Tucker format, and hierarchical Tucker format.

2.1 Tensors

The mode-i *matricization* (or *unfolding*) of a d-way tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ reshapes the tensor into a matrix $A_{(i)}$ with size $n_i \times n_1 \dots n_{i-1} n_{i+1} \dots n_d$. In this unfolding, the i-th dimension is swapped to first index and fibers corresponding to the i-th mode are mapped into the rows of the matrix. The mode-i unfolding operation will be represented as $\texttt{unfold}(\mathcal{A}, i)$ in the rest of the text.

The *contraction* between two tensors $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and $\mathcal{B} \in \mathbb{R}^{n_d \times n_{d+1} \times \cdots \times n_D}$ along the d-th dimension of $\mathcal{A}$ and the first dimension of $\mathcal{B}$ is a binary operation represented as

$$\mathcal{C} = \mathcal{A} \ {}_d\!\times_1 \ \mathcal{B}, \quad \text{where} \quad \mathcal{C}\left(i_1, i_2, \dots, i_{d-1}, i_{d+1}, \dots, i_d\right) = \textstyle\sum_{j=1}^{n_d} \mathcal{A}\left(i_1, \dots, i_{d-1}, j\right) \mathcal{B}\left(j, i_{d+1}, \dots, i_D\right),$$

and the output $\mathcal{C} \in \mathbb{R}^{n_1 \times \cdots \times n_{d-1} \times n_{d+1} \times \cdots \times n_D}$ becomes a $(D-2)$-way tensor. The subscripts on either side of the $\times$ sign indicate the contraction axes of the tensors on their respective sides.

Let $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ be a d-dimensional tensor, and $B_i \in \mathbb{R}^{m_i \times n_i}$ be matrices with $i = 1, \dots, d$. Then *multi-index contraction* between $\mathcal{A}$ and B_i is defined as[1]

$$\mathcal{C} = \mathcal{A} \times [\![B_1, \dots, B_d]\!], \quad \text{where} \quad \mathcal{C} = \left(\cdots\left(\left(\left(\mathcal{A} \ {}_1\!\times_2 \ B_1\right) \ {}_2\!\times_2 \ B_2\right) \ {}_3\!\times_2 \ \cdots\right) \ {}_d\!\times_2 \ B_d\right), \quad (1)$$

with $\mathcal{C} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$ the resulting d-dimensional tensor. The idea in (1) can be generalized to a contraction with d (or fewer) tensors $\mathcal{B}_i$ along their last dimensions in a similar manner. In case of contraction with d three-way tensors $\mathcal{B}_i \in \mathbb{R}^{m_{1_i} \times m_{2_i} \times n_i}$, (1) is denoted as

$$\mathcal{C} = \mathcal{A} \times [\![\mathcal{B}_1, \dots, \mathcal{B}_d]\!], \quad \text{where} \quad \mathcal{C} = \left(\cdots\left(\left(\left(\mathcal{A} \ {}_1\!\times_3 \ \mathcal{B}_1\right) \ {}_2\!\times_3 \ \mathcal{B}_2\right) \ {}_3\!\times_3 \ \cdots\right) \ {}_d\!\times_3 \ \mathcal{B}_d\right), \quad (2)$$

with $\mathcal{C} \in \mathbb{R}^{m_{1_1} \times m_{1_2} \times m_{2_1} \times m_{2_2} \times \cdots \times m_{d_1} \times m_{d_2}}$ being the resulting $2d$ dimensional tensor. Finally, it may be useful to only contract along a subset of dimensions. To this end, we can define a slightly modified version of (2) as

$$\mathcal{C} = \mathcal{A} \underset{\mathcal{I}}{\times} [\![\mathcal{B}_{i_1}, \dots, \mathcal{B}_{i_m}]\!], \quad \text{where} \quad \mathcal{C} = \left(\cdots\left(\left(\left(\mathcal{A} \ {}_{i_1}\!\times_3 \ \mathcal{B}_{i_1}\right) \ {}_{i_2}\!\times_3 \ \mathcal{B}_{i_2}\right) \ {}_{i_3}\!\times_3 \ \cdots\right) \ {}_{i_m}\!\times_3 \ \mathcal{B}_{i_m}\right) \quad (3)$$

to represent a contraction along m directions that are specified by the index set $\mathcal{I} = \{i_1, \dots, i_m\}$, with $i_j \in 1, \dots, d$ denoting the dimension of $\mathcal{C}$ that is involved with the contraction.

1. Even though we have provided examples strictly contracting the second dimension of the matrices with the tensor, the multi-index contraction can happen between any appropriate axis of the matrix and the tensor in question.

Concatenation of two tensors along the k-th dimension is a binary operation. Concatenating two d-dimensional tensors $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_k \times \cdots \times n_d}$ and $\mathcal{B} \in \mathbb{R}^{n_1 \times \cdots \times n_{k-1} \times m_k \times n_{k+1} \times \cdots \times n_d}$ along their k-th dimension is denoted by

$$\mathcal{C} = \mathcal{A} \overset{k}{\oplus} \mathcal{B}, \quad \text{where} \quad \mathcal{C}\left(i_1, \ldots, i_k, \ldots, i_d\right) = \begin{cases} \mathcal{A}\left(i_1, \ldots, i_k, \ldots, i_k\right) & \text{if } i_k \leq n_k \\ \mathcal{B}\left(i_1, \ldots, (i_k - n_k), \ldots, i_k\right) & \text{otherwise} \end{cases}. \quad (4)$$

Concatenation of a tensor with $\mathbf{0}_{m \times n \times p}$, a tensor of zeros with size $m \times n \times p$ will be referred to as *padding.*

2.2 Tucker format

When tensors have additional structure, it may be possible to effectively represent them in a more compact format. In this section we describe the Tucker format, which will form the basis of the hierarchical Tucker format we consider later. L.R. Tucker proposed a way to decompose a third order tensor into three factor matrices and a core tensor, coining the name *Tucker format* for this family of tensor representation format (Tucker, 1963; Tucker et al., 1964; Tucker, 1966). In Tucker format, a d dimensional tensor $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ is represented as a contraction between a d-dimensional core tensor $\mathcal{C} \in \mathbb{R}^{r_1 \times \cdots \times r_d}$ and d factor matrices $U_i \in \mathbb{R}^{n_i \times r_i}$,

$$\mathcal{Y} = \mathcal{C} \times [\![U_1, \ldots, U_d]\!]. \quad (5)$$

The Tucker representation reduces the storage from $\mathcal{O}\left(n^d\right)$ to $\mathcal{O}\left(dnr + r^d\right)$, where r_i are potentially much smaller than n_i. One of the well established ways to compute Tucker representation of a tensor is through the `HOSVD` (`H`igher `O`rder `S`ingular `V`alue `D`ecomposition) algorithm (De Lathauwer et al., 2000b; Tucker, 1966). In literature, there are numerous efficient methods of computing the Tucker representation of a tensor including direct (Vannieuwenhoven et al., 2012; Kressner and Perisa, 2017), randomized (Che and Wei, 2019; Kressner and Perisa, 2017; Tsourakakis, 2010; Zhang et al., 2018; Zhou et al., 2014; Minster et al., 2022; Sun et al., 2020), and iterative (De Lathauwer et al., 2000a; Kroonenberg and De Leeuw, 1980; Wen and So, 2015; Chachlakis et al., 2020; Kressner and Perisa, 2017; Tsourakakis, 2010; Eldén and Savas, 2009) algorithms.

2.3 Hierarchical Tucker Format

The Tucker format scales exponentially with the number of dimensions d, which prohibits its usage for high-dimensional tensors. One way to alleviate this curse of dimensionality is to exploit structure of the Tucker core. Such structure can be exploited via a recursive Tucker decomposition. In Oseledets and Tyrtyshnikov (2009), the resulting format is named as *tree Tucker*, and in Grasedyck (2010) it is mentioned as *hierarchical Tucker* (HT). The HT format introduces a hierarchy among dimensions and splits them accordingly. In the simplest, and most commonly used, binary splitting case, the Tucker core $\mathcal{C}$ in Equation (5) is reshaped into a tensor with $d/2$ dimensions and a decomposition of this core is created, recursively. The HT format has storage complexity $\mathcal{O}(dnR + dR^3)$, cubic in the maximum HT rank R.

The HT representation of a tensor is a couple $\mathbf{H} = (\mathbf{T}, \boldsymbol{\mathcal{G}})$, where $\mathbf{T}$ is a *dimension tree* and $\boldsymbol{\mathcal{G}}$ is a set of *HT cores.* The dimension tree is a connected graph of $2d - 1$ nodes,

$\mathbf{T} = \{\mathbf{N}_{\ell,i_\ell}\}$, that specifies a contraction ordering of the $2d-1$ HT cores, $\boldsymbol{\mathcal{G}} = \{\mathcal{G}_{\ell,i_\ell}\}$. The tree has depth p and $|\mathbf{T}_\ell|$ nodes in layer ℓ. A layer is a set of nodes equidistant from the root. The indices of each node in the tree $\mathbf{N}_{\ell,i_\ell} \in \mathbf{T}$ run from $\ell = 0,\ldots,p$ and $i_\ell = 1,\ldots,|\mathbf{T}_\ell|$, and each node describes how the corresponding core $\mathcal{G}_{\ell,i_\ell}$ contracts with its neighbors. Contracting all cores according to the dimension tree yields a reconstruction of the represented tensor. The root, interior, and leaves of the tree structure correspond to so-called *root*, *transfer*, and *leaf* cores. Figure 2 illustrates various configurations of dimension trees for a five-dimensional tensor.

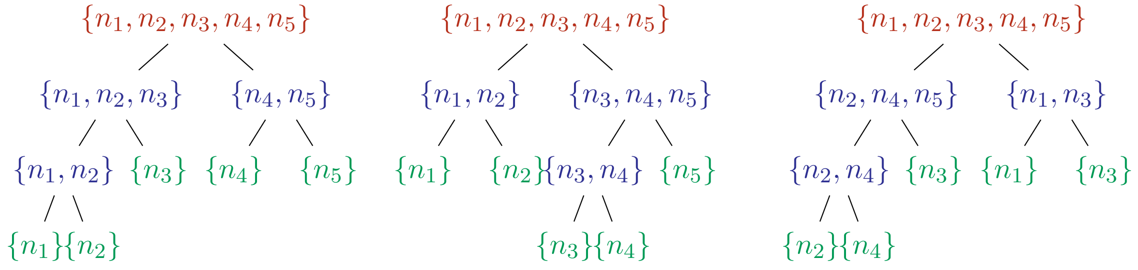


Figure 2: *Three possible configurations of the dimensions for a five dimensional tensor with shape $n_1 \times n_2 \times n_3 \times n_4 \times n_5$. Each dimension tree describes a different interaction between dimensions and therefore yields different compression performance. For a more detailed study on the effect of the axis reordering, please refer to Appendix D.3. Root, transfer, and leaf nodes are colored in red, blue, and green, respectively.*

For balanced binary trees, the depth is determined by $p = \lceil \log_2(d) \rceil$, with $\lceil\cdot\rceil$ defined as $\min\{i \in \mathbb{Z} | i \geq \cdot\}$. In this study, we focus exclusively on balanced dimension trees, which means that the leaf nodes appear on both the last level (p) and the penultimate level ($p-1$). The order in which dimensions are arranged significantly impacts the interaction among them, and consequently influences the result of the decomposition, as empirically demonstrated in Appendix D.3. Figure 2 also shows that there are always a total of d leaf cores, however these leaf cores are not always in the same layer. It will be useful to keep track of the dimension to which a leaf core refers. To this end, we will denote $d_{\ell,i_\ell} \in \{1,\ldots,d\}$ to be the dimension corresponding to leaf core $\mathcal{G}_{\ell,i_\ell}$. The subscript is only valid if the core is a leaf node.

Formally, a node is a couple $\mathbf{N}_{\ell,i_\ell} = (\mathbf{S}_{\ell,i_\ell}, \mathbf{P}_{\ell,i_\ell})$ that specifies the indices of the successors and parents with which $\mathcal{G}_{\ell,i_\ell}$ contracts. For each type of core we have:

Leaf core : $\mathcal{G}_{\ell,i_\ell} \in \mathbb{R}^{n_i \times r_{\ell,i_\ell}}$; $\mathbf{S}_{\ell,i_\ell} = \emptyset$, $\mathbf{P}_{\ell,i_\ell} = (\ell-1, \lceil i_\ell/2 \rceil)$

Transfer core : $\mathcal{G}_{\ell,i_\ell} \in \mathbb{R}^{r_{\ell+1,\alpha_{\ell+1}} \times r_{\ell+1,\alpha_{\ell+1}+1} \times r_{\ell,i_\ell}}$, $\mathbf{S}_{\ell,i_\ell} = \{(\ell+1, \alpha_{\ell+1}), (\ell+1, (\alpha_{\ell+1}+1))\}$, $\mathbf{P}_{\ell,i_\ell} = (\ell-1, \lceil i_\ell/2 \rceil)$

Root core : $\mathcal{G}_{0,1} \in \mathbb{R}^{r_{1,1} \times r_{1,2}}$, $\mathbf{S}_{0,1} = \{(1,1),(1,2)\}$ $\mathbf{P}_{0,1} = \emptyset$,

where $\alpha_{\ell+1} = 2(i_\ell - \beta_{\ell,i_\ell}) - 1$ indexes the appropriate successor node, and β_{ℓ,i_ℓ} is the number of leaf nodes in layer ℓ up to i_ℓ node of that layer. For a node $\mathbf{N}_{\ell-1,t}$, the order of successors $\mathbf{S}_{\ell-1,t} = \{\mathbf{N}_{\ell,i}, \mathbf{N}_{\ell,j}\}$ is determined by comparing their position within the ℓ-th layer, i.e., by ranking i and j.

The collection of ranks, called the HT ranks, $\mathbf{R} = [r_{1,1}, r_{1,2}, r_{2,1}, \ldots, r_{p,|\mathbf{T}_p|}]$ correspond to the sizes of the edges along which the leaves, transfer, and root core nodes contract. Contraction with a parent occurs along the last dimension of a node, while contraction with successors occurs with the first two.

To obtain the full tensor we can therefore perform contractions of each core with parents and successors according to the map provided by $\mathbf{T}$. A particularly illuminating contraction ordering is root-to-leaves. Root-to-leaves contraction begins contraction with the root node and moves down the tree, layer by layer. Overall, this contraction represents the reconstruction of a tensor $\mathcal{Y}$ from its HT format as

$$\mathcal{Y} = (\cdots((\mathcal{G}_{0,1} \times [\![\mathcal{G}_{1,1}, \mathcal{G}_{1,1}]\!]) \times [\![\mathcal{G}_{2,1}, \mathcal{G}_{2,2}, \mathcal{G}_{2,3}, \mathcal{G}_{2,4}]\!]) \times \cdots) \times [\![\mathcal{G}_{p,1}, \ldots, \mathcal{G}_{p,|\mathbf{T}_p|}]\!]. \quad (6)$$

This representation shows why the HT format is interpreted as a *hierarchical decomposition of the Tucker core tensor.* Indeed the last contraction is over the leaves, which can be interpreted as the Tucker leaves.

2.3.1 Leaves-to-root error truncated hierarchical Tucker decomposition

In this section we describe an algorithm to represent a given tensor $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ in low-rank format. In practice, $\mathcal{Y}$ is rarely exactly low rank, and hence decomposition algorithms generate a representation with a prescribed accuracy. Two strategies to compute a hierarchical Tucker representation include: 1) leaves-to-root decomposition (Grasedyck, 2010, Alg. 2), and 2) root-to-leaves decomposition (Kressner and Tobler, 2014, Alg. 5). Our proposed incremental method updates the hierarchical Tucker representation from leaves to root, and we limit discussion to this case.

Leaves-to-root decomposition essentially applies the HOSVD (or a variant) to each layer of the tree $\mathbf{T}$, beginning at the p-th layer. First, a truncated singular value decomposition is performed for $|\mathbf{T}_p|$ unfoldings — each unfolding corresponding to the dimensions of the i-th node in the layer. For each, an error truncated `SVD` with a Frobenius norm bound ε_{nw} on the residual $E_{p,i}$ is computed,

$$Y_{(d_{p,i})} = U_{p,i}\Sigma_{p,i}V_{p,i}^T + E_{p,i} \quad \|E_{p,i}\|_F \le \varepsilon_{nw}, \quad i = 1, \ldots, |\mathbf{T}_p| \quad (7)$$

where $U_{p,i} \in \mathbb{R}^{n_i \times r_{p,i}}$ is the matrix of left singular vectors with the truncation rank $r_{p,i}$, $\Sigma_{p,i} \in \mathbb{R}^{r_{p,i} \times r_{p,i}}$ is the matrix of singular values, and $V_{p,i}^T \in \mathbb{R}^{r_{p,i} \times m}$ is the matrix of right singular vectors. All nodes at the p-th layer correspond to leaf cores, yielding $\mathcal{G}_{p,i} = U_{p,i}$. The terms $\Sigma_{p,i}$ and $V_{p,i}$ are discarded in the leaves-to-root decomposition[2]. Once (7) is repeated for all leaves on layer p, the orthogonal $U_{p,i}$ matrices are contracted with $\mathcal{Y}$ along the dimensions refered to by the leaves

$$\bar{\mathcal{C}}_p = \mathcal{Y} \underset{\mathcal{I}}{\times} [\![U_{p,1}^T, \ldots, U_{p,|\mathbf{T}_p|}^T]\!], \text{ where } \quad \mathcal{I} = \{d_{p,i}; i = 1, \ldots, |\mathbf{T}_p|\} \quad (8)$$

to obtain an intermediate core $\bar{\mathcal{C}}_p$ that will be decomposed recursively in subsequent steps. Prior to such a decomposition, this intermediate core must be reshaped. In a perfect binary tree, there are d leaves (one for each dimension), and reshaping transforms $\bar{\mathcal{C}}_p$ into a tensor

2. If a variant of the `ST-HOSVD` algorithm (Vannieuwenhoven et al., 2012) is used to perform the computations, then Σ and V might be used in the subsequent steps within the layer.

$\mathcal{C}_{p-1}$ of dimensions $r_{p,1}r_{p,2} \times r_{p,3}r_{p,4} \times \cdots \times r_{p,d-1}r_{p,d}$. More generally, the dimension tree defines the groupings by defining the successors of each node. To this end, we can define the reshaped dimensions using an *index set* according to

$$\mathbf{I}_{\mathbf{T}_{p-1}} = \bigcup_{j=1}^{|\mathbf{T}_{p-1}|} \begin{cases} n_{d_{p-1,j}} & \text{if } \mathbf{N}_{p-1,j} \text{ is a leaf} \\ \prod\limits_{(p,m)\in\mathbf{S}_{p-1,j}} r_{p,m} & \text{else.} \end{cases}, \quad \text{so that} \quad \mathcal{C}_{p-1} = \texttt{reshape}\left(\bar{\mathcal{C}}_p,\ \mathbf{I}_{\mathbf{T}_{p-1}}\right). \tag{9}$$

Once we obtain $\mathcal{C}_{p-1}$, the error truncated `SVD` analogous to (7) and reshaping steps are repeatedly used to decompose each level to obtain $\mathcal{C}_\ell$ for $\ell = p-2, \ldots, 1$. For these subsequent levels, the left-singular vectors $U_{\ell,i}$ of the truncated SVD of the unfoldings $C_{\ell,(i)}$ become the transfer cores. Transfer cores in the HT format are three dimensional and thus the left singular vectors are reshaped according to

$$\mathcal{G}_{\ell,i} = \texttt{reshape}\left(U_{\ell,i}, \mathbf{I}_{\mathbf{N}_{\ell,i}}\right), \quad \text{where } \mathbf{I}_{\mathbf{N}_{\ell,i}} = \left\{ \bigcup_{(\ell+1,j)\in\mathbf{S}_{\ell,i}} r_{\ell+1,j} \right\} \cup r_{\ell,i}. \quad i = 1, \ldots, |\mathbf{T}_\ell|, \tag{10}$$

whereas any leaf cores are reshaped analogously to the approach at the p-th layer.

The decomposition is completed after a final error truncated `SVD` of $\mathcal{C}_1 \in \mathbb{R}^{r_{1,1}\times r_{1,2}}$

$$\mathcal{C}_1 = U_1\Sigma_1 V_1^T + E_1, \tag{11}$$

with $U_1 \in \mathbb{R}^{r_{1,1}\times r_0}$, $\Sigma_1 \in \mathbb{R}^{r_0\times r_0}$, $V_1^T \in \mathbb{R}^{r_0\times r_{1,2}}$ and $E_1 \in \mathbb{R}^{r_{1,1}\times r_{1,2}}$. Unlike the previous levels, all singular vectors and values are kept: U_1 is reshaped into $\mathcal{G}_{1,1}$, V_1^T is reshaped into $\mathcal{G}_{1,2}$, and the singular values Σ_1 become the root core $\mathcal{G}_{0,1}$. This procedure comes with the following guarantee

Theorem 1 (Adapted from Kressner and Tobler (2014) Lemma B.2) *For a d-dimensional tensor $\mathcal{Y} \in \mathbb{R}^{n_1\times\cdots\times n_d}$, the best HT approximation $\tilde{\mathcal{Y}}$ with an absolute approximation error $\|\mathcal{Y} - \tilde{\mathcal{Y}}\|_F \leq \varepsilon_{abs}$ can be obtained by prescribing a node-wise truncation error $\varepsilon_{nw} = \frac{\varepsilon_{abs}}{\sqrt{2d-3}}$ such that*

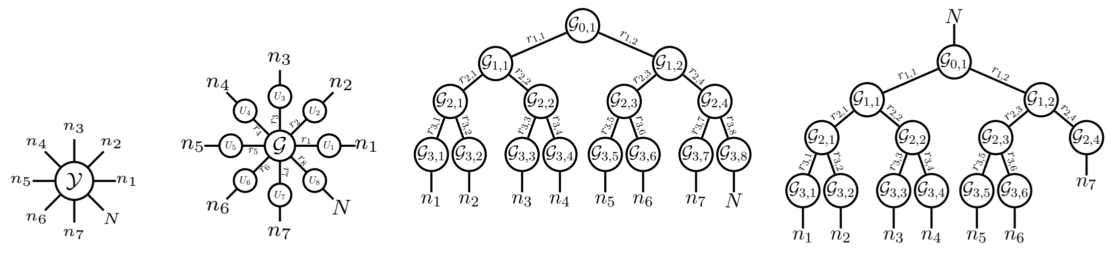
$$\|E_i\|_F \leq \varepsilon_{nw} \tag{12}$$

for all truncated `SVD` computations.

In the subsequent section we describe modifications to the HT format and corresponding approximation algorithms that are needed to adapt to the context of streaming batches of tensors.

3 Methodology

This section first presents the idea of computing an approximation for a batch of similar tensors in hierarchical Tucker format, then proposes a method to update an existing approximation in batch hierarchical Tucker format when new batches of tensors become available.



$\mathcal{Y}$
n_1 n_2 n_3 n_4 n_5 n_6 n_7 N
$\mathcal{G}$
U_1 U_2 U_3 U_4 U_5 U_6 U_7 U_8
r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8
$\mathcal{G}_{0,1}$
$\mathcal{G}_{1,1}$ $\mathcal{G}_{1,2}$
$\mathcal{G}_{2,1}$ $\mathcal{G}_{2,2}$ $\mathcal{G}_{2,3}$ $\mathcal{G}_{2,4}$
$\mathcal{G}_{3,1}$ $\mathcal{G}_{3,2}$ $\mathcal{G}_{3,3}$ $\mathcal{G}_{3,4}$ $\mathcal{G}_{3,5}$ $\mathcal{G}_{3,6}$ $\mathcal{G}_{3,7}$ $\mathcal{G}_{3,8}$
$r_{1,1}$ $r_{1,2}$ $r_{2,1}$ $r_{2,2}$ $r_{2,3}$ $r_{2,4}$
$r_{3,1}$ $r_{3,2}$ $r_{3,3}$ $r_{3,4}$ $r_{3,5}$ $r_{3,6}$ $r_{3,7}$ $r_{3,8}$

(a) *8D tensor* (b) *Tucker format* (c) *HT format* (d) *Batch HT format*

Figure 3: *Tensor network diagrams of an 8D tensor and its representations in Tucker format, hierarchical Tucker format, and batch hierarchical Tucker format. The streaming/batch dimension is labeled N*

3.1 Batch hierarchical Tucker

In this section, we aim to provide a solution to Problem 1 by presenting a hierarchical Tucker decomposition algorithm for batches of tensors. One approach to compressing a batch of tensors in HT format would be to treat the batch dimension as an additional dimension of the tensor, resulting in the batch dimension represented as another Tucker leaf on the p-th layer, as shown in Figure 3c. Assuming that the individual tensors in a batch are similar, i.e., low rank within the batch, such an approach can lead to an inefficient/inaccurate representation. As an alternative to adding a new leaf, we propose grouping the batch dimension into the root, as shown in Figure 3d. To do this, we exclude the batch dimension from the dimension tree during construction as if we are decomposing a single tensor from the batch.

Specifically, for an N-batch of d-dimensional tensors $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$, we use a dimension tree $\mathbf{T}$ corresponding to a tensor with only the first d dimensions of size $n_1, \ldots, n_d$. Throughout the decomposition process, the batch dimension is absorbed in the `SVD` computation and leaf/node contraction. Practically, this is performed by appending the batch size N to each layer's index set $\mathbf{I}_{\mathbf{T}_\ell}$ for $\ell = 0, \ldots, p$. The batch dimension remains intact throughout the decomposition process, and no SVD is performed for a reshaping relevant to this last dimension. As an example, after performing `HOSVD` on the p-th layer (except the batch dimension N) and contracting the obtained leaves as shown in Equation (8), we end up with the intermediate core tensor $\bar{\mathcal{C}}_p \in \mathbb{R}^{r_1 \times \cdots \times r_{|\mathbf{T}_p|} \times n_{|\mathbf{T}_p|+1} \times \cdots \times n_d \times N}$. We then reshape $\bar{\mathcal{C}}_p$ according to the modified index set $\mathbf{I}_{\mathbf{T}_{p-1}}$ and carry on with the decomposition. Since the batch dimension has been excluded from any `HOSVD` computation throughout the entire process, the root core $\mathcal{G}_{0,1}$ ends up being 3 dimensional where the third/new dimension has size N. The difference between an HT and a batch-HT is shown in Figure 3. Figure 4 depicts this procedure step by step for a tensor with $d = 5$.[3]

In this format, only the dimensionality of the root node is increased by one, all other transfer transfer nodes are still three dimensional and leaf-nodes remain two-dimensional. The pseudocode of the corresponding algorithm `BHT-l2r` is presented in Algorithm 1. The-

3. For simplicity of presentation, the above discussion assumed that the batch dimension is ordered as the last dimension of the tensor. However, as long as the batch dimension is added to the dimension tree properly (i.e. inserted to the correct order), the position of the batch index does not matter.

oretical guarantees for the approximation error upper bound for the `BHT-l2r` algorithm essentially follows from Theorem 1 and is provided in Appendix A as Theorem 4 for completion. Appendix D.1 presents a detailed comparison between HT and BHT formats on both scientific and image data.

Algorithm 1 `BHT-l2r`: Error truncated leaves-to-root batch hierarchical Tucker decomposition

```
 1: Input
 2:     $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$    Input tensor
 3:     $\mathbf{T}$    Dimension tree of the decomposition with depth $p$
 4:     $\varepsilon_{rel}$    Relative error tolerance
 5: Output
 6:     $\mathbf{H}_{\mathcal{Y}}$    Hierarchical Tucker representation with leaves and cores $\mathcal{G}_{\ell,i_\ell}$
 7: $\varepsilon_{nw} \leftarrow \varepsilon_{rel} \|\mathcal{Y}\|_F / \sqrt{2d-2}$    ▷ Node-wise error tolerance for SVD
 8: $\mathcal{C} \leftarrow \mathcal{Y}$
 9: for $i = 1, \dots, |\mathbf{T}_p|$ do    ▷ Compute the leaves on layer $p$
10:     $C \leftarrow$ unfold$(\mathcal{C}, d_{p,i})$    ▷ $d_{p,i} \in \{1, \dots, d\}$ is the dimension corresponding to the leaf node $\mathbf{N}_{p,i}$
11:     $\mathcal{G}_{p,i} \leftarrow$ SVD$(C, \varepsilon_{nw})$    ▷ Only the left singular vectors are kept with $\mathbf{R}_{p,i} = r_{p,i}$
12:     $\mathbf{I}_{\mathbf{N}_{p,i}} \leftarrow \{n_i, r_{p,i}\}$    ▷ Create index set for the leaf nodes
13: end for
14: $\mathcal{C} \leftarrow \mathcal{C} \underset{\mathcal{I}}{\times} [\![\mathcal{G}_{p,1}, \dots, \mathcal{G}_{p,|\mathbf{T}_p|}]\!]$    ▷ $\mathcal{I} = \{d_{p,i}; i = 1, \dots, |\mathbf{T}_p|\}$ as in (8)
15: for $\ell = p-1$ to 1 do
16:     $\mathcal{C} \leftarrow$ reshape$(\mathcal{C}, \mathbf{I}_{\mathbf{T}_\ell})$    ▷ $\mathbf{I}_{\mathbf{T}_\ell} \cup \{N\}$ with $\mathbf{I}_{\mathbf{T}_\ell}$ constructed using (9)
17:     $\mathcal{G}_{\ell,1}, \dots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|} \leftarrow$ HOSVD$(\mathcal{C}, \varepsilon_{nw})$    ▷ $\mathbf{I}_{\mathbf{N}_{\ell,j}}$ are created using (10) for $j = 1, \dots, |\mathbf{T}_\ell|$
18:     for $j = 1, \dots, |\mathbf{T}_\ell|$ do
19:         $\mathcal{G}_{\ell,j} \leftarrow$ reshape$(\mathcal{G}_{\ell,j}, \mathbf{I}_{\mathbf{N}_{\ell,j}})$    ▷ Folds $\mathcal{G}_{\ell,j}$ into 3D if $\mathbf{N}_{\ell,i_\ell}$ is a transfer node
20:     end for
21:     $\mathcal{C} \leftarrow \mathcal{C} \times [\![\mathcal{G}_{\ell,1}, \dots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|}]\!]$
22: end for
23: $\mathcal{G}_{0,1} \leftarrow$ reshape$(\mathcal{C}, \mathbf{I}_{\mathbf{N}_{0,1}})$    ▷ $\mathbf{I}_{\mathbf{N}_{0,1}} = \{r_{1,1}, r_{1,2}, N\}$
```

3.2 Incremental Updates

In this section we propose a solution to Problem 2 via a method to update an existing HT representation incrementally when new batches of tensors become available. Assume that at time k, we have an HT approximation $\mathbf{H}_{\mathcal{X}^{k-1}}$ of the accumulation tensor $\mathcal{X}^{k-1}$ in batch-HT format $(\mathbf{T}, \boldsymbol{\mathcal{G}}^{k-1})$. Then, a new batch of N^k d-dimensional tensors $\mathcal{Y}^k \in \mathbb{R}^{n_1 \times \cdots n_d \times N^k}$ arrives, and our task is to update $\boldsymbol{\mathcal{G}}^{k-1}$ to $\boldsymbol{\mathcal{G}}^k$ so as to generate a new approximation $\mathbf{H}_{\mathcal{X}^k}$. In other words, we describe an approach to update each core $\mathcal{G}^{k-1}_{\ell,i_\ell}$ into a new core $\mathcal{G}^{k}_{\ell,i_\ell}$, assuming an unchanged dimension tree $\mathbf{T}$. The proposed approach has three components: 1) project onto existing HT cores 2) compute residuals, and 3) update HT cores. The overall algorithm, `HT-RISE`, is provided in Algorithm 2 and each step is described in detail next.

Algorithm 2 `HT-RISE`: Incremental updates to a tensor represented as batch hierarchical Tucker

```
Input
    $\mathcal{Y}^k \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N^k}$    streamed $N^k$-batch of tensors at the $k$-th step
    $\mathbf{H}_{\mathcal{X}^{k-1}}$    Hierarchical Tucker representation with dimension tree $\mathbf{T}$ and cores $\mathcal{G}^{k-1}_{\ell,j}$
    $\mathbf{I}^{k-1}$    index set from $(k-1)$-th batch
    $\varepsilon_{rel}$    desired relative error truncation threshold
Output
    $\mathbf{H}_{\mathcal{X}^k}$    *Updated* hierarchical Tucker representation with dim. tree $\mathbf{T}$ and cores $\mathcal{G}^k_{\ell,j}$
    $\mathbf{I}^k$    *Updated* index set
Project streamed batch and check the representation quality
$\varepsilon_{des} \leftarrow \varepsilon_{rel} \|\mathcal{Y}^k\|_F$
$\mathcal{C} \leftarrow \mathcal{Y}^k$
for $\ell = p$ to 1 do    ▷ Project $\mathcal{Y}^k$ onto existing cores
    $\mathcal{C} \leftarrow \mathcal{C} \underset{\mathcal{I}}{\times} [\![\mathcal{G}_{\ell,1}, \ldots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|}]\!]$    ▷ $\mathcal{I} = \{d_{p,i}; i = 1, \ldots, |\mathbf{T}_p|\}$ needed just for the $p$-th layer
    $\mathcal{C} \leftarrow$ reshape$\left(\mathcal{C}, \mathbf{I}^{k-1}_{\mathbf{T}_{\ell-1}}\right)$    ▷ Reshape $\mathcal{C}$ for the next layer
end for
if $\sqrt{\|\mathcal{Y}^k\|_F^2 - \|\mathcal{C}\|_F^2} \leq \varepsilon_{des}$ then
    Skip updating all cores *except the root node*
    $\mathcal{G}^k \leftarrow \mathcal{G}^{k-1} \quad \forall \mathcal{G}^{k-1} \in \mathbf{H}_{\mathcal{X}^{k-1}} \setminus \mathcal{G}^{k-1}_{0,1}$
else
    Update cores on the last layer
    $\mathcal{C} \leftarrow \mathcal{Y}^k$
    $\varepsilon_{nw} \leftarrow \varepsilon_{des}/\sqrt{2d-2}$    ▷ Compute node-wise truncation error tolerance from $\varepsilon_{des}$
    for $j = 1$ to $|\mathbf{T}_p|$ do
        $C \leftarrow$ unfold$(\mathcal{C}, d_{p,j})$    ▷ $d_{p,j} \in \{1, \ldots, d\}$ is the index of the dimension corresponding to $\mathbf{N}_{p,j}$
        $R_{p,j} \leftarrow \Pi^{k-1}_{p,j} C$    ▷ $\Pi^{k-1}_{p,j}$ is computed according to (14)
        $U^k_R \leftarrow$ SVD$(R_{p,j}, \varepsilon_{nw})$    ▷ Error-truncated SVD on the residual according to (15)
        $\mathcal{G}^k_{p,j}, \mathbf{I}^k_{\mathbf{N}_{p,j}} \leftarrow$ expandCore$(\mathbf{N}_{p,j}, \mathbf{I}^{k-1}_{\mathbf{N}_{p,j}}, \mathcal{G}^{k-1}_{p,j}, U^k_R)$    ▷ Using Algorithms B.1 and B.2
        $\mathcal{G}^{k-1}_{p-1,m} \leftarrow$ padWithZeros$(\mathcal{G}^{k-1}_{p-1,m}, t, r_{R_{p,j}})$    ▷ Assume $\mathbf{N}_{p,j}$ is the $t$-th successor of $\mathbf{N}_{p-1,m}$
    end for
    $\mathbf{I}^k_{\mathbf{T}_{p-1}} \leftarrow$ updateIndexSet$(\mathbf{I}^{k-1}_{\mathbf{T}_{p-1}}, \mathbf{T}_{p-1})$    ▷ Update using Algorithm B.1 according to (9)
    $\mathcal{C} \leftarrow \mathcal{C} \underset{\mathcal{I}}{\times} [\![\mathcal{G}^k_{p,1}, \ldots, \mathcal{G}^k_{p,|\mathbf{T}_p|}]\!]$    ▷ $\mathcal{I} = \{d_{p,i}; i = 1, \ldots, |\mathbf{T}_p|\}$ as in (8)
    Update cores on the remaining layers
    for $\ell = p-1$ to 1 do
        $\mathcal{C} \leftarrow$ reshape$\left(\mathcal{C}, \mathbf{I}^k_{\mathbf{T}_\ell}\right)$
        for $j = 1$ to $|\mathbf{T}_\ell|$ do
            $C \leftarrow$ unfold$(\mathcal{C}, \ell)$
            $R_{\ell,j} \leftarrow \Pi^{k-1}_{\ell,j} C$    ▷ $\Pi^{k-1}_{\ell,j}$ is computed according to (14)
            $U^k_R \leftarrow$ SVD$(R_{\ell,j}, \varepsilon_{nw})$    ▷ Compute error truncated SVD on the residual according to (15)
            $\mathcal{G}^k_{\ell,j}, \mathbf{I}^k_{\mathbf{N}_{\ell,j}} \leftarrow$ expandCore$(\mathbf{N}_{\ell,j}, \mathbf{I}^{k-1}_{\mathbf{N}_{\ell,j}}, \mathcal{G}^{k-1}_{\ell,j}, U^k_R)$    ▷ Using Algorithms B.1 and B.2
            $\mathcal{G}^{k-1}_{\ell-1,m} \leftarrow$ padWithZeros$(\mathcal{G}^{k-1}_{\ell-1,m}, t, r_{R_{\ell,j}})$    ▷ Assume $\mathbf{N}_{\ell,j}$ is the $t$-th successor of $\mathbf{N}_{\ell-1,m}$
        end for
        $\mathbf{I}^k_{\mathbf{T}_{\ell-1}} \leftarrow$ updateIndexSet$(\mathbf{I}^{k-1}_{\mathbf{T}_{\ell-1}}, \mathbf{T}_{\ell-1})$
        $\mathcal{C} \leftarrow \mathcal{C} \times [\![\mathcal{G}^k_{\ell,1}, \ldots, \mathcal{G}^k_{\ell,|\mathbf{T}_\ell|}]\!]$
    end for
end if
$\mathcal{G}^k_{0,1} \leftarrow \mathcal{G}^{k-1}_{0,1} \oplus^3 \mathcal{C}$
```

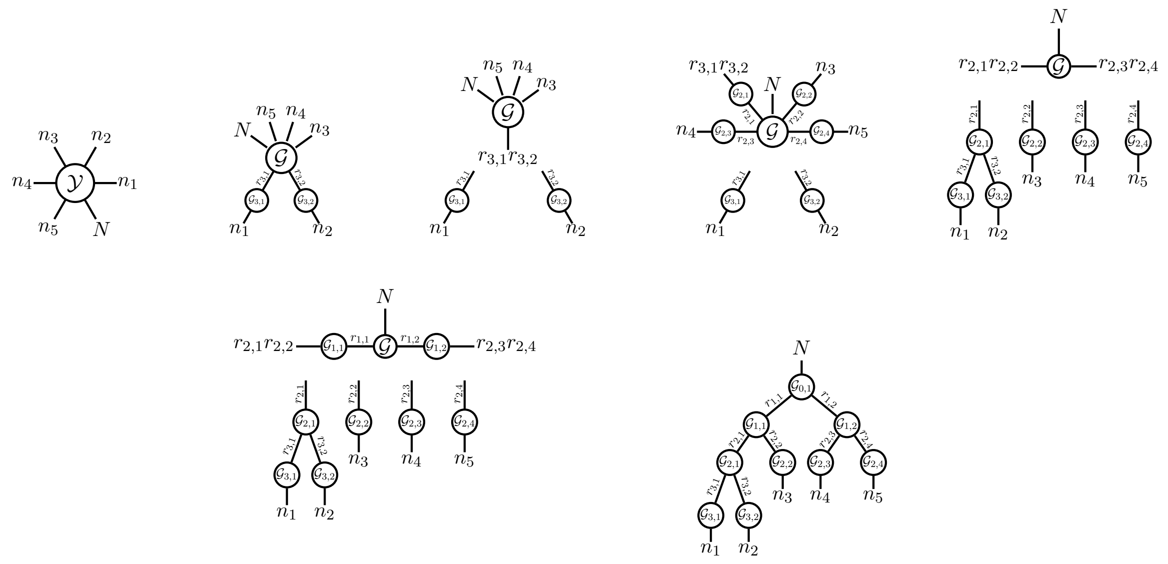


Figure 4: *Step-by-step decomposition of an N-batch of 5-D tensors with the `BHT-l2r` algorithm (Algorithm 1). The decomposition starts with the leaves of the last layer. Since the dimension tree is constructed beforehand, Algorithm 1 has the information about which dimensions' leaves will be on which layer through the dimension tree* $\mathbf{T}$. *Note that the batch dimension (N) remains intact throughout the entire decomposition process.*

Step 1: Projection onto existing HT cores Our proposed approach is centered around the fact that both Tucker leaves and Tucker cores are matrices with orthonormal columns (or under reshapings) when trained with Algorithm 1. This structure allows us to compute an approximation of newly streamed data using an existing set of hierarchical cores through a simple projection. This approximation will then guide refinement.

The projection is done by contracting the incoming batch with existing cores sequentially in leaves to the root direction (Algorithm 2 lines 12-15) and results in $\tilde{\mathcal{C}}_1$,the latent representation of $\mathcal{Y}^k$ using the cores of $\mathbf{H}_{\mathcal{X}^{k-1}}$. Since the projection onto the cores is simply a series of orthogonal projections, we can compute how well a set of existing HT representation approximates the input tensor by computing the difference in Frobenius norm between the input tensor and the projected tensor[4]. This error is then used to determine if the cores need to be updated. Then, the error of the projection ε_{proj} can be directly computed as

$$\varepsilon_{proj} = \sqrt{\|\mathcal{Y}^k\|_F^2 - \|\bar{\mathcal{C}}_1^k\|_F^2}. \tag{13}$$

If ε_{proj} is above the desired threshold ε_{abs}, the cores are updated to reduce the error below ε_{abs}. These updates will start from the last layer (layer p) and propagate towards the root of the hierarchical representation (layer 0) sequentially.

Step 2: Computing residuals To identify the missing orthogonal directions in the existing HT representation, we compute the residual $R_{\ell,j}$ for each core $\mathcal{G}_{\ell,j}^{k-1}$ that corresponds to the missing information related to the newest data batch. The residual is computed by

4. Please refer to Claim 1 for details and proof.

projecting the mode-j unfolding of the core $\mathcal{C}_\ell^k$ onto the orthogonal complement of the existing core $\mathcal{G}_{\ell,j}^{k-1}$ as

$$R_{\ell,j} = \Pi_{\ell,j}^{k-1} C_{\ell,(j)}^k, \quad \text{where} \quad \Pi_{\ell,j}^{k-1} = I - U_{\ell,j}^{k-1}\left(U_{\ell,j}^{k-1}\right)^T \text{ with } U_{\ell,j}^{k-1} = \begin{cases} \mathcal{G}_{\ell,j}^{k-1} & \text{if } \mathbf{N}_{\ell,j} \text{ is a leaf} \\ \texttt{reshape}\left(\mathcal{G}_{\ell,j}^k, \left[\alpha, r_{\ell,j}^{k-1}\right]\right) & \text{else} \end{cases}, \tag{14}$$

where $\Pi_{\ell,j}^{k-1}$ is the projection operator and $\alpha = \frac{1}{r_j^\ell}\prod_{\gamma\in\mathbf{I}^{k-1}_{\mathbf{N}_j^\ell}}\gamma$. Note that the projection operator $\Pi_{\ell,j}^{k-1}$ slightly differs for the Tucker leaves and the Tucker cores. Since $\mathcal{G}_{\ell,j}$ are already 2-dimensional for leaf nodes, there is no reshaping required, while the transfer nodes $\mathcal{G}_{\ell,j}^{k-1}$ must first be reshaped into orthonormal matrices $U_{\ell,j}^{k-1} \in \mathbb{R}^{\alpha\times r_{\ell,j}^{k-1}}$. Once the residual $R_{\ell,j}$ is computed, the next step is to find the directions in which $\mathcal{G}_{\ell,j}^{k-1}$ must be expanded.

Step 3: Performing core updates The idea behind incremental updates is similar to Aksoy et al. (2024a). It seeks to append directions that span the residual to the existing basis. The core update process starts with an error truncated `SVD` on the residual $R_{\ell,j} \in \mathbb{R}^{\alpha\times\beta}$ with α same as Step 2 and $\beta = \frac{1}{\alpha}\prod_{\theta\in\mathbf{I}_{\mathbf{T}_\ell}}\theta$ as

$$R_{\ell,j} = U_{R_{\ell,j}}^k \Sigma_{R_{\ell,j}}^k (V_{R_{\ell,j}}^k)^T + E_{R_{\ell,j}}, \tag{15}$$

such that $U_{R_{\ell,j}}^k \in \mathbb{R}^{\alpha\times r_{R_{\ell,j}}}$, $\Sigma_{R_{\ell,j}}^k \in \mathbb{R}^{r_{R_{\ell,j}}\times r_{R_{\ell,j}}}$, and $V_{R_{\ell,j}}^k \in \mathbb{R}^{\beta\times r_{R_{p,1}}}$. Similar to `BHT-l2r` algorithm, we distribute the error uniformly over the cores and determine $r_{R_{\ell,j}}$ such that the truncation error $\|E_{R_{\ell,j}}\|_F$ is at most $\varepsilon_{abs}/\sqrt{2d-2}$. Other approaches to distribute the error over the cores can be considered as well. We include one such alternative method in Appendix C.

Since we compute $U_{R_{\ell,j}}^k$ from the residual $R_{\ell,j}$, we have $U_{R_{\ell,j}}^k \perp U_{\ell,j}^{k-1}$ by definition. This allows us to expand the orthogonal bases of $U_{\ell,j}^{k-1}$ by simply concatenating with $U_{R_{\ell,j}}^k$ as

$$U_{\ell,j}^k = \begin{bmatrix} U_{\ell,j}^{k-1} & U_{R_{\ell,j}}^k \end{bmatrix}, \tag{16}$$

with $U_{\ell,j}^k \in \mathbb{R}^{\alpha\times r_{\ell,j}^k}$, such that $r_{\ell,j}^k = r_{\ell,j}^{k-1} + r_{R_{\ell,j}}$. An update of the index set $\mathbf{I}_{\mathbf{N}_{\ell,j}}^k$ using (10) follows (16) to reflect the updated rank $r_{\ell,j}^k$. The new basis $U_{\ell,j}^k$ can then be appropriately reshaped into transfer or leaf cores in the same manner as for the HT described in Section 2.3.1. The new core $U_{\ell,j}^k$ is then reshaped appropriately to form $\mathcal{G}_{\ell,j}^k$.

After updating $\mathcal{G}_{\ell,j}^{k-1}$ to $\mathcal{G}_{\ell,j}^k$, the new dimensions result in a shape mismatch between $\mathcal{G}_{\ell,j}^k$ and its parent Tucker core $\mathcal{G}_{\ell-1,t}^{k-1}$ on layer $(\ell-1)$. If $\mathcal{G}_{\ell-1,t}^{k-1} \in \mathbb{R}^{r_{\ell,j}^{k-1}\times r_{\ell,m}^{k-1}\times r_{\ell-1,t}^{k-1}}$, then the core is padded according to

$$\mathcal{G}_{\ell-1,t}^{k-1} = \mathcal{G}_{\ell-1,t}^{k-1} \overset{1}{\oplus} \mathbf{0}_{r_{R_{\ell,1}}\times r_{\ell,m}^{k-1}\times r_{\ell-1,t}^{k-1}}, \quad \text{or} \quad \mathcal{G}_{\ell-1,t}^{k-1} = \mathcal{G}_{\ell-1,t}^{k-1} \overset{2}{\oplus} \mathbf{0}_{r_{\ell,j}^{k-1}\times r_{R_{\ell,1}}\times r_{\ell-1,t}^{k-1}}, \tag{17}$$

depending on the order of $\mathcal{G}_{\ell,j}^{k-1}$ in the set of successors of $\mathcal{G}_{\ell-1,t}^{k-1}$. Following (17), the index sets of $\mathbf{N}_{\ell-1,t}$ is updated using (10) to reflect the padding.

Steps 2 and 3 are repeated for all nodes on the ℓ-th layer (i.e., for $j = 1, \ldots, |\mathbf{T}_\ell|$). Although we describe the updates to be sequential within a layer, the updates can be computed in parallel to increase the computational speed. Once all $\mathcal{G}_{\ell,j}^{k-1}$ are updated to $\mathcal{G}_{\ell,j}^{k}$ for $j = 1, \ldots, |\mathbf{T}_\ell|$, the index set of $(\ell-1)$-th layer is updated to $\mathbf{I}_{\mathbf{T}_{\ell-1}}^{k}$ using (9) to reflect the rank updates.

Step 1 (revisited): Projection onto updated HT cores Following core updates, the algorithm projects $\mathcal{C}_\ell^k$ onto the *updated* cores on the ℓ-th layer by

$$\bar{\mathcal{C}}_\ell^k = \mathcal{C}_\ell^k \times [\![\mathcal{G}_{\ell,1}^k, \ldots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|}^k]\!]. \tag{18}$$

Then, the algorithm reshapes $\bar{\mathcal{C}}_\ell^k$ using the index set $\mathbf{I}_{\ell-1}^k$ to obtain $\mathcal{C}_{\ell-1}^k$. After reshaping, the algorithm returns to Step 2 and repeats the process for all layers up to to $\ell = 1$.

Once we update the cores on the first layer, $\mathcal{G}_{1,1}^k$ and $\mathcal{G}_{1,2}^k$ are used to project $\mathcal{C}_1^k$ and obtain $\bar{\mathcal{C}}_1^k$, the representation of $\mathcal{Y}^k$ using the updated Tucker cores. Finally, we update $\bar{\mathcal{G}}_0^{k-1}$ as

$$\mathcal{G}_0^k = \bar{\mathcal{G}}_0^{k-1} \overset{3}{\oplus} \bar{\mathcal{C}}_1^k, \quad \text{where} \quad \bar{\mathcal{C}}_1^k = \mathcal{C}_1^k \times [\![\mathcal{G}_{1,1}^k, \mathcal{G}_{1,2}^k]\!], \tag{19}$$

and therefore conclude updating $\mathbf{H}_{\mathcal{X}^{k-1}}$ to $\mathbf{H}_{\mathcal{X}^k}$. The flow of this presented update scheme is summarized in Algorithm 2 as `HT-RISE`.

4 Numerical Experiments

In this section, we compare the proposed `HT-RISE` algorithm against an incremental tensor train decomposition algorithm, `TT-ICE`*, which is proven to demonstrate state-of-the-art compression performance for the tensor train format in Aksoy et al. (2024a).

Comparisons are made on both scientific as well as image based datasets. Scientific datasets include compressible Navier-Stokes simulations from PDEBench dataset (Takamoto et al., 2022), as well as simulations of a PDE-driven chaotic system of self-oscillating gels (Alben et al., 2019). For image based datasets, we will compare the algorithms with Minecraft video frames from MineRL Basalt competition dataset (Milani et al., 2024) and multispectral images from the BigEarthNet dataset (Sumbul et al., 2019, 2021). Table 1 summarizes the datasets used in the experiments. We also include further analyses including the effect of tensor reshapings (Appendix D.2) and the effect of different axis reorderings (Appendix D.3) on the performance of `HT-RISE` in the appendices.

As a preview, we observe that `TT-ICE`* to perform better on simpler datasets such as the self-oscillating gel snapshots or at higher relative error tolerances, while `HT-RISE` performs better on larger datasets with intricate multi-scale features. Furthermore, `HT-RISE` is able to generalize better to unseen data with fewer batches compared to `TT-ICE`*. For image datasets, `HT-RISE` retains more of the qualitative features of the original images compared to `TT-ICE`* at the same relative error threshold.

All experiments are executed on University of Michigan compute nodes on the Lighthouse cluster with 16 Intel(R) Xeon(R) Platinum 8468 cores and 64GB of memory. All algorithms presented in this work are implemented in Python using the `NumPy` library (Harris et al., 2020). We used the publicly available version of `TT-ICE`* from GitHub in our experiments.

Table 1: *Summary of the datasets used in the experiments. Train units and test units refer to the number of simulations, videos, or images in the training and test sets, respectively. The batch size refers to the number of simulations or images in a single batch. The batch shape refers to the original shape of a single batch before performing any reshaping/resizing operations. The total size refers to the total size of the dataset on disk. Sims, vids, and ims refer to simulations, videos, and images, respectively.*

Name	Train units	Test units	Batch size	Batch shape	Total size
PDEBench	480 sims	120 sims	1 sim	$64 \times 64 \times 64 \times 5 \times 21 \times 1$	66 GB
Self-oscillating gels	8,000 sims	15,000 sims	1 sim	$3367 \times 3 \times 10 \times 1$	18 GB
Basalt MineRL	5449 vids	17 vids	20 frames	$360 \times 640 \times 3 \times 20$	183 GB
BigEarthNet	566,712 ims	23,612 ims	100 ims	$120 \times 120 \times 12 \times 100$	104 GB

4.1 Performance Metrics

This section presents the performance metrics used to compare the algorithms. The metrics are chosen to evaluate the accuracy of the approximation, the compression ratio, the reduction ratio, the execution time, and the generalization performance.

4.1.1 Compression ratio (CR) and Reduction ratio (RR)

The compression ratio is a measure of how much each approximation compresses the original accumulation. It is defined as

$$CR = \frac{\texttt{num\_elem}(\mathcal{X}^k)}{\texttt{num\_elem}(\mathbf{H}_{\mathcal{X}^k})}, \tag{20}$$

where $\texttt{num\_elem}(\mathcal{X}^k)$ denotes the number of elements of the original accumulation and $\texttt{num\_elem}(\mathbf{H}_{\mathcal{X}^k})$ is the number of elements of the approximation, which corresponds to the sum of number of elements in all HT cores. A compression ratio of 1 indicates that the approximation does not compress the original accumulation at all, while a compression ratio below 1 indicates to an inefficient approximation that uses more elements to approximate the original accumulation.

As shown in Chen et al. (2024), the `TT-ICE`* algorithm also provides a latent space representation of the data, which can be used in downstream learning tasks. For downstream tasks, the size of the input space plays a key role in computational efficiency. Therefore in this study we also investigate how much reduction is achieved through the incremental tensor decomposition algorithms and call this the *reduction ratio* (RR). The RR is the ratio of the size of a single tensor in the accumulation to the size of the latent space. It is defined as

$$RR = \frac{\texttt{num\_elem}(\mathcal{Y}^{k,i})}{\texttt{num\_elem}(\texttt{encode}(\mathcal{Y}^{k,i}))}, \tag{21}$$

where $\mathcal{Y}^{k,i}$ denotes a single tensor from the N^k-batch of tensors, and $\texttt{project}(\mathcal{Y}^{k,i})$ denotes the resulting latent space representation from encoding $\mathcal{Y}^{k,i}$ using the relevant algorithm — e.g., `TT-ICE`*, `HT-RISE`, etc..

4.1.2 Compression time

Compression time is a measure of how long it takes to update the approximation. Note that this does not consider the time it takes to load and preprocess the data.

In addition to the compression time, we also set a maximum walltime limit for each experiment. The maximum walltime considers the time it takes to load the data onto the memory, preprocess the data (if necessary), update the approximation, and check the approximation error of the updated cores on the test set. If an experiment exceeds the maximum walltime, the experiment is terminated and the metrics at the time of termination are reported.

4.1.3 Relative test error

When using the compression for downstream learning tasks, having a latent mapping that achieves a similar approximation error for both in- and out-of-sample datasets is valuable. The relative test error (RTE) is a measure of how well the approximation generalizes to unseen data. It is measured using the relative approximation error of tensors averaged over the test set. This performance is expected to get better as each approximation method is presented with more data. In the worst case, we expect the RTE to converge to ε_{rel} asymptotically in the limit of infinite data. Specifically, the RTE is defined as

$$RTE = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \frac{\|\mathcal{Y}^i_{\text{test},k} - \tilde{\mathcal{Y}}^i_{\text{test},k}\|_F}{\|\mathcal{Y}^i_{\text{test},k}\|_F}, \tag{22}$$

where $\mathcal{Y}^i_{\text{test},k}$ is the i-th tensor in the test set and $\tilde{\mathcal{Y}}^i_{\text{test},k}$ is the reconstruction of $\mathcal{Y}^i_{\text{test},k}$ using the approximation $\mathbf{H}_{\mathcal{X}^k}$. Note that the RTE is averaged over individual tensors in the test set, which can lead to higher RTE values than the target ε_{rel} especially when the HT-cores are updated with batches of tensors.

The RTE is recomputed if the approximation is updated with streamed tensor. If the approximation is updated frequently, computing the RTE becomes a dominant part of the computational cost and often becomes the reason for the experiment to exceed the maximum walltime.

4.2 Scientific data results

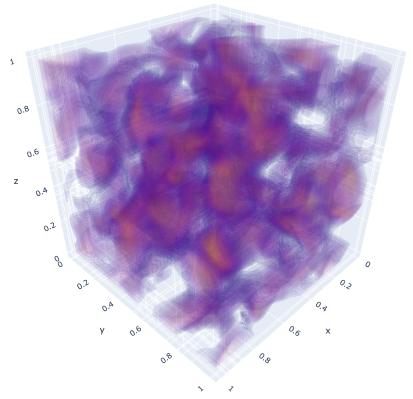
This section includes tests of `HT-RISE`'s performance to compress simulation outputs of high-dimensional PDEs.

One of the main motivations for developing `HT-RISE`is to provide a tool for scientists to analyze large-scale scientific data (Aksoy et al., 2022). In this section we compare the performance of `HT-RISE` with other incremental tensor decomposition algorithms on scientific data. We will use two different scientific datasets: PDEBench (Takamoto et al., 2022) and a dataset comprised of self-oscillating gel simulations (Alben et al., 2019).

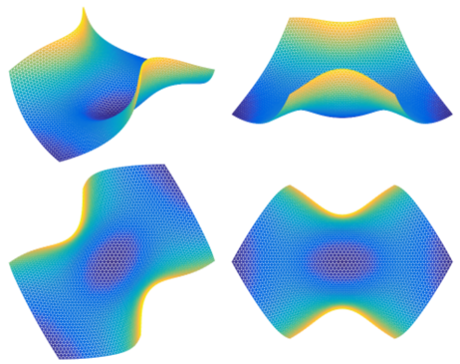
Each snapshot of simulation contains states corresponding to various physical quantities (e.g. displacement, velocity, pressure, density etc.). Therefore, we normalize each physical quantity individually similar to Aksoy et al. (2024b). In this work we consider maximum absolute value ($\mathcal{Y}_{\text{maxabs}}$), unit vector ($\mathcal{Y}_{\text{unitvec}}$), and z-score ($\mathcal{Y}_{\text{z-score}}$) normalizations in addition to compressing the unnormalized simulations. The normalizations are computed as follows:

$$\mathcal{Y}_{\text{maxabs}} = \frac{\mathcal{Y}}{\max(|\mathcal{Y}|)}, \qquad \mathcal{Y}_{\text{unitvec}} = \frac{\mathcal{Y}}{\|\mathcal{Y}\|}, \qquad \mathcal{Y}_{\text{z-score}} = \frac{\mathcal{Y} - \mu}{\sigma}, \tag{23}$$

where $\mathcal{Y}$ is the original tensor, and μ, σ are the mean and the standard deviation of the entries of $\mathcal{Y}$. Note that we do not learn a set of normalization parameters that encompasses the entire accumulation. Instead, we treat each simulation in the training set individually and normalize their states separately.



(a) *3D compressible Navier-Stokes simulations from PDEBench dataset*



(b) *Snapshots from self-oscillating gel simulations*

Figure 5: *Example snapshots from scientific datasets.*

4.2.1 Self-oscillating gel simulations

The first scientific dataset arises from solutions of a parametric PDE that simulates the motion of a hexagonal sheet of self-oscillating gels. This dataset is used both in donwstream learning tasks such as inverse design (Aksoy et al., 2022) as well as in experiments of incremental tensor decompositions (Aksoy et al., 2024a) as a benchmark dataset.

The motion of the gel is governed by the following time-dependent parametric PDE (Alben et al., 2019)

$$\mu \frac{\partial r}{\partial t} = f_s\left(r, \mathbf{K_s}, \eta\right) + f_B\left(r\right), \quad \eta\left(x, y, t, \mathbf{A}, \mathbf{k}\right) = 1 + \mathbf{A} \sin\left(2\pi\left(\mathbf{k}\sqrt{x^2+y^2} - t\right)\right), \tag{24}$$

where the bold terms indicate the input parameters to the forward model that define the characteristics of the excitation as well as the mechanical properties of the gel. More specifically, $\mathbf{K_s}$ denotes the stretching stiffness of the sheet, $\mathbf{k}$ determines the wavenumber of the sinusoidal excitation, and $\mathbf{A}$ determines the amplitude of the wave traveling on the sheet. Other terms governing the overdamped sheet dynamics are: internal damping coefficient μ, material coordinates $r = (x, y, z)$, stretching force f_s, bending force f_b, rest strain η, and time t.

The simulations are chaotic, but we use 10 sequential timesteps from each simulation as our data. Specifically, we seek to compress the x, y, and z coordinates of 3367 mesh nodes on a hexagonal gel sheet for 10 time snapshots as shown in Figure 5b. To summarize, the data consists of $3367 \times 3 \times 10$ tensors for *each* parameter combination that contain the coordinate information of the mesh. We refer to those output tensors as *simulations* for brevity and treat them as individual incremental units. To increase the dimensionality of the data, we reshape the 3-dimensional original tensor into $7 \times 13 \times 37 \times 3 \times 10$ and accumulate as batches of single simulation trajectories along an auxiliary 6-th dimension.

For this dataset, we have separate training and test sets. To construct the training set, we uniformly discretize the 3-dimensional parameter space into 20 values along each dimension and use their cross product to obtain $20 \times 20 \times 20 = 8000$ unique parameter combinations and then simulate each of those parameter combinations using the approach in Alben et al. (2019). For the test set, randomly sample from the parameter space to obtain 15,000 unique parameter combinations and simulate each of those parameter combinations. We use the training set to update the approximation and the test set to evaluate the generalization performance. Since the selection of the training set is not random, we execute each experiment only once and report the performance. We set the maximum wall time for this experiment to 2 days.

We repeat the experiments for two relative error tolerances: $\varepsilon_{rel} = 0.10$, and $\varepsilon_{rel} = 0.01$. Figure 6 shows the best results for compression ratio and reduction ratio from the selected normalization methods for both ε_{rel} settings. Figure 7 shows the best results for compression time and test error for both ε_{rel} settings. More detailed results including different normalization methods are presented in Table 2 below and Figures D.12 to D.15 in Appendix D.

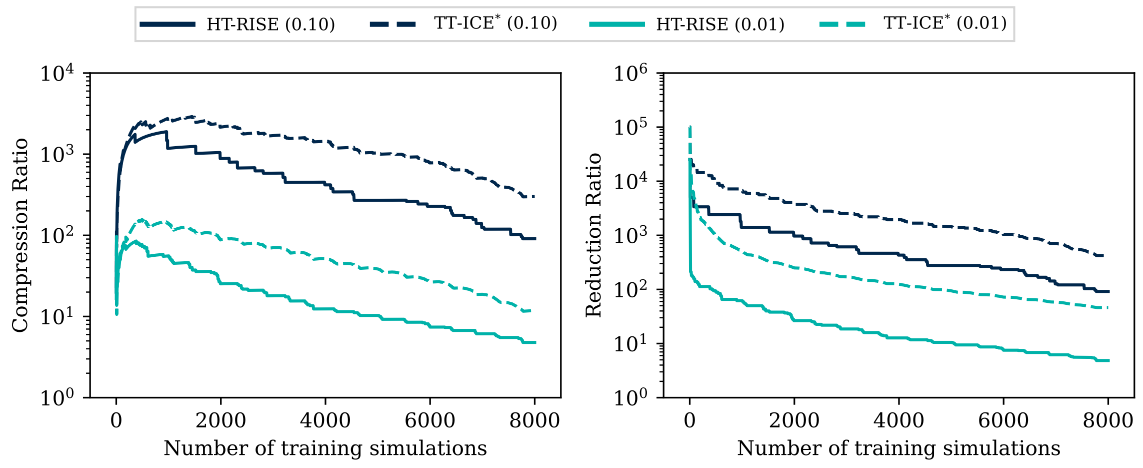


Figure 6: *Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the self-oscillating gel dataset.* `TT-ICE`* *offers* $2.5 - 3.3\times$ *the CR and* $4.6 - 9.5\times$ *the RR of* `HT-RISE`*. For more detailed comparisons, please refer to Figure D.12 for experiments with* $\varepsilon_{rel} = 0.10$ *and Figure D.14 for experiments with* $\varepsilon_{rel} = 0.01$.

Figure 6 shows the results of experiments without using any normalization. `HT-RISE` at $\varepsilon_{rel} = 0.10$ achieves a CR of 90.70 and RR of 91.82 whereas `TT-ICE`* achieves a CR of 300.4 and RR of 420.9. Similar to $\varepsilon_{rel} = 0.10$, `HT-RISE` at $\varepsilon_{rel} = 0.01$ achieves a lower CR ($4.79\times$) compared to `TT-ICE`* ($11.86\times$) and a lower RR ($4.85\times$) compared to `TT-ICE`*

(46.27×). This can be explained by the relatively small size and the simpler nature of the self-oscillating gel dataset. This allows `TT-ICE`* to find and exploit a low-rank structure across the tensors in the accumulation. This claim is also supported by the reduction ratio of `TT-ICE`*, which corresponds to a latent space size of 2183 (recall that the training set has 8000 simulations).

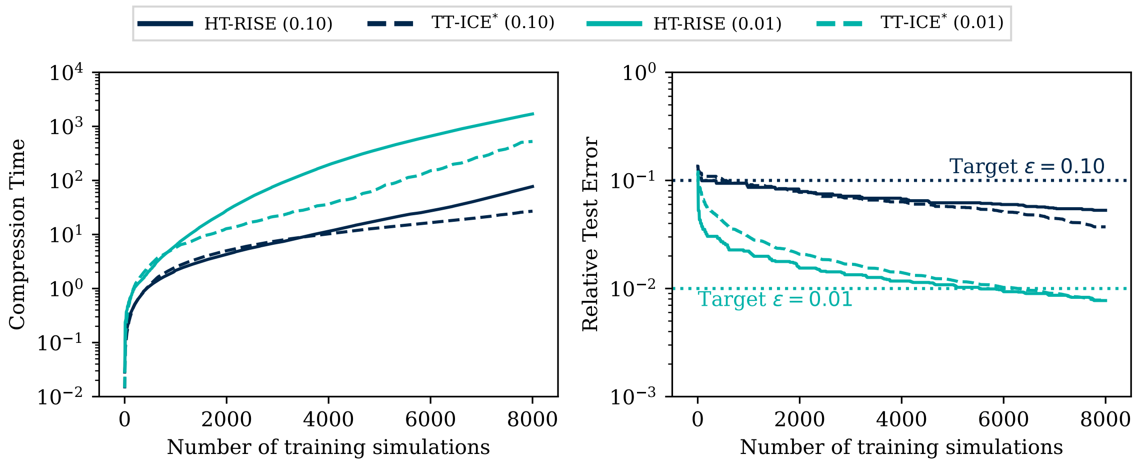


Figure 7: *Compression time (left) and Relative Test Error (right) of the algorithms on the self-oscillating gel dataset.* ***`HT-RISE`*** *takes* $2.8 - 3.2\times$ *the time it takes* ***`TT-ICE`**** *to complete compressing the stream. Neither method struggles to reduce the RTE below the target* ε *for both* $\varepsilon_{rel} = 0.10$ *and* 0.01*. For more detailed comparisons, please refer to Figure D.13 for experiments with* $\varepsilon_{rel} = 0.10$ *and Figure D.15 for experiments with* $\varepsilon_{rel} = 0.01$*.*

Figure 7 shows the results of experiments on compression time and relative test error without using any normalization. `HT-RISE` at $\varepsilon_{rel} = 0.10$ takes 77.16 seconds to compress all 8000 simulations and achieves a RTE of 0.052. On the other hand, `TT-ICE`* takes 26.99 seconds to compress all 8000 simulations and achieves a RTE of 0.037. `HT-RISE` at $\varepsilon_{rel} = 0.01$ takes 1699.8 seconds to compress all 8000 simulations and achieves a RTE of 0.007. On the other hand, `TT-ICE`* takes 526.9 seconds to compress all 8000 simulations and achieves a RTE of 0.007. `HT-RISE` takes $2.8 - 3.2\times$ the time it takes `TT-ICE`* to complete compressing the stream.

Neither method struggles to reduce the RTE below the target ε for both $\varepsilon_{rel} = 0.10$ and 0.01. This is expected as the self-oscillating gel dataset is relatively simple and has a low-dimensional structure that can be captured by both methods. This is evidenced by both algorithms driving the RTE well below the target ε. The hierarchy of dimensions introduced by the HT format does not provide a significant advantage in terms of approximation performance but becomes a hindrance in terms of computational efficiency. This is reflected as the difference in compression time between `HT-RISE` and `TT-ICE`*.

Table 2 summarizes the rest of our experiments with the self-oscillating gel dataset. The table shows the total time taken to compress the entire dataset, the compression ratio, the reduction ratio, and the mean relative approximation error over the test set. The table also shows the normalization method used for each experiment. The experiments are repeated for two relative error tolerances: $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.01$. The results further support our claim that `HT-RISE` is overcomplicated for this dataset and therefore `TT-ICE`* performs better in terms of our metrics. `TT-ICE`* consistently outperforms `HT-RISE` in terms of compression ratio, compression time, and relative test error, except for the experiments at $\varepsilon_{rel} = 0.01$ using unit vector and z-score normalizations. At those experiments `TT-ICE`* runs into maximum walltime issues.

Table 2: *Summary of the compression experiments with the self-oscillating gel dataset. Norm: method of normalization, Algorithm: the incremental tensor decomposition algorithm, #Sims: number of simulations compressed, Comp. Time: total time in seconds, CR: compression ratio, RR: reduction ratio, RTE: mean relative test error over the test set. ‡ indicates that the experiment did not complete due to a timeout, † indicates that the experiment did not complete due to running out of memory.*

ε_{rel}	Norm	Algorithm	#Sims	Comp. Time (s)	CR	RR	RTE
0.10	None	HT-RISE	8000	77.16	90.70	91.82	0.052
		TT-ICE*	8000	26.99	300.4	420.9	0.037
	UnitVec	HT-RISE	8000	203.2	35.45	36.16	0.059
		TT-ICE*	8000	39.29	124.9	220.1	0.059
	Z-score	HT-RISE	8000	226.9	33.67	34.35	0.066
		TT-ICE*	8000	41.83	117.6	215.4	0.058
0.01	None	HT-RISE	8000	1699.8	4.79	4.85	0.007
		TT-ICE*	8000	526.9	11.86	46.27	0.007
	UnitVec	HT-RISE	8000	1929.2	4.11	4.17	0.012
		TT-ICE*	5100‡	119.4	27.15	70.34	0.020
	Z-score	HT-RISE	8000	3029.1	3.03	3.08	0.098
		TT-ICE*	5086‡	177.1	16.69	67.25	0.022

4.2.2 PDEBench 3D Navier-Stokes simulations

Next we compare the performance of `HT-RISE`with other incremental tensor decomposition algorithms on a more challenging scientific dataset. PDEBench dataset (Takamoto et al., 2022) is a benchmark suite for scientific machine learning tasks. It provides diverse datasets with distinct properties based on 11 well-known time-dependent and time-independent PDEs.

From the PDEBench dataset, we use the 3D compressible Navier-Stokes simulations with $M = 1.0$ and turbulent initial conditions. The data consists of 3 velocity (v_x, v_y, v_z), pressure, and density fields at 21 time steps for 600 different initial conditions. The simulation space is dicretized into a $64 \times 64 \times 64$ grid. The compressible Navier stokes equations that are used to generate the dataset are:

$$\begin{aligned} \partial_t \rho + \nabla \cdot (\rho \mathbf{v}) &= 0, \\ \rho \left(\partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v} \right) &= -\nabla p + \eta \Delta \mathbf{v} + \left(\zeta + \frac{\eta}{3} \right) \nabla (\nabla \cdot \mathbf{v}), \\ \partial_t (\epsilon + \rho \nu^2) + \nabla \cdot \left[(p + \epsilon + \frac{\rho \nu^2}{2}) \mathbf{v} - \mathbf{v} \cdot \sigma' \right] &= \mathbf{0}, \end{aligned} \tag{25}$$

where ρ is the mass density, $\mathbf{v}$ is the fluid velocity, p is the gas pressure, ϵ is the internal energy, σ' is the viscous stress tensor, and η and ζ are shear and bulk viscosities, respectively. Each simulation from the dataset creates a 5-dimensional tensor of size $64 \times 64 \times 64 \times 5 \times 21$. This results in simulations that are $\sim 270\times$ larger than the self-oscillating gel simulations.

To increase the dimensionality of the dataset, we reshape each simulation into a 8-dimensional tensor of shape $8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 5 \times 21$ and accumulate in batches of single simulation trajectory along an auxiliary 9-th dimension.

We randomly split 600 simulations into training and test sets with a ratio of 80% and 20%, respectively. We use the training set to update the approximation and the test set to evaluate the generalization performance. Since the selection of the training set is random, we repeat each experiment with five different seeds and report the average performance. We set the maximum wall time for each experiment to four days. We repeat the experiments for two relative error tolerances: $\varepsilon_{rel} = 0.10$, and $\varepsilon_{rel} = 0.05$. Figure 8 shows the best results for compression ratio and reduction ratio from the selected normalization methods for both ε_{rel} settings. Figure 9 shows the best results for compression time and test error for both ε_{rel} settings. More detailed results are presented in Table 3 below and Figures D.8 to D.11 in Appendix D.

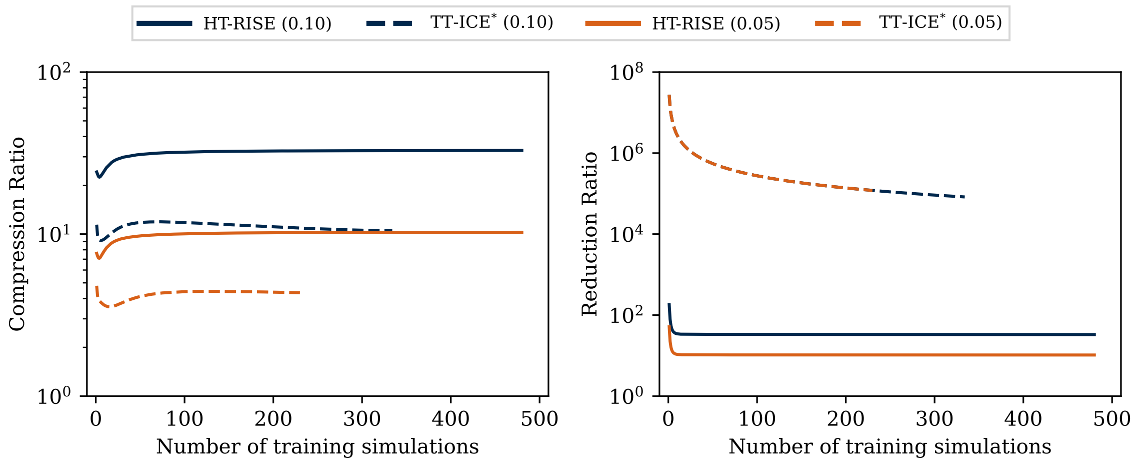


Figure 8: *Compression ratio (CR - left) and reduction ratio (RR- - right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes dataset.* `HT-RISE` *offers* $2.4 - 3.1\times$ *the CR of* `TT-ICE`* *but results in orders of magnitude lower RR.* `TT-ICE`* *does not complete the entire stream due to maximum walltime timeout whereas* `HT-RISE` *successfully completes the task. The results are averaged over 5 seeds. For more detailed comparisons, please refer to Figure D.8 for experiments with* $\varepsilon_{rel} = 0.10$ *and Figure D.10 for experiments with* $\varepsilon_{rel} = 0.05$.

Figure 8 shows the results of experiments without using any normalization. `HT-RISE` at $\varepsilon_{rel} = 0.10$ achieves a CR of 32.83 and RR of 33.09 whereas `TT-ICE`* achieves a CR of 10.49 and RR of 82,658. Note that `TT-ICE`* runs into maximum walltime issues at this experiment and is only able to compress 333 simulations out of 480. Similar to $\varepsilon_{rel} = 0.10$, `HT-RISE` at $\varepsilon_{rel} = 0.05$ achieves a higher CR ($10.2\times$) compared to `TT-ICE`* ($4.34\times$) but a lower RR ($10.34\times$) compared to `TT-ICE`* ($119,674\times$). This time `TT-ICE`* runs into maximum walltime issues and is only able to compress 229 simulations out of 480. The orders of magnitude discrepancy between the RR of `TT-ICE`* and `HT-RISE` is caused by the fact that the size of the latent space representation is upper bounded by the number of tensors in the accumulation for `TT-ICE`*. As `HT-RISE` does not have such a limit, the latent space grows in parallel to the complexity of the streamed data. `TT-ICE`* hitting the latent space upper bound further indicates that `TT-ICE`* is not able to find a low-rank structure *across* the tensors in the accumulation in contrast to self-oscillating gels dataset.

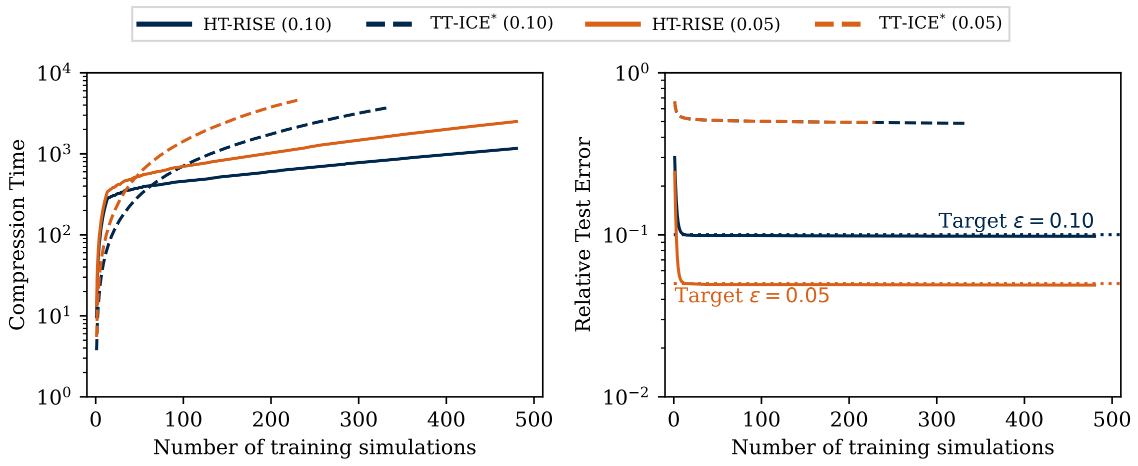


Figure 9: *Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes dataset. Until `TT-ICE`* hits the maximum walltime limit, `TT-ICE`* takes $1.8 - 3.2\times$ of the time it takes `HT-RISE` to complete compressing the stream. In addition, `TT-ICE`* struggles to reduce the RTE below the target ε levels in both cases whereas `HT-RISE` reduces the RTE below the ε within 15 simulations. Recall that `TT-ICE`* does not complete the entire stream due to maximum walltime timeout whereas `HT-RISE` successfully completes the task. The results are averaged over 5 seeds. For more detailed comparisons, please refer to Figure D.9 for experiments with $\varepsilon_{rel} = 0.10$ and Figure D.11 for experiments with $\varepsilon_{rel} = 0.05$.*

Figure 9 shows the results of experiments on compression time and relative test error without using any normalization. `HT-RISE` at $\varepsilon_{rel} = 0.10$ takes 1165.90 seconds to compress all 480 simulations and achieves a RTE of 0.098. On the other hand, `TT-ICE`* takes 3727.38 seconds to compress 333 simulations and achieves a RTE of 0.487. `HT-RISE` at $\varepsilon_{rel} = 0.05$ takes 2505.58 seconds to compress all 480 simulations and achieves a RTE of 0.049. On the other hand, `TT-ICE`* takes 4581.78 seconds to compress 229 simulations and achieves a relative test error of 0.493. `TT-ICE`* struggles to reduce the RTE below the target ε levels in both cases whereas `HT-RISE` reduces the relative test error below the ε within the first 15 simulations. This observation further supports the claim that `TT-ICE`* is not able to find a generalized low-rank structure across the tensors in the accumulation. As a result of its hierarchical structure, `HT-RISE` discovers a low-rank structure across the tensors in the accumulation and is able to generalize well to unseen data. As a result, `HT-RISE` requires much fewer updates to the approximation to achieve the target ε than `TT-ICE`*. This is also reflected in the compression time as both `HT-RISE` curves taper off as soon as the RTE falls below their respective ε, whereas curves for `TT-ICE`* continue to increase steadily. Frequent updates of TT-ICE* is also the reason why TT-ICE* runs into maximum walltime issues, as each update is succeeded with RTE computation over the *entire* test set.

Table 3 summarizes the rest of our experiments with the PDEBench 3D turbulent Navier-Stokes dataset. The table shows the total time taken to compress the entire dataset, the compression ratio, the reduction ratio, and the RTE. The table also shows the normalization method used for each experiment. The experiments are repeated for two relative error tolerances: $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.05$. The results show that HT-RISE consistently outperforms TT-ICE* in terms of compression ratio, compression time, and RTE. HT-RISE also completes the task within the maximum walltime limit (4 days) in all experiments (except for Z-score normalization at $\varepsilon_{rel} = 0.05$, where it runs into memory issues), whereas TT-ICE* runs into maximum walltime issues in all experiments.

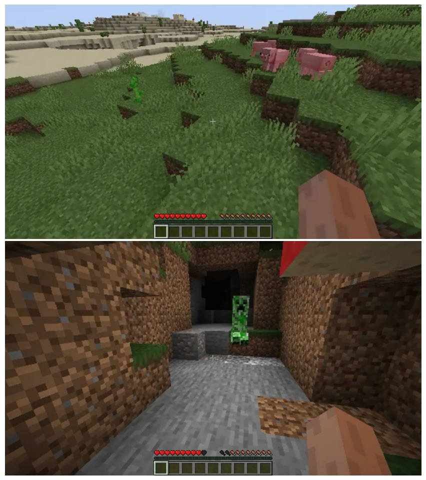
Table 3: *Summary of the compression experiments with the PDEBench 3D turbulent Navier-Stokes dataset. The results are averaged over 5 seeds. Norm: method of normalization, Algorithm: the incremental tensor decomposition algorithm, #Sims: number of simulations compressed, Comp. Time: total time in seconds, CR: compression ratio, RR: reduction ratio, RTE: mean relative test error over the test set. Please refer to Figures D.8 and D.9 for experiments with* $\varepsilon_{rel} = 0.10$ *and Figures D.10 and D.11 for experiments with* $\varepsilon_{rel} = 0.05$. *MaxAbs: Maximum absolute value normalization, None: No normalization, UnitVec: Unit vector normalization, Z-score: Z-score normalization. ‡ indicates that the experiment did not complete due to a timeout, † indicates that the experiment did not complete due to running out of memory.*

ε_{rel}	Norm	Algorithm	#Sims	Comp. Time (s)	CR	RR	RTE
0.10	MaxAbs	HT-RISE	480	1706.84	19.73	19.89	0.087
		TT-ICE*	292‡	3824.33	7.94	94,264	0.609
	None	HT-RISE	480	1165.90	32.83	33.09	0.098
		TT-ICE*	333‡	3727.38	10.49	82,658	0.487
	UnitVec	HT-RISE	480	1863.73	16.80	16.92	0.096
		TT-ICE*	267‡	3989.26	6.72	102,705	0.762
	Z-score	HT-RISE	480	3007.92	9.08	9.20	0.097
		TT-ICE*	213‡	4287.14	3.97	128,622	0.962
0.05	MaxAbs	HT-RISE	480	3594.18	7.08	7.18	0.044
		TT-ICE*	194‡	4462.31	3.53	141,154	0.617
	None	HT-RISE	480	2505.58	10.26	10.34	0.049
		TT-ICE*	229‡	4581,78	4.34	119,674	0.493
	UnitVec	HT-RISE	480	3911.02	6.26	6.30	0.048
		TT-ICE*	177‡	4503.89	2.93	154,635	0.771
	Z-score	HT-RISE	477†	7122.89	3.18	3.19	0.048
		TT-ICE*	144‡	4877.19	1.82	189,828	0.970

Summmary of conclusions from experiments with scientific data: The experiments with scientific data demonstrate that HT-RISE provides superior generalization performance compared to TT-ICE* across both datasets. HT-RISE consistently reduces the RTE below the target ε levels within fewer simulations. TT-ICE* results in significantly higher RR due to the properties of the TT-format across both datasets but falls short in CR when the dataset becomes large. The importance of normalization for scientific data is also demonstrated in Tables 2 and 3, showing that it can significantly affect the performance of the compression algorithms. Overall, the results suggest that HT-RISE is more suitable for scientific datasets with complex multi-scale structures, while `TT-ICE`* may be more appropriate for simpler datasets where the low-rank structure can be effectively captured.

4.3 Image data

In addition to the scientific data, we will also compare the algorithms on image data. The image data will be obtained from two different sources: The MineRL Basalt competition dataset (Milani et al., 2024) and multispectral images from the BigEarthNet dataset (Sumbul et al., 2019, 2021).


(a) *MineRL Basalt competition dataset*

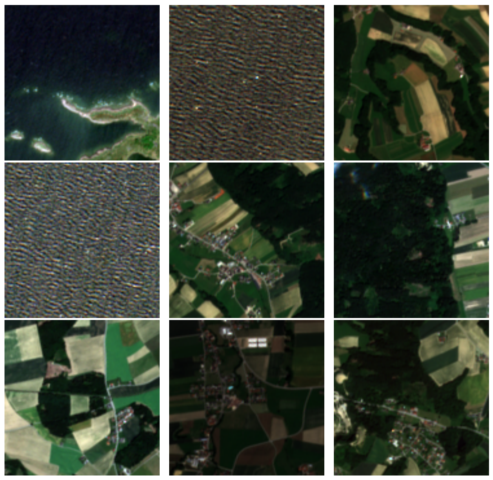

(b) *Sentinel2 images (RGB channels only)*

Figure 10: *Example frames from image-based datasets.*

4.3.1 Minecraft frames

First person games are of interest for inverse reinforcement learning purposes. The MineRL Basalt competition dataset (Milani et al., 2024) contains gameplay episodes from the game Minecraft and is designed for the purpose of training agents to perform tasks in the game. The dataset contains gameplay episodes from 4 different tasks, which are designed to be challenging for reinforcement learning agents. Each episode is an end-to-end gameplay video that starts at the corresponding initial setting for each scenario and ends with successful completion of the assignment. Therefore, each episode varies in duration. The original videos have a resolution of 640×360 pixels and a frame rate of 20 frames per second.

The MineRL dataset contains demonstrations for the following 4 tasks:

- **Find Cave:** Look around for a cave. When you are inside one, press ESCAPE to end the minigame.

- **Make Waterfall:** After spawning in a mountainous area with a water bucket and various tools, the agent builds a waterfall and then repositions themself to "take a scenic picture" of the same waterfall.

- **Build Animal Pen:** After spawning in a village, the agent builds an animal pen next to one of the houses in a village using fence posts from inventory to build one animal pen that contains at least two of the same animal.

- **Build Village House:** Taking advantage of the items in the inventory, the agent builds a new house in the style of the village (random biome), in an appropriate location (e.g. next to the path through the village), without harming the village in the process. Then gives a brief tour of the house (i.e. spin around slowly such that all of the walls and the roof are visible).

The dataset contains approximately 20,000 gameplay episodes in total. This translates to approximately 160 GB of gameplay epsiodes *per task*. In this study we only focus on the *Find Cave* task. The *Find Cave* task has 5466 gameplay episodes, totaling a size of 183GB on disk. Similar to Baker et al. (2022), we downsample the 640×360 frames to 128×128. This results in a 4-dimensional dataset $128 \times 128 \times 3 \times N_{frames}$, where N_{frames} is the number of frames in a video. Furthermore, we increase the dimensionality of the dataset by reshaping the original frames into 9-dimensional tensor of size $2 \times 4 \times 4 \times 4 \times 8 \times 8 \times 2 \times 3 \times N_{frames}$ and accumulate them as batches of 20 along the 9-th dimension, i.e. $N_{frames} = 20$.

We select 17 videos from the dataset as test set at random and use the rest for training. Since the size of the dataset exceeds the allocated memory for each experiment by almost a factor of three, we do not expect the algorithms to compress all 5466 videos. This experiment aims to push the algorithms to their limits given limited computational and hardware resources. We repeat the experiments for three relative error tolerances: $\varepsilon_{rel} = 0.30$, $\varepsilon_{rel} = 0.20$, and $\varepsilon_{rel} = 0.10$. The maximum wall time for this experiment is set to two days. Since the selection of the test set is random, we execute each experiment five times and report the average performance. The results are summarized in Table 4 and are presented in detail in Figures 11 and 12.

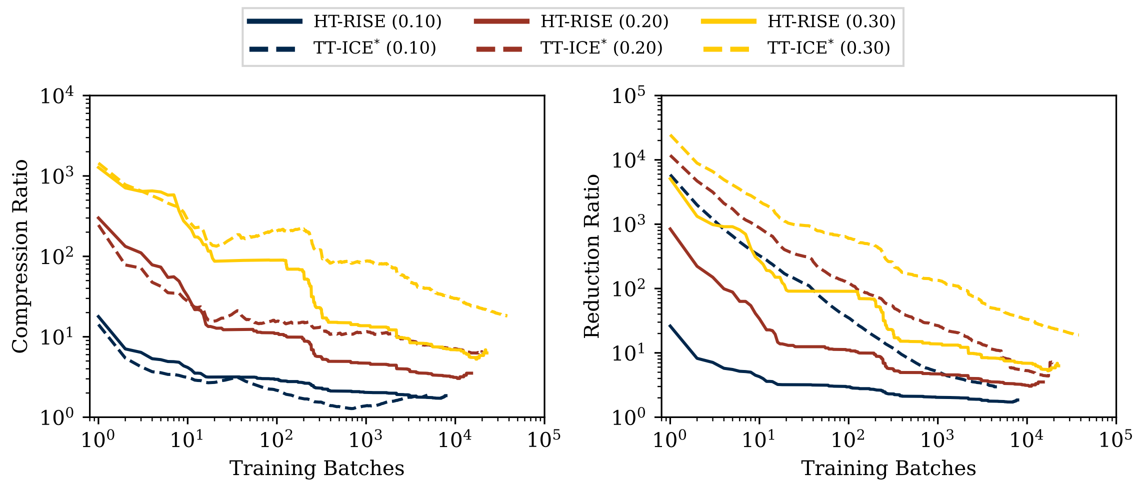


Figure 11: *Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the Basalt MineRL dataset.*`TT-ICE*` *offers* $1 - 2.9\times$ *the CR and* $1.6 - 3\times$ *the RR of* `HT-RISE`*. The results are averaged over 5 seeds.*

Figure 11 shows the results of experiments on compression ratio and reduction ratio for the MineRL Basalt competition dataset. `HT-RISE` at $\varepsilon_{rel} = 0.30$ achieves a CR of 6.29 and RR of 6.29 whereas `TT-ICE`* achieves a CR of 18.3 and RR of 18.9. Similar to $\varepsilon_{rel} = 0.30$, `HT-RISE` at $\varepsilon_{rel} = 0.20$ achieves a lower CR ($3.53\times$) compared to `TT-ICE`* ($6.45\times$) and a lower RR ($3.53\times$) compared to `TT-ICE`* ($7.13\times$). This trend continues for $\varepsilon_{rel} = 0.10$ where `HT-RISE` achieves a lower CR ($1.85\times$) compared to `TT-ICE`* ($1.88\times$) and a lower RR ($1.85\times$) compared to `TT-ICE`* ($2.96\times$).

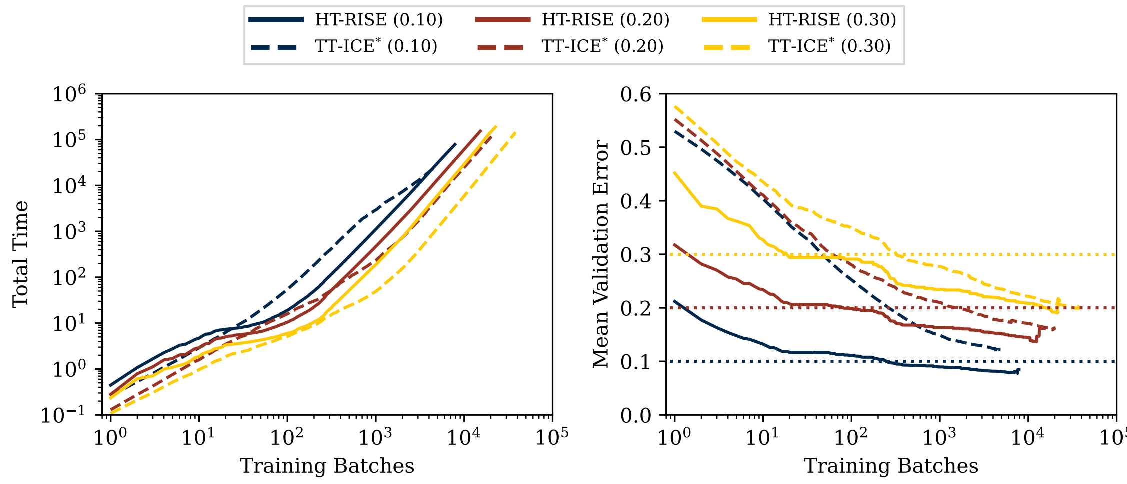


Figure 12: *Compression time (left) and Relative Test Error (right) of the algorithms on the Basalt MineRL competition dataset. Each target ε level is shown with dashed lines of their corresponding color.* `HT-RISE` *takes* $2.8-3.2\times$ *the time it takes* `TT-ICE`* *to complete compressing the stream. Only* `TT-ICE`* *at* $\varepsilon_{rel} = 0.1$ *struggles to reduce the RTE below the target* ε *threshold. The results are averaged over 5 seeds.*

Figure 12 shows the results of experiments on compression time and relative test error for the MineRL Basalt competition dataset. `HT-RISE` at $\varepsilon_{rel} = 0.30$ takes $188,405$ seconds to compress 22,625 batches (449k frames - 375 videos) and achieves a RTE of 0.208. On the other hand, `TT-ICE`* takes $143,873$ seconds to compress 38,146 batches (757k frames - 635 videos) and achieves a RTE of 0.202. `HT-RISE` at $\varepsilon_{rel} = 0.20$ takes $153,146$ seconds to compress 15,222 batches (302k frames - 252 videos) and achieves a RTE of 0.166. On the other hand, `TT-ICE`* takes $116,418$ seconds to compress 20,350 batches (404k frames - 339 videos) and achieves a RTE of 0.161. `HT-RISE` at $\varepsilon_{rel} = 0.10$ takes $78,286$ seconds to compress 7863 batches (156k frames - 127 videos) and achieves a RTE of 0.083. On the other hand, `TT-ICE`* takes $26,756$ seconds to compress 4861 batches (96k frames - 80 videos) and achieves a RTE of 0.119. Only `TT-ICE`* at $\varepsilon_{rel} = 0.10$ struggles to reduce the RTE below the target ε threshold. The trend in Figure 12 suggests that this issue pertains to the lack of sufficient training data for `TT-ICE`* to learn a generalizable approximation and should be resolved once the algorithm is presented with further gameplay episodes. Another important takeaway from Figure 12 is that for $\varepsilon_{rel} = 0.20$ and $\varepsilon_{rel} = 0.30$, `HT-RISE` reduces the RTE below the target ε threshold within significantly less training batches compared to `TT-ICE`*. `HT-RISE` crosses the RTE threshold in 82 and 19 batches, whereas `TT-ICE`* crosses the RTE threshold in 1597 and 322 batches for $\varepsilon_{rel} = 0.20$ and $\varepsilon_{rel} = 0.30$, respectively. This suggests that `HT-RISE` generalizes better with less training data compared to `TT-ICE`* regardless of the target ε level.

One important thing to note from Table 4 is that except for $\varepsilon_{rel} = 0.10$, `TT-ICE`* compresses more frames than `HT-RISE`. Furthermore, in contrast to `HT-RISE`, `TT-ICE`* runs into walltime issues rather than memory issues for all target ε levels. This suggests that `TT-ICE`* is more memory efficient than `HT-RISE`. This is no surprise as the number of tensors in the accumulation goes to extreme levels (+100k), the RR dominates the overall cost of storing the compressed representations in memory. Since `TT-ICE`* has a 1-dimensional (i.e. a vector) latent space as opposed to the 2-dimensional (i.e. a matrix) latent space of `HT-RISE`, the growth of the latent space is slower for `TT-ICE`* compared to `HT-RISE`.

Table 4: *Summary of the compression experiments with the MineRL Basalt competition dataset. The results are averaged over 5 seeds. Algorithm: the incremental tensor decomposition algorithm, #Batches: number of batches compressed, Comp. Time: total time in seconds, CR: compression ratio, RR: reduction ratio, RTE: mean relative test error over the test set. ‡ indicates that the experiment did not complete due to a timeout, † indicates that the experiment did not complete due to running out of memory.*

ε_{rel}	Algorithm	#Batches	Comp. Time	CR	RR	RTE
0.30	HT-RISE	22,625†‡	188,405	6.29	6.29	0.208
	TT-ICE*	38,146‡	143,873	18.3	18.9	0.202
0.20	HT-RISE	15,222†	153,146	3.53	3.53	0.166
	TT-ICE*	20,350‡	116,418	6.45	7.13	0.161
0.10	HT-RISE	7863†	78,286	1.85	1.85	0.083
	TT-ICE*	4861‡	26,756	1.88	2.96	0.119

4.3.2 Multispectral images

Another natural occurence of high-dimensional data in the form of images is multispectral satellite imagery. The BigEarthNet dataset (Sumbul et al., 2021, 2019) consists of 590,326 image patches, each of which is a 120x120x12 pixel multispectral image. The dataset has a size of 66GB. The images are taken from the Sentinel-2 satellite and contain 12 spectral bands. Table 5 lists the spectral bands and their wavelengths. The images are taken from 125 different locations around the world. The dataset is originally designed for land cover classification and land use analysis. Therefore, the images are labeled with 43 different land cover classes. As we do not aim to showcase the use cases of the latent space learned through incremental tensor decompositions in this work, we do not consider the labels of the images in any part of the compression experiments.

From the 590,326 images, we select 23,602 images at random as the test set and use the rest for training. We repeat the experiments for two relative error tolerances: $\varepsilon_{rel} = 0.30$ and $\varepsilon_{rel} = 0.10$. The maximum wall time for this experiment is set to 2 days. Since the selection of the test set is random, we execute each experiment 5 times and report the average performance. The results are summarized in Table 6 and are presented in detail in Figures D.16 and D.17. To increase the dimensionality of the dataset, we reshape the

Table 5: *Names and wavelengths of spectral bands of Sentinel2 images. SWIR: Short Wave Infrared, NIR: Near Infrared. The bands are ordered in increasing wavelength.*

Band Name	Central Wavelength (nm)	Bandwidth (mm)	Description
B01	442.7	21	Coastal aerosol
B02	492.4	66	Blue
B03	559.8	36	Green
B04	664.6	31	Red
B05	704.1	15	Vegetation Red Edge 1
B06	740.5	15	Vegetation Red Edge 2
B07	782.8	20	Vegetation Red Edge 3
B08	832.8	106	NIR
B08A	864.7	21	Narrow NIR
B09	945.1	20	Water Vapor
B11	1613.7	91	SWIR 1
B12	2202.4	175	SWIR 2

original multispectral images into 5-dimensional tensors of size $12 \times 10 \times 12 \times 10 \times 12$ and accumulate them as batches of 100 along an auxiliary 6-th dimension.

Table 6: *Summary of the compression experiments with the BigEarthNet multispectral satellite imagery dataset. The results are averaged over 5 seeds. Algorithm: the incremental tensor decomposition algorithm, #Batches: number of batches compressed, Comp. Time: total time in seconds, CR: compression ratio, RR: reduction ratio, RTE: mean relative test error over the test set. ‡ indicates that the experiment did not complete due to a timeout, † indicates that the experiment did not complete due to running out of memory.*

ε_{rel}	Algorithm	#Batches	Comp. Time	CR	RR	RTE
0.30	HT-RISE	5668	2482.3	1154.3	962.04	0.242
	TT-ICE*	5668	6239.6	2274.0	1901.1	0.260
0.15	HT-RISE	5668	14,896	91.45	76.21	0.149
	TT-ICE*	5668	18,866	117.9	100.8	0.133
0.10	HT-RISE	5668	49,408	32.68	27.24	0.104
	TT-ICE*	5668‡	44,687	35.31	31.03	0.088
0.05	HT-RISE	1781†	20,880	7.29	6.07	0.054
	TT-ICE*	129‡	1720.0	3.59	19.76	0.069

Table 6 shows that both algorithms are able to completely compress the entire dataset, which exceeds the allocated memory for the experiments, without running into any memory issues for target relative error thresholds $\varepsilon_{rel} = 0.30 - 0.10$. Three out of five experiments for `TT-ICE`* at $\varepsilon_{rel} = 0.10$ and five out of five experiments at $\varepsilon_{rel} = 0.05$ did not complete due to a timeout. `HT-RISE` at $\varepsilon_{rel} = 0.05$ runs out of the allocated memory after compressing 1781 batches (∼178,000 images).

For higher relative error thresholds, `TT-ICE`* achieves higher CR and RR than `HT-RISE` but takes a longer time. At $\varepsilon_{rel} = 0.30$, `HT-RISE` achieves a CR of 1154.3× and a RR of 962.04×

in 2482 seconds, whereas TT-ICE* surpasses HT-RISE with a CR of 2274.0$\times$ and a RR of 1901.1$\times$ in 6239 seconds. The results at $\varepsilon_{rel} = 0.15$ show that TT-ICE* compresses the training dataset in 18,866 seconds, which is 1.27$\times$ the time it takes HT-RISE to compress the same data. However, TT-ICE* achieves a CR of 117.9$\times$ and a RR of 100.8$\times$ in contrast to HT-RISE with a CR of 91.5$\times$ and a RR of 76.2$\times$.

At $\varepsilon_{rel} = 0.10$, TT-ICE* compresses the training dataset in 44,687 seconds and achieves a CR of 35.31$\times$ and a RR of 31.03$\times$. HT-RISE compresses the same data in 49,408 seconds and achieves a CR of 32.68$\times$ and a RR of 27.24$\times$. Note that despite the compression time for HT-RISE at $\varepsilon_{rel} = 0.10$ is longer than TT-ICE*, HT-RISE successfully completes the compression task within the allocated maximum walltime. This indicates that TT-ICE* performs significantly more updates to its approximation compared to HT-RISE and therefore spends most of the allocated walltime to compute the RTE. One interesting thing to note here is that the RTE of HT-RISE is above the target $\varepsilon_{rel} = 0.10$. The updates are computed for batches of 100 images but the RTE is computed over single images in the test set. This discrepancy in the batch size used for updates and the batch size used for computing the RTE is likely the reason for the high RTE.

At $\varepsilon_{rel} = 0.05$, HT-RISE compresses 1781 batches in 20,880 seconds and achieves a CR of 7.29$\times$ and a RR of 6.07$\times$. TT-ICE* compresses 129 batches in 1720 seconds and achieves a CR of 3.59$\times$ and a RR of 19.76$\times$. Note that at this target ε level, TT-ICE* runs into a timeout issue and does not complete the compression task. On the other hand, HT-RISE runs into memory issues for all five seeds and does not complete the compression task. Further quantitative comparisons are provided in Figures D.16 and D.17 in Appendix D.6.

Figures 13 and 14 present a qualitative comparison for the reconstructions of the multispectral images from the BigEarthNet dataset using ε_{rel} between 0.05 and 0.30. Images in Figure 13 are generated by reconstructing latent space representations of a training image learned during the compression process. Images in Figure 14 are generated by reconstructing latent space representations of a test image that was not seen during the compression process. This is done by first projecting the unseen multispectral image onto the cores of the corresponding tensor networks and then reconstructing the image from the projection. The images are displayed in RGB format, where only the bands B02, B03, and B04 are used.

Figure 13 shows that the quality of the reconstructed images decreases as the target ε level increases. Specifically beyond $\varepsilon_{rel} = 0.10$, the images become significantly pixelated and lose their original features. For $\varepsilon_{rel} = 0.05$ and 0.10, HT-RISE reconstructs sharper images compared to TT-ICE*. This becomes evident especially at the finer details such as roads. HT-RISE at $\varepsilon_{rel} = 0.05$ results in the best reconstruction. This reconstruction has slight discoloration in comparison to the original image and maintains the fine features well.

To demonstrate the generalization performance qualitatively, we present the reconstructions of the test images in Figure 14. Similar to training data, the quality of the reconstructed images decreases as the target ε level increases. However, this time the difference between HT-RISE and TT-ICE* is more pronounced. For $\varepsilon_{rel} = 0.05$ and 0.10, TT-ICE* introduces a significant amount of noise in the images which reduces the image quality. While offering a better reconstruction quality at all target ε levels, HT-RISE at $\varepsilon_{rel} = 0.05$ results in the best reconstruction. In parallel with our findings with the training images, the

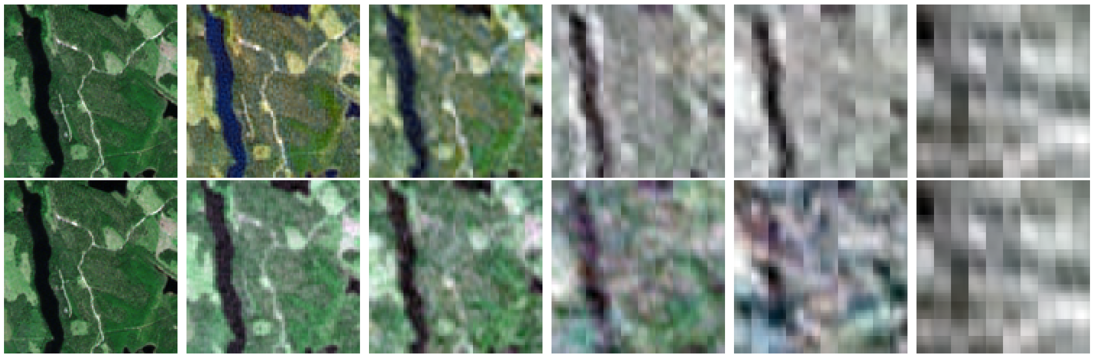

(a) Original
(b) $\varepsilon_{rel} = 0.05$
(c) $\varepsilon_{rel} = 0.10$
(d) $\varepsilon_{rel} = 0.15$
(e) $\varepsilon_{rel} = 0.20$
(f) $\varepsilon_{rel} = 0.30$

Figure 13: *Reconstructed multispectral images from BigEarthNet's training set, captured by the Sentinel-2 satellite, are displayed. Images compressed using* `HT-RISE` *are in the top row, while those compressed with* `TT-ICE`* *appear in the bottom row. Different columns represent reconstructions at varying ε target levels, from $\varepsilon_{rel} = 0.05$ to $\varepsilon_{rel} = 0.30$. Image quality decreases severely after $\varepsilon_{rel} = 0.10$ for both algorithms. For $\varepsilon_{rel} = 0.05$ and 0.10,* `HT-RISE` *results in shaper images compared to* `TT-ICE`*. *These images have been reconstructed from their latent representations, with only the RGB bands (B02, B03, B04) being depicted.*

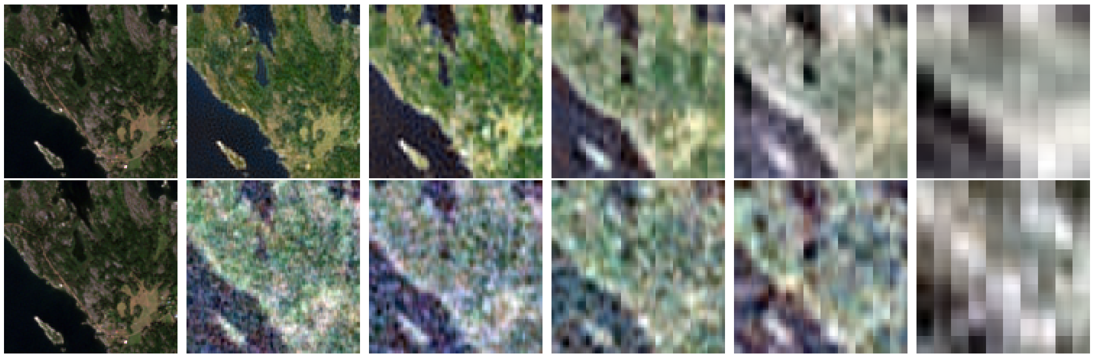

(a) Original
(b) $\varepsilon_{rel} = 0.05$
(c) $\varepsilon_{rel} = 0.10$
(d) $\varepsilon_{rel} = 0.15$
(e) $\varepsilon_{rel} = 0.20$
(f) $\varepsilon_{rel} = 0.30$

Figure 14: *Reconstructed multispectral images from BigEarthNet's test set, captured by the Sentinel-2 satellite, are displayed. Images compressed using* `HT-RISE` *are in the top row, while those compressed with* `TT-ICE`* *appear in the bottom row. Different columns represent reconstructions at varying ε target levels, from $\varepsilon_{rel} = 0.05$ to $\varepsilon_{rel} = 0.30$. Image quality decreases severely after $\varepsilon_{rel} = 0.15$ for* `HT-RISE` *and after $\varepsilon_{rel} = 0.10$ for* `TT-ICE`*. *For all ε_{rel} levels,* `HT-RISE` *results in shaper images compared to* `TT-ICE`*. *These images have been reconstructed from their latent representations, with only the RGB bands (B02, B03, B04) being depicted.*

reconstruction has slight discoloration in comparison to the original image and maintains the fine features well.

Summmary of conclusions from experiments with image data: Similar to the experiments with scientific data, the experiments with image data also demonstrate that `HT-RISE` provides superior generalization performance compared to `TT-ICE`* across both investigated datasets. `HT-RISE` consistently achieves lower RTE at all ε levels within fewer frames and compresses in less time than `TT-ICE`*. `TT-ICE`* starts off with higher RR due to the properties of the TT-format but the RRs of both algorithms become comparable towards the end of the experiments, regardless of the target ε level. One interesting observation here is that for higher ε levels, `TT-ICE`* achieves higher CR than `HT-RISE` but the image reconstructions yield lower quality at those target ε levels. For the MineRL competition dataset `TT-ICE`* compresses more frames than `HT-RISE` except for $\varepsilon = 0.10$, but this is caused by the 1D latent space of `TT-ICE`* which is more memory efficient than the 2D latent space of `HT-RISE` when $> 200,000$ frames are used. Overall, the results suggest that `HT-RISE` is more suitable for image datasets especially when the training dataset is limited or the visual quality of the reconstructions is important. `TT-ICE`* may be more appropriate when higher errors are allowed for compression.

5 Conclusion

In this work we proposed the batch hierarchical Tucker format, a slightly modified but more efficient version of the hierarchical Tucker format that is also suitable for incremental updates, and a new incremental tensor decomposition algorithm called Hierarchical Incremental Tucker (`HT-RISE`), which, to the best of our knowledge, is the first incremental tensor decomposition algorithm that updates an approximation in the hierarchical Tucker format. We compared `HT-RISE` with an state-of-the-art incremental tensor decomposition algorithm `TT-ICE`* using two PDE-based and two image-based datasets. The results indicate that in datasets with multi-scale features `HT-RISE` offers a fast and memory efficient way to compress high-dimensional data with a low relative test error. In simpler datasets, `TT-ICE`* outperforms `HT-RISE` in terms of compression ratio, compression time, and relative test error. The results also suggest that `HT-RISE` generalizes better with less training data compared to `TT-ICE`* regardless of the target ε level and the dataset.

Future work includes parallelizing `HT-RISE` to speed up the compression process, provide a version of `HT-RISE` to take advantage of GPU acceleration, and extending the `HT-RISE` algorithm to work with n-ary trees. Furthermore, we plan to use the latent space learned through `HT-RISE` in downstream learning tasks (such as generative modeling) to showcase the usability of the learned representations through `HT-RISE`.

Acknowledgments and Disclosure of Funding

We acknowledge partial support by Los Alamos National Laboratories under the project "Algorithm/Software/Hardware Co-design for High Energy Density applications" at the University of Michigan, and partial support from the Automotive Research Center at the University of Michigan (UM) in accordance with Cooperative Agreement W56HZV-19-2-

0001 with U.S. Army DEVCOM Ground Vehicle Systems Center. This work used computing resources provided by an AFOSR DURIP under Program Manager Dr. Fariba Fahroo and grant number FA9550- 23-1-006 and additional computational resources and services provided by Advanced Research Computing (ARC), a division of Information and Technology Services (ITS) at the University of Michigan, Ann Arbor

Appendix A. Proofs

This section contains mathematical proofs for the correctness of the `BHT-l2r` and `HT-RISE` algorithms.

To guarantee the correctness of the `BHT-l2r` algorithm, we first need to show that we can compute an upper bound for the approximation error incurred at an individual layer. This is done in the following lemma.

Lemma 2 (Layerwise approximation error) *Given an N-batch of d-dimensional tensors $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$, the approximation error incurred at the ℓ-th layer $\sqrt{\|\mathcal{C}_{\ell+1}\|_F^2 - \|\mathcal{C}_\ell\|_F^2}$ is upper bounded by*

$$\sqrt{\sum_{i=1}^{|\mathbf{T}_\ell|} \|E_{\ell,i}\|_F^2}, \qquad \textit{where} \qquad E_{\ell,i} := C_{\ell,(i)} - U_{\ell,i}\Sigma_{\ell,i}V_{\ell,i}^T$$

is the truncated portion the SVD performed on the mode-i unfolding of the intermediate tensor $\mathcal{C}_\ell$ at the ℓ-th layer.

Proof This is a straight forward result following the application of (Grasedyck, 2010, Lemma 3.8) to each node in $\mathbf{T}_\ell$. In Equation (8), we have shown that the core tensor at the ℓ-th layer is computed by contracting the core tensor at the $(\ell+1)$-th layer with orthonormal matrices obtained through error-truncated SVDs on the ℓ-th layer. Thus, decomposing the ℓ-th layer can be seen analogous to the notion *successive truncation* of Lemma 3.8 of Grasedyck (2010). Therefore, we can upper bound the approximation error at the ℓ-th layer with the sum of the Frobenius norms of the truncated error matrices $E_{\ell,i}$. ■

Theorem 2 is a crucial step in providing an upper bound on the total approximation error of the `BHT-l2r` algorithm. The total approximation error is computed by summing the approximation errors at each layer, as shown in the following theorem.

Theorem 3 (Total approximation error (Grasedyck, 2010, Lemma 3.10)) *Given an N-batch of d-dimensional tensors $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$, the approximation error of the reconstructed batch hierarchical Tucker decomposition $\tilde{\mathcal{Y}}$ is upper bounded by $\sum_{\ell=0}^{p} \sum_{i=1}^{|\mathbf{T}_\ell|} \|E_{\ell,i}\|_F^2$,*

$$\|\mathcal{Y} - \tilde{\mathcal{Y}}\|_F^2 \leq \sum_{\ell=0}^{p} \sum_{i=1}^{|\mathbf{T}_\ell|} \|E_{\ell,i}\|_F^2, \tag{26}$$

where p is the depth of the dimension tree $\mathbf{T}$.

Proof This is a direct generalization of Theorem 2 to the entire dimension tree $\mathbf{T}$. Once an upper bound on the approximation error at each layer is established, the total approximation error can be computed by summing the approximation errors at each layer. ■

This next corollary provides a direct application of how the approximation error can be distributed over the individual cores of the hierarchical tensor network.

Corollary 4 (`BHT-l2r` approximation error) *Let $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$ be an N-batch of d-dimensional tensors and $\mathbf{H}_{\mathcal{Y}}$ be its approximation in batch HT format computed with Algorithm 1 with an absolute error tolerance ε_{abs}, cores $\mathcal{G}$, and dimension tree $\mathbf{T}$. Then, the reconstruction of this approximation using eq. (6), $\tilde{\mathcal{Y}}$, satisfies $\|\mathcal{Y} - \tilde{\mathcal{Y}}\|_F \leq \varepsilon_{abs}$ if a nodewise error tolerance ε_{nw} is chosen such that*

$$\varepsilon_{nw} = \frac{\varepsilon_{abs}}{\sqrt{2d-2}}. \tag{27}$$

Proof The proof follows from Theorem 3. If we set $\|E_{\ell,i}\| \leq \varepsilon_{nw}$ for all ℓ and i, then using (26) the expression for the overall approximation error becomes

$$\|\mathcal{Y} - \tilde{\mathcal{Y}}\|_F^2 \leq \sum_{\ell=0}^{p} \sum_{i=1}^{|\mathbf{T}_\ell|} \varepsilon_{nw}^2. \tag{28}$$

Since ε_{nw} is assumed constant for all layers and nodes, the double sum on the right-hand side is simply repeated $2d-2$ times, making the expression

$$\|\mathcal{Y} - \tilde{\mathcal{Y}}\|_F^2 \leq (2d-2)\varepsilon_{nw}^2.$$

Setting $\varepsilon_{nw} = \varepsilon_{abs}/\sqrt{2d-2}$ and taking the square root of both sides yields (26) and therefore completes the proof. ■

Once we can guarantee an approximation error upper bound to `BHT-l2r`, a next natural step is to compute the exact approximation error of incurred by the `BHT-l2r` algorithm. We use Equation (13) as a proxy for the approximation error. In the following claim we will prove that the relative error of the approximation can be computed directly from the Frobenius norms of the original tensor and its projected latent space representation.

Claim 1 (Orthogonal reconstructions) *Given an N^k-batch of d-dimensional tensors $\mathcal{Y}^k \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N^k}$, an accumulation of d-dimensional tensors in batch HT format $\mathbf{H}_{\mathcal{X}^{k-1}}$, the projection of $\mathcal{Y}^k$ onto the HT-cores $\bar{\mathcal{C}}_1^k \in \mathbb{R}^{r_{1,1} \times r_{1,2} \times N^k}$, and the reconstruction of $\mathcal{Y}^k$ from $\bar{\mathcal{C}}_1^k$ using the HT-cores $\tilde{\mathcal{Y}}^k$, the error of approximation is equal to the difference in squared Frobenius norm of $\tilde{\mathcal{Y}}^k$ and $\bar{\mathcal{C}}_1^k$, i.e,*

$$\|\mathcal{Y}^k - \tilde{\mathcal{Y}}^k\|_F = \sqrt{\|\mathcal{Y}^k\|_F^2 - \|\bar{\mathcal{C}}_1^k\|_F^2}.$$

Proof The proof is direct and has two steps. First we will show that $\|\mathcal{Y}^k - \tilde{\mathcal{Y}}^k\|_F^2 = \|\mathcal{Y}^k\|_F^2 - \|\tilde{\mathcal{Y}}^k\|_F^2$ and then we will show that $\|\tilde{\mathcal{Y}}^k\|_F^2 = \|\bar{\mathcal{C}}_1^k\|_F^2$.

First, we take the square of the approximation error to get $\|\mathcal{Y}^k - \tilde{\mathcal{Y}}^k\|_F^2$. Then, by using $\|A+B\|_F^2 = \|A\|_F^2 + \|B\|_F^2 + 2\langle A, B\rangle_F$ we expand the squared Frobenius norm of the approximation error as

$$\|\mathcal{Y}^k - \tilde{\mathcal{Y}}^k\|_F^2 = \|\mathcal{Y}^k\|_F^2 - 2\langle \mathcal{Y}^k, \tilde{\mathcal{Y}}^k\rangle_F + \|\tilde{\mathcal{Y}}^k\|_F^2$$

Without loss of generality, we can split $\mathcal{Y}$ into $\mathcal{Y}^k = \mathcal{Y}^k_{\|} + \mathcal{Y}^k_{\perp}$, where $\mathcal{Y}^k_{\|}$ is the projection of $\mathcal{Y}^k$ onto the HT-cores $\mathcal{G}^{k-1}$ and $\mathcal{Y}^k_{\perp}$ is the orthogonal component of $\mathcal{Y}^k_{\|}$ with respect to the HT-cores of the accumulation $\mathcal{X}^{k-1}_{\mathbf{H}}$. Then, using the properties of the Frobenius inner product we can write $\langle \mathcal{Y}^k, \tilde{\mathcal{Y}}^k\rangle_F = \langle \mathcal{Y}^k_{\|}, \tilde{\mathcal{Y}}^k\rangle_F + \langle \mathcal{Y}^k_{\perp}, \tilde{\mathcal{Y}}^k\rangle_F$. Since $\tilde{\mathcal{Y}}^k$ is the reconstruction of $\bar{\mathcal{C}}^k_1$ using the HT-cores, the first term is equal to the squared Frobenius norm of $\tilde{\mathcal{Y}}^k$. The second term is zero since $\mathcal{Y}^k_{\perp}$ is by definition orthogonal to $\tilde{\mathcal{Y}}^k$. Therefore, we can write the squared Frobenius norm of the approximation error as

$$\|\mathcal{Y}^k - \tilde{\mathcal{Y}}^k\|_F^2 = \|\mathcal{Y}^k\|_F^2 - \|\tilde{\mathcal{Y}}^k\|_F^2$$

and complete the first part of the proof.

For the second part, we need to show that $\|\tilde{\mathcal{Y}}^k\|_F^2 = \|\bar{\mathcal{C}}^k_1\|_F^2$. The reconstruction of $\tilde{\mathcal{Y}}^k$ from $\bar{\mathcal{C}}^k_1$ is done by performing a sequence of outer products with the HT-cores $\mathcal{G}^{k-1}$. For the first layer of HT-cores, the outer product that yields the intermediate core for the i-th tensor in the N^k-batch, $\tilde{\mathcal{C}}^k_2(:,:,:,:,i)$[5], can be shown as a reshaping of

$$\sum_{p,q=1}^{r_{1,1},r_{1,2}} \bar{\mathcal{C}}^k_1(p,q,i)\mathcal{G}^{k-1}_{1,1}(:,:,p) \otimes \mathcal{G}^{k-1}_{1,2}(:,:,q),$$

where $\otimes$ denotes the Kronecker product. Note that $\mathcal{G}^t_{\ell,j}(:,:,i)$ is an orthonormal matrix for all t,ℓ,j,i if it is obtained through `bht-l2r` or `HT-RISE`. For Kronecker products, we have $\|A\otimes B\| = \|A\|\|B\|$ for any induced norm, therefore we have $\|\tilde{\mathcal{C}}^k_2(:,:,:,:,i)\|_F^2 = \|\bar{\mathcal{C}}^k_1(p,q,i)\mathcal{G}^{k-1}_{1,1}(:,:,p)\otimes\mathcal{G}^{k-1}_{1,2}(:,:,q)\|_F^2 = \|\bar{\mathcal{C}}^k_1(p,q,i)\|_F^2$ and $\|\tilde{\mathcal{C}}^k_2\|_F^2 = \|\bar{\mathcal{C}}^k_1\|_F^2$.

Since the orthonormality of $\mathcal{G}^{k-1}_{\ell,j}(:,:,i)$ holds for all t,ℓ,j,i, the same argument can be applied to all layers of the HT-cores. Therefore, the squared Frobenius norm of the reconstruction is equal to the squared Frobenius norm of the projection, i.e., $\|\tilde{\mathcal{Y}}^k\|_F^2 = \|\bar{\mathcal{C}}^k_1\|_F^2$. This completes the proof. ■

Claim 1 also applies to the `HT-RISE` algorithm, as the reconstruction of the streamed tensor from the hierarchical Tucker approximation is done in the same way as in `BHT-l2r`, using orthonormal cores.

Finally with then next theorem we provide a proof of correctness for the `HT-RISE` algorithm.

Theorem 5 **(`HT-RISE` approximation error)**

Proof The proof uses Theorem 3 and is direct. Let $\bar{\mathcal{C}}^k_\ell$ (for any $\ell = p,\dots,1$ and $\bar{\mathcal{C}}_p = \mathcal{Y}^k$) be the intermediate tensor obtained at the ℓ-th layer while processing the streamed tensor

5. $\mathcal{A}(:,:,:,:,i)$ refers to the i-th mode-5 slice for the 5-dimensional tensor $\mathcal{A}$

$\mathcal{Y}^k$ by `HT-RISE`. After updating $\mathcal{G}_{\ell,j}^{k-1}$ to $\mathcal{G}_{\ell,j}^{k}$ using $\bar{\mathcal{C}}_\ell^k$ and Equations (14) to (16), we can write $\bar{\mathcal{C}}_\ell^k$ as

$$\bar{C}_{\ell,(j)}^k = U_{\ell,j}^{k-1}(U_{\ell,j}^{k-1})^T \bar{C}_{\ell,(j)}^k + U_{R_{\ell,j}}^k \Sigma_{R_{\ell,j}}^k (V_{R_{\ell,j}}^k)^T + E_{R_{\ell,j}}, \tag{29}$$

where $U_{\ell,j}^{k-1}$ is the reshaped core $\mathcal{G}_{\ell,j}^{k-1}$. The first term on the RHS is the part of $\bar{\mathcal{C}}_\ell^k$ that is already represented in $\mathcal{X}_{\mathbf{H}}^{k-1}$, and the second term is the part represented due to the added basis vectors $U_{R_{\ell,j}}^k$. If we consider the updated core $U_{\ell,j}^k = \begin{bmatrix} U_{\ell,j}^{k-1} & U_{R_{\ell,j}}^k \end{bmatrix}$, we then can manipulate (29) to resemble Theorem 2 as

$$\bar{C}_{\ell,(j)}^k = U_{\ell,j}^k (U_{\ell,j}^k)^T \bar{C}_{\ell,(j)}^k + E_{R_{\ell,j}} = U_{\ell,j}^k \Sigma_{\ell,j}^k (V_{\ell,j}^k)^T + E_{R_{\ell,j}},$$

where the error in the approximation of $\bar{\mathcal{C}}_\ell^k$ is given by $E_{\ell,j}$. This allows us to compute an upper bound on the error incurred for each HT-core on the ℓ-th layer using Theorem 2 and accumulate the layerwise approximation upper bounds to obtain a total approximation error using Theorem 3. ■

Since we have shown that we can guarantee an upper bound on the approximation error of the `HT-RISE` algorithm, now the last step is to show that the updates performed by `HT-RISE` do not affect the representation of the tensors that were streamed in the past. This is shown in the following theorem.

Theorem 6 (Error guarantees for the past stream) *Let $\mathcal{X}_m^k \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ be the m-th tensor in the accumulation $\mathcal{X}^k$ at time k, and $\mathbf{H}_{\mathcal{X}^k}$ be the hierarchical Tucker approximation computed using Algorithm 2 (`HT-RISE`) with cores $\mathcal{G}_{\ell,j}^k$. For any $t \geq k$, the reconstruction of the m-th tensor from $\mathbf{H}_{\mathcal{X}_m^t}$, denoted $\tilde{\mathcal{X}}_m^t$, equals the reconstruction from $\mathbf{H}_{\mathcal{X}_m^k}$, i.e., $\tilde{\mathcal{X}}_m^k = \tilde{\mathcal{X}}_m^t$.*

Proof The proof simply demonstrates that the core modifications performed by `HT-RISE` have no impact on the representation of earlier tensors and is similar to (Aksoy et al., 2024a, Theorem 4). To demonstrate $\tilde{\mathcal{X}}_m^k = \tilde{\mathcal{X}}_m^t$, we need to reconstruct $\mathbf{H}_{\mathcal{X}_m^t}$. Let $\bar{C}_1^{k,m} \in \mathbb{R}^{r_{1,1}^k \times r_{1,2}^k}$ be the slice of the root core $\bar{\mathcal{C}}_1^k$ that corresponds to $\mathcal{X}_m^k$. For any $t > k$, $\bar{C}_1^{k,m}$ gets padded with zeros according to Equation (17) and becomes

$$\bar{C}_1^{t,m} = \begin{bmatrix} \bar{C}_1^{k,m} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \tag{30}$$

with shape $r_{1,1}^t \times r_{1,2}^t$.

Since the reconstruction happens in the root-to-leaf direction, we use the $\bar{C}_1^{t,m}$ from (30) and contract with the HT-cores on the first level as $\tilde{\mathcal{C}}_1^{t,m} = \bar{C}_1^{t,m} \,{}_1\times_3\, \mathcal{G}_{1,1}^t \,{}_3\times_3\, \mathcal{G}_{1,2}^t$. As a result, we obtain $\tilde{\mathcal{C}}_1^{t,m} \in \mathbb{R}^{r_{2,1}^t \times r_{2,2}^t \times r_{2,3}^t \times r_{2,4}^t}$ is the intermediate core of layer 1. After the contraction, we compute the (i_1, i_2, i_3, i_4)-th element of $\tilde{\mathcal{C}}_1^{t,m}$ as

$$\tilde{\mathcal{C}}_1^{t,m}(i_1, i_2, i_3, i_4) = \sum_{\alpha=1}^{r_{1,1}^t} \sum_{\beta=1}^{r_{1,2}^t} \bar{C}_1^{t,m}(\alpha, \beta) \mathcal{G}_{1,1}^t(i_1, i_2, \alpha) \mathcal{G}_{1,2}^t(i_3, i_4, \beta). \tag{31}$$

However, since $\bar{C}_1^{t,m}(\alpha,\beta)=0\ \forall\alpha>r_{1,1}^k,\ \beta>r_{1,2}^k$, we can rewrite Equation (31) as

$$\tilde{\mathcal{C}}_1^{t,m}(i_1,i_2,i_3,i_4)=\sum_{\alpha=1}^{r_{1,1}^k}\sum_{\beta=1}^{r_{1,2}^k}\bar{C}_1^{k,m}(\alpha,\beta)\mathcal{G}_{1,1}^t(i_1,i_2,\alpha)\mathcal{G}_{1,2}^t(i_3,i_4,\beta).$$

Next, we consider the padding of the contracted cores. Since cores on layer ℓ are padded with zeros to ensure dimensional consistency after updating layer $\ell+1$ as in Equation (17), the entries of $\mathcal{G}_{1,1}^t(i_1,i_2,:r_{1,1}^k)$ and $\mathcal{G}_{1,2}^t(i_3,i_4,:r_{1,2}^k)$ are all zero for any $i_j>r_{2,j}^k$, where $\mathcal{G}_{\ell,j}^t(\alpha,\beta,:r_{\ell,j}^k)$ is the mode-3 fiber of $\mathcal{G}_{\ell,j}^t$ at position (α,β) with depth $r_{\ell,j}^k$. Therefore, $\tilde{\mathcal{C}}_1^{t,m}(i_1,i_2,i_3,i_4)=0$ for any $i_j>r_{2,j}^k$. For a full dimension tree this results in

$$\begin{aligned}\tilde{\mathcal{C}}_2^{t,m}(i_1,\dots,i_8)=&\sum_{\alpha=1}^{r_{2,1}^t}\sum_{\beta=1}^{r_{2,2}^t}\sum_{\gamma=1}^{r_{2,3}^t}\sum_{\theta=1}^{r_{2,4}^t}\tilde{\mathcal{C}}_1^{t,m}(\alpha,\beta,\gamma,\theta)\mathcal{G}_{2,1}^t(i_1,i_2,\alpha)\cdots\mathcal{G}_{4,1}^t(i_7,i_8,\alpha),\\ =&\sum_{\alpha=1}^{r_{2,1}^k}\sum_{\beta=1}^{r_{2,2}^k}\sum_{\gamma=1}^{r_{2,3}^k}\sum_{\theta=1}^{r_{2,4}^k}\tilde{\mathcal{C}}_1^{k,m}(\alpha,\beta,\gamma,\theta)\mathcal{G}_{2,1}^t(i_1,i_2,\alpha)\cdots\mathcal{G}_{4,1}^t(i_7,i_8,\alpha).\end{aligned}\tag{32}$$

for the second layer. This process is repeated for layers $\ell=3,\dots,p-1$. Then for the last layer p, the reconstruction is performed as

$$\begin{aligned}\tilde{\mathcal{X}}_m^t=\tilde{\mathcal{C}}_p^{t,m}(i_1,\dots,i_d)=&\sum_{\alpha=1}^{r_{p,1}^t}\cdots\sum_{\zeta=1}^{r_{p,d}^t}\tilde{\mathcal{C}}_{p-1}^{t,m}(\alpha,\dots,\zeta)\mathcal{G}_{p,1}^t(i_1,\alpha)\cdots\mathcal{G}_{p,d}^t(i_d,\zeta),\\ =&\sum_{\alpha=1}^{r_{p,1}^k}\cdots\sum_{\zeta=1}^{r_{p,d}^k}\tilde{\mathcal{C}}_{p-1}^{k,m}(\alpha,\dots,\zeta)\mathcal{G}_{p,1}^t(i_1,\alpha)\cdots\mathcal{G}_{p,d}^t(i_d,\zeta).\end{aligned}\tag{33}$$

Equations (31) to (33) all use the ranks up to $r_{\ell,j}^k$ while reconstructing $\tilde{\mathcal{X}}_m^t$. This shows that for any $t\geq k$, the reconstruction of the m-th tensor in the accumulation $\mathbf{H}_{\mathcal{X}^t}$ is the same as the reconstruction of the m-th tensor in the accumulation $\mathbf{H}_{\mathcal{X}^k}$. Therefore, the error guarantees of the `HT-RISE` algorithm are preserved for the past stream. ■

Appendix B. Supplementary Algorithms

This section contains pseudocodes for the supplementary algorithms that are used in Algorithms 1 and 2.

- `updateIndexSet`: During the update process with `HT-RISE` (Algorithm 2), it is important to keep track of the added basis vectors to each core. To ensure dimensional consistency throughout the update process, the `updateIndexSet` function (algorithm B.1) updates the index sets corresponding to layers and nodes.
- `expandCore`: Once the missing basis vectors of a core are identified, then the next step is to merge the missing basis vectors with the existing ones stored in a Tucker core.

Algorithm B.1 `updateIndexSet`: Updates the index set of a given object

```
1: Input
2:     $\mathbf{I}_{\Gamma_\alpha}$                  index set of the object $\Gamma$
3:     $\Gamma_\alpha$                     object to update the index set
4: Output
5:     $\mathbf{I}_{\Gamma_\alpha}$                  updated index set
6: if $\Gamma_\alpha$ is a node then                  ▷ $\alpha$ is a tuple with layer and node indices
7:     $\ell, j \leftarrow \alpha$
8:     $\mathbf{I}_{\Gamma_\alpha} \leftarrow \begin{cases} \{n_{d_{\ell,j}}, r_{\ell,j}\} & \text{if } \Gamma_\alpha \text{ is a leaf} \\ \left\{ \bigcup_{(\ell+1,p)\in \mathbf{S}_{\ell,i}} r_{\ell+1,p} \right\} \cup \{r_{\ell,i}\} & \text{if } \Gamma_\alpha \text{ is a transfer node.} \end{cases}$
9: else if $\Gamma$ is a layer then                   ▷ $\alpha$ is the layer index
10:    $\mathbf{I}_{\Gamma_\alpha} \leftarrow \bigcup_{j=1}^{|\Gamma_\alpha|} \begin{cases} n_{d_{\alpha,j}} & \text{if } \Gamma_{\alpha,j} \text{ is a leaf} \\ \prod_{(\alpha+1,m)\in \mathbf{S}_{\alpha,j}} r_{\alpha+1,m} & \text{if } \Gamma_{\alpha,j} \text{ is a transfer node.} \end{cases}$   ▷ $\Gamma_{\alpha,j}$ is the $j$-th node on layer $\alpha$
11: end if
```

The `expandCore` (Algorithm B.2) algorithm takes in a node, along with its dimension tree, existing Tucker core and the new basis vectors corresponding to that node and updates the core.

- `padWithZeros`: The `padWithZeros` algorithm is used to pad a Tucker core with zeros to ensure dimensional consistency after updating the core. The algorithm is presented in Algorithm B.3.

Appendix C. Alternative approach to determine an upper bound to the approximation error $\|E_{\ell,j}\|_F$

In this section, we propose an alternative method to determine an upper bound to the approximation error $\|E_{\ell,j}\|_F$ for `BHT-l2r` and `HT-RISE` algorithms.

Since all the operations in both algorithms are projections onto truncated orthogonal bases, we can compute the error of approximation directly by comparing the norm of an intermediate tensor $\mathcal{C}_\ell$ to the original d-dimensional tensor $\mathcal{Y}$. This allows us to update the nodewise error tolerance ε_{nw} adaptively throughout both algorithms. Let ε_{abs} be the determined absolute error tolerance for the hierarchical Tucker approximation. Both `BHT-l2r` and `HT-RISE` will perform $(2d-2)$ `SVD`s to either compute a hierarchical Tucker representation or update an existing one. At the beginning of both algorithms, the nodewise error tolerance ε_{nw} is set to $\varepsilon_{abs}/\sqrt{2d-2}$.

Once the computation of the ℓ-th layer is complete and $\mathcal{Y}$ is projected onto $\mathcal{G}_{\ell,j}$, we can track the current error of approximation. Let $\bar{\mathcal{C}}_\ell$ be that approximation. Then, the current error of approximation is then computed as

$$\varepsilon_\ell = \sqrt{\|\mathcal{Y}\|_F^2 - \|\bar{\mathcal{C}}_\ell\|_F^2}, \tag{34}$$

Algorithm B.2 `expandCore`: Expands the basis of a Tucker core

```
Input
    $\mathbf{N}_{\ell,j}$              node whose core is to be expanded
    $\mathbf{I}_{\mathbf{N}_{\ell,j}}$              index set of the node
    $\mathcal{G}_{\ell,j}$              core to be expanded
    $U$              $r_U$ orthogonal vectors to be appended to $\mathcal{G}_{\ell,j}$
Output
    $\mathcal{G}_{\ell,j}$              *updated* core
    $\mathbf{I}_{\mathbf{N}_{\ell,j}}$              *updated* index set
if $\mathbf{N}_{\ell,j}$ is a leaf node then
    $\mathcal{G}_{\ell,j} \leftarrow \begin{bmatrix}\mathcal{G}_{\ell,j} & U\end{bmatrix}$ ▷ $\mathcal{G}_{\ell,j}$ is already in orthonormal matrix form, we can directly append $U$ to $\mathcal{G}_{\ell,j}$
    $\mathbf{I}_{\mathbf{N}_{\ell,j}} \leftarrow$ updateIndexSet($\mathbf{I}_{\mathbf{N}_{\ell,j}}, \mathbf{N}_{\ell,j}$)    ▷ Update the index set of the node to reflect the new rank
else if $\mathbf{N}_{\ell,j}$ is a transfer node then
    $\alpha \leftarrow \frac{1}{r_{\ell,j}} \prod_{\gamma \in \mathbf{I}_{\mathbf{N}_{\ell,j}}} \gamma$
    $\mathcal{G}_{\ell,j} \leftarrow$ reshape($\mathcal{G}_{\ell,j}, [\alpha, r_{\ell,j}]$)   ▷ $\mathcal{G}_{\ell,j}$ is reshaped to an orthonormal matrix before appending $U$
    $\mathcal{G}_{\ell,j} \leftarrow \begin{bmatrix}\mathcal{G}_{\ell,j} & U\end{bmatrix}$
    $\mathbf{I}_{\mathbf{N}_{\ell,j}} \leftarrow$ updateIndexSet($\mathbf{I}_{\mathbf{N}_{\ell,j}}, \mathbf{N}_{\ell,j}$)    ▷ Update the index set of the node to reflect the new rank
    $\mathcal{G}_{\ell,j} \leftarrow$ reshape($\mathcal{G}_{\ell,j}, I_{\mathbf{N}_{\ell,j}}$)                    ▷ Fold $\mathcal{G}_{\ell,j}$ back to 3D core
end if
```

Algorithm B.3 `padWithZeros`: Pad a hierarchical Tucker core with zeros

```
1: Input
2:     $\mathcal{G}_{\ell,j} \in \mathbb{R}^{\alpha\times\beta\times r}$          HT-core to be padded
3:     $t$          index of the node to be padded among the children of its parent node
4:     $r'$          size of the zeros to be padded
5: Output
6:     $\mathcal{G}_{\ell,j}$          padded HT-core
7: $\mathbf{0} \in \mathbb{R}^{r'\times\beta\times r}$                    ▷ Assume $t = 1$ for simplicity
8: $\mathcal{G}_{\ell,j} \leftarrow \mathcal{G}_{\ell,j} \oplus^t \mathbf{0}$                    ▷ $\mathcal{G}_{\ell,j} \in \mathbb{R}^{(\alpha+r'\times\beta\times r)}$
```

where ε_ℓ denotes the error of approximation at the ℓ-th layer. To update the nodewise error tolerance ε_{nw} for the upcoming computations, we define an error budget ε_{rem} as

$$\varepsilon_{rem} = \sqrt{\varepsilon_{abs}^2 - \varepsilon_\ell^2}. \tag{35}$$

In addition to updating the remaining error budget, we also update the nodewise error tolerance ε_{nw} accordingly as

$$\varepsilon_{nw} = \frac{\varepsilon_{rem}}{\sqrt{\mathtt{svd}_{rem}}}, \tag{36}$$

with $\mathtt{svd}_{rem}$ being the number of remaining `SVD`s to be performed.

This approach aims to compensate for the actual error of approximation incurred at each layer being lower than the desired approximation error as shown in Theorem 4 and Theorem 5. By updating the nodewise error tolerance ε_{nw} adaptively, we can get progressively more aggressive in the truncation of the Tucker cores as we move up the dimension tree. This approach is particularly useful when the original tensor $\mathcal{Y}$ is comprised of many small dimensions.

Appendix D. Additional results

This section contains the additional results for the experiments conducted in Section 4. In addition to those, this section further presents the comparison of the batch hierarchical Tucker format and the hierarchical Tucker format.

D.1 Comparison of the batch hierarchical Tucker format and the hierarchical Tucker format

This section considers the comparison of the batch hierarchical Tucker format and the hierarchical Tucker format. We compute the approximations in batch hierarchical Tucker format (Figure 3d) using the `BHT-l2r` algorithm (Algorithm 1) and the hierarchical Tucker format (Figure 3c) using the leaf-to-root compression algorithm of Grasedyck (2010, Alg. 2). We ran all algorithms until they run out of the allocated 64GB memory. Please note that in the experiments in this section do not include any incremental updates as the algorithms compared here are both one-shot algorithms. At every batch size, we compute the compression using the respective formats and algorithms from scratch.

We conduct our experiments using the PDEBench dataset with $\varepsilon_{rel} = 0.10$ and 0.05 as well as the BigEarthNet dataset with $\varepsilon_{rel} = 0.05$, 0.10, 0.15, and 0.30 with the same reshapings as in Section 4. For all scenarios, we compare compression ratio and compression time of both algorithms. Analogous to other experiments, the results are averaged over 5 seeds. When computed with `BHT-l2r`, our proposed batch hierarchical Tucker format results in a significant reduction in compression time in comparison to the regular hierarchical Tucker format computed with the leaves-to-root decomposition algorithm presented in Grasedyck (2010, Alg. 2) while returning better compression ratios. Figures D.1 and D.2, and Figure D.3 show the results for the PDEBench and BigEarthNet datasets, respectively.

Figure D.1 shows the results of the comparison between HT format and BHT format in CR and compression time for $\varepsilon_{rel} = 0.10$. Note that the batch hierarchical Tucker format achieves higher compression in comparison to their hierarchical Tucker format version using the same normalization. At this target relative error level, both algorithms compress the same amount of simulations except for HT format with z-score normalization. Another intersting observation from Figure D.1 is that the BHT format results in faster compression irrespective of the normalization method used.

Figure D.2 shows the results of the comparison between HT format and BHT format in CR and compression time for $\varepsilon_{rel} = 0.05$. Note that similar to the case of $\varepsilon_{rel} = 0.10$, the BHT format offers higher compression and lower compression time compared to the HT format. This time, the efficiency difference between BHT and HT manifests itself in the compression of more simulations. The BHT format compresses at least 6 simulations more than the HT format across all normalization methods. In parallel to the results of

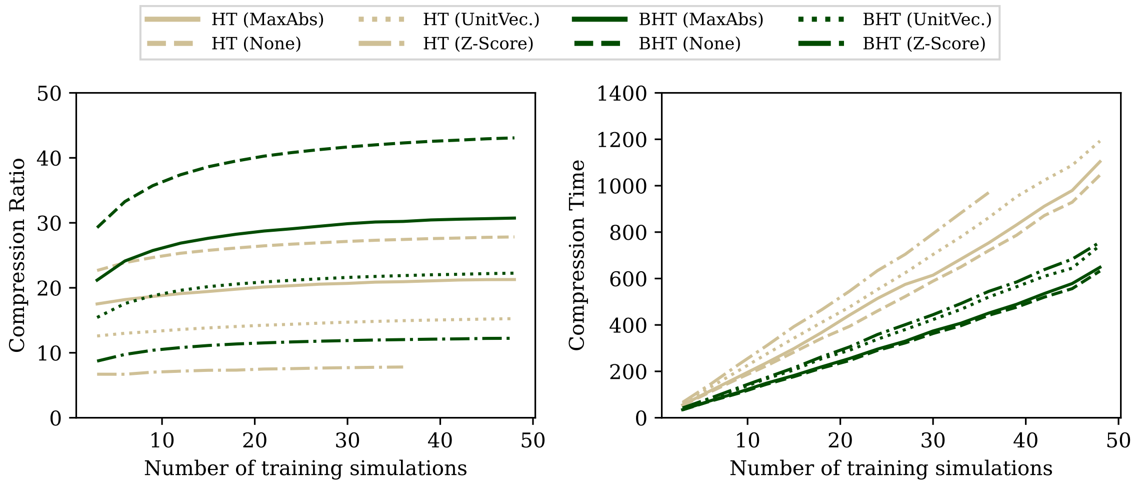


Figure D.1: *Comparison of hierarchical Tucker leaf-to-root compression and batch hierarchical Tucker decomposition in terms of compression ratio (CR - left) and compression time (right) on the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *using various normalization methods. For the same normalization methods, the batch hierarchical Tucker format achieves on average* $1.5\times$ *compression over the hierarchical Tucker format, while running on average* $1.7\times$ *faster. The batch hierarchical Tucker format compresses 12 simulations more than the hierarchical Tucker format using z-score normalization. All experiments are run until failure due to insufficient memory. MaxAbs: Maximum absolute value normalization, None: No normalization, UnitVec: Unit vector normalization, ZScore: Z-score normalization. The results are averaged over 5 seeds.*

$\varepsilon_{rel} = 0.10$, the BHT format results in faster compression irrespective of the normalization method used.

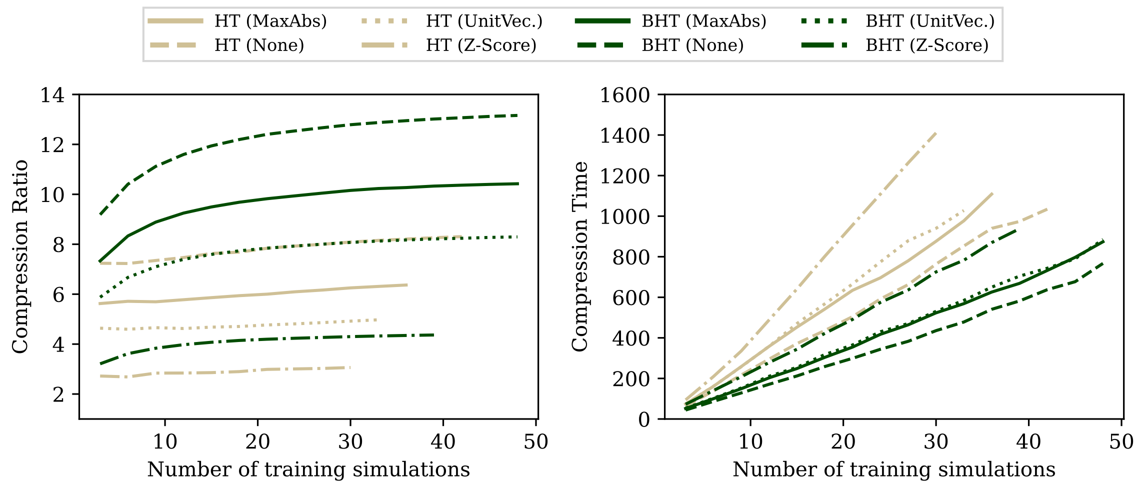


Figure D.2: *Comparison of HT format and BHT format in terms of compression ratio (CR - left) and compression time (right) on the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.05$ *using various normalization methods. For the same normalization methods, the BHT format achieves* $1.4-1.6\times$ *compression over the HT format, while running* $1.6-1.9\times$ *faster. The BHT format compresses at least 6 simulations more than the HT format at all normalization methods. All experiments are run until failure due to insufficient memory. Please refer to the caption of Figure D.1 for the normalization methods.*

In addition to the 3D Navier-Stokes simualtions, we also conduct experiments on the BigEarthNet dataset. Figure D.3 shows the results of the comparison between HT format and BHT format in CR and compression time for the BigEarthNet dataset. In line with our findings from the PDEBench dataset, the BHT format offers higher compression and lower compression time compared to the HT format, specifically at higher target relative error levels. At the higher target relative error level, $\varepsilon_{rel} = 0.30$, the BHT format achieves $6.2\times$ compression over the HT format. The difference between HT and BHT formats become less pronounced as the target relative error level decreases. At $\varepsilon_{rel} = 0.15$, the BHT format achieves only $1.5\times$ compression over the HT format. At the lower target relative error level, $\varepsilon_{rel} = 0.10$, both formats yield comparable CRs with the HT format achieving $1.05\times$ compression over the BHT format. Finally at $\varepsilon_{rel} = 0.05$, the HT format achieves $1.3\times$ compression over the BHT format. In terms of compression time, the BHT format consistently outperforms the HT format. Over the target relative errors investigated, the BHT format runs $1.86 - 3.86\times$ faster than the HT format.

One interesting observation from Figure D.3 is that despite the higher compression ratios achieved by the BHT format, the HT format compresses more images than the BHT format at $\varepsilon_{rel} = 0.30$ and $\varepsilon_{rel} = 0.15$. The BHT format compresses 700 images less than the HT format at $\varepsilon_{rel} = 0.30$ and 100 images less than the HT format at $\varepsilon_{rel} = 0.15$. However, once the HT format surpasses the BHT format in terms of CR, the BHT format compresses more images. The BHT format compresses 100 images more than the HT format at $\varepsilon_{rel} = 0.10$ and 500 images more than the HT format at $\varepsilon_{rel} = 0.05$.

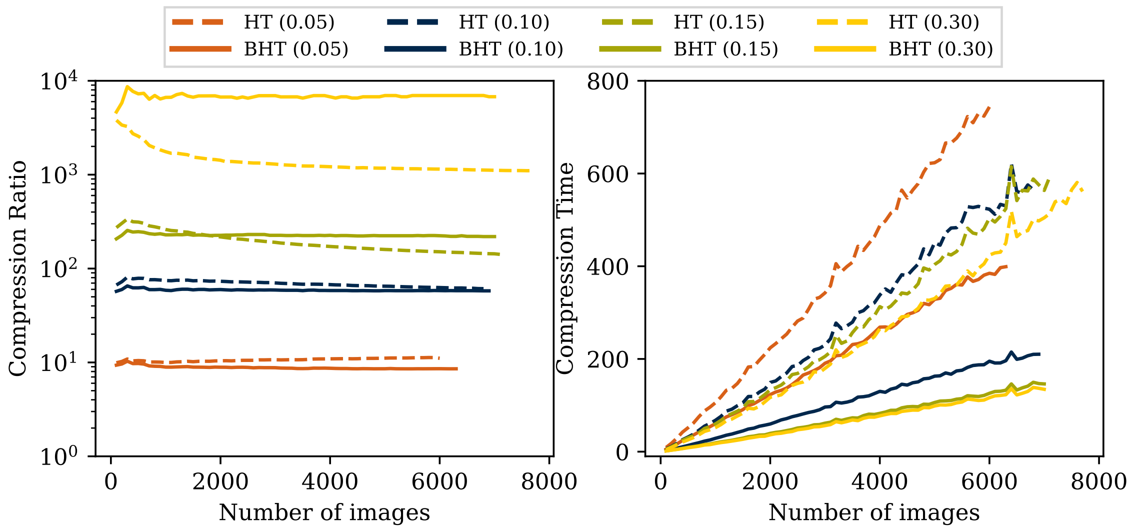


Figure D.3: *Comparison of hierarchical Tucker format and batch hierarchical Tucker format in terms of compression ratio (CR - left) and compression time (right) on the BigEarthNet dataset with* $\varepsilon_{rel} = 0.05 - 0.30$. *The BHT format achieves* $6.2\times$ *compression over the HT format at* $\varepsilon_{rel} = 0.3$ *while running* $3.75\times$ *faster. At* $\varepsilon_{rel} = 0.1$, *the BHT format achieves* $0.95\times$ *compression over the HT format while running* $2.69\times$ *faster. The BHT format compresses 100 images more and 700 images less than HT format at* $\varepsilon_{rel} = 0.1$ *and* $\varepsilon_{rel} = 0.3$, *respectively. All experiments are run until failure due to insufficient memory.*

D.2 Effect of the reshaping on compression performance

This section investigates the effect of reshaping on the compression performance of the `HT-RISE` algorithm. We consider the PDEBench 3D turbulent Navier-Stokes dataset with $\varepsilon_{rel} = 0.10$ and no normalization. We consider four different reshaping of the tensor. The baseline reshaping is denoted as $8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 5 \times 21 \times 1$. We alter the dimensionality by reshaping the tensor. Note that we only change the dimensionality of the tensor by altering the dimensions that correspond to the spatial discretization of the simulation. Figure D.4 shows the compression time and relative test error of the algorithms using various reshaping. Figure D.5 shows the comparison of compression time and RTE of the algorithms using the considered reshapings. Table D.1 presents the shapes of the tensors, and the resulting compression time, CR, RR, as well as RTE. For this type of experiments we set the maximum walltime limit to 4 days, similar to the experiments in Section 4.2.2

In Figure D.4 we observe that the changing the dimensionality of the tensor has a significant impact on the CR. The $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 5 \times 21 \times 1$ reshaping results in the highest CR ($33.59\times$) and RR ($33.87\times$) among the investigated reshaping. The $64 \times 64 \times 64 \times 5 \times 21 \times 1$ reshaping results in the lowest CR ($18.86\times$) and the reshaping $2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 5 \times 21 \times 1$ results in the lowest

Table D.1: *Results of experiments investigating the effect of reshaping on the compression performance of the* `HT-RISE` *algorithm using the PDEBench 3D turbulent Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$. *Time: Compression time in seconds, CR: Compression ratio, RR: Reeduction ratio, RTE: Relative test error.* ‡ *denotes that the algorithm failed to compress the dataset due to maximum walltime limit (389/480 simulations).*

Tensor Shape	Time(s)	CR	RR	RTE
$8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 5 \times 21 \times 1$	1158.7	32.68	32.94	0.098
$64 \times 64 \times 64 \times 5 \times 21 \times 1$	5289.4‡	18.86	32.59	0.108
$4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 5 \times 21 \times 1$	3563.5	33.59	33.87	0.098
$2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 5 \times 21 \times 1$	11,339	25.64	28.83	0.099

RR (28.83×). Furthermore, the $64 \times 64 \times 64 \times 5 \times 21 \times 1$ reshaping runs into maximum walltime timeout and therefore fails to compress the entire dataset.

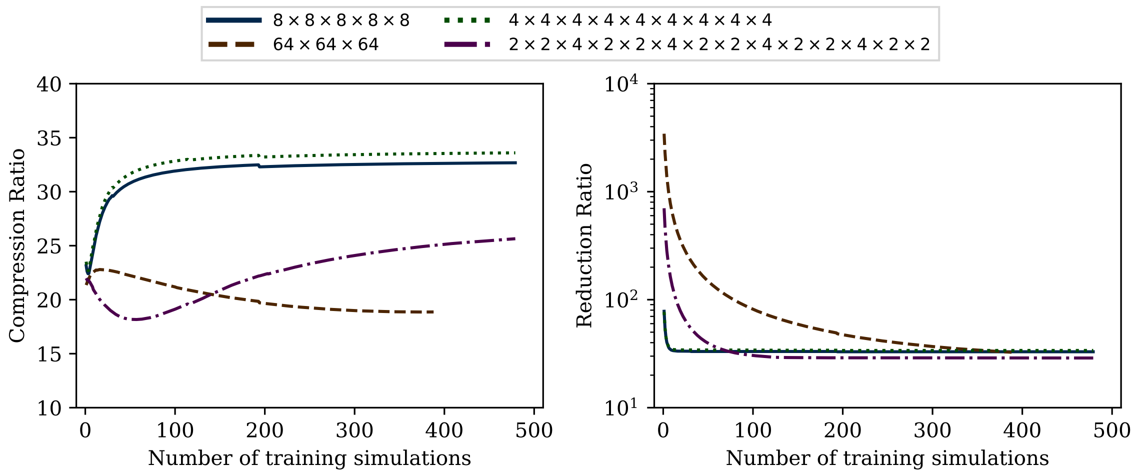


Figure D.4: *Comparison of various tensor reshapings in terms of compression ratio (CR - left) and reduction ratio (RR - right) on the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *using no normalization. The reshaping* $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 5 \times 21 \times 1$ *results* 1.8× *the CR of the worst performing reshaping,* $64 \times 64 \times 64 \times 5 \times 21 \times 1$. *Similarly, the reshaping* $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 5 \times 21 \times 1$ *results in* 1.2× *of the RR of the worst performing reshaping,* $2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 5 \times 21 \times 1$.

Figure D.5 shows that the baseline reshaping $8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 5 \times 21 \times 1$ results in the lowest compression time among the investigated reshapings (1158.7s). The reshaping $2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 5 \times 21 \times 1$ results in the highest compression time (11, 339s). The reshaping $64 \times 64 \times 64 \times 5 \times 21 \times 1$ runs into maximum walltime timeout and therefore is not considered in this comparison. However, the growth in compression time suggests that it would have been the slowest reshaping among the investigated ones if the entire dataset was successfully compressed.

Another observation from Figure D.5 is that all reshapings result in almost the same RTE $(0.098 - 0.099)$ except for $64 \times 64 \times 64 \times 5 \times 21 \times 1$ reshaping. The $64 \times 64 \times 64 \times 5 \times 21 \times 1$ reshaping results in the highest RTE among the investigated reshapings (0.108). However, this is likekly due to the fact that the algorithm fails to compress the entire dataset due to maximum walltime limit. The effect of the reshaping on the RTE is at the number of simulations it takes to reduce the RTE below the target ε_{rel}. The $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 5 \times 21 \times 1$ reshaping crosses the ε_{rel} threshold at 12 simulations, whereas the $2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 5 \times 21 \times 1$ reshaping crosses the ε_{rel} threshold at 127 simulations.

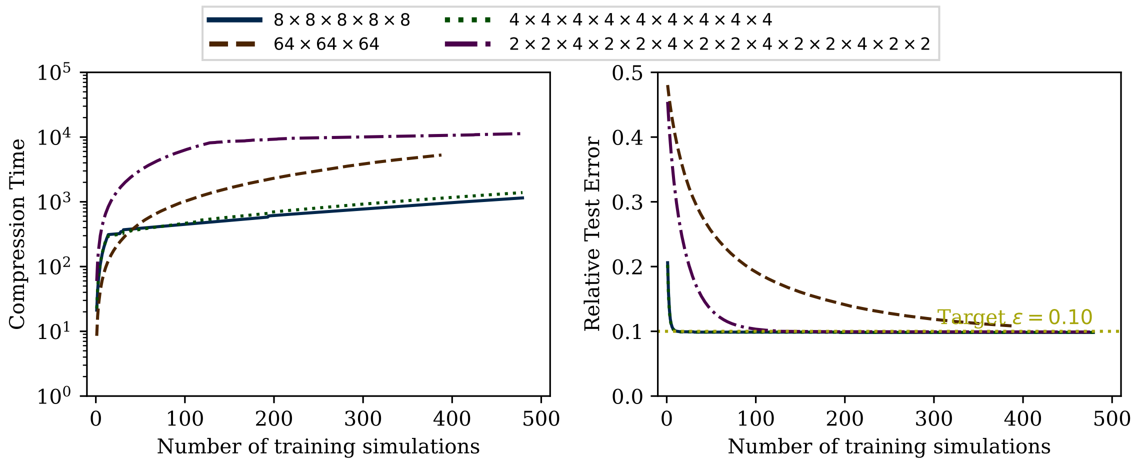


Figure D.5: *Comparison of various tensor reshapings in terms of compression time (left) and relative test error (RTE - right) on the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *using no normalization. The reshaping* $2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times 2 \times 5 \times 21 \times 1$ *takes* $9.8\times$ *the time it takes the reshaping* $8 \times 8 \times 8 \times 8 \times 8 \times 8 \times 5 \times 21 \times 1$ *to compress the dataset. All reshapings except for* $64 \times 64 \times 64 \times 5 \times 21 \times 1$ *result in almost the same RTE.*

D.3 Effect of the axis ordering on compression performance

This section considers an empirical investigation of the effect of axis ordering on the compression performance of the `HT-RISE` algorithm. As the ordering of the axes determine the order of interatction between the dimensions, it is expected that the axis ordering has an effect on the compression performance of the `HT-RISE` algorithm. We investigate the effect of axis ordering on the compression performance of the `HT-RISE` algorithm using the PDEBench 3D turbulent Navier-Stokes dataset with $\varepsilon_{rel} = 0.10$ and no normalization. We consider four different axis reorderings of the tensor. The baseline axis ordering is denoted as $[0, 1, 2, 3, 4, 5, 6, 7, 8]$. The other axis reorderings are obtained by permuting the indices of the tensor. Figure D.6 shows the compression time and relative test error of the algorithms using various axis reorderings. Figure D.7 shows the comparison of compression time and mean validation error of the algorithms using various axis reorderings. Table D.2 presents the axis ordering, shapes of the corresponding tensors, and the resulting compression time, CR, RR, as well as RTE.

Figure D.6 shows that the baseline axis ordering results in the highest CR and RR among the investigated axis reorderings. The baseline axis ordering achieves $32.68\times$ CR and $32.94\times$ RR, whereas the second best axis ordering, Transpose A, achieves $26.24\times$ CR

Table D.2: *Results of experiments investigating the effect of axis ordering on the compression performance of the* `HT-RISE` *algorithm using the PDEBench 3D turbulent Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$. *Axis orderings are given in the form of a permutation of the indices of the tensor. Two of the dimensions with magnitude 8 are emphasized to make different axis reorderings distinguishable. The investigated axis reorderings of the axes result in up to* 1.72× *higher in compression time, CR and RR, while yielding almost the same RTE. Time: Compression time in seconds, CR: Compression ratio, RR: Reeduction ratio, RTE: Relative test error.*

Name	Axis Ordering	Tensor Shape	Time(s)	CR	RR	RTE
Baseline	[0,1,2,3,4,5,6,7,8]	$8 \times 8 \times \mathbf{8} \times 8 \times \mathit{8} \times 8 \times 5 \times 21 \times 1$	1158.7	32.68	32.94	0.098
Transpose A	[0,1,2,7,4,5,6,3,8]	$8 \times 8 \times \mathbf{8} \times 21 \times \mathit{8} \times 8 \times 5 \times 8 \times 1$	1306.7	26.24	26.41	0.098
Transpose B	[0,4,2,7,1,5,6,3,8]	$8 \times \mathit{8} \times 8 \times 21 \times \mathbf{8} \times 8 \times 5 \times 8 \times 1$	1928.7	18.93	19.06	0.097
Transpose C	[0,1,2,6,4,5,3,7,8]	$8 \times 8 \times \mathbf{8} \times 5 \times \mathit{8} \times 8 \times 8 \times 21 \times 1$	1992.9	25.59	26.21	0.098

and 26.41× RR. CR and RR drop down to 18.93× and 19.06× for the worst performing axis ordering, Transpose B.

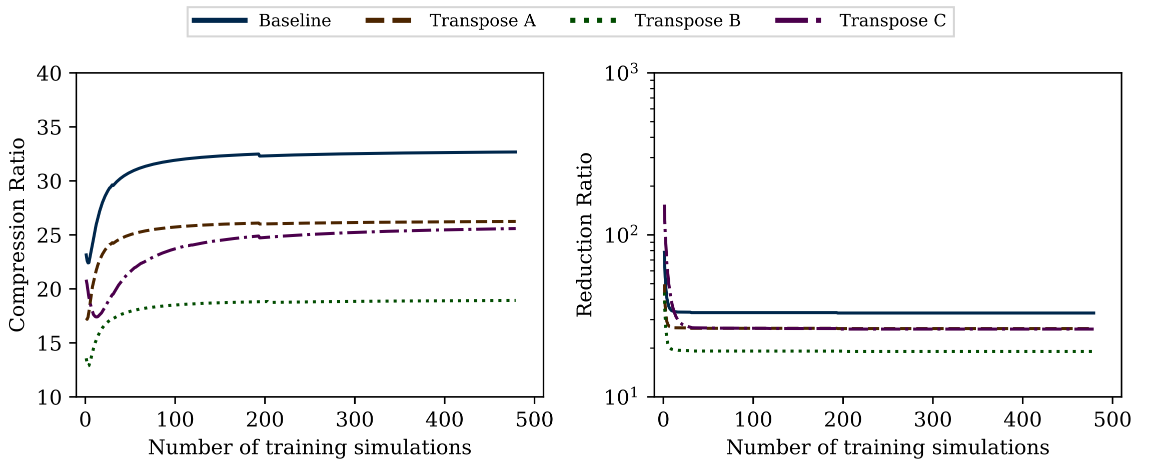


Figure D.6: *Comparison of various axis orderings in terms of compression ratio (CR - left) and reduction ratio (RR - right) on the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *using no normalization. The baseline axis ordering results in the highest CR and RR among the investigated axis reorderings. Changing the axis ordering of the tensor results in up to* 1.72× *the CR and RR of the worst performing axis ordering.*

Figure D.7 shows that the baseline axis ordering also results in the lowest compression time among the investigated axis reorderings. The baseline axis ordering achieves 1158.7 seconds of compression time, whereas the second best axis ordering, Transpose A, results in 1306.7 seconds of compression time. Compression time increases up to 1992.9 seconds for worst performing axis ordering, Transpose C. In terms of RTE, all axis reorderings yield almost the same RTE around 0.098.

D.4 Additional PDEBench results

This section presents the detailed results discussed in Section 4.2.2. Similar to Table 3, we present experiments on the PDEBench 3D turbulent Navier-Stokes simulations with

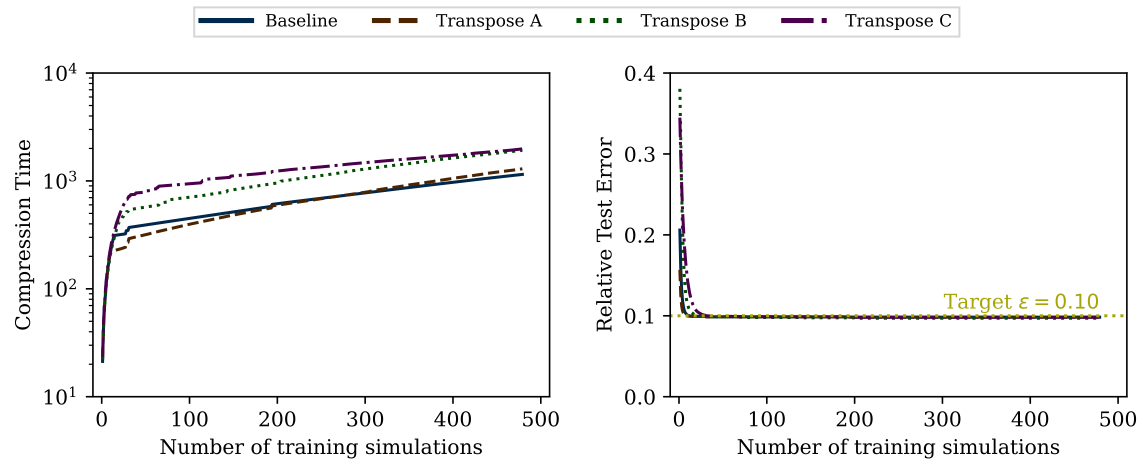


Figure D.7: *Comparison of various axis orderings in terms of compression time (left) and relative test error (RTE - right) on the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *using no normalization. The baseline axis ordering results in the lowest compression time among the investigated axis reorderings. Changing the axis ordering of the tensor results in up to* $1.72\times$ *the compression time of best performing axis ordering. All axis reorderings result in almost the same RTE.*

$\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.05$. In contrast to Section 4.2.2, here we provide results for all normalization methods used in the experiments. Figures D.8 and D.9 show the results for experiments with $\varepsilon_{rel} = 0.10$ and Figures D.10 and D.11 show the results for experiments with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.05$.

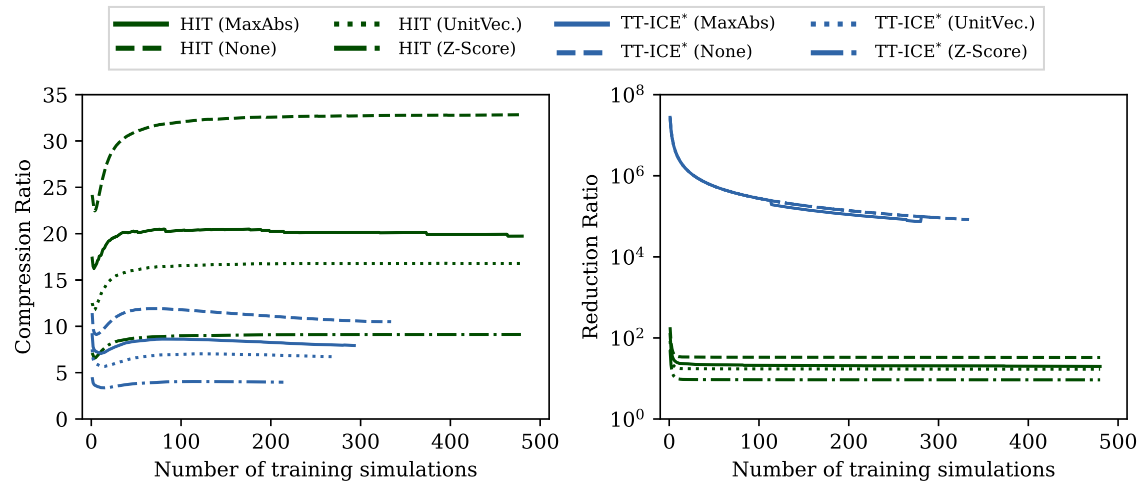


Figure D.8: *Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *and using various normalization methods.* `TT-ICE*` *offers orders of magnitude higher reduction ratio while yielding less compression ratio compared to* `HT-RISE`*. All experiments with* `TT-ICE*` *terminates prematurely due to timeout. MaxAbs: Maximum absolute value normalization, None: No normalization, UnitVec: Unit vector normalization, ZScore: Z-score normalization. The results are averaged over 5 seeds.*

In parallel to the findings in Figure 8, Figure D.8 shows that `HT-RISE` results in higher compression ratio but multiple orders of magnitude lower reduction ratio against `TT-ICE`*. This discrepancy is caused by the fact that the maximum size of the latent space is upper bounded by the number of tensors in the accumulation for `TT-ICE`*. This limitation also results in the high RTE for `TT-ICE`* as it struggles to reduce the approximation error on the test set anywhere near the target ε_{rel} in Figure D.9. On the other hand, `HT-RISE` is able to reduce the RTE below the target ε_{rel} error within a handful of training simulations across all normalization methods. As a result, `HT-RISE` achieves faster compression time across all normalization methods due to the less updates required to reach the target ε_{rel} error.

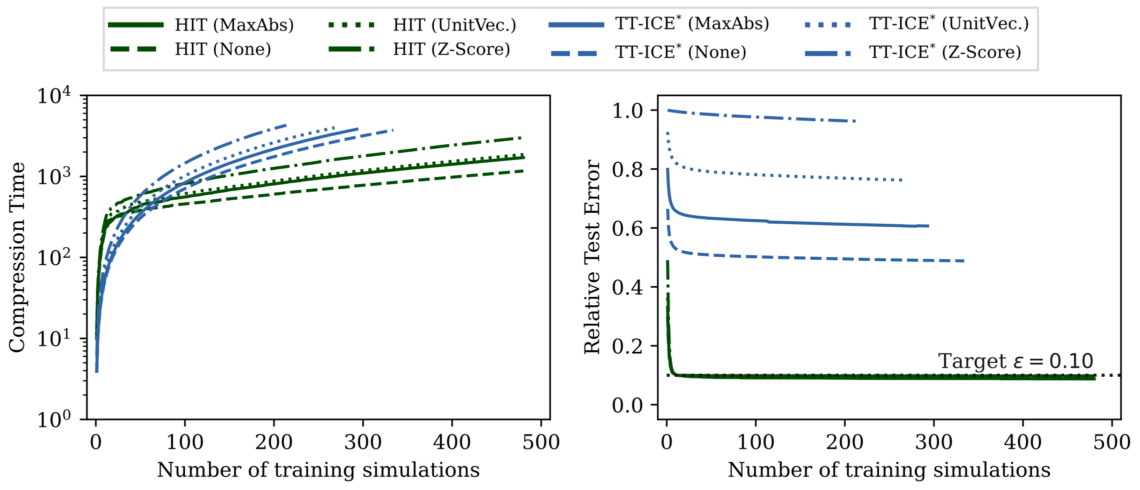


Figure D.9: *Comparison of compression time and mean validation error of algorithms using the PDEBench 3D Navier-Stokes dataset with* $\varepsilon_{rel} = 0.10$ *and using various normalization methods. TT-ICE** *fails to reduce the approximation error on the test set whereas `HT-RISE`achieves below* ε_{rel} *error within a handful of training simulations. This results in less updates to `HT-RISE`and therefore less compression time. All experiments with `TT-ICE`** *terminates prematurely due to timeout. Please refer to the legend of Figure D.8 for the normalization methods. The results are averaged over 5 seeds.*

A similar trend is observed in Figure D.10 and Figure D.11 for the experiments with $\varepsilon_{rel} = 0.05$. `TT-ICE`* offers orders of magnitude higher reduction ratio while yielding less compression ratio compared to `HT-RISE`. Similarly, `TT-ICE`* fails to reduce the approximation error on the test set whereas `HT-RISE` achieves below ε_{rel} error within a handful of training simulations. This issue becomes a serious bottleneck as `TT-ICE`* hardly compresses 200 simulations out of 480 during the allocated 4-day maximum walltime. On the other hand, `HT-RISE` only fails at z-score normalization due to insufficient memory.

D.5 Additional self-oscillating-gel simulations results

This section presents the detailed results discussed in Section 4.2.1. Similar to Table 2, we present experiments on the self oscillating gel simulations with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.01$. However, in this section we provide results for z-score normalization and unit vector

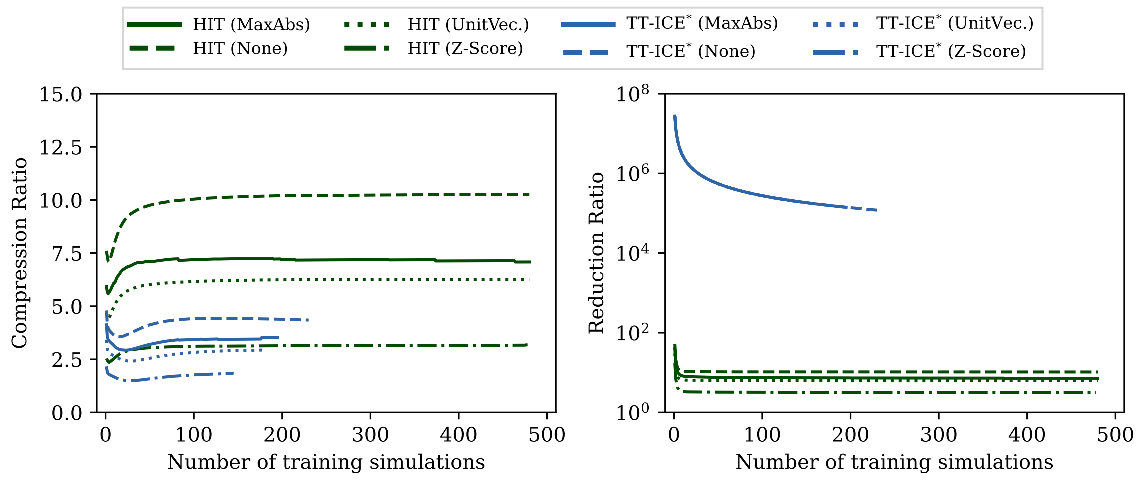


Figure D.10: *Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes dataset with $\varepsilon_{rel} = 0.05$ and using various normalization methods.* **`TT-ICE*`** *offers orders of magnitude higher reduction ratio while yielding less compression ratio compared to* **`HT-RISE`**. *All experiments with* **`TT-ICE*`** *terminates prematurely due to timeout.* **`HT-RISE`** *only fails at z-score normalization due to insufficient memory. Please refer to the legend of Figure D.8 for the normalization methods. The results are averaged over 5 seeds.*

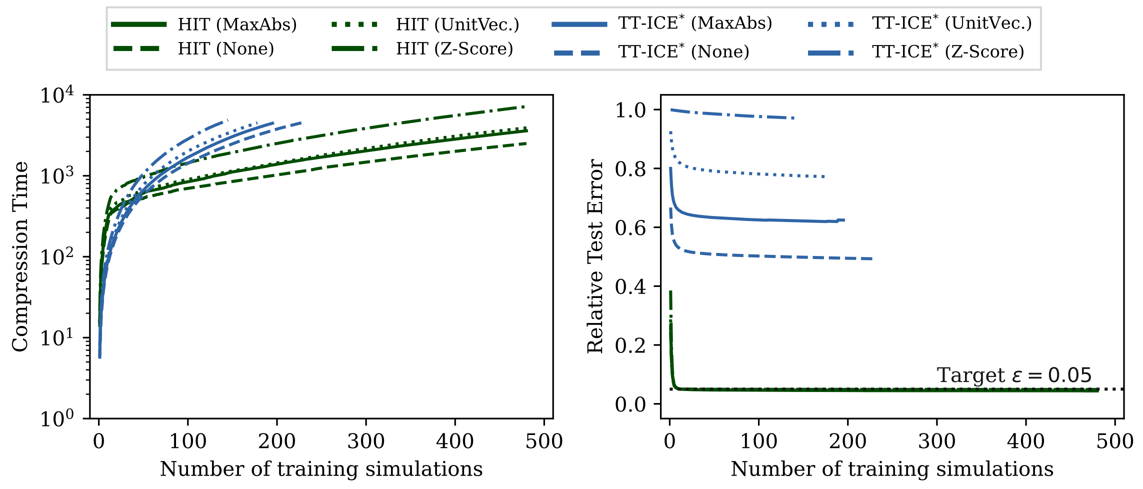


Figure D.11: *Comparison of compression time (left) and Relative Test Error (right) of algorithms using the PDEBench 3D Navier-Stokes dataset with $\varepsilon_{rel} = 0.05$ and using various normalization methods. TT-ICE* fails to reduce the approximation error on the test set whereas* `HT-RISE`*achieves below ε_{rel} error within a handful of training simulations. This results in less updates to* `HT-RISE`*and therefore less compression time. All experiments with* `TT-ICE*` *terminates prematurely due to timeout.* `HT-RISE` *only fails at z-score normalization due to insufficient memory. Please refer to the legend of Figure D.8 for the normalization methods. The results are averaged over 5 seeds.*

normalization in addition to experiments without any normalization. Figures D.12 and D.13 show the results for experiments with $\varepsilon_{rel} = 0.10$ and Figures D.14 and D.15 show the results for experiments with $\varepsilon_{rel} = 0.01$.

Figure D.12 shows the compression ratio and reduction ratio of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.10$ and using all investigated normalization methods. At this target relative error level, both algorithms successfully complete the compression task for all normalization methods. `TT-ICE`* offers $3.3 - 3.5\times$ the CR, and $4.6 - 6.3\times$ the RR of `HT-RISE`.

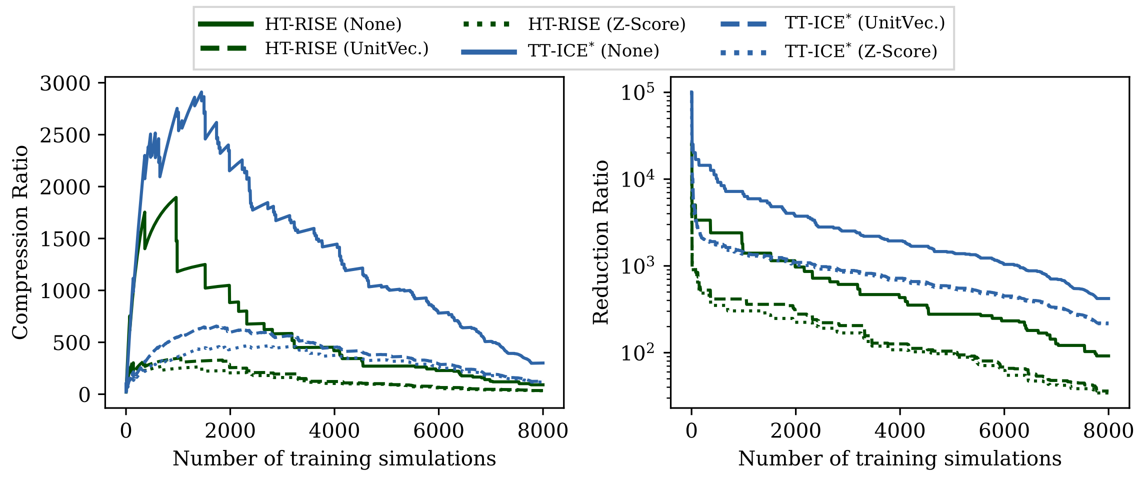


Figure D.12: *Compression ratio (CR - left) and Reduction ratio (RR - right) of the algorithms on the self-oscillating gel dataset with* $\varepsilon_{rel} = 0.10$ *and using various normalization methods.* ***`TT-ICE`**** *offers orders of magnitude higher reduction ratio while yielding less compression ratio compared to* ***`HT-RISE`****. All experiments with* ***`TT-ICE`**** *terminates prematurely due to timeout. MaxAbs: Maximum absolute value normalization, None: No normalization, UnitVec: Unit vector normalization, ZScore: Z-score normalization. The results are averaged over 5 seeds.*

Figure D.13 shows the compression time and relative test error of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.10$ with all investigated normalization methods. Both methods were able to reduce the approximation error below the target ε_{rel} threshold for all normalziation methods within similar number of training simulations. Due to the low dimensionality of the problem, `TT-ICE`* resulted in a lower compression time. `HT-RISE` takes $2.9 - 5.4\times$ the time it takes for `TT-ICE`* to compress the dataset.

Figure D.14 shows the compression ratio and reduction ratio of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.01$ and using all investigated normalization methods. As the target relative error level becomes tighter, the compression ratio and reduction ratio of the algorithms decrease significantly. Furthermore, `TT-ICE`* runs into timeout for unit vector and z-score normalizations. In contrast to that, `HT-RISE` completes the compression task for all normalization methods without running into any issues. For all normalization methods `TT-ICE`* offers $2.5 - 6.6\times$ the CR of `HT-RISE` and $9.5 - 21.8\times$ the RR of `HT-RISE`.

Figure D.15 shows the compression time and relative test error of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.01$ and using all investigated normalization methods. Both methods were able to reduce the approximation error below the target ε_{rel} threshold

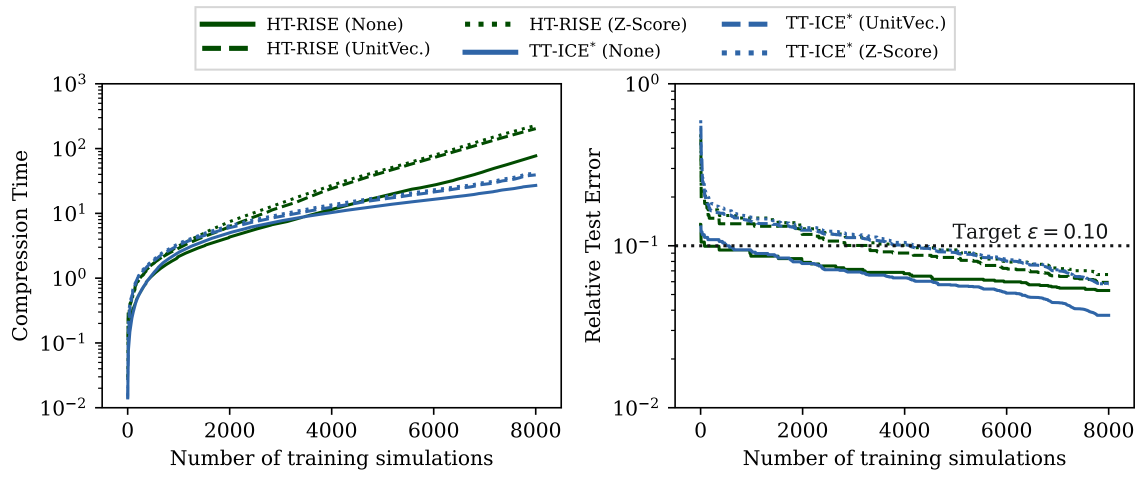


Figure D.13: *Compression time (left) and Relative Test Error (right) of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.10$ and using various normalization methods. TT-ICE* offers orders of magnitude higher reduction ratio while yielding less compression ratio compared to* ***`HT-RISE`****. All experiments with* ***`TT-ICE`***** terminates prematurely due to timeout. Please refer to the caption of Figure D.12 for normalization methods. The results are averaged over 5 seeds.*

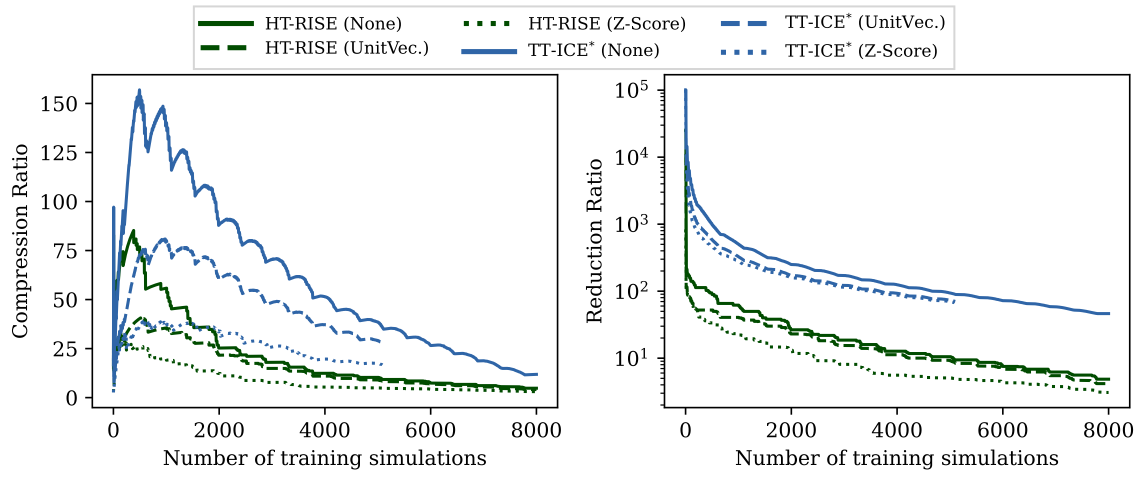


Figure D.14: *Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.01$ and using various normalization methods. None: No normalization, UnitVec: Unit vector normalization, ZScore: Z-score normalization. TT-ICE* offers an order of magnitude higher reduction ratio while yielding comparable compression ratio compared to* ***`HT-RISE`****.*

when no normalization is employed. In addition to that, `HT-RISE` was able to cross the target ε_{rel} line with z-score normalization as well. It is not fair to compare the methods in terms of total compression time as `TT-ICE`* fails to compress the entire dataset due to the maximum walltime limit at two of the normalization methods. However, it is worth noting that in Figure D.15 `TT-ICE`* runs faster than `HT-RISE` for all normalization methods for same number of simulations. Note that only `HT-RISE` with z-score normalization can cross the target ε_{rel} line in addition to the cases with no normalization.

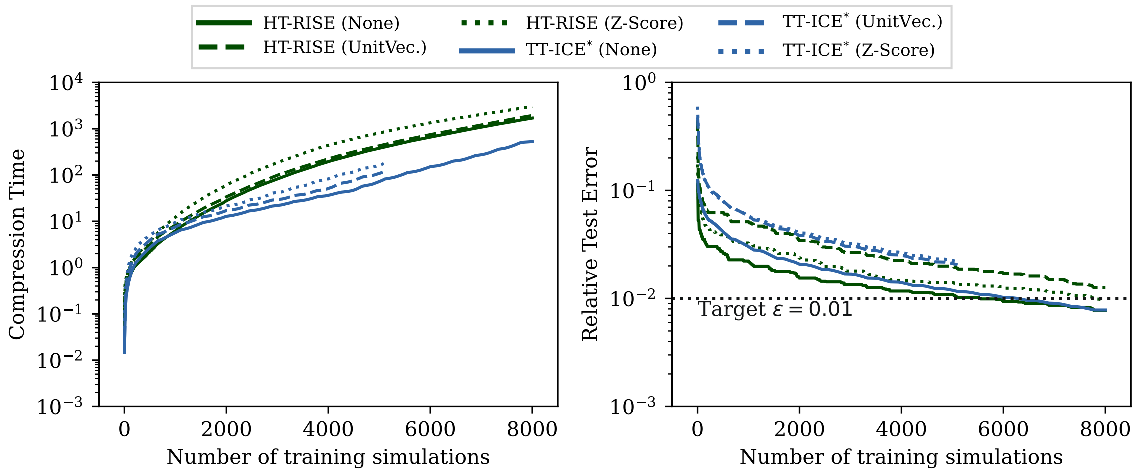


Figure D.15: *Comparison of compression time (left) and Relative test error (right) of algorithms using the Self-oscillating gel simulation dataset with $\varepsilon_{rel} = 0.01$ and using various normalization methods. Both methods were able to reduce the approximation error below the target ε_{rel} threshold when no normalization is employed. In addition to that, HT-RISE was able to cross the target ε_{rel} line with z-score normalization as well. Due to the low dimensionality of the problem, TT-ICE* resulted in a lower compression time. Please refer to the legend of Figure D.14 for the normalization methods.*

D.6 Additional BigEarthNet results

This section presents the detailed results discussed in Section 4.3.2. Similar to Table 6, we present experiments on the BigEarthNet dataset with $\varepsilon_{rel} = 0.05 - 0.30$. Figure D.16 presents the results of the experiments on compression ratio and reduction ratio of the algorithms whereas Figure D.17 shows the results of the experiments considering compression time and relative test error of the algorithms.

Figure D.16 shows that at high target relative error levels, `TT-ICE`* offers higher CR and RR than `HT-RISE`. At $\varepsilon_{rel} = 0.30$, `TT-ICE`* achieves 2274× CR and 1901× RR whereas `HT-RISE` achieves 1154× CR and 962× RR. The discrepancy between two algorithms reduces as the target relative error level decreases. At $\varepsilon_{rel} = 0.15$, `TT-ICE`* achieves 117× CR and 100× RR whereas `HT-RISE` achieves 91× CR and 76× RR. At $\varepsilon_{rel} = 0.10$, the performance of the two algorithms become even more comparable, where `TT-ICE`* achieves 35× CR and 31× RR whereas `HT-RISE` achieves 32× CR and 27× RR. At these target relative error levels, neither algorithm stuggles to compress the dataset. However at $\varepsilon_{rel} = 0.05$, `TT-ICE`*

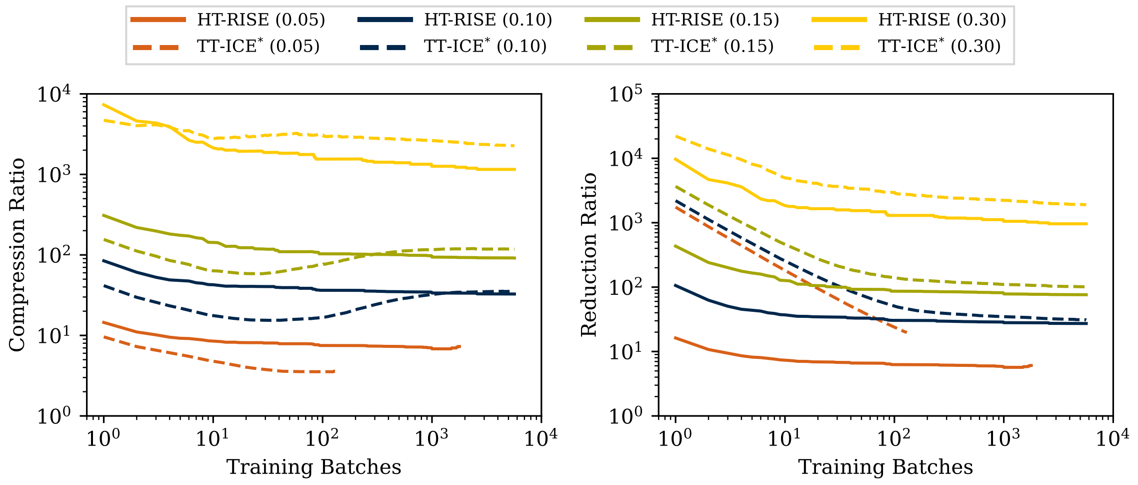


Figure D.16: *Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the BigEarthNet dataset with* $\varepsilon_{rel} = 0.05 - 0.30$*.* ***TT-ICE**** *offers higher CR* ($1.1 - 2\times$ *that of* ***HT-RISE****) for all target relative error levels but* $\varepsilon_{rel} = 0.05$*. At* $\varepsilon_{rel} = 0.05$*,* ***HT-RISE*** *achieves* $2\times$ *the CR of* ***TT-ICE**** *at the portion that it* ***TT-ICE**** *is able to compress.* ***TT-ICE**** *also offers higher RR* ($1.1 - 2\times$ *that of* ***HT-RISE****) for* $\varepsilon_{rel} = 0.10 - 0.30$*. The results are averaged over 5 seeds.*

runs into maximum walltime timeout and therefore fails to compress the entire dataset and HT-RISE runs into maximum memory limit and therefore fails to compress the entire dataset. The portion of the dataset that TT-ICE* is able to compress results in $3.59\times$ CR and $19.76\times$ RR whereas HT-RISE results in $7.29\times$ CR and $6.07\times$ RR. However, we need to acknowledge that the portion of the dataset that TT-ICE* is able to compress is significantly smaller than the portion of the dataset that HT-RISE is able to compress.

Figure D.17 shows that irrespective of the target relative error level, HT-RISE results in lower, if not comparable, compression time compared to TT-ICE*. At $\varepsilon_{rel} = 0.30$ HT-RISE compresses the entire dataset in 2482s whereas TT-ICE* compresses the entire dataset in 6239s. At $\varepsilon_{rel} = 0.15$ the difference becomes less pronounced as HT-RISE compresses the entire dataset in $14,896$s whereas TT-ICE* compresses the entire dataset in $18,866$s. At $\varepsilon_{rel} = 0.10$ TT-ICE* becomes slightly faster as HT-RISE compresses the entire dataset in $49,408$s and TT-ICE* compresses the entire dataset in $44,687$s. As discussed above, at $\varepsilon_{rel} = 0.05$ neither algorithm is able to compress the entire dataset. At $\varepsilon_{rel} = 0.05$, TT-ICE* compresses 129 batches in 1720s whereas in the same amount of time HT-RISE compresses 443 batches. At the point of failure, HT-RISE compresses 1781 batches in $20,880$s.

D.7 Qualitative MineRL results

This section presents a qualitative comparison of the compressed frames using reconstructions. Figure D.18 shows the reconstructed video frames from the Basalt MineRL competition dataset using TT-ICE* and HT-RISE algorithms with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.30$. In Figure D.18, we observe that HT-RISE in fact results in a better visual quality compared to the frame of TT-ICE* of the same target relative error level. Despite the fact that TT-ICE* achieves almost $3\times$ the CR and the RR of HT-RISE, the visual quality of the frame re-

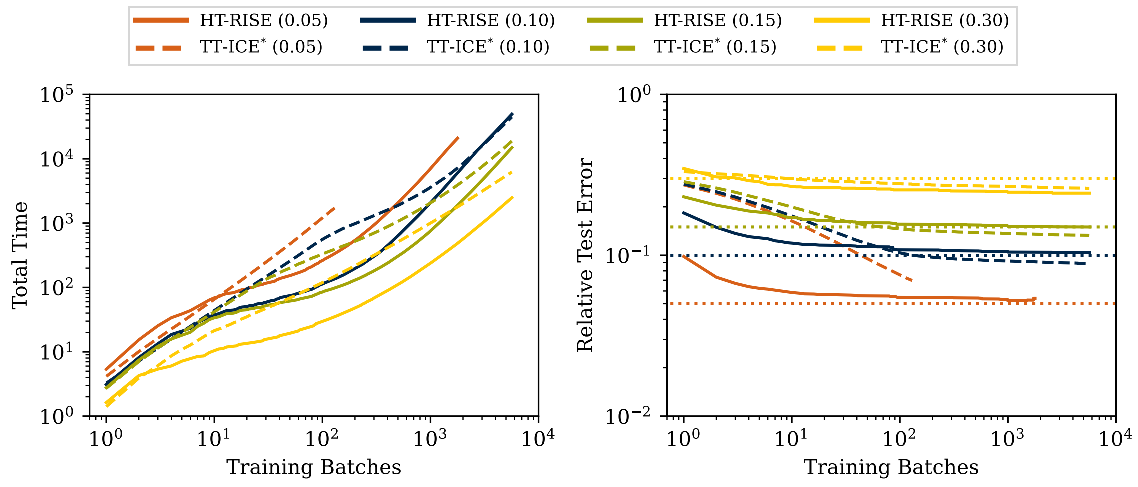
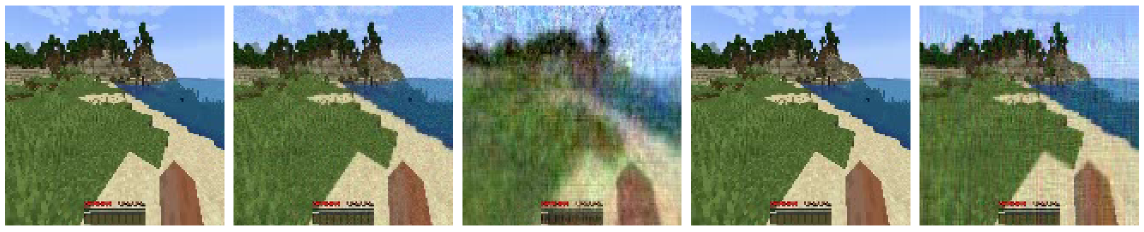


Figure D.17: *Compression time (left) and relative test error (RTE - right) of the algorithms on the BigEarthNet dataset with $\varepsilon_{rel} = 0.05 - 0.30$. For all ε_{rel} levels,* ***HT-RISE*** *results in lower, if not comparable, compression time compared to* ***TT-ICE******. For all ε_{rel} levels,* ***HT-RISE*** *is able to reduce the RTE faster than* ***TT-ICE******. Each target relative error level is represented with horizontal dotted lines in their respective colors. The results are averaged over 5 seeds.*



(a) *Downsampled* (b) $\varepsilon_{rel} = 0.10$ (c) *TT* $\varepsilon_{rel} = 0.30$ (d) *HT* $\varepsilon_{rel} = 0.10$ (e) *HT* $\varepsilon_{rel} = 0.30$

Figure D.18: *Reconstructed video frames from Basalt MineRL competition dataset using* ***TT-ICE***** (TT) and* ***HT-RISE*** *(HT) algorithms with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.30$. The downsampled frame is provided for baseline comparison. Visual quality of the reconstructed frames is comparable to the downsampled frames except for the case with* ***TT-ICE***** at $\varepsilon_{rel} = 0.30$. Please refer to Table 4 and Figures 11 and 12 for quantitative results.*

constructed by `TT-ICE`* at $\varepsilon_{rel} = 0.30$ is significantly worse than the downsampled frame. The reconstruction of `TT-ICE`* at $\varepsilon_{rel} = 0.30$ is significantly blurrier compared to the reconstruction of `HT-RISE` at $\varepsilon_{rel} = 0.30$. In addition to that, we see that the visual quality of the reconstruction from `HT-RISE` at $\varepsilon_{rel} = 0.10$ is comparable, if not identical, to the downsampled frame even though `HT-RISE` offers a CR and a RR around $1.85\times$.

References

Doruk Aksoy, Silas Alben, Robert D Deegan, and Alex A Gorodetsky. Inverse design of self-oscillatory gels through deep learning. *Neural Computing and Applications*, 34(9): 6879–6905, 2022.

Doruk Aksoy, David J Gorsich, Shravan Veerapaneni, and Alex A Gorodetsky. An incremental tensor train decomposition algorithm. *SIAM Journal on Scientific Computing*, 46 (2):A1047–A1075, 2024a.

Doruk Aksoy, Sruti Vutukury, Thomas A. Marks, Joshua D. Eckels, and Alex A. Gorodetsky. Compressed analysis of electric propulsion simulations using low rank tensor networks. 2024b.

Silas Alben, Alex A Gorodetsky, Donghak Kim, and Robert D Deegan. Semi-implicit methods for the dynamics of elastic sheets. *Journal of Computational Physics*, 399: 108952, 2019.

Bowen Baker, Ilge Akkaya, Peter Zhokov, Joost Huizinga, Jie Tang, Adrien Ecoffet, Brandon Houghton, Raul Sampedro, and Jeff Clune. Video pretraining (vpt): Learning to act by watching unlabeled online videos. *Advances in Neural Information Processing Systems*, 35:24639–24654, 2022.

J Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition. *Psychometrika*, 35 (3):283–319, 1970.

Dimitris G Chachlakis, Ashley Prater-Bennette, and Panos P Markopoulos. L1-norm higher-order orthogonal iterations for robust tensor analysis. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4826–4830. IEEE, 2020.

Maolin Che and Yimin Wei. Randomized algorithms for the approximations of tucker and the tensor train decompositions. *Advances in Computational Mathematics*, 45(1):395–428, 2019.

Brian Chen, Doruk Aksoy, David J Gorsich, Shravan Veerapaneni, and Alex Gorodetsky. Low-rank tensor-network encodings for video-to-action behavioral cloning. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL `https://openreview.net/forum?id=w4DXLzBPPw`.

Saibal De, Zitong Li, Hemanth Kolla, and Eric T Phipps. Efficient computation of tucker decomposition for streaming scientific data compression. *arXiv preprint arXiv:2308.16395*, 2023.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors. *SIAM journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000a.

Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000b.

Lars Eldén and Berkant Savas. A newton–grassmann method for computing the best multilinear rank-(r_1, r_2, r_3) approximation of a tensor. *SIAM Journal on Matrix Analysis and applications*, 31(2):248–271, 2009.

Lars Grasedyck. Hierarchical singular value decomposition of tensors. *SIAM journal on matrix analysis and applications*, 31(4):2029–2054, 2010.

Charles R. Harris, K. Jarrod Millman, Stéfan J.van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020. doi: 10.1038/s41586-020-2649-2. URL `https://doi.org/10.1038/s41586-020-2649-2`.

Richard A Harshman et al. Foundations of the parafac procedure: Models and conditions for an "explanatory" multi-modal factor analysis. *UCLA working papers in phonetics*, 16 (1):84, 1970.

Jean Kossaifi, Nikola Kovachki, Kamyar Azizzadenesheli, and Anima Anandkumar. Multi-grid tensorized fourier neural operator for high-resolution pdes. *arXiv preprint arXiv:2310.00120*, 2023.

Daniel Kressner and Lana Perisa. Recompression of hadamard products of tensors in tucker format. *SIAM Journal on Scientific Computing*, 39(5):A1879–A1902, 2017.

Daniel Kressner and Christine Tobler. Algorithm 941: Htucker—a matlab toolbox for tensors in hierarchical tucker format. *ACM Transactions on Mathematical Software (TOMS)*, 40(3):1–22, 2014.

Daniel Kressner, Bart Vandereycken, and Rik Voorhaar. Streaming tensor train approximation. *SIAM Journal on Scientific Computing*, 45(5):A2610–A2631, 2023.

Pieter M Kroonenberg and Jan De Leeuw. Principal component analysis of three-mode data by means of alternating least squares algorithms. *Psychometrika*, 45:69–97, 1980.

Isabell Lehmann, Evrim Acar, Tanuj Hasija, Vince D Calhoun, Peter J Schreier, and Tülay Adali. Multi-task fmri data fusion using iva and parafac2. In *ICASSP 2022-2022 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1466–1470. IEEE, 2022.

Huazhong Liu, Laurence T Yang, Yimu Guo, Xia Xie, and Jianhua Ma. An incremental tensor-train decomposition for cyber-physical-social big data. *IEEE Transactions on Big Data*, 7(2):341–354, 2018.

Yiran Luo, Het Patel, Yu Fu, Dawon Ahn, Jia Chen, Yue Dong, and Evangelos E Papalexakis. Trawl: Tensor reduced and approximated weights for large language models. *arXiv preprint arXiv:2406.17261*, 2024.

Qing Mai and Xin Zhang. *Statistical Methods for Tensor Data Analysis*, pages 817–829. Springer London, London, 2023. ISBN 978-1-4471-7503-2. doi: 10.1007/978-1-4471-7503-2_39. URL `https://doi.org/10.1007/978-1-4471-7503-2_39`.

Osman Asif Malik and Stephen Becker. Low-rank tucker decomposition of large tensors using tensorsketch. *Advances in neural information processing systems*, 31, 2018.

Thomas Marks and Alex Gorodetsky. Hall thruster simulations in warpx. 2024.

Stephanie Milani, Anssi Kanervisto, Karolis Ramanauskas, Sander Schulhoff, Brandon Houghton, and Rohin Shah. Bedd: The minerl basalt evaluation and demonstrations dataset for training and benchmarking agents that solve fuzzy tasks. *Advances in Neural Information Processing Systems*, 36, 2024.

Rachel Minster, Zitong Li, and Grey Ballard. Parallel randomized tucker decomposition algorithms. *arXiv preprint arXiv:2211.13028*, 2022.

Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33 (5):2295–2317, 2011.

Ivan V Oseledets and Eugene E Tyrtyshnikov. Breaking the curse of dimensionality, or how to use svd in many dimensions. *SIAM Journal on Scientific Computing*, 31(5):3744–3759, 2009.

Yannis Panagakis, Jean Kossaifi, Grigorios G Chrysos, James Oldfield, Mihalis A Nicolaou, Anima Anandkumar, and Stefanos Zafeiriou. Tensor methods in computer vision and deep learning. *Proceedings of the IEEE*, 109(5):863–890, 2021.

Tobias Pfaff, Meire Fortunato, Alvaro Sanchez-Gonzalez, and Peter W Battaglia. Learning mesh-based simulation with graph networks. *arXiv preprint arXiv:2010.03409*, 2020.

Shaden Smith, Kejun Huang, Nicholas D Sidiropoulos, and George Karypis. Streaming tensor factorization for infinite data sources. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 81–89. SIAM, 2018.

Gencer Sumbul, Marcela Charfuelan, Begüm Demir, and Volker Markl. Bigearthnet: A large-scale benchmark archive for remote sensing image understanding. In *IGARSS 2019-2019 IEEE International Geoscience and Remote Sensing Symposium*, pages 5901–5904. IEEE, 2019.

Gencer Sumbul, Arne De Wall, Tristan Kreuziger, Filipe Marcelino, Hugo Costa, Pedro Benevides, Mario Caetano, Begüm Demir, and Volker Markl. Bigearthnet-mm: A large-scale, multimodal, multilabel benchmark archive for remote sensing image classification and retrieval [software and data sets]. *IEEE Geoscience and Remote Sensing Magazine*, 9(3):174–180, 2021.

Yiming Sun, Yang Guo, Charlene Luo, Joel Tropp, and Madeleine Udell. Low-rank tucker approximation of a tensor from streaming data. *SIAM Journal on Mathematics of Data Science*, 2(4):1123–1150, 2020.

Makoto Takamoto, Timothy Praditia, Raphael Leiteritz, Daniel MacKinlay, Francesco Alesiani, Dirk Pflüger, and Mathias Niepert. Pdebench: An extensive benchmark for scientific machine learning. *Advances in Neural Information Processing Systems*, 35:1596–1611, 2022.

Fan Tian, Misha E Kilmer, Eric Miller, and Abani Patra. Tensor bm-decomposition for compression and analysis of spatio-temporal third-order data. *arXiv preprint arXiv:2306.09201*, 2023.

Charalampos E Tsourakakis. Mach: Fast randomized tensor decompositions. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 689–700. SIAM, 2010.

Ledyard R Tucker. Implications of factor analysis of three-way matrices for measurement of change. *Problems in measuring change*, 15(122-137):3, 1963.

Ledyard R Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.

Ledyard R Tucker et al. The extension of factor analysis to three-dimensional matrices. *Contributions to mathematical psychology*, 110119, 1964.

Nick Vannieuwenhoven, Raf Vandebril, and Karl Meerbergen. A new truncation strategy for the higher-order singular value decomposition. *SIAM Journal on Scientific Computing*, 34(2):A1027–A1052, 2012.

Fuxi Wen and Hing Cheung So. Robust multi-dimensional harmonic retrieval using iteratively reweighted hosvd. *IEEE Signal Processing Letters*, 22(12):2464–2468, 2015.

Yifan Yang, Jiajun Zhou, Ngai Wong, and Zheng Zhang. Loretta: Low-rank economic tensor-train adaptation for ultra-low-parameter fine-tuning of large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3161–3176, 2024.

Chao Zeng and Michael K Ng. Incremental cp tensor decomposition by alternating minimization method. *SIAM Journal on Matrix Analysis and Applications*, 42(2):832–858, 2021.

Jiani Zhang, Arvind K Saibaba, Misha E Kilmer, and Shuchin Aeron. A randomized tensor singular value decomposition based on the t-product. *Numerical Linear Algebra with Applications*, 25(5):e2179, 2018.

Zhengwu Zhang, Genevera I Allen, Hongtu Zhu, and David Dunson. Tensor network factorizations: Relationships between brain structural connectomes and traits. *Neuroimage*, 197:330–343, 2019.

Guoxu Zhou, Andrzej Cichocki, and Shengli Xie. Decomposition of big tensors with low multilinear rank. *arXiv preprint arXiv:1412.1885*, 2014.