Incremental Tensor Decompositions for Scientific Machine Learning and Bayesian Inference

by

Doruk Aksoy

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Aerospace Engineering and Scientific Computing)
in The University of Michigan
2025

Doctoral Committee:

Professor Alex A. Gorodetsky, Chair
Professor Laura Balzano
Professor Karthik Duraisamy
Professor Shravan Veerapaneni

Doruk Aksoy

doruk@umich.edu

ORCID iD: 0000-0002-6828-499X

© Doruk Aksoy 2025

To my families, both by blood and by choice...

ACKNOWLEDGEMENTS

This work would not be possible without the support and guidance of my adviser, *Prof. Alex Arkady Gorodetsky*, who has been a constant source of growth and inspiration throughout my academic journey. I am deeply thankful and grateful to him for giving me a chance to work with him and always pushing me to be better. I will carry the values and lessons I have learned from him throughout my career. I am also grateful to *Prof. Shravan Veerapaneni* for his invaluable insights and support over many years. I would also like to thank my committee members, *Prof. Laura Balzano* and *Prof. Karthik Duraisamy*, for their constructive feedback and support on this thesis. Their expertise and vision will be instrumental in shaping the future of this work.

Over the years, I have had the pleasure of working with many *very* bright individuals at the University of Michigan. Maybe the most important of these is my dear friend and my first collaborator *Dr. Efe Can Balta*, who has been a constant source of support and mentorship both at the very beginning of my academic career and throughout my time at the University of Michigan. Although the work we did is not within the scope of this thesis, I believe that the work we did together has been essential in starting my academic career.

During my PhD journey, I have collaborated with many others who have been instrumental in either making this work possible or making it better. First and foremost, I would like to thank *Dr. David J. Gorsich* for his unwavering support for my work and for tensor methods in general. I would also like to thank my collaborators *Dr. Brian Chen*, *Mihir Vador*, *Dr. Mark J. Brudnak*, *Sruti Vutukury*, *Dr. Thomas A. Marks*, *Joshua D. Eckels*, *Pranav Bahl*, and *Motaz AbuHijleh* for being a part of the research presented in this thesis.

Now comes the hardest and most important part of the acknowledgments. I never thought it would be this hard to thank people but writing this section was an interesting walk down the memory lane. I would like to start the looong list of people I would like to thank with my friends back home. I would like to start that list with my childhood friends *Cem, Murat, Kansu* and *Ezgi*, who have been with me for almost as long as I can remember, thank you guys for growing up together. I would also like to thank my friends from my high school years, *Anıl G., Anıl K.,* and *Deniz*, for always being there. From my high school years, there has to be a special mention for *Kerem, Can, Dünya,* and *Saruhan.* Thank you for providing

me entertainment and an immeasurable amount of banter on a daily basis to this day, I always miss watching Galatasaray matches with you at Gate 415 and chant until we lose our voices.

Between the coursework and spending an ungodly amount of time for BuALERT, I made a lot of great friends during my years at Boğaziçi including *Oğuz, Oğuzhan, Letisya, İlke* and *Sezer*. A special mention goes to *Yiğit* and *Sezin* for almost never refusing my offers for a pint and listening to my rants about life, work, and everything in between. I thank you all for being amazing friends.

When I left İstanbul in 2018 I had a bumpy relationship with Ann Arbor. Especially in my first years I had a lot more downs than ups. I would like to thank *İrem* and *Onur C.* for helping me get through those times by always being on the other side of the phone/screen listening to me. Another important support mechanism I had in my first year was *Andy*, I thank him for never turning down my offers for a walk around Munger.

Luckily, I was surrounded with amazing people in Ann Arbor. I am grateful to them for their generosity, kindness, and support.[1] Thank you *Yunus* for letting me crash at your place for a while, I am grateful for your generosity and friendship. Thank you *Efe* and *Yeliz* for countless drinks, advice and emotional support, even at times I didn't know I needed. Thank you *Anil A.* for being an amazing roommate and an older brother to me. Thank you *Anil Y.* for our classic "wednesday" evenings and listening me ranting about the most random stuff for hours. Also thank you for making me apply to UM in the first place. Thank you *Didem D.* for being an amazing listener and keeping me company while Anıl and Rüya are lost in their heated debates. Thank you *Ahmet* for being "the pitmaster" and the star of our cooking extravaganzas. My Turkish family in Ann Arbor would not be complete without mentioning *Koray, Ege, Hüseyin, Onur B., Alperen, Doğan, Çağıl* and *Didem I.* I thank you all for being a part of my journey and making my time in Ann Arbor **so much more** enjoyable.

I was surrounded by amazing people both in the Aerospace Engineering department, and in the North Campus. A huge mention there goes to *Nick* for being an amazing duo in the office, and for being an amazing travel partner during each conference trip. Another important person on my journey was *Sahil*, who has been a great friend and a source of inspiration by being the textbook definition of "getting things done". I would also like to thank *Sanjan* for being an amazing friend and a great source of fun even during our limited time together. I am grateful for all the times we spent together, whether it was working on research, discussing life, or just enjoying a beer. I also thank all past and present members of the Computational Autonomy group *Artem, Aditya, Thomas D., Josh, Liliang, Thomas*

[1] and also bearing with me during a quite dark period of my life when I had long hair...

M., Pranav, Carleen and *Trung*. Other than my colleagues, I was lucky enough to have countless chats with *Alice, Mohammad, Saja, Thomas C., Marco, Vansh, Dom, Ilya, Isaac, Vyas, Arkady, Tanya, Miriam*, and many more. Thank you all mostly for putting up with my dark sense of humor and my random rants about life.

Before proceeding to the most emotional part of section, I would like to express my heartfelt gratitude towards Mr. William's fine establishment, also known as *Bill's Beer Garden*, for being a sanctuary for me and being an important pillar of my mental stability (and also for the countless discounted pints I had there).

Over the years, I had to say farewell to some of my family members but also welcomed new ones. This long list of thank you's would be incomplete if I did not mention my late grandparents, *Ayla, Cahit,* and *Naciye*, I hope they are proud of me from wherever they are. Especially my grandmother, *Ayla*, whom I promised to be a "Doctor" when I was preparing for my university entrance exam. I know that this is not the kind of doctor she had in mind but I would like to believe that she is proud of me nonetheless. On the other hand, I would like to mention the new additions my chosen family, *Kemal, Aras, Deniz, Lumi, Can* and *Ela*. I hope you have a great life ahead of you and I hope I can be a good uncle to you, even from afar.

There is one person that I want to thank for having an immeasurable role in my upbringing. That person is no one other than my "cicannem" *Ünzile*, who took care of me every day since I was nine months old. Thank you for treating me with such care and compassion as if I am your own child. None of this would be possible without two people: my parents, *Hasan* and *Güzin*. I couldn't even imagine how hard it must have been for them to see me go to the other side of the world, only seeing me once or maybe twice a year. There is nothing in this world I could do to repay them for even a fraction of what they have done for me. I am grateful for their unconditional love, limitless support, and unwavering belief in me. At my graduation ceremony from Boğaziçi, I had a banner that read "*Anne söz bu tören son*"[2]. I guess this time I can promise that this is the last one mom. I am grateful for everything both of you have done for me, from the very first day I was born to this day. I am grateful for your sacrifices, your love, and your support. I am grateful for the values you have instilled in me, making me the person I am today. I owe everything I have to you, iyi ki varsınız.

My better half, my partner in crime, my love, *Rüya*, I am grateful for everything you have done for me. I am lucky to have you in my life. I thank your parents *Sevin* and *Mustafa Kemal* for raising such an incredible human being. Thank you Rüya for being my anchor, my sole source of strength, and my biggest supporter. Thank you for always being there for me, even when I was at my lowest. Thank you for cooking me delicious meals, for taking an

[2] "Mom, this is the last ceremony, I promise" in English.

amazing care of me when I needed, for always being there to listen to the same rants over and over again, and for always inspiring me to stand up and keep going when I felt at rock bottom. Thank you for always believing in me, even when I didn't believe in myself. Thank you for cultivating a sense of aesthetics in me, helping me prepare countless figures for my papers, presentations, and posters. If any of my works look good to the eye, it is thanks to you. Thank you for being the source of joy, light, and happiness in my life. Thank you for being my goofball. I am grateful for all the moments we have shared, and I look forward to creating many more memories together. *I love you.*

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES x

LIST OF TABLES xxiv

LIST OF ALGORITHMS xxix

LIST OF APPENDICES xxx

LIST OF ACRONYMS xxxi

ABSTRACT xxxiii

CHAPTER

1 Introduction **1**

1.1 Motivation 1
1.1.1 The Curse of Dimensionality 2
1.1.2 The Challenge of Large-Scale Data 4
1.1.3 Goal of the Thesis 7
1.2 Dimensionality Reduction Methods 8
1.2.1 Linear Methods 9
1.2.2 Non-linear Methods 13
1.2.3 Deep Learning Based Methods 15
1.2.4 Tensor Decompositions 16
1.2.5 Comparison of Dimensionality Reduction Methods 18
1.2.6 Issues with the Existing Methods 19
1.3 Contributions of the Thesis 21
1.4 Outline and Structure of the Thesis 22

2 Tensors and Tensor Decompositions **23**

2.1 Background on Tensors and Tensor Algebra 23
2.2 Tensor Decomposition Formats 29
2.2.1 CANDECOMP/PARAFAC (CP) Format 29
2.2.2 Tucker Format 31

2.2.3 Tensor-Train (TT) Format . . . 33
2.2.4 Hierarchical Tucker (HT) Format . . . 35
2.2.5 Batch-HT (BHT) Format . . . 37
2.2.6 Other Tensor Decomposition Formats . . . 38
2.2.7 Summary of Tensor Decomposition Formats . . . 39
2.3 Incremental Tensor Decomposition Algorithms . . . 41
2.3.1 CP Format . . . 41
2.3.2 Tucker Format . . . 41
2.3.3 Tensor Train Format . . . 42
2.3.4 Hierarchical Tucker Format . . . 45
2.4 Summary . . . 45

3 Proposed Algorithms . . . 46

3.1 The TT-ICE Algorithm . . . 46
3.1.1 Limitations of Existing Approaches . . . 47
3.1.2 The Algorithm . . . 48
3.1.3 Experiments . . . 59
3.1.4 Datasets and Experiments . . . 60
3.2 Batch Hierarchical Tucker format and the BHT-l2r algorithm . . . 68
3.2.1 Details on the Hierarchical Tucker Format . . . 69
3.2.2 Batch Hierarchical Tucker Format . . . 73
3.2.3 Comparison of the Batch Hierarchical Tucker Format and the Hierarchical Tucker Format . . . 75
3.3 The HT-RISE Algorithm . . . 78
3.3.1 Numerical Experiments . . . 84
3.3.2 Numerical Experiments with Scientific Data . . . 85
3.3.3 Numerical Experiments with Image Data . . . 90
3.4 Summary . . . 96
3.5 Acknowledgments . . . 99

4 Applications Considering Scientific Data . . . 103

4.1 Compression of Particle-in-Cell Simulations . . . 103
4.1.1 SPT-100 Plume Simulation Test Case . . . 105
4.1.2 Data Compression Pipeline . . . 109
4.1.3 Data Normalization . . . 109
4.1.4 Compression Results . . . 110
4.2 Bayesian Inference in Tensor-Network Latent Spaces . . . 118
4.2.1 Methodology . . . 119
4.2.2 Experimental Setup . . . 122
4.2.3 Results and Discussion . . . 123
4.3 Summary . . . 124
4.4 Acknowledgements . . . 125

5 Applications Considering Image and Video Data . . . 126

5.1 Compression of High-Resolution Images . . . 126

5.1.1 The Unreal Engine 5 Environments . . . 127
5.1.2 Compressing Camera Streams . . . 128
5.2 Behavioral Cloning using Tensor-Network Embeddings . . . 137
5.2.1 Related Works . . . 139
5.2.2 Methodology . . . 140
5.2.3 Mapping Unseen Data to Latent Space . . . 143
5.2.4 Action Predictor . . . 144
5.2.5 Experiments . . . 145
5.2.6 Results . . . 149
5.3 Generative Diffusion Models using Tensor-Network Embeddings . . . 157
5.3.1 Diffusion Models and Problem Set-Up . . . 158
5.3.2 Dataset, Architecture, and Training . . . 161
5.3.3 Results and Discussion . . . 164
5.4 Summary . . . 166
5.5 Acknowledgements . . . 169
6 Conclusion and Future Work . . . 170
6.1 Discussion of Contributions and Insights . . . 170
6.2 Limitations . . . 172
6.3 Future Work . . . 173
6.3.1 Algorithmic Developments . . . 173
6.3.2 Applications in Machine Learning and Data Science . . . 174

APPENDICES . . . 176

BIBLIOGRAPHY . . . 225

LIST OF FIGURES

FIGURE

1.1 Overview of the goal of the thesis. We aim to propose novel incremental algorithms for low-rank tensor decompositions as a solution to the challenges of processing large-scale and streaming data. In addition to resulting in compression, the proposed methods also provide a mapping between the original high-dimensional data and a low-dimensional latent space (shown as the orange parts of the tensor networks), and a latent representation of the data (shown as the green parts of the tensor networks). The resulting low-dimensional representations can then be used for various downstream applications, such as supervised learning and Bayesian inference. 8

2.1 Tensor network diagram representations for arrays of various dimensions 24
2.2 Tensor network diagrams depicting various tensor contractions. The edges connecting the nodes represent the dimensions that are being contracted. The resulting tensor is represented by a node with edges corresponding to its new dimensions. 28
2.3 Visualizations for the **C**ANDECOMP/**P**ARAFAC (CP) format 30
2.4 Visualizations for the Tucker format 31
2.5 Visualizations for the **T**ensor **T**rain (TT) format 33
2.6 Visualizations for the **H**ierarchical **T**ucker (HT) format 35
2.7 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 B.1.2. Root, transfer, and leaf nodes are colored in red, blue, and green, respectively. 36
2.8 Visualizations for the batch-HT format 37

3.1 Flow of the proposed **T**ensor **T**rain **I**ncremental **C**ore **E**xpansion (TT-ICE) algorithm. The algorithm updates the representation of an accumulated tensor in tensor train format as new batches of tensors become available. Green represents the input data and red represents the output data. 47
3.2 Example visualizations from the datasets used in the compression experiments. (a) A frame from a Ms.Pac-Man gameplay session. (b) Snapshots from self-oscillating gel simulation using the simulation developed in [9]. 61

3.3 Comparison of execution time vs total number of observations in the tensor train for different versions of TT-ICE, TT-ICE* and ITTD[161] with two different ε settings. TT-ICE*: full TT-ICE* algorithm with all heuristics, TT-ICE: TT-ICE, Subselect: subselecting frames from runs, SS: subselecting frames from runs and skip updating the first d TT-cores, ITTD 5: ITTD with rounding at every fifth increment step. ITTD has worst compression time while TT-ICE* with all heuristics performs best. ITTD Fails to compress the entire stream for $\varepsilon_{des} = 0.01$. 63
3.4 Comparison of mean relative reconstruction error (RRE) and mean relative test error (RTE) vs total number of observations in the tensor train for different versions of TT-ICE, TT-ICE* and ITTD[161] with two different ε settings. We present the mean RRE over compressed runs in the accumulation and mean RTE over the validation set after each increment step. Data points connected with a solid line represent the mean RRE and data points connected with a dashed line represent the mean RTE. Please refer to Figure 3.3 for a description of the legend. For $\varepsilon_{des} = 0.1$ TT-cores returned from ITTD have much lower RTE than desired. RTE of TT-cores trained with TT-ICE and TT-ICE* converge to RRE as accumulation size increases. ITTD Fails to compress the entire stream for $\varepsilon_{des} = 0.01$. 64
3.5 Comparison of compression ratio vs total number of observations in the tensor train for different versions of TT-ICE, TT-ICE*, and ITTD[161] with two different ε settings. Please refer to Figure 3.3 for a description of the legend. ITTD has the worst compression ratio while TT-ICE* with all heuristics performs the best. ITTD Fails to compress the entire stream for $\varepsilon_{des} = 0.01$. 65
3.6 Comparison of execution time vs total number of simulations in the tensor train for Algorithm 1, Algorithm 2 and ITTD[161] with two different ε settings. TT-ICE* outperforms TT-ICE by one order of magnitude. ITTD Fails to compress the entire stream for both cases. TT-ICE*: full TT-ICE* with all heuristics defined in Section 3.1.2.3, TT-ICE: TT-ICE algorithm (Algorithm 1), ITTD5: ITTD with rounding at every fifth increment step, ITTD2: ITTD with rounding at every second increment step. . . . 67
3.7 Comparison of compression ratio vs total number of simulations in the tensor train for Algorithm 1, Algorithm 2 and ITTD[161] with two different ε settings. Please refer to Figure 3.6 for a description of the legend. TT-ICE* outperforms TT-ICE* but both methods show comparable compression performances. ITTD Fails to compress the entire stream for both cases. 67
3.8 Comparison of mean RRE vs total number of simulations in the tensor train for TT-ICE, TT-ICE* and ITTD[161] with two different ε settings. Please refer to Figure 3.6 for a description of the legend. TT-ICE* provides mean RRE closer to ε_{des} in both cases. ITTD Fails to compress the entire stream for both cases. 68
3.9 Step-by-step decomposition of an N-batch of 5-D tensors with the **B**atch **H**ierarchical **T**ucker **l**eaves-**to-r**oot (BHT-l2r) algorithm (Algorithm 3). The decomposition starts with the leaves of the last layer. Since the dimension tree is constructed beforehand, Algorithm 3 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. 74

3.10 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 data set with $\varepsilon_{rel} = 0.10$ using various normalization methods. For the same normalization methods, the batch hierarchical Tucker format achieves on average 1.5× compression over the hierarchical Tucker format, while running on average 1.7× 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. 77
3.11 Comparison of HT format and BHT format in terms of compression ratio (CR - left) and compression time (right) on the PDEBench 3D Navier-Stokes data set 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 3.10 for the normalization methods. 78
3.12 Comparison of hierarchical Tucker format and batch hierarchical Tucker format in terms of compression ratio (CR - left) and compression time (right) on the BigEarthNet data set with $\varepsilon_{rel} = 0.05 - 0.30$. The BHT format achieves 6.2× compression over the HT format at $\varepsilon_{rel} = 0.3$ while running 3.75× faster. At $\varepsilon_{rel} = 0.1$, the BHT format achieves 0.95× compression over the HT format while running 2.69× 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. 79
3.13 Flow of the proposed **H**ierarchical **T**ucker **R**apid **I**ncremental **S**ubspace **E**xpansion (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. 80
3.14 Example snapshots from scientific data sets. 86
3.15 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.5 for experiments with $\varepsilon_{rel} = 0.10$ and Figure D.7 for experiments with $\varepsilon_{rel} = 0.01$. 88
3.16 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.6 for experiments with $\varepsilon_{rel} = 0.10$ and Figure D.8 for experiments with $\varepsilon_{rel} = 0.01$. 88

3.17 Compression ratio (CR - left) and reduction ratio (RR- - right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes data set. 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. 91
3.18 Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes data set. 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. 91
3.19 Example frames from image-based data sets. 92
3.20 Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the Basalt MineRL data set.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. 94
3.21 Compression time (left) and Relative Test Error (right) of the algorithms on the Basalt MineRL competition data set. 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. 94
3.22 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. 97
3.23 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. 97

4.1 *Proposed compressed analysis pipeline. High-fidelity simulation environments generate large-scale simulation data. This data can be compressed incrementally into a tensor-network format and then analyzed without decompression.* 104

4.2 Images of the SPT-100 Hall thruster. Figure 4.2a shows the thruster in display. Note that this figure is cropped to show the SPT-100 thruster only. Figure 4.2b shows an operating electric Hall thruster identical to those that will be used to propel the Psyche spacecraft. 106
4.3 Visualisation of the simulation domain. The domain is [-2,10] meters for the x axis, and [0,10] meters for y and z axes. The thruster centerline is aligned with the x axis and the exit plane of the thruster is located at $x = 0\,\text{m}$. Note that the thruster is not shown to scale for visualisation purposes. 107
4.4 Visualisation of the plume in the simulation domain where $R = \sqrt{y^2 + z^2}$. θ is a coordinate to define distance from the thruster injection surface and is defined as $\theta = tan^{-1}(z/x) \approx tan^{-1}(y/x)$. The plume is approximately symmetric about the thruster centerline. 113
4.5 (Left) Neutral atom number density contours from the original dataset. (Right) Neutral atom number density from the reconstructed dataset using a square root normalization, and a target relative error $\varepsilon = 10^{-2}$. Neutral and ion number density fields are compressed together. The reconstructed dataset has a compression ratio of 8.08×. These comparisons show that the reconstructed dataset accurately captures near-field and far-field neutral number densities as well as the backflow phenomena behind the thruster caused by CEX collisions. 113
4.6 (Left) Ion number density contours from the original dataset. (Right) Ion number density contours from the reconstructed dataset using a square root normalization, and a target relative error $\varepsilon = 10^{-2}$. Neutral and ion number density fields are compressed together. As in the neutral number density contours, the reconstructed dataset captures the ion number density contours in the near-field, far-field, and CEX plasma with sufficient resolution. 114
4.7 (Left) Angular profile of ion current density of the plume at 100 cm ±12.5 cm from the thruster injection plate. (Right) Relative error between the ion current density profiles between the original and reconstructed datasets with various normalization combinations and a fixed target relative error of $\varepsilon = 10^{-4}$. At higher angles, i.e. further away from the thruster injection surface, the reconstructed dataset deviates from the original with a max relative error of 0.615. A logarithmic-logarithmic normalization with target relative error $\varepsilon = 10^{-4}$ results in a compression ratio of 6.23× with relatively low relative error even at the higher plume angles. log: logarithmic normalization, zscore: z-score normalization, unitvector: unit vector normalization, none: no normalization. 115
4.8 (Left) Electric field in the near-field plume region from the original dataset. (Middle) Electric field from the reconstructed dataset using a z-score normalization, and $\varepsilon = 10^{-1}$ target relative error. The components of the electric field are compressed together. This yields a very high $CR = 354,796\times$ as well as high accuracy to the original electric field contours. (Right) shows the absolute error in electric field components. The largest error is 0.356 V in the E_x component. 116

4.9 Derived surface QoI from the original and reconstructed unstructured mesh data. Compression with no normalization and a target relative error of $\varepsilon = 10^{-6}$ results in a reconstructed dataset with CR = 5.077×. (Left) Electric potential quantities from compressed and reconstructed datasets are nearly one-to-one. (Right) The total current quantity from reconstruction follows the trend of the original quantity with a maximum relative error of 1.56 seen around iteration 13500. 118
4.10 Percentage relative error of the estimation of the full system state using the Bayesian inference framework with different tensor networks. Figures (a) and (b) show the median, and Figures (c) and (d) show the 90-th percentile of the relative error. The reported results in this figure are averaged over 40 random sensor placements for each tensor network and number of sensors. 123

5.1 High-fidelity Unreal Engine 5 environments used for creating the high-resolution image streams. The environments are designed to simulate the TAS 500 real-world off-road driving environment, and they include a variety of terrains and weather conditions. 127
5.2 Example frames from the high-resolution camera streams recorded in the Unreal Engine 5 environments. The frames are captured at a resolution of 960×540 pixels and at a rate of 60 frames per second. The streams are recorded using a front-facing camera with 90° field-of-view mounted on the hood of a vehicle that drives through the environments. 128
5.3 Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments. The first column shows the downsampled frames, and the subsequent columns show the reconstructions from the compressed frames using different relative error thresholds. First row shows the reconstructions from the forest environment, and the second row shows the reconstructions from the winter environment. The frames are downsampled to a resolution of 384 × 224 pixels, and the compression is performed using the HT-RISE algorithm. The relative error thresholds used for compression are shown in the captions of the respective columns. Note that the first column shows the downsampled frames, which are used as input to the tensor networks. The compression ratios achieved are shown in Table 5.1. 130
5.4 Cross-environment generalization performance of the mappings learned from the HT-RISE algorithm. The first column shows the original downsampled frames. **Top row:** Reconstructions of forest frames using a model trained on the winter environment. **Bottom row:** Reconstructions of winter frames using a model trained on the forest environment. The subsequent columns show reconstructions at different relative error thresholds (ε). 131
5.5 Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments using naive quantization of the latent representations. First image shows the original downsampled frame, second image shows the reconstruction from the original FP64 latent representations, and the third image shows the reconstruction from the quantized latent representations. Following this naive approach to quantization results in significant visual artifacts in the reconstructed images. . . . 134

5.6 Schematic of the outlier-aware quantization approach for the latent representations. The latent representations are split into two parts: the outliers and the rest of the values. The outliers are defined as the values that are above a certain threshold, which is usually set as the values outside the range $[-1, 1]$. The rest of the values are then quantized to UINT8, while the outliers are stored as FP64 values as sparse arrays. This allows us to achieve a much higher compression ratio while maintaining a high reconstruction quality. 135
5.7 Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments using outlier-aware quantization of the latent representations. First image shows the original downsampled frame, second image shows the reconstruction from the original FP64 latent representations, third image shows the reconstruction from the naive quantization of the latent representations, and the fourth image shows the reconstruction from the outlier-aware quantization of the latent representations. The outlier-aware quantization approach achieves much closer reconstruction quality to the original FP64 latent representations. 136
5.8 Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments using just the outliers of the latent representations. First image shows the original downsampled frame, second image shows the reconstruction from the original FP64 latent representations, third image shows the reconstruction from the outlier-aware quantization of the latent representations, and fourth image shows the reconstruction from just the outliers of the latent representations. For the "Just outliers" approach, we only stored the outliers beyond $[-0.3, 0.3]$, which results in a sparse array of 30 kB per frame. 137
5.9 Imitation learning framework. Our focus is to enhance latent space learning depicted by the blue box. We compare the TT, autoencoder (AE) and variational autoencoder (VAE) approaches. Orange cubes represent images from gameplay videos. The extracted latent features are then used in the action-prediction network to predict actions. The architecture of the action predictor remains the same for all three encoding methods. 141
5.10 Example reconstruction of a Ms. Pac-Man frame using TT (with $\varepsilon = 0.01$), AE and VAE. TT reconstruction is visually the most similar to the original frame. The size of the latent space for all three models is 11583, which is 11.5% of the original size ($210 \times 160 \times 3$). 145

5.11 Speedup of the end-to-end training procedure of the AE, VAE, and TT-based predictors compared to a predictor trained using the full RGB frames. Extracting a latent space reduces end-to-end training time in all cases. In the low data regime, using the TT-GPU leads to a 26.1× to 29.7× faster training time than using the full RGB frames and a 8.2× to 9.6× speedup compared to the AE and VAE. In the moderate data case (160 trajectories), using TT-GPU leads to a 3.4× to 9.8× faster training time compared to using the full RGB frames and a 2.6× to 4.8× speedup compared to the AE and VAE. Note that the difference in speedups between the 5 trajectory case and 160 trajectory case is due to the difference in training epochs used in each case (50 in the 5 trajectory case and 5 in the 160 trajectory case); the speedups are similar when the number of training epochs is held fixed while the number of training trajectories is changed, as shown in Section 5.2.6.3. BR:Beam Rider, B:Breakout, MP:Ms.Pac-Man, P:Pong, Q:Q*bert, S:Seaquest, SI:Space Invaders. 150
5.12 Action prediction accuracies of trained predictors using encodings from the TT, AE, and VAE, as well as a trained predictor using the full-sized original RGB frames. Training a predictor using TT encodings yields similar and sometimes better prediction accuracies as training a predictor using AE or VAE encodings. Error bounds are calculated as two times the standard error of gameplay scores calculated across ten (in the 5 trajectory case) or three (in the 160 trajectory case) experiments. Please refer to the caption of Figure 5.11 for the game abbreviations. 150
5.13 Normalized gameplay scores achieved by each of the behavioral cloning models, trained using either encodings from the TT, AE, VAE, or the original RGB images. A score of 0 corresponds to performance of an agent taking random actions and a score of 1 corresponds to performance of the RL agent. Training a predictor using TT encodings yields similar gameplay scores as training a predictor using AE or VAE encodings. Error bounds are calculated as two times the standard error of gameplay scores calculated across ten (in the 5 trajectory case) or three (in the 160 trajectory case) experiments. 151
5.14 Results of the experiments investigating the effect of number of trajectories used to train the behavioral cloning framework on computational speedup and action prediction accuracy using different encoding methods for the game of Seaquest. As the number of training trajectories increases, action prediction accuracy also increases. Computational benefits of TT-ICE* over the AE, VAE, or no encoding remain similar across different number of trajectories. Action prediction accuracies are only presented for TT-CPU since TT-CPU and TT-GPU result in the same latent space. 153
5.15 Comparison of behavioral cloning accuracy versus end-to-end training time for the AE, VAE, and TT, when keeping the number of data points seen by each algorithm the same by using only a single pass through the data. Points in the upper left indicate lower training time and higher accuracy. Note that TT-GPU tends to perform better in terms of lower training time and higher accuracy than the AE and VAE. 154

5.16 Maximum tensor rank (red) and flop count (teal) for the TT during training. As training progresses, the maximum tensor rank and the flop count increase, but this growth rate decreases. The slow growth in the TT ranks suggests that the TT-ICE* algorithm is able to learn a generalizable latent structure; as the amount of data seen by the TT-ICE* algorithm increases, smaller updates to the TT ranks are needed to represent new data. In cases like this where the tensor ranks of the TT grow slowly, the encoding time for the TT will scale well since the flop count grows quadratically with the TT ranks r. 156
5.17 Generative modeling pipeline using tensor network embeddings. The pipeline consists of a tensor network, a denoising diffusion model with a UNet architecture, and action inputs. The tensor network compresses the high-dimensional game frames into a low-dimensional latent space, which is then used as input to the denoising diffusion model along with the user actions. We use the same tensor network to decompress the generated latent space back into high-dimensional game frames for visual inspection. 158
5.18 Denoising diffusion process. $q(x_t|x_{t-1})$ represents the forward process, which adds noise to the data at each time step t. $p_\theta(x_{t-1}|x_t)$ represents the reverse process, where θ are the parameters of the denoising model. In line with existing literature, we use a UNet-based neural network as p_θ. Figure adapted from [106]. 159
5.19 Reconstructions of downsized ATARI game frames using HT-RISE at different relative error levels. The first row shows the original downsized game frames, while the second row shows the reconstructed frames using HT-RISE at a relative error level denoted by the corresponding captions. Please refer to Table 5.10 for the compression rates achieved by HT-RISE for the displayed relative error levels. Displayed games (from left to right): Pong, Breakout, Ms.Pac-Man, Q*bert. 163
5.20 Visuals from the generative modeling of ATARI games using tensor-network embeddings. The first row shows the ground truth frames from the ATARI game. The second row shows the corresponding latent space representations obtained through HT-RISE, which captures the essential features of the game. The third row shows the denoised RGB frames generated by the diffusion model, while the fourth row shows the denoised latent space representations. RGB diffusion suffers from stability issues, while latent diffusion yields stable results. Columns correspond to different time steps in the diffusion process, with $t = 0$ being the initial state and $t = 160$ being the final state. 165
5.21 Dynamics of the latent space representations during the diffusion process. The figure shows the 5 largest (top) and 5 smallest (bottom) values from the latent space representations for each frame. Red lines indicate the original latent space values obtained by projecting the ground truth frames onto the latent space, while blue lines indicate the denoised latent space values obtained by performing the diffusion process using ground truth frames as input. Finally, the yellow lines indicate the denoised latent space values obtained by performing the diffusion process using the generated frames as input. Both methods are able to track the overall dynamics of the latent space. . 166

5.22 Bleeding within the frames during autoregressive frame generation process. The figure shows the denoised frames generated by the diffusion models at different time steps. The top row shows frames from the diffusion model trained on the full RGB dataset (600 episodes), while the bottom row shows frames from the diffusion model trained on the full latent dataset (600 episodes). The frames are generated using a context window of 4 previous frames and actions. The image generation process starts well for the RGB model but then begins to produce artifacts, such as bleeding of the playing field to the score field. The issue gets worse as the game time progresses, leading to the total loss of the score field and the frames suddenly diverge to pure static noise. This behavior is not observed in the latent diffusion model, which is able to track the dynamics of the game until the end of the episode. The columns correspond to different time steps in the autoregressive generation process denoted in their respective captions. 167

A.1 Original frames from the Atari dataset (top) along with their respective reconstructions from the TT-cores compressed using TT-ICE (middle) and ITTD5 (bottom) with $\varepsilon_{des} = 0.1$. There are visible visual artifacts on some of the frames. There is no visible difference between TT-ICE and ITTD5 reconstructions. Left-to-right: Breakout, Beam Rider, Enduro, Ms.Pac-Man, Pong, Q*bert, Seaquest, Space Invaders 179

A.2 Original frames from the Atari dataset (top) along with their respective reconstructions (bottom) from the TT-cores compressed using TT-ICE with $\varepsilon_{des} = 0.01$. There are no visible artifacts on the frames. ITTD5 reconstructions are not included here since ITTD5 fails at all games for $\varepsilon_{des} = 0.01$. Left-to-right: Breakout, Beam Rider, Enduro, Ms.Pac-Man, Pong, Q*bert, Seaquest, Space Invaders 179

A.3 Comparison of relative errors of TT-ICE* and TT-FOA[1] averaged over 20 repetitions. TT-ICE*: TT-ICE* with $\varepsilon_{des} = 10^{-8}$, TT-FOA: TT-FOA initialized with correct TT-ranks, TT-FOA (IR): TT-FOA initialized with insufficient TT-ranks, TT-FOA (OR): TT-FOA initialized with overestimated TT-ranks. Unlike TT-FOA, TT-ICE* achieves a low relative error without the need for a convergence period and discovers the correct underlying TT-ranks. In case of underestimated TT-ranks, TT-FOA fails to achieve low relative error. 180

B.1 Comparison of various tensor reshapings in terms of compression ratio (CR - left) and reduction ratio (RR - right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using no normalization. The reshaping $4\times4\times4\times4\times4\times4\times4\times4\times4\times4\times5\times21\times1$ results $1.8\times$ the CR of the worst performing reshaping, $64\times64\times64\times5\times21\times1$. Similarly, the reshaping $4\times4\times4\times4\times4\times4\times4\times4\times4\times4\times5\times21\times1$ results in $1.2\times$ of the RR of the worst performing reshaping, $2\times2\times4\times2\times2\times4\times2\times2\times4\times2\times2\times4\times2\times2\times5\times21\times1$. 184

B.2 Comparison of various tensor reshapings in terms of compression time (left) and relative test error (RTE - right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using no normalization. The reshaping $2\times2\times4\times2\times2\times4\times2\times2\times4\times2\times2\times4\times2\times2\times5\times21\times1$ takes $9.8\times$ the time it takes the reshaping $8\times8\times8\times8\times8\times8\times5\times21\times1$ to compress the data set. All reshapings except for $64\times64\times64\times5\times21\times1$ result in almost the same RTE. 184

B.3 Comparison of various axis orderings in terms of compression ratio (CR - left) and reduction ratio (RR - right) on the PDEBench 3D Navier-Stokes data set 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. . . . 186
B.4 Comparison of various axis orderings in terms of compression time (left) and relative test error (RTE - right) on the PDEBench 3D Navier-Stokes data set 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× the compression time of best performing axis ordering. All axis reorderings result in almost the same RTE. 186

D.1 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. 197
D.2 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.1 for the normalization methods. The results are averaged over 5 seeds. . 198
D.3 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.1 for the normalization methods. The results are averaged over 5 seeds. 199
D.4 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.1 for the normalization methods. The results are averaged over 5 seeds. . 199

D.5 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. 200
D.6 Compression time (left) and Relative Test Error (right) of the algorithms on the self-oscillating gel data set 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.5 for normalization methods. The results are averaged over 5 seeds. 201
D.7 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. 201
D.8 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.7 for the normalization methods. . . . 202
D.9 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. 203
D.10 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. 204
D.11 Reconstructed video frames from Basalt MineRL competition data set 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 3.4, and Figures 3.20 and 3.21 for quantitative results. 205

E.1 Time for end-to-end training procedure in the limited data regime, averaged across all experiments, as a fraction of the time required to train the full-frame predictor. Extracting a latent space before training a predictor reduces end-to-end training time in all cases, but using the TT-GPU leads to a 26.1× to 29.7× reduction in training time. Compared to the AE / VAE, using the TT-GPU leads to a 8.2× to 9.6× speedup. 208
E.2 Time for the end-to-end training procedure in the moderate data regime, averaged across all experiments, as a fraction of the time it takes to train a predictor using the full RGB frames. Extracting a latent space reduces end-to-end training time in all cases, but using TT-GPU leads to a 3.4× to 9.8× reduction in training time compared to no encoding and a 2.6× to 4.8× speedup relative to the AE / VAE. 208
E.3 Action prediction accuracy of behavioral cloning frameworks using different encoding methods at varying ε levels. As the latent space size is increased for the encoding-based predictors, the action prediction accuracy of the predictors tends to increase. The speedup offered by TT does not cause any performance deterioration. Furthermore, the nonlinearity offered by AE/VAE does not result in an improved latent space representation that achieves higher action prediction accuracy with a smaller latent space. Since Pong with $\varepsilon = 0.15$ and $\varepsilon = 0.10$ returns the same latent size, the accuracy for $\varepsilon = 0.15$ is omitted. Please refer to Table E.2 for latent space sizes corresponding to each ε level for each game. 210
E.4 Total time elapsed to train behavioral cloning frameworks using different encoding methods at varying ε levels. As the latent space size increases, the total training time for the various encoding-based predictors increases. Across all games and ε levels, TT-GPU encoding offers the fastest end-to-end training time. Please refer to Table E.2 for latent space sizes corresponding to each ε level for each different game. 210

F.1 (Right) Neutral number density contours are shown for reconstructions using a square root normalization, $\varepsilon = 10^{-1}$ target relative error, and where particle number density fields are compressed together. Increasing the input target relative error in results in extreme compression with a CR $53,384\times$, however, decreases contour resolution and fails to capture the correct backflow phenomena when compared to the reconstruction seen in Figure 4.5. Here, the TT approximation overestimates the number density in the backflow region. 213
F.2 (Right) Neutral number density contours are shown for reconstructions using a logarithmic normalization with base 10, $\varepsilon = 10^{-1}$ target relative error, and where particle number density fields are compressed together. Although the CR is 16× here, the reconstructed dataset has decreased resolution and more statistical noise in the plume contours for X < 4 m when compared to Figure 4.5. This is likely due to the normalization method used; the base-10 logarithm shrinks the range of values aggressively and makes the reconstruction sensitive to numerical errors. 214
F.3 (Right) Neutral number density contours are shown for reconstructions using a logarithmic-logarithmic normalization, $\varepsilon = 10^{-2}$ relative error, and where neutral and ion number density fields are compressed together. Cascading the same normalization method two times resulted in accurate contour modeling for X < 4 m but decreased resolution in the 10^{14} contour. CR = 1.38×. 215

F.4 (Right) Neutral number density contours are shown for reconstructions using a unit vector normalization, $\varepsilon = 10^{-2}$ target relative error, and where particle number density fields are compressed together. Although we have a CR of 5287×, we see an inaccurate approximation of the backflow region. 215
F.5 (Right) Neutral number density contours are shown for reconstructions using a z-score normalization, $\varepsilon = 10^{-2}$ target relative error, and where neutral and ion number density fields are compressed together. This results in an approximation with a CR of 5213×. 216
F.6 The reconstructed dataset used the same compression parameters as in Figure F.5, however, here, the neutral atoms are compressed separately from the other fields. Compressing separately increased the CR to 24,563×. 217
F.7 Neutral number density contours are shown for reconstructions using a logarithmic normalization, $\varepsilon = 10^{-2}$ target relative error, and where neutral and ion number density fields are compressed together. The CR is 1.37×. 217
F.8 (Right) The reconstructed dataset has a CR of 1.91× and the same compression input parameters as in Figure F.7, however, the neutral atom field is compressed separately from the other species. We see similar resolution of the reconstruction contours as in Figure F.7 and a marginally increased CR. 218
F.9 (Right) The reconstructed ion number density contours here correspond to when the neutral atoms and three types of ions are all compressed separately. Please refer to Table ßF.3 for the details of compression parameters. The reconstructed ion number density contours are very close to the original contours while having an average CR of 2.88×. 219
F.10 (Right) Neutral number density contours obtained through compression with binary ordering of subdomains, z-score normalization, and a target relative error of $\varepsilon = 10^{-2}$. This combination achieves a high CR of 5213.6×. 221
F.11 (Right) Ion number density contours obtained through compression with a spatially-aware domain input, z-score normalization, and a target relative error of $\varepsilon = 10^{-2}$. The CR is increases to 5287.6×. 222
F.12 Neutral number density contours square root normalization and $\varepsilon = 10^{-2}$ using a binary ordering of subdomains in compression, yielding a CR of 8.07× 222
F.13 Neutral number density contours square root normalization and $\varepsilon = 10^{-2}$ using the spatially-aware domain input to compression. This input schema still yields accurate reconstructions to the original with reasonable compression ratio of 7.23×. 223
F.14 Ion number density contours square root normalization and $\varepsilon = 10^{-2}$ using the binary ordering of subdomains into compression, yielding a CR of 8.07×. 223
F.15 Ion number density contours square root normalization and $\varepsilon = 10^{-2}$ using the spatially-aware domain input to compression. This yields accurate reconstructions while having a nontrivial CR of 7.23×. 224

LIST OF TABLES

TABLE

1.1 Comparison of dimensionality reduction methods across key properties for large-scale applications. The proposed methods in this thesis are denoted with the relevant section numbers. Note that the TT-ICE algorithm is not the only incremental algorithm for the TT format, but it is the only one that provides theoretical guarantees on the approximation error and scales to large-scale data. Symbols: ✓ = Well-supported, △ = Possible with modifications, ∘ = Partially supported/Limited, × = Not supported/Poor performance. Please see the text for more details on the interpretation of each symbol for the context of each property. 18

2.1 Comparison of storage complexity of tensor decomposition formats for a d-dimensional tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$. The storage complexity is expressed in terms of the number of dimensions d, the size of each dimension n, and the rank r of the respective decomposition. Please note that each format requires a different r to approximate the same tensor at the same accuracy. 39

2.2 Detailed comparison of tensor decomposition formats highlighting their respective strengths and weaknesses 40

3.1 Summary of the data sets 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 data set on disk. Sims, vids, and ims refer to simulations, videos, and images, respectively. . . . 84

3.2 Summary of the compression experiments with the self-oscillating gel data set. 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. 87

3.3 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.1 and D.2 for experiments with $\varepsilon_{rel} = 0.10$ and Figures D.3 and D.4 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. 90
3.4 Summary of the compression experiments with the MineRL Basalt competition data set. 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. . . 93
3.5 Summary of the compression experiments with the BigEarthNet multispectral satellite imagery data set. 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. 96

4.1 Compressed data metrics for neutral and ion number density states for a time averaged snapshot with 3.6 million particles. With a square root normalization combination and a target relative error of $\varepsilon = 10^{-2}$, the TT-decomposition results in states with a reduced size on disk by 8.076×. 114
4.2 Compressed data metrics for the ion current density state for multiple normalization combinations. Compression of the ion current density is very sensitive to the normalization method selected. log: logarithmic normalization, zscore: z-score normalization, unitvector: unit vector normalization, none: no normalization. 115
4.3 Compressed data metrics for electric fields. The components of the electric field are compressed together using a z-score normalization. The electric field data is highly compressible and achieves extreme compression even at relatively high target error levels. The electric fields are not time-varying, and the ions are only accelerated electrostatically, leading to a latent space size of 1 and high compression. 116
4.4 Compressed data metrics for surface quantity states. The components of the surface mesh data are compressed together without normalization and using a target error of $\varepsilon = 10^{-6}$. Even though the transient part of the simulation is included for this case study, the surface fields turn out to be compressible at a very high speed. 118

5.1 Compression results for the high-resolution camera streams recorded in the Unreal Engine 5 environments. The compression is performed using the HT-RISE algorithm. The compression ratios in this table are elementwise compression ratios, which are defined as the ratio of the number of elements in the original tensor to the number of elements in the compressed tensor. Rel. Error is the relative error threshold set for the HT-RISE algorithm. Time is the time taken to compress all the frames in the respective training datasets. Latent Shape is the shape of the latent representation of a single frame obtained after compression. Note that the compression reported here is with respect to the downsampled frames, which have a resolution of 384×224 pixels. 129
5.2 Quantitative evaluation of the reconstruction quality for the high-resolution camera streams recorded in the Unreal Engine 5 environments. The evaluation is performed using the mean squared error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index Measure (SSIM). The results are shown for different relative error thresholds used in the compression. Note that a higher PSNR and SSIM indicate a better reconstruction quality, while a lower MSE indicates a better reconstruction quality. 133
5.3 Quantitative evaluation of the reconstruction quality for the high-resolution camera streams recorded in the Unreal Engine 5 winter environment using different quantization approaches at $\varepsilon = 0.15$. The evaluation is performed using the mean squared error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index Measure (SSIM). The results are shown for different relative error thresholds used in the compression. Note that a higher PSNR and SSIM indicate a better reconstruction quality, while a lower MSE indicates a better reconstruction quality. 137
5.4 Latent sizes L for the AE, VAE, and TT for the games tested. Note that the latent size is determined automatically for the TT given the prescribed relative error bound of 0.01. That latent size is then used for the AE and VAE. 148
5.5 Summary of the number of trajectories used to train the models in the limited and moderate data cases. Note that each row corresponds to separate trajectories. . . . 148
5.6 Reconstruction quality of the TT, AE, and VAE in the low data regime. The reported statistics are MSE (using 255 as maximum pixel value) and PSNR of the reconstructions versus the original frames. For each game, the bolded values are the smallest MSE and highest PSNR. 151
5.7 Reconstruction quality of the TT, AE, and VAE in the moderate data regime. The reported statistics are the MSE (using 255 as maximum pixel value) and PSNR of the reconstructions versus the original frames. For each game, the bolded values are the smallest MSE and highest PSNR. 152
5.8 Median time to encode the training and validation data in seconds using the various encodings. In the 5 trajectory case, the encoding time for TT-GPU is between $0.4\times$ and $2.8\times$ that of the AE and VAE. In the 160 trajectory case, the encoding time for TT-GPU is between $1.5\times$ and $8.8\times$ that of the AE and VAE. 156
5.9 Ratio of encoding times and flop counts between the TT and AE/VAE. Although the flop counts are much higher for the TT than the AE/VAE, this does not lead to a proportionate increase in encoding time. 157

5.10 Compression rates achieved by HT-RISE for different ATARI games. The compression rates are reported in this table are elementwise compression rates, i.e., the ratio of the number of elements in the original tensor to the number of elements in the compressed tensor. The $\otimes$ symbol indicates that the relative error level is too low for a meaningful compression, while the $*$ symbol indicates that the relative error level is too high for maintaining the visual quality of the reconstructed frames. Please refer to Figure 5.19 for visualizations of the original and reconstructed frames. 162

A.1 Summary statistics for the other ATARI games. Game: name of the game, Machine: code-name of the computer that conducted the experiments, Method: decomposition algorithm used (TT-ICE*: Algorithm 2, TT-ICE: Algorithm 1, ITTD: [161, Alg 3]), ε: the desired relative error bound for the given decomposition method, Frames: total number of the frames compressed in the TT-cores, Used: number of frames used to update the TT-cores, Time: total execution time of the algorithm, mean RRE: final mean RRE at the end of the experiment, CR: compression ratio. For each experiment, the best performance is shown in bold for Time, mean RRE, and CR. All failures are due to insufficient memory. 178

B.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 data set with $\varepsilon_{rel} = 0.10$. Time: Compression time in seconds, CR: Compression ratio, RR: Reduction ratio, RTE: Relative test error. ‡ denotes that the algorithm failed to compress the data set due to maximum walltime limit (389/480 simulations). . . . 183

B.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 data set 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$\times$ higher in compression time, CR and RR, while yielding almost the same RTE. Time: Compression time in seconds, CR: Compression ratio, RR: Reduction ratio, RTE: Relative test error. 185

D.1 Names and wavelengths of spectral bands of Sentinel2 images. SWIR: Short Wave Infrared, NIR: Near Infrared. The bands are ordered in increasing wavelength. . . . 196

E.1 Architecture for the encoding part of **A**uto**e**ncoder (AE)s and **V**ariational **A**uto**e**ncoder (VAE)s. After each intermediate layer we also apply batch norm followed by LeakyReLU - these are excluded from the table for simplicity. The asymmetric padding in the first layer is to account for the shape of the input. 206

E.2 Latent size as epsilon for the TT is changed. The reported numbers are an average across three seeds (which use different training data). 209

E.3 Comparison of the gameplay scores achieved by the RL agent compared to an agent that acts randomly and a human player. The human gameplay scores are taken from [182]. 211

E.4 Comparison of the gameplay lengths (number of time steps from the beginning of the game to the end) achieved by the RL agent compared to an agent that moves randomly. Note that gameplay lengths for a human player were not available from [182]. . . . 211

F.1 Compression metrics for number density states using various normalization methods and target relative errors ε. 213

F.2 Quantitative compression performance comparison between compressing number density states together vs compressing neutral number density separately. Compressing number density separately increases the CR. However, the amount of increase depends heavily on the normalization method preferred. Note that we approximate the size corresponding to the neutral atom density by dividing the total size of the compression by 4. All other metrics reported for number densities compressed together are for the entire compression. 218

F.3 Compressed data metrics for ion number density states compressed separately. By compressing states separately, we can achieve extreme compression up to $553,846\times$. Note that the total CR is severely impacted by the compression of slow-moving CEX ions and can be greatly improved by finding a more performant compression setting. 219

F.4 Domain input structures for compression. The eight subdomains can be fed into the algorithm in a binary order which does not preserve the spatial relation between the subdomains in physical space, i.e (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}). We investigate inputting a single spatially-aware domain where the subdomain respective positions in physical space are considered. 220

F.5 Compression metrics for number density states using binary ordering and spatially-aware inputs. Using spatially-aware subdomain inputs results in accurate reconstructions, but not significantly increased resolution over binary ordered inputs. Further, the compression ratios between the two input schemas are comparable. 224

LIST OF ALGORITHMS

ALGORITHM
1 `TT-ICE`: Incremental update of a tensor train decomposition 50
2 `TT-ICE`*: Incremental update of a tensor train decomposition with heuristic performance upgrades 100
3 `BHT-l2r`: Error truncated leaves-to-root batch hierarchical Tucker decomposition 101
4 `HT-RISE`: Incremental updates to a tensor represented as batch hierarchical Tucker 102
5 Mapping unseen data to latent space by orthogonal projections onto TT-cores 144
6 `updateIndexSet`: Updates the index set of a given object 193
7 `expandCore`: Expands the basis of a Tucker core 193
8 `padWithZeros`: Pad a hierarchical Tucker core with zeros 194

LIST OF APPENDICES

A Aditional Comparisons for TT-ICE Algorithm 176

B Additional Performance Analysis of the HT-RISE Algorithm 182

C Proofs for HT-RISE and BHT-l2r Algorithms 187

D Auxiliary Subroutines and Additional Information on HT-RISE Analyses 192

E Behavioral Cloning Framework 206

F Additional Results on Particle in Cell Simulation Compression 212

LIST OF ACRONYMS

TT **T**ensor **T**rain

MPS **M**atrix **P**roduct **S**tate

TT-SVD **T**ensor **T**rain **S**ingular **V**alue **D**ecomposition

HOSVD **H**igher-**O**rder **S**ingular **V**alue **D**ecomposition

CP **C**ANDECOMP/**P**ARAFAC

TT-ICE **T**ensor **T**rain **I**ncremental **C**ore **E**xpansion

HT **H**ierarchical **T**ucker

HT-RISE **H**ierarchical **T**ucker **R**apid **I**ncremental **S**ubspace **E**xpansion

BHT-l2r **B**atch **H**ierarchical **T**ucker **l**eaves-**t**o-**r**oot

SVD **S**ingular **V**alue **D**ecomposition

PCA **P**rincipal **C**omponent **A**nalysis

PDE **P**artial **D**ifferential **E**quation

NMF **N**on-negative **M**atrix **F**actorization

ICA **I**ndependent **C**omponent **A**nalysis

LDA **L**atent **D**iscriminant **A**nalysis

AE **A**uto**e**ncoder

VAE **V**ariational **A**uto**e**ncoder

RBM **R**estricted **B**oltzmann **M**achine

SGD **S**tochastic **G**radient **D**escent

AI artificial intelligence

MLP **M**ulti-**L**ayer **P**erceptron

UMAP **U**niform **M**anifold **A**pproximation and **P**rojection

QoI quanity of interest

ALS **A**lternating **L**east **S**quares

FA **F**actor **A**nalysis

CCA **C**anonical **C**orrelation **A**nalysis

tSNE **t**-Distributed **S**tochastic **N**eighbor **E**mbedding

LLE **L**ocally **L**inear **E**mbedding

Isomap **Iso**metric feature **map**ping

MERA **M**ulti-scale **E**ntanglement **R**enormalization **A**nsatz

PEPS **P**rojected **E**ntangled **P**air **S**tates

TT-FOA **T**ensor **T**rain **F**irst **O**rder **A**pproximation

ITTD **I**ncremental **T**ensor **T**rain **D**ecomposition

RRE relative reconstruction error

CR compression ratio

RR reduction ratio

RL reinforcement learning

SoI system of interest

UE5 **U**nreal **E**ngine 5

RTE relative test error

ABSTRACT

This thesis addresses the fundamental challenges of processing large-scale, high-dimensional data frequently arising in machine learning and scientific computing. It focuses on developing novel incremental tensor decomposition algorithms that enable efficient processing of high-dimensional data streams and large-scale stationary datasets. This work is motivated by the necessity for scalable dimensionality reduction techniques that can provide an error-bounded approximation of high-dimensional data while providing a useful low-dimensional representation for downstream tasks such as Bayesian inference, behavioral cloning, and generative modeling.

As modern scientific simulations and data acquisition systems generate multi-dimensional datasets at unprecedented scales, ranging from terabytes in computational fluid dynamics to petabytes in atmospheric reanalysis, traditional dimensionality reduction methods fail to accommodate the streaming nature and memory constraints of these applications. On the other hand, large-scale machine learning models such as large language models and foundational models, which are trained on massive datasets, require efficient data representation techniques to reduce storage and computational costs while preserving essential data characteristics.

This thesis presents three novel algorithms for computing tensor decompositions: the Tensor-Train Incremental Core Expansion (TT-ICE) algorithm is an incremental algorithm for updating an existing approximation of tensors in tensor-train format when new tensors become available, the Batch Hierarchical Tucker leaves-to-root (BHT-l2r) algorithm computes an approximation of a batch of tensors in batch-modified hierarchical Tucker format, and the Hierarchical Tucker Rapid Incremental Subspace Expansion (HT-RISE) algorithm is another incremental algorithm for updating an existing approximation of tensors in batch hierarchical Tucker format. These algorithms are designed to overcome the limitations of existing approaches by providing theoretical guarantees on the approximation error, enabling downstream applications, providing support for data streams, and preserving the high-dimensional structure of the data.

The effectiveness of these methods is demonstrated across diverse applications in scientific computing and machine learning. For scientific data, TT-ICE achieves compression ratios

up to $300,000\times$ on particle-in-cell simulations to increase storage efficiency. It also provides compression ratios up to $32.8\times$ on chaotic PDE-based simulations of a sheet of self-oscillating gels while preserving reconstruction quality suitable for Bayesian inversion. For image and video data, HT-RISE offers on-disk compression performance comparable to JPEG encoding while maintaining the perceptual structure of the images. This enables effective downstream applications including behavioral cloning with $2.6 - 9.6\times$ training speedup and generative modeling through diffusion models.

This thesis demonstrates that tensor network embeddings can serve as powerful dimensionality reduction tools that preserve essential data characteristics while dramatically reducing storage and computational requirements. These findings contribute to the fields of scientific machine learning and high-dimensional data analysis by providing scalable algorithms that democratize access to advanced machine learning techniques for resource-constrained environments and enable real-time processing of streaming high-dimensional data. The developed methods open new possibilities for online learning, incremental model updates, and efficient data representation in scientific computing and artificial intelligence applications.

CHAPTER 1

Introduction

1.1 Motivation

Humans have evolved to perceive and process vast amounts of information from their environment, allowing them to make sense of their surroundings and learn about the world. Every second, we are bombarded with a multitude of sensory inputs, including visual, auditory and tactile stimuli. Our brains are developed to handle this deluge of information, focusing on the relevant aspects while ignoring the noise.

In the modern world, we are pursuing the advent of transferring the ability to learn to machines, aspiring to create intelligent systems that can process and understand data in a similar way to humans. On the road to creating artificial intelligence (AI), we have started with understanding and modeling the world around them, leading to the development of various mathematical and physical concepts. Then we have built upon these concepts to create algorithms and models that can simulate the world. These simulations have enabled us to create worlds within our world and analyze scenarios without actually experiencing or measuring them in real life.

Reducing the necessity of physical experiments has allowed for the exploration of complex systems and phenomena that would otherwise be very challenging or impossible to study. Then we started saving "memories" of those worlds in the form of data. Then we started accumulating these memories, building datasets that represent various aspects of the world.

We are not only creating data to understand the physical world, but also to understand how we work, how we interact with one another, and how we make decisions. With every interaction, every decision, and every action we generate data. We are now generating and collecting data at an unprecedented rate, but this data is not just a byproduct of our lives. It has become the fundamental reason why we exist in the lens of the digital age. All humanity is seen merely as data points, generating data to be processed and analyzed to be understood how they can be manipulated, how they can be subconsciously pushed to consume more.

To achieve all this, not only we need to understand the generated data but also we need to analyze and extract meaningful insights from it. The ability to extract insights from data paves the road to developments in many fields, including healthcare, drug discovery, aviation, and scientific research. As the generated data is a reflection of our world, it is inherently complex and multidimensional. A sequence of snapshots from a scientific simulation has three dimensions for space, one for time, and one for the quantities of interest (QoIs), resulting in a five-dimensional array. On the other hand, a video has two spatial dimensions, one temporal dimension, and one for color channels, resulting in a four-dimensional array.

Traditional data representation and analysis approaches, such as matrices, often fall short when it comes to handling such complex data structures. Flattening these naturally multidimensional data into matrices leads to a loss of multidimensional interactions and relationships, making it harder to be discovered by machine learning algorithms.

Tensors are natural extensions of matrices into higher dimensions, allowing for the retention of the multidimensional structure of the data. Tensor methods are gaining traction as a powerful tool for representing and analyzing higher-dimensional data. They can capture intricate relationships within the data, making them suitable for applications in machine learning, computer vision, and signal processing. The ability to represent high-dimensional data in the form of tensors led to the exploration of low-rank structures within tensors, enabling efficient representations of the data. Some of the most common tensor decomposition formats include CANDECOMP/PARAFAC (CP) format [95, 41], Tucker format [260, 259, 261], and Tensor-Train (TT) format [191]. In addition to allowing efficient storage of high-dimensional data, tensor decompositions can also act as a dimensionality reduction technique. This becomes a particularly useful tool when downstream applications, such as supervised learning or Bayesian inference, are of interest.

1.1.1 The Curse of Dimensionality

One of the major challenges in working with large-scale high-dimensional data is the *curse of dimensionality*. The phrase "curse of dimensionality" refers to the phenomenon where the storage and computational requirements for managing and working with the data grow exponentially with the number of dimensions. This phenomenon also affects the tools and algorithms used to *reduce* the dimensionality of the data, making it increasingly difficult to analyze and process them as the data gets larger.

On the way to achieving artificial general intelligence (or artificial superintelligence), we need to leverage information from various sources, including text, images, audio, and video. This leads to the generation of multimodal datasets that are not only large but also very

high-dimensional. For example, a single color video frame at 720p[1] is a 3-dimensional tensor, with 3 million pixels per frame, and when combined with the temporal dimension, it results in a four-dimensional tensor. This number does not even consider the multiple audio channels accompanying each frame. The high-dimensional, multi-modal nature of modern datasets makes it challenging to process and analyze them effectively. One common approach in tackling the curse of dimensionality in imaging applications is to downsize the images to a lower resolution, but this is often limited by the need to preserve the essential features of the data [20].

The curse of dimensionality, however, extends far beyond conventional machine learning and computer vision applications. Wherever high-fidelity modeling of complex phenomena is required, researchers experience similar exponential growth in data complexity. In the scientific machine learning [21] community[2], researchers are dealing with large-scale high-dimensional datasets that arise from first-principles simulations, such as computational fluid dynamics, molecular dynamics, and climate modeling. An eye-opening example demonstrating how things can get out of hand easily is the data generated by the S3D simulation developed by Sandia National Laboratories [51]. The S3D simulation is a high-fidelity simulation of turbulent combustion, which considers a three-dimensional domain discretized into a grid of $512 \times 512 \times 512$. Tracking 64 species using double precision floating-point numbers, the simulation generates approximately *70 gigabytes* of data *per time step* [138].

One common challenge across domains is the need to process and analyze these large-scale datasets to extract meaningful insights and make predictions. For the conventional machine learning community, this often entails training recommender engines, image classifiers, generative models or large language models. For the scientific machine learning community, this often entails training surrogate models, performing uncertainty quantification, or solving inverse problems. In both cases, the size of data poses significant challenges in terms of computational resources, memory requirements, and even in some cases, storage, especially for resource-limited hardware such as mobile devices or edge computing platforms.

This leads to the need for discovering ways to represent high-dimensional data more efficiently. This is where dimensionality reduction techniques come into play. Dimensionality reduction methods aim to reduce the number of dimensions in a dataset while maintaining the essence of the data. Next, we will discuss the main challenges in the context of dimensionality reduction for high-dimensional data.

[1] In today's standards even this is a relatively low resolution for video, but it shows the complexity of modern video data.

[2] also known as the *SciML*, *AI4Science* or *AI+Science* community

1.1.2 The Challenge of Large-Scale Data

The problem of *big data* is not a recent concept, but it has been exacerbated with the advent of modern technologies and the increasing availability of data. Nowadays, gigabyte-sized datasets have become commonplace, and terabyte-sized datasets are slowly becoming the norm, rather than being the exception.

Especially with the advent of training "large language models" (LLMs) and other large-scale foundational models, we are referring to datasets that are composed of *the entire internet* or *the entire corpus of scientific literature*. For example, it is rumored that Google turned to their catalog of 20 billion videos available on YouTube to train their latest generative model VEO3 [119]. Even 1% of that collection is estimated to contain around ~ 4400 *years* worth of videos.

This trend doesn't change for the scientific machine learning community either. One illuminating example here is the ERA5 dataset [98], which is a global atmospheric reanalysis dataset that provides hourly estimates of a large number of atmospheric, land, and oceanic variables. The ERA5 dataset is estimated to contain over *9 petabytes* of data, covering the period from 1959 to the present day, and is used for various applications, including climate research, weather forecasting, and environmental monitoring.

This is a staggering amount of data, and it is not feasible to process it all at once. One cannot even imagine performing one-shot dimensionality reduction on such large datasets, as the computational requirements may even exceed the globally available computational resources.[3] Processing such large datasets requires different algorithmic paradigms that can handle the data without loading the entire dataset into memory at once.

1.1.2.1 Incremental Algorithms for Static Datasets

As a solution to the challenges of processing large-scale stationary datasets, researchers have been exploring incremental algorithms for dimensionality reduction. Incremental algorithms are designed to process data in smaller chunks, allowing for efficient handling of large-scale data. With every new chunk of data, these algorithms update the existing representation, rather than recomputing it anew. This approach eliminates the need to load the whole dataset into the working memory at once, which is often a prohibitive step for large datasets.

Since incremental algorithms usually work with a fixed batch size, their memory footprint remains constant regardless of the total dataset size. Additionally, the smaller batch size also allows for faster processing times, as the algorithms focus on a subset of the data at a time. For incremental algorithms, the batch size becomes a crucial hyperparameter that can

[3] This might be an overstatement, but it is not too far from the truth.

be tuned to balance memory usage and processing speed.

Another important advantage of incremental algorithms is the ability to terminate the processing early, obtaining a useful representation after processing only a subset of the data. Using a cross-validation-based approach, one can determine if the representation is sufficiently accurate for the entire dataset, or if more data is needed to improve the representation quality.[4]

In case the data is distributed across multiple nodes, the incremental nature of these algorithms can be extended to support parallelization across different data chunks, enabling distributed processing of large datasets. This eliminates the need for transferring the entire dataset to a single node, which can be a bottleneck in distributed systems. Instead, each node can process its own chunk of data and update the existing representation independently.

One of the biggest challenges in designing effective incremental algorithms is maintaining the quality of the representation as the number of updates increases. Errors can accumulate over time, leading to degraded representations. Therefore, it is crucial to design incremental algorithms that can maintain a bounded error accumulation while being computationally tractable. This often requires careful analysis of how approximation errors propagate through successive updates.

1.1.2.2 The Problem of Processing Streaming Data

Streaming data represents a fundamentally different challenge from large-scale batch processing. Unlike incremental algorithms that process static datasets in manageable chunks, streaming data involves continuous data generation often at a high rate. Another key distinction is that streaming data is often unbounded, meaning that the data can arrive indefinitely without a predetermined end point. This requires algorithms that can operate indefinitely without knowing when the stream will end.

This is particularly relevant in scenarios such as sensor networks, social media platforms, and autonomous vehicles, where data is generated continuously and needs to be processed in real-time. For some safety-critical applications, such as autonomous vehicles, the data needs to be processed within a fraction of a second to make critical decisions, such as avoiding pedestrians or other vehicles.

With the increase in the computational capacity thanks to the leadership-class supercomputers [19], we are now able to run simulations with extremely high fidelity. Although these simulations usually generate snapshots at a much slower rate than that of autonomous vehicles, they can still generate terabytes of data in a single run. Especially for simulations with

[4]Please note that this approach can only provide guarantees on the representation error only *in expectation*, not in absolute terms.

very small time steps, such as particle-in-cell simulations, researchers often discard some intermediate snapshots to reduce the amount of data stored. However, this can lead to the loss of phenomena that occur on very small time scales.

A key challenge in developing algorithms for streaming data is maintaining the approximation quality of earlier data points as new data arrives, especially in scenarios where the source of the data is not stationary. This means that the underlying distribution of the data changes over time, leading to potential drift in the data. One can prefer to focus on the most recent data and employ a *forgetting factor* to gradually reduce the influence of older data points. However, this approach can lead to the risk of losing important information from the earlier data points. Alternatively, providing a bounded error guarantee for the entire stream can lead to having bloated representations, that hinder the efficiency of the algorithm.

1.1.2.3 Dimensionality Reduction for (Scientific) Machine Learning

As stated in the motivation section, the ultimate goal of machine learning is to extract meaningful insights from data. This often involves training discriminative models, helping to classify, cluster, or predict results based on the data in hand. Alternatively for scientific machine learning, this often involves training surrogate models, providing a low-cost alternative to expensive simulations, or performing uncertainty quantification, providing insights into the reliability of the predictions.

The curse of dimensionality poses a substantial challenge in this context as well. Processing high-dimensional data directly results in increased computational requirements and costs, both for training as well as for inference. This becomes prohibitive for many applications especially for institutions with limited budgets or computational resources.

Dimensionality reduction methods can help alleviate these challenges by transforming the high-dimensional data into a lower-dimensional representation, allowing for more efficient processing and analysis. Even for large-scale machine learning applications, modules that learn an embeddings are often implemented within the model architectures. In most of the cases, training of these modules are done in an end-to-end fashion, meaning that the model learns the embedding while training on the downstream task. However, such an approach leads to learning an embedding that is tailored to the specific downstream task, which may not be suitable for other tasks or applications.

Having a separate dimensionality reduction step and learning a task-agnostic embedding from the data can provide a more general representation that can be used for various downstream tasks. Moreover, having guaranteed approximation quality for the learned embeddings can provide a more reliable and generalizable representation that can be used for various downstream tasks. This can lead to more performant and reliable models in the

inference phase, as the learned mappings extract the essential features of the data.

These examples highlight the necessity and importance of developing efficient dimensionality reduction algorithms that can handle large-scale data streams. With this thesis, we aim to address the challenges posed by high-dimensional data streams by proposing fast and resource-efficient algorithms.

1.1.3 Goal of the Thesis

The primary goal of this thesis is to develop novel incremental tensor decomposition algorithms that efficiently process large-scale and streaming high-dimensional data while maintaining theoretical guarantees on approximation quality.

Specifically, this thesis aims to:

- Develop incremental algorithms for Tensor-Train and Hierarchical Tucker formats that can process data streams while maintaining bounded approximation errors
- Exploit the unique properties of different tensor network formats to enable real-time processing capabilities
- Extract low-rank structures from multidimensional data while enabling efficient updates as new data arrives
- Demonstrate effectiveness of the developed algorithms through downstream applications of the learned representations in scientific computing, behavioral cloning, and Bayesian inference

By addressing the fundamental challenges of the curse of dimensionality and large-scale data processing through tensor-based approaches, this thesis aims to democratize advanced machine learning techniques across various fields. Figure 1.1 provides an overview of our approach.

Before getting into the details of the proposed tensor-based approaches, we first provide a brief overview of the landscape of existing dimensionality reduction methods and position tensor decompositions within this landscape in the next section. This will help to understand the advantages and disadvantages of different dimensionality reduction methods, and how tensor decompositions can be used to address the challenges of large-scale and streaming data.

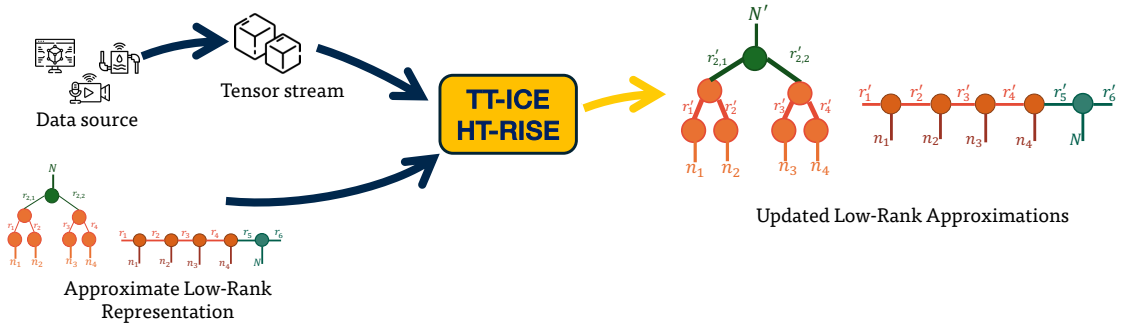


Figure 1.1: Overview of the goal of the thesis. We aim to propose novel incremental algorithms for low-rank tensor decompositions as a solution to the challenges of processing large-scale and streaming data. In addition to resulting in compression, the proposed methods also provide a mapping between the original high-dimensional data and a low-dimensional latent space (shown as the orange parts of the tensor networks), and a latent representation of the data (shown as the green parts of the tensor networks). The resulting low-dimensional representations can then be used for various downstream applications, such as supervised learning and Bayesian inference.

1.2 Dimensionality Reduction Methods

Dimensionality reduction seeks to find efficient representations of high-dimensional data that maintain the essential characteristics of the original data while discarding redundancy and noise.

The core principle of dimensionality reduction is based on the observation (or assumption) that high-dimensional data often lies on or near a much lower-dimensional manifold embedded within the high-dimensional space. This phenomenon is sometimes referred to as the *manifold hypothesis*, which posits that meaningful data typically exists in a lower-dimensional latent space, even when represented in the original high-dimensional space.

The fundamental trade-off in dimensionality reduction is between the size of the latent representation and the amount of information preserved from the original data. The goal is to find a mapping $f : \mathbb{R}^D \to \mathbb{R}^d$ where $d \ll D$, such that the reduced representation captures the most defining features of the original data. This mapping can be linear (as in PCA) or non-linear (as in manifold learning methods), depending on the underlying structure of the data.

Each dimensionality reduction method has its own objectives and trade-offs, some methods focus on preserving variance (e.g., PCA), while others focus on preserving pairwise distances (e.g., MDS), local structure (e.g., LLE), or global topological structure (e.g., UMAP). The effectiveness of dimensionality reduction is ultimately measured by its utility for downstream tasks such as visualization, machine learning, data analysis, and storage.

As datasets grow to unprecedented scales and arrive in streaming fashion, traditional

batch approaches to dimensionality reduction become computationally infeasible. This challenge has motivated the development of incremental algorithms that can process data in chunks while maintaining global coherence in the reduced representation. Unfortunately, not all dimensionality reduction methods are suitable for incremental processing or require significant modifications to support incremental updates.

With this conceptual foundation in mind, we now examine specific families of dimensionality reduction methods, analyzing how each addresses these general principles and where incremental variants have been developed to handle large-scale data challenges. We categorize these methods into four main categories: linear methods, non-linear methods, deep learning-based methods, and tensor decomposition methods. Although deep learning-based methods and tensor methods can be considered as a subset of non-linear methods, we treat them separately due to their distinct characteristics and applications. Finally, we compare and discuss the issues with each of these categories of methods, highlighting their strengths and weaknesses in the context of large-scale data and the problem of incremental data processing.

1.2.1 Linear Methods

The first family of dimensionality reduction methods we consider are linear methods. These methods perform a linear transformation of the data to extract a lower-dimensional representation. In the context of multidimensional data, consider a dataset comprised of N standard color images of shape $h \times w \times c$, where h is the height, w is the width, and c is the number of color channels. Linear methods typically flatten the images into vectors of size hwc and then stack each individual image vector into a matrix of size $hwc \times N$. The dimensionality reduction is then performed on this matrix, leading to a lower-dimensional representation of the data. Some of the most common linear dimensionality reduction methods include:

- **Principal Component Analysis (PCA)**: PCA [199, 108] is a widely used linear method that identifies the directions of maximum variance in the data. It computes the eigenvectors of the covariance matrix of the data and projects the data onto a lower-dimensional subspace spanned by the top eigenvectors. PCA is widely used in various applications, including image compression [75], feature extraction [146], and data visualization [121]. It is particularly effective when the data is linearly separable and a few principal components capture the most significant features of the data.

- **Factor Analysis (FA)**: FA [239, 255] is a statistical method that models the observed data as a linear combination of latent factors. It assumes that the observed variables are influenced by a smaller number of unobserved factors, which can be estimated from

the data. FA is particularly useful for identifying underlying structures in the data and reducing dimensionality while preserving the relationships between variables. It is often used in marketing [241], psychology [82] and social sciences [167] to identify the underlying factors that contribute to the observed behavior or responses of individuals.

- **Latent Discriminant Analysis (LDA)**: LDA [81, 211, 212] is a supervised dimensionality reduction technique that aims to find a linear combination of features that best separates different classes in the data. It is often used in classification [23] tasks, where the goal is to find a low-dimensional representation of the data that maximizes the separation between different classes. LDA is very similar to PCA in that it finds a linear transformation of the data, but it differs in that it focuses on maximizing the separation between different classes rather than maximizing the variance of the data. This makes LDA particularly useful dimensionality reduction tool for classification [23] or clustering [158] tasks.

- **Independent Component Analysis (ICA)**: ICA [62, 117] assumes that the observed data is a linear combination of statistically independent components, and aims to find a linear transformation that separates these components. It is often used in signal processing [123, 274] and blind source separation [59] tasks. ICA mathematically models the observed data as a linear combination of independent components.

- **Canonical Correlation Analysis (CCA)**: CCA [109] is a method that finds linear combinations of two sets of variables such that the correlations between the combinations are maximized. It is particularly useful for exploring relationships between two datasets and can be used for dimensionality reduction by projecting the data onto the canonical variables. CCA is effective for capturing the shared information between two datasets. This makes CCA particularly useful for multi-view learning [92, 127] and multi-modal data analysis [249], where the goal is to find a low-dimensional representation of the data that captures the relationships between different views or modalities.

- **Non-negative Matrix Factorization (NMF)**: NMF[151, 153, 240, 252] is a method that factorizes a non-negative matrix into two non-negative matrices. It is particularly useful for applications where the data is non-negative, such as text data [226, 96], audio signals [30], images [160], and computational biology [71]. NMF is effective for extracting meaningful features from non-negative data while preserving the non-negativity constraint. NMF is also known under the name of *positive matrix factorization* [196, 195, 13].

Among linear methods, we will have a slightly closer look at **S**ingular **V**alue **D**ecomposition (SVD) as it plays a central role in the algorithms proposed in this thesis. Singular value decomposition(SVD) [85, 86] eliminates the need for centering the data matrix and provides a more numerically stable option for performing linear dimensionality reduction. Furthermore, SVD eliminates the need to construct the covariance matrix $C = \frac{1}{n-1}AA^T$, which can become prohibitive for very large n. SVD is one of the most well known and widely used dimensionality reduction tool in the literature, with examples from various fields, including image compression [206], natural language processing [89], and recommender systems [33].

SVD represents a matrix $A \in \mathbb{R}^{m \times n}$ as a product of three matrices:

$$A = U\Sigma V^T, \tag{1.1}$$

where $U \in \mathbb{R}^{m \times m}$ and $V \in \mathbb{R}^{n \times n}$ are orthogonal matrices, $\Sigma \in \mathbb{R}^{m \times n}$ is a rectangular diagonal matrix with non-negative real values on the diagonal. There is an inherent connection between the right singular vectors V and the eigenvectors of the covariance matrix from the PCA. The dimensionality reduction is achieved by truncating the SVD to the top k singular values and their corresponding singular vectors, resulting in the following approximation:

$$A \approx U_k \Sigma_k V_k^T, \tag{1.2}$$

where $U_k \in \mathbb{R}^{m \times k}$, $\Sigma_k \in \mathbb{R}^{k \times k}$, and $V_k \in \mathbb{R}^{n \times k}$ are the truncated matrices containing the top k singular values and their corresponding singular vectors, also known as the *rank* of the matrix $A_k = U_k \Sigma_k V_k^T$. The Eckart-Young-Mirsky theorem [77, 180] states that the truncated SVD provides the best low-rank approximation of the matrix A in terms of the Frobenius norm and the error is given by:

$$\|A - U_k \Sigma_k V_k^T\|_F = \sqrt{\sum_{i=k+1}^{\min(m,n)} \sigma_i^2}, \tag{1.3}$$

where $\|\cdot\|_F$ denotes the Frobenius norm and σ_i is the i-th singular value of the matrix A. When the truncation rank k is determined based on the desired approximation error, we refer to this as an *error-truncated SVD* and when the truncation rank k is determined based on the desired reduced dimensionality, we refer to this as a *rank-truncated SVD*. The SVD is a powerful tool for dimensionality reduction, as it can capture the most important features of the data while discarding the noise and less significant components.

Computing the SVD of an $m \times n$ matrix requires $\mathcal{O}(\min(m,n)mn)$ time and $\mathcal{O}(\min(m,n)^2)$

space, which can be prohibitive for very large datasets Even for a 720p image (1280×720 pixels) with 3 color channels, m becomes $2,764,800$ and even for an hour long video at 60 frames per second, we have $n = 216,000$ frames. Considering video based datasets with hundreds, if not thousands of hours of video, both the time and space complexity of computing the SVD becomes prohibitive. Incremental algorithms for computing the SVD can help address this issue by allowing us to compute and update the SVD of a matrix using batches of data, rather than requiring the entire matrix to be available at once.

In the literature there are various algorithms that are designed to perform incremental SVD [34, 39, 32, 22]. In addition to solving the large-scale data processing problem, these algorithms also provide a way to update the SVD of a matrix as new data arrives, which is particularly useful in online learning scenarios. The approach proposed by Brand [32] is one of the most widely used incremental SVD algorithms. It reduces the time complexity to $\mathcal{O}(mnr)$ and the space complexity to $\mathcal{O}((m+n)r)$, where r is the rank of the approximation.

Consider we have an existing SVD of a matrix $A = U\Sigma V^T$ like Equation (1.1) and we want to update it with a new batch of data $B \in \mathbb{R}^{m \times n_b}$, where n_b is the number of new columns. The update starts with projecting the new data B onto the existing subspace spanned by the left singular vectors U and finding the residual of the orthogonal projection:

$$\tilde{B} = U^T B, \quad \text{and} \quad R = B - U\tilde{B}. \tag{1.4}$$

The residual R represents the part of the new data that is not captured by the existing subspace. We can then assume a decomposition of the residual $R = JK$ such that $J \in \mathbb{R}^{m \times n_b}$ is an orthogonal matrix. A possible choice to obtain such a decomposition is to use the QR decomposition, which can be computed efficiently. Using this information, we can construct the augmented data matrix $\tilde{A} = \begin{bmatrix} A & B \end{bmatrix} \in \mathbb{R}^{m \times (n+n_b)}$ as:

$$\tilde{A} = \begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} U & J \end{bmatrix} \begin{bmatrix} \Sigma & \tilde{B} \\ 0 & K \end{bmatrix} \begin{bmatrix} V & 0 \\ 0 & \mathrm{I} \end{bmatrix}^T. \tag{1.5}$$

Note that the left and right matrices in Equation (1.5) are orthogonal matrices. Then we can compute the SVD of the augmented data matrix $\tilde{A}$ by performing an SVD on the block upper triangular matrix as:

$$S = \begin{bmatrix} \Sigma & \tilde{B} \\ 0 & K \end{bmatrix} = \tilde{U}\tilde{\Sigma}\tilde{V}^T, \tag{1.6}$$

where $\tilde{U} \in \mathbb{R}^{(r+m) \times r'}$, $\tilde{\Sigma} \in \mathbb{R}^{r' \times r'}$, and $\tilde{V} \in \mathbb{R}^{(r'+n_b) \times r'}$ are the left singular vectors, singular values, and right singular vectors of the augmented data matrix S with the new approxima-

tion rank r'. The updated SVD can then be obtained as:

$$\tilde{A} = (U\tilde{U})S(\tilde{V}^T V^T) = U'\Sigma' V'^T. \tag{1.7}$$

Performing orthogonal projections onto the existing subspace spanned by the left singular vectors and then computing the SVD of the residual will play a central role in the algorithms proposed in this thesis. We save a more detailed discussion of how this approach is used in the algorithms proposed in this thesis to Chapter 3.

1.2.2 Non-linear Methods

The methods presented in Section 1.2.1 are all powerful methods that have a wide range of applications in the literature. However, they all suffer from the same limitation: they all assume that the latent structure of the data can be revealed through linear transformations. In many cases, this assumption falls short, as the real-world data is often non-linear and complex. To address this limitation of the linear methods, a number of non-linear dimensionality reduction tools are proposed in the literature. These methods aim to find a non-linear transformation of the data that captures the underlying structure of the data, while preserving the relationships between the data points. Below we provide a brief overview of some of the most common non-linear dimensionality reduction methods.

- **Kernel PCA**: Kernel PCA [221, 176] is an extension of PCA that uses kernel methods to perform non-linear dimensionality reduction. It maps the data into a higher-dimensional feature space using a kernel function [222] and then applies PCA in that space. This allows it to capture non-linear relationships in the data while still providing a low-dimensional representation. Kernel PCA is used in various applications, such as image processing [76], speech recognition [157], and bioinformatics [213]. It is particularly effective when the data is non-linearly separable and the kernel function captures the underlying structure of the data. However, Kernel PCA has limitations when dealing with very large datasets, as the kernel matrix can become prohibitively large and computationally expensive to compute. Furthermore, the choice of kernel function and its parameters can significantly affect the performance of Kernel PCA.

- **Isometric feature mapping (Isomap)**: Isomap [254] is a non-linear dimensionality reduction method that aims to preserve the geodesic distances between data points in a high-dimensional space. It is particularly useful for data that lies on a low-dimensional manifold embedded in a higher-dimensional space. This creates a fundamental distinction from linear methods such as Principal Component Analysis (PCA), as Isomap

captures the true nonlinear geometry of curved manifolds rather than relying on Euclidean distance preservation. Isomap is used in various applications, such as facial recognition [163], and image classification [188]. It is particularly effective when the data lies on a low-dimensional manifold embedded in a higher-dimensional space, as it captures the true non-linear geometry of the manifold.

- **Locally Linear Embedding (LLE)**: LLE [217] is a non-linear dimensionality reduction method that aims to preserve the local structure of the data by reconstructing each data point as a linear combination of its neighbors. This creates a clear contrast to Isomap's geodesic distance preservation, as LLE focuses on local linearity rather than global distances. It is particularly useful for data that lies on a low-dimensional manifold embedded in a higher-dimensional space. LLE works by first constructing a neighborhood graph of the data points, where each of the n points are connected to their k nearest neighbors. LLE has been widely used in various fields due to its ability to preserve local structure while providing a meaningful low-dimensional representation of complex data. Some examples include hyperspectral image analysis [166, 275], gene expression data analysis [202], tissue classification [162] and tumor detection [301].

- **Laplace Eigenmaps**: Laplace Eigenmaps [28] is a non-linear dimensionality reduction method that constructs a graph representation of the data and then finds a low-dimensional embedding that preserves the local structure of the data by constructing the graph Laplacian. It is particularly useful for data that lies on a low-dimensional manifold embedded in a higher-dimensional space. Laplace Eigenmaps aim to find a low-dimensional representation of the data that preserves the local relationships between the data points, while also ensuring that the representation is smooth and continuous. Laplace Eigenmaps works by first constructing a neighborhood graph of the data points, where each of the n points are connected to their k nearest neighbors. The method then computes the eigenvectors of the Laplacian matrix of the graph and uses them to find a low-dimensional embedding that preserves the local structure of the data. Laplace Eigenmaps have been used in medical image analysis [159], biomedical network analysis [296], and semantic word embeddings [28].

- **t-Distributed Stochastic Neighbor Embedding (tSNE)**: (tSNE) [264] is a non-linear dimensionality reduction technique that is particularly effective for visualizing high-dimensional data in a lower-dimensional space, typically in two or three dimensions. Its key distinction from other non-linear methods is that it uses heavy-tailed distributions to model the pairwise similarities between data points, which allows it to

capture the local structure of the data more effectively. Stochastic Neighbor Embeddings were first introduced by Roweis and Hinton [102], van der Maaten and Hinton then proposed the t-distributed variant tSNE [264] and has since become one of the most popular methods for visualizing high-dimensional data. tSNE has been used primarily for visualizing high-dimensional data, such as single-cell RNA sequencing [136], human genetic data analysis [154], and drug discovery [125].

- **Uniform Manifold Approximation and Projection (UMAP)**: (UMAP) [173] is a non-linear dimensionality reduction technique that is particularly effective for visualizing high-dimensional data in a lower-dimensional space. It works by constructing a fuzzy topological representation of the data and then optimizing a low-dimensional embedding that preserves the local and global structures of the data. UMAP is widely used for visualizing complex datasets, such as images, text, and biological data. It is particularly effective for capturing both local and global structures in the data and revealing clusters or patterns that may not be apparent in the original high-dimensional space. UMAP is computationally efficient and can handle large datasets, making it a popular choice for dimensionality reduction and visualization in various applications such as population genetics [73], single cell RNA sequencing [26], and image analysis [173].

1.2.3 Deep Learning Based Methods

The next family of dimensionality reduction methods we consider are based on deep learning techniques. These methods leverage the power and expressivity of neural networks to learn a low-dimensional representation of the data. The key overarching advantage of the deep learning based methods is that they can learn complex non-linear transformations of the data, which can capture intricate patterns and structures that are often difficult to model with other non-linear methods. Also, deep-learning based methods can be trained end-to-end, allowing for joint optimization of the dimensionality reduction and downstream tasks, such as classification or clustering. This can lead to improved performance on these tasks compared to traditional dimensionality reduction methods. Finally, deep learning based methods can scale to large datasets and can be trained on GPUs, making them suitable for high-dimensional data with millions of samples. Below we provide a brief overview of some of the most popular deep learning based dimensionality reduction methods.

- **AE**: AEs [103, 271] are probably the most well-known and widely used deep learning based dimensionality reduction methods. They are a type of neural network that learns

to encode the input data into a lower-dimensional representation and then decode it back to the original data. The key idea is to train the network to minimize the reconstruction error between the input and the output, which encourages the network to learn a compact representation of the data. AEs are widely used for dimensionality reduction, feature extraction, and data compression. They can be used for various types of data, including images, text, and audio. AEs can also be used in combination with other techniques, such as clustering or classification, to improve performance on these tasks. AEs have been widely used in various applications, such as image compression [57], anomaly detection [54], and feature extraction [118]. They have also been shown to be effective for dimensionality reduction in high-dimensional data, where traditional methods may struggle to capture the underlying structure of the data.

- **VAE**: VAEs [134, 135] are a generative model that learns a probabilistic mapping from the input data to a lower-dimensional latent space. It is an extension of AEs that learn a distribution over the latent space, allowing for more flexible and expressive representations and generation of new samples from the learned latent space. VAEs are widely used for generative modeling, data synthesis, and representation learning. The generative capability makes VAEs particularly useful for applications such as data augmentation [120], where generating new samples can help improve the performance of downstream tasks. They can be used for various types of data, including images, text, and audio.

- **Restricted Boltzmann Machine (RBM)**: RBMs [233, 101] are another type of generative model that learns a probabilistic mapping from the input data to a lower-dimensional latent space. RBMs are a type of undirected graphical model that consists of a visible layer and a hidden layer, where the visible layer represents the input data and the hidden layer represents the latent variables. They are an extension of the Boltzmann machines [2] where RBMs are restricted to have a bipartite structure, meaning that there are no connections within the visible layer or within the hidden layer. RBMs have been widely used in various applications, such as collaborative filtering [219], classification [150], and feature extraction. They have also been shown to be effective for dimensionality reduction in high-dimensional data, where traditional methods may struggle to capture the underlying structure.

1.2.4 Tensor Decompositions

The final family of dimensionality reduction methods we consider are low-rank tensor decompositions. As tensors are generalizations of matrices to higher dimensions, tensor decom-

positions can be considered as extensions of matrix decompositions to higher-dimensional data. Tensor decompositions aim to represent a larger tensor using simpler linear algebra objects, such as vectors, matrices, or (smaller) tensors. We reserve a deeper discussion of tensor algebra, including a closer look at tensor decomposition formats, various algorithms to compute tensor decompositions, and incremental algorithms, for Chapter 2. However, we provide a high-level overview of tensor decompositions in this section to motivate the use of tensors for dimensionality reduction.

We can analyze the tensor decomposition formats under the following three main categories:

- **Flat-sum formats:** These formats represents a tensor as a sum of outer products of lower-dimensional linear algebra objects, such as vectors or matrices. Examples for this family include the CP format [41], Tensor BM-decompositon [256], and block-term decomposition [67].

- **Core-based formats:** These formats represent a tensor as a smaller core tensor multiplied by factor matrices. Examples for this family include the Tucker decomposition [260].

- **Network-based formats:** These formats represent a tensor as a network of smaller tensors, such as the HT format [90, 192], TT format [191], Tensor-Ring format [300], Tensor-Star format [303], and many others.

There is no "one-size-fits-all" tensor decomposition format, and the choice of the format depends on the specific application and the properties of the data. Each format has its own advantages and disadvantages in terms of computational efficiency, interpretability, and scalability to large datasets. For example, CP format is often preferred for its simplicity and interpretability, as it represents a tensor as a sum of rank-one tensors. However, it may not be suitable for large tensors due to its high computational cost. On the other hand, Tucker decomposition is more flexible and can handle larger tensors, but it encounters computational challenges when dealing with very high-dimensional data. HT and TT formats are particularly useful for large-scale data, as they can represent tensors with a much smaller number of parameters compared to other formats, but they may be far less interpretable.

Tensor decompositions are particularly useful for representing images, videos, and scientific simulations as they can preserve the multidimensional structure of the data while providing a compact representation. Additionally, in some cases, tensor decompositions also result in a latent representation of the data, which becomes useful for various downstream tasks, such as supervised learning, clustering, and Bayesian inference.

1.2.5 Comparison of Dimensionality Reduction Methods

In this section, we provide a comparison of the dimensionality reduction methods discussed in this chapter across several key properties that are important for large-scale applications. The comparison is summarized in Table 1.1. The properties discussed in the table depend on the specific algorithms picked for each method, but we

Table 1.1: Comparison of dimensionality reduction methods across key properties for large-scale applications. The proposed methods in this thesis are denoted with the relevant section numbers. Note that the TT-ICE algorithm is not the only incremental algorithm for the TT format, but it is the only one that provides theoretical guarantees on the approximation error and scales to large-scale data. Symbols: ✓ = Well-supported, △ = Possible with modifications, ∘ = Partially supported/Limited, × = Not supported/Poor performance. Please see the text for more details on the interpretation of each symbol for the context of each property.

Method	Streaming Data	Large-Scale Scalability	Theoretical Guarantees	High-Dim. Data	Out-of-Sample Mapping
Linear Methods					
PCA	✓	✓	✓	∘	✓
SVD	✓	✓	✓	∘	✓
FA	∘	∘	∘	△	✓
LDA	✓	∘	✓	△	✓
ICA	✓	∘	∘	△	✓
CCA	✓	∘	✓	∘	✓
NMF	✓	∘	∘	∘	✓
Non-linear Methods					
Kernel PCA	△	△	∘	∘	✓
Isomap	△	△	∘	×	△
LLE	×	×	∘	×	△
tSNE	△	∘	×	△	△
UMAP	△	✓	×	△	✓
Deep Learning Methods					
AE	✓	✓	×	✓	✓
VAE	✓	✓	×	✓	✓
RBM	✓	∘	×	∘	✓
Tensor Decompositions					
CP	✓	∘	∘	✓	✓
Tucker	✓	∘	✓	✓	✓
TT	✓ (Sec. 3.1)	✓	✓	✓	✓
HT	×	✓	✓	✓	✓
Batch-HT	✓ (Sec. 3.3)	✓	✓ (Sec. 3.2)	✓	✓

Interpretation of the Comparison: Table 1.1 provides a systematic comparison of the dimensionality reduction methods discussed in this chapter across five critical properties for large-scale data applications:

- **Streaming Data Support:** The ability to process data incrementally as it arrives, without requiring the entire dataset to be available simultaneously. Most traditional methods (×) require batch processing, while incremental variants exist for some methods (✓). If there is a budgeted or amortized approach to handle streaming data, it is marked with (△).

- **Large-Scale Scalability:** The computational and memory efficiency when dealing with datasets containing millions or billions of data points. Methods marked with ($\times$) struggle with computational complexity, while ($\circ$) indicates reasonable scalability with some limitations. Cases that scale via approximate feature maps, subsampling, or ANN/landmark graphs are marked ($\triangle$). Deep learning and tensor methods ($\checkmark$) demonstrate superior scalability.

- **Theoretical Guarantees:** The availability of mathematical bounds on approximation error or convergence properties. Linear methods generally provide strong theoretical foundations ($\checkmark$), while many non-linear and deep learning methods lack rigorous error bounds ($\times$). Some of the methods, provide guarantees under specific conditions/assumptions (e.g., manifold smoothness, separability, or regularization) are marked ($\circ$).

- **High-Dimensional Data:** Effectiveness when the number of features greatly exceeds the number of samples, or when dealing with naturally high-dimensional data structures. Traditional methods often suffer from the curse of dimensionality ($\times$). Cases that become effective with regularization or a PCA pre-step are marked ($\triangle$), while modern approaches like tensor methods are specifically designed for such scenarios ($\checkmark$).

- **Out-of-Sample Mapping:** The ability to project new, previously unseen data points into the learned low-dimensional representation. Most methods support this natively ($\checkmark$), while some manifold-learning-based approaches require modifications or add-ons (such as parametric alternatives, Nyström approximations, or per-point optimizations) ($\triangle$) or lack this capability entirely ($\times$).

This comparison reveals that while traditional linear methods provide theoretical rigor, they struggle to scale to modern data challenges. Non-linear methods offer flexibility but often lack scalability and theoretical guarantees. Deep learning approaches provide scalability and expressiveness but sacrifice theoretical understanding. The tensor decomposition methods, particularly the incremental algorithms proposed in this thesis, aim to combine the best aspects: theoretical guarantees, scalability, and streaming capability.

1.2.6 Issues with the Existing Methods

While the existing dimensionality reduction methods have been widely used and have shown effectiveness in various applications, each method has its own limitations and challenges. With this work, we aim to provide solutions to the following issues:

- **Scalability to extreme-scale data:** Contemporary applications often involve massive datasets. Many existing methods struggle to scale well to extremely large datasets, which are becoming increasingly common both in scientific fields and in industrial applications.

- **Non-linearity:** Many existing methods are linear in nature and may not capture the complex non-linear relationships in the data. This can limit their effectiveness in applications where the data has non-linear structures.

- **Mapping unseen data to the low-dimensional space:** Some existing methods do not provide a way to map unseen data points to the low-dimensional space, which can limit their applicability in real-world scenarios where mapping new data points to the learned low-dimensional space is essential for downstream tasks.

- **High-dimensional data:** Many existing methods struggle with high-dimensional data, where the number of features is much larger than the number of samples. This can lead to overfitting and poor generalization performance.

- **Streaming data:** Some existing methods are designed for static datasets and do not consider the streaming nature of data, where new data points arrive continuously over time. This is also helpful for applications where the data sits on a distributed system, or when the data is too large to fit into memory at once.

- **Computational efficiency:** Some of the discussed methods are computationally expensive and may not be suitable for real-time applications or large-scale datasets. This can limit their applicability in practice.

- **Lack of theoretical guarantees:** Many existing methods (especially the deep learning based methods) do not provide theoretical guarantees on the approximation error of the low-dimensional representation, which can make it difficult to assess the quality of the learned representations and their suitability for downstream tasks.

These limitations indicate that there is a need for novel methods that can address the emerging challenges for large-scale data-driven applications. The goal of this thesis is to propose novel approaches that can effectively address these challenges while providing the necessary theoretical guarantees and computational efficiency.

1.3 Contributions of the Thesis

The main contributions of this work include the development of two novel algorithms that compute and update an error-guaranteed approximation of large-scale, high-dimensional data using low-rank tensor decompositions. The proposed algorithms are designed to handle streaming data and can provide theoretical bounds on the approximation error for the compressed data, even in the case of infinite streams. Additionally, we provide practical evidence that the proposed result in effective low-dimensional representations of the data, which can be used for various downstream applications, such as supervised learning, generative modeling and Bayesian inference. Specifically, the contributions of this thesis are as follows:

- In Section 3.1, we propose a state-of-the-art incremental algorithm that computes and updates the tensor train representation of an approximation of a tensor in TT format.
- In Section 3.2, we propose a batch modified version of the hierarchical Tucker format that is designed for batches of data. We also show the efficacy of the proposed format against the naive HT format in terms of computational efficiency and storage requirements.
- In Section 3.3, we propose a second novel incremental algorithm that computes and updates the batch modified hierarchical Tucker representation of a tensor batch-HT format.
- In Sections 3.1 to 3.3, we provide theoretical guarantees on the approximation upper bound for all proposed algorithms. Furthermore, we show that both algorithms maintain the approximation error upper bounds even in the case of infinite streams of data.
- In Chapter 3, we conduct extensive experiments on real-world datasets to validate the performance of the proposed methods. We demonstrate that the proposed algorithms can effectively handle large-scale data and provide low-dimensional representations.
- In Chapters 4 and 5, we explore the applications of the latent representations obtained from the proposed algorithms in various domains, including compression of scientific datasets, behavioral cloning, generative modeling, and Bayesian inference.

These contributions have led to a number of conference presentations and peer-reviewed publications. The relevant publications originated from this work are:

1. An Incremental Tensor Train Decomposition Algorithm [7]

2. Compressed Analysis of Electric Propulsion Simulations using Low Rank Tensor Networks [8]

3. Low-Rank Tensor-Network Encodings for Video-to-Action Behavioral Cloning [49]

4. Incremental Hierarchical Tucker Decomposition [6]

1.4 Outline and Structure of the Thesis

Chapter 2 provides a detailed introduction to tensors, including some basic definitions, notations, and properties. We also discuss the different tensor decomposition formats, algorithms to compute them, and incremental algorithms for updating the decompositions as new data arrives. Then, Chapter 3 presents the main contributions of this thesis, including the TT-ICE algorithm, the batch-HT format and the HT-RISE algorithm. Chapter 4 presents some downstream applications of the proposed algorithms in scientific machine learning context, including performing Bayesian inference on the latent space obtained through the tensor network encodings. Chapter 5 presents some downstream applications of the proposed algorithms concerning image and video based datasets, including behavioral cloning and generative modeling. Finally, Chapter 6 concludes the thesis with a summary of the contributions and a discussion of potential future directions.

CHAPTER 2

Tensors and Tensor Decompositions

This chapter focuses on providing the necessary background on tensors and tensor decompositions. We start with a mathematical and notational overview of tensors. Then we will take a closer look at the foundational tensor decomposition formats relevant our work such as CANDECOMP/PARAFAC (CP), Tucker, Tensor Train (TT), Hierarchical Tucker (HT), and Batch-HT.

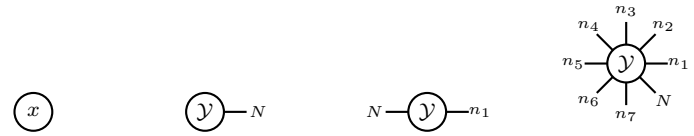
2.1 Background on Tensors and Tensor Algebra

Tensors are mathematical objects that generalize scalars, vectors, and matrices to higher dimensions. They are the natural way to represent and manipulate real-world data arising in many fields, such as engineering [220], social sciences [155, 231], and machine learning [267]. For example, a color image is represented as a 3D tensor, where the three dimensions correspond to the height, width, and color channels (e.g., RGB). Adding the time dimension to the image data results in a video, which is represented as a 4D tensor. Higher-dimensional structures appear naturally in scientific computing, simulating the fluid flow in a 3D space over time and tracking multiple quanity of interest (QoI)s (e.g., pressure, velocity, temperature) at each point in space and time results in a 5D tensor. Tensors can also be used to represent complex relationships between multiple variables, such as in social network analysis, where the nodes represent individuals and the edges represent relationships between them.

The notion of tensors does not only generalize towards higher dimensions, but also encapsulates the lower-dimensional objects such as scalars, vectors, and matrices. For example, a scalar is a 0D tensor (e.g., $\alpha \in \mathbb{R}$), a vector is a 1D tensor (e.g., $a \in \mathbb{R}^m$), and a matrix is a 2D tensor (e.g., $A \in \mathbb{R}^{m \times n}$). Anything beyond that is referred to as a higher-order tensor (e.g., $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$). Each element of a tensor is indexed by a set of indices corresponding to the individual dimensions of the tensor. As an example, a 3D tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$

has elements indexed by three indices, such as $\mathcal{A}(i, j, k)$, where $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$, and $k = 1, \ldots, n_3$. Note that this definition of tensors is purely mathematical and does not imply any specific physical interpretation.

Tensors can be visualized as multidimensional arrays, where each element is indexed by a set of indices corresponding to the dimensions of the tensor. For example, a 2D tensor (matrix) can be intuitively pictured as a grid of elements, while a 3D tensor can be pictured as a cube of elements. Unfortunately, creating a mental image of higher-dimensional tensors is not straightforward, as our brains are not equipped to visualize more than three dimensions. Thankfully, tensor network diagrams provide a way to visualize tensors and their relationships in a more intuitive manner. Tensor network diagrams aid us mortals in visualizing tensors and their operations using a graphical representation. In these diagrams, tensors are represented as nodes, and the connections between the nodes represent the indices of the tensors. Each node corresponds to a tensor, and each edge corresponds to an index (or dimension) of the tensor. The number of edges connected to a node corresponds to the number of dimensions of the tensor. For example, a 2D tensor (matrix) is represented as a node with two edges, while a 3D tensor is represented as a node with three edges. Figure 2.1 shows tensor network diagrams for tensors of various dimensions, including 0D, 1D, 2D, and 8D tensors.



(a) 0D tensor (b) 1D tensor (c) 2D tensor (d) 8D tensor

Figure 2.1: Tensor network diagram representations for arrays of various dimensions

We can extract various views from a tensor even without changing any of its elements. These views can reveal different insights and correlations across different dimensions of the tensor. This is done through fixing a subset of indices and extracting the corresponding *subtensor*. To indicate that we take all elements along a specific index, we use the colon notation. For example, for a 3D tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the subtensor corresponding to the first index can be obtained by fixing the second and third indices, resulting in a matrix $\mathcal{A}(i, :, :)$ for $i = 1, \ldots, n_1$. Similarly, the subtensor corresponding to the second index can be obtained by fixing the first and third indices, resulting in a matrix $\mathcal{A}(:, j, :)$ for $j = 1, \ldots, n_2$. The subtensor corresponding to the third index can be obtained by fixing the first and second indices, resulting in a matrix $\mathcal{A}(:, :, k)$ for $k = 1, \ldots, n_3$.

Some of those subtensors have special names. A *slice* of a tensor is a 2D subtensor

obtained by fixing all but two of the indices. For example, for the 3D tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the j-th slice corresponding to the second index can be obtained by fixing the second index as j and taking all the elements along the first and third indices, resulting in a matrix $\mathcal{A}(:, j, :) \in \mathbb{R}^{n_1 \times n_3}$ for $j = 1, \ldots, n_2$.

A *fiber* of a tensor is a 1D subtensor obtained by fixing all but one of the indices. For example, for the same 3D tensor $\mathcal{A}$, the (i, j)-th fiber corresponding to the third index can be obtained by fixing the first and second indices as i and j, respectively, resulting in a vector $\mathcal{A}(i, j, :) \in \mathbb{R}^{n_3}$ for $i = 1, \ldots, n_1$ and $j = 1, \ldots, n_2$.

Sometimes we need to convert a tensor into a matrix to perform certain operations or to analyze the data in a different way. This is done through a process called *unfolding* or *matricization*. Unfolding a tensor involves rearranging its dimensions into a matrix format, where each row corresponds to a specific index of the tensor, and each column corresponds to the remaining indices. The resulting matrix can then be used for various operations, such as matrix multiplication, singular value decomposition, or other linear algebra techniques.

A familiar example of unfolding is the *vectorization* of a matrix, where we stack the columns of a matrix to construct a single column vector. For example, for a matrix $A \in \mathbb{R}^{n_1 \times n_2}$, the vectorization results in a vector $a \in \mathbb{R}^{n_1 n_2}$ where the elements are ordered as $a(i + (j-1)n_1) = A(i, j)$ for $i = 1, \ldots, n_1$ and $j = 1, \ldots, n_2$. This is a special case of unfolding, where we reshape the matrix into a vector. We will refer to this special case of unfolding as $\text{vec}(\cdot)$.

In higher dimensions, *mode-n unfolding* (or matricization) of a tensor is a way to reorganize the tensor into a matrix by rearranging its dimensions. This operation is performed by taking the fibers along the n-th dimension of the tensor and rearranging them into columns of the resulting matrix. For example, for our 3D tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the mode-1 unfolding results in a matrix $A_{(1)} \in \mathbb{R}^{n_1 \times n_2 n_3}$, where the first dimension corresponds to the first index of the tensor, and the second dimension corresponds to the product of the other two indices. Similarly, the mode-2 unfolding results in a matrix $A_{(2)} \in \mathbb{R}^{n_2 \times n_1 n_3}$, and the mode-3 unfolding results in a matrix $A_{(3)} \in \mathbb{R}^{n_3 \times n_1 n_2}$. In general, the mode-$n$ unfolding of a d-dimensional tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ results in a matrix $A_{(n)} \in \mathbb{R}^{n_n \times (n_1 n_2 \cdots n_{n-1} n_{n+1} \cdots n_d)}$, where the first dimension corresponds to the n-th index of the tensor, and the second dimension is the product of all other dimensions. The mode-n unfolding operation will be represented as $\texttt{unfold}(\mathcal{A}, n)$ in the rest of the text. Please note that this operation is not the same as the *reshaping* operation, which takes the elements of a tensor individually and rearranges them into the targeted shape without caring about maintaining the order of the fibers, either starting from the first dimension or the last dimension.

The reverse of the unfolding operation is called *folding*, which takes a matrix (or a lower-

dimensional tensor) and rearranges it back into a higher-dimensional tensor. The folding operation is simply performed as reshaping the tensor into the desired shape. For example, for a matrix $A \in \mathbb{R}^{n_1 \times n_2 n_3}$, the folding operation results in a 3D tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ by reshaping the matrix into the desired shape. This operation will become useful when we want to artificially increase the dimensionality of a tensor to gain computational advantages in the context of tensor decompositions.

Tensors can also be manipulated using various operations, such as tensor addition, multiplication, and contraction. *Tensor addition* is performed element-wise, where two tensors of the same shape are added together by adding their corresponding elements. For example, for two 4D tensors $\mathcal{A} \in \mathbb{R}^{m \times n \times p \times q}$ and $\mathcal{B} \in \mathbb{R}^{m \times n \times p \times q}$, the tensor addition is defined as $\mathcal{C}(i,j,k,l) = \mathcal{A}(i,j,k,l) + \mathcal{B}(i,j,k,l)$ for $i = 1, \ldots, m$, $j = 1, \ldots, n$, $k = 1, \ldots, p$, and $l = 1, \ldots, q$.

Tensor multiplication can be defined in several ways, including the outer product and inner product. The outer product of two tensors results in a new tensor with a higher dimension, while the inner product results in a scalar or a lower-dimensional tensor. For example, for two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{p \times q}$, the outer product results in a 4D tensor $C \in \mathbb{R}^{m \times n \times p \times q}$, where each element is defined as $C(i,j,k,l) = A(i,j) \circ B(k,l)$ for $i = 1, \ldots, m$, $j = 1, \ldots, n$, $k = 1, \ldots, p$, and $l = 1, \ldots, q$. The inner product of two tensors leads to a reduction of the total number of dimensions, resulting in a scalar or a lower-dimensional tensor, depending on the dimensions of the tensors involved. As a familiar example, take two matrices $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$, the contraction of A and B results in a 2D tensor $C \in \mathbb{R}^{m \times p}$, where each element is defined as $C(i,j) = \sum_{k=1}^{n} A(i,k)B(k,j)$ for $i = 1, \ldots, m$ and $j = 1, \ldots, p$.

The contraction operation generalizes to higher-dimensional tensors, where the dimensions to be contracted need to be specified. The *matrix-tensor product* is a special case of tensor contraction, where a matrix is multiplied with a tensor along one of its dimensions. For example, for a 3D tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ and a matrix $B \in \mathbb{R}^{n_3 \times m}$, the matrix-tensor product results in a 3D tensor $\mathcal{C} \in \mathbb{R}^{n_1 \times n_2 \times m}$, where each element is defined as $\mathcal{C}(i,j,k) = \sum_{\ell=1}^{n_3} \mathcal{A}(i,j,\ell)B(\ell,k)$ for $i = 1, \ldots, n_1$, $j = 1, \ldots, n_2$, and $k = 1, \ldots, m$.

When we have multiple matrices to contract with a tensor, we can define a *multi-index contraction* operation. 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 $i = 1, \ldots, d$. The multi-index contraction between $\mathcal{A}$ and B_i is defined as[1]

$$\mathcal{C} = \mathcal{A} \times [\![B_1, \ldots, B_d]\!], \quad \text{where} \quad \mathcal{C} = (\cdots(((\mathcal{A} \ {}_1\times_2 \ B_1) \ {}_2\times_2 \ B_2) \ {}_3\times_2 \ \cdots) \ {}_d\times_2 \ B_d), \quad (2.1)$$

[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.

with $\mathcal{C} \in \mathbb{R}^{m_1 \times \cdots \times m_d}$ the resulting d-dimensional tensor. The idea in (2.1) 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 $\mathcal{B}_i \in \mathbb{R}^{m_{1_i} \times m_{2_i} \times n_i}$, (2.1) is denoted as

$$\mathcal{C} = \mathcal{A} \times [\![\mathcal{B}_1, \ldots, \mathcal{B}_d]\!], \quad \text{where} \quad \mathcal{C} = (\cdots(((\mathcal{A} \ {}_1\times_3 \ \mathcal{B}_1) \ {}_2\times_3 \ \mathcal{B}_2) \ {}_3\times_3 \ \cdots) \ {}_d\times_3 \ \mathcal{B}_d), \quad (2.2)$$

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 (2.2) as

$$\mathcal{C} = \mathcal{A} \underset{\mathcal{I}}{\times} [\![\mathcal{B}_{i_1}, \ldots, \mathcal{B}_{i_m}]\!], \quad \text{where} \quad \mathcal{C} = (\cdots(((\mathcal{A} \ {}_{i_1}\times_3 \ \mathcal{B}_{i_1}) \ {}_{i_2}\times_3 \ \mathcal{B}_{i_2}) \ {}_{i_3}\times_3 \ \cdots) \ {}_{i_m}\times_3 \ \mathcal{B}_{i_m}) \quad (2.3)$$

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

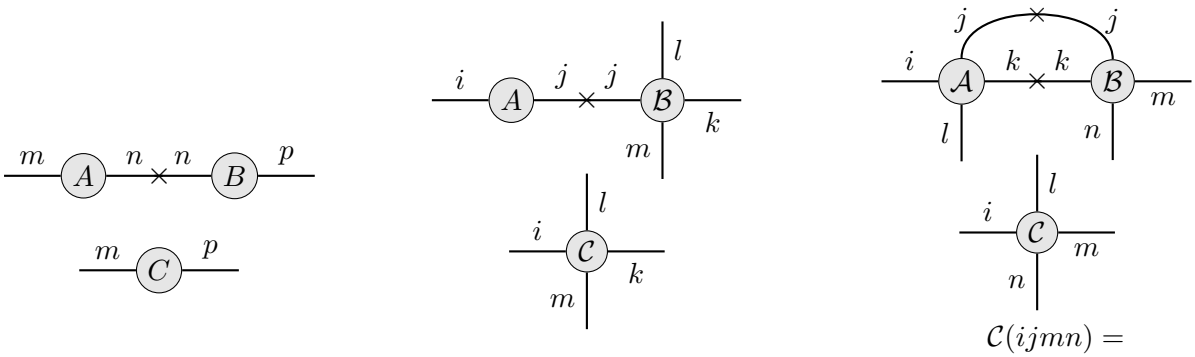
The *contraction* between two tensors $\mathcal{A} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$ and $\mathcal{B} \in \mathbb{R}^{n_d \times n_{d+1} \times \cdots \times n_D}$ along the d-th dimension of $\mathcal{A}$ and the first dimension of $\mathcal{B}$ is a binary operation represented as

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

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.

Using tensor network diagrams contraction of two tensors is denoted by connecting two nodes representing the tensors with edges along the indices that are being contracted. For example Figure 2.2a shows the contraction of two matrices, where the edges corresponding to the second dimension of the first matrix and the first dimension of the second matrix are connected, indicating that these dimensions are contracted. The resulting tensor is a 2D tensor (i.e. a matrix), which is represented by a node with two edges corresponding to its dimensions. Similarly, Figure 2.2b shows the contraction of a matrix with a 4D tensor, where the edges corresponding to the second dimension of the matrix and the first dimension of the tensor are connected, indicating that these dimensions are contracted. The resulting tensor is a 4D tensor, which is represented by a node with four edges corresponding to its dimensions. Finally, Figure 2.2c shows the contraction of two 4D tensors, where two edges corresponding to each tensor are connected, indicating that these dimensions are contracted. The resulting tensor is a 4D tensor, which is represented by a node with two edges corresponding to its

dimensions.



$C(ik) = \sum_j A(ij)B(jk)$

(a) Matrix-matrix

$\mathcal{C}(iklm) = \sum_j A(ij)\mathcal{B}(jklm)$

(b) Tensor-matrix

$\mathcal{C}(ijmn) = \sum_{k,l} \mathcal{A}(ijkl)\mathcal{B}(klmn)$

(c) Tensor-tensor

Figure 2.2: Tensor network diagrams depicting various tensor contractions. The edges connecting the nodes represent the dimensions that are being contracted. The resulting tensor is represented by a node with edges corresponding to its new dimensions.

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

$$\mathcal{C} = \mathcal{A} \oplus^k \mathcal{B}, \quad \text{where} \quad \mathcal{C}\left(i_1, \ldots, i_k, \ldots, i_d\right) = \begin{cases} \mathcal{A}\left(i_1, \ldots, i_k, \ldots, i_k\right) & \text{if } i_k \leq n_k \\ \mathcal{B}\left(i_1, \ldots, (i_k - n_k), \ldots, i_k\right) & \text{otherwise} \end{cases}. \quad (2.4)$$

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*.

As streaming data plays a central role for this thesis, we need to define the necessary notation to work with tensors in a streaming fashion. Unless otherwise noted, we reserve the superscripts to denote the index of tensors in a stream for the remainder of this thesis. An N-batch of d-dimensional tensors is a collection of N tensors, each of which is d-dimensional. We denote an N-batch of d-dimensional tensors as $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$.

A particularly useful concept in the context of this thesis is the notion of a *tensor stream*. Parallel to the definitions in [7, 6], we define a *tensor stream* (or alternatively, *stream of tensors*), as a sequence of N^k-batch 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 $\mathbf{n}^k = \sum_{i=1}^{k} N^i$.

2.2 Tensor Decomposition Formats

When tensors have additional structure, it may be possible to effectively represent them in a more compact format. Tensor decompositions allow us to exploit the structure of tensors to represent them in a more efficient way, reducing the storage and computational resources required to work with them. This makes them a particularly attractive tool to beat the curse of dimensionality arising from high-dimensional data.

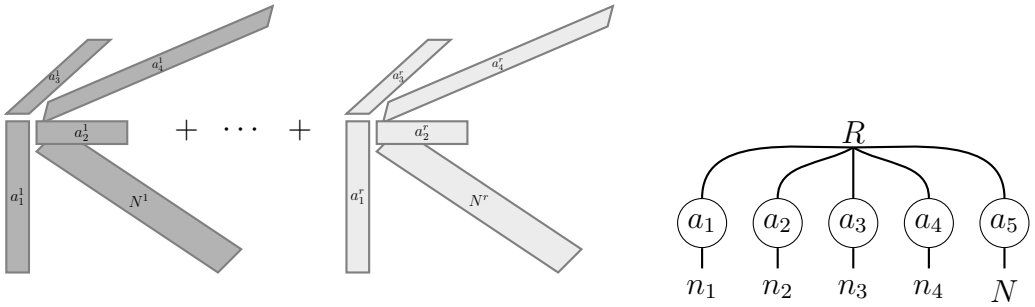
From the three tensor decomposition families mentioned in Section 1.2.4, the most common tensor decomposition formats are the CANDECOMP/PARAFAC (CP), Tucker, and Tensor Train (TT) formats. Additionally, we will also discuss the Hierarchical Tucker (HT) and introduce the Batch-HT format for the purposes of this thesis. Each of these formats has its own advantages and disadvantages, and the choice of format depends on the specific application and the properties of the data being analyzed.

While presenting the individual tensor decomposition formats, we will also discuss the underlying mathematical principles and algorithms used to compute these decompositions. Some examples to the well known algorithms are the Alternating Least Squares (**A**lternating **L**east **S**quares (ALS)) [41, 95], **H**igher-**O**rder **S**ingular **V**alue **D**ecomposition (HOSVD) [68], and the **T**ensor **T**rain **S**ingular **V**alue **D**ecomposition (TT-SVD) [191] algorithms. At the end of this section we will also summarize and highlight the differences between the formats and their suitability for different types of data and applications.

This section is intended to provide a soft introduction to the tensor decomposition formats and their properties. We will not go into the mechanistic details of the algorithms used to compute these decompositions, as they are well-documented in the literature. Instead, we will focus on the key concepts and properties of each format, and summarize their respective strengths and weaknesses. We kindly refer the curious reader to the excellent reviews by Kolda and Bader [137], Sidiropoulos et al. [230], and Cichocki [60] for a more in-depth discussion on tensor decompositions and their applications.

2.2.1 CANDECOMP/PARAFAC (CP) Format

The CANDECOMP/PARAFAC (CP) format is a widely used canonical tensor decomposition format that represents a tensor as a sum of rank-one tensors. The idea of representing a tensor as a sum of rank-one tensors was first introduced by Hitchcock in the 1920s [104, 105] and again in the 1940s by Cattell [42, 43]. Finally in the 1970s Carroll [41] proposed the same concept under the name CANDECOMP (standing for *CANonical DECOMPosition*) and Harshman [95] proposed the concept under the name PARAFAC (standing for *PARAllel FACtor analysis*). The two names were used interchangeably until the 1990s, then in the



(a) Pictorial representation (b) Tensor network diagram

Figure 2.3: Visualizations for the CP format

2000s Kiers [130] proposed the name CP to unify the two concepts. Approximation of a d-dimensional tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ in the CP format is defined as

$$\mathcal{X} \approx \sum_{r=1}^{R} a_1^r \circ a_2^r \circ \cdots \circ a_d^r, \tag{2.5}$$

where $a_i^r \in \mathbb{R}^{n_i}$ are the r-th factor vector for the i-th dimension of the tensor, and R is the rank of the approximation with typically $R \ll n_i$. Elementwise, Equation (2.5) can be expressed as

$$\mathcal{X}(i_1, i_2, \ldots, i_d) \approx \sum_{r=1}^{R} a_1^r(i_1) a_2^r(i_2) \cdots a_d^r(i_d), \tag{2.6}$$

where $i_j = 1, \ldots, n_j$ for all $j = 1, \ldots, d$.

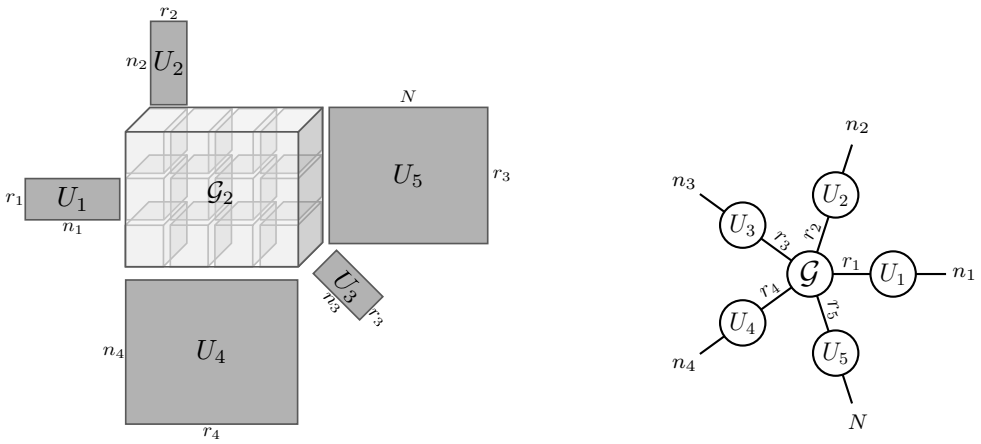
Analogous to the matrix case, the exact rank of a tensor (or alternatively the *tensor rank*) is defined as the smallest number of rank-one tensors needed to represent the tensor [104, 144]. In other words, it is the smallest R such that Equations (2.5) and (2.6) become an equality. It is an important property of tensors and plays a crucial role in determining the effectiveness of the CP decomposition. The actual tensor rank can be much larger than the approximation rank R used in the CP decomposition, especially for high-dimensional tensors.

Unlike the matrix rank, which is well-defined and can be computed efficiently, computing the tensor rank is an NP-hard problem [100, 97]. Therefore, it is usually estimated using various heuristics [36] or algorithms [35, 56]. In some cases, the rank R is chosen manually (or on a trial-and-error base) depending on the desired level of approximation or the specific application requirements.

The CP representation reduces the storage from $\mathcal{O}(n^d)$ to $\mathcal{O}(dnR)$, where $n = \max(n_1, n_2, \ldots, n_d)$ is the maximum dimension of the tensor and R is the rank of the approximation. The CP decomposition of a tensor can be computed using various algorithms,

including `ALS` [41, 95], `CPRAND` [25], `SPALS` [55], and `PMF3` [194]. The CP decomposition becomes particularly attractive when the rank R is much smaller than the dimensions of the tensor, as it results in a significant reduction in the number of entries required to represent the tensor. Another advantage of the CP decomposition is that it can result in an interpretable decomposition, making it easier to draw insights from the data. This is particularly useful in applications such as chemometrics [183], biochemistry [122], and cancer risk assessment [27], where the data can be interpreted in terms of underlying factors or latent variables.

2.2.2 Tucker Format



(a) Pictorial representation (b) Tensor network diagram

Figure 2.4: Visualizations for the Tucker format

The Tucker format can be seen as the higher dimensional generalization of the matrix SVD. It represents a tensor as a product of a smaller core tensor and a set of factor matrices, each corresponding to one dimension of the tensor. Tucker format was first introduced by Tucker in the 1960s [259, 261, 260] under the name *multilinear factor analysis* and later introduced as *three-mode PCA* by Kroonenberg and De Leeuw [143] in the 1980s, and finally as *HOSVD* by De Lathauwer et al. [68] in the 2000s. The Tucker decomposition of a d-dimensional tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is defined as

$$\mathcal{X} \approx \mathcal{G} \ {}_1\times_2 \ A_1 \ {}_2\times_2 \ A_2 \ {}_3\times_2 \ \cdots \ {}_d\times_2 \ A_d, \tag{2.7}$$

where $\mathcal{G} \in \mathbb{R}^{r_1 \times r_2 \times \cdots \times r_d}$ is the core tensor, $A_i \in \mathbb{R}^{n_i \times r_i}$ is the factor matrix for the i-th dimension of the tensor, and r_i is the rank of the approximation along the i-th dimension.

Alternatively, we can express the Tucker decomposition in a more compact form using the multi-index contraction notation from Equation (2.1) as

$$\mathcal{X} \approx \mathcal{G} \times [\![A_1, A_2, \ldots, A_d]\!]. \tag{2.8}$$

Elementwise, Equation (2.7) can be expressed as

$$\mathcal{X}(i_1, i_2, \ldots, i_d) \approx \sum_{j_1=1}^{r_1} \sum_{j_2=1}^{r_2} \cdots \sum_{j_d=1}^{r_d} \mathcal{G}(j_1, j_2, \ldots, j_d) A_1(i_1, j_1) A_2(i_2, j_2) \cdots A_d(i_d, j_d), \tag{2.9}$$

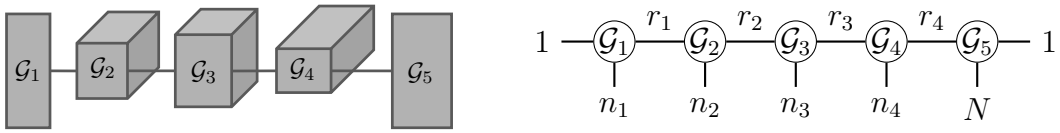
where $i_j = 1, \ldots, n_j$ for all $j = 1, \ldots, d$ and $j_i = 1, \ldots, r_i$ for all $i = 1, \ldots, d$. Note that the Tucker ranks r_i are not necessarily the same for all dimensions and can be chosen independently based on the structure of the tensor and the desired level of approximation. Furthermore, please note that the Tucker ranks are different from the tensor rank R defined in Section 2.2.1. It is noteworthy that when $r_i = R$ for all $i = 1, \ldots, d$, and for a superdiagonal core tensor $\mathcal{G}$, the Tucker decomposition becomes equivalent to the CP decomposition in Equation (2.5). However, the flexibility of choosing individual r_is result in much smaller Tucker ranks than the tensor rank R.

The Tucker representation reduces the storage from $\mathcal{O}\left(n^d\right)$ to $\mathcal{O}\left(dnr + r^d\right)$, where $r = \max(r_1, \ldots, r_d)$. Note that 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 [68, 260]. In literature, there are a plethora of efficient methods for computing the Tucker representation of a tensor including direct [266, 140], randomized [47, 140, 258, 299, 302, 179, 248], and iterative [69, 143, 280, 45, 140, 258, 80] algorithms.

Tucker format is used in a wide range of applications, including scientific data compression [138], signal processing [70], and image processing [129]. Specifically for scientific applications, Tucker format is shown to be very effective for achieving extreme levels of compression while maintaining a high level of accuracy in approximating the original tensor [138, 66]. As it is, the Tucker decomposition is not unique when computed with HOSVD. As long as one performs the inverse of the rotation on the core tensor, and the factor matrices, the same tensor can be represented in multiple ways. To address this issue, one can impose additional constraints on the factor matrices, such as orthogonality or non-negativity [218]. This can help to ensure that the decomposition is unique and interpretable. However, this may come at the cost of increased computational complexity and reduced flexibility in the decomposition. Finally, even though the Tucker format results in a much smaller core tensor $\mathcal{G}$, the storage still grows exponentially with the number of dimensions d and the ranks r_i,

which can be a limitation for very high-dimensional tensors.

2.2.3 Tensor-Train (TT) Format



(a) Pictorial representation (b) Tensor network diagram

Figure 2.5: Visualizations for the TT format

The Tensor-Train (TT) format is a tensor decomposition format that represents a tensor as a product of 3D tensors arranged in a chain-like structure. This structure also often leads to interpretations of the TT-format as a *linear tensor network*. The TT-format was first introduced by Oseledets in 2011 [191] and has since become a popular choice for representing high-dimensional tensors due to its efficiency and flexibility. It is also known as the ***M**atrix **P**roduct **S**tate (MPS)* representation in the physics literature [201, 269, 189, 223, 3]. The TT representation of a d-dimensional tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is defined as

$$\mathcal{X} = \mathcal{G}_1 \ {}_3\times_1 \ \mathcal{G}_2 \ {}_4\times_1 \ \cdots \ {}_{d+2}\times_1 \ \mathcal{G}_d, \tag{2.10}$$

where $\mathcal{G}_i \in \mathbb{R}^{\mathbf{r}_{i-1} \times n_i \times \mathbf{r}_i}$ is the i-th *TT-core*, and $\mathbf{r}_i$ is the *TT-rank* of the i-th core tensor with $\mathbf{r}_0 = \mathbf{r}_d = 1$. Theorem 2.1 of Oseledets et al. [191] shows that the TT-ranks $\mathbf{r}_i$ are equal to the rank of the corresponding unfolding of the tensor $\mathcal{X}$, i.e., $\mathbf{r}_i = \operatorname{rank}(X_{(i)^*})$, where $X_{(i)^*}$ is a special unfolding of $\mathcal{X}$, which is obtained by simply reshaping $\mathcal{X}$ as

$$X_{(i)^*} = \texttt{reshape}\left(\mathcal{X}, \prod_{j=1}^{i} n_j, \prod_{j=i+1}^{d} n_j\right). \tag{2.11}$$

Elementwise, Equation (2.10) can be expressed as

$$\mathcal{X}(i_1, i_2, \ldots, i_d) = \sum_{j_0=1}^{r_0} \sum_{j_1=1}^{r_1} \cdots \sum_{j_{d-1}=1}^{r_{d-1}} \mathcal{G}_1(j_0, i_1, j_1)\mathcal{G}_2(j_1, i_2, j_2) \cdots \mathcal{G}_d(j_{d-1}, i_d, j_d), \tag{2.12}$$

where $i_j = 1, \ldots, n_j$ for all $j = 1, \ldots, d$ and $j_i = 1, \ldots, r_{i-1}$ for all $i = 1, \ldots, d$. The TT representation reduces the storage from $\mathcal{O}\left(n^d\right)$ to $\mathcal{O}\left(dnr^2\right)$, where r is the maximum TT-rank.

Note that Equations (2.10) and (2.12) have an equality for the case of exact TT representation, but for the purposes of this thesis, we will consider an *approximate* TT representation where the tensor $\mathcal{X}$ is approximated by a tensor in TT format. Therefore, with a slight abuse of notation, we will refer to the *ranks of the approximate representation* as the TT-ranks r_i.

One of the most popular methods for computing the TT-decomposition is the TT-SVD algorithm [191], which is a sequential algorithm that computes the TT representation by iteratively reshaping the tensor and performing SVD on the resulting matrices. Some other noteworthy algorithms for computing the TT-decomposition include the `TT-cross` algorithm [190], the `TT-HSVD` algorithm [304], and the `TT-MALS` algorithm [107].

Once a TT is computed, basic algebraic operations, such as addition and multiplication, can be computed in closed form. For example, two d-way tensors in TT-format with cores $\{\mathcal{G}_i\}_{i=1}^d$ and $\{\mathcal{H}_i\}_{i=1}^d$ can be added if both of the tensors have the size $n_1 \times \cdots \times n_d$. The resulting tensor $\mathcal{D} = \mathcal{G} + \mathcal{H}$ is calculated using the following rule

$$\mathcal{D}_m(j_m) = \begin{cases} \begin{bmatrix} \mathcal{G}_1(j_1) & \mathcal{H}_1(j_1) \end{bmatrix} & , \ m = 1; \\ \begin{bmatrix} \mathcal{G}_m(j_m) & 0 \\ 0 & \mathcal{H}_m(j_m) \end{bmatrix} & , \ m = 2, \dots d-1; \\ \begin{bmatrix} \mathcal{G}_d(j_d) \\ \mathcal{H}_d(j_d) \end{bmatrix} & , \ m = d, \end{cases} \tag{2.13}$$

where $\mathcal{G}_m(j_m) \in \mathbb{R}^{r_{m-1} \times r_m}$ denotes the j_m-th slice of the m-th TT-core $\mathcal{G}_m$. If $\mathcal{G}$ has TT-ranks r_i and $\mathcal{H}$ has TT-ranks $\bar{r}_i$, the sum $\mathcal{D}$ will have the TT-ranks $\hat{r}_i = r_i + \bar{r}_i$ with the exception of $\hat{r}_0 = \hat{r}_d = 1$.

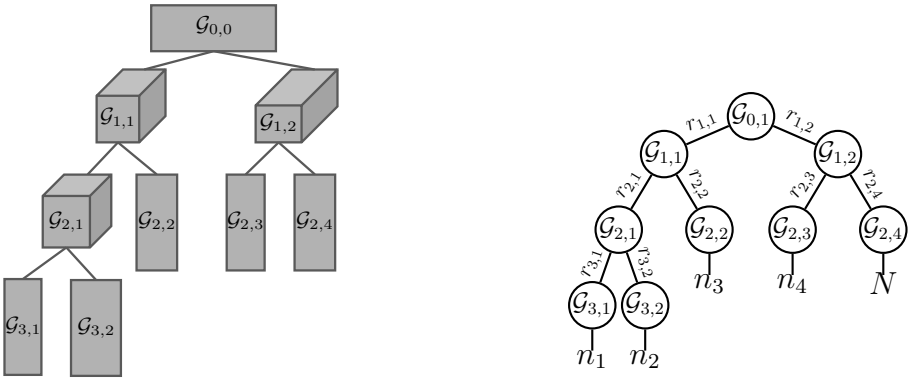
The TT format is particularly well-suited for high-dimensional tensors, as it can efficiently represent tensors with a large number of dimensions while maintaining a low rank. It has been successfully applied in various fields, including machine learning [293], computer vision [72], signal processing [289], and scientific computing [74, 65].

TT-format is attractive not only because it breaks the curse of dimensionality and is one of the simplest network-based tensor decomposition formats, but also it enables performing efficient tensor operations such as addition, multiplication, and contraction in a computationally efficient manner. This property of TT makes it particularly appealing for building scalable solvers for high-dimensional problems in scientific computing [74, 87, 88].

One of the main drawbacks of the TT-format is that it can be sensitive to the choice of TT-ranks, and the resulting decomposition may not always be unique. Additionally, the TT-representation may not always provide an interpretable decomposition, making it challenging

to draw insights from the data. Finally, the TT-format can be computationally expensive to compute for very high-dimensional tensors, especially when the TT-ranks are large. This can limit its applicability in some cases, particularly when dealing data with highly intricate features.

2.2.4 Hierarchical Tucker (HT) Format



(a) Pictorial representation (b) Tensor network diagram

Figure 2.6: Visualizations for the HT format

HT-format is a hierarchical tensor decomposition format that represents a tensor with a combination of 2D and 3D tensors, which are ordered according to a tree-like contraction scheme. It is first proposed by Oseledets and Tyrtyshnikov in 2009 [192] under the name *tree Tucker* to break the curse of dimensionality for high-dimensional tensors, and later introduced as *hierarchical Tucker* by Grasedyck in 2010 [90]. The HT-format can be seen as a generalization of the Tucker format, where the core tensor is recursively reshaped and decomposed into smaller tensors, resulting in a hierarchical structure that allows for efficient representation of high-dimensional tensors.

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 $\mathcal{G}$ in Equation (2.7) is reshaped into a tensor with $d/2$ dimensions and a decomposition of this core is created, recursively.

The HT representation of a d-dimensional tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ is defined through two components: a *dimension tree* $\mathbf{T}$ and a set of *HT-cores* $\boldsymbol{\mathcal{G}}$. The dimension tree specifies the contraction ordering of the HT-cores, while the HT-cores are the tensors that contain the low-rank approximation of the tensor.

Specifically, the dimension tree $\mathbf{T}$ is a connected graph of $2d - 1$ nodes, which 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.

As the HT-format introduces a hierarchy among dimensions, the dimension tree $\mathbf{T}$ plays a crucial role in determining the relationships between dimensions and how they interact with each other. Figure 2.7 illustrates various configurations of dimension trees for a five-dimensional tensor. Each configuration represents a different way of grouping dimensions, and therefore, results in a different compression performance.

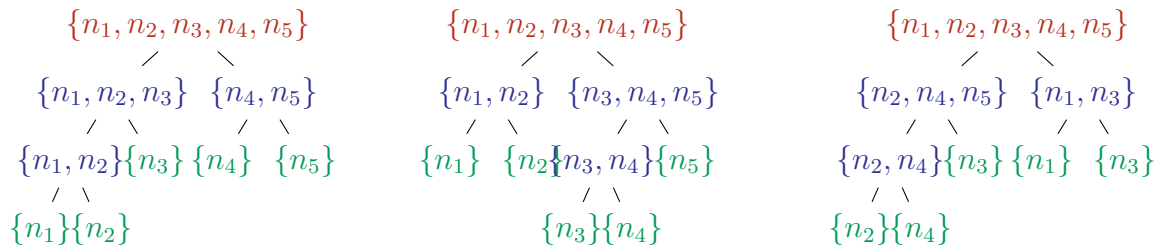


Figure 2.7: Three possible configurations of the dimensions for a five dimensional tensor with shape $n_1 \times n_2 \times n_3 \times n_4 \times n_5$. Each dimension tree describes a different interaction between dimensions and therefore yields different compression performance. For a more detailed study on the effect of the axis reordering, please refer to Appendix B.1.2. Root, transfer, and leaf nodes are colored in red, blue, and green, respectively.

As HT-format is obtained by performing recursive decompositions of the original tensor $\mathcal{X}$, it is not straightforward to express the HT-representation in a single equation. To obtain the full tensor we can 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 $\mathcal{X}$ from its HT-format as

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

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*.

The HT-format reduces the original $\mathcal{O}(dnR+R^d)$ storage complexity of the Tucker format

to $\mathcal{O}(dnr + dr^3)$, where R is the maximum Tucker rank and r is the maximum HT-rank. There are several algorithms for computing the HT decomposition of a tensor, including the root-to-leaves algorithm [90, 141], and the leaves-to-root algorithm [141]. The HT-format excels in representing high-dimensional tensors with a low rank, and it is particularly well-suited for applications where the tensor has a hierarchical structure or where the dimensions can be naturally grouped into subproblems. It has been successfully applied in various fields, including machine learning, computer vision, signal processing, and scientific computing [90, 141, 6]. One of the biggest bottlenecks of the HT-format is the need for auxiliary structures such as the dimension tree, which increases the complexity of developing and implementing algorithms to compute a representation in this format.

2.2.5 Batch-HT (BHT) Format



(a) Pictorial representation (b) Tensor network diagram

Figure 2.8: Visualizations for the batch-HT format

Batch-hierarchical Tucker format builds onto the foundation laid by the HT-format and was proposed in [6] by Aksoy and Gorodetsky in 2024 to address the challenges of compressing a batch of tensors in HT format. The BHT format is designed to efficiently represent a batch of tensors while maintaining the hierarchical structure and properties of the HT format. The key idea behind the BHT format is to treat the batch dimension as an additional dimension in the tensor representation, allowing for efficient compression and representation of a batch 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[2], as shown in Figure 2.6. Assuming that the individual tensors in a batch are similar, i.e., low rank within the batch, such an

[2]or the $(p-1)$-th layer, depending on the number of dimensions of the tensor or the order of the batch dimension

approach can lead to an inefficient/inaccurate representation. As an alternative to adding a new leaf, the batch-HT-format proposes grouping the batch dimension into the root, as shown in Figure 2.8. To do this, the batch-HT-format excludes the batch dimension from the dimension tree during construction as if a single tensor from the batch is being decomposed.

Specifically, for an N-batch of d-dimensional tensors $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$, batch-HT-format uses 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.

Another motivation behind the batch-HT-format is to allow performing incremental updates to the `HT` representation of a batch of tensors. As the batch dimension N would be treated as any other dimension, the HT-representation is not suitable for incremental updates. In addition to offering a more efficient representation of a batch of tensors, the batch-HT-format also allows for incremental updates to the `HT` representation, which becomes particularly handy when dealing with large datasets or streaming data. The only algorithm for computing the batch-HT-decomposition is the BHT-l2r algorithm [6], which is a leaves-to-root decomposition algorithm that computes the batch-HT representation starting from the leaves and moving towards the root.

2.2.6 Other Tensor Decomposition Formats

There are several other tensor decomposition formats that have been proposed in the literature, each with its own advantages and disadvantages. Some of these formats include:

- Tensor Ring [300]: A format that represents a tensor as a circular network of low-rank tensors, similar to a TT format, but with a circular structure that allows for efficient representation of periodic tensors.

- **M**ulti-scale **E**ntanglement **R**enormalization **A**nsatz (MERA) [270]: A format that represents a tensor as a hierarchical structure, allowing for efficient representation of quantum states in many-body systems.

- **P**rojected **E**ntangled **P**air **S**tates (PEPS) [269, 268]: A format that represents a tensor as a network of entangled pairs, allowing for efficient representation of quantum states in two-dimensional systems.

- Tensor wheel [284]: A format that represents a tensor as a network with a wheel-like structure, with a central core tensor and surrounding tensors connected in a circular manner.
- Tensor star [303]: A format that represents a tensor as a network of 3D and 4D tensors, which are arranged in a star-like structure.

These formats are less commonly used than the CP, Tucker, and Tensor Train formats, but they can be useful in specific applications where their unique properties provide advantages over the more traditional formats.

2.2.7 Summary of Tensor Decomposition Formats

As it is stated earlier, there does not exist a universally optimal low-rank format, and the choice of format depends on the specific application and the properties of the data. Each format has its own strengths and weaknesses, and the choice of format should be made based on the specific requirements of the problem at hand. This section serves as a summary of the presented tensor decomposition formats, highlighting their key characteristics and weaknesses.

One of the most important aspects of tensor decomposition formats is their storage complexity. Table 2.1 summarizes the storage complexity of each format, which is a crucial factor in determining the feasibility of using a particular format for a given problem.

Table 2.1: Comparison of storage complexity of tensor decomposition formats for a d-dimensional tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$. The storage complexity is expressed in terms of the number of dimensions d, the size of each dimension n, and the rank r of the respective decomposition. Please note that each format requires a different r to approximate the same tensor at the same accuracy.

Format	**Storage**
CP	$\mathcal{O}(dnr)$
Tucker	$\mathcal{O}(dnr + r^d)$
TT	$\mathcal{O}(dnr^2)$
HT	$\mathcal{O}(dnr + dr^3)$
batch-HT	$\mathcal{O}(dnr + dr^3)$

In addition to storage complexity, it is also important to consider the strengths and weaknesses of each format. Table 2.2 provides a brief overview of the strengths and weaknesses of each format, highlighting their key characteristics and limitations.

Table 2.2: Detailed comparison of tensor decomposition formats highlighting their respective strengths and weaknesses

Format	**Strengths**	**Weaknesses**
CP	• Simple and interpretable components • Most efficient storage complexity • Useful for finding latent factors	• Computing tensor rank is NP-hard • May require high rank for good approximation • Best rank-R approximation might not exist
Tucker	• More flexible than CP • Easy to compute via HOSVD • Effective for data compression	• Storage complexity exponential in dimensions • Less interpretable than CP • Core tensor can be difficult to analyze
TT	• Linear storage complexity in dimensions • Enables efficient tensor operations • Well-suited for scientific computing	• Sensitive to dimension ordering • Less interpretable components • Performance depends on TT-ranks
HT	• Linear storage complexity in dimensions • Efficient for high-dimensional tensors • More expressive than TT	• Higher implementation complexity (i.e. dimension tree) • Sensitive to dimension ordering • Less efficient for tensors with simple structure
batch-HT	• Efficient for batches of similar tensors • Supports incremental updates • Inherits benefits of the HT format	• Inherits complexity of the HT format • Less effective if tensors in batch are dissimilar • Relatively new with fewer established applications

2.3 Incremental Tensor Decomposition Algorithms

Recall that the goal of this thesis is to develop efficient incremental low-rank tensor decomposition algorithms that can handle large-scale high-dimensional data streams. This section presents some of the existing incremental tensor decomposition algorithms that are relevant to this thesis.

2.3.1 CP Format

As this thesis focuses mainly on incremental tensor decomposition algorithms in the TT- and HT-format, we will only briefly discuss the existing incremental tensor decomposition algorithms in the CP format. Some notable algorithms for incremental CP decomposition are the `iCP-AM` [298] and *CP-stream* [232], `BICP` [113], `SamBaTen` [91] algorithms. These algorithms mostly use optimization based approaches to update the CP representation of a tensor when a new tensor becomes available.

2.3.2 Tucker Format

Similar to the CP format, Tucker format does not exactly fall within the scope of this thesis, but it is important to mention some of the existing incremental algorithms for Tucker decomposition for completeness. In literature, there are various ways to compute the Tucker decomposition of multidimensional arrays incrementally.

We can summarize existing incremental algorithms under the following 3 categories:

- **Optimization based:** These algorithms formulate updating the Tucker decomposition as an optimization problem and solve it iteratively. The optimization problem is typically a least squares problem that minimizes the difference between the original tensor and the Tucker approximation. `Tucker-TS` algorithm from [169] and "One-pass sketch and low rank recovery" algorithm from [248] are examples of optimization based algorithms. These algorithms also employ randomized matrix sketching to reduce the problem size and computational cost. However, these algorithms are not suitable for infinite streams.

- **Numerical linear algebra based:** These algorithms compute the Tucker decomposition by utilizing numerical linear algebra techniques. The Tucker decomposition is computed by performing a series of matrix factorizations and contractions. The `ST-HOSVD` algorithm from [66], `I-TSvd` algorithm from [145], the `D-TuckerO` algorithm

from [124], the STA algorithm from [247], and the "low-rank updates to Tucker decomposition algorithm" from [24] are examples of numerical linear algebra based algorithms. These algorithms mostly require generating auxiliary matrices either to compute the approximation for the current step or to incorporate the history of the stream while computing an approximation for the current step. [145] and [124] uses fixed rank approximation.

- **Other:** There are algorithms in the literature that tackle the incremental decomposition problem differently. These algorithms do not fit into the above categories. For example, the incrementalHoSVD algorithm from [234], the D-L1-Tucker algorithm from [44], and both variants of WTA algorithm from [246] focus on computing an approximation for a sliding window of the stream. The method proposed in [112] focuses solely on tracking the subspace projections of the unfoldings of the accumulation for 3-dimensional streams. The STenSr algorithm from [229] uses incremental SVD to track the orthogonal bases just for the temporal dimension and focuses on identifying anomalous tensors in the stream. In contrast to all algorithms mentioned above, the eOTD algorithm from [285] increments the Tucker decomposition of a given tensor along *all* modes of the tensor.

2.3.3 Tensor Train Format

As this thesis proposes an incremental algorithm for updating an approximation of tensors in TT-format, we will have a much deeper look into the algorithms that are already available in the literature for this purpose. From the existing literature, we will focus on two algorithms that are relevant to the problem we are trying to solve in this thesis. These algorithms are the **T**ensor **T**rain **F**irst **O**rder **A**pproximation (TT-FOA) [1] and **I**ncremental **T**ensor **T**rain **D**ecomposition (ITTD) [161]. The TT-FOA algorithm is an optimization-based approach that updates the TT-cores of the existing accumulation tensor, while the ITTD algorithm performs separate computations of TT-SVD for each new tensor and performs recompression of the TT-cores. One of the other prominent algorithms for computing an approximation in TT-format from a data stream is the STTA algorithm [142] from Kressner et al. in 2023. However, as it uses sketching operators, it does not fall exactly under the category of incremental algorithms that we consider in this thesis.

This section investigates the existing approaches for updating an existing representation of a tensor in the TT format when a new tensors become available. The algorithms we consider in this section are the TT-FOA [1] and ITTD [161]. The core difference between these two approaches is that TT-FOA assumes fixed TT-ranks and uses an optimization-

based approach to update the TT-cores, while ITTD performs separate computations of `TT-SVD` for each new tensor and performs recompression of the TT-cores.

The update procedure for incremental TT-decomposition algorithms can be viewed as a two-step process. The first step is to preprocess the newly arrived tensor $\mathcal{Y}^{k+1} \in \mathbb{R}^{n_1 \times \cdots \times n_d}$, and the second step is to update the TT-cores of the previous accumulation tensor $\mathcal{X}^k$ to obtain new TT-cores for $\mathcal{X}^{k+1}$. Below we describe how TT-FOA and ITTD handle these two steps.

Preprocessing $\mathcal{Y}^{k+1}$ At time $k+1$, TT-FOA poses the problem of preprocessing the new tensor $\mathcal{Y}^{k+1}$ using the TT-cores of the approximate accumulation $\hat{\mathcal{X}}^k$ as the following regularized optimization problem

$$g^{k+1} = \underset{g \in \mathbb{R}^{r_d \times 1}}{\operatorname{argmin}} \|\mathcal{Y}^{k+1} - \hat{\mathcal{X}}^k \ {}_{(d+1)}\times_1 \ g\|_F^2 + \frac{\rho}{2}\|g\|_2^2, \tag{2.14}$$

where g^{k+1} is the representation of $\mathcal{Y}^{k+1}$ as the $(d+1)$-th TT-core of the accumulation $\mathcal{X}^k$, ρ is a small regularization parameter, and the contraction of first d TT-cores $\hat{\mathcal{X}}^k \in \mathbb{R}^{n_1 \times \cdots \times n_d \times r_d}$ is computed as

$$\hat{\mathcal{X}}^k = \mathcal{G}_1 \ {}_3\times_1 \ \cdots \ {}_3\times_1 \ \mathcal{G}_d. \tag{2.15}$$

Note that these first d cores are common for all compressed data, and only the final core is unique to a particular datum. In other words, one can view the first d cores as the basis of approximation that is common to all the data and the final core as the coefficients of this basis that are specific to each datum. Without the regularization term, the optimal solution g^{k+1} would be the projection of $\mathcal{Y}^{k+1}$ onto the basis defined by $\hat{\mathcal{X}}^k$. Thus, any component of the new data not represented by the existing accumulation would be discarded. The regularization biases the solution to be closer to zero.

In [1], the optimization problem is solved using the randomized sketching technique [168]. The closed-form solution is

$$g^{k+1} = \left(\mathcal{L}(\hat{X}^k_{(d)})^T \mathcal{L}(\hat{X}^k_{(d)}) + \rho I_{r_d}\right)^{-1} \mathcal{L}(\hat{X}^k_{(d)})^T \mathcal{L}(y^{k+1}), \tag{2.16}$$

with $\hat{X}^k_{(d)} \in \mathbb{R}n_1 \ldots n_d \times r_d$ as the mode-d unfolding of $\hat{\mathcal{X}}^k$, $y^{k+1} \in \mathbb{R}n_1 \ldots n_d$ as the reshaping of $\mathcal{Y}^{k+1}$ into a vector, and $\mathcal{L}(\cdot)$ as the sketching map that samples the same set of rows from $\hat{X}^k_{(d)}$ and y^{k+1}.

ITTD follows another approach. It reshapes the streamed d-way tensor $\mathcal{Y}^{k+1}$ to be a $d+1$-way tensor with size $n_1 \times \cdots \times n_d \times 1$ and computes the TT-decomposition of the reshaped tensor using the TT-SVD algorithm, without leveraging any information from the

accumulation tensor. As a result, it requires the computation of a new sequence of singular value decompositions to discover an entirely new basis — a computation step not needed by TT-FOA or our proposed approach.

Updating the TT-approximation of $\mathcal{X}^k$ to that of $\mathcal{X}^{k+1}$ TT-FOA assumes fixed TT-ranks and adopts a gradient descent procedure to update the TT-cores of $\mathcal{X}^k$. With this fixed-rank assumption, TT-FOA cannot guarantee a predefined upper bound for representation error except in the limit. TT-FOA uses the following optimization objective to update the TT-cores of $\mathcal{X}^k$

$$\mathcal{G}_i = \underset{\mathcal{G}\in\mathbb{R}r_{i-1}\times n_i\times r_i}{\operatorname{argmin}} \sum_{j=1}^{k+1} \lambda^{k+1-j} \|\mathcal{Y}^j - \mathcal{A}_i \ {}_{(i+1)}\times_1 \ \mathcal{G} \ {}_{(i+2)}\times_1 \ \mathcal{B}_i^j\|_F^2, \tag{2.17}$$

where

$$\begin{aligned} \mathcal{A}_i &= \mathcal{G}_1 \ {}_3\times_1 \ \cdots \ {}_3\times_1 \ \mathcal{G}_{i-1}, \\ \mathcal{B}_i^j &= \mathcal{G}_{i+1} \ {}_3\times_1 \ \cdots \ {}_3\times_1 \ core_d \ {}_3\times_1 \ g^j, \end{aligned} \tag{2.18}$$

and $\lambda \in (0, 1]$ is the forgetting factor discounting the effect of previous tensors. Note that only the superscript of the forgetting factor λ is for exponentiation. g^j denotes the coefficient vector representing $\mathcal{Y}^j$ in the $(d+1)$-th TT-core. Note that the objective function updates the i-th core $\mathcal{G}_i$ independent from the other cores and allows updating the cores in parallel. The TT-FOA algorithm provides a recursive approach to efficiently update this core, and we leave further details to [1].

Next, we describe how ITTD updates the accumulated tensor. Recall that at ITTD, this update is preceded by reshaping the d-way tensor $\mathcal{Y}^{k+1}$ into a $d+1$-way tensor with the $(d+1)$-th dimension as 1 and calculating an independent TT-decomposition for $\mathcal{Y}^{k+1}$. ITTD then merges the newly compressed tensor with the existing accumulation through a specific addition operation. To this end, the TT-core of $\mathcal{Y}^{k+1}$ corresponding to the growing dimension is padded with zeros so that the growing dimension becomes $(k+1)$. The same padding procedure is repeated for the TT-core of $\mathcal{X}^k$ so that the growing dimension becomes $(k+1)$. This zero-padding process is done to ensure the dimensional consistency between the two TT-representations. While padding zeros to the TT-cores, the ITTD algorithm places the padding tensors so that both tensors can be added without interfering with each other. ITTD algorithm then combines the TT-cores of the streamed tensor $\mathcal{Y}^{k+1}$ and the existing TT-cores of $\mathcal{X}^k$ by adding them in TT-format using the rule shown in (2.13). After the addition of these two tensors in TT-format, the TT-ranks are the sum of the component ranks. Therefore, an optional final step of the ITTD algorithm, the TT-rounding procedure

of [191, Alg 2] is executed to reorthogonalize and recompress the TT-cores after addition.

2.3.4 Hierarchical Tucker Format

As discussed in Section 2.2.4, the HT-format is not suitable for incremental updates as is. The HT-format is designed to treat the batch dimension (or the accumulation dimension) as any other dimension of the tensor and therefore the batch dimension becomes a leaf on the hierarchical tree. To the best of our knowledge, there exists no incremental tensor decomposition algorithm for the HT-format that can handle a batch of tensors or an accumulation of tensors, which underlines the importance of the batch-HT-format that will be introduced in Section 3.2 and the HT-RISE algorithm that will be introduced in Section 3.3.

2.4 Summary

This chapter provided an overview of tensor decomposition formats and incremental tensor decomposition algorithms. We discussed the most commonly used tensor decomposition formats, including CP, Tucker, Tensor Train, Hierarchical Tucker, and Batch-Hierarchical Tucker formats. Each format has its own strengths and weaknesses, and the choice of format depends on the specific application and the properties of the data.

In the next chapter, we will present the central contribution of this thesis, the TT-ICE algorithm, the HT-RISE algorithm, and the BHT-l2r algorithm.

CHAPTER 3

Proposed Algorithms

This chapter presents the algorithms we have developed to address the challenges of performing low-rank tensor decompositions for large-scale streaming data. We can summarize the key contributions of this chapter as follows:

1. The TT-ICE algorithm, an incremental algorithm that expands the bases of an approximation in the TT format as new data becomes available (Section 3.1)
2. The Batch hierarchical Tucker format and the BHT-l2r algorithm, enabling the approximation of batches of tensors in the batch-HT format (Section 3.2)
3. The HT-RISE algorithm, an incremental algorithm that updates the hierarchical tensor network of an accumulated tensor in the batch-HT format as new batches of tensors become available (Section 3.3)

3.1 The TT-ICE Algorithm

This section presents the first contribution of the thesis, namely the TT-ICE algorithm. Figure 3.1 illustrates the high-level flow of the proposed algorithm. We formally define the problem we aim to tackle in this section through TT-ICE as follows:

Problem 1. *(Updating an existing TT-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 TT-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 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 $\hat{\mathcal{X}}^k$.*

Please note that Section 3.1 is a slightly modified version of manuscript "An Incremental Tensor Train Decomposition Algorithm" [7] published at SIAM Journal on Scientific Computing.

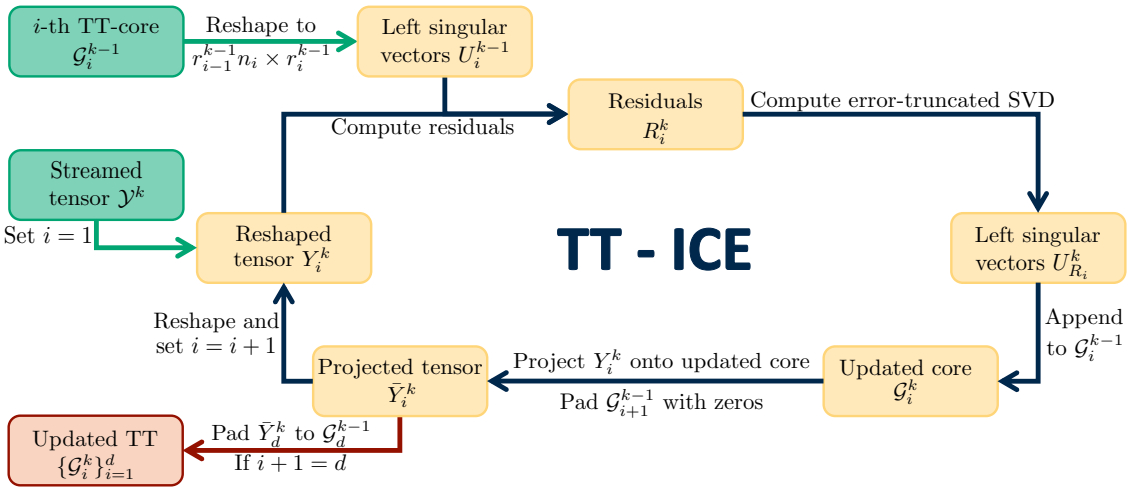


Figure 3.1: Flow of the proposed TT-ICE algorithm. The algorithm updates the representation of an accumulated tensor in tensor train format as new batches of tensors become available. Green represents the input data and red represents the output data.

Next, we will discuss why the existing approaches in the literature fall short of solving this problem by going through their lower-level details.

3.1.1 Limitations of Existing Approaches

As presented in Section 2.3.3, there are two main algorithms in the literature that solve Problem 1 – the TT-FOA algorithm and the ITTD algorithm. Both of these approaches have limitations that prevent them from being used in practice for large-scale streaming data.

The main limitation of TT-FOA is that the TT-representation error cannot be guaranteed for two reasons. First, due to the random initialization and the recursive update scheme, the TT-FOA algorithm starts with a high representation error. As time progresses and new observations become available, this error converges to a steady-state value. However, the lower bound of this steady-state value depends on the second limitation – that the TT-ranks of the accumulation are fixed as an input to the TT-FOA algorithm. If the selected TT-ranks are insufficient to represent the data accurately, there is no choice but to restart the entire TT-FOA algorithm with higher TT-ranks. Furthermore, the forgetting factor influences the TT-cores of the accumulation, so that representing the recently streamed tensors is more important than the earlier parts of the accumulation. This results in a loss of representation accuracy if the information in the tensor stream varies over time.

The limitation of the ITTD algorithm stems from the inefficiency of using the existing TT-approximation of accumulation $\mathcal{X}^k$ to aid in the incorporation of the new data tensor.

If the streamed data is highly structured, the TT-SVD step in ITTD is likely to return TT-cores that are redundant with the TT-cores of the accumulation. Therefore, the addition in TT-format likely results in linearly dependent TT-cores and superfluous TT-ranks that make TT-rounding an obligatory step for adequate compression.

In the case of an unbounded tensor stream, such a rounding procedure is required; otherwise, the TT-cores would grow without bound. However, the rounding procedure has a complexity that scales with the cube of the TT-ranks. This leads to a vicious cycle between rank growth and speed for ITTD. Putting off TT-rounding allows faster execution time, but the rank growth causes a cubic increase in time for future rounding attempts. On the other hand, avoiding rounding results in unbounded rank growth, which makes the rounding procedure impossible due to memory limitations.

In contrast to the TT-FOA algorithm, the approach we propose does not require time to converge to a steady-state error for an accumulation since truncated SVD computes the best low-rank approximation. Furthermore, when the current TT-ranks of the decomposed accumulation are not sufficient to represent the streamed tensor $\mathcal{Y}^{k+1}$ up to a prescribed precision, our approach increases the TT-ranks accordingly to meet the precision criterion. Most importantly, our approach guarantees that the whole accumulation is represented within their prescribed precision bounds at all times.

On the other hand, in contrast to the ITTD algorithm, we provide a more controlled rank growth by using the TT-cores of $\mathcal{X}^k$ as a foundation and expanding them only with complementing information obtained through $\mathcal{Y}^{k+1}$. Furthermore, for the edge case where the current TT-cores of the accumulation can represent the streamed data sufficiently accurate, our method can skip updating the TT-cores of the accumulation.

In the next section, we present our incremental TT algorithm, which provides a solution to all the shortcomings of existing incremental TT-decomposition algorithms in detail.

3.1.2 The Algorithm

In this section, we present a new incremental TT-decomposition algorithm. Similar to the existing work, we tackle the problem of computing the incremental TT-decomposition of tensor streams in two steps: *preprocessing* and *updating*. However, a distinguishing feature of our approach is that it processes and updates each dimension sequentially in a single pass. Furthermore, the subsequent algorithm considers a slightly generalized setting in which an N^{k+1}-batch of tensors become available during the $(k+1)$-th increment, such that $\mathcal{Y}^{k+1} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N^{k+1}}$.

3.1.2.1 Overview

This section walks through the steps of TT-ICE and presents a pseudocode for TT-ICE.

The proposed algorithm updates the TT-cores $\{\mathcal{G}_i^k\}_{i=1}^{d+1}$ of the approximate accumulation tensor $\hat{\mathcal{X}}^k$ to obtain updated cores $\{\mathcal{G}_i^{k+1}\}_{i=1}^{d+1}$ of the approximate accumulation tensor $\hat{\mathcal{X}}^{k+1}$. Recall that the i-th TT-core $\mathcal{G}_i \in \mathbb{R}^{r_{i-1}\times n_i \times r_i}$ is a reshaped matrix of left singular vectors $U_i \in \mathbb{R}^{r_{i-1}n_i \times r_i}$ such that,

$$\mathcal{G}_i = \texttt{reshape}\left(U_i, [r_{i-1}, n_i, r_i]\right), \tag{3.1}$$

when computed using the TT-SVD algorithm. Let $\{U_i^k\}_{i=1}^{d+1}$ denote the reshapings of the cores at increment k according to Equation (3.1), these will then be updated to obtain $\{U_i^{k+1}\}_{i=1}^{d+1}$. At each increment, these matrices will be orthonormal, and the update will ensure they remain orthonormal. Similarly, we will update the ranks $\{r_i^k\}_{i=0}^{d+1}$ to $\{r_i^{k+1}\}_{i=0}^{d+1}$. The approach is provided in Algorithm 1, and we describe each step next.

Preprocessing the first dimension of $\mathcal{Y}^{k+1}$ The first dimension is processed by projecting the reshaped tensor $Y_1 \equiv Y_{(1)}^{k+1}$ onto the orthogonal complement of the space spanned by the first dimension U_1^k through

$$R_1^k = Y_{(1)}^{k+1} - U_1^k U_1^{k^T} Y_{(1)}^{k+1} = \left(I - U_1^k U_1^{k^T}\right) Y_1, \quad R_1^k \in \mathbb{R}^{n_1 \times n_2 \cdots N^{k+1}}, \tag{3.2}$$

as provided in Line 9. The column-space of this residual is orthogonal to U_1^k, and the corresponding basis is extracted in the updated phase for each dimension via the truncated SVD.

Updating the i-th core $\mathcal{G}_i^k \to \mathcal{G}_i^{k+1}$ The update step now uses both the residual R_i^k and an intermediate, padded version of the i-th core $U_i^{k,pad}$ to compute the updated core U_i^{k+1}. First the residual is decomposed via the truncated SVD

$$R_i^k = U_{R_i}^k \Sigma_{R_i}^k V_{R_i}^k + E_i^k = U_{R_i}^k V_{R_i}^* + E_i^k, \quad \|E_i^k\|_F = \epsilon_i, \tag{3.3}$$

where the second equality multiplies the singular value matrix and the right singular vectors. The left singular vectors are orthogonal to $U_i^{k,pad}$, so the updated basis can be obtained by appending these new directions as in Line 14

$$U_i^{k+1} = \begin{bmatrix} U_i^{k,pad} & U_{R_i}^k \end{bmatrix} \in \mathbb{R}^{r_{i-1}^{k+1} n_i \times (r_i^{k+1})}, \tag{3.4}$$

where the new TT-rank becomes $r_i^{k+1} = r_i^k + r_{R_i}$.

Algorithm 1 `TT-ICE`: Incremental update of a tensor train decomposition

Input

$\left\{U_i^k\right\}_{i=1}^{d+1}$ reshaped cores of the TT-decomposition of the accumulation $\mathcal{X}^k$

$\mathcal{Y}^{k+1} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N^{k+1}}$ new tensor

$\{\epsilon_i\}_{i=1}^{d}$ SVD truncation tolerances

Output

$\left\{U_i^{k+1}\right\}_{i=1}^{d+1}$ updated cores for the accumulation $\mathcal{X}^{k+1}$

Preprocessing the first dimension

$Y_1 = \texttt{reshape}\left(\mathcal{Y}^{k+1}, \left[n_1, n_2 \dots N^{k+1}\right]\right)$

$R_1^k = \left(I - U_1^k U_1^{k^T}\right) Y_1$

$U_1^{k,pad} \leftarrow U_1^k$ ▷ First core has no padding

for $i = 1$ to $d-1$ **do**

Updating i-th core $\mathcal{G}_i^k \to \mathcal{G}_i^{k+1}$

$U_{R_i}^k, r_{R_i}, V_{R_i}^k \leftarrow \texttt{SVD}\left(R_i^k, \epsilon_i\right)$ ▷ $U_{R_i} \in \mathbb{R}^{r_{i-1}^k n_i \times r_{R_i}}$

$U_i^{k+1} \leftarrow \begin{bmatrix} U_i^{k,pad} & U_{R_i}^k \end{bmatrix}$

$U_{i+1}^{k,pad} \leftarrow \texttt{reshape}\left(\begin{bmatrix} \texttt{reshape}\left(U_{i+1}^k, [r_i^k, n_{i+1} r_{i+1}^k]\right) \\ \mathbf{0}_{r_{R_i} \times n_{i+1} r_{i+1}^k} \end{bmatrix}, [r_i^{k+1} n_{i+1}, r_{i+1}^k]\right)$ ▷ Pad for rank compat.

Preprocessing the subsequent dimensions of $\mathcal{Y}^{k+1}$

$Y_{i+1} \leftarrow \texttt{reshape}\left(U_i^{k+1^T} Y_i, [r_i^{k+1} n_{i+1}, n_{i+2} \cdots N^{k+1}]\right)$

$R_{i+1}^k \leftarrow \left(I - U_{i+1}^{k,pad} U_{i+1}^{k,pad^T}\right) Y_{i+1}$

end for

Updating the d-th core

$U_{R_d}^k, r_{R_d}, V_{R_d}^k \leftarrow \texttt{SVD}\left(R_d^k, \epsilon_d\right)$

$U_d^{k+1} \leftarrow \begin{bmatrix} U_d^{k,pad} & U_{R_d}^k \end{bmatrix}$

Updating the last core

$Y_{d+1} \leftarrow U_d^{k+1^T} Y_d$ ▷ No need to reshape for Y_{d+1} since $U_d^{k+1^T} Y_d \in \mathbb{R}^{r_d^{k+1} \times N^{k+1}}$

$U_{d+1}^{k+1} \leftarrow \begin{bmatrix} U_{d+1}^{k,pad} & Y_{d+1} \end{bmatrix}$

The updated TT-core becomes $\mathcal{G}_i^{k+1} = \texttt{reshape}\left(U_i^{k+1}, \left[r_{i-1}^{k+1}, n_i, r_i^{k+1}\right]\right)$. Note that this update increases the rank between dimensions i and $i+1$ from r_i^k to r_i^{k+1}. This new dimension causes a discrepancy that prohibits the contraction between the i-th and $(i+1)$-th cores. Therefore, the $(i+1)$-th core must be padded with zeros

$$U_{i+1}^{k,pad} \leftarrow \texttt{reshape}\left(\begin{bmatrix}\texttt{reshape}\left(U_{i+1}^k, [r_i^k, n_{i+1}r_{i+1}^k]\right) \\ \mathbf{0}_{r_{R_i} \times n_{i+1} r_{i+1}^k}\end{bmatrix}, [r_i^{k+1} n_{i+1}, r_{i+1}^k]\right). \tag{3.5}$$

Note that this padding leaves $U_{i+1}^{k,pad}$ as orthonormal and does not change the span defined by the components of the first r_i^k rows of the reshaping.

After this next core, we have a partially updated representation of the accumulation for $\hat{\mathcal{X}}^{k+1}$

$$\hat{\mathcal{X}}^{k+1} = \mathcal{G}_1^{k+1} \ {}_{3\times 1} \ \cdots \ {}_{i+1\times 1} \ \mathcal{G}_i^{k+1} \ {}_{i+2\times 1} \ \mathcal{G}_{i+1}^{k,\text{pad}} \ {}_{i+3\times 1} \ \mathcal{G}_{i+2}^{k} \ {}_{i+4\times 1} \ \cdots \ {}_{d+2\times 1} \ \mathcal{G}_{d+1}^{k}, \tag{3.6}$$

where $\mathcal{G}_{i+1}^{k,\text{pad}}$ refers to the padded core formed by reshaping $U_{i+1}^{k,pad}$ following the update in (3.5). After this step, we can proceed to pre-process in preparation for updating the next dimension.

Preprocessing the subsequent dimensions of $\mathcal{Y}^{k+1}$ Preprocessing preparation for the update of the next dimension begins by updating the projection of the new data onto the updated core and reshaping

$$Y_{i+1} = \texttt{reshape}\left(U_i^{k+1^T} Y_i, [r_i^{k+1} n_{i+1}, n_{i+2} \cdots N^{k+1}]\right). \tag{3.7}$$

This projection uses the *updated* tensor cores U_i^{k+1} and therefore has dimensions corresponding to the updated rank r_i^{k+1}. Then, the residual with respect to the span of the next core is obtained via a projection onto the orthogonal complement to the existing space

$$R_i^k = \left(I - U_i^{k,pad} U_i^{k,pad^T}\right) Y_{i+1}. \tag{3.8}$$

Updating the last core The procedure in (3.7)-(3.5) is repeated sequentially for the first d dimensions. Then after updating $U_d^{k,pad}$ to U_d^{k+1} and padding U_{d+1}^k with zeros as in (3.5), projecting Y_d onto U_d^{k+1} as in Line 24 yields Y_{d+1}, with which we can update $U_{d+1}^{k,pad}$ as

$$U_{d+1}^{k+1} \leftarrow \begin{bmatrix} U_{d+1}^{k,pad} & Y_{d+1} \end{bmatrix}, \tag{3.9}$$

and obtain the representation for $\hat{\mathcal{X}}^{k+1}$ in TT-format.

3.1.2.2 Analyses of the Error Bounds

This section provides proof that TT-ICE can achieve and maintain a target accuracy throughout the compression process.

We first show that Algorithm 1 provides guaranteed reconstruction error of a new tensor given appropriate truncation of the truncated SVDs. Following the completion of TT-ICE, the approximation of the most recently compressed data point can be obtained by

$$\hat{\mathcal{Y}}^{k+1} = \mathcal{G}_1^{k+1} \ {}_3\times_1 \ \cdots \ {}_{d+1}\times_1 \ \mathcal{G}_d^{k+1} \ {}_{d+2}\times_1 \ \mathcal{G}_{d+1}^{k+1}[-n_{d+1}^{k+1}:], \tag{3.10}$$

where the index $[-n_{d+1}^{k+1}:]$ denotes the last n_{d+1}^{k+1} columns of the core, as seen in Equation (3.9). Recall that $\mathcal{G}_{d+1}^{k+1}$ is a matrix and that the columns $\mathcal{G}_{d+1}^{k+1}[-n_{d+1}^{k+1}:]$ are the only elements of the updated accumulation that are unique to the $(k+1)$-th data point. We now show that the algorithm enables a well-controlled approximation error of this latest tensor.

Theorem 1. *Let $\{U_i^k\}_{i=1}^{d+1}$ denote the unfolded TT-cores of the approximate accumulation $\hat{\mathcal{X}}^k$, $\mathcal{Y}^{k+1}$ be a new streamed tensor and $\{\epsilon_i\}_{i=1}^{d}$ be the SVD truncation tolerances, then the TT-ICE algorithm computes a TT-approximation $\hat{\mathcal{Y}}^{k+1}$ (3.10) satisfying*

$$\|\mathcal{Y}^{k+1} - \hat{\mathcal{Y}}^{k+1}\|_F \leq \sqrt{\sum_{i=1}^{d} \epsilon_i^2}. \tag{3.11}$$

Proof. The proof is similar to that of [191, Thm 2.2] and is by induction. For $d = 1$, the statement follows from the properties of the truncated SVD.

Now we consider an arbitrary $d > 1$. The incremental updates start with the first dimension. We start with processing the first core, which works on the first unfolding of the new data. First, we show that computing the SVD of the residual leads to a low-rank approximation of $Y_{(1)}^{k+1}$ with an approximation error bounded by a controlled truncation tolerance. The definitions of the residual and projection given by Equations (3.2) lead to

$$Y_{(1)}^{k+1} = U_1^k U_1^{k^T} Y_{(1)}^{k+1} + R_1^k = U_1^k U_1^{k^T} Y_1 + U_{R_1}^k V_{R_1}^k + E_1^k = U_1^{k+1} \underbrace{\begin{bmatrix} U_1^{k^T} Y_1 \\ V_{R_1}^k \end{bmatrix}}_{B_1} + E_1^k = U_1^{k+1} B_1 + E_1^k, \tag{3.12}$$

where the third equality appends the existing basis U_1^k and the new directions obtained from the SVD of the residual $U_{R_1}^k$ to form the updated core U_1^{k+1}. Thus, after the first dimension is processed, the first core of $\mathcal{G}_1^{k+1}$ is obtained, as a reshaping of U_1^{k+1}. Equation (3.12)

demonstrates that this core is multiplied by a tensor $\mathcal{B}_1 \in \mathbb{R}^{r_1^{k+1} n_2 \times n_3 \cdots n_{d+1}^{k+1}}$ that has reshaping $B_1 \in \mathbb{R}^{r_1^{k+1} \times n_2 \cdots n_{d+1}^{k+1}}$, and that the truncated SVD implies the approximation error ϵ_1^2:

$$\|Y_{(1)}^{k+1} - U_1^{k+1} B_1\|_F^2 \leq \epsilon_1^2. \tag{3.13}$$

The algorithm then proceeds to decompose the still high-dimensional B_1 into an approximation. Let $\hat{B}_1$ denote this approximation of the remaining dimensions so that we now seek to bound the error

$$\|\mathcal{Y}^{k+1} - \hat{\mathcal{Y}}^{k+1}\|_F^2 = \|Y_{(1)}^{k+1} - U_1^{k+1} \hat{B}_1\|_F^2. \tag{3.14}$$

Next, this error can be rewritten in terms of the known error (3.13) and the subsequent approximation error incurred by $\hat{B}_1$. To this end, add and subtract the exact B_1 to obtain

$$\begin{aligned} \|\mathcal{Y}^{k+1} - \hat{\mathcal{Y}}^{k+1}\|_F^2 = \|Y_{(1)}^{k+1} - U_1^{k+1} \hat{B}_1\|_F^2 &= \|Y_{(1)}^{k+1} - U_1^{k+1} \left(\hat{B}_1 + B_1 - B_1 \right)\|_F^2, && (3.15) \\ &= \|Y_{(1)}^{k+1} - U_1^{k+1} B_1\|_F^2 + \|U_1^{k+1} \left(B_1 - \hat{B}_1 \right)\|_F^2, && (3.16) \\ &\leq \epsilon_1^2 + \|B_1 - \hat{B}_1\|_F^2, && (3.17) \end{aligned}$$

where U_1^{k+1} has orthonormal columns and the inequality in Eq. (3.17) arises from Eq. (3.13).

At this stage of the algorithm, we have updated U_1^{k+1} and padded U_2^k with $r_{R_1} \times n_2 r_2^k$ zeros according to (3.5). The rest of the algorithm proceeds by now performing the same approximation for B_1, beginning with a projection onto the zero padded second core, $U_2^{k,pad}$. First we note that the orthonormality of the left singular vectors implies $B_1 = U_1^{(k+1)^T} Y_{(1)}^{k+1} = U_1^{(k+1)^T} Y_1$ from (3.12). Combined with Equation (3.7), these facts imply that $Y_2 \equiv B_1$ up to a reshaping. Then the next residual becomes

$$\left(I - U_2^{k,pad} U_2^{k,pad^T} \right) B_1 \equiv \left(I - U_2^{k,pad} U_2^{k,pad^T} \right) Y_2 = R_2^k. \tag{3.18}$$

Similar to (3.12), we can expand B_1 into the components parallel to the space spanned by the padded by $U_2^{k,pad}$ and orthogonal to this space so that

$$B_1 = U_2^{k,pad} U_2^{k,pad^T} B_1 + R_2^k = U_2^{k,pad} U_2^{k,pad^T} Y_2 + U_{R_2}^k V_{R_2}^k + E_2^k = U_2^{k+1} \underbrace{\begin{bmatrix} U_2^{k,pad^T} Y_2 \\ V_{R_2}^k \end{bmatrix}}_{B_2} + E_2^{k+1} = U_2^{k+1} B_2 + E_2^k. \tag{3.19}$$

Similar to (3.13), the truncated SVD for this dimension ensured $\|E_2^k\|_F^2 \leq \epsilon_2^2$. Then, parallel

to (3.15)-(3.17), this time we expand the error in B_1 according to

$$\begin{aligned}\|B_1 - \hat{B}_1\|_F^2 =& \|B_1 - U_2^{k+1}\hat{B}_2\|_F^2, \\ =& \|B_1 - U_2^{k+1}\left(\hat{B}_2 + B_2 - B_2\right)\|_F^2, \\ =& \|B_1 - U_2^{k+1}B_2\|_F^2 + \|U_2^{k+1}\left(B_2 - \hat{B}_2\right)\|_F^2, \\ \leq& \epsilon_2^2 + \|B_2 - \hat{B}_2\|_F^2.\end{aligned} \tag{3.20}$$

If we rewrite Equation (3.19) for B_{d-1}, we see that

$$B_{d-1} = U_d^{k,pad}U_d^{k,pad^T}B_{d-1} + R_d^k \equiv U_d^{k,pad}U_d^{k,pad^T}Y_d + U_{R_d}^k V_{R_d}^k + E_d^k = U_d^{k+1}\underbrace{\begin{bmatrix} U_d^{k,pad^T}Y_d \\ V_{R_d}^k \end{bmatrix}}_{B_d} + E_d^{k+1} = U_d^{k+1}B_d + E_d^k \tag{3.21}$$

with $B_d \in \mathbb{R}^{r_d^{k+1} \times n_{d+1}^{k+1}}$, which is equal to Y_{d+1}. We directly append Y_{d+1} to the $d+1$-th core and conclude our update procedure for $\mathcal{Y}^{k+1}$. Therefore, for a $d+1$ dimensional tensor, repeating the update for the first d dimensions successively yields the approximation error

$$\|\mathcal{Y}^{k+1} - \hat{\mathcal{Y}}^{k+1}\|_F^2 \leq \sum_{i=1}^{d} \epsilon_i^2. \tag{3.22}$$

Computing the square root of this term yields (3.11) and thus concludes our proof. □

A straightforward corollary shows that TT-ICE can then ensure that the compression of the new data point can be achieved to any desired tolerance.

Corollary 1. *Let $\{U_i^k\}_{i=1}^{d+1}$ denote the unfolded TT-cores of the approximate accumulation $\hat{\mathcal{X}}^k$, $\mathcal{Y}^{k+1}$ be a new streamed tensor. If $\epsilon_i = \epsilon = \varepsilon_{des}\|\mathcal{Y}^{k+1}\|_F/\sqrt{d}$, then*

$$\frac{\|\mathcal{Y}^{k+1} - \hat{\mathcal{Y}}^{k+1}\|_F}{\|\mathcal{Y}^{k+1}\|_F} \leq \varepsilon_{des} \tag{3.23}$$

for any $\varepsilon_{des} > 0$.

Proof. The proof is a direct result of Theorem 1. If we assume $\epsilon_i = \epsilon$ for all i and simply set

$$\sqrt{\sum_{i=1}^{d} \epsilon_i^2} = \epsilon\sqrt{d} = \varepsilon_{des}\|\mathcal{Y}^{k+1}\|_F, \tag{3.24}$$

we can assure the desired relative error upper bound ε_{des} by truncating the SVD on Line 21 of TT-ICE at $\epsilon = \varepsilon_{des}\|\mathcal{Y}^{k+1}\|_F/\sqrt{d}$. □

The last step is to ensure that the update of $\mathcal{X}^k$ does not reduce the approximation error of the previously compressed elements $\mathcal{Y}^i$ for $i \leq k$.

Theorem 2. *Let $\{U_i^k\}_{i=1}^{d+1}$ denote the unfolded TT-cores of the approximate accumulation $\hat{\mathcal{X}}^k$ such that the approximation of any existing tensor $\mathcal{Y}^\ell$ satisfies $\|\mathcal{Y}^\ell - \hat{\mathcal{Y}}^\ell\|_F \leq \epsilon\|\mathcal{Y}^\ell\|_F$ for $\ell = 1, \ldots, k$, and $\epsilon > 0$. Let $\mathcal{Y}^{k+1}$ be a new streamed tensor, and $\{\epsilon_i\}_{i=1}^d$ be the SVD truncation tolerances; then the updated TT-cores $\{U_i^{k+1}\}_{i=1}^{d+1}$ computed by TT-ICE represent an approximate accumulation $\hat{\mathcal{X}}^{k+1}$ that still satisfies $\|\mathcal{Y}^\ell - \hat{\mathcal{Y}}^\ell\|_F \leq \epsilon\|\mathcal{Y}^\ell\|_F$. for $\ell = 1, \ldots, k$.*

Proof. The proof shows simply that the core modifications performed by TT-ICE have no impact on the representation of earlier tensors. First recall from (3.10) that the approximation of the ℓ-th tensor is given by

$$\hat{\mathcal{Y}}^\ell = \mathcal{G}_1^k \ {}_3\times_1 \ \cdots \ {}_{d+1}\times_1 \ \mathcal{G}_d^k \ {}_{d+2}\times_1 \mathcal{G}_{d+1}^k\left(\mathcal{Y}^\ell\right), \quad \ell = 1, \ldots, k, \tag{3.25}$$

where we slightly abuse the notation by letting $\mathcal{G}_{d+1}^k\left(\mathcal{Y}^\ell\right)$ denote the columns of $\mathcal{G}_{d+1}^k$ corresponding to $\mathcal{Y}^\ell$.

First, we note that TT-ICE only appends columns to the last core, so that the columns corresponding to the ℓ-th tensor are unchanged when $\mathcal{G}_{d+1}^k$ is updated to $\mathcal{G}_{d+1}^{k+1}$. Thus it remains to show that the portions of $\mathcal{G}_i^k$ that multiply together and finally multiply the corresponding columns of $\mathcal{G}_{d+1}^k$ remain unchanged as the U_i^k are updated to U_i^{k+1}.

Let $G_i^k[i_1] \in \mathbb{R}^{r_{i-1}\times r_i}$ denote the slice of each TT-core for $i = 1 \ldots n_i$. Then (3.25) can be written for each element of $\mathcal{Y}^\ell$ according to

$$\hat{\mathcal{Y}}^\ell(i_1, \ldots, i_d, : \mathcal{Y}^\ell) = \mathcal{G}_1^k[i_1]\mathcal{G}_2^k[i_2]\cdots\mathcal{G}_d^k[i_d]\mathcal{G}_{d+1}^k[: \mathcal{Y}^\ell], \tag{3.26}$$

where $[: \mathcal{Y}^\ell]$ is used to denote the columns of the last core pertaining to $\hat{Y}^\ell$ so that $\mathcal{G}_{d+1}^k[: \mathcal{Y}^\ell] \equiv \mathcal{G}_{d+1}^k\left(\mathcal{Y}^\ell\right)$. Next, the zero padding and column appending of each U_i^k imply the following relationship between G_i^k and G_i^{k+1}

$$\mathcal{G}_j^{k+1}[i_j] = \begin{cases} \begin{bmatrix} \mathcal{G}_1^k[i_1] & L_1 \end{bmatrix} & , \ j = 1; \\ \begin{bmatrix} \mathcal{G}_j^k[i_j] & L_{j,1} \\ 0 & L_{j,2} \end{bmatrix} & , \ j = 2, \ldots d; \\ \begin{bmatrix} \mathcal{G}_{d+1}^k[i_{d+1}] \\ 0 \end{bmatrix} & , \ j = d+1, \end{cases} \tag{3.27}$$

where the zeros in the lower left block come from the padding, and the $L_{j,l}$ matrices arise from appending the new directions. Now the representation of the ℓ-th tensor in the stream with the new cores becomes

$$\hat{\mathcal{Y}}^{\ell} = \begin{bmatrix} \mathcal{G}_1^k[i_1] & L_1 \end{bmatrix} \begin{bmatrix} \mathcal{G}_2^k[i_2] & L_{2,1} \\ 0 & L_{2,2} \end{bmatrix} \cdots \begin{bmatrix} \mathcal{G}_d^k[i_d] & L_{d,1} \\ 0 & L_{d,2} \end{bmatrix} \begin{bmatrix} \mathcal{G}_d^k[i_{d+1}] \\ 0 \end{bmatrix} = \mathcal{G}_1^k[i_1]\mathcal{G}_2^k[i_2]\cdots\mathcal{G}_d^k[i_{d+1}] = \hat{\mathcal{Y}}^{\ell}, \tag{3.28}$$

where the locations of the zeros enable the new terms to appropriately cancel all the new terms. Thus the representation of previously compressed tensors does not change, and therefore their reconstruction error remains the same after the core update. □

3.1.2.3 Heuristics to Improve Performance

This section discusses three heuristic modifications to TT-ICE to reduce the computational load and memory requirements of our approach.

These approaches aim to reduce the number of core updates between $\mathcal{X}^k$ and $\mathcal{X}^{k+1}$. First, we provide a metric based on TT-ranks to decide whether an attempt at updating a TT-core should even be made. Second, when a new batch of N^{k+1} tensors is presented at increment $k+1$, we describe an approach to subselect from this batch to only perform an update with a smaller number of tensors. This update is based on each of the approximation errors of individual tensors in the streamed batch. Third, we propose a mechanism to determine when it is feasible to skip the core updating process.

***Core occupancy* for reducing the number of cores to be updated** The TT-ICE lagorithm proposes a scheme for updating all the TT-cores $\{\mathcal{G}_i^k\}_{i=1}^{d+1}$ at each increment. However, the new information provided by increment may not be uniform throughout all these TT-cores. If we can detect the core where the most information loss occurs, focusing update efforts on that core could increase the efficiency of Algorithm 1.

We propose a heuristic named *core occupancy* to measure how much information is already represented by a core to determine if it should be updated. Recall that the i-th TT-core is constructed by reshaping the left singular vectors $U_i^k \in \mathbb{R}^{r_{i-1}^k n_i \times r_i^k}$. As a result, there are r_i^k orthogonal vectors from the possible $r_{i-1}^k n_i$ basis vectors. For a given TT-core, core occupancy represents the ratio of the truncation rank r_i^k over the maximum rank possible (the number of rows $r_{i-1}^k n_i$). It can also be seen as a ratio of the column to row-ranks:

$$\texttt{occupancy}\left(\mathcal{G}_i^k\right) = \frac{r_i^k}{r_{i-1}^k n_i}. \tag{3.29}$$

If a core has a high occupancy ratio, then we can skip its update.

Selecting a subset of tensors from the batch $\mathcal{Y}^{k+1}$ to update $\{\mathcal{G}_i^k\}_{i=1}^{d+1}$ If the new data comes in a batch (i.e. $N^{k+1} > 1$), another way of reducing the computational cost of updating the TT-cores is to use a reduced number of tensors from the batch $\mathcal{Y}^{k+1}$ to update the first d cores of the accumulation tensor. To subselect which elements of the batch to use for the update, we first compute the projection of $\mathcal{Y}^{k+1}$ onto TT-cores $\{\mathcal{G}_i^k\}_{i=1}^{d+1}$ and use the relative projection error of the individual observations as a heuristic to select a subset of $\mathcal{Y}^{k+1}$. With a slight abuse of notation, we denote the i-th observation in $\mathcal{Y}^{k+1}$ as $\mathcal{Y}^{k+1}(i)$, where $i = 1, \ldots, N^{k+1}$. Let $\tilde{\mathcal{Y}}^{k+1}$ be the approximation of $\mathcal{Y}^{k+1}$ using the TT-cores $\{\mathcal{G}_i^k\}_{i=1}^{d+1}$. Similarly, let $\tilde{\mathcal{Y}}^{k+1}(i)$ denote the i-th observation in $\tilde{\mathcal{Y}}^{k+1}$. Then, the vector $\varepsilon_{\mathcal{Y}^{k+1}} \in \mathbb{R}^{N^{k+1}}$ is the vector of relative errors of individual observations, where

$$\varepsilon_{\mathcal{Y}^{k+1}}(i) = \frac{\|\mathcal{Y}^{k+1}(i) - \tilde{\mathcal{Y}}^{k+1}(i)\|_F}{\|\mathcal{Y}^{k+1}(i)\|_F}, \tag{3.30}$$

for $i = 1, \ldots, N^{k+1}$.

Recall that we can use Algorithm 1 with truncated SVD and update the TT-cores of $\hat{\mathcal{X}}^k$ to $\hat{\mathcal{X}}^{k+1}$ so that $\mathcal{Y}^{k+1}$ is represented within a truncation error threshold. Let ε_{des} denote the determined relative error threshold as in Corollary 1. Before attempting an update, we calculate the approximation error of each new data point $\varepsilon_{\mathcal{Y}^{k+1}}$ with the existing TT-cores using (3.30) and then compute its average, $\texttt{mean}\left(\varepsilon_{\mathcal{Y}^{k+1}}\right)$. If the average approximation error is greater than the desired error tolerance ($\texttt{mean}\left(\varepsilon_{\mathcal{Y}^{k+1}}\right) > \varepsilon_{des}$), then we will proceed to update at least some of the cores. In particular, we only update the TT-cores of the accumulation using the data points for which the approximation error exceeds ε_{des}. Let $\mathcal{D}^{k+1}$ be that constructed new $(d+1)$-way tensor with a reduced number of observations such that

$$\mathcal{D}^{k+1} = \left\{\mathcal{Y}^{k+1}(i) : \ \varepsilon_{\mathcal{Y}^{k+1}}(i) > \varepsilon_{des}\right\}_{i=1}^{N^{k+1}}. \tag{3.31}$$

Since the algorithm operates on a subset of observations, the truncation parameter needs to be adjusted accordingly. Using the same ε_{des} as $\mathcal{Y}^{k+1}$ for $\mathcal{D}^{k+1}$ enforces a tighter relative error upper bound on $\mathcal{Y}^{k+1}$. This tighter error bound results in increased TT-ranks that impair the compression performance. To avoid this, we need to relax the truncation parameter ε_{des} using the approximation error of the discarded observations. Let $\mathcal{D}_C^{k+1}$ be the tensor constructed with those discarded observations such that $\mathcal{D}_C^{k+1} = \left\{\mathcal{Y}^{k+1}(i) : \ \varepsilon_{\mathcal{Y}^{k+1}}(i) \leq \varepsilon_{des}\right\}_{i=1}^{N^{k+1}}$. Then, we compute the relaxed relative error toler-

ance ε_{upd} for the tensor of selected observations $\mathcal{D}^{k+1}$ as

$$\varepsilon_{upd} = \sqrt{\frac{\left(\varepsilon_{des}\|\mathcal{Y}^{k+1}\|_F\right)^2 - \|\mathcal{D}_C^{k+1} - \tilde{\mathcal{D}}_C^{k+1}\|_F^2}{\|\mathcal{D}^{k+1}\|_F^2}}, \tag{3.32}$$

where $\tilde{\mathcal{D}}_C^{k+1}$ is the approximation of $\mathcal{D}_C^{k+1}$ using $\{U_i^k\}_{i=1}^{d+1}$. The first term in the numerator is the maximum amount of error that TT-ICE can allow for $\mathcal{Y}^{k+1}$ in the Frobenius norm, the second term is the approximation error of the discarded observations in the Frobenius norm, and the denominator is the Frobenius norm of the tensor of selected observations. Since the discarded observations have a relative error $\leq \varepsilon_{des}$, subtracting the approximation error gives how much error TT-ICE can allow if it only uses $\mathcal{D}^{k+1}$ to update the TT-cores. The denominator functions as a normalizing factor, converting the error in the numerator to a relative error for incremental updates. Through ε_{upd}, the low approximation error of the discarded observations will be balanced by tolerating slightly more error for $\mathcal{D}^{k+1}$ and the overall approximation $\hat{\mathcal{Y}}^{k+1}$ will have a relative error closer to ε_{des}. Note that for the edge case where all $\varepsilon_{\mathcal{Y}^{k+1}}(i) > \varepsilon_{des}$, we have $\mathcal{D}^{k+1} = \mathcal{Y}^{k+1}$ and $\mathcal{D}_C^{k+1} = \emptyset$. This results in $\varepsilon_{upd} = \varepsilon_{des}$, therefore providing a consistent method to update ε_{des}.

If the observations in the same batch have similar norms, we can approximate Eq. (3.32) by replacing the norm operators with the count of observations in each tensor $|\cdot|$ and get

$$\varepsilon_{upd} \approx \frac{\varepsilon_{des}N^{k+1} - \varepsilon_{\mathcal{D}_C^{k+1}}|\mathcal{D}_C^{k+1}|}{|\mathcal{D}^{k+1}|}, \tag{3.33}$$

where $\varepsilon_{\mathcal{D}_C^{k+1}}$ is the mean relative error of the observations in $\mathcal{D}_C^{k+1}$ computed analogously to (3.30).

Through these modifications, we prevent a superfluous increase in TT-ranks. After determining ε_{upd}, we simply compute the truncation parameter for the SVD using ε_{upd} as $\epsilon = \frac{\varepsilon_{upd}}{\sqrt{d}}\|\mathcal{D}^{k+1}\|_F$.

Skip updating the first d TT-cores Our final heuristic is to skip updating the cores if the average error of the tensors in a batch is less than the desired threshold: $\texttt{mean}\left(\varepsilon_{\mathcal{Y}^{k+1}}\right) \leq \varepsilon_{des}$. We justify this heuristic via the following argument.

Let ε_{k+1} represent the relative error of approximating the new tensor via the existing cores

$$\varepsilon_{k+1} = \frac{\|\mathcal{Y}^{k+1} - \tilde{\mathcal{Y}}^{k+1}\|_F}{\|\mathcal{Y}^{k+1}\|_F} = \sqrt{\frac{\sum_{i=1}^{n_{d+1}^{k+1}} \|\mathcal{Y}^{k+1}(i) - \tilde{\mathcal{Y}}^{k+1}(i)\|_F^2}{\sum_{i=1}^{n_{d+1}^{k+1}} \|\mathcal{Y}^{k+1}(i)\|_F^2}}, \tag{3.34}$$

where $\tilde{\mathcal{Y}}^{k+1}$ is the approximation of $\mathcal{Y}^{k+1}$ with the existing TT-cores. If $\varepsilon_{k+1} \leq \varepsilon_{des}$, then the first d TT-cores of $\mathcal{X}_k$ can represent $\mathcal{Y}^{k+1}$ sufficiently accurate and do not need updates to meet the desired accuracy. As a result, the error-truncated SVD in Line 13 of Algorithm 1 returns empty matrices and TT-ICE will return the first d TT-cores without an update. In order to save invaluable computation time, we can calculate ε_{k+1} using (3.34) and complete the core update by appending the TT-representation of $\mathcal{Y}^{k+1}$ after projecting onto the TT-cores of $\mathcal{X}^k$.

Let $\hat{Y}^{k+1}$ be the projection of $\mathcal{Y}^{k+1}$ onto the first d TT-cores $\{\mathcal{G}_i^k\}_{i=1}^d$. Then, updating $\mathcal{G}_{d+1}^k$ is simply done by appending $\hat{Y}^{k+1}$ to $\mathcal{G}_{d+1}^k$ as

$$\mathcal{G}_{d+1}^{k+1} \leftarrow \begin{bmatrix} \mathcal{G}_{d+1}^k & \hat{Y}^{k+1} \end{bmatrix}. \tag{3.35}$$

However, an explicit computation of (3.34) can become computationally expensive if the batch consists of a high number of observations. In that case, we investigate the use of the surrogate of $\varepsilon_{\mathcal{Y}^{k+1}}$ from (3.30) as an approximation to ε_{k+1}. In future works, this strategy might be adapted to sampling tensors in the batch to have a stochastic approximation of the average. Meanwhile, the numerical experiments in Section 3.1.3 use $\texttt{mean}(\varepsilon_{\mathcal{Y}^{k+1}})$ to determine if TT-ICE* should skip updating the first d TT-cores. As an example, Figure 3.4 provides empirical proof that using the mean as an approximation does not result in a violation of the relative error upper bound. However, note that there might be other, more conservative heuristic measures available to approximate ε_{k+1}.

The pseudocode of the modified algorithm is provided in Algorithm 2. The `project` function on Line 36 projects $\mathcal{Y}^{k+1}$ sequentially onto $\{U_i^{k+1}\}_{i=1}^d$ and reshapes into proper dimensions in a way similar to Line 33.

3.1.3 Experiments

In this section, we compare the performance of TT-ICE and TT-ICE* with existing approaches in compressing large-scale data including videos from gameplay sequences and grid data from physics-based simulations. In all the experiments, the TT-SVD algorithm [191, Alg 1] is applied to obtain an initial set of TT-cores and subsequently, the cores are updated using each of the incremental algorithms. In all of the results provided, ITTDk corresponds to updating the TT-cores of the accumulation using ITTD and performing TT-rounding [191, Alg 2] after every k-th update step. For comparisons between TT-FOA and our approach, we refer to Appendix A.1.1. We do not include it here because TT-FOA does not support rank adaptation and does not offer an upper bound on approximation error. As a result, it either performed poorly or was unable to handle the datasets considered.

3.1.3.1 Evaluation Criteria

This section contains performance metrics we use to compare performance of algorithms.

The performance of the incremental algorithms is evaluated using three main criteria. First, the compression ratio (CR) of the accumulation $\mathcal{X}^k$ with core sizes $n_1 \times n_2 \times \cdots \times n_{d+1}$ is defined as

$$CR = \frac{\text{number of elements of full tensor}}{\text{number of parameters in compressed representation}} = \frac{\prod_{i=1}^{d+1} n_i}{\sum_{i=1}^{d+1} r_{i-1} n_i r_i}, \tag{3.36}$$

where r_i is the i-th TT-rank. Second, the RRE of the same tensor is given by

$$RRE = \frac{\|\mathcal{X}^k - \hat{\mathcal{X}}^k\|_F}{\|\mathcal{X}^k\|_F}. \tag{3.37}$$

Similarly, we can measure the error in the representation of *unseen* data: since the last dimension of $\mathcal{X}^k$ provides a latent representation for individual observations stored in the stream, as shown in Eq. (3.10), we can use the first d cores to estimate a latent representation for unseen data by projecting onto the first d dimensions. We shall call the error in the estimation the RTE, which can again be computed using the Frobenius error similar to RRE. An RTE lower than the target tolerance ε_{des} indicates that the existing basis of the accumulation is expressive enough to represent this unseen data. Our final evaluation metric is the *walltime*, which measures the CPU time spent executing the steps of the corresponding incremental algorithms and does not include the time for data to load into memory.

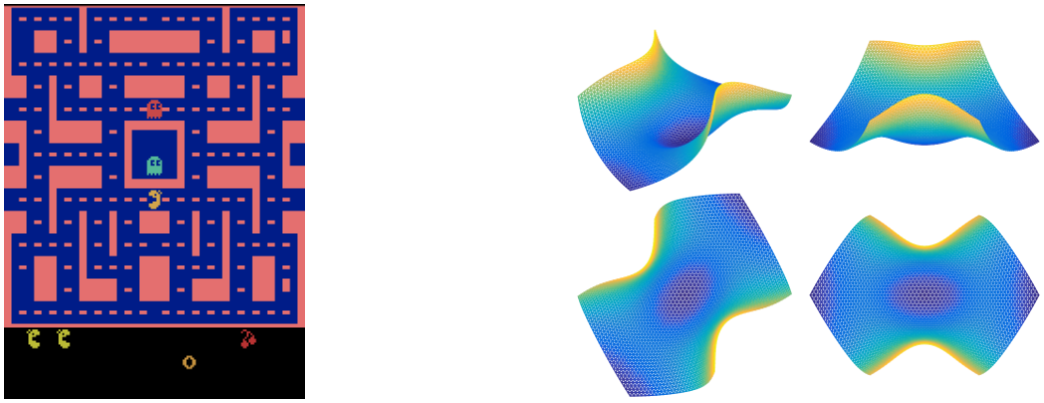
3.1.4 Datasets and Experiments

This section includes tests on two different types of datasets (snapshots in Figure 3.2): (i) Raw pixel data from an ATARI game-playing reinforcement learning (RL) agent [50] and (ii) Grid data from a numerical simulation of self-oscillating gels [9].

3.1.4.1 ATARI Gameplay Sequences

This section includes tests of TT-ICE's performance to compress raw pixel data.

The first dataset involves a sequential set of video game screenshots. It is often useful to reduce the dimension of this type of video data to enable learning in the latent space. This dataset consists of a collection of gameplay screen captures from various ATARI games. The dataset is collected using a video game-playing reinforcement learning (RL) agent trained by a Deep Q-Network model [182] from the RL-Baselines Zoo package [208] and further



(a) (b)

Figure 3.2: Example visualizations from the datasets used in the compression experiments. (a) A frame from a Ms.Pac-Man gameplay session. (b) Snapshots from self-oscillating gel simulation using the simulation developed in [9].

trained using the Stable Baselines platform [99]. The ATARI games we used are Ms.Pac-Man, Enduro, Seaquest, Q*bert, Breakout, Pong, and Beamrider. We present the results of experiments for Ms.Pac-Man game captures in this section and the rest of the games in the Appendix.

Each game has multiple individual gameplay sessions, which we refer to as *runs*. Each run may have different durations and therefore a different number of frames (see Figure 3.2a) with dimensions $210 \times 160 \times 3$ ($Width \times Height \times RGB$). Each individual run is treated as an incremental unit and reshaped into a 5-way tensor of dimension $30 \times 28 \times 40 \times 3 \times N^k$, where N^k is the number of frames in the k-th run. Depending on the duration of each gameplay sequence, each run consists of a varying number of frames. Therefore, the 5-way tensors are stacked along the fifth dimension.

We compared TT-ICE, TT-ICE*, and ITTD5 on all of the different game datasets, and the results are summarized in Table A.1. A more detailed study of the efficiency of each algorithm was done for Ms.Pac-Man. For the comprehensive experiments with Ms.Pac-Man gameplay sequences, we used a training set of 60 runs for incremental updates and a validation set of 160 runs to measure the prediction error on unseen runs. This prediction tests the performance of the TT-format to find a suitable latent space for describing Ms.Pac-Man frames. We present the results of detailed experiments in Figures 3.3 to 3.5. In those figures, *Subselect* means updating TT-cores using only the observations from $\mathcal{Y}^{k+1}$ with relative error higher than ε_{des} and *SS* means skipping updates for the first d TT-cores when $\texttt{mean}(\varepsilon_{\mathcal{Y}^{k+1}}) \leq \varepsilon_{des}$ along with *Subselect* heuristic. Please note that the *Occupancy* heuristic is only used when the complete TT-ICE* algorithm is used.

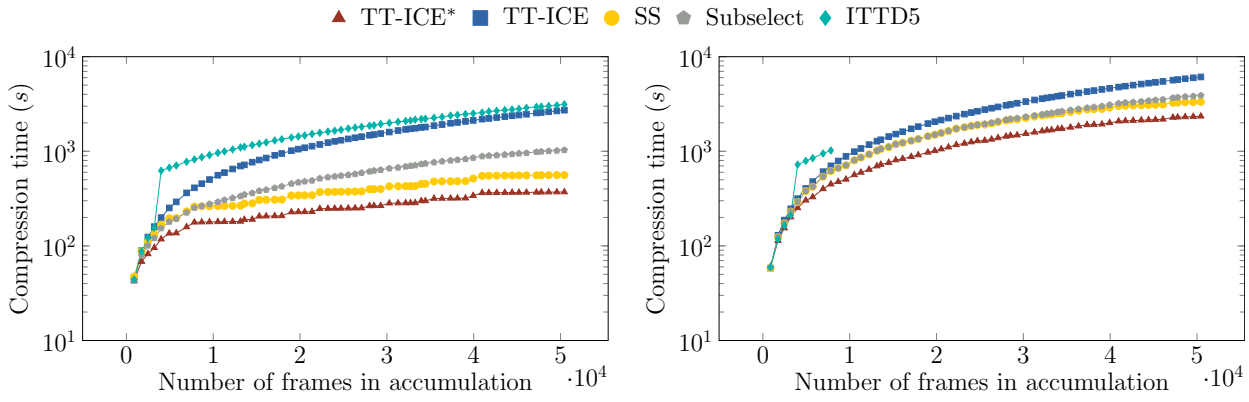
For all other games in Table A.1, we used a training set of 60 runs and a validation set of 100 runs. Enduro and Pong were the only exceptions to that setup since they had a much higher number of frames in each run. This resulted in much higher memory requirements than all other games even just for storing the runs. Again, due to the high number of frames per run, the validation tests required a significant amount of time. Therefore we used a training set of 40 runs and a validation set of 40 runs. During the repetitions for different algorithms, the streaming order was the same for all methods subject to investigation. All experiments were repeated for two ε_{des} settings, $\varepsilon_{des} = 0.1$ and $\varepsilon_{des} = 0.01$. Finally, an occupancy threshold of 0.8 is set for this class of experiments.

Compression results This section includes the results of experiments on Ms.Pac-Man frames, and then comment on the compression experiments with other ATARI games.

Figures 3.3a and 3.3b show the influence of each variation of TT-ICE on the overall compression time and compare these results with ITTD5. In Figure 3.3a, ITTD5 performs worse than all of the TT-ICE variations at $\varepsilon_{des} = 0.1$. Specifically, Figure 3.3a depicts that implementing heuristic improvements provides at least 62% reduction in compression time. This improvement in compression time reaches its peak when all of the heuristic upgrades are implemented (i.e. when the complete TT-ICE* algorithm is used). Note that when the truncation tolerance is tightened to $\varepsilon_{des} = 0.01$, the problem with ITTD intensifies. This time, ITTD5 fails to compress the full duration of the stream due to insufficient memory and fails at the 10th increment while attempting TT-rounding. On the other hand, TT-ICE and variations of TT-ICE* display performances in parallel to the experiments with $\varepsilon_{des} = 0.1$.

Figures 3.4a and 3.4b investigate a difference in the representation error of the TT-cores trained with different algorithms. For each case, we present the mean RRE over the compressed portion of the training set and the mean RTE over the entire validation set after each increment. Figure 3.4a shows that all methods can successfully represent the streamed data within the desired relative error upper bound ε_{des}. Furthermore, Figure 3.4a depicts that the mean RTE of TT-ICE and TT-ICE* converge asymptotically to the mean RRE. Since the mean RTE represents the representation quality of the validation set, these results indicate that a suitable basis is found prior to observing the validation dataset itself. Once the basis to represent observations within the desired accuracy is complete, TT-ICE will not be able to find any unique orthogonal directions to expand the bases of TT-cores for a stream.

Figures 3.4a and 3.4b present the mean RTE for ITTD5 only at the steps where rounding is performed and do not connect the data points with dashed line. This is caused by the fact that without reorthogonalization, TT-cores obtained through ITTD are not suitable for

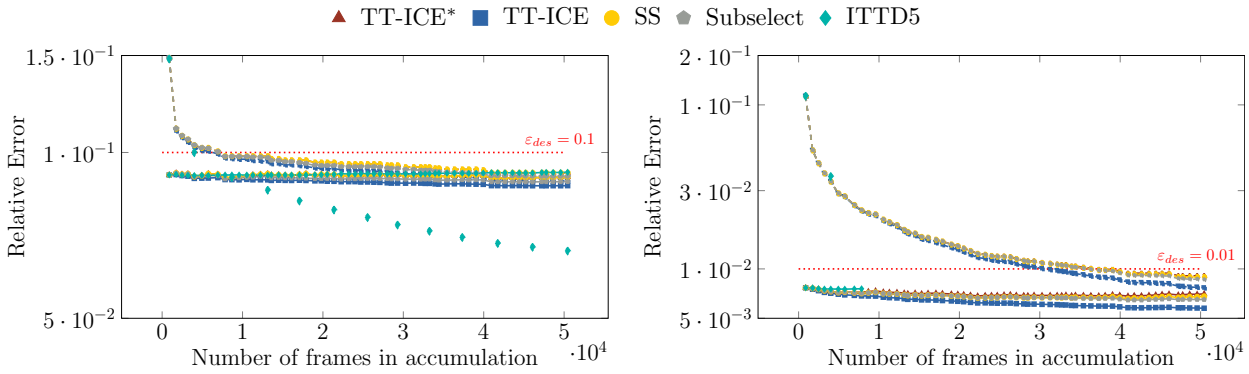


(a) $\varepsilon_{des} = 0.1$ (b) $\varepsilon_{des} = 0.01$

Figure 3.3: Comparison of execution time vs total number of observations in the tensor train for different versions of TT-ICE, TT-ICE* and ITTD[161] with two different ε settings. TT-ICE*: full TT-ICE* algorithm with all heuristics, TT-ICE: TT-ICE, Subselect: subselecting frames from runs, SS: subselecting frames from runs and skip updating the first d TT-cores, ITTD 5: ITTD with rounding at every fifth increment step. ITTD has worst compression time while TT-ICE* with all heuristics performs best. ITTD Fails to compress the entire stream for $\varepsilon_{des} = 0.01$.

assimilating data (projection and prediction). This is an artifact caused by implementing addition in TT-format and the uncontrolled rank inflation and lack of core orthogonality. However, an interesting point from Figure 3.4a is that the mean RTE falls well below the mean RRE for ITTD5 after reorthogonalization. The results for TT-ICE and TT-ICE* are similar for $\varepsilon_{des} = 0.01$ in Figure 3.4b. Unfortunately, ITTD5 cannot display the same pattern in this case, since it terminates prematurely due to insufficient memory.

Finally, Figures 3.5a and 3.5b indicate differences in compression ratio between algorithms. Figure 3.5a shows that the TT-ICE variations have comparable performance to each other. Furthermore, Figure 3.5a shows selecting any TT-ICE variation results in almost two orders of magnitude higher compression over ITTD5. This significantly lower compression ratio of ITTD also explains its reduction in mean RTE after TT-rounding. The superfluous increase in TT-ranks allows the TT-cores trained with ITTD to cover a much larger portion of the multidimensional basis. Right after reorthogonalization, this larger basis provides an increase in the generalization capability of the TT-cores but also results in a much lower compression ratio. The rounding step also provides a positive jump in the compression ratio, but when the TT-cores are incremented further with ITTD, this improvement decays quickly. When the relative error tolerance changes to $\varepsilon_{des} = 0.01$, the low compression ratio problem for ITTD5 is exacerbated where the compression ratio falls below the critical value of 1 for ITTD5. Having a compression ratio lower than 1 means that TT -cores have more entries than the original accumulation. Despite having a much lower compression ratio than



(a) $\varepsilon_{des} = 0.1$ (Dotted line)

(b) $\varepsilon_{des} = 0.01$ (Dotted line)

Figure 3.4: Comparison of mean RRE and mean RTE vs total number of observations in the tensor train for different versions of TT-ICE, TT-ICE* and ITTD[161] with two different ε settings. We present the mean RRE over compressed runs in the accumulation and mean RTE over the validation set after each increment step. Data points connected with a solid line represent the mean RRE and data points connected with a dashed line represent the mean RTE. Please refer to Figure 3.3 for a description of the legend. For $\varepsilon_{des} = 0.1$ TT-cores returned from ITTD have much lower RTE than desired. RTE of TT-cores trained with TT-ICE and TT-ICE* converge to RRE as accumulation size increases. ITTD Fails to compress the entire stream for $\varepsilon_{des} = 0.01$.

the case with $\varepsilon_{des} = 0.1$, all of the TT-ICE variations again perform comparably with a compression ratio around 3.5×.

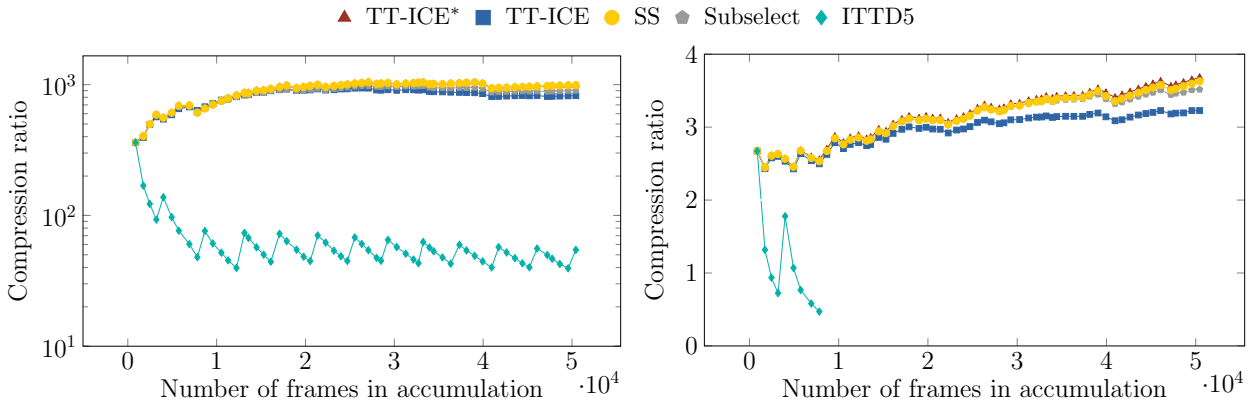
To summarize, the investigations yielded a decrease in compression time, an increase in compression ratio, and improved execution stability when TT-ICE is preferred over ITTD. Furthermore, investigations show an additional decrease in compression time and an increase in compression ratio when TT-ICE* is preferred over TT-ICE.

3.1.4.2 Self-Oscillating Gel Simulations

This section includes tests of TT-ICE's performance to compress simulation outputs of high-dimensional **P**artial **D**ifferential **E**quation (PDE)s.

When high-dimensional systems are of interest, the size of the simulation outputs can become prohibitive. In the extreme, even the storage of the data may not be feasible. Streaming compression algorithms can become essential when the outputs of a dynamical simulation become sequentially available. Moreover, tensor decompositions can extract more information than incremental matrix decomposition methods to be used to learn low-dimensional representations, for example, for inverse design [5].

The second dataset arises from solutions of a parametric PDE that simulates the motion of a hexagonal sheet of self-oscillating gels. The motion of the gel is governed by the following



(a) $\varepsilon_{des} = 0.1$ (b) $\varepsilon_{des} = 0.01$

Figure 3.5: Comparison of compression ratio vs total number of observations in the tensor train for different versions of TT-ICE, TT-ICE*, and ITTD[161] with two different ε settings. Please refer to Figure 3.3 for a description of the legend. ITTD has the worst compression ratio while TT-ICE* with all heuristics performs the best. ITTD Fails to compress the entire stream for $\varepsilon_{des} = 0.01$.

time-dependent parametric PDE

$$\mu \frac{\partial r}{\partial t} = f_s\left(r, \mathbf{K_s}, \eta\right) + f_B\left(r\right), \quad \eta\left(x, y, t, \mathbf{A}, \mathbf{k}\right) = 1 + \mathbf{A}\sin\left(2\pi\left(\mathbf{k}\sqrt{x^2+y^2} - t\right)\right), \quad (3.38)$$

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.

We uniformly sample from this 3-dimensional space of input parameters to obtain 6400 unique parameter combinations and then simulate each of those parameter combinations using the approach in [9]. 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 each time snapshot as shown in Figure 3.2b.

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. For compression experiments, each simulation is reshaped into tensors of size $7 \times 13 \times 37 \times 3 \times 10 \times 1$ and stacked along the sixth dimension to obtain the accumulation. We conducted all the experiments

on M3 machine that has a Xeon Silver 4110 processor and 16GB memory. Figures 3.6 to 3.8 show the results of those experiments. The occupancy threshold of TT-ICE* is set to 1 for this dataset. This prevents update attempts when the basis for one dimension is fully explored (i.e. the TT-core has full rank). Finally, we compute the mean of RRE over compressed simulations in the accumulation.

Despite the suitability of TT-FOA to this tensor stream, we can't provide any results for TT-FOA. Even when we assume that TT-FOA is initialized with the final ranks of TT-ICE* with $\varepsilon_{des} = 0.1$, the memory required by the auxiliary matrices, matrix inversions, and Kronecker products in the algorithm exceeds the memory of M3 machine and fails at the first step[1].

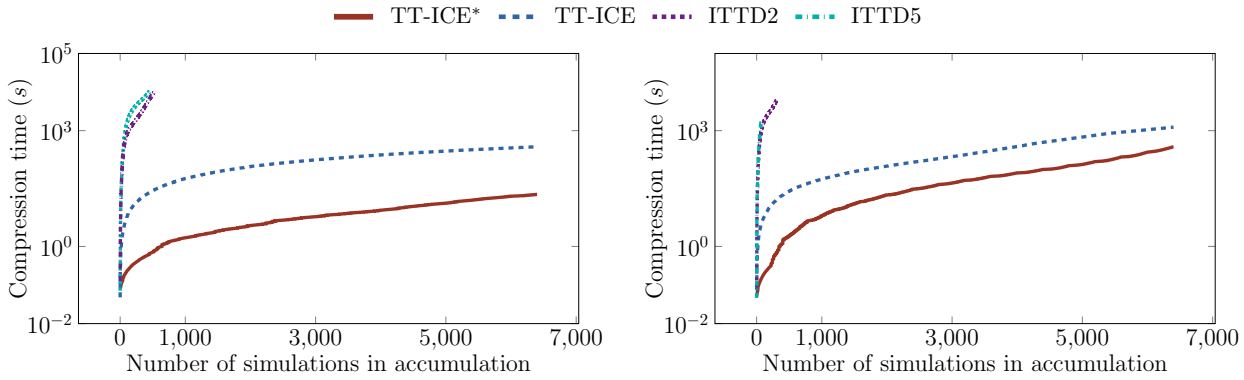
Compression results This section includes the results of experiments on self-oscillating gel simulation snapshots.

Unlike the previous set of experiments involving ATARI gameplay frames, ITTD methods fail to compress the entire stream in both ε_{des} settings due to insufficient memory while attempting TT-rounding. In all scenarios, TT-ICE* outperforms TT-ICE, but both algorithms are able to compress the entire stream successfully.

Figures 3.6a and 3.6b depict the differences in compression speed between algorithms. Figure 3.6a shows that it takes more time for both ITTD2 and ITTD5 to compress 1/20 of the entire stream than for TT-ICE and TT-ICE* to compress the entire stream. Figure 3.6a also clearly illustrates the benefits of implementing heuristic upgrades, since TT-ICE* compresses the entire stream an order of magnitude faster than TT-ICE. This difference in compression time becomes less in Figure 3.6b with $\varepsilon_{des} = 0.01$ but TT-ICE* still provides a significant reduction in time in comparison to TT-ICE. At $\varepsilon_{des} = 0.01$, ITTD methods fail even earlier than before and compress at most 1/50 of the entire stream.

Figures 3.7a and 3.7b present the differences in the compression ratio between algorithms. Figure 3.8a shows that both TT-ICE and TT-ICE* achieve excessive compression close to $10^4\times$ in the early stages of the stream and then exhibit a decay in compression ratio. This behavior is related to the streaming order of the simulations and is expected. Towards the beginning, the stream consists of simulations with similar parameter combinations. This leads to simulations exhibiting similar motion patterns and allows the accumulation to be represented with a small basis. Then, as the stream progresses, the accumulation consists of simulations from a greater variety of parameter combinations and calls for an expansion in the bases. The same pattern repeats itself in Figure 3.7b for $\varepsilon_{des} = 0.01$, but this time the

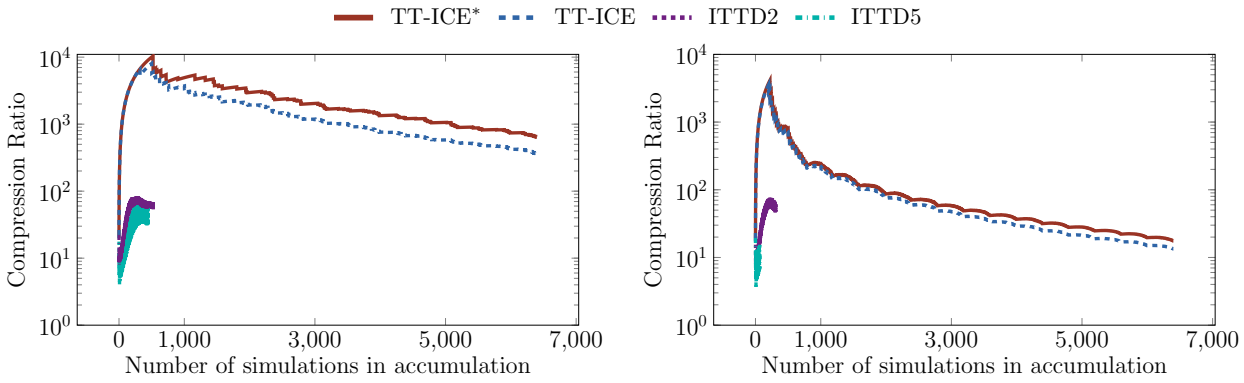
[1]When we repeat the same experiment with M1 machine, which has the same processor with M3 and has higher memory, computing one step of the stream takes more than $1500s$.



(a) $\varepsilon_{des} = 0.1$ (b) $\varepsilon_{des} = 0.01$

Figure 3.6: Comparison of execution time vs total number of simulations in the tensor train for Algorithm 1, Algorithm 2 and ITTD[161] with two different ε settings. TT-ICE* outperforms TT-ICE by one order of magnitude. ITTD Fails to compress the entire stream for both cases. TT-ICE*: full TT-ICE* with all heuristics defined in Section 3.1.2.3, TT-ICE: TT-ICE algorithm (Algorithm 1), ITTD5: ITTD with rounding at every fifth increment step, ITTD2: ITTD with rounding at every second increment step.

decay is greater than it was for $\varepsilon_{des} = 0.1$. Since both ITTD methods fail to compress the entire stream, we can not make meaningful comments on their compression performance. However, Figures 3.7a and 3.7b indicate that the peak compression achieved is consistently two orders of magnitude less than that achieved by the TT-ICE methods.

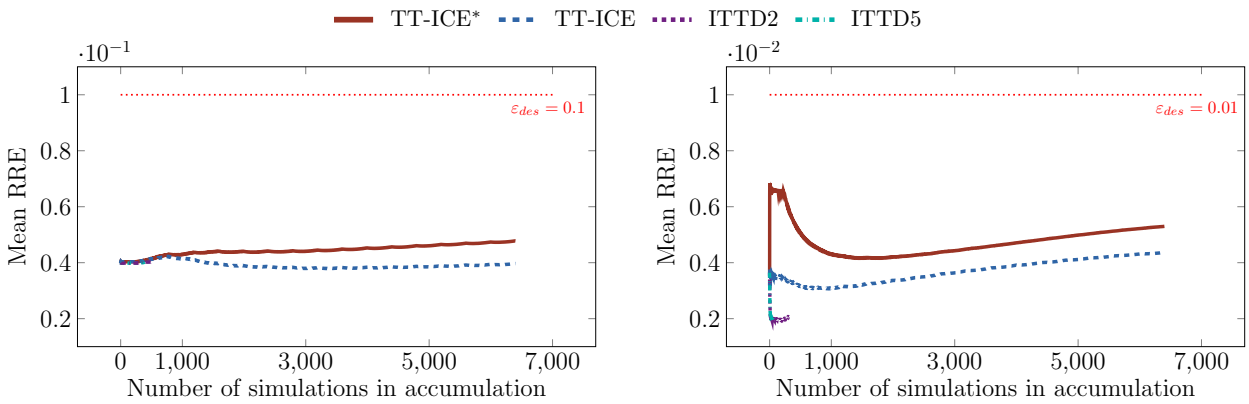


(a) $\varepsilon_{des} = 0.1$ (b) $\varepsilon_{des} = 0.01$

Figure 3.7: Comparison of compression ratio vs total number of simulations in the tensor train for Algorithm 1, Algorithm 2 and ITTD[161] with two different ε settings. Please refer to Figure 3.6 for a description of the legend. TT-ICE* outperforms TT-ICE* but both methods show comparable compression performances. ITTD Fails to compress the entire stream for both cases.

Finally, Figures 3.8a and 3.8b indicate differences in reconstruction error between al-

gorithms. Figure 3.8a shows that TT-ICE* has a mean RRE slightly closer to ε_{des} than TT-ICE. On the other hand, Figure 3.8b shows that TT-ICE* has nearly twice the error of TT-ICE, but this difference in mean RRE diminishes to almost a constant offset as the stream progresses. In both Figure 3.8a and Figure 3.8b, ITTD methods start with lower mean RRE values. This can be explained by the significantly lower compression performance of these methods, where the higher coverage in the bases results in both lower compression and lower mean RRE. However, we cannot draw meaningful conclusions for ITTD since both ITTD2 and ITTD5 fail prematurely.



(a) $\varepsilon = 0.1$ (b) $\varepsilon_{des} = 0.01$

Figure 3.8: Comparison of mean RRE vs total number of simulations in the tensor train for TT-ICE, TT-ICE* and ITTD[161] with two different ε settings. Please refer to Figure 3.6 for a description of the legend. TT-ICE* provides mean RRE closer to ε_{des} in both cases. ITTD Fails to compress the entire stream for both cases.

Mean RRE tests conclude the experiments with self-oscillating gel simulations. Similar to experiments with ATARI data, TT-ICE and TT-ICE* algorithms yield reduced compression time, increased compression ratio, and improved execution stability over ITTD.

3.2 Batch Hierarchical Tucker format and the BHT-l2r algorithm

This section presents the second contribution of this thesis, which is the batch hierarchical Tucker format and the BHT-l2r algorithm that computes a low-rank approximation of a batch of tensors in this format. The algorithm is designed to provide a basis for incremental updates of the hierarchical Tucker representation as new batches of tensors become available. The algorithm is based on the leaves-to-root decomposition of the hierarchical Tucker format,

which is adapted to handle batches of tensors. The batch hierarchical Tucker format and the BHT-l2r algorithm are essential components of the HT-RISE algorithm presented in Section 3.3.

We formally define the problem we aim to tackle in this section as follows:

Problem 2. *(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.

3.2.1 Details on the Hierarchical Tucker Format

Please note that this section contains some duplicate information on the hierarchical Tucker format that is already presented in Section 2.2.4. However, we include it here for completeness and to provide a self-contained description of the hierarchical Tucker format, which is essential for understanding the batch hierarchical Tucker format and the BHT-l2r algorithm.

As its name suggests, the hierarchical Tucker (HT) format is a tensor network format that generalizes the Tucker format [260]. Recall that, 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 core tensor $\mathcal{C} \in \mathbb{R}^{\mathbf{r}_1 \times \cdots \times \mathbf{r}_d}$ and d factor matrices $U_i \in \mathbb{R}^{n_i \times \mathbf{r}_i}$, where $\mathbf{r}_i$ are the Tucker ranks,

$$\mathcal{Y} = \mathcal{C} \times [\![U_1, \ldots, U_d]\!]. \tag{3.39}$$

The Tucker representation reduces the storage from $\mathcal{O}\left(n^d\right)$ to $\mathcal{O}\left(dn\mathbf{r} + \mathbf{r}^d\right)$, where $\mathbf{r}_i$ are potentially much smaller than n_i.

However, 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 the structure of the Tucker core. Such structure can be exploited via a recursive Tucker decomposition. In [192], the resulting format is named as *tree Tucker*, and in [90] 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 (3.39) is reshaped into a tensor with

Please note that Section 3.2 contains slightly modified parts of the manuscript "Incremental Hierarchical Tucker Decomposition" [6] currently under review at SIAM Journal on Scientific Computing.

$d/2$ 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 2.7 illustrates various configurations of dimension trees for a five-dimensional tensor. 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 B.1.2). Figure 2.7 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 $\mathbf{T}$. Recall the illuminating example of root-to-leaves contraction from Section 2.2.4. 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

$$\mathcal{Y} = (\cdots((\mathcal{G}_{0,1} \times [\![\mathcal{G}_{1,1}, \mathcal{G}_{1,1}]\!]) \times [\![\mathcal{G}_{2,1}, \mathcal{G}_{2,2}, \mathcal{G}_{2,3}, \mathcal{G}_{2,4}]\!]) \times \cdots) \times [\![\mathcal{G}_{p,1}, \ldots, \mathcal{G}_{p,|\mathbf{T}_p|}]\!]. \tag{3.40}$$

3.2.1.1 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 [90, Alg. 2], and 2) root-to-leaves decomposition [141, 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,

$$Y_{(d_{p,i})} = U_{p,i}\Sigma_{p,i}V_{p,i}^T + E_{p,i} \quad \|E_{p,i}\|_F \leq \varepsilon_{nw}, \quad i = 1, \ldots, |\mathbf{T}_p| \tag{3.41}$$

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 (3.41) is repeated for all leaves on layer p, the orthogonal $U_{p,i}$ matrices are contracted with $\mathcal{Y}$ along the dimensions referred to by the leaves as

$$\bar{\mathcal{C}}_p = \mathcal{Y} \underset{\mathcal{I}}{\times} [\![U_{p,1}^T, \ldots, U_{p,|\mathbf{T}_p|}^T]\!], \text{ where } \quad \mathcal{I} = \{d_{p,i}; i = 1, \ldots, |\mathbf{T}_p|\}, \tag{3.42}$$

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 [266] 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

$$\mathbf{I}_{\mathbf{T}_{p-1}} = \bigcup_{j=1}^{|\mathbf{T}_{p-1}|} \begin{cases} n_{d_{p-1,j}} & \text{if } \mathbf{N}_{p-1,j} \text{ is a leaf} \\ \prod_{(p,m)\in\mathbf{S}_{p-1,j}} r_{p,m} & \text{else.} \end{cases}, \quad \text{so that} \quad \mathcal{C}_{p-1} = \texttt{reshape}\left(\bar{\mathcal{C}}_p,\ \mathbf{I}_{\mathbf{T}_{p-1}}\right). \tag{3.43}$$

Once we obtain $\mathcal{C}_{p-1}$, the error truncated `SVD` analogous to (3.41) 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

$$\mathcal{G}_{\ell,i} = \texttt{reshape}\left(U_{\ell,i}, \mathbf{I}_{\mathbf{N}_{\ell,i}}\right), \quad \text{where } \mathbf{I}_{\mathbf{N}_{\ell,i}} = \left\{\bigcup_{(\ell+1,j)\in\mathbf{S}_{\ell,i}} r_{\ell+1,j}\right\} \cup r_{\ell,i}. \quad i = 1, \ldots, |\mathbf{T}_\ell|, \tag{3.44}$$

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 $\mathcal{C}_1 \in \mathbb{R}^{r_{1,1}\times r_{1,2}}$

$$\mathcal{C}_1 = U_1 \Sigma_1 V_1^T + E_1, \tag{3.45}$$

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 3 (Adapted from [141] 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*

$$\|E_i\|_F \leq \varepsilon_{nw} \tag{3.46}$$

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.

3.2.2 Batch Hierarchical Tucker Format

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 2.6. 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 2.8. 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 (3.42), 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 Figures 2.6 and 2.8. Figure 3.9 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 nodes are still three-dimensional and leaf-nodes remain two-dimensional. The pseudocode of the corresponding algorithm BHT-l2r is presented in Algorithm 3. Theoretical guarantees for the approximation error upper bound for the BHT-l2r algorithm essentially follows from Theorem 3 and is presented as Corollary 2 in Section 3.2.2.1[4]. Section 3.2.3 presents a detailed comparison between HT and BHT formats on both scientific and image data.

[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.

[4]Please refer to the Appendix C.1 for proofs of theorems.

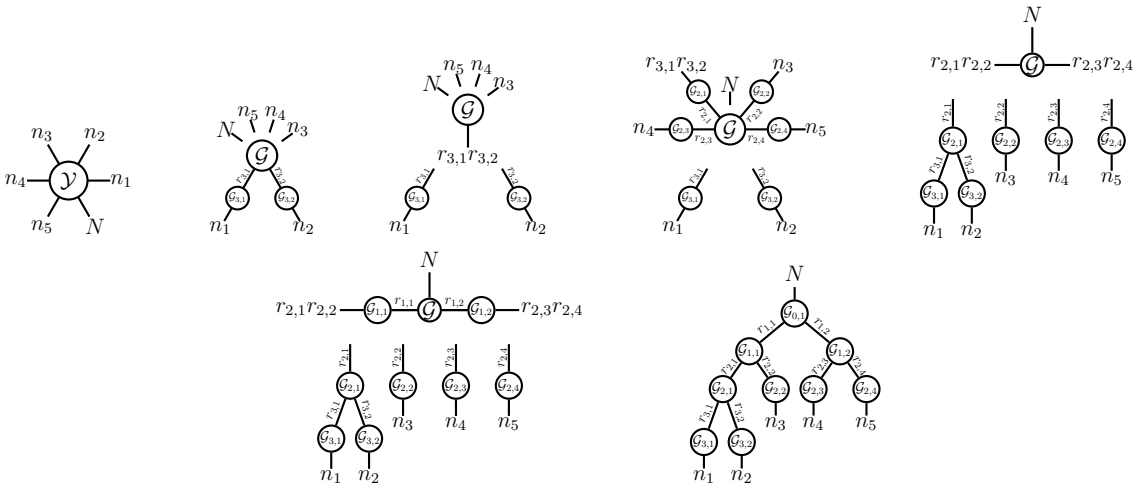


Figure 3.9: Step-by-step decomposition of an N-batch of 5-D tensors with the BHT-l2r algorithm (Algorithm 3). The decomposition starts with the leaves of the last layer. Since the dimension tree is constructed beforehand, Algorithm 3 has the information about which dimensions' leaves will be on which layer through the dimension tree $\mathbf{T}$. Note that the batch dimension (N) remains intact throughout the entire decomposition process.

3.2.2.1 Approximation Error of the BHT-l2r Algorithm

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 1 (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.

Lemma 1 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 4 (Total approximation error [90, 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* $\sum_{\ell=0}^{p}\sum_{i=1}^{|\mathbf{T}_\ell|}\|E_{\ell,i}\|_F^2$,

$$\|\mathcal{Y}-\tilde{\mathcal{Y}}\|_F^2 \leq \sum_{\ell=0}^{p}\sum_{i=1}^{|\mathbf{T}_\ell|}\|E_{\ell,i}\|_F^2, \tag{3.47}$$

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

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 2 (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 3 with an absolute error tolerance* ε_{abs}*, cores* $\mathcal{G}$*, and dimension tree* $\mathbf{T}$*. Then, the reconstruction of this approximation using Equation* (3.40)*,* $\tilde{\mathcal{Y}}$*, satisfies* $\|\mathcal{Y}-\tilde{\mathcal{Y}}\|_F \leq \varepsilon_{abs}$ *if a node-wise error tolerance* ε_{nw} *is chosen such that*

$$\varepsilon_{nw} = \frac{\varepsilon_{abs}}{\sqrt{2d-2}}. \tag{3.48}$$

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 (3.49) 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,*

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

3.2.3 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 2.6) using the BHT-l2r algorithm (Algorithm 3) and the hierarchical Tucker format (Figure 2.8) using the leaf-to-root compression algorithm of [90, Alg. 2]. 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 ran all algorithms until they run out of the allocated 64 GB memory.

We conduct our experiments using the PDEBench data set with $\varepsilon_{rel} = 0.10$ and 0.05 as well as the BigEarthNet data set with $\varepsilon_{rel} = 0.05$, 0.10, 0.15, and 0.30 with the same reshapings as in Section 3.1.3. 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 [90, Alg. 2] while returning better compression ratios. Figures 3.10 and 3.11, and Figure 3.12 show the results for the PDEBench and BigEarthNet data sets, respectively.

Figure 3.10 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 interesting observation from Figure 3.10 is that the BHT format results in faster compression irrespective of the normalization method used.

Figure 3.11 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 $\varepsilon_{rel} = 0.10$, the BHT format results in faster compression irrespective of the normalization method used.

In addition to the 3D Navier-Stokes simulations, we also conduct experiments on the BigEarthNet data set. Figure 3.12 shows the results of the comparison between HT format and BHT format in CR and compression time for the BigEarthNet data set. In line with our findings from the PDEBench data set, 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× 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× compression over the HT format. At the lower target relative

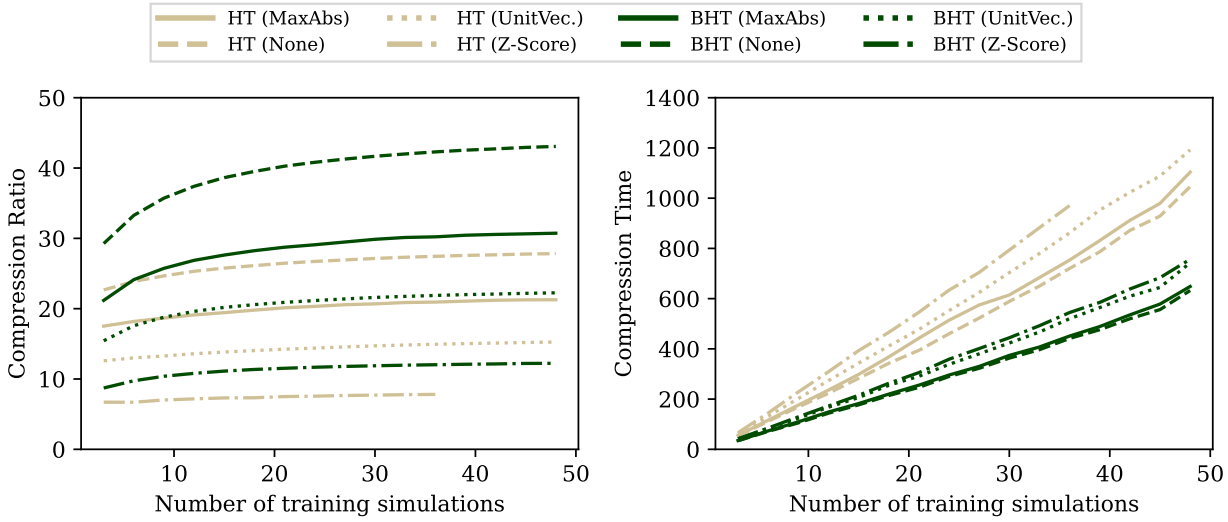


Figure 3.10: Comparison of hierarchical Tucker leaf-to-root compression and batch hierarchical Tucker decomposition in terms of compression ratio (CR - left) and compression time (right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using various normalization methods. For the same normalization methods, the batch hierarchical Tucker format achieves on average 1.5× compression over the hierarchical Tucker format, while running on average 1.7× 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.

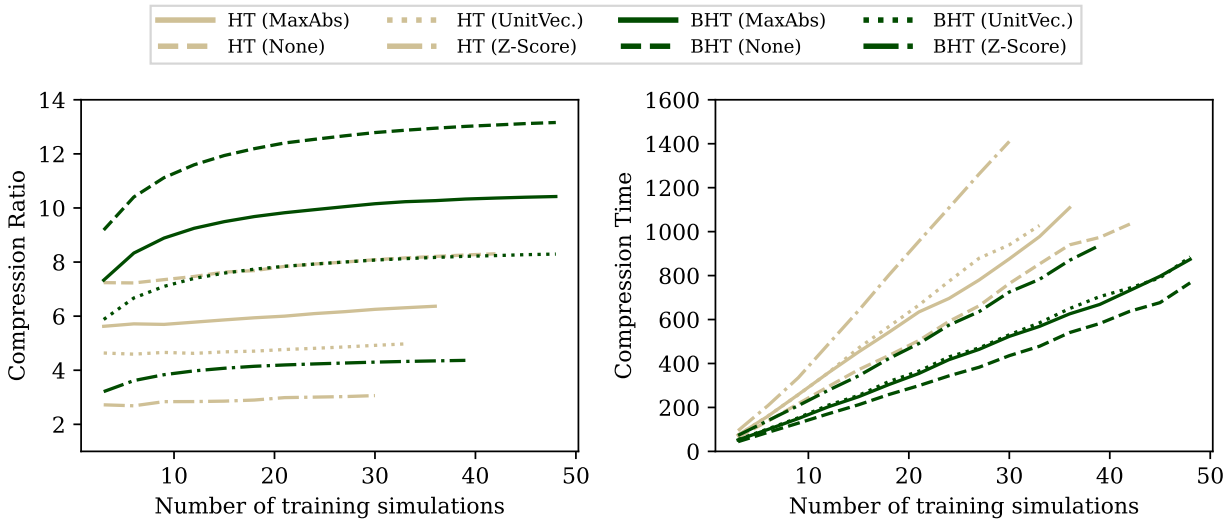


Figure 3.11: Comparison of HT format and BHT format in terms of compression ratio (CR - left) and compression time (right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.05$ using various normalization methods. For the same normalization methods, the BHT format achieves $1.4 - 1.6\times$ 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 3.10 for the normalization methods.

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 3.12 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 $\varepsilon_{rel} = 0.05$.

3.3 The HT-RISE Algorithm

This section presents the third contribution of this thesis, which is an incremental algorithm for updating an existing batch hierarchical Tucker representation of an accumulated tensor

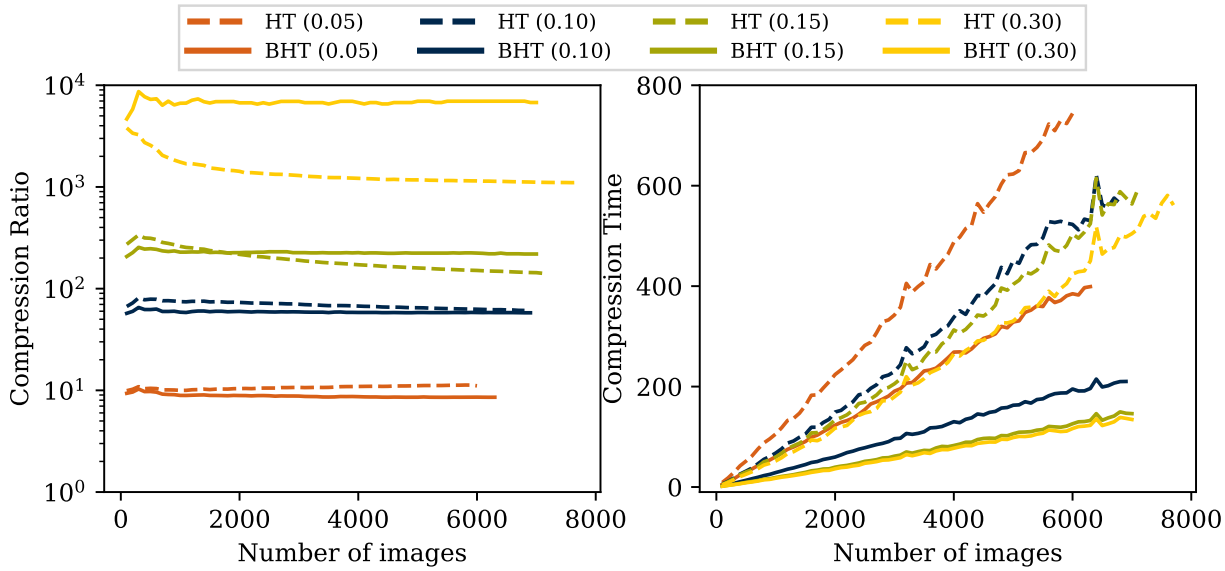


Figure 3.12: Comparison of hierarchical Tucker format and batch hierarchical Tucker format in terms of compression ratio (CR - left) and compression time (right) on the BigEarthNet data set with $\varepsilon_{rel} = 0.05 - 0.30$. The BHT format achieves 6.2× compression over the HT format at $\varepsilon_{rel} = 0.3$ while running 3.75× faster. At $\varepsilon_{rel} = 0.1$, the BHT format achieves 0.95× compression over the HT format while running 2.69× 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.

as new batches of tensors become available. The algorithm is called HT-RISE and its flow is illustrated in Figure 3.13. To the best of our knowledge, this is the first algorithm that can perform incremental updates in the hierarchical Tucker format.

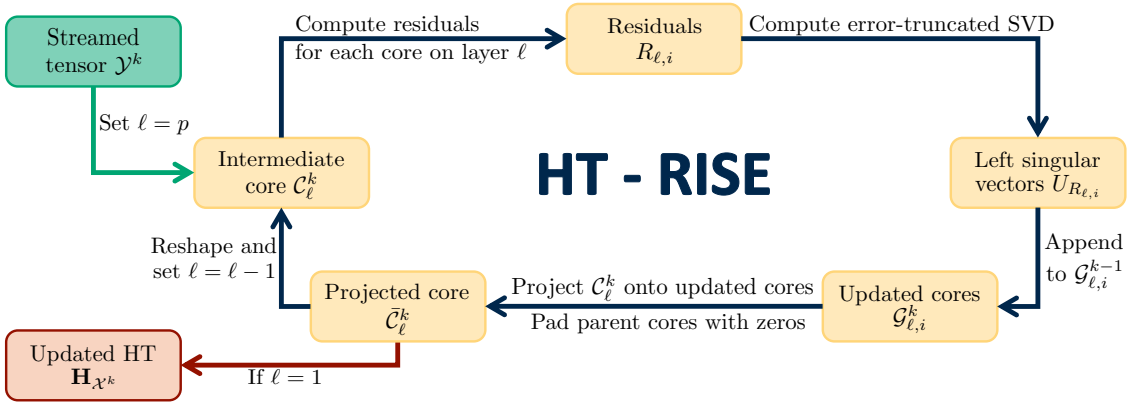


Figure 3.13: Flow of the proposed HT-RISE algorithm. The algorithm updates the hierarchical Tucker representation of an accumulated tensor in batch hierarchical Tucker form as new batches of tensors become available. Green represents the input data and red represents the output data.

Formally, the problem of updating an existing hierarchical Tucker representation as new batches of tensors become available can be stated as follows:

Problem 3. *(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 $\hat{\mathcal{X}}^k$.*

In this section we propose a solution to Problem 3 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 $\mathcal{G}_{\ell,i_\ell}^{k-1}$ into a new core $\mathcal{G}_{\ell,i_\ell}^k$, assuming an unchanged dimension tree $\mathbf{T}$. The proposed approach has three components: 1) projection onto existing HT cores 2) compute residuals, and 3) update HT cores. The overall algorithm, HT-RISE, is provided in Algorithm 4 and each step is described in detail next.

Please note that Section 3.3 contains slightly modified parts of the manuscript "Incremental Hierarchical Tucker Decomposition" [6] currently under review at SIAM Journal on Scientific Computing.

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 reshapings thereof) when constructed using Algorithm 3. 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 4 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[5]. 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

$$\varepsilon_{proj} = \sqrt{\|\mathcal{Y}^k\|_F^2 - \|\bar{\mathcal{C}}_1^k\|_F^2}. \tag{3.49}$$

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 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

$$R_{\ell,j} = \Pi_{\ell,j}^{k-1} C_{\ell,(j)}^k, \quad \text{where} \quad \Pi_{\ell,j}^{k-1} = I - U_{\ell,j}^{k-1} \left(U_{\ell,j}^{k-1}\right)^T \text{with } U_{\ell,j}^{k-1} = \begin{cases} \mathcal{G}_{\ell,j}^{k-1} & \text{if } \mathbf{N}_{\ell,j} \text{ is a leaf} \\ \texttt{reshape}\left(\mathcal{G}_{\ell,j}^k, [\alpha, r_{\ell,j}^{k-1}]\right) & \text{else} \end{cases}, \tag{3.50}$$

where $\Pi_{\ell,j}^{k-1}$ is the projection operator and $\alpha = \frac{1}{r_j^\ell} \prod_{\gamma \in \mathbf{I}_{\mathbf{N}_j^\ell}^{k-1}} \gamma$. 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.

[5]Please refer to Claim 1 for details and proof.

Step 3: Performing core updates The idea behind incremental updates is similar to [7]. 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

$$R_{\ell,j} = U^k_{R_{\ell,j}}\Sigma^k_{R_{\ell,j}}(V^k_{R_{\ell,j}})^T + E_{R_{\ell,j}}, \tag{3.51}$$

such that $U^k_{R_{\ell,j}} \in \mathbb{R}^{\alpha\times r_{R_{\ell,j}}}$, $\Sigma^k_{R_{\ell,j}} \in \mathbb{R}^{r_{R_{\ell,j}}\times r_{R_{\ell,j}}}$, and $V^k_{R_{\ell,j}} \in \mathbb{R}^{\beta\times r_{R_{p,1}}}$. 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 D.2.

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

$$U^k_{\ell,j} = \begin{bmatrix} U^{k-1}_{\ell,j} & U^k_{R_{\ell,j}} \end{bmatrix}, \tag{3.52}$$

with $U^k_{\ell,j} \in \mathbb{R}^{\alpha\times r^k_{\ell,j}}$, such that $r^k_{\ell,j} = r^{k-1}_{\ell,j} + r_{R_{\ell,j}}$. An update of the index set $\mathbf{I}^k_{\mathbf{N}_{\ell,j}}$ using (3.44) follows (3.52) to reflect the updated rank $r^k_{\ell,j}$. The new basis $U^k_{\ell,j}$ can then be appropriately reshaped into transfer or leaf cores in the same manner as for the HT described in Section 3.2.1.1. The new core $U^k_{\ell,j}$ is then reshaped appropriately to form $\mathcal{G}^k_{\ell,j}$.

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

$$\mathcal{G}^{k-1}_{\ell-1,t} = \mathcal{G}^{k-1}_{\ell-1,t} \overset{1}{\oplus} \mathbf{0}_{r_{R_{\ell,1}}\times r^{k-1}_{\ell,m}\times r^{k-1}_{\ell-1,t}}, \quad \text{or} \quad \mathcal{G}^{k-1}_{\ell-1,t} = \mathcal{G}^{k-1}_{\ell-1,t} \overset{2}{\oplus} \mathbf{0}_{r^{k-1}_{\ell,j}\times r_{R_{\ell,1}}\times r^{k-1}_{\ell-1,t}}, \tag{3.53}$$

depending on the order of $\mathcal{G}^{k-1}_{\ell,j}$ in the set of successors of $\mathcal{G}^{k-1}_{\ell-1,t}$. Following (3.53), the index set of $\mathbf{N}_{\ell-1,t}$ is updated using (3.44) 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 $\mathcal{G}^{k-1}_{\ell,j}$ are updated to $\mathcal{G}^k_{\ell,j}$ for $j = 1,\ldots,|\mathbf{T}_\ell|$, the index set of $(\ell-1)$-th layer is updated to $\mathbf{I}^k_{\mathbf{T}_{\ell-1}}$ using (3.43) 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

$$\bar{\mathcal{C}}_\ell^k = \mathcal{C}_\ell^k \times [\![\mathcal{G}_{\ell,1}^k, \ldots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|}^k]\!]. \tag{3.54}$$

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 $\mathcal{G}_0^{k-1}$ as

$$\mathcal{G}_0^k = \mathcal{G}_0^{k-1} \overset{3}{\oplus} \bar{\mathcal{C}}_1^k, \quad \text{where} \quad \bar{\mathcal{C}}_1^k = \mathcal{C}_1^k \times [\![\mathcal{G}_{1,1}^k, \mathcal{G}_{1,2}^k]\!], \tag{3.55}$$

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 4 as HT-RISE. Furthermore, a python implementation of HT-RISE is available on GitHub.

Analysis of approximation error guarantees 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. The next theorem provides a proof of correctness for the HT-RISE algorithm.

Theorem 5 (Approximation error bound for the HT-RISE algorithm)**.** *The HT-RISE algorithm maintains the approximation error bound* $\|\mathcal{Y}^k - \hat{\mathcal{Y}}^k\|_F \leq \varepsilon_{des}$ *for all* k*, where* $\hat{\mathcal{Y}}^k$ *is the approximation of the streamed tensor at time* k *in hierarchical Tucker format.*

Since Theorem 5 tells 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 for HT-RISE)**.** *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 4 (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.,* $\tilde{\mathcal{X}}_m^k = \tilde{\mathcal{X}}_m^t$*.*

3.3.1 Numerical Experiments

In this section, we compare the proposed HT-RISE algorithm against a state-of-the-art incremental tensor train decomposition algorithm, `TT-ICE`* [7] on scientific simulation and image datasets. Scientific datasets include the PDEBench dataset [251] of compressible Navier-Stokes simulations and a second dataset of a chaotic PDE system modeling self-oscillating gels [9]. Image datasets use Minecraft video frames from MineRL Basalt competition dataset [178] and multispectral images from the BigEarthNet dataset [244, 245]. Table 3.1 summarizes the datasets used in the experiments. We also include further analyses including the effect of tensor reshapings (Appendix B.1.1) and axis reorderings (Appendix B.1.2) in the supplementary materials.

Table 3.1: Summary of the data sets 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 data set 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

All experiments are executed on University of Michigan compute nodes on the Lighthouse cluster using 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 [94]. We use `TT-ICE`* from GitHub for our experiments.

3.3.1.1 Performance Metrics

This section presents the accuracy, compression ratio, reduction ratio, execution time, and generalization performance metrics used for comparisons.

Compression and Reduction ratio (CR/RR) The Compression Ratio (CR) and reduction ratio (RR) are

$$CR = \frac{\texttt{num_elem}(\mathcal{X}^k)}{\texttt{num_elem}(\mathbf{H}_{\mathcal{X}^k})} \quad \text{and} \quad RR = \frac{\texttt{num_elem}(\mathcal{Y}^{k,i})}{\texttt{num_elem}(\texttt{encode}(\mathcal{Y}^{k,i}))}, \tag{3.56}$$

respectively. Here, $\texttt{num_elem}(\mathcal{X}^k)$ denotes the number of elements of the original accumulation; $\texttt{num_elem}(\mathbf{H}_{\mathcal{X}^k})$ is the sum of number of elements in all HT cores; $\mathcal{Y}^{k,i}$ 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., $\texttt{TT-ICE}^*$, HT-RISE, etc..

A compression ratio of 1 indicates that the approximation does not compress the original accumulation at all, while a compression ratio below 1 indicates an inefficient approximation that uses more elements to approximate the original accumulation. The RR measures the ratio of the size of a single tensor in the accumulation to the size of the latent space. The RR is particularly useful for downstream learning and prediction tasks, see e.g., [49], for more details.

3.3.1.2 Compression Time and Walltime

Compression time is a measure of how long it takes to update the approximation – it does not consider the time to load and preprocess the data, update the approximation, and check the approximation error on the test set. To account for these aspects we also set a maximum walltime for each experiment. If an experiment exceeds the maximum walltime, the experiment is terminated and the metrics at the time of termination are reported.

3.3.1.3 Relative Test Error

The relative test error (RTE) measures the quality of generalization to unseen (test) data — it is measured using the relative approximation error of tensors averaged over the test set

$$RTE = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} \frac{\|\mathcal{Y}^i_{\text{test},k} - \tilde{\mathcal{Y}}^i_{\text{test},k}\|_F}{\|\mathcal{Y}^i_{\text{test},k}\|_F}, \tag{3.57}$$

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 $\mathbf{H}_{\mathcal{X}^k}$. The RTE is averaged over individual tensors in the test set, which can lead to higher RTE values than the target ε_{rel} that was used to inform batches. The RTE is recomputed if the approximation is updated with streamed tensor, and frequent updating can cause the RTE computation to become a dominant part of the walltime cost.

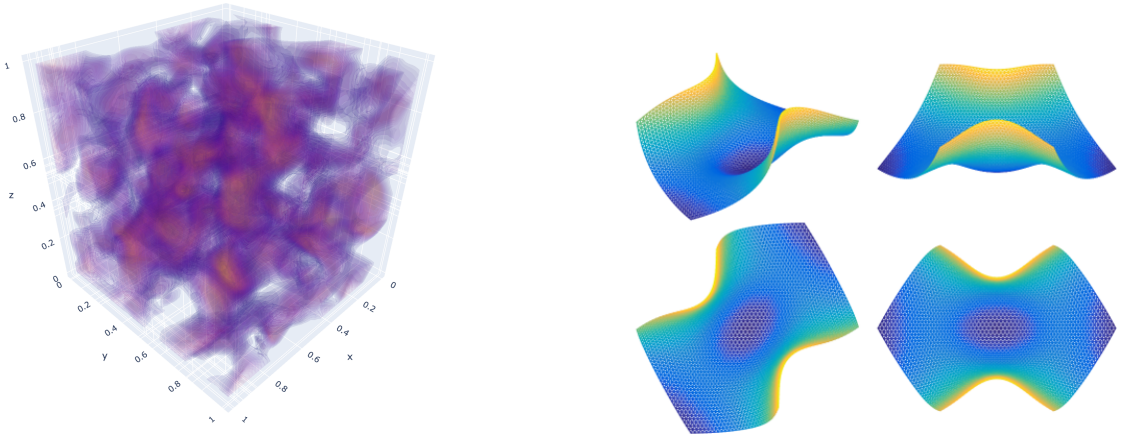
3.3.2 Numerical Experiments with Scientific Data

This section reports HT-RISE's performance on the scientific simulation datasets. Both the oscillating-gels and the PDEBench datasets consist of simulation snapshots. Each snapshot

contains states corresponding to various physical quantities (e.g. displacement, velocity, pressure, density etc.). To ensure appropriate compression, it is often useful to normalize each physical quantity individually similar to [8]. Here, we consider maximum absolute value ($\mathcal{Y}_{\text{maxabs}}$), unit vector ($\mathcal{Y}_{\text{unitvec}}$), and z-score ($\mathcal{Y}_{\text{z-score}}$) normalizations:

$$\mathcal{Y}_{\text{maxabs}} = \frac{\mathcal{Y}}{\max(|\mathcal{Y}|)}, \qquad \mathcal{Y}_{\text{unitvec}} = \frac{\mathcal{Y}}{\|\mathcal{Y}\|}, \qquad \mathcal{Y}_{\text{z-score}} = \frac{\mathcal{Y} - \mu}{\sigma}, \tag{3.58}$$

where $\mathcal{Y}$ is the original tensor, and μ, σ are the mean and the standard deviation of the entries of $\mathcal{Y}$. We normalize the states of each snapshot separately.



(a) 3D compressible Navier-Stokes simulations from PDEBench data set

(b) Snapshots from self-oscillating gel simulations from Section 3.1.4.2

Figure 3.14: Example snapshots from scientific data sets.

3.3.2.1 Self-Oscillating Gel Simulations

For this experiment, we use the same self-oscillating gel snapshot dataset [9, 4] that we have used for the experiments in Section 3.1.4.2. Recall that the training dataset arises from discretizing the 3-dimensional parameter space into a tensor product 20 values along each dimension, resulting in 8000 simulations. The test set is uniformly and randomly generated from the parameter space and consists of 15,000 unique parameter combinations and simulate each of those parameter combinations. 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.

Table 3.2 summarizes the results for varying normalization approaches and tolerances after completion of the compression of all data. These results indicate that, *when the algorithms*

are able to complete, `TT-ICE`* consistently outperforms HT-RISE in terms of compression ratio, compression time, and relative test error. For the experiments at $\varepsilon_{rel} = 0.01$ using unit vector and z-score normalizations, `TT-ICE`* runs into maximum walltime issues.

Table 3.2: Summary of the compression experiments with the self-oscillating gel data set. 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

To investigate this further, we show results for the un-normalized data as a function of the number of training samples in Figure 3.15 and Figure 3.16. More detailed results, including for different normalizations are presented in the supplementary material. Here we see that `TT-ICE`* outperforms in compression ratio, reduction ratio, compression time, and relative test error for all dataset sizes. We attribute this to the relatively simple structure of the self-oscillating gel dataset. In the next section we consider a more challenging multi-scale dataset where the conclusions differ.

3.3.2.2 PDEBench 3D Navier-Stokes Simulations

Next we compare the performance of HT-RISE on the more challenging PDEBench benchmark dataset [251]. From the PDEBench data set, 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

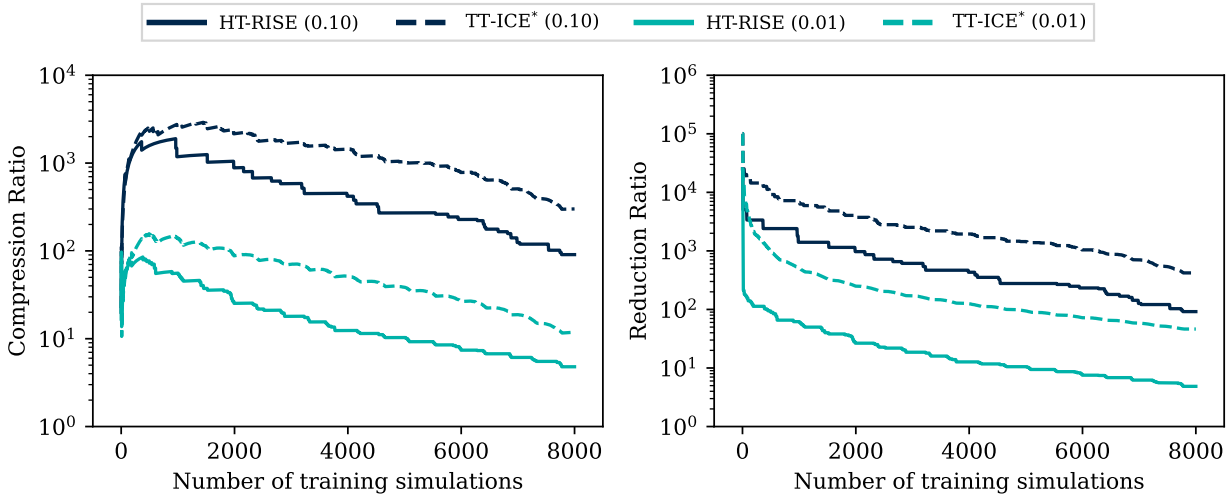


Figure 3.15: Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the self-oscillating gel dataset. `TT-ICE`* offers $2.5 - 3.3\times$ the CR and $4.6 - 9.5\times$ the RR of HT-RISE. For more detailed comparisons, please refer to Figure D.5 for experiments with $\varepsilon_{rel} = 0.10$ and Figure D.7 for experiments with $\varepsilon_{rel} = 0.01$.

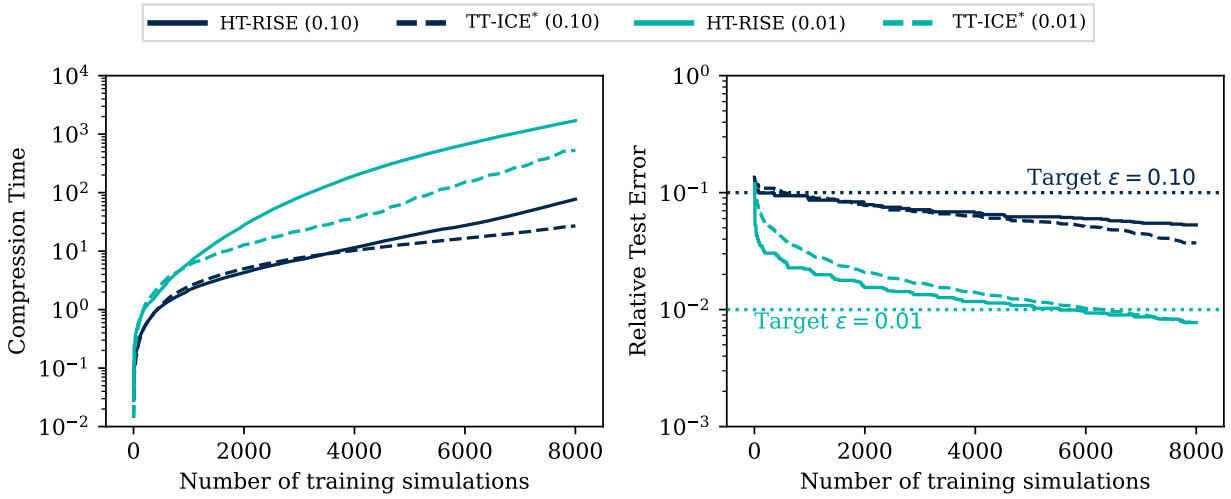


Figure 3.16: Compression time (left) and Relative Test Error (right) of the algorithms on the self-oscillating gel dataset. HT-RISE takes $2.8-3.2\times$ the time it takes `TT-ICE`* to complete compressing the stream. Neither method struggles to reduce the RTE below the target ε for both $\varepsilon_{rel} = 0.10$ and 0.01. For more detailed comparisons, please refer to Figure D.6 for experiments with $\varepsilon_{rel} = 0.10$ and Figure D.8 for experiments with $\varepsilon_{rel} = 0.01$.

equations that are used to generate the data set are:

$$\begin{aligned} \partial_t \rho + \nabla \cdot (\rho \mathbf{v}) &= 0, \\ \rho \left(\partial_t \mathbf{v} + \mathbf{v} \cdot \nabla \mathbf{v}\right) &= -\nabla p + \eta \Delta \mathbf{v} + \left(\zeta + \frac{\eta}{3}\right) \nabla (\nabla \cdot \mathbf{v}), \\ \partial_t (\epsilon + \rho \nu^2) + \nabla \cdot \left[(p + \epsilon + \frac{\rho \nu^2}{2}) \mathbf{v} - \mathbf{v} \cdot \sigma' \right] &= \mathbf{0}, \end{aligned} \tag{3.59}$$

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 results in a 5-dimensional tensor of size $64 \times 64 \times 64 \times 5 \times 21$ — $\sim 270\times$ larger than the self-oscillating gel dataset. To increase the dimensionality of the tensor, 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.

Table 3.3 summarizes the rest of our experiments with the PDEBench 3D turbulent Navier-Stokes data set in a similar way as for the self-oscillating gels. Here, the conclusion differs — HT-RISE now consistently outperforms `TT-ICE`* in terms of compression ratio, compression time, and RTE. Moreover, 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.

To investigate this behavior further, we again show results for the un-normalized data as a function of the number of training samples in Figure 3.17 and Figure 3.18. More detailed results, including for different normalizations are presented in the supplementary material. Here we see some new behavior. HT-RISE out-performs on compression ratio, compression speed, and generalization error on the test set. Note that it appears that `TT-ICE`* outperforms on reduction ratio, but this result is misleading in light of the relative test error. `TT-ICE`* is able to achieve higher reduction ratio, at the expense of any generalization capability. Indeed, for this problem it is not able to converge to the target error for the test set, even though it is guaranteed to do so *on the training data*.

Overall, we find that on this more complex dataset, `TT-ICE`* is not able to find a generalized low-rank structure across the tensors in the accumulation. While, as a result of its

Table 3.3: 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.1 and D.2 for experiments with $\varepsilon_{rel} = 0.10$ and Figures D.3 and D.4 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

hierarchical structure, HT-RISE discovers a low-rank structure across the tensors in the accumulation and is able to generalize well to unseen data. This outcome means that, HT-RISE requires much fewer updates to the approximation to achieve the target ε than `TT-ICE*` — leading to decreased compression time.

3.3.3 Numerical Experiments with Image Data

Next, we investigate image data. The image data will be obtained from two different sources: The MineRL Basalt competition data set [178] and multispectral images from the BigEarthNet data set [244, 245].

3.3.3.1 Minecraft Frames

First person games are of interest for inverse reinforcement learning purposes. The MineRL Basalt competition data set [178] contains gameplay episodes from the game Minecraft and

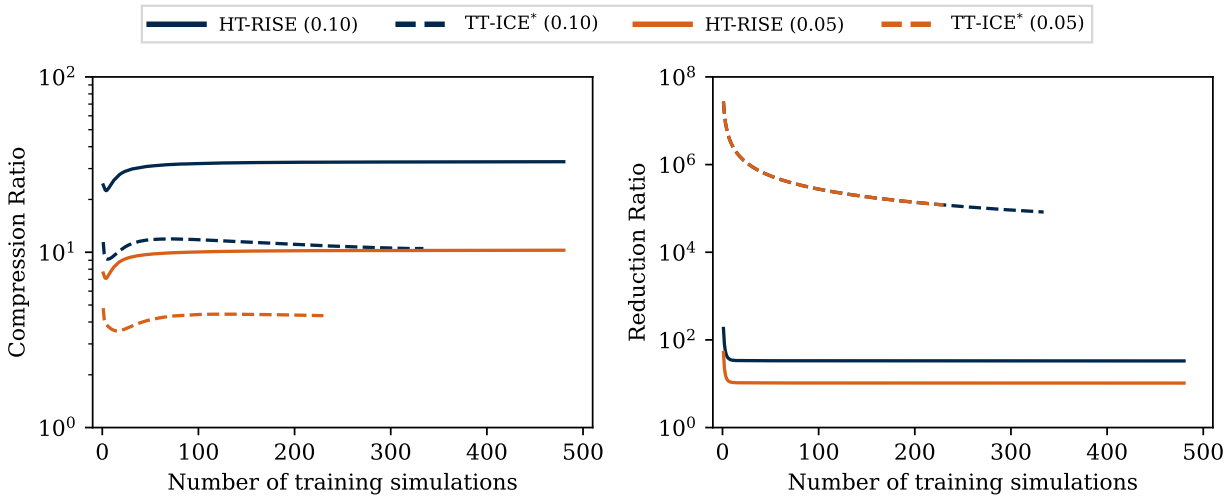


Figure 3.17: Compression ratio (CR - left) and reduction ratio (RR- - right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes data set. HT-RISE offers $2.4-3.1\times$ the CR of `TT-ICE`* but results in orders of magnitude lower RR. `TT-ICE`* does not complete the entire stream due to maximum walltime timeout whereas HT-RISE successfully completes the task. The results are averaged over 5 seeds.

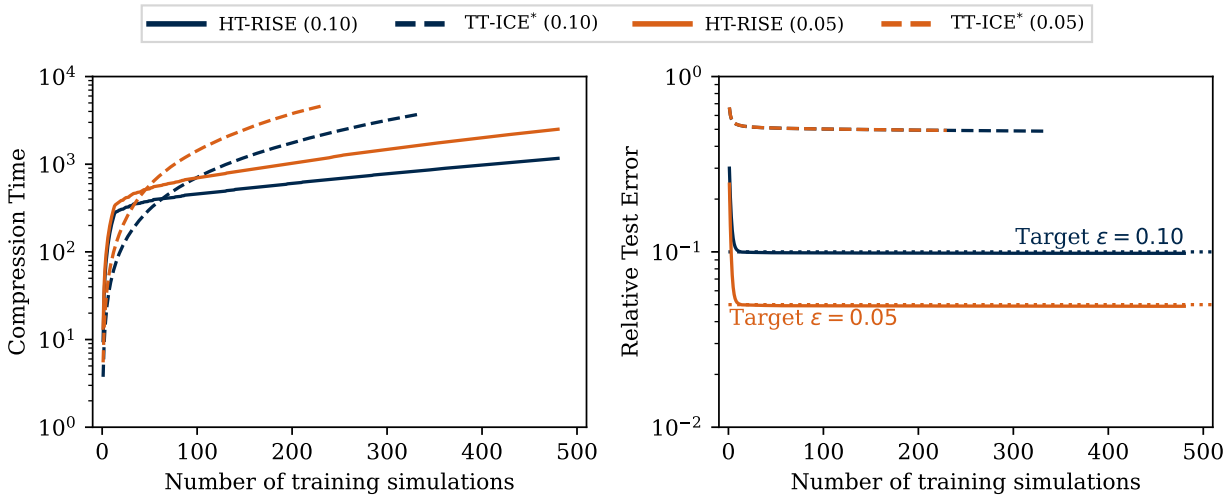
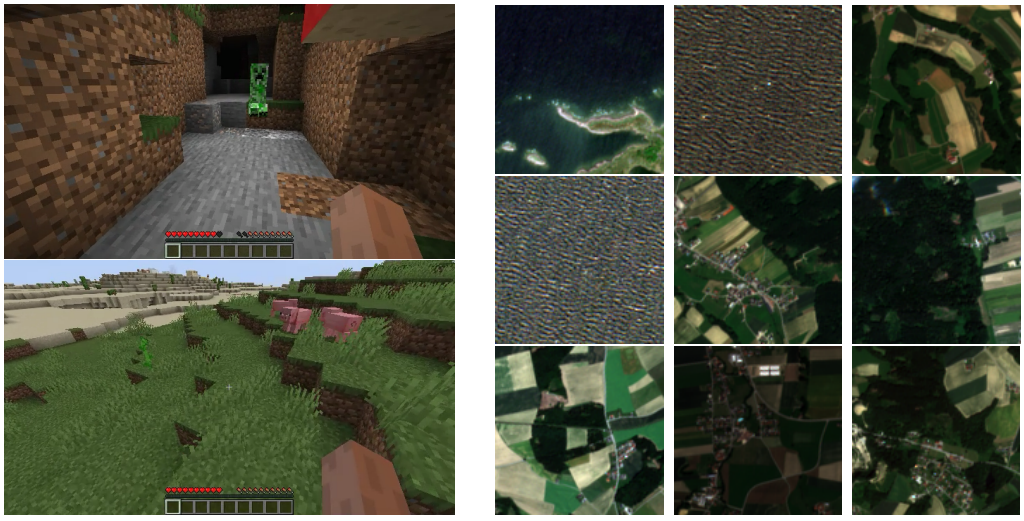


Figure 3.18: Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes data set. Until `TT-ICE`* hits the maximum walltime limit, `TT-ICE`* takes $1.8-3.2\times$ of the time it takes HT-RISE to complete compressing the stream. In addition, `TT-ICE`* 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.

is designed for the purpose of training agents to perform tasks in the game. The data set



(a) MineRL Basalt competition data set (b) Sentinel2 images (RGB channels)

Figure 3.19: Example frames from image-based data sets.

contains gameplay episodes from 4 different tasks, which are designed to be challenging for reinforcement learning agents. More detailed information about the specific tasks can be found in Appendix D.3 of the supplementary materials. 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 data set 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, at which the player spawns at a random biome looks around for a cave and enters it. The *Find Cave* task has 5466 gameplay episodes, totaling a size of 183GB on disk. Similar to [20], we downsample the 640×360 frames to 128×128. This results in a 4-dimensional data set $128 \times 128 \times 3 \times N_{frames}$, where N_{frames} is the number of frames in a video. Furthermore, we increase the dimensionality of the data set 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 data set as test set at random and use the rest for training. Since the size of the data set 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.

Table 3.4 summarizes the results; 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 behavior is expected since 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`*.

Table 3.4: Summary of the compression experiments with the MineRL Basalt competition data set. 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

To investigate this behavior further, we again show results as a function of the number of training samples in Figure 3.20 and Figure 3.21. More results, including qualitative comparisons are presented in the supplementary material. Here we see that while `TT-ICE`* out-performs on compression ratio, and reduction ratio, the computational times are similar between the two methods. Moreover, the HT-RISE displays faster convergence with training data, which explains the worse reduction ratio at small training set sizes. For example, Figure 3.21 shows 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.

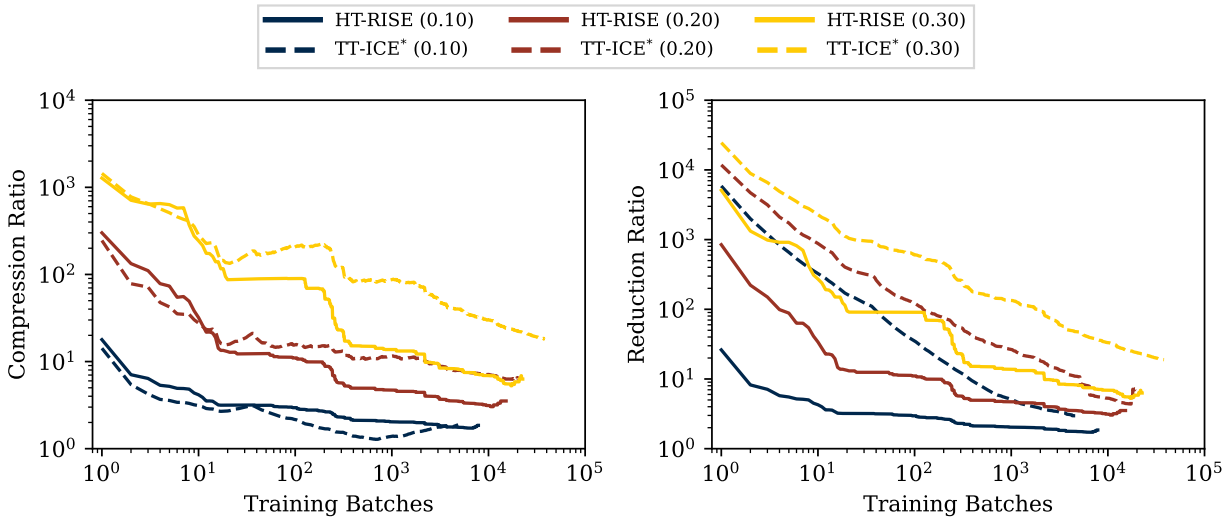


Figure 3.20: Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the Basalt MineRL data set.`TT-ICE`* offers $1-2.9\times$ the CR and $1.6-3\times$ the RR of HT-RISE. The results are averaged over 5 seeds.

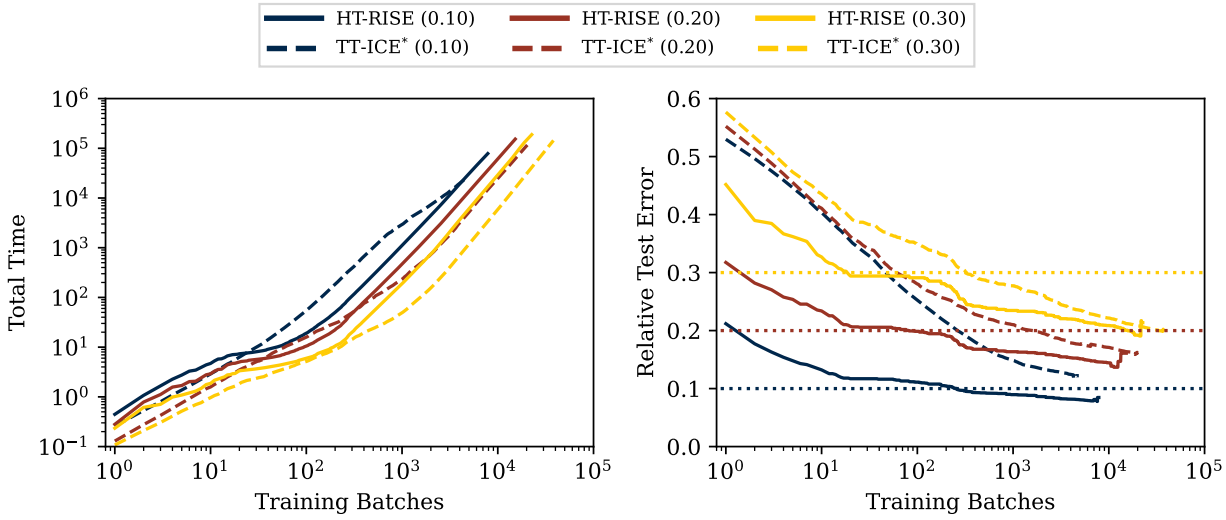


Figure 3.21: Compression time (left) and Relative Test Error (right) of the algorithms on the Basalt MineRL competition data set. Each target ε level is shown with dashed lines of their corresponding color. HT-RISE takes $2.8-3.2\times$ the time it takes `TT-ICE`* to complete compressing the stream. 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.

3.3.3.2 Multispectral Images

Another natural occurence of high-dimensional data in the form of images is multispectral satellite imagery. The BigEarthNet data set [245, 244] consists of 590,326 image patches, each of which is a 120x120x12 pixel multispectral image. The data set has a size of 66GB. The images are taken from the Sentinel-2 satellite and contain 12 spectral bands. Table D.1 lists the spectral bands and their wavelengths. The images are taken from 125 different locations around the world. The data set is originally designed for land cover classification and land use analysis, but we ignore any such labels for our experiments.

From the 590,326 images, we select 4% of them 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 two days. Since the selection of the test set is random, we execute each experiment 5 times and report the average performance. To increase the dimensionality of the data set, we reshape the 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.

The results are summarized in Table 3.5, with more detail in Figures D.9 and D.10. Table 3.5 shows that both algorithms are able to completely compress the entire data set, 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 the tightest error tolerance $\varepsilon_{rel} = 0.05$ runs out of the allocated memory after compressing 1781 batches ($\sim$178,000 images). For higher relative error thresholds, `TT-ICE`* achieves higher CR and RR than HT-RISE but takes a longer time. For the tightest error tolerances TT-ICE begins to run into timeout issues, while HT-RISE exhibits memory issues only.

Experiments with increasing numbers of training samples, follow a similar pattern to that of the PDEBench dataset, where HT-RISE significantly out-performs. Figure D.9 and Figure D.10 in the supplementary materials display more detailed results as a function of the number of training samples. Instead of showing these results here, Figures 3.22 and 3.23 present a qualitative comparison for the reconstructions of the multispectral images from the BigEarthNet data set using ε_{rel} between 0.05 and 0.30. Images in Figure 3.22 are generated by reconstructing latent space representations of a training image learned during the compression process. Images in Figure 3.23 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

Table 3.5: Summary of the compression experiments with the BigEarthNet multispectral satellite imagery data set. 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

displayed in RGB format, where only the bands B02, B03, and B04 are used.

Figure 3.22 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 3.23. 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 reconstruction has slight discoloration in comparison to the original image and maintains the fine features well.

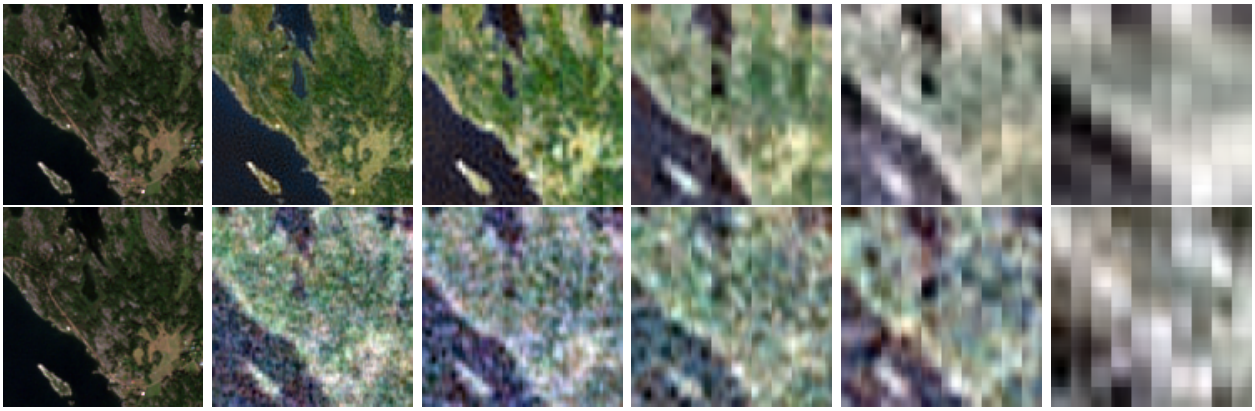
3.4 Summary

In this work, we proposed a new algorithm, TT-ICE, to incrementally update a TT-decomposition to compress a stream of data. Our algorithm TT-ICE improves on the existing



(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 3.22: 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 3.23: 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.

state-of-the-art because (1) it maintains a desired error tolerance *for all data increments*, (2) it updates its ranks without excessive growth, and (3) it maintains orthogonality of the TT-cores to enable efficient projection and prediction for uncompressed data. We provide proof that the TT-ICE algorithm maintains its accuracy throughout the compression process. We then provide three heuristics to improve on this algorithm and show empirical evidence that they improve performance with little sacrifice in accuracy. This enhanced version of TT-ICE is also guaranteed to maintain the accuracy of the already compressed portion of the stream. However, this guarantee does not necessarily hold for the portion of the stream compressed with TT-ICE* since heuristics are used in the first place. Experimental results on two different types of data demonstrate the superior performance of TT-ICE over ITTD and provide empirical proof of the additional benefits obtained from the heuristic upgrades we implemented in TT-ICE*. For simulation data, TT-ICE* achieves twice the compression ratio of TT-ICE in half the time. For image data, TT-ICE* achieves a comparable, if not higher, compression ratio with up to 80% reduction in the TT-ICE time. Moreover, on resource-limited hardware, TT-ICE and TT-ICE* have proven themselves to be reliable and performant compression methods.

Extensions of this work will attempt to theoretically justify the heuristics used in TT-ICE*. It will also deploy the proposed methods to enable scalable machine learning in the latent space identified by the compression, e.g., for inverse design [5] and behavioral cloning [50].

In Section 3.2 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 in Section 3.3 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 a state-of-the-art incremental tensor decomposition algorithm `TT-ICE*` using two PDE-based and two image-based data sets. The results indicate that in more complex data sets 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 data sets, `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 data set.

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.

3.5 Acknowledgments

The research presented in Section 3.1 was supported in part 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. Section 3.1 has also received partial support from the Department of Energy Office of Scientific Research, ASCR, under grant DE-SC0020364. We thank Brian Chen for preparing the ATARI game dataset [48]. The code for TT-ICE and TT-ICE* is publicly available on github.com/dorukaks/TT-ICE.

The research presented in Sections 3.2 and 3.3 was supported in part by Los Alamos National Laboratories under the project "Algorithm/Software/Hardware Co-design for High Energy Density applications"; and by the Automotive Research Center in accordance with Cooperative Agreement W56HZV-19-2-0001 with U.S. Army DEVCOM Ground Vehicle Systems Center. Sections 3.2 and 3.3 used computing resources provided by an AFOSR DURIP # 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.

Algorithm 2 `TT-ICE*`: Incremental update of a tensor train decomposition with heuristic performance upgrades

Input

$\left\{U_i^k\right\}_{i=1}^{d+1}$ reshaped cores of the TT-decomposition of the accumulation $\mathcal{X}^k$

$\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N^{k+1}}$ new tensor

τ occupancy threshold (suggested=0.8)

ε_{des} relative error upper bound

Output

$\left\{U_i^{k+1}\right\}_{i=1}^{d+1}$ updated cores for the accumulation $\mathcal{X}^{k+1}$

Check representation accuracy with existing TT-cores and *Skip* updating if sufficient

$\varepsilon_{\mathcal{Y}}(i) \leftarrow \frac{\|\mathcal{Y}(i)-\tilde{\mathcal{Y}}(i)\|_F}{\|\mathcal{Y}(i)\|_F}\Big|_{i=1}^{N^{k+1}}$ ▷ $\tilde{\mathcal{Y}}$ is the approximation of $\mathcal{Y}$ using U^k, and $\varepsilon_{\mathcal{Y}} \in \mathbb{R}^{N^{k+1}}$ as defined in (3.30)

if $\texttt{mean}(\varepsilon_{\mathcal{Y}}) \leq \varepsilon_{des}$ **then**

$\{U_i^{k+1}\}_{i=1}^d \leftarrow \{U_i^k\}_{i=1}^d$

else

***Subselect* observations**

$\mathcal{D} \leftarrow \{\mathcal{Y}(j) : \ \varepsilon_{\mathcal{Y}}(j) > \varepsilon_{des}\}_{j=1}^{N^{k+1}}$ ▷ $\mathcal{D} \subset \mathcal{Y}$ will be used at update

$\varepsilon_{upd} \leftarrow \sqrt{\frac{(\varepsilon_{des}\|\mathcal{Y}\|_F)^2 - \left(\|\mathcal{D}_C - \tilde{\mathcal{D}}_C\|_F\right)^2}{\|\mathcal{D}\|_F^2}}$ ▷ $\mathcal{D} \cup \mathcal{D}_C = \mathcal{Y}$, $\tilde{\mathcal{D}}_C$ is approximation of $\mathcal{D}_C$ using U^k

$\epsilon = \frac{\varepsilon_{upd}}{\sqrt{d}}\|\mathcal{D}\|_F$ ▷ ϵ is the truncation parameter for SVD

Perform incremental updates with the selected observations

$D_1 = \texttt{reshape}(\mathcal{D}, [n_1, n_2 \dots n_d N^{\mathcal{D}}])$ ▷ $N^{\mathcal{D}}$ is the number of selected observations

$U_1^{k,pad} \leftarrow U_1^k$ ▷ First core has no padding

for $i = 1$ to d **do**

Check core *Occupancy*

if $\texttt{occupancy}(U_i^{k,pad}) \geq \tau$ **then**

$U_i^{k+1} \leftarrow U_i^{k,pad}$

$r_i^{k+1} \leftarrow r_i^k$

$U_{i+1}^{k,pad} \leftarrow U_{i+1}^k$

else

Perform incremental update

$R_i^k = \left(I - U_i^{k,pad} U_i^{k,pad^T}\right) D_i$

$U_{R_i}^k \leftarrow \texttt{SVD}\left(R_i^k, \epsilon\right)$

$U_i^{k+1} \leftarrow \left[U_i^{k,pad} \quad U_{R_i}^k\right]$

$U_{i+1}^{k,pad} \leftarrow \texttt{reshape}\left(\begin{bmatrix} \texttt{reshape}\left(U_{i+1}^k, \left[r_i^k, n_{i+1} r_{i+1}^k\right]\right) \\ \mathbf{0}_{r_{R_i} \times n_{i+1} r_{i+1}^k} \end{bmatrix}, [r_i^{k+1} n_{i+1}, r_{i+1}^k]\right)$

end if

$D_{i+1} \leftarrow \texttt{reshape}(U_i^{k+1^T} D_i, [r_i^{k+1} n_{i+1}, n_{i+2} \dots N^{\mathcal{D}}])$

end for

end if

$\hat{Y}^{k+1} \leftarrow \texttt{project}(\mathcal{Y}^{k+1}, \{U_i^{k+1}\}_{i=1}^d)$ ▷ $\hat{Y}$ is obtained by sequentially projecting $\mathcal{Y}^{k+1}$ onto $\{U_i^{k+1}\}_{i=1}^d$

$U_{d+1}^{k+1} \leftarrow \left[U_{d+1}^k \quad \hat{Y}^{k+1}\right]$ ▷ $\hat{Y}^{k+1} \in \mathbb{R}^{r_d \times n_{d+1}^{k+1}}$ are the columns

Algorithm 3 `BHT-l2r`: Error truncated leaves-to-root batch hierarchical Tucker decomposition

Input
  $\mathcal{Y} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$ Input tensor
  $\mathbf{T}$ Dimension tree of the decomposition with depth p
  ε_{rel} Relative error tolerance
Output
  $\mathbf{H}_{\mathcal{Y}}$ Hierarchical Tucker representation with leaves and cores $\mathcal{G}_{\ell,i_\ell}$
$\varepsilon_{nw} \leftarrow \varepsilon_{rel}\|\mathcal{Y}\|_F/\sqrt{2d-2}$ ▷ Node-wise error tolerance for SVD
$\mathcal{C} \leftarrow \mathcal{Y}$
for $i = 1, \ldots, |\mathbf{T}_p|$ **do** ▷ Compute the leaves on layer p
  $C \leftarrow \texttt{unfold}\,(\mathcal{C}, d_{p,i})$ ▷ $d_{p,i} \in \{1, \ldots, d\}$ is the dimension corresponding to the leaf node $\mathbf{N}_{p,i}$
  $\mathcal{G}_{p,i} \leftarrow \texttt{SVD}\,(C, \varepsilon_{nw})$ ▷ Only the left singular vectors are kept with $\mathbf{R}_{p,i} = r_{p,i}$
  $\mathbf{I}_{\mathbf{N}_{p,i}} \leftarrow \{n_i, r_{p,i}\}$ ▷ Create index set for the leaf nodes
end for
$\mathcal{C} \leftarrow \mathcal{C} \underset{\mathcal{I}}{\times} [\![\mathcal{G}_{p,1}, \ldots, \mathcal{G}_{p,|\mathbf{T}_p|}]\!]$ ▷ $\mathcal{I} = \{d_{p,i}; i = 1, \ldots, |\mathbf{T}_p|\}$ as in (3.42)
for $\ell = p-1$ to 1 **do**
  $\mathcal{C} \leftarrow \texttt{reshape}\,(\mathcal{C}, \mathbf{I}_{\mathbf{T}_\ell})$ ▷ $\mathbf{I}_{\mathbf{T}_\ell} \cup \{N\}$ with $\mathbf{I}_{\mathbf{T}_\ell}$ constructed using (3.43)
  $\mathcal{G}_{\ell,1}, \ldots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|} \leftarrow \texttt{HOSVD}(\mathcal{C}, \varepsilon_{nw})$ ▷ $\mathbf{I}_{\mathbf{N}_{\ell,j}}$ are created using (3.44) for $j = 1, \ldots, |\mathbf{T}_\ell|$
  for $j = 1, \ldots, |\mathbf{T}_\ell|$ **do**
    $\mathcal{G}_{\ell,j} \leftarrow \texttt{reshape}\left(\mathcal{G}_{\ell,j}, \mathbf{I}_{\mathbf{N}_{\ell,j}}\right)$ ▷ Folds $\mathcal{G}_{\ell,j}$ into 3D if $\mathbf{N}_{\ell,i_\ell}$ is a transfer node
  end for
  $\mathcal{C} \leftarrow \mathcal{C} \times [\![\mathcal{G}_{\ell,1}, \ldots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|}]\!]$
end for
$\mathcal{G}_{0,1} \leftarrow \texttt{reshape}\left(\mathcal{C}, \mathbf{I}_{\mathbf{N}_{0,1}}\right)$ ▷ $\mathbf{I}_{\mathbf{N}_{0,1}} = \{r_{1,1}, r_{1,2}, N\}$

Algorithm 4 `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 $\mathcal{G}_{\ell,j}^{k-1}$
- $\mathbf{I}^{k-1}$ index set from $(k-1)$-th batch
- ε_{rel} desired relative error truncation threshold

Output
- $\mathbf{H}_{\mathcal{X}^k}$ *Updated* hierarchical Tucker representation with dim. tree $\mathbf{T}$ and cores $\mathcal{G}_{\ell,j}^{k}$
- $\mathbf{I}^{k}$ *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

  $\mathcal{C} \leftarrow \texttt{reshape}\left(\mathcal{C}, \mathbf{I}^{k-1}_{\mathbf{T}_{\ell-1}}\right)$ ▷ 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*

  $\mathcal{G}^k \leftarrow \mathcal{G}^{k-1} \qquad \forall \mathcal{G}^{k-1} \in \mathbf{H}_{\mathcal{X}^{k-1}} \setminus \mathcal{G}_{0,1}^{k-1}$

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 ε_{des}

  for $j = 1$ to $|\mathbf{T}_p|$ **do**

    $C \leftarrow \texttt{unfold}\,(\mathcal{C}, d_{p,j})$ ▷ $d_{p,j} \in \{1, \ldots, d\}$ is the index of the dimension corresponding to $\mathbf{N}_{p,j}$

    $R_{p,j} \leftarrow \Pi_{p,j}^{k-1} C$ ▷ $\Pi_{p,j}^{k-1}$ is computed according to (3.50)

    $U_R^k \leftarrow \texttt{SVD}(R_{p,j}, \varepsilon_{nw})$ ▷ Error-truncated `SVD` on the residual according to (3.51)

    $\mathcal{G}_{p,j}^k, \mathbf{I}^k_{\mathbf{N}_{p,j}} \leftarrow \texttt{expandCore}(\mathbf{N}_{p,j}, \mathbf{I}^{k-1}_{\mathbf{N}_{p,j}}, \mathcal{G}_{p,j}^{k-1}, U_R^k)$ ▷ Using Algorithms 6 and 7

    $\mathcal{G}_{p-1,m}^{k-1} \leftarrow \texttt{padWithZeros}(\mathcal{G}_{p-1,m}^{k-1}, t, r_{R_{p,j}})$ ▷ Assume $\mathbf{N}_{p,j}$ is the t-th successor of $\mathbf{N}_{p-1,m}$

  end for

  $\mathbf{I}^k_{\mathbf{T}_{p-1}} \leftarrow \texttt{updateIndexSet}(\mathbf{I}^{k-1}_{\mathbf{T}_{p-1}}, \mathbf{T}_{p-1})$ ▷ Update using Algorithm 6 according to (3.43)

  $\mathcal{C} \leftarrow \mathcal{C} \underset{\mathcal{I}}{\times} [\![\mathcal{G}_{p,1}^k, \ldots, \mathcal{G}_{p,|\mathbf{T}_p|}^k]\!]$ ▷ $\mathcal{I} = \{d_{p,i}; i = 1, \ldots, |\mathbf{T}_p|\}$ as in (3.42)

  Update cores on the remaining layers

  for $\ell = p-1$ to 1 **do**

    $\mathcal{C} \leftarrow \texttt{reshape}\left(\mathcal{C}, \mathbf{I}^k_{\mathbf{T}_\ell}\right)$

    for $j = 1$ to $|\mathbf{T}_\ell|$ **do**

      $C \leftarrow \texttt{unfold}\,(\mathcal{C}, \ell)$

      $R_{\ell,j} \leftarrow \Pi_{\ell,j}^{k-1} C$ ▷ $\Pi_{\ell,j}^{k-1}$ is computed according to (3.50)

      $U_R^k \leftarrow \texttt{SVD}(R_{\ell,j}, \varepsilon_{nw})$ ▷ Compute error truncated `SVD` on the residual according to (3.51)

      $\mathcal{G}_{\ell,j}^k, \mathbf{I}^k_{\mathbf{N}_{\ell,j}} \leftarrow \texttt{expandCore}(\mathbf{N}_{\ell,j}, \mathbf{I}^{k-1}_{\mathbf{N}_{\ell,j}}, \mathcal{G}_{\ell,j}^{k-1}, U_R^k)$ ▷ Using Algorithms 6 and 7

      $\mathcal{G}_{\ell-1,m}^{k-1} \leftarrow \texttt{padWithZeros}(\mathcal{G}_{\ell-1,m}^{k-1}, t, r_{R_{\ell,j}})$ ▷ Assume $\mathbf{N}_{\ell,j}$ is the t-th successor of $\mathbf{N}_{\ell-1,m}$

    end for

    $\mathbf{I}^k_{\mathbf{T}_{\ell-1}} \leftarrow \texttt{updateIndexSet}(\mathbf{I}^{k-1}_{\mathbf{T}_{\ell-1}}, \mathbf{T}_{\ell-1})$

    $\mathcal{C} \leftarrow \mathcal{C} \times [\![\mathcal{G}_{\ell,1}^k, \ldots, \mathcal{G}_{\ell,|\mathbf{T}_\ell|}^k]\!]$

  end for

end if

$\mathcal{G}_{0,1}^k \leftarrow \mathcal{G}_{0,1}^{k-1} \oplus^3 \mathcal{C}$

CHAPTER 4

Applications Considering Scientific Data

This chapter discusses the application of tensor decompositions in the context of scientific data, focusing on the use of tensor decompositions for data compression and Bayesian inference. We can summarize the key contributions of this chapter as follows:

1. Empirical demonstration of the effectiveness of TT-ICE for compressing scientific data, achieving compression rates up to $300,000\times$ while preserving the ability to reconstruct high-quality data. (Section 4.1)

2. Empirical demonstration that the low-dimensional embeddings obtained through TT-ICE can be used for the downstream task of Bayesian inversion. (Section 4.2)

4.1 Compression of Particle-in-Cell Simulations

This section presents our approach to compressing and analyzing the simulation data generated by electric propulsion simulations. We first describe the simulation that generates the data of interest. We then describe our compression methodology. Finally, we describe the outputs that are compared between the original and compressed datasets. Figure 4.1 shows a high-level overview of the proposed compressed analysis pipeline.

Particle-in-cell (PIC) methods [257] are computational techniques that simulate the behavior of charged particles in electromagnetic fields by representing particles as discrete computational elements that move through a spatial grid. In these methods, particles carry charge and current information, while electromagnetic fields are calculated on a fixed mesh using Maxwell's equations. The coupling between particles and fields occurs through interpolation: particle quantities are deposited onto the grid to compute field sources, and grid-based fields are interpolated back to particle positions to determine forces and update

Note that this section is based on the work "Compressed Analysis of Electric Propulsion Simulations using Low Rank Tensor Networks" presented at the 38th International Electric Propulsion Conference. [8]

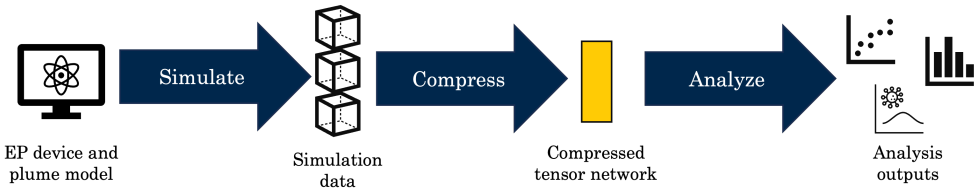


Figure 4.1: *Proposed compressed analysis pipeline. High-fidelity simulation environments generate large-scale simulation data. This data can be compressed incrementally into a tensor-network format and then analyzed without decompression.*

particle motion. This approach enables the simulation of complex plasma phenomena while maintaining computational tractability for systems containing large numbers of particles.

Many spacecraft propulsion systems operate by accelerating a plasma electrostatically. These *electric propulsion* (EP) devices achieve high efficiency but are difficult to simulate predictively due to the presence of complex kinetic plasma effects. PIC methods are therefore useful for simulating these devices, as they can capture the detailed interactions between charged particles and electromagnetic fields. Hall thrusters are a type of EP device that uses a magnetic field to trap electrons and create a closed drift motion, which in turn accelerates ions to produce thrust. The simulation of Hall thrusters is particularly challenging due to the complex interactions between the plasma, magnetic fields, and electric fields, as well as the need to resolve small-scale phenomena such as electron mobility and ionization processes.

Particle-in-cell methods [64] are widely used to simulate Hall thrusters [61, 250]. Fully-kinetic particle-in-cell simulations have been used to gain insight into the micro-scale physics governing the macroscopic performance of these thrusters.[149, 46] Hybrid methods, where the heavier species are treated as particles while the electrons are treated as a fluid, have been extensively used for many years for device-scale discharge plasma simulation[177], performance prediction,[200] and plume simulation.[16]

Each time snapshot of a PIC simulation contains information on the particle positions, velocities, and field quantities, and derived quantities at that time step. To achieve high-fidelity results, these simulations often require a large number of particles (on the order of 10^8) and a fine spatial resolution. Considering the fine time discretization required to capture the fast dynamics of the plasma, the number of time snapshots to simulate the device over a realistic operational time can be on the order of 10^4 or more. Combining these factors, the total amount of data generated by a single PIC simulation can easily reach hundreds of gigabytes if not terabytes. As a remedy to this problem, researchers often resort to storing a temporally subsampled version of the simulation data. Even in that case the size of the generated simulation data makes it challenging to store and use the data in further analysis

and downstream learning tasks.

For this section, we simulate an electric propulsion device under on-orbit conditions using *Thermophysics Universal Research Framework* (TURF) software package, developed by the United States Air Force Research Laboratory (AFRL). The simulation outputs a structured volume mesh capturing both field quantities and particle number densities, as well as an unstructured triangular surface mesh capturing electric potential and current on the thruster surfaces. We compress the different mesh files individually and group the states within each mesh component according to their properties. Note that our objective is not to match the numerical results of the simulation [31] exactly, but to generate a typical and scalable dataset for a Hall thruster plume for compression analysis. This 3D plume simulation in TURF would require significantly more particles to replicate the resolution of Boyd's 2D axis-symmetric plume simulation, particularly in the backflow region of the computational domain.

Finally, we compare the analysis outputs generated from the original and compressed datasets to quantify the loss in simulation output fidelity from tensor network compression both qualitatively and quantitatively.

4.1.1 SPT-100 Plume Simulation Test Case

The SPT-100 (Stationary Plasma Thruster-100) is a Hall-effect electric propulsion system originally developed in the former Soviet Union and later commercialized internationally. This thruster operates by creating a plasma discharge in a magnetic field, where electrons are trapped in closed drift orbits while ions are accelerated axially to produce thrust. Figure 4.2 shows images of the SPT-100 thruster, including a display image of the thruster and an image of the thruster in operation. The thruster features an annular discharge channel with permanent magnets providing the radial magnetic field necessary for Hall current formation. Due to its proven flight heritage and reliable performance, the SPT-100 has been extensively used for satellite station-keeping, orbit raising, and interplanetary missions, making it a representative test case for electric propulsion simulation studies.

Our test case is a 3D hybrid-particle-in-cell (PIC) simulation of the plume of an SPT-100 Hall thruster performed in TURF [172]. The case is an extension that is based on the 2D axisymmetric model of Boyd et al. [31].

Our domain consists of a quarter of the plume, and extends from (-2, 0, 0) meters to (10, 10, 10) meters. The computational domain is partitioned into 8 subdomains, uniformly in each direction. The subdomains are evenly distributed to Message Passing Interface (MPI) processes to parallelize the simulation in TURF [17]. We discretize the larger computational



(a) SPT-100 thruster [147] (b) SPT-100 thruster in operation [148]

Figure 4.2: Images of the SPT-100 Hall thruster. Figure 4.2a shows the thruster in display. Note that this figure is cropped to show the SPT-100 thruster only. Figure 4.2b shows an operating electric Hall thruster identical to those that will be used to propel the Psyche spacecraft.

domain into a structured grid of $120 \times 100 \times 100$ cells, giving us grid cells of 10 cm.

The simulation contains three species — Xenon atoms, singly-charged Xenon ions, and electrons. We treat both the atoms and ions as computational macroparticles, and model the electrons as an unmagnetized, isothermal, collisionless fluid. The virtual thruster exit plane is an annulus of inner diameter 60 mm and outer diameter of 100 mm centered at (0, 0, 0) meters, with the plume centerline aligned with the x-axis. The neutral atoms are injected with a number density of $10^{18}\,\mathrm{m}^{-3}$, face-normal velocity of $300\,\mathrm{m/s}$, and a temperature of $1000\,\mathrm{K}$. The ions are injected with a density of $10^{17}\,\mathrm{m}^{-3}$, a normal velocity of $17\,\mathrm{km/s}$, and a temperature of $4\,\mathrm{eV}$. We assume both species have an effective divergence angle of 15° and are injected with radially-uniform properties. The Boltzmann relation is used with an electron temperature of 3 eV, fixed over the computational domain.

We model the thruster surface as a solid plate of aluminum with dimensions $120\,\mathrm{mm} \times 120\,\mathrm{mm} \times 10\,\mathrm{mm}$. The potential at the surface is set to zero, and ions that collide with this surface neutralize and reflect diffusely. We remove particles that cross the domain boundaries from the simulation, and set the electric field normal to these boundaries to zero. Figure 4.3 provides a sketch for the simulation domain as well as the positioning of the thruster exit plane.

TURF models charge-exchange (CEX) collisions between neutrals and ions using the

direct simulation Monte Carlo (DSMC) [14, 18] method, with a total cross section, σ_{cex}, of

$$\sigma_{cex} = \left(0.8423 \times 10^{-20}\right)\left[-23.30 \log_{10}(v_{\text{rel}}) + 142.21\right] \quad \text{m}^2, \tag{4.1}$$

where v_{rel} is the relative velocity of the colliding particles in m/s. The code handles elastic momentum-exchange (MEX) collisions with the Monte Carlo collision (MCC) method assuming a background neutral temperature of 1000 K and a total collision cross-section, σ_{mex}, of

$$\sigma_{mex}(\text{Xe}, \text{Xe}^+) = \frac{6.42 \times 10^{-16}}{v_{\text{rel}}} \quad \text{m}^2. \tag{4.2}$$

To compute the post-collision velocities, TURF employs a modified variable hard sphere (VHS) model and assumes isotropic scattering.s

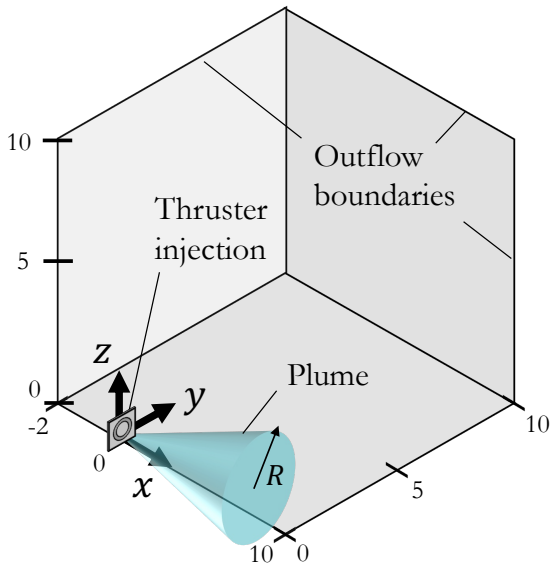


Figure 4.3: Visualisation of the simulation domain. The domain is [-2,10] meters for the x axis, and [0,10] meters for y and z axes. The thruster centerline is aligned with the x axis and the exit plane of the thruster is located at $x = 0\,\text{m}$. Note that the thruster is not shown to scale for visualisation purposes.

Following Boyd [31], the simulation proceeds in two steps: an initial warm-up phase where neutral atoms are injected until a steady state is reached, followed by a second phase where ions are introduced. The warm-up phase uses a timestep of $\Delta t = 5 \times 10^{-6}\,\text{s}$ and proceeds for 1500 iterations, corresponding to a simulation time of $t = 7.5\,\text{ms}$. Once the neutral atoms reach steady state, ions are injected with a $\Delta t = 5 \times 10^{-6}\,\text{s}$, continuing for an additional 10800 iterations until a simulation end time of $t_f = 61.5\,\text{ms}$. A simulation with the grid spacing described above and 3.6 million physical particles takes about 17 hours using 16

CPU cores over 4 compute nodes, with Intel Xeon Gold 6154 processors and 25 GB RAM per core (100 GB memory in total).

The simulation has three types of outputs: 1) uniform structured cut-cell volume data, 2) uniform unstructured surface field data, and 3) trajectories of neutral atom and ion particles. Volume data is captured on a structured mesh and consists of particle number densities, ion current density, and electric field quantities.

Using the settings described above, the simulation generates a structured mesh file (.vts) of size 337 MB for each saved iteration for volume data. Surface data is captured on an unstructured triangular mesh and consists of floating potential and total current to the thruster surface from the plasma. Using the settings described above, the simulation generates an unstructured mesh file (.vtu) of size 228 KB for each saved iteration for surface data. At each saved iteration the simulation generates a trajectory file in the form of a geometric structure data object (.vtp) with size 3.64 GB. The .vtp files consist of points, lines, and/or polygons and holds the positions and velocities of each computational neutral and ion particle in the simulation at a particular time step. An area of future effort is compression of such poly-data files. For a typical plasma simulation such as this one, storing all trajectory files and further utilizing them for analysis becomes quickly intractable. Luckily, one would not need to save the entire simulation dataset to do fully-kinetic analyses such as anomalous electron transport and closures of kinetic plume models. Adopting incremental tensor decomposition algorithms would allow efficient compression to be done on simulation iteration files as they are generated.

In this section, we focus on capturing the steady state behavior of the particles in the simulated system and therefore discard the snapshot files corresponding to the first 5000 iterations. These 5000 iterations encompass neutral atoms reaching steady state and transient behavior after initial ion injection. Between iterations $5000 - 12300$, we save the structured volume mesh at every 50th iteration. A subset of 1000 iterations, corresponding to iterations $11300 - 12300$ where plasma quantities are in steady state, are reconstructed and used for structured mesh analysis presented in Section 4.1.4.2.

For our compression analysis, we consider a surface dataset corresponding to a simulation with a slightly modified setting to the one presented above for the volume mesh compression. A larger surface mesh with an area of $0.14\,\mathrm{m}^2$ is used to provide a substantial dataset for compression demonstrative purposes. Further, the neutrals atoms and ions are injected at the same time and the simulation time step is reduced to $\Delta t = 1 \times 10^{-7}$ to see sufficient transient behavior. We compress and reconstruct snapshots for iterations 0-15000, where every 100 iterations are saved. These reconstructions are used for unstructured surface mesh analysis presented in Section 4.1.4.2.

4.1.2 Data Compression Pipeline

This section presents the data compression pipeline for simulation data generated by TURF. We first describe the tensor decomposition format we prefer to compress the simulation outputs. We then describe the normalization methods we employ to preprocess the data for increased compression performance.

For this work, we consider the TT-format to compress the TURF simulation data. One of the key advantages of the TT-format is that it allows us to reconstruct an entry/slice of the approximated tensor $\mathcal{X}$ without reconstructing the full tensor. Additionally, TT-format enables other analyses and operations such as integration, addition, differentiation etc. in the compressed state. As an example of entry retrieval, the entry $\mathcal{X}(i_1, \ldots, i_d)$ can be computed as

$$\mathcal{X}(i_1, \ldots, i_{d-1}) = \sum_{\alpha_1, \ldots, \alpha_d}^{r_1, \ldots, r_{d-1}} \mathcal{G}_1(:, \alpha_1)\mathcal{G}_2(\alpha_1, i_1, \alpha_2) \cdots \mathcal{G}_{d-2}(\alpha_{d-2}, i_{d-1}, \alpha_{d-1})\mathcal{G}_d(\alpha_{d-1}, i_d). \quad (4.3)$$

Recall that when the snapshots are accumulated along the last dimension, the first $d-1$ TT-cores will be same for all snapshots and only the last TT-core will be unique to a snapshot. Therefore, the slices of the last TT-core $\mathcal{G}_d$ can also be used as a latent representation for individual snapshots.

Instead, we employ an incremental alternatives to `TT-SVD` [7, 161, 1, 277]. From the existing alternatives, we use the `TT-ICE`* algorithm[7] due to its low memory footprint and high computational speed while compressing scientific simulations.[1] For the compression experiments in this work, we reshape the mesh data arrays into shapes with size $\mathcal{O}(10)$ to obtain a balanced growth in TT-ranks. [7, 49] In addition to the reshaping, we create an additional dimension and increment along that last dimension using the `TT-ICE`* algorithm.

4.1.3 Data Normalization

Due to the nature of the simulations, there are high magnitude differences across states and spatial domain. These values can range 18 orders of magnitude, from 10^0 to 10^{18}. Since `TT-ICE`* performs a sequence of error-truncated SVDs to expand the basis vectors stored in each TT-core, the algorithm favors the accuracy of representation at locations with higher number densities and allows errors in the sparsely populated regions. This becomes an issue especially when modelling the backflow region.

[1] Of the heuristics available to accelerate TT-ICE, we only use the "occupancy" and "skip" heuristics of `TT-ICE`*, which avoid computing an SVD if a given TT-core is already full rank and skip the update of a TT-core if the presented data can already be represented within the desired target error using the existing set of TT-cores.

To address this issue, we employ different normalization methods such as z-score, logarithmic, square root, and unit vector normalization. Z-score normalization scales the data to have a mean of zero and a standard deviation of one. With μ_x,σ_x being the mean and standard deviation of the data $x \in \mathbb{R}^k$, z-score normalization computes the normalized data $x_{norm} \in \mathbb{R}^k$ as

$$x_{norm} = \frac{x - \mu_x}{\sigma_x}. \tag{4.4}$$

Another normalization method we adopted is the unit vector normalization. Given the same x as above, unit vector normalization scales x to have a magnitude of one such that

$$x_{norm} = \frac{x}{\|x\|_F}, \tag{4.5}$$

where $\|x\|_F$ is the Frobenius norm of x.

In addition to (4.4) and (4.5), we also experimented with nonlinear functions to normalize the states. Logarithmic scaling with base e ($x_{norm} = \log(x)$), logarithmic scaling with base 10 ($x_{norm} = \log_{10}(x)$), and square root scaling ($x_{norm} = \sqrt{x}$) are some example nonlinear functions we included in this work.

Note that even when states for different quantities of interest (QoI) are compressed together, the normalization methods are applied to each state separately. In some cases, we also experimented using a combination of these normalization methods. This allowed us to first reduce the spread of values and then perform a statistically significant normalization. When cascading 2 normalization methods, we first apply the normalization methods in the order of mention. That is, in case of unit vector-logarithmic normalization, the data is first processed through unit vector normalization and then the outputs of that normalization are further processed using logarithmic normalization.

To summarize, we preprocess the simulation data prior to compression using the `TT-ICE`* algorithm. We provide up to two normalization methods and a target relative error as an input to the compression pipeline. As output, the pipeline returns normalization constants (for each state of each time snapshot) and the TT-cores.

4.1.4 Compression Results

This section presents the results we obtained from compressing TURF simulation data using the `TT-ICE`* algorithm. Specifically, we compare reconstruction of different analysis using the compressed and uncompressed formats to determine if their is significant loss of information. We first discuss the metrics we use to compare the performance of the TT-decomposition briefly and present the results of the compression experiments. We conducted all compression

experiments on a single compute node, with 24 Intel Xeon Gold 8468 processors and 24 GB RAM.

4.1.4.1 Metrics

To evaluate the performance of the TT-decomposition, we use the following metrics: 1) size on disk, 2) compression ratio and 3) compression time.

Size on disk refers to the size of the compressed dataset stored on disk. In addition to the TT-cores, this also includes the normalization constants and other metadata required to reconstruct the original dataset. Compression ratio (CR) is simply the ratio of the number of elements in the original dataset to the number of elements in the compressed dataset (i.e., TT-cores). Compression ratio is simply computed as

$$CR = \frac{\prod_{i=1}^{d} n_i}{\sum_{i=1}^{d} r_{i-1} n_i r_i}, \tag{4.6}$$

where r_i is the i-th TT-rank and n_i is the i-th dimension of the tensor. Note that CR only considers the states included in compression as an input to the compression pipeline. That means, if a subset of all mesh states are compressed, the numerator of CR will be adjusted accordingly.

Finally, compression time is the time taken to compute the TT-decomposition of the dataset using the `TT-ICE`* algorithm. Note that this time neither includes the time taken to normalize the data nor takes into account the time taken to convert the VTK files to NumPy arrays.

4.1.4.2 Compression Experiments

We only consider snapshots from iterations that correspond to the steady-state portion of the simulation for compression. In particular, we consider the snapshots corresponding to iterations $5000 - 12300$ for compression. We group the states with similar units together. For example, number densities are compressed together, electric field potentials in x,y,z directions are compressed together, and the ion current is compressed on its own.

For qualitative analyses, we select a subset of the steady state simulation iterations for the compression analyses. In particular, we consider 20 snapshots, corresponding to the last 1000 iterations of the baseline simulation, where every 50 iterations are saved. We reconstruct snapshots corresponding to these iterations of interest using the relevant slices of the TT-cores and compare against the original snapshots from the dataset. In the main text we report the results with highest qualitative agreement and provide other combinations

that might be of interest to the community in the appendices.

We report the following QoI using the considered snapshots for the volume data: 1) angular profiles of ion current density 2) contours of neutral atom and ion number density, and 3) electric field contours in the plume. We average those plasma quantities over the selected subset of 1000 iterations for volume and compare the the averaged original and averaged compressed datasets. We perform the following analyses for the surface data: 1) floating potential on the thruster surface and 2) total current to the thruster surface from the plasma. The trends of these quantities are compared between the original and compressed datasets for the full simulation. Since we want to capture the transient behavior on the thruster surface, we do not average the surface snapshots. Note that each field in the mesh files can be compressed individually or together with other fields. A separate compression pipeline for each state allows for greater flexibility in choosing input parameters to yield more accurate approximations. We report the normalization and target relative error ε settings that yield QoI with the greatest integrity to the original dataset; supplementary results are discussed in Appendix.

Volume data in structured mesh files The states stored in the structured mesh files are the neutral atom number density, ion number density, ion current, and electric field values along the x, y and z directions in the computational domain across n_s state variables. We reshape this 5-dimensional $60\times50\times50\times8\times n_s$ tensor into $6\times10\times10\times5\times10\times5\times8\times n_s\times1$ after normalizing each state of the snapshot individually. The results in this section compress the neutral and 3 ion species density together for $n_s = 4$. The ion current density is compressed by itself for $n_s = 1$. The three components of the electric field are compressed together for $n_s = 3$. We investigate alternative compression groupings in the Appendix.

For Figures 4.5 and 4.6 below, we compress the number densities, i.e., neutral and ion number density fields ($n_s = 4$). These figures depict the neutral atom and ion number density contours, respectively. The contours correspond to the quantities on the $X - R$ plane, where R describes the distance from the thruster centerline as shown in Figure 4.4. The reconstructed dataset is an average of the reconstructions of iterations $11300 - 12300$ using square root normalization and a target relative error of $\varepsilon = 10^{-2}$. This compression setting results in a compression ratio of $8.075\times$. Table 4.1 present further details on compression performance including compression time, size on disk and TT-ranks.

The reconstructions in Figures 4.5 and 4.6 accurately capture the near-field and far-field particle number densities and the backflow phenomena behind the thruster caused by CEX collisions with sufficient resolution, maintaining integrity of the data trends even after compression. The square root normalization allows for an accurate approximation of both

the densely populated region near the thruster injection plane and the sparsely populated region behind the thruster. Further, a compression time of 135.6 seconds for the dataset is marginal considering potential substantial reduction in analytical computation time and storage savings afforded by the compressed format.

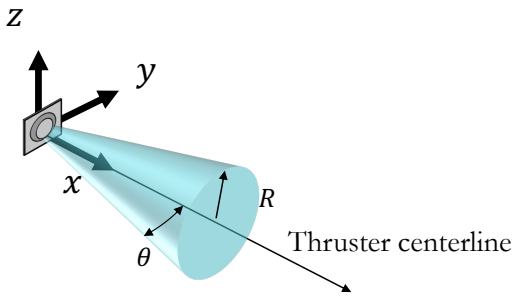


Figure 4.4: Visualisation of the plume in the simulation domain where $R = \sqrt{y^2 + z^2}$. θ is a coordinate to define distance from the thruster injection surface and is defined as $\theta = tan^{-1}(z/x) \approx tan^{-1}(y/x)$. The plume is approximately symmetric about the thruster centerline.

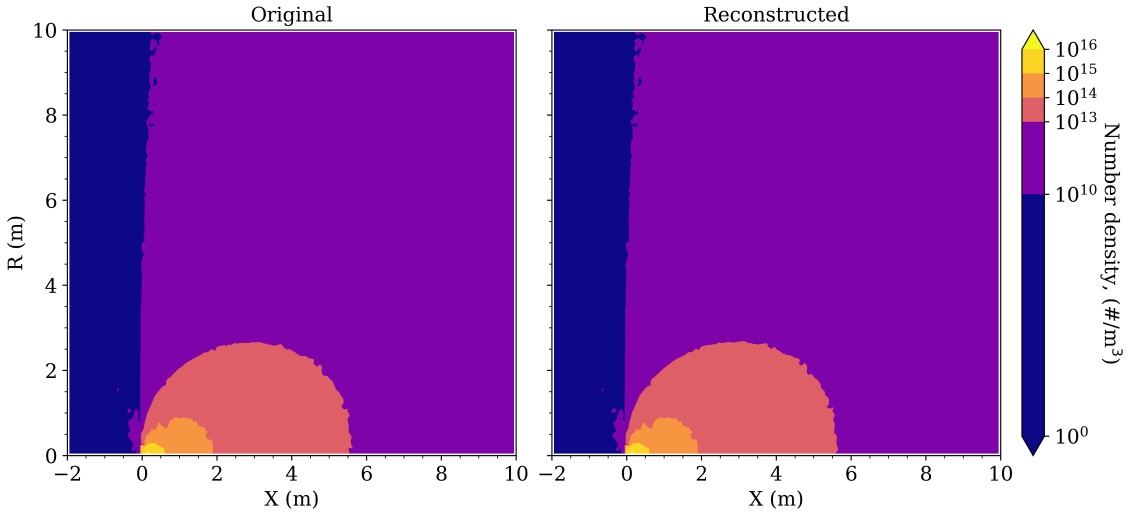


Figure 4.5: (Left) Neutral atom number density contours from the original dataset. (Right) Neutral atom number density from the reconstructed dataset using a square root normalization, and a target relative error $\varepsilon = 10^{-2}$. Neutral and ion number density fields are compressed together. The reconstructed dataset has a compression ratio of 8.08×. These comparisons show that the reconstructed dataset accurately captures near-field and far-field neutral number densities as well as the backflow phenomena behind the thruster caused by CEX collisions.

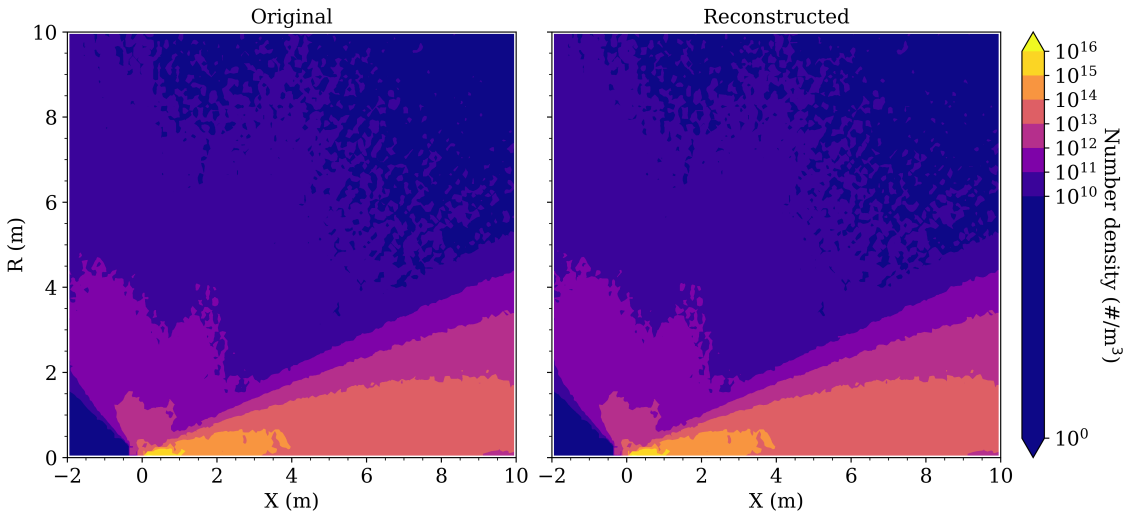


Figure 4.6: (Left) Ion number density contours from the original dataset. (Right) Ion number density contours from the reconstructed dataset using a square root normalization, and a target relative error $\varepsilon = 10^{-2}$. Neutral and ion number density fields are compressed together. As in the neutral number density contours, the reconstructed dataset captures the ion number density contours in the near-field, far-field, and CEX plasma with sufficient resolution.

Table 4.1: Compressed data metrics for neutral and ion number density states for a time averaged snapshot with 3.6 million particles. With a square root normalization combination and a target relative error of $\varepsilon = 10^{-2}$, the TT-decomposition results in states with a reduced size on disk by 8.076×.

	Size	Time (s)	ε	CR	TT-ranks	Latent size
Original	5.5 GB	-	-	-	-	-
Compressed	681 MB	135.6	10^{-2}	8.075	1, 6, 60, 586, 2829, 2590, 535, 71, 34, 1	34

Ion current densities in the plume from thruster injection are obtained directly from the computational ion particles in the TURF simulation. Figure 4.7 shows a current density profile at 100 cm ± 12.5 cm from the thruster injection surface. The reconstructions presented are from different normalization combinations at a fixed target relative error of $\varepsilon = 10^{-4}$. Table 4.1 and Table 4.2 present further details on compression performance including compression time, size on disk and TT-ranks. The reconstructed current density profiles have little to no loss of accuracy close to the thruster centerline. Because at larger plume angles, i.e. further away from the thruster injection surface, there are orders of magnitudes fewer ion particles than in the near-field plume, quantities are prone to more statistical noise and are harder to capture accurately. At large plume angles, the reconstructed dataset deviates from the original with a maximum relative error of 0.615, corresponding to the TT-compression done with a unit vector normalization.

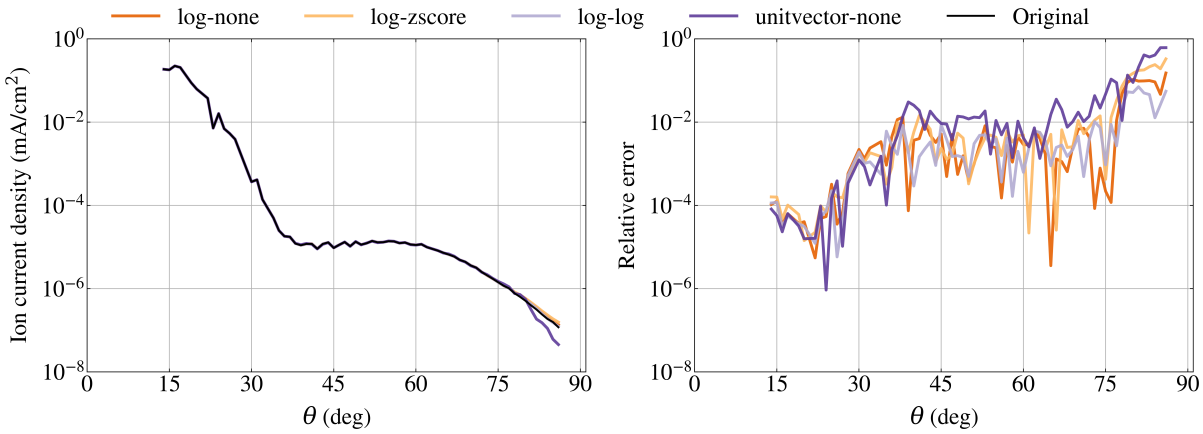


Figure 4.7: (Left) Angular profile of ion current density of the plume at 100 cm ±12.5 cm from the thruster injection plate. (Right) Relative error between the ion current density profiles between the original and reconstructed datasets with various normalization combinations and a fixed target relative error of $\varepsilon = 10^{-4}$. At higher angles, i.e. further away from the thruster injection surface, the reconstructed dataset deviates from the original with a max relative error of 0.615. A logarithmic-logarithmic normalization with target relative error $\varepsilon = 10^{-4}$ results in a compression ratio of 6.23× with relatively low relative error even at the higher plume angles. log: logarithmic normalization, zscore: z-score normalization, unitvector: unit vector normalization, none: no normalization.

Table 4.2: Compressed data metrics for the ion current density state for multiple normalization combinations. Compression of the ion current density is very sensitive to the normalization method selected. log: logarithmic normalization, zscore: z-score normalization, unitvector: unit vector normalization, none: no normalization.

	Size	Time (s)	ε	CR	TT-ranks	Latent size
Original	1.38 GB	-	-	-	-	-
log-none	157 MB	45.89	10^{-4}	8.76	1, 6, 51, 392, 1525, 1027, 265, 131, 131, 1	131
log-zscore	158 MB	46.82	10^{-4}	8.71	1, 6, 51, 392, 1527, 1033, 267, 132, 132, 1	132
log-log	221 MB	62.51	10^{-4}	6.23	1, 6, 53, 443, 1793, 1247, 304, 150, 150, 1	150
unitvector-none	17 MB	13.61	10^{-4}	81.9	1, 6, 41, 219, 636, 206, 70, 35, 35, 1	35

Figure 4.8 considers the the electric field components in the plasma close to the thruster injection plane, i.e. E_x, E_y, E_z ($n_s = 3$). CEX collisions, modeled by DSMC in TURF, result in slow ions that are sensitive to the electric fields in this region. These slow ions can be pulled behind the thruster creating the backflow plasma. We use z-score normalization and $\varepsilon = 10^{-1}$ target relative error for compression. Table 4.3, present further details on compression performance including compression time, size on disk and TT-ranks. These inputs yield a very high $CR = 354,796\times$ with little to no loss in resolution and accuracy of the QoI. Further, a compression time of only 15.31 seconds for a dataset of size 4.2 GB can afford a significant reduction in analysis time. In the near-field plume during steady

state, the ions are accelerated electrostatically; the electric fields are not time-varying. This is supported by the fact that electric field latent space size is 1.

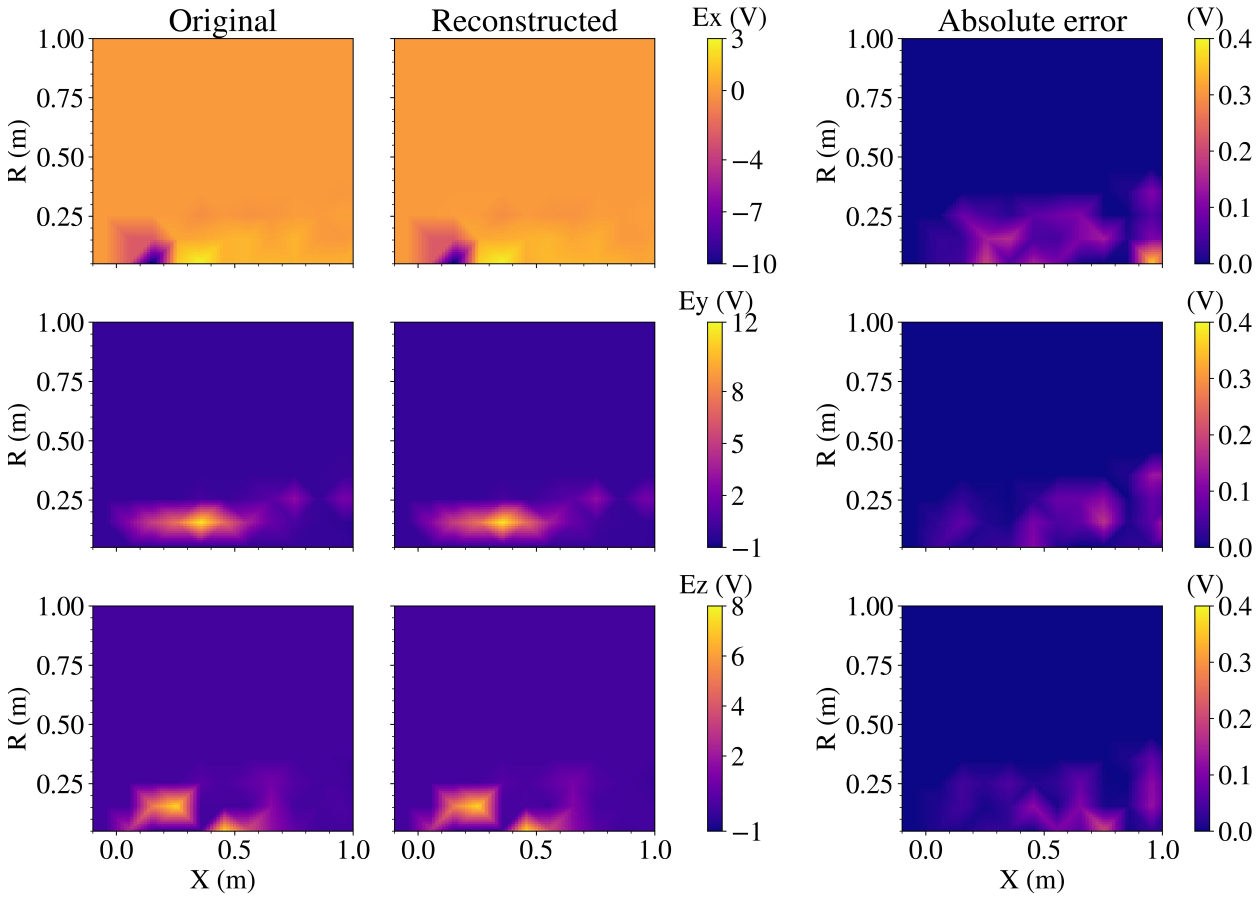


Figure 4.8: (Left) Electric field in the near-field plume region from the original dataset. (Middle) Electric field from the reconstructed dataset using a z-score normalization, and $\varepsilon = 10^{-1}$ target relative error. The components of the electric field are compressed together. This yields a very high $CR = 354,796\times$ as well as high accuracy to the original electric field contours. (Right) shows the absolute error in electric field components. The largest error is 0.356 V in the E_x component.

Table 4.3: Compressed data metrics for electric fields. The components of the electric field are compressed together using a z-score normalization. The electric field data is highly compressible and achieves extreme compression even at relatively high target error levels. The electric fields are not time-varying, and the ions are only accelerated electrostatically, leading to a latent space size of 1 and high compression.

	Size	Time (s)	ε	CR	TT-ranks	Latent size
Original	4.2 GB	-	-	-	-	-
Compressed	35 KB	15.31	10^{-1}	354,796	1, 1, 3, 3, 11, 8, 3, 3, 1, 1	1

Surface data in unstructured mesh files The fields stored in the unstructured mesh files are floating potential, electron current to the surface, ion current to the surface, total current to the surface, and charge accumulation on the surface of the thruster. The surface data has 1100 data-points for each of the 5 QoI. We reshape the 1100×5 surface snapshot array into a $10 \times 10 \times 11 \times 5$ array after normalizing each state of the snapshot individually, similar to the case with volume snapshots. For this class of experiments, we compress all 5 surface states together and report the following derived surface QoI: 1) floating potential integrated over the surface and 2) total current from the plasma to the surface.

Electrons and ions injected by the EP device contribute to the total currents in the plasma. When the surface potential is less than the plasma potential, the electron current to the thruster surface is calculated by integrating the Maxwellian distribution and assuming quasi-neutrality [15]. When the surface potential is larger than the plasma potential and the thin-sheath assumption can be made, the electron current is equal to the saturation current [174]. The ion current to the surface is computed using the electric flux through the surface[16]. The total current to the device surface from the plasma is shown in Figure 4.9. In the steady state portion of the simulation after around iteration 10000, under the assumption of quasi-neutrality, the electron current is equal to the ion current and the total current value asymptotes to 0 mA.

The floating potential on the thruster surface is calculated in TURF using the ion and electron temperatures, plasma potential, and the total current in the plasma. The floating potential on the surface is shown in Figure 4.9.

Figure 4.9 and Table 4.4 show the comparison between the derived surface QoI from the original and compressed datasets and the compression metrics. The reconstructed states, compressed without applying any normalization and using a target error of $\varepsilon = 10^{-6}$, results in a relatively high CR of 5.08×. The electric potential approximation is nearly exact to the original with a relative error of 3.52×10^{-10}. The reconstructed total current quantity closely follows the trend of the original until iteration 10000, after that point, the reconstructions deviate towards a total current of zero. The maximum relative error of 1.56 seen around iteration 13500 can be attributed to the fact that the entire simulation dataset was included in the approximation, including the iterations with transient behavior. Compressing the transient and steady state parts of the simulation separately may yield a better overall approximation performance.

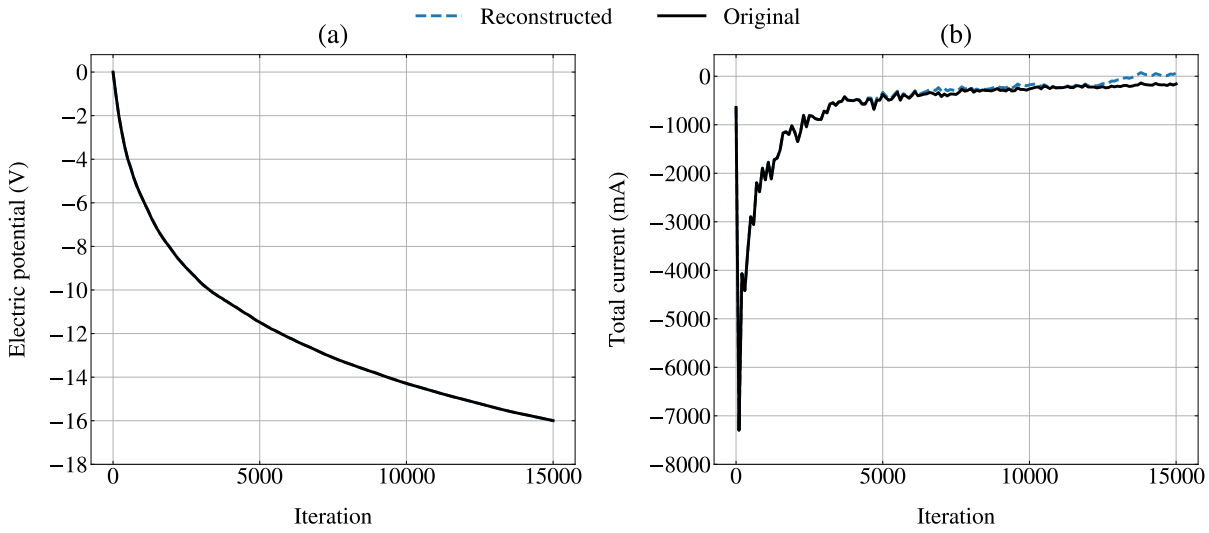


Figure 4.9: Derived surface QoI from the original and reconstructed unstructured mesh data. Compression with no normalization and a target relative error of $\varepsilon = 10^{-6}$ results in a reconstructed dataset with CR = 5.077×. (Left) Electric potential quantities from compressed and reconstructed datasets are nearly one-to-one. (Right) The total current quantity from reconstruction follows the trend of the original quantity with a maximum relative error of 1.56 seen around iteration 13500.

Table 4.4: Compressed data metrics for surface quantity states. The components of the surface mesh data are compressed together without normalization and using a target error of $\varepsilon = 10^{-6}$. Even though the transient part of the simulation is included for this case study, the surface fields turn out to be compressible at a very high speed.

	Size (GB)	Time (s)	ε	CR	TT-ranks	Latent size
Original	6.6 MB	-	-	-	-	
Compressed	1.3 MB	0.39	10^{-6}	5.077	1, 5, 16, 176, 125, 1	125

4.2 Bayesian Inference in Tensor-Network Latent Spaces

Estimating the full state of a complex, high-dimensional system from a limited number of (potentially noisy) observations is a common challenge in many scientific and engineering applications. While direct numerical simulations can produce high-fidelity representations of such systems, practical applications are typically limited to a small number of sensors due to cost, physical intrusiveness, or technical limitations [193, 290, 187, 281]. This gives rise to an inverse problem [253, 243], where the goal is to infer the full state of the system from partial observations.

Bayesian inference provides a principled framework for addressing such inverse problems by combining prior knowledge about the system with observed data to obtain a posterior distribution that reflects our updated beliefs about the system's state [83, 273]. Additionally, Bayesian inference allows us to quantify the uncertainty in our estimates, which is crucial for making informed decisions based on the available information.

However, for systems with high dimensionality, which is typical for discretized PDE-based models, performing Bayesian inference can become computationally prohibitive. The storage and inversion of large covariance matrices, which scale cubically with the number of dimensions, quickly becomes intractable even for moderately discretized systems. This problem is exacerbated when the inference process needs to be executed repetitively, such as in outer loop applications like optimal experimental design [10]. Reducing the state space of system of interest (SoI)s is a common approach to make the inference process more tractable for optimal experimental design [282, 283, 78]. There are several approaches to reduce the computational cost of Bayesian optimization, such as using random forests for dimensionality reduction [58], tailored feature selection [170], or using randomized SVD-based dimensionality reduction [283].

This section demonstrates a solution to this challenge by performing Bayesian inference in a low-dimensional latent space obtained through a tensor network in TT-format. Specifically, we use the TT-ICE* algorithm presented in Section 3.1 to train the tensor network on a set of high-dimensional training snapshots obtained from numerical simulations of the SoI. As a result of this training, we obtain a low-dimensional embedding space and a mapping between the full state space and the embedding space. We then use this learned mapping to perform Bayesian inference in the low-dimensional latent space, achieving a significant reduction in the dimensionality of the problem and, consequently, significantly reducing the computational cost of the inference process.

This section is organized as follows. First, we detail the methodology, explaining how the TT-representation defines a latent space and how the Bayesian inverse problem is formulated within it, both for full and limited sensor observations. Next, we present the experimental setup using a dataset of PDE-driven simulations. Finally, we discuss the results, demonstrating the effectiveness of our approach in accurately reconstructing the full system state from sparse measurements and conclude with the implications of our findings.

4.2.1 Methodology

Our approach integrates a data-driven latent space model obtained using TT-ICE* with a formal Bayesian inference framework. The method is composed of three main steps: 1)

learning a map from the full space to the latent space using TT-ICE*, 2) formulating a linear Gaussian inverse problem that connects the latent variables to the observations, and 3) adapting this formulation for the case of limited sensor observations.

4.2.1.1 Learning the Latent Space with TT-ICE*

We begin with a collection of N high-dimensional snapshots from a system of interest $\mathcal{Y}^i \in \mathbb{R}^{n_1 \times \cdots \times n_d}$, for $i = 1, \ldots, N$. These snapshots are stacked along an auxiliary $(d+1)$-th dimension to form a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times \cdots \times n_d \times N}$ and constitute the training data for our model. We then learn a low-rank representation of $\mathcal{X}$ in TT-format using the TT-ICE* algorithm, resulting in an approximation of the form:

$$\mathcal{X} \approx \mathcal{G}_1 \ {}_3\times_1 \ \mathcal{G}_2 \ {}_4\times_1 \ \cdots \ {}_{(d+1)}\times_1 \ \mathcal{G}_d \ {}_{(d+2)}\times_1 \ \mathcal{G}_{d+1}, \tag{4.7}$$

where $\mathcal{G}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$ are the TT-cores with $r_0 = r_{d+1} = 1$. The first d TT-cores define a mapping $\mathcal{F}_y : \mathbb{R}^{r_d} \to \mathbb{R}^{n_1 \times \cdots \times n_d}$ from the latent space to the full state space, while the last TT-core $\mathcal{G}_{d+1}$ contains the latent representations of each snapshot. Specifically, the i-th snapshot can be reconstructed as:

$$\mathcal{Y}^i \approx \mathcal{F}_y(\theta^i) := \mathcal{G}_1 \ {}_3\times_1 \ \mathcal{G}_2 \ {}_4\times_1 \ \cdots \ {}_{(d+1)}\times_1 \ \mathcal{G}_d \ {}_{(d+2)}\times_1 \ \theta^i, \tag{4.8}$$

where $\theta^i = \mathcal{G}_{d+1}(:, i, :)$ is the latent representation of the i-th snapshot. The mapping $\mathcal{F}_y$ can also be expressed in matrix form as $F_y \in \mathbb{R}^{n_y \times r_d}$, where $n_y = \prod_{i=1}^{d} n_i$, allowing us to write the vectorized form of the snapshot as $y^i = F_y \theta^i$.

4.2.1.2 Formulating the Linear Gaussian Inverse Problem

With the latent space and mapping established, we formulate a linear Gaussian inverse problem. We assume that the latent variable θ follows a Gaussian prior distribution $\theta \sim \mathcal{N}(\mu_{pr}, \Sigma_{pr})$, where the mean μ_{pr} and covariance Σ_{pr} are estimated from the latent representations of the training snapshots. The observation model is defined as:

$$y = F_y \theta + \epsilon, \tag{4.9}$$

where $y \in \mathbb{R}^{n_y}$ is the vectorized full state observation, and $\epsilon \sim \mathcal{N}(0, \Sigma_{meas})$ represents uncorrelated Gaussian measurement noise with covariance $\Sigma_{meas} = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_{n_y}^2)$. Given

an observation y, the posterior distribution of θ can be computed in closed form as:

$$\Sigma_{post} = \left(F_y^\top \Sigma_{meas}^{-1} F_y + \Sigma_{pr}^{-1}\right)^{-1}, \tag{4.10}$$

$$\mu_{post} = \Sigma_{post}\left(F_y^\top \Sigma_{meas}^{-1} y + \Sigma_{pr}^{-1}\mu_{pr}\right). \tag{4.11}$$

The posterior mean μ_{post} provides the maximum a posteriori estimate of the latent variable θ, while the posterior covariance Σ_{post} quantifies the uncertainty in this estimate.

4.2.1.3 Adapting for Limited Sensor Observations

In many practical scenarios, we may not have access to the full state observation y due to limitations in measurement capabilities or computational resources. Instead, we often have a limited number of sensor measurements $y_s \in \mathbb{R}^p$, where $p \ll n_y$. To accommodate this, we introduce a boolean sensor map $S \in \mathbb{B}^{p \times n_y}$ that selects the observed components from the full state. The observation model is then modified to:

$$y_s = SF_y\theta + \epsilon_s, \tag{4.12}$$

where $\epsilon_s \sim \mathcal{N}(0, \Sigma_{meas}^s)$ represents the measurement noise associated with the sensor observations, with covariance $\Sigma_{meas}^s = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_p^2)$. We further simplify the problem by assuming that the measurement noise is uncorrelated and has the same variance across all sensors, i.e., $\Sigma_{meas}^s = \sigma^2 I_p$. The posterior distribution of θ given the limited observations y_s can be computed similarly to the full observation case, yielding:

$$\Sigma_{post}^s = \left((SF_y)^\top (\Sigma_{meas}^s)^{-1} SF_y + \Sigma_{pr}^{-1}\right)^{-1}, \tag{4.13}$$

$$\mu_{post}^s = \Sigma_{post}^s\left((SF_y)^\top (\Sigma_{meas}^s)^{-1} y_s + \Sigma_{pr}^{-1}\mu_{pr}\right). \tag{4.14}$$

To construct the sensor map S, we enforce the following constraints:

$$S_{ij} \in \{0,1\}, \quad \sum_{j=1}^{n_y} S_{ij} = 1, \quad \sum_{i=1}^{p} S_{ij} \leq 1, \tag{4.15}$$

where S_{ij} denotes the element in the i-th row and j-th column of S. The first constraint ensures that each entry of S is binary, the second ensures that each sensor measures exactly one location of the simulation space, and the third ensures that no two sensors measure the same position in the simulated space. Note that the choice of S is a design parameter that can be optimized based on the specific application and available resources.

To summarize, we consider a linear Gaussian Bayesian inverse problem in the TT-latent

space. The unknown is a latent vector $\theta \in \mathbb{R}^{r_d}$ associated with a high-dimensional snapshot. The first d TT-cores define a map $\mathcal{F}_y$ from latent to full states and its matrix form $F_y \in \mathbb{R}^{n_y \times r_d}$, where $n_y = \prod_{i=1}^{d} n_i$. With Gaussian noise $\epsilon \sim \mathcal{N}(0, \Sigma_{meas})$ and a Gaussian prior $\theta \sim \mathcal{N}(\mu_{pr}, \Sigma_{pr})$ learned from the slices of $\mathcal{G}_{d+1}$, the observation model is $y = F_y\theta + \epsilon$. When only a subset of entries is observed, we introduce a boolean sensor map $S \in \mathbb{B}^{p \times n_y}$ and obtain $y_s = SF_y\theta + \epsilon_s$ with $\epsilon_s \sim \mathcal{N}(0, \Sigma^s_{meas})$. Closed-form expressions for the posterior mean and covariance follow from the linear Gaussian update.

4.2.2 Experimental Setup

We validate our approach using a dataset of PDE-driven simulations; namely the self-oscillating gel snapshots, previously described in Section 3.1.4.2. This dataset captures the chaotic dynamics of a hexagonal gel sheet under parametric excitation.

The training dataset of the tensor network is generated by simulating the system for 8000 unique parameter combinations, obtained by discretizing a 3D parameter space. The test set consists of $15,000$ randomly sampled parameter combinations from the same 3D parameter space. Note that the parameters for the test set are not a recombination of the training parameters. The test parameters are sampled from the same continuous parameter space, whereas the training parameters are sampled from a discretized grid.

Each simulation constitutes ten time snapshots of the gel's state (x, y, and z coordinates) on a hexagonal grid of 3367 nodes, resulting in a tensor of size $3367 \times 3 \times 10$. Similar to Section 3.1.4.2, we reshape each simulation into a tensor of size $7 \times 13 \times 33 \times 3 \times 10 \times 1$ and stream these tensors into the TT-ICE* algorithm to train the tensor network.

We train five different tensor networks with varying relative error tolerances $\varepsilon \in \{0.20, 0.10, 0.05, 0.03, 0.01\}$. A lower ε results in a more accurate reconstruction map F_y but yields a larger latent dimension r_d. For each trained model, we simulate state estimation by placing $p \in \{5, 10, 20, 40, 60, 80\}$ sensors randomly on the gel's surface. For this study, we assume that each sensor measures all three coordinate components (x, y, and z coordinates) at its placed node. We repeat the sensor placement and state estimation process 40 times for each combination of ε and p to account for variability due to random sensor placement.

Evaluation Metrics. The performance of the state estimation is quantified by the percentage relative error between the estimated full state $\hat{y}$ and the true state y:

$$\text{Relative Error} = \frac{\|y - \hat{y}\|_2}{\|y\|_2} \times 100\%. \tag{4.16}$$

To ensure robust statistics against outliers, we report the median and the 90th percentile of the relative error across the 15,000 test samples, averaged over 40 sensor placements.

4.2.3 Results and Discussion

The results of the state estimation experiments are summarized in Figure 4.10. The figure shows the relative estimation error as functions of the number of sensors (p) for five different relative error tolerances (ε) used during the training of the tensor network. Specifically, Figures 4.10a and 4.10b show the median relative error for the training and test sets, respectively, while Figures 4.10c and 4.10d show the 90th percentile relative error for the training and test sets, respectively.

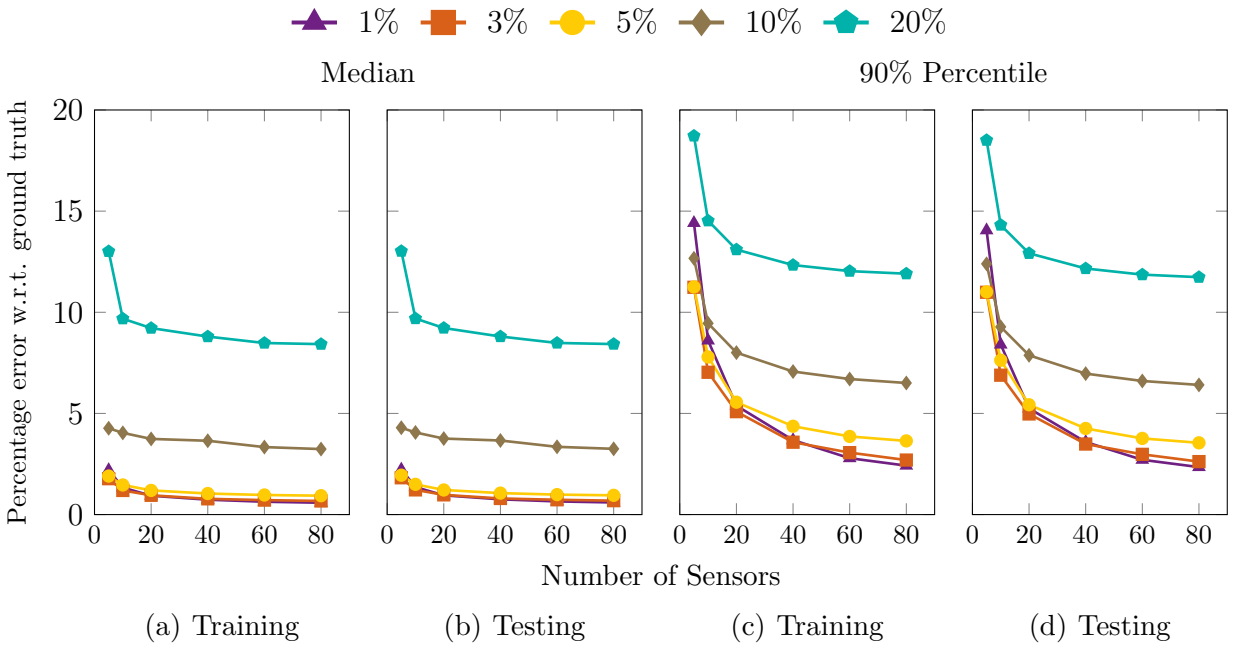


Figure 4.10: Percentage relative error of the estimation of the full system state using the Bayesian inference framework with different tensor networks. Figures (a) and (b) show the median, and Figures (c) and (d) show the 90-th percentile of the relative error. The reported results in this figure are averaged over 40 random sensor placements for each tensor network and number of sensors.

Several key trends are evident from the results. First, the framework successfully estimates the full system state with reasonable accuracy, even with a limited number of sensors. For instance, the most accurate tensor network ($\varepsilon = 0.01$) achieves a median relative error below 5% with just 5 sensors (0.15% of all nodes). This highlights the power of combining a high-fidelity latent space model with Bayesian inference.

Second, as expected, the accuracy of the state estimation improves with an increasing number of sensors. This trend is consistent across all tensor networks, demonstrating that additional measurements provide more information to constrain the posterior distribution of the latent variables. The returns diminish as the number of sensors increases, indicating that beyond a certain point, additional sensors contribute less to improving accuracy.

Third, the choice of the relative error tolerance ε during the training of the tensor network significantly impacts the estimation accuracy. Lower values of ε lead to more accurate reconstruction maps F_y and consequently better state estimation performance. However, this comes at the cost of increased latent dimension r_d, which in turn increases the computational cost of the inference.

Finally, the performance on the test set closely mirrors that on the training set, indicating that the learned latent space generalizes well to unseen parameter combinations. This is crucial for practical applications where the system may operate under conditions not represented in the training data.

In summary, the results demonstrate the effectiveness of performing Bayesian inference in a low-dimensional latent space learned through TT-ICE*. The approach enables accurate state estimation from sparse measurements while maintaining computational tractability, making it a promising tool for real-world applications involving high-dimensional systems. Future work could explore wrapping this framework within an outer loop optimization for sensor placement, further enhancing its practical utility.

4.3 Summary

This chapter presented two applications of TT-ICE in the context of scientific data: compression of scientific simulations and Bayesian inference. In Section 4.1, we demonstrated the effectiveness of TT-ICE for compressing scientific data, achieving compression rates up to $300,000\times$ while preserving the ability to reconstruct high-quality data. In Section 4.2, we showed that the low-dimensional embeddings obtained through TT-ICE* can be used for the downstream task of Bayesian inversion.

The results of Section 4.1 demonstrate the feasibility of using tensor network compression to reduce the size of simulation data generated by electric propulsion simulations. By learning an approximate TT-representation incrementally from the simulation snapshots, we can reduce the data size $8-350,000\times$ without significant sacrifice in the accuracy. This reduction may also allow for increasing the frequency at which simulations are saved.

In addition to providing storage savings, tensor network compression allows for efficient analysis of the simulation data. The analysis outputs generated from the original and com-

pressed datasets are compared to quantify the loss in simulation output fidelity from tensor network compression. The results show that the compressed datasets can capture the key features of the original datasets with high fidelity. Furthermore, the latent representation that is learned during the compression process can be used in reduced-order modeling [79]. Future work will also focus on extending the use of tensor network compression to other types of EP simulations [171] and to develop a tool that can work in tandem with simulation codes to progressively compress their outputs.

The results of Section 4.2 demonstrate that both the tensor networks and the low-dimensional embeddings obtained through TT-ICE* can be used for Bayesian inference. The results also provide evidence that the generalization capabilities of the tensor networks allow for accurate inference for the unseen test data. Although this line of work is still in its early stages, the results are promising and suggest that tensor networks can be used to enable or accelerate outer loop Bayesian optimization or uncertainty quantification tasks. Future steps for this work will include integrating tensor networks into the optimization loop to fully exploit the properties of tensor networks.

4.4 Acknowledgements

The research presented in Section 4.1 was supported in part through computational resources provided by Advanced Research Computing at the University of Michigan. Funding for this work was provided by NASA in part through the Joint Advanced Propulsion Institute (JANUS) under grant number 80NSSC21K1118. Additionally, we would like to acknowledge support by Los Alamos National Laboratory under the project "Algorithm/Software/Hardware Co-design for High Energy Density applications."

CHAPTER 5

Applications Considering Image and Video Data

This chapter discusses the application of tensor decompositions in the context of image and video data, focusing on the use of tensor decompositions for data compression, behavioral cloning, and generative modeling. We can summarize the key contributions of this chapter as follows:

1. Empirical demonstration of the effectiveness of HT-RISE for compressing video streams and high-resolution images from Unreal Engine 5 environments, achieving compression commensurate with JPEG encoding, while preserving the ability to reconstruct high-quality images and videos. (Section 5.1)

2. Empirical demonstration that the low-dimensional embeddings obtained through TT-ICE* can be used for the downstream task of behavioral cloning. (Section 5.2)

3. Empirical demonstration of a reduction in training time ($2.6-9.6\times$ faster when trained on GPU), while achieving comparable behavioral cloning accuracy and gameplay scores compared to autoencoder-based approaches for extracting a latent space. (Section 5.2)

4. Empirical demonstration that the low-dimensional embeddings obtained through HT-RISE can be effectively used for generative modeling tasks, such as denoising diffusion models. (Section 5.3)

5.1 Compression of High-Resolution Images

This section discusses the application of incremental tensor decompositions in the context of high-resolution image streams from high-fidelity simulation environments created in Unreal Engine 5. We demonstrate how tensor network mappings can generalize to unseen

environments by learning a structure from training images. Furthermore, we show how the extracted latent representations can challenge an industry-standard lossy image compression algorithm, JPEG, while maintaining a high reconstruction quality.

5.1.1 The Unreal Engine 5 Environments

The **U**nreal **E**ngine 5 (UE5) environments considered in this section are high-fidelity environments created as part of a collaboration with the Oakland University's Intelligent Robotics Laboratory. The environments are generated to simulate the TAS 500 real-world off-road driving environment [175], which is a high-fidelity simulation environment that can be used for testing and training autonomous vehicles.

To demonstrate the efficacy of tensor networks in this task, we consider two different environments in this section: a forest environment (Figure 5.1a) and a winter environment (Figure 5.1b). The environments are created using the Unreal Engine 5 game engine, which is a high-fidelity game engine that can be used to create realistic simulation environments. The environments include a variety of terrains and weather conditions, which can be used to test the robustness of the compression algorithms and the generalization capabilities of the learned latent representations.



(a) Forest Environment (b) Winter Environment

Figure 5.1: High-fidelity Unreal Engine 5 environments used for creating the high-resolution image streams. The environments are designed to simulate the TAS 500 real-world off-road driving environment, and they include a variety of terrains and weather conditions.

The camera streams recorded from these environments are color frames with a resolution of 960×540 pixels, that are captured at a rate of 60 frames per second. The camera streams are recorded using a front-facing camera with 90° field-of-view mounted on the hood of an off-road vehicle that drives through the environments. Figure 5.2 shows example frames from the high-resolution camera streams recorded in the Unreal Engine 5 environments.



(a) Forest Environment (b) Winter Environment

Figure 5.2: Example frames from the high-resolution camera streams recorded in the Unreal Engine 5 environments. The frames are captured at a resolution of 960×540 pixels and at a rate of 60 frames per second. The streams are recorded using a front-facing camera with 90° field-of-view mounted on the hood of a vehicle that drives through the environments.

Using an expert controller, we drive on a pre-defined path through the environments, recording a total of 16 demonstrations in each environment. Each demonstration takes approximately 80 seconds to complete, resulting in a total of approximately 76,800 frames per environment. The demonstrations are split into 12 training demonstrations and 4 test demonstrations, with the training demonstrations used to learn the latent representations and the test demonstrations used to evaluate the compression performance and the reconstruction quality.

5.1.2 Compressing Camera Streams

In this section, we present the approach we use to compress the high-resolution camera streams recorded in the Unreal Engine 5 environments. We use the HT-RISE algorithm (Algorithm 4) to compress the camera streams. The hierarchical structure resulting from the HT-RISE algorithm is shown to be effective for maintaining the structure of satellite images in Section 3.3.3.2.

To compress the camera streams, we first downsize the frames to a resolution of 384×224 in a way similar to the approach used in [20]. This downsampling is done to reduce the computational cost of training the downstream behavioral cloning model, which is not the focus of this section. The downsampling is performed by using a bicubic interpolation method, which is a common technique for resizing images while preserving the quality of the image. The downsampled frames are then reshaped into a 7-way tensor with dimensions $4 \times 8 \times 7 \times 24 \times 16 \times 3 \times 1$, where the first five dimensions are reshapings of the spatial dimensions of the frames, and the last two dimensions represent the color channels (RGB) and the batch

size, respectively. We accumulate the frames along the last dimension, treating individual frames as a single batch. Before compression, we normalize the frames by dividing the pixel values by 255, which is the maximum pixel value for RGB images. This normalization step is important for ensuring that the pixel values are in the range $[0, 1]$, which is a common practice in image processing and machine learning.

Table 5.1: Compression results for the high-resolution camera streams recorded in the Unreal Engine 5 environments. The compression is performed using the HT-RISE algorithm. The compression ratios in this table are elementwise compression ratios, which are defined as the ratio of the number of elements in the original tensor to the number of elements in the compressed tensor. Rel. Error is the relative error threshold set for the HT-RISE algorithm. Time is the time taken to compress all the frames in the respective training datasets. Latent Shape is the shape of the latent representation of a single frame obtained after compression. Note that the compression reported here is with respect to the downsampled frames, which have a resolution of 384×224 pixels.

Env.	**Rel. Error**	**Time (s)**	**Compression**	**Latent Shape**
Forest	0.30	906.9	99.04×	42×62
	0.25	986.9	19.24×	90×149
	0.20	1,079	7.33×	137×257
	0.15	1,157	4.16×	174×356
	0.10	1,240	2.89×	203×440
	0.05	1,378	2.05×	219×574
Winter	0.30	493.7	716.0×	15×24
	0.25	544.9	102.4×	40×63
	0.20	683.3	25.48×	73×135
	0.15	788.1	9.79×	113×233
	0.10	878.4	4.61×	155×361
	0.05	1,064	2.30×	206×544

Figure 5.3 shows the reconstructions from compressed frames for both the forest and the winter environments. The compression results are presented in Table 5.1, where we show the compression ratios achieved for different relative error thresholds. Note that we can achieve as high as 19.24× compression for the forest environment and 102.4× compression for the winter environment, while maintaining a substantial portion of the structure in the images. Please note that after reconstruction, we multiply the pixel values by 255 to restore the original pixel value range and clip the values to the range $[0, 255]$ to ensure that the pixel values are valid RGB values. Finally, we reshape the reconstructed frames back to their original shape of 384×224 pixels for proper visualization.

One benefit of choosing tensor networks for compression is their ability to generalize to unseen environments. We test that ability by training tensor networks on one of the environments (such as the forest environment) and using the trained tensor networks to map



(a) Downsampled (b) $\varepsilon = 0.05$ (c) $\varepsilon = 0.15$ (d) $\varepsilon = 0.25$

Figure 5.3: Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments. The first column shows the downsampled frames, and the subsequent columns show the reconstructions from the compressed frames using different relative error thresholds. First row shows the reconstructions from the forest environment, and the second row shows the reconstructions from the winter environment. The frames are downsampled to a resolution of 384×224 pixels, and the compression is performed using the HT-RISE algorithm. The relative error thresholds used for compression are shown in the captions of the respective columns. Note that the first column shows the downsampled frames, which are used as input to the tensor networks. The compression ratios achieved are shown in Table 5.1.

the frames from the other environment (such as the winter environment) into the latent space. The resulting latent representations are then used to reconstruct the frames from the other environment, which is shown in Figure 5.3.

In Figure 5.4, we show examples of the cross-environment generalization performance of the mappings learned from the HT-RISE algorithm. Note that for low relative error thresholds (such as $\varepsilon = 0.05$), the reconstructions from the compressed frames are accurate to the point that the reconstructions are visually indistinguishable from the original frames. As the relative error threshold increases, the reconstructions become less accurate, first losing the color accuracy (specifically the reconstructions from the forest environment) and then at higher relative error thresholds (such as $\varepsilon = 0.25$) losing the structure of the images as well.

A noteworthy observation regarding cross-environment generalization is the tensor networks' ability to reconstruct objects in the projected images that are not even present in their training environment. For example, in addition to the discrepancy in the type of the trees in the forest and winter environments, the winter environment contains large boulders that are not present in the forest environment. Despite this substantial difference, the tensor networks trained on the forest environment are able to reconstruct the boulders in the winter environment, as shown in Figure 5.4. This further underlines the ability of the tensor networks to learn the structure of the images and generalize to unseen environments, which

shows promise for using tensor networks for compressing high-resolution image streams in a variety of environments.



(a) Downsampled (b) $\varepsilon = 0.05$ (c) $\varepsilon = 0.15$ (d) $\varepsilon = 0.25$

Figure 5.4: Cross-environment generalization performance of the mappings learned from the HT-RISE algorithm. The first column shows the original downsampled frames. **Top row:** Reconstructions of forest frames using a model trained on the winter environment. **Bottom row:** Reconstructions of winter frames using a model trained on the forest environment. The subsequent columns show reconstructions at different relative error thresholds (ε).

In addition to qualitative comparisons, we also evaluate the compression performance by comparing the quality of the reconstructions to the original downsampled frames using well established quantitative metrics such as the mean squared error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index Measure (SSIM). To provide context for the quantitative evaluation, we define the three metrics used:

Mean Squared Error (MSE): The MSE measures the average of the squared differences between the original and reconstructed pixel values:

$$\text{MSE} = \frac{1}{n_{frames}} \sum_{i=1}^{n_{frames}} \|\mathcal{Y}^i - \hat{\mathcal{Y}}^i\|_F^2 \tag{5.1}$$

where $\mathcal{Y}^i$ is the original frame, $\hat{\mathcal{Y}}^i$ is the reconstructed frame, n_{frames} is the total number of frames, and $\|\cdot\|_F$ denotes the Frobenius norm. Lower MSE values indicate better reconstruction quality.

Peak Signal-to-Noise Ratio (PSNR): The PSNR is expressed in decibels and measures the ratio between the maximum possible power of a signal and the power of corrupting noise:

$$\text{PSNR} = 10 \log_{10} \left(\frac{\text{MAX}_I^2}{\text{MSE}} \right) \tag{5.2}$$

where MAX_I is the maximum possible pixel value of the image (255 for 8-bit images). Higher

PSNR values indicate better reconstruction quality.

Structural Similarity Index Measure (SSIM): The SSIM [279, 278] assesses the visual impact of three characteristics of an image: luminance (l), contrast (c), and structure(s). The SSIM index is calculated between two windows of pixel values x and y of common size, from corresponding locations in two images to be compared. The individual components l, c, s are defined as follows:

$$l(x,y) = \frac{2\mu_x\mu_y + c_1}{\mu_x^2 + \mu_y^2 + c_1}, \quad c(x,y) = \frac{2\sigma_x\sigma_y + c_2}{\sigma_x^2 + \sigma_y^2 + c_2}, \text{ and } \quad s(x,y) = \frac{\sigma_{xy} + c_3}{\sigma_x\sigma_y + c_3} \tag{5.3}$$

where μ_x and μ_y are the means of patches x and y, σ_x^2 and σ_y^2 are their respective variances, σ_{xy} is their covariance, and c_1, c_2, and c_3 are constants to avoid division by zero. The SSIM index between patch x and y is then calculated as:

$$\text{SSIM}(x,y) = l(x,y)^\alpha \cdot c(x,y)^\beta \cdot s(x,y)^\gamma. \tag{5.4}$$

These SSIM values can be aggregated across the full images by averaging or other variations. For an image, it is typically calculated using a sliding Gaussian window of size 11×11 or a block window of size 8×8. We use the default settings for the constants c_1, c_2, c_3, as well as the exponents α, β, and γ in our evaluation, following the python implementation available from the `scikit-image` library [265] SSIM values range from -1 to 1, where 1 indicates perfect similarity, 0 indicates no similarity, and -1 indicates perfect anti-correlation.

Using these metrics, we evaluate the reconstruction quality of the compressed frames for different relative error thresholds. The evaluation is performed on the test datasets, which consist of 4 demonstrations from each environment. The results are shown in Table 5.2.

In parallel with our observations from the visual comparisons, we observe that the reconstructed frames from the low relative error thresholds (such as $\varepsilon = 0.05$) achieve very high SSIM values (0.91 for both environments), indicating that the reconstructions preserve the structure of the images. The sharp degradation of the reconstruction quality is observed as the relative error threshold increases, with the SSIM dropping to 0.39 for the forest environment and 0.43 for the winter environment at the highest relative error threshold ($\varepsilon = 0.30$). Recall from Table 5.1 that for $\varepsilon = 0.30$, we achieve a compression ratio of 99.04× for the forest environment and 716.0× for the winter environment. This points to the limits of the compression algorithm through the trade-off between the compression ratio and the reconstruction quality for usability of the latent representations in downstream tasks.

Table 5.2: Quantitative evaluation of the reconstruction quality for the high-resolution camera streams recorded in the Unreal Engine 5 environments. The evaluation is performed using the mean squared error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index Measure (SSIM). The results are shown for different relative error thresholds used in the compression. Note that a higher PSNR and SSIM indicate a better reconstruction quality, while a lower MSE indicates a better reconstruction quality.

Env.	**Rel. Error**	**MSE (↓)**	**PSNR (↑)**	**SSIM (↑)**
Forest	0.30	75.35	29.37	0.39
	0.25	68.15	29.82	0.54
	0.20	60.99	30.31	0.69
	0.15	52.59	30.97	0.80
	0.10	41.78	31.98	0.88
	0.05	34.05	32.90	0.91
Winter	0.30	76.05	29.32	0.43
	0.25	68.50	29.78	0.51
	0.20	58.69	30.45	0.65
	0.15	48.90	31.25	0.76
	0.10	38.31	32.31	0.85
	0.05	27.58	33.73	0.91

5.1.2.1 The Challenge of On-Disk Compression for Image Data

Note that we have only considered the compression in terms of number of elements in the tensor, which does not take the space required to store the latent representations into account. In the case of scientific data, the original data is often stored as double-precision floating point numbers (also known as FP64), which requires 8 bytes per element. This creates a parallel between the elementwise compression ratio and the resulting space reduction on disk. However, in the case of image data, the original data is often stored as unsigned 8-bit integers (also known as UINT8), which requires only 1 byte per element. This means that the elementwise compression ratio does not directly translate to the space reduction on disk, as the latent representations are often stored either as double-precision or single-precision floating point numbers, which require 4 or 8 bytes per element. This means that the space reduction on disk is often much lower than the elementwise compression ratio, if not lower than 1.

This section explores the potential ways of bridging the gap between the elementwise compression ratio and the space reduction on disk for image data. One such way is to quantize the latent representations to UINT8, which would allow us to store the latent representations in a much more space-efficient manner.

Naive Quantization of Latent Representations To perform the quantization, we first subtract the minimum value of the latent representations from each element, then we scale the values to the range $[0, 255]$ by multiplying by a scaling factor. Note that the scaling factor depends on the maximum value of the shifted latent representations and is specific to each frame. After scaling, we round the values to the nearest integer and cast the arrays to UINT8. This process effectively quantizes the latent representations to 8 bits per element, which allows us to store the latent representations in a much more space-efficient manner. However, this naive approach of quantizing the latent representations to UINT8 does not work well in practice. As shown in Figure 5.5, direct quantization of the latent representations causes visual artifacts in the reconstructed images, incurring a significant loss in the reconstruction quality.



(a) Original frame (b) FP64 latents (c) Quantized latents

Figure 5.5: Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments using naive quantization of the latent representations. First image shows the original downsampled frame, second image shows the reconstruction from the original FP64 latent representations, and the third image shows the reconstruction from the quantized latent representations. Following this naive approach to quantization results in significant visual artifacts in the reconstructed images.

Outlier-Aware Quantization of Latent Representations When we have a closer look at the FP64 latent representations, we observe that a large portion (roughly $95-99\%$) of the values in the latent representations are close to zero, with a very small number of outliers that have much larger values in magnitude. This means that the naive quantization process can not capture the structure of the latent representations, as the majority of the values are close to zero and the outliers are not properly represented. Instead, we propose a mopre sophisticated, outlier-aware approach to quantization, which is illustrated in Figure 5.6. The idea is to split the latent representations into two parts: the outliers and the rest of the values. The outliers are defined as the values that are above a certain threshold, which is usually set as the values outside the range $[-1, 1]$. The rest of the values are then quantized to UINT8, while the outliers are stored as FP64 values as sparse arrays. This allows us to

achieve a much higher compression ratio while maintaining a high reconstruction quality, as shown in Figure 5.7.

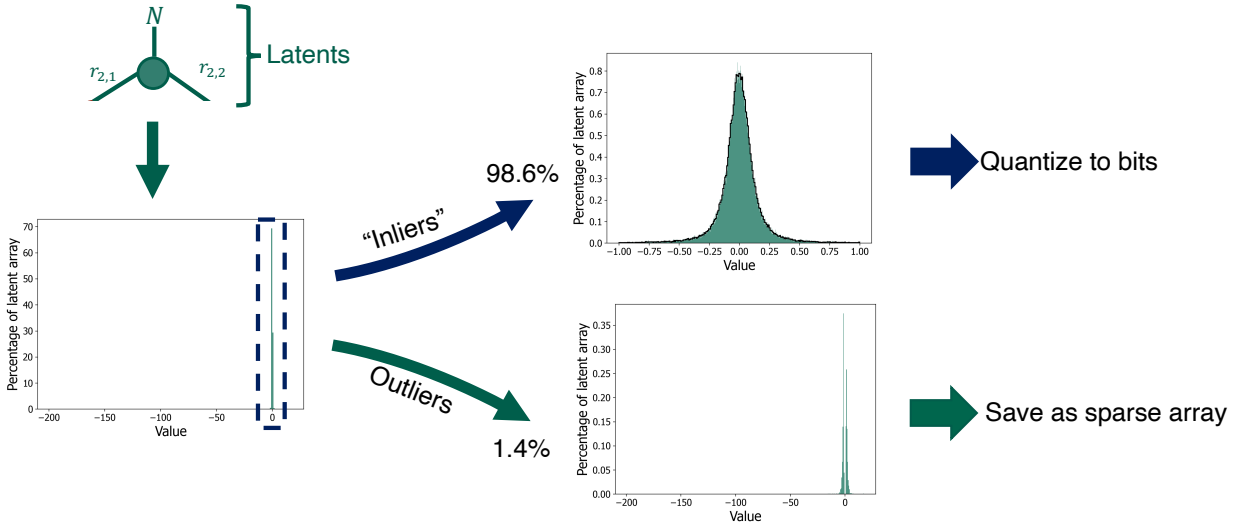


Figure 5.6: Schematic of the outlier-aware quantization approach for the latent representations. The latent representations are split into two parts: the outliers and the rest of the values. The outliers are defined as the values that are above a certain threshold, which is usually set as the values outside the range $[-1, 1]$. The rest of the values are then quantized to UINT8, while the outliers are stored as FP64 values as sparse arrays. This allows us to achieve a much higher compression ratio while maintaining a high reconstruction quality.

The improvement achieved by the outlier-aware quantization approach is not only visual, but also quantitative. For example, for the $\varepsilon = 0.15$ relative error threshold used in Figure 5.7, the MSE of the reconstruction from the outlier-aware quantization is 48.91, while the MSE of the reconstruction from the FP64 latent representations is 48.90. Additionally, the PSNR of the reconstruction from the outlier-aware quantization is 31.24, while the PSNR of the reconstruction from the FP64 latent representations is 31.25. Finally, the SSIM of the reconstruction from the outlier-aware quantization is 0.76, while the SSIM of the reconstruction from the FP64 latent representations is 0.76. This shows that the outlier-aware quantization approach offers additional on-disk compression without resulting in a notable loss in the reconstruction quality, which is a significant improvement over the naive quantization approach. As for the on-disk comparison, the combined storage requirement for the outlier-aware quantization approach is around 30 kB per frame, while the original JPEG compression algorithm achieves a size of around 36 kB per frame for the same relative error threshold.[1] This means that the outlier-aware quantization approach achieves on-disk

[1] sparse FP64 array for outliers with size 4 kB and dense UINT8 array for values between $[-1, 1]$ with size 26 kB



(a) Original frame (b) FP64 latents (c) Naive quant. (d) Outlier-aware quan.

Figure 5.7: Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments using outlier-aware quantization of the latent representations. First image shows the original downsampled frame, second image shows the reconstruction from the original FP64 latent representations, third image shows the reconstruction from the naive quantization of the latent representations, and the fourth image shows the reconstruction from the outlier-aware quantization of the latent representations. The outlier-aware quantization approach achieves much closer reconstruction quality to the original FP64 latent representations.

compression commensurate with the JPEG compression algorithm, while providing a useful latent representation that can be used for downstream tasks.

Storing Only the Outliers of the Latent Representations Taking the outlier-aware quantization approach one step further, we can also consider the case where we only store the outliers and reconstruct the rest of the values as zeros. This means that we only store the values that are beyond the threshold, and we fill the rest of the values as zeros during reconstruction. Figure 5.8 shows the reconstruction results for this approach, which we refer to as "Just outliers". Note that even though we only store the outliers, the structure of the frame is still mostly preserved. Another important thing to keep in mind is that the range of the outliers is a tunable parameter, and we can adjust it to achieve different reconstruction qualities. In this case, we set the range to $[-0.3, 0.3]$ and only store the outliers that are beyond this range. This results in a sparse array of 30 kB per frame, which is significantly smaller than the original FP64 latent representations.

The visual degradation for the "Just outliers" approach is not as severe as the naive quantization approach, but it still manifests itself in the quantitative metrics. For the $\varepsilon = 0.15$ relative error threshold used in Figure 5.8, the MSE of the reconstruction from this approach is 63.43, while the MSE of the reconstruction from the FP64 latent representations is 48.90. Additionally, the PSNR of the reconstruction from this approach is 30.11, while the PSNR of the reconstruction from the FP64 latent representations is 31.25. Finally, the SSIM of the reconstruction from this approach is 0.59, while the SSIM of the reconstruction from the FP64 latent representations is 0.76. This shows that even though we achieve a significant compression on disk by only storing the outliers, we lose additional reconstruction quality in the process. We summarize the quantitative comparison of the quantization approaches



(a) Original frame (b) FP64 latents (c) Outlier-aware (d) Just outliers

Figure 5.8: Reconstruction results for the high-resolution camera streams recorded in the Unreal Engine 5 environments using just the outliers of the latent representations. First image shows the original downsampled frame, second image shows the reconstruction from the original FP64 latent representations, third image shows the reconstruction from the outlier-aware quantization of the latent representations, and fourth image shows the reconstruction from just the outliers of the latent representations. For the "Just outliers" approach, we only stored the outliers beyond $[-0.3, 0.3]$, which results in a sparse array of 30 kB per frame.

in Table 5.3.

Table 5.3: Quantitative evaluation of the reconstruction quality for the high-resolution camera streams recorded in the Unreal Engine 5 winter environment using different quantization approaches at $\varepsilon = 0.15$. The evaluation is performed using the mean squared error (MSE), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index Measure (SSIM). The results are shown for different relative error thresholds used in the compression. Note that a higher PSNR and SSIM indicate a better reconstruction quality, while a lower MSE indicates a better reconstruction quality.

Quantization meth.	**MSE (↓)**	**PSNR (↑)**	**SSIM (↑)**
FP64 latents	48.90	31.25	0.76
Naive quant.	94.19	28.39	0.23
Outlier-aware quant.	48.90	31.24	0.76
Just outliers	63.43	30.11	0.59

This section demonstrated the efficacy of tensor networks for compressing high-resolution image streams, while also addressing the challenge of on-disk compression for image data using tensor decompositions. Next, we explore the use cases of the compressed latent representations in downstream tasks. The first use case we consider will be the use of the compressed latent representations for behavioral cloning.

5.2 Behavioral Cloning using Tensor-Network Embeddings

Learning by imitation is a paradigm that enables teaching complex tasks through observation of expert behavior [115] including object manipulation [228], robotic locomotion [276], and

autonomous driving [37]. Several approaches for learning from data have been proposed, including behavioral cloning and inverse reinforcement learning.

This section considers the task of behavioral cloning, which is a supervised learning approach that aims to learn a map between expert demonstrations and actions taken by the expert. Specifically, we consider the task of learning from frame-action pairs in the context of video games, where the goal is to learn a map from a sequence of video frames to the actions taken by an expert player. The learned map can then be used to predict actions for new video frames, thereby enabling the agent to play the game without explicit programming of the game rules or strategies.

In this context, learning directly from raw pixel data is computationally challenging due to the high resolution of game frames. This challenge is often addressed by downsampling the frames and usage of large-scale computational resources. An illustrative example is the OpenAI VPT [20] model trained for the game of Minecraft, which first downsamples the original frame size of 640×360 to 128×128. Furthermore, the model is trained using $70,000$ hours of gameplay, has half a billion parameters, and is trained for nine days on 720 NVIDIA V100 GPUs.

One approach to tackle the computational challenges of using raw video frames as input is to provide hand-tuned features (such as movement, location, terrain, visibility, etc.) rather than, or in addition to, raw pixel data. With this additional information, important elements of the game are provided and thus do not have to be learned solely from video data. This is the approach taken by some reinforcement learning and imitation learning works, such as [29] for Dota 2 and [272] for Starcraft II. However, such an approach requires human effort and insight in identifying traits that would be useful for a learning algorithm and developing a means for extracting this data. As an alternative to defining hand-crafted features, there have been various approaches proposed for learning a latent space representation via self-supervised learning, where auxiliary tasks are used to train a neural network that maps high-dimensional input data into a lower dimensional space [53]. In this setting, autoencoder-based deep learning architectures have been a common method of choice for identifying this latent space [126, 38, 50, 53, 12], although other approaches have also been explored [224, 197, 53].

This section demonstrates that the latent space representations obtained through a tensor network approach, specifically the Tensor Train (TT) format, can be used for the downstream task of behavioral cloning and results in a significant reduction in end-to-end training time, without causing any setbacks in the action prediction accuracy. Specifically, we use

Note that Section 5.2 is based off of the work "Low-rank tensor-network encodings for video-to-action behavioral cloning." published at Transactions on Machine Learning Research.

the TT-ICE* algorithm presented as Algorithm 2 to learn an error-guaranteed latent space representations from the video frames of ATARI games [48].

Several works have found that some layers of trained neural networks can be replaced with tensor networks with little to no degradation in accuracy while improving the computational and memory costs of running these networks [185, 84, 294, 133, 152, 227]. In this section, we test whether encodings provided by neural networks can similarly be replaced by tensor networks. However, in contrast to the above works, we train the tensor network encodings directly, instead of first training a neural network encoding then replacing it (or components of it) with tensor networks.

In this section, we demonstrate that this approach does provide a practical and highly effective alternative to deep-learning based encodings by reducing the requirements of hyper-parameter tuning, thereby offering shorter training times for behavioral cloning. Moreover, the TT-encoding-based predictors achieve similar or higher behavioral cloning accuracies than predictors built on encodings from a convolutional autoencoder (AE) or variational autoencoder (VAE) [134]. Although other neural network approaches for feature extraction are available (see [53] for examples), we chose AEs and VAEs since like the TT, these are trained to minimize a loss that is related to reconstruction error.

5.2.1 Related Works

In machine learning literature, AEs and VAEs are commonly used as feature extraction tools. [205] uses a convolutional autoencoder (CAE) to extract features from images of a tactile sensor. The extracted features are then used to estimate contact object shape, edge position, orientation and indentation depth using shallow neural networks. [291] extracts features from gearbox vibration data using CAE and uses the extracted features to identify fault modes. [203] uses CAE to compress and denoise high dimensional images to a smaller size and then use the encoded images for classification, [297] uses a stacked AE to extract features from hyperspectral images and use those features in classification, and [40] uses AE-extracted-features to determine the level of traffic congestion from video frames.

There is also increased attention to transformer based methods. [287] uses a transformer encoder to generate pair-wise relations for the objects in the image and uses a transformer decoder to generate captions. [131] uses a transformer based architecture as a feature extraction step in dual arm robot manipulation. However, training complex architectures like transformers increases the computational cost and data requirements even further. In [131], it is reported that 8 NVIDIA V100 GPUs were used in training.

Although tensor methods and neural networks are both capable of feature extraction,

the question of whether the nonlinearity of neural networks can lead to features that are better suited for downstream tasks than the multilinear tensor methods remains. Previous studies indicate that the increased expressiveness of neural networks might be excessive for certain tasks and computationally expensive compared to similar tensor-based methods. Several works have explored compressing portions of large neural networks after training, including fully connected layers [185] and convolutional layers [84, 294, 133, 152, 227]. The fine-tuning of large pre-trained neural networks can also be simplified by exploiting low rank structure; [111] explores using low-rank matrices to update the weights of large language models when fine-tuning these models for downstream tasks. Other work has also explored using tensor operations directly within the network architecture itself during training as a means for parameter reduction and model compression; examples include Tensor Train Recurrent Neural Networks [288], LR-TT-DNN [207], Tensor Contraction Layers [139], Tensor Regression Layers [139], and Higher-Order Tensor RNNs [292]. The ability to replace portions of deep neural networks with tensor networks either during or after training suggests that alternative representations may exist that do not require as much computational power and time to train, while not sacrificing accuracy. However, such works propose embedding tensor networks *within* neural networks and therefore require optimization and fine-tuning to capture the nonlinearity of the compressed layer(s) accurately. As such, they don't mitigate the issues surrounding the complexity of optimization approaches.

In this section, we explore using a tensor network in TT-format as an alternative to neural network based approaches, where the TT is used to extract features from video frame data for behavioral cloning. This approach is possible through incremental algorithms (such as the TT-ICEalgorithm [7]) with approximation guarantees, which provide the basis for memory efficient training of these models (in contrast to TT-SVD of [191], or other optimization-based incremental approaches). In comparison to using neural network based encodings like convolutional AEs and VAEs, we show that using a TT leads to significantly faster training times without sacrificing behavioral cloning accuracy.

5.2.2 Methodology

The overarching objective is to extract a *game-specific* playing strategy from video demonstrations of an experienced agent. We first describe our learning framework; then we present the methods to extract a low-dimensional representation and discuss how TT-decomposition fits within this setting. Finally, we briefly present the action predictor that maps latent representations to actions.

The imitation learning framework we use is shown in Figure 5.9. It consists of two

parts: an encoding transformation (labeled "Frame encoder" in Figure 5.9) and an action predictor ("Action predictor" in Figure 5.9). The encoding transformation serves to reduce the dimensionality of the space so that the action predictor can potentially be learned more efficiently. In this section we aim to demonstrate that we accelerate computation of the encoding transformation through utilizing tensor networks.

The encoder works by mapping frames of a video sequence into an L dimensional latent-space. We compare three such encoders: two neural network-based encoders determined from an AE and VAE, and a low-rank encoder based on the TT-decomposition. The dimensionality of the latent space is typically determined by the formulation of the encoding transformation. For example, AEs and VAEs would treat this as a hyperparameter over which to optimize, whereas the TT-decomposition adaptively determines this this dimension to a user-specified error tolerance. We further note that each of the three methods outlined above also have an ability to reconstruct an image from the latent space, which potentially can be useful for investigating the latent space.

Although there are other neural-network approaches for learning a latent reprsentation, these mostly change the loss/auxiliary objective that is used to train the network. For instance, whereas AEs/VAEs are trained to minimize reconstruction error, contrastive methods instead learn to minimize a distance between related samples (such as two distorted views of the same image [52]) and maximize a distance between unrelated samples. Because the main difference is the training objective, the training time for AEs/VAEs should be similar to contrastive methods, assuming similar neural network architectures. We benchmark our results of using the TT against AEs/VAEs due to the similarity in their training objectives, which is to minimize reconstruction error.

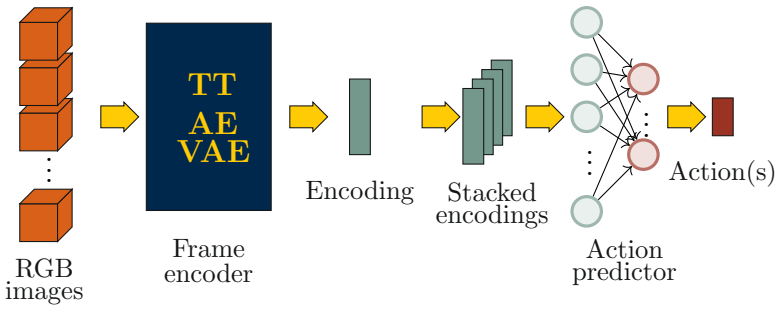


Figure 5.9: Imitation learning framework. Our focus is to enhance latent space learning depicted by the blue box. We compare the TT, autoencoder (AE) and variational autoencoder (VAE) approaches. Orange cubes represent images from gameplay videos. The extracted latent features are then used in the action-prediction network to predict actions. The architecture of the action predictor remains the same for all three encoding methods.

In the case of RGB images, a video sequence is a 4-dimensional tensor $\mathcal{X}^k \in \mathbb{S}^{H\times W\times C\times N^k}$

with $\mathbb{S} = [0, 255]$. The first three dimensions correspond to the height (H), width (W), and color (C) channels; and the last dimension corresponds to the index of an individual image within the video sequence.

In the video application considered in this section, we first reshape $\mathcal{X}$ into a 6-way tensor of shape $n_1 \times \ldots \times n_5 \times N^k$, where $\prod_{i=1}^{5} n_i = HWC$. By reshaping $\mathcal{X}$ to have smaller dimensions, we aim to limit the maximum TT-rank for each dimension. This also results in a slower growth in overall TT-ranks and yields higher compression. However, increasing the number of dimensions may result in slower computational speed because the TT-format is computed sequentially. In other words, increasing the number of dimensions will increase the number of TT-cores and therefore increase the number of SVDs needed to compute the TT-decomposition.

After reshaping, the tensor of a video sequence is represented in TT-format as

$$\mathcal{X} \approx \mathcal{G}_1 \ {}_3\times_1 \ \mathcal{G}_2 \ {}_3\times_1 \ \cdots \ {}_3\times_1 \ \mathcal{G}_6, \tag{5.5}$$

where $\mathcal{A} \ {}_3\times_1 \ \mathcal{B}$ denotes the tensor contraction between the third dimension of $\mathcal{A}$ and first dimension of $\mathcal{B}$, and $\mathcal{G}_i \in \mathbb{R}^{r_{i-1} \times n_i \times r_i}$. This representation often leads to a reduction in the size of the data. Recall that the representation shown in Equation (5.5) reduces the storage complexity from $\mathcal{O}(n^d)$ to $\mathcal{O}(ndr^2)$, where $r = \max i \in \{0, \ldots, d\} r_i$.

Note that in Equation (5.5), the first 5 TT-cores, $\mathcal{G}_1, \ldots, \mathcal{G}_5$, correspond to reshapings of the H, W, and C of the images – these are common to all images. The final dimension indexes the unique frame and therefore only the final core is unique to a data point. To be more specific, for the i-th image stored in $\mathcal{X}$, we have the reconstruction

$$\mathcal{X}\,(i) \approx \mathcal{G}_1 \ {}_3\times_1 \ \cdots \ {}_3\times_1 \ \mathcal{G}_5 \ {}_3\times_1 \ \mathcal{G}_6(:, i), \tag{5.6}$$

where $\mathcal{G}_6(:, i)$ denotes the i-th column of $\mathcal{G}_6$. As a result of this, we can use the representations stored in $\mathcal{G}_6$ as encodings of the frames and the fifth TT-rank, r_5, becomes the size of the latent space L.

However, performing the `TT-SVD` on large tensors requires loading the full tensor into memory and performing a sequence of singular value decompositions. As a result, this approach requires a significant amount of memory. The memory of our existing computational resources limits us to a maximum number of around 30,000 frames for `TT-SVD`. We use TT-ICE* [7] to compute the TT-decomposition of larger tensors, which allows us to compute the TT-decomposition incrementally, without requiring the full tensor to be loaded into memory. This is a critical enabling innovation for this application, as it allows us to compute the TT-decomposition of large video sequences without running into memory issues.

To train the tensor network, we create batches of $B = 2500$ frames from gameplay sequences and update the basis vectors stored in TT-cores using TT-ICE*. Although TT-ICE* does not require a fixed batch size as shown in [7], we train the tensor network using a fixed batch size to mimic the training of the NN based methods. Once a pass is completed over a batch of images, their latent representations are stacked to $\mathcal{G}_6$. Note that while `TT-SVD` computes the TT decomposition of a $n_1 \times \ldots \times n_d \times N$ tensor in time $\mathcal{O}(dn^{d+1}N)$ [114], TT-ICE* computes a tensor train using batches over the final dimension. Letting B be the batch size and b be the total number of updates, TT-ICE* calculates the TT decomposition in time $O(bdn^{d+1}B)$. If all batches are used to update the TT, this leads to the same computational complexity as TT-SVD (with $bB = N$). However, since TT-ICE* learns a generalizable space, not all B frames from a batch are needed to update the TT-cores. TT-ICE* updates the TT-cores by selecting the subset of frames that have an approximation error higher than the threshold. Furthermore, in the edge case where the TT-cores are highly representative of a batch, TT-ICE* skips updating TT-cores and directly computes the latent space embeddings using Algorithm 5. Through these modifications TT-ICE* achieves $bB \leq N$. Please refer to Table A.1. in [7] for more details. Even for the worst case of $bB = N$, TT-ICE* offers a substantial reduction in memory needed to compute the TT-decomposition in addition to eliminating the necessity of storing all frames at once.

Also note that TT-ICE* explores and increases the number of the basis vectors for each dimension of the tensor stream adaptively based on the user-specified error tolerance. This removes the need to manually tune the size of the latent space L for the encoded images.

In case a problem has some special properties to allow an *a-priori* upper bound on TT-rank, the rank and complexity must be adapted in response to received data. The datasets used in this work do not readily come with useful upper-bounds on the TT-ranks. As a result, we set a relative error tolerance of 1% for each training batch and the TT-ranks are automatically determined by TT-ICE* (please refer to Theorem 1 and Theorem 2 in Section 3.1.2.2 for proofs of approximation error)[2]. In the asymptotic limit of the data, TT-ICE* will be able to represent any tensor because of its rank adaptation property.

Next, we discuss how out-of-sample gameplay frames are mapped to the latent space by each encoding method.

5.2.3 Mapping Unseen Data to Latent Space

For AE/VAE, the latent space representation of unseen data is acquired directly through a forward pass of the trained encoder with the desired image. On the other hand for TT, the

[2]When the *occupancy* heuristic is not used, the approximation error guarantees proven for TT-ICE apply to TT-ICE* as well.

notion of forward pass is replaced with sequential projection onto cores.

Note that the first 5 TT-cores contain reshaped orthonormal vectors when TT-cores are trained with TT-ICE*. We obtain r_i orthonormal vectors for each dimension by reshaping the first 5 TT-cores as

$$U_i = \texttt{reshape}\left(\mathcal{G}_i, [r_{i-1}n_i, r_i]\right). \tag{5.7}$$

These orthonormal vectors are then used to encode any new images from the same game type. We obtain the latent embedding of images by simply projecting the appropriately reshaped images onto U_i. The pseudocode of this procedure is provided in Algorithm 5[3]. Note that as the ranks r of the TT increase, the computational time of Algorithm 5 grows quadratically with r. In particular, the number of flops for the projection grows at a rate of $O(drn^{d-1}N\max(r,n))$, where N is the number of new images to project, $n = \max_{i\in\{1,\dots,d\}} n_i$, and d is the dimensionality of the tensor. For this specific application, we set $n = 16$ and $d = 5$.

The AE, VAE, and TT all allow for reconstructing frames from the latent space. As such, we can visualize the reconstructed frames to get a sense of what is lost in encoding the frames. Figure 5.10 shows an example of reconstructed frames from the latent space for each encoding method.

Algorithm 5 Mapping unseen data to latent space by orthogonal projections onto TT-cores

Input
  $\{\mathcal{G}_i\}_{i=1}^5$ — TT-cores trained with TT-ICE*
  $\mathcal{Y}_j \in \mathbb{R}^{n_1\times\cdots\times n_5\times N_j}$ — j-th batch of frames that contain N_j frames to be projected
Output
  $g_j \in \mathbb{R}^{r_5\times N_j}$ — encodings of the j-th batch of frames
$g_j \leftarrow \texttt{reshape}\left(\mathcal{Y}_j, [n_1, n_2 \dots n_5 N_j]\right)$
for $i = 1$ to 4 **do**
  $U \leftarrow \texttt{reshape}\left(\mathcal{G}_i, [r_{i-1}n_i, r_i]\right)$ ▷ Note that for $i = 1$ we have $r_0 = 1$
  $g_j \leftarrow U^T g_j$
  $g_j \leftarrow \texttt{reshape}\left(g_j, [r_i n_{i+1}, n_{i+2} \dots N_j]\right)$
end for
$U \leftarrow \texttt{reshape}\left(\mathcal{G}_5, [r_4 n_5, r_5]\right)$
$g_j \leftarrow U^T g_j$

5.2.4 Action Predictor

To extract the game-specific playing strategy, we employ a simple **M**ulti-**L**ayer **P**erceptron (MLP) based architecture. We stack the latent representation of the last 4 frames together

[3]This algorithm is adapted from Algorithm 2

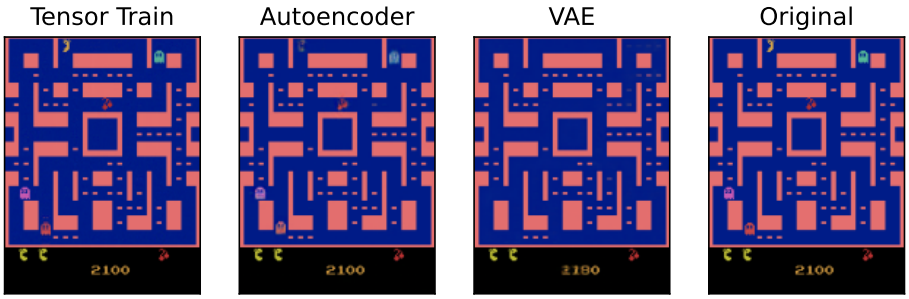


Figure 5.10: Example reconstruction of a Ms. Pac-Man frame using TT (with $\varepsilon = 0.01$), AE and VAE. TT reconstruction is visually the most similar to the original frame. The size of the latent space for all three models is 11583, which is 11.5% of the original size ($210 \times 160 \times 3$).

and obtain a $4L$ vector as input to the action predictor. Then the action predictor outputs the predicted action for a given timestep. Using the previous four frames as input allows the model to use some amount of temporal information and is a common choice for experiments conducted on ATARI games (see, e.g., [182]).

The action predictor consists of 5 hidden layers with 50 neurons each. Every hidden layer is then followed by a batch norm and ReLU operation. Finally, the output layer has size equal to the number of possible actions for each game, which is interpreted as the probability of taking each action. Next, we present the computational experiments we have conducted.

5.2.5 Experiments

In this section, we describe our comparative study, the dataset we selected for the experiments, evaluation metrics we used, and results we obtained.

5.2.5.1 Dataset

The dataset consists of game frames from demonstrations from a set of ATARI games, in particular: BeamRider, Breakout, MsPacman, Pong, Qbert, Seaquest, and SpaceInvaders[4]. Our data is collected using trained Deep Q-Networks [182], which are obtained from the RL-Baselines Zoo package [208] and further trained using the Stable Baselines platform [99] for 10^7 samples (states and actions), using the tuned hyperparameters from RL-Baselines Zoo. The chosen ATARI games were selected due to the availability of trained models and tuned hyperparameters in the RL-Baselines Zoo package. The reinforcement learning algorithm

[4]We note that we have also tested on Enduro, but due to computational limitations for the AE/VAE, results are not presented. In the moderate data case, the AE / VAE took roughly 30 hours to train per seed compared to 4.9 hours for TT-GPU.

used to generate the model (Deep Q-Networks) was chosen both due to the availability of trained models and tuned hyperparameters as well as the fact that Deep Q-Networks have been shown to perform well on ATARI games [182].

Rather than using downsized grayscale $84 \times 84 \times 1$ images (before stacking) as is often done in the literature, we instead use the original full-sized RGB images ($210 \times 160 \times 3$), since our work deals directly with image compression and dimensionality reduction. For each game, we generate 200 trajectories, a subset of which is used as a training set and a subset of which is used as a validation set. More details about the split are given in Section 5.2.5.2. Another 50 trajectories are generated as a test set, which is used to evaluate the action prediction accuracy of the final trained models. We define a trajectory as a sequence of states (frames), rewards, and actions taken by the RL agent, starting at the beginning of the game and ending when the RL agent loses a life in the game or the end of the game is reached. The dataset can be found at [48].

5.2.5.2 Setup

The procedure for our experiments is as follows:

1. Train an AE, VAE, and TT on the frames from a set of gameplay videos. The dimensionality of the latent space L for the TT is determined automatically by TT-ICE*. After the TT is trained, an AE and VAE are trained with a latent space size fixed to be equal to the size L of the latent space from the TT. This is done to provide a fair comparison of the methods. The latent size L for the various games are shown in Table 5.4. Furthermore, the VAE and AE are trained using a range of L2 regularization coefficients and initial learning rates.

2. For the AE and VAE, an additional portion of the gameplay videos are held out for validation. The validation reconstruction errors are used to select a VAE and AE for encoding. Note that for TT-decomposition no such validation procedure is needed.

3. Using the encodings generated by the TT or the AE / VAE as input, we train simple feed-forward networks to predict the actions taken by the trained reinforcement learning agent from which data is generated, using a range of hyperparameters.

4. For each of the AE, VAE, and TT encodings, we select the predictor that yields the lowest validation action prediction loss, as measured by cross-entropy loss on a separate validation set.

In this work, we consider training the AE, VAE, and TT separately to first learn an encoding, before training a predictor on the encodings directly. This is in contrast to end-to-end approaches, where the encodings would be trained jointly with the predictors. An advantage of training the encoders separately from the action predictors is reduced training time, as the expensive encoders are not further finetuned for the particular task of behavioral cloning. A second advantage is that the encodings are smaller than the full frames, leading to lower memory and storage requirements. This approach mimics the approach taken in several reinforcement learning papers, which have found that decoupling training an encoder from training the reinforcement learning agent is an effective approach - examples include [197, 198, 242, 295, 225].

The predictors and AE / VAE were set up and trained using PyTorch and trained on NVIDIA Tesla V100 GPUs. The TT is trained either using GPU (TT-GPU, NVIDIA Tesla V100) or CPU (TT-CPU, Intel Xeon Gold 6154 processors), using the TT-ICE* package - timings are reported for both. Note that the only difference between TT-GPU and TT-CPU is the device that the computation took place. This is achieved by replacing NumPy functions [94] used for computations with their CuPy counterparts [186] for TT-GPU. In parallel to the findings in Section 3.1, we use $n_i = \mathcal{O}(10)$ for $i = 1, \ldots, 5$ and therefore reshaped the image batches to $15 \times 14 \times 16 \times 10 \times 3 \times N$, where N is the fixed batch size used for the TT (in our experiments, $N = 2500$).

In this work, we consider two cases: 1) a limited data regime where only 5 gameplay trajectories are used as training data for each game and 2) a moderate data regime where 160 gameplay trajectories are used as training data for each game. A summary of the number of trajectories used in the limited and moderate data settings is shown in Table 5.5. In addition to these two scenarios, we also provide a more detailed investigation on the effects of changing the latent space size (Appendix E.2.2) and the amount of training data (Section 5.2.6.3) on action prediction accuracy and computational speedup.

Action predictors are trained with the Adam-W optimizer [165] with a cosine learning rate decay [164] for fifty epochs in the low data case and five epochs in the moderate data case. These predictors are trained using the same range of initial learning rates and weight decay penalties as the ones used to train the AE and VAE encoders (shown in Appendix E.1). An additional 1 or 20 trajectories (in the limited and moderate data regimes, respectively) not used during training are then used as a validation set to evaluate the trained predictors, with the predictors yielding the lowest cross-entropy action prediction loss selected.

We also include the results of a predictor trained using the original, full-sized RGB frames, of dimension $210 \times 160 \times 3$. Because the "no encoding" predictor uses the full-sized images, it is expected that these accuracies will be higher than the predictors using the encodings -

Table 5.4: Latent sizes L for the AE, VAE, and TT for the games tested. Note that the latent size is determined automatically for the TT given the prescribed relative error bound of 0.01. That latent size is then used for the AE and VAE.

	Latent Size L
BeamRider	14645
Breakout	3561
MsPacman	11962
Pong	1287
Qbert	2267
Seaquest	15163
SpaceInvaders	17191

Table 5.5: Summary of the number of trajectories used to train the models in the limited and moderate data cases. Note that each row corresponds to separate trajectories.

	Limited Data	Moderate Data
Training Set for Training Encoder and Predictor	5	160
Validation Set for Selecting Best Encoder (ONLY AE/VAE)	1	20
Validation Set for Selecting Best Predictor	1	20
Test Set for Evaluating Final Models	50	50

these are included as a comparison as an upper bound on accuracy we can expect.

Additional details on training and the neural network architectures can be found in Appendix E.1.

5.2.5.3 Metrics

We compare performance of the methods using four metrics: training time, reconstruction quality, action prediction accuracy, and normalized gameplay score. We then report the averaged results for each of those metrics for each regime in Section 5.2.6.

Training Time. We compare the time required to train each framework against the time needed to train a predictor on full frames. The training time for each framework can further be broken down into the time it takes to train the encoder, the time it takes to encode the data, and the time it takes to train the action predictor given the encodings. These results are shown in Appendix E.2.1 as Figures E.1 and E.2.

Reconstruction Quality. We measure the reconstruction quality of the TT against an AE and VAE using peak-signal-to-noise ratio (PSNR) and mean-squared error (MSE) as metrics.

Action Prediction Accuracy. We compare the accuracy of the actions suggested by each framework to actions taken by the RL agent. The output of the behavioral cloning agents is a vector of probabilities for each action - we calculate the proportion of times the action with the highest probability is equal to the action with the highest probability outputted by the RL agent.

Normalized Gameplay Score. We normalize gameplay scores achieved by each framework so that a score of 0 corresponds to performance of an agent taking random actions and a score of 1 corresponds to performance of the RL agent. In particular, given the score s achieved by a behavioral cloned agent, the mean score s_{RL} achieved by the RL agent and the mean score s_{random} achieved by an agent taking random actions, we calculate the normalized score to be

$$s_{\text{normalized}} = \frac{s - s_{\text{random}}}{s_{\text{RL}} - s_{\text{random}}}. \tag{5.8}$$

5.2.6 Results

5.2.6.1 Limited Data Regime

The benefits of bypassing the need for training a neural network encoder can be observed in the total training time of the different methods, as shown in Figure 5.11. In particular, the training time of the TT-based predictor trained on GPU (TT-GPU) is 8.2× to 9.6× faster compared to the AE and VAE-based predictors and 26.1× to 29.7× faster compared to the predictor trained with no encoding. When the TT encoding is trained on CPU (but with the predictor still trained on GPU), the TT-predictor is still 3.3× to 5.4× faster than the AE and VAE predictors trained on GPU and 10.4× to 17.5× faster than the predictor trained with no encoding.

This speedup comes with an improvement in reconstruction quality, as shown in Table 5.6; the TT leads to a lower MSE and higher PSNR compared to the AE and VAE in six of the seven games. This speedup also comes at no cost in terms of behavioral cloning accuracy (Figure 5.12a) and achieved gameplay scores (Figure 5.13a), compared to training using encodings generated by an AE or VAE.

5.2.6.2 Moderate Data Regime

In the case where 160 trajectories are used for training data, we find once again that the TT-based predictor provides a large speedup relative to the AE, VAE, and full frame predictors. In particular, as shown in Figure 5.11, using the TT-encoder trained on GPU led to a 3.4× to 9.8× faster training time compared to the no-encoding predictor. Compared to the AE

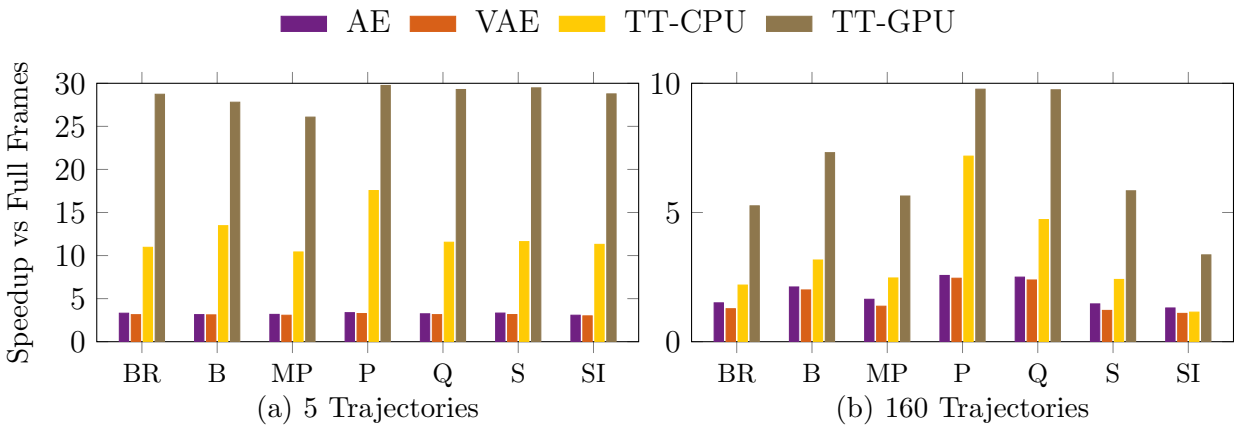


Figure 5.11: Speedup of the end-to-end training procedure of the AE, VAE, and TT-based predictors compared to a predictor trained using the full RGB frames. Extracting a latent space reduces end-to-end training time in all cases. In the low data regime, using the TT-GPU leads to a 26.1× to 29.7× faster training time than using the full RGB frames and a 8.2× to 9.6× speedup compared to the AE and VAE. In the moderate data case (160 trajectories), using TT-GPU leads to a 3.4× to 9.8× faster training time compared to using the full RGB frames and a 2.6× to 4.8× speedup compared to the AE and VAE. Note that the difference in speedups between the 5 trajectory case and 160 trajectory case is due to the difference in training epochs used in each case (50 in the 5 trajectory case and 5 in the 160 trajectory case); the speedups are similar when the number of training epochs is held fixed while the number of training trajectories is changed, as shown in Section 5.2.6.3. BR:Beam Rider, B:Breakout, MP:Ms.Pac-Man, P:Pong, Q:Q*bert, S:Seaquest, SI:Space Invaders.

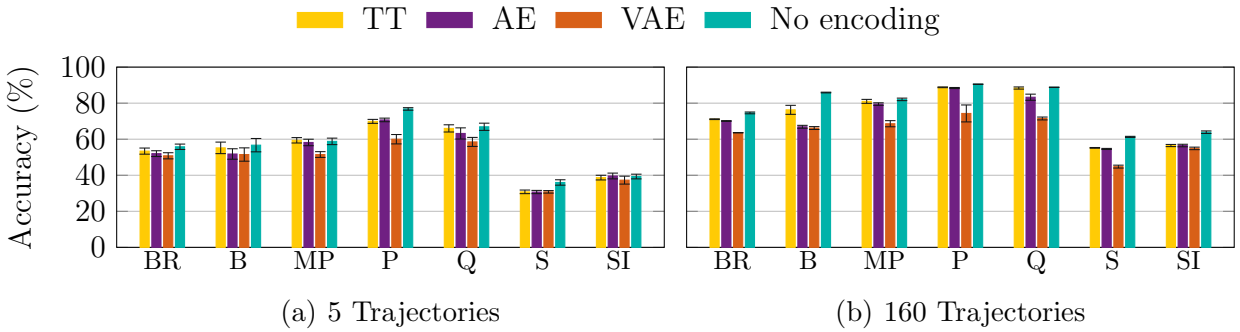
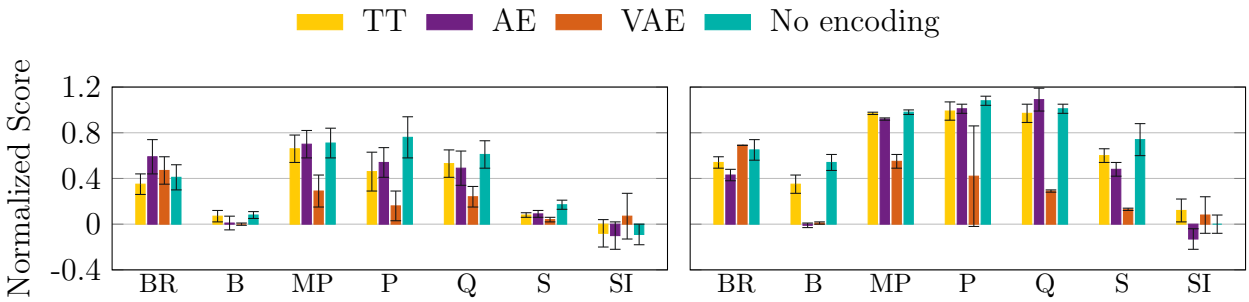


Figure 5.12: Action prediction accuracies of trained predictors using encodings from the TT, AE, and VAE, as well as a trained predictor using the full-sized original RGB frames. Training a predictor using TT encodings yields similar and sometimes better prediction accuracies as training a predictor using AE or VAE encodings. Error bounds are calculated as two times the standard error of gameplay scores calculated across ten (in the 5 trajectory case) or three (in the 160 trajectory case) experiments. Please refer to the caption of Figure 5.11 for the game abbreviations.

and VAE, the end-to-end training time of the TT-based predictor is 2.6× to 4.8× faster. Training the TT on CPU still usually leads to a speedup over the AE and VAE on GPU (0.9×

Table 5.6: Reconstruction quality of the TT, AE, and VAE in the low data regime. The reported statistics are MSE (using 255 as maximum pixel value) and PSNR of the reconstructions versus the original frames. For each game, the bolded values are the smallest MSE and highest PSNR.

	MSE			PSNR		
	TT	AE	VAE	TT	AE	VAE
BeamRider (BR)	**6.3**	30.5	47.6	**46.8**	35.3	33.0
Breakout (B)	**33.0**	67.6	80.9	**39.4**	32.9	30.3
MsPacman (MP)	**27.1**	58.0	103.3	**38.7**	32.5	28.9
Pong (P)	2.0	**0.7**	56.9	46.7	**61.6**	36.9
Qbert (Q)	**6.8**	33.5	68.5	**45.6**	34.3	30.8
Seaquest (S)	**13.9**	30.8	74.0	**38.6**	33.9	29.8
SpaceInvaders (SI)	**80.3**	144.5	176.4	**32.4**	28.4	26.2



(a) 5 Trajectories (b) 160 Trajectories

Figure 5.13: Normalized gameplay scores achieved by each of the behavioral cloning models, trained using either encodings from the TT, AE, VAE, or the original RGB images. A score of 0 corresponds to performance of an agent taking random actions and a score of 1 corresponds to performance of the RL agent. Training a predictor using TT encodings yields similar gameplay scores as training a predictor using AE or VAE encodings. Error bounds are calculated as two times the standard error of gameplay scores calculated across ten (in the 5 trajectory case) or three (in the 160 trajectory case) experiments.

Table 5.7: Reconstruction quality of the TT, AE, and VAE in the moderate data regime. The reported statistics are the MSE (using 255 as maximum pixel value) and PSNR of the reconstructions versus the original frames. For each game, the bolded values are the smallest MSE and highest PSNR.

	MSE			PSNR		
	TT	AE	VAE	TT	AE	VAE
BeamRider (BR)	**0.4**	18.0	26.0	**58.1**	37.6	36.4
Breakout (B)	**1.2**	7.8	19.8	**51.0**	40.1	36.8
MsPacman (MP)	**1.7**	16.2	75.7	**48.1**	37.6	30.2
Pong (P)	1.8	**0.0**	6.9	47.0	**71.4**	40.7
Qbert (Q)	**0.8**	12.1	36.4	**52.7**	39.0	33.8
Seaquest (S)	**1.0**	8.4	52.9	**49.5**	39.9	31.4
SpaceInvaders (SI)	**0.4**	18.5	80.2	**56.1**	36.5	30.1

to 2.9× faster), showing the computational efficiency of TT-ICE even on CPU compared to GPU-based deep learning.

These results come with an improvement in reconstruction quality and no cost in terms of behavioral cloning accuracy or gameplay scores. The reconstructions generated by the TT have lower error compared to the reconstructions generated by the AE and VAE, as shown in Table 5.7. Furthermore, we find improved behavioral cloning accuracy of the predictor trained using the TT encodings versus the predictors trained on the AE or VAE encodings for some of the games, as seen in Figure 5.12b. The gameplay scores achieved by each of the encoding methods is comparable, as shown in Figure 5.13b.

Note that in some games, the gameplay scores achieved by the predictors, including the one trained with no encoding, are much lower than the scores achieved by the RL agent. This is likely due to an issue encountered in imitation learning where the distribution of data used to train the agent differs from the distribution of data encountered while testing the agent, since the agent's actions directly affect the future states it encounters. This leads to compounding errors and hurts the performance of the imitation learning agent [216]. These low scores are also consistent with [128], which found that behavioral cloning from human gameplay was unable to match human-level gameplay scores for the ATARI games considered in that paper.

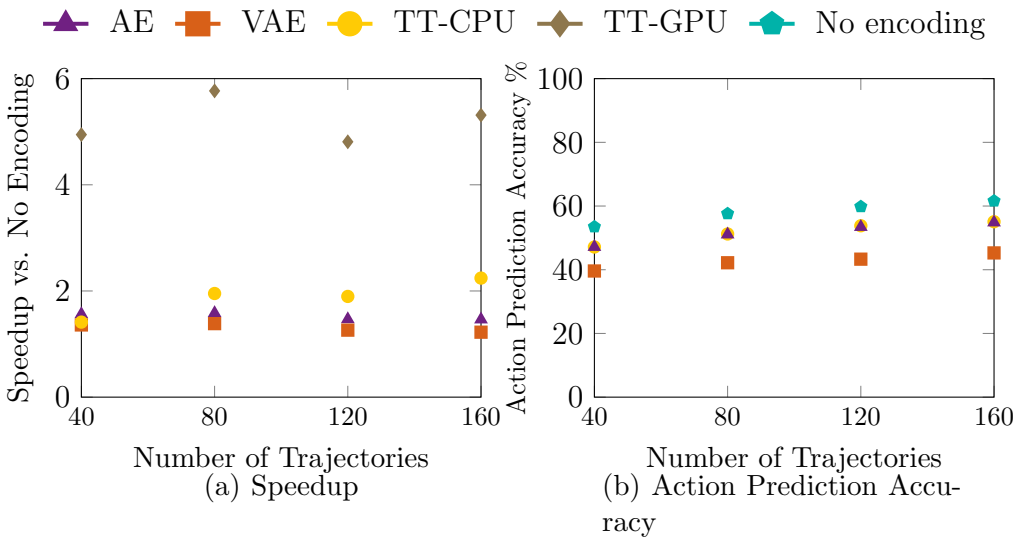
5.2.6.3 Scalability of the Tensor Train

Differences in the speedup between the limited data regime and moderate data regime are due primarily to the number of training epochs used in training the AE/VAE - we use 50 training trajectories in the limited data regime and 5 in the moderate data regime. The

reason for this difference is that we found that more training epochs were needed in the limited data regime to achieve similar accuracies via the AE/VAE.

To make the speedups of the TT more comparable across different amounts of training data, we run a small study to examine the effect of increasing the number of gameplay trajectories used to train the behavioral cloning framework on both training time and prediction accuracy. The AE and VAE are trained using five epochs, matching the training setup used in the moderate data regime. We ran this for the game of Seaquest, given the relatively low prediction accuracies we achieved compared to some of the other ATARI games. As expected, Figure 5.14 shows that the accuracy increases as more training data is used, suggesting that better prediction accuracies can be achieved with more data.

We note that TT's speedup over the AE / VAE and no encoding remains similar as more data is used, as long as the number of training epochs is fixed. These results suggest that the timing advantage may remain for the TT even when more data is available for behavioral cloning.



(a) Speedup

(b) Action Prediction Accuracy

Figure 5.14: Results of the experiments investigating the effect of number of trajectories used to train the behavioral cloning framework on computational speedup and action prediction accuracy using different encoding methods for the game of Seaquest. As the number of training trajectories increases, action prediction accuracy also increases. Computational benefits of TT-ICE* over the AE, VAE, or no encoding remain similar across different number of trajectories. Action prediction accuracies are only presented for TT-CPU since TT-CPU and TT-GPU result in the same latent space.

5.2.6.4 Prediction Accuracy Versus Training Time

A significant difference between training the AE / VAE versus the TT is the number of passes through each data point. Since the amount of training data is limited, multiple passes through the data are used for the AE/VAE to increase the number of update steps. On the other hand, the TT is trained using an algorithm that requires only a single pass through the data. As a result, the AE/VAE process more data points during training than the TT, albeit with repeating data.

To address this difference, we run experiments where each algorithm is trained with one epoch and as such each algorithm processes the same number of data points. The resulting accuracies and timings are plotted in Figure 5.15. As can be seen, TT-GPU tends to achieve higher accuracies and lower end-to-end training time than the AE/VAE.

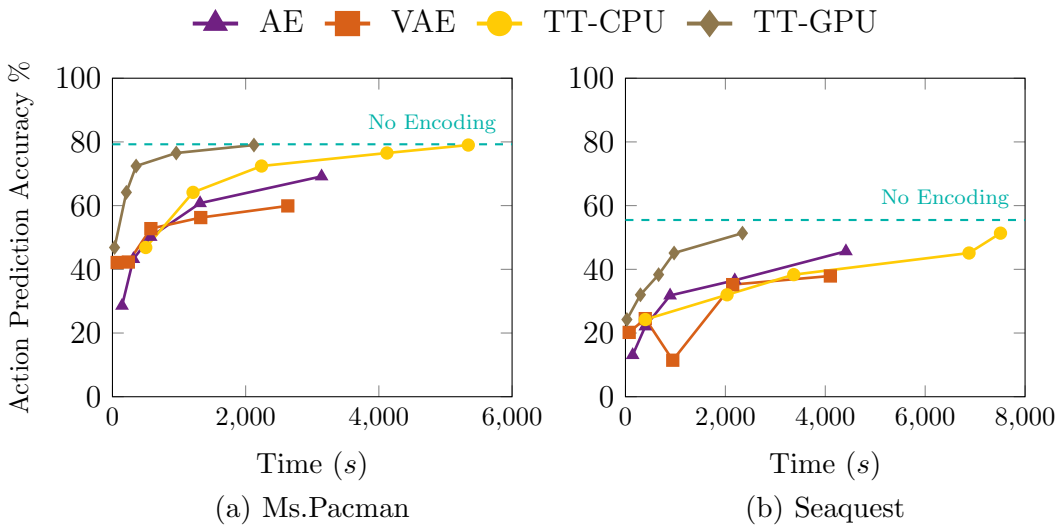


Figure 5.15: Comparison of behavioral cloning accuracy versus end-to-end training time for the AE, VAE, and TT, when keeping the number of data points seen by each algorithm the same by using only a single pass through the data. Points in the upper left indicate lower training time and higher accuracy. Note that TT-GPU tends to perform better in terms of lower training time and higher accuracy than the AE and VAE.

5.2.6.5 Encoding Times and Inference Speed

During inference, new frames are passed through the TT, AE, or VAE, before the resulting latent vector is passed into a predictor. Because the architecture of the predictor is held fixed, differences in inference speed will stem from differences in the time it takes to encode frames using the various methods. These encoding times are compared in Table 5.8.

In the 5 trajectory case, the encoding time for TT-GPU is between 0.4× and 2.8× that of the AE and VAE. In the 160 trajectory case, the encoding time for TT-GPU is between 1.5× and 8.8× that of the AE and VAE. The current Python implementation of TT-ICE* is not optimized for an inference task like the one we have in this work. As shown in Algorithm 5, encoding new data using the TT-encoder is a task that is sequential in the dimensions. More efficient parallel approaches may be envisioned that can further accelerate the encoding task. There are multiple works in the literature on scaling such operations [63, 235], and perhaps these can be considered in the future. However, even with this inefficiency, our results indicate faster training.

To determine why encoding is slower for the TT, we can investigate both flop counts and actual compute time. Moreover, we can see how these vary with dataset size. Flop counts (e.g., the number of multiply-add operations) for the AE and VAE are calculated using the sovrasov2018ptflops package [238]. Flop counts for the TT are calculated by counting the number of flops per matrix multiplication in Algorithm 5 (lines 9 and 13).

In Table 5.9, the ratio in encoding times and flop counts between the TT and AE/VAE are shown. Although the flop counts in the TT are generally higher, this does not lead to a proportional increase in encoding time (e.g., TT-GPU uses 5.8× as many flops as the AE in the 5 trajectory case for BeamRider, but the encoding time for TT-GPU is 2.0× as long as the encoding time for the AE). This is partly due to the number of sequential calculations needed for each encoding method; in our experiments, the AE/VAE requires thirteen sequential layers (intermediate layers, including batch norm layers and activation functions), while TT projection requires only five tensor contractions (the number of tensor cores minus one).

We also explore the scalability of the inference speed of the TT with the amount of data. During training, the tensor ranks r of the TT increase as needed to represent new data. The number of flop counts for encoding using the TT grows quadratically with r. As such, the scalability of the inference speed of the TT depends on the ability of the TT to learn a generalizable latent structure that can accurately represent new data, as this will lead to fewer and smaller increases in r as new data is introduced.

It is difficult to determine this scalability *a-priori*, but we empirically investigate the growth in the tensor ranks and flop counts in Figure 5.16. As the amount of data increases, the maximum rank of the TT grows, but the growth rate tends to level off. The slow growth of the TT ranks suggest that for these ATARI games, the TT is able to learn a generalizable latent structure and scales well with additional data.

An area of future research will be to improve the encoding time of the TT. However, we reinforce that though the encoding time of the TT is longer than that of the AE and VAE,

Table 5.8: Median time to encode the training and validation data in seconds using the various encodings. In the 5 trajectory case, the encoding time for TT-GPU is between 0.4× and 2.8× that of the AE and VAE. In the 160 trajectory case, the encoding time for TT-GPU is between 1.5× and 8.8× that of the AE and VAE.

	5 Trajectories				160 Trajectories			
	AE	VAE	TT-CPU	TT-GPU	AE	VAE	TT-CPU	TT-GPU
BeamRider	3.6	3.6	160.9	7.2	156.9	139.9	1906.5	673.5
Breakout	2.2	0.9	29.7	0.9	27.9	26.1	165.1	54.8
MsPacman	2.9	2.9	111.8	7.9	89.4	90.9	1153.2	484.4
Pong	12.8	12.4	519.2	25.4	433.4	448.1	1994.3	680.5
Qbert	2.0	2.0	64.1	2.9	60.3	60.6	388.5	123.6
Seaquest	3.2	3.1	116.3	7.9	128.3	132.9	1714.4	619.4
SpaceInvaders	2.1	1.9	73.8	3.1	71.7	72.3	1387.0	632.6

the overall time for training the TT on GPU is still much lower than that of the AE and VAE since training the encoding is the most time-intensive task.

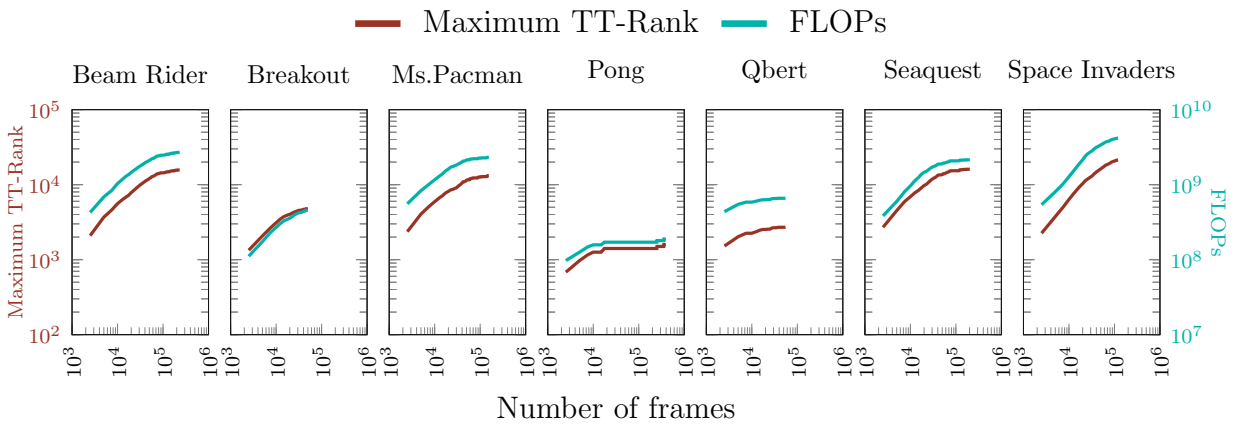


Figure 5.16: Maximum tensor rank (red) and flop count (teal) for the TT during training. As training progresses, the maximum tensor rank and the flop count increase, but this growth rate decreases. The slow growth in the TT ranks suggests that the TT-ICE* algorithm is able to learn a generalizable latent structure; as the amount of data seen by the TT-ICE* algorithm increases, smaller updates to the TT ranks are needed to represent new data. In cases like this where the tensor ranks of the TT grow slowly, the encoding time for the TT will scale well since the flop count grows quadratically with the TT ranks r.

5.2.6.6 Additional Results

Appendix E.2.1 provides a more detailed investigation into the timing differences between the methods. The speedups achieved by the TT compared to the AE and VAE are split into three components:

Table 5.9: Ratio of encoding times and flop counts between the TT and AE/VAE. Although the flop counts are much higher for the TT than the AE/VAE, this does not lead to a proportionate increase in encoding time.

		5 Trajectories		160 Trajectories	
		TT-GPU / AE	TT-GPU / VAE	TT-GPU / AE	TT-GPU / VAE
Encoding Time Ratio	BeamRider	2.0	2.0	4.3	4.8
	Breakout	0.4	1.0	2.0	2.1
	MsPacman	2.8	2.8	5.4	5.3
	Pong	2.0	2.0	1.6	1.5
	Qbert	1.5	1.5	2.0	2.0
	Seaquest	2.5	2.5	4.8	4.7
	SpaceInvaders	1.4	1.6	8.8	8.7
Flop Count Ratio	BeamRider	5.8	4.8	12.8	7.8
	Breakout	0.8	0.8	3.9	3.0
	MsPacman	7.6	6.5	12.7	8.0
	Pong	1.9	1.7	2.3	2.0
	Qbert	5.3	4.9	7.5	6.2
	Seaquest	4.8	3.9	10.4	6.3
	SpaceInvaders	5.0	4.5	16.3	9.7

1. Time to train the encoder.

2. Time to encode training and validation data used to train the action predictor.

3. Time to train the action predictor.

To ensure that our results are consistent across different settings, we also examine how changing the latent space size (Appendix E.2.2) affects the training time and action prediction accuracy of the AE, VAE, and TT. The results are consistent: we find that as the latent space size and number of training trajectories are changed, TT achieves similar prediction accuracies as the AE and VAE but provides a speedup in training time.

In Appendix E.2.3, we provide the gameplay scores achieved by the RL agent used to generate the demonstration data we used for behavioral cloning. The RL agent achieves gameplay scores that are higher than a randomly acting agent and achieves human level performance in two of the seven games.

5.3 Generative Diffusion Models using Tensor-Network Embeddings

The final application we consider in this thesis is the use of tensor network embeddings for generative modeling of high-dimensional data, such as images and videos. In this section, we

demonstrate the effectiveness of tensor network embeddings in denoising diffusion models. Figure 5.17 illustrates the generative modeling pipeline using tensor network embeddings we considered in this section.

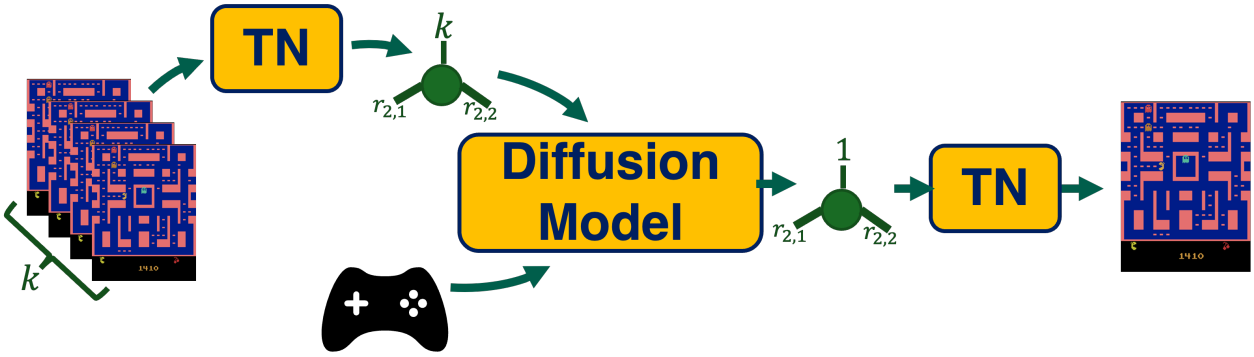


Figure 5.17: Generative modeling pipeline using tensor network embeddings. The pipeline consists of a tensor network, a denoising diffusion model with a UNet architecture, and action inputs. The tensor network compresses the high-dimensional game frames into a low-dimensional latent space, which is then used as input to the denoising diffusion model along with the user actions. We use the same tensor network to decompress the generated latent space back into high-dimensional game frames for visual inspection.

5.3.1 Diffusion Models and Problem Set-Up

Diffusion models [106, 236, 184] are a class of generative models that learn to generate new data points by modeling a reversal of the diffusion process, starting from pure noise to the original data distribution. They have gained significant attention in recent years due to their ability to generate high-quality images, videos, and other types of data [210, 214, 204]. The diffusion process consists of two main components: the forward process and the reverse process. The forward process gradually adds noise to the data, while the reverse process learns to denoise the data. Figure 5.18 illustrates both of these processes[5].

The forward process is defined as a Markov chain that gradually adds noise to the data, leading to a sequence of noisy data points. The forward process can be represented as a sequence of conditional distributions $q(x_t|x_{t-1})$, where x_t is the noisy data at time step t and x_{t-1} is the noisy data at the previous time step. The forward process starts with the original data x_0 and gradually adds noise until the data is indistinguishable from pure noise. The forward process can be represented as a sequence of Gaussian distributions, where the mean and variance of the distribution change over time. For an arbitrary time step t, the forward

[5]Note that this figure is adapted from [106]

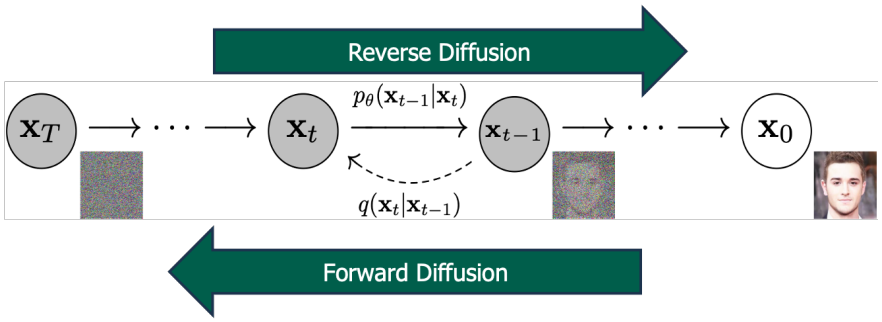


Figure 5.18: Denoising diffusion process. $q(x_t|x_{t-1})$ represents the forward process, which adds noise to the data at each time step t. $p_\theta(x_{t-1}|x_t)$ represents the reverse process, where θ are the parameters of the denoising model. In line with existing literature, we use a UNet-based neural network as p_θ. Figure adapted from [106].

process can be defined as:

$$q(x_t|x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}x_{t-1}, \beta_t \mathrm{I}), \tag{5.9}$$

where β_t is the variance level at time step t, and I is the identity matrix. The forward process can be iterated to generate a sequence of noisy data points, starting from the original data x_0 and adding noise at each time step until reaching pure noise at x_T, where T is the total number of time steps. We can also add noise to the original data x_0 directly to obtain the noised data at time step t as follows:

$$q(x_t|x_0) = \mathcal{N}(x_t; \sqrt{\alpha_t}x_0, (1-\alpha_t)\mathrm{I}), \tag{5.10}$$

where $\alpha_t = \prod_{s=1}^{t}(1-\beta_s)$ is the cumulative product of the noise levels up to time step t. This allows us to directly sample noisy data points from the original data distribution at any time step t.

The amount of noise added at each time step is controlled by a noise schedule, which defines how the noise level changes over time. Two common choices for the variance schedule $\{\beta_t\}_{t=1}^{T}$ are linear and cosine schedules. The linear schedule is defined as:

$$\beta_t = \beta_{\text{start}} + \frac{t-1}{T-1}(\beta_{\text{end}} - \beta_{\text{start}}), \tag{5.11}$$

where β_{start} and β_{end} are the starting and ending variance levels, respectively. Typical values are $\beta_{\text{start}} = 1 \times 10^{-4}$ and $\beta_{\text{end}} = 2 \times 10^{-2}$.

The cosine schedule, proposed by [184], provides a more gradual noise addition and is

defined as:

$$\alpha_t = \frac{f(t)}{f(0)}, \quad \text{where} \quad f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2, \tag{5.12}$$

with $s = 8 \times 10^{-3}$ being a small offset to prevent β_t from being too small near $t = 0$. The variance schedule is then computed as $\beta_t = 1 - \alpha_t/\alpha_{t-1}$.

The reverse process is the key component of diffusion models, as it learns to generate new data points by reversing the diffusion process. The reverse process is defined as a sequence of conditional distributions $p_\theta(x_{t-1}|x_t)$, where θ are the parameters of the denoising model. Specifically, our goal is to learn $p_\theta(x_{t-1}|x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \beta_t \mathrm{I})$. For $\epsilon \sim \mathcal{N}(0, \mathrm{I})$, the objective for μ_θ is to predict $\frac{1}{\sqrt{1-\beta_t}}(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\epsilon)$ for a given x_t. After algebraic reparametrization, sampling $x_{t-1} \sim p_\theta(x_{t-1}|x_t)$ boils down to computing the following:

$$x_{t-1} = \frac{1}{\sqrt{1-\beta_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\alpha_t}}\epsilon_\theta(x_t, t)\right) + \sqrt{\beta_t}\epsilon, \tag{5.13}$$

where $\epsilon_\theta(x_t, t)$ is the predicted noise at time step t given the noisy data x_t and the time step t. After the reparametrization in Equation (5.13), the training objective for the denoising model becomes minimizing the following expectation:

$$\mathbb{E}_{x_0,\epsilon}\left[\frac{\beta_t}{2(1-\beta)(1-\alpha)}\|\epsilon - \epsilon_\theta\left(\sqrt{\alpha_t}x_0 + \sqrt{1-\alpha_t}\epsilon, t\right)\|^2\right], \tag{5.14}$$

which is analogous to the denoising score matching objective [237]. This approach encourages the denoising model to predict the noise added to the data at each time step, allowing it to reconstruct the original data from the noisy data.

A typical choice of architecture for the denoising model is the UNet architecture [215], which has been shown to be effective for image generation tasks [106, 236]. The UNet architecture consists of a series of downsampling and upsampling layers, allowing it to capture both local and global features of the data. The downsampling layers reduce the spatial dimensions of the data, while the upsampling layers increase the spatial dimensions back to the original size. The skip connections between the downsampling and upsampling layers allow the model to retain important features from the original data, improving the quality of the generated data.

Instead of performing the denoising process in the high-dimensional data space, performing the denoising process in a lower-dimensional latent space has been shown to be effective in improving the quality of the generated data and reducing the computational complexity [262]. This approach allows for more efficient denoising and generation of new data points, as the low-dimensional latent space captures the essential features of the data while

reducing the computational complexity.

5.3.2 Dataset, Architecture, and Training

Using generative models to create game environments has been a topic of interest in the field of reinforcement learning and game AI [11, 263]. Use of recurrent neural networks for simulating the game state is also investigated in [93, 132]. In [110], the authors proposed the use of an action-conditioned generative model to learn a world model for driving environments. The use of transformer architectures for video generation has also been explored, such as in [286], where the authors proposed a decoder-only transformer architecture for action-conditioned video generation.

In this section we consider the ATARI games, which is a popular benchmark for reinforcement learning [182, 181, 156], behavioral cloning [50, 49] and generative modeling [11] tasks. Despite the simplistic nature of the graphical fidelity, ATARI game frames are still high-dimensional images that capture the game state at each time step. The goal is to generate new game frames that are visually similar to the original frames while capturing the dynamics of the game. We investigate using a low-dimensional latent space representation obtained through low-rank tensor decompositions to reduce the dimensionality of the game frames, allowing for efficient denoising and generation of new frames. Specifically, we use the HT-RISE algorithm from Section 3.3 to compress the ATARI game frames into a low-dimensional latent space representation. In the previous sections, we have demonstrated that HT-RISE can learn a generalizable low-dimensional representation that captures the essential features of the data, while allowing for efficient compression and accurate reconstruction.

Since the dataset we used in Section 5.2 [48] contains demonstrations from only expert agents, it explores only a small portion of the state space. To ensure that both the tensor networks as well as the trained diffusion models can learn the full dynamics of the game, we train 12 different RL agents using the Stable-Baselines3 library [209] on Ms.Pac-Man game with increments of 5 million frames, starting from 5 million frames up to 60 million frames, simulating 12 different experience levels. For each of these agents, we collect 50 gameplay episodes each for training, resulting in a total of 600 gameplay episodes. Additionally, we recorded 5 episodes per experience level as a separate test set, resulting in a total of 60 gameplay episodes for testing. Each episode consists of a sequence of game frames and actions taken by the agent at each time step. The actions are discrete and correspond to the available actions in the ATARI game environment, such as moving left, right, up, down, or performing no action. Each episode runs until the game is over, which results in a varying number of frames per episode.

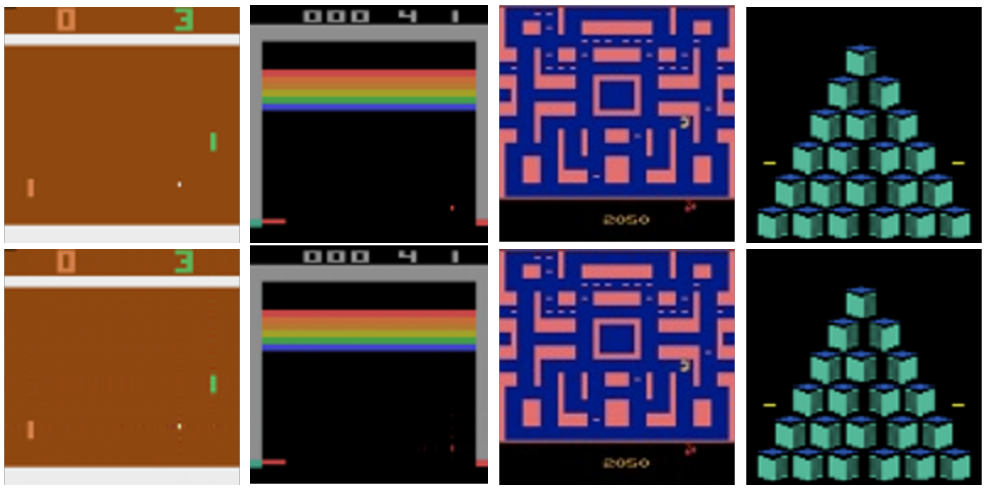
Recall that each Atari game frame has the shape $210 \times 160 \times 3$, where the first two dimensions correspond to the height and width of the frame, and the last dimension corresponds to the RGB color channels. In parallel to the work in [11], we resize the frames to $96 \times 96 \times 3$. This is slightly different from the size usend in [11], which is $64 \times 64 \times 3$, as we found that any resolution lower than the 96×96 leads to challenges for HT-RISE to find and exctract a low-rank representation with minimal visual artifacts.

We reshape each downsized $96 \times 96 \times 3$ frame into a 9D tensor of shape $2 \times 4 \times 4 \times 3 \times 4 \times 4 \times 6 \times 3 \times 1$. Since the original games are running at a frame rate of 20 frames per second, we use batches of 20 frames (resulting in tensors of shape $2 \times 4 \times 4 \times 3 \times 4 \times 4 \times 6 \times 3 \times 20$) to train the tensor network incrementally using HT-RISE. Our empirical experience shows that a healthy trade-off between the compression rate and the reconstruction quality is achieved around a compression ratio of $2\times$. Usually any compression ratio above $2\times$ leads to loss of crucial information for individual games (e.g., the ball in Pong, or the beads in Ms.Pac-Man). On the other side of the spectrum, picking a very restrictive relative error tolerance leads to a very low compression rate, which makes the use of tensor networks for compression obsolete. Therefore we pick the relative error tolerances for each game depending on the resulting compression rate, as shown in Table 5.10. The relative error tolerances are picked to be 1%, 2%, 3%, and 4% for Pong, Breakout, Ms.Pac-Man, and Q*bert, respectively.

Table 5.10: Compression rates achieved by HT-RISE for different ATARI games. The compression rates are reported in this table are elementwise compression rates, i.e., the ratio of the number of elements in the original tensor to the number of elements in the compressed tensor. The $\otimes$ symbol indicates that the relative error level is too low for a meaningful compression, while the $*$ symbol indicates that the relative error level is too high for maintaining the visual quality of the reconstructed frames. Please refer to Figure 5.19 for visualizations of the original and reconstructed frames.

Rel. Err.	**Pong**	**Breakout**	**Ms.Pac-Man**	**Q*bert**
1%	2.03×	$\otimes$	$\otimes$	$\otimes$
2%	5.70×	1.98×	1.97×	$\otimes$
3%	$*$	2.82×	2.80×	1.94×
4%	$*$	$*$	$*$	2.57×

As for the architecture, we use a UNet architecture [215] for the denoising diffusion model, which is a common choice for image generation tasks [106, 11, 236]. The architectural novelty is not the focus of this work, but rather the use of tensor network embeddings for generative modeling. Therefore, we adapted the architecture presented in [11] to work with the low-dimensional latent space representations obtained through HT-RISE. In parallel to that work, we used 3 downsampling layers and 3 upsampling layers. To ensure that the downsampling and upsampling layers are compatible with the low-dimensional latent space



(a) 1% (b) 2% (c) 3% (d) 4%

Figure 5.19: Reconstructions of downsized ATARI game frames using HT-RISE at different relative error levels. The first row shows the original downsized game frames, while the second row shows the reconstructed frames using HT-RISE at a relative error level denoted by the corresponding captions. Please refer to Table 5.10 for the compression rates achieved by HT-RISE for the displayed relative error levels. Displayed games (from left to right): Pong, Breakout, Ms.Pac-Man, Q*bert.

representations, we padded the input data to ensure that the dimensions are divisible by 16. The only major difference we employed is that we used 8 layers in each of the UNet blocks, instead of 4 layers as in [11]. This is done to ensure that the model has enough capacity to learn the denoising process in the low-dimensional latent space. Note that we did not explore any latent-tailored architectures, which may result in improved performance.

The diffusion model is trained using the score matching objective [116], which is a common choice for diffusion models [106, 236]. The training is performed using the AdamW optimizer [165] with a learning rate of 3×10^{-4} and a batch size of 8. The training is performed for 350 epochs. We also employ an exponential learning rate decay with a decay factor of 0.99. We split the dataset into training and validation sets, with 80% of the data used for training and 20% for validation.

To guide the diffusion process, we used the 4 previous frames and actions as additional inputs to the denoising model. This is done to provide a context for the trained model to generate new frames that are consistent with the previous frames and input actions. The action space consists of 9 discrete actions, which are represented as one-hot vectors. Note that this case study is a preliminary exploration of the use of tensor network embeddings for generative modeling, and the investigation is not exhaustive. Therefore, we only present our findings on the Ms.Pac-Man game for brevity in the next section.

As the use of tensor-network embeddings increase the efficiency of the data we use in

the sense that we can provide more information for the same amount of data, we wanted to explore the effects of the number of training episodes on the quality of the diffusion process. Therefore we trained one model with all 600 episodes of RGB frames, one model with all 600 episodes of latent frames, and one model with only 300 episodes of RGB frames.

5.3.3 Results and Discussion

All trained models performed considerably well in terms of the stability and accuracy of the generated images when the diffusion process is conditioned using frames and actions from the ground truth gameplay episodes. Therefore we omit them from our analysis.

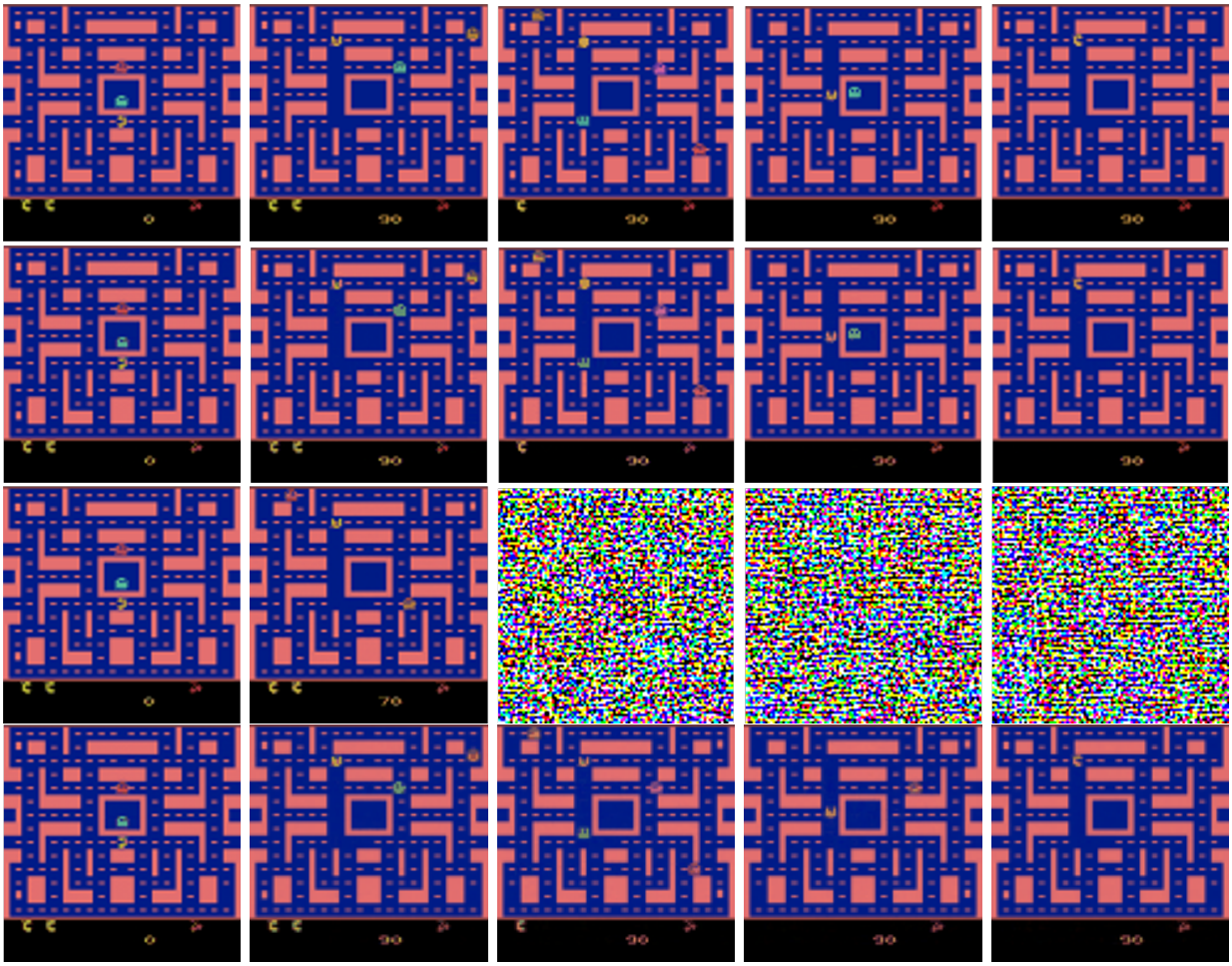
To evaluate the performance of the trained models, we perform autoregressive frame generation using actions from ground truth gameplay episodes. Figure 5.20 shows the performance of the models trained on the full latent dataset and the half-sized RGB dataset during autoregressive frame generation using a context window of 4 previous frames and actions. The first two rows show the original frames and the reconstructions from the latent space, respectively. The third row shows the frames generated by the diffusion model trained on the RGB frames. The last row shows the frames generated by the diffusion model trained on the latent embeddings.

The diffusion process quickly diverges to pure static noise within 4 seconds of real time for the RGB frames, while the diffusion done in the latent space is able to track the ground truth trajectories until the end of the episode. This serves as an evidence for the additional stability and robustness provided by expanding the training dataset to contain a similar amount of data[6].

The reduced dimensionality of the latent representations make it easier for the diffusion model to learn the denoising process, as it can focus on the essential features of the data without being overwhelmed by the high-dimensional noise. Figure 5.21 shows how the latent space representations evolve over time during both autoregressive generation and "closed-loop" generation, where the diffusion model is conditioned on the ground truth latent space representations. The figure on top shows how the 5 largest latent variables evolve over time, whereas the bottom figure shows the evolution of the 5 smallest latent variables. In both cases we see that the generative model is able to track the dynamics of the latent space representations over time, which is a promising direction for future work.

Finally, we also investigated if the diffusion models trained on the downsized RGB frames suffer from instability issues when the number of episodes used for training is matched to the number of episodes used for training the latent diffusion models. Figure 5.22 shows the

[6]Note that the notion of "similar amount of data" is in terms of number of floating point values, as the latent space representations are roughly 2 times smaller than the downsized RGB frames.



(a) $t = 0$ (b) $t = 40$ (c) $t = 80$ (d) $t = 120$ (e) $t = 160$

Figure 5.20: Visuals from the generative modeling of ATARI games using tensor-network embeddings. The first row shows the ground truth frames from the ATARI game. The second row shows the corresponding latent space representations obtained through HT-RISE, which captures the essential features of the game. The third row shows the denoised RGB frames generated by the diffusion model, while the fourth row shows the denoised latent space representations. RGB diffusion suffers from stability issues, while latent diffusion yields stable results. Columns correspond to different time steps in the diffusion process, with $t = 0$ being the initial state and $t = 160$ being the final state.

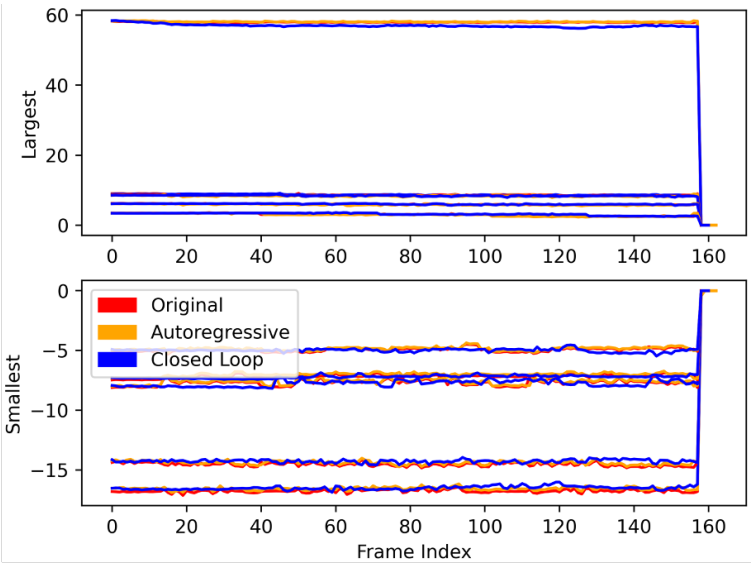
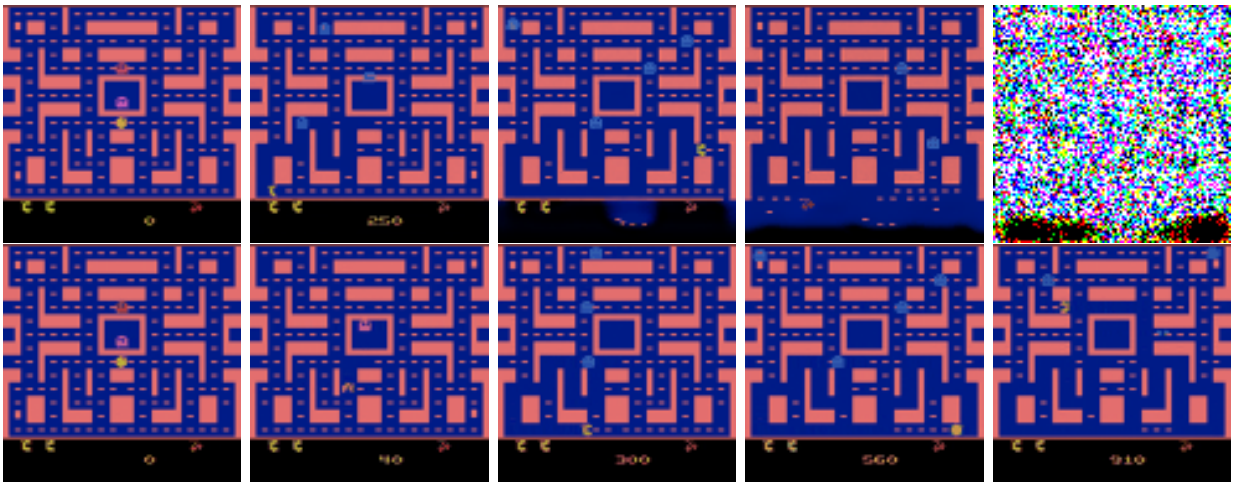


Figure 5.21: Dynamics of the latent space representations during the diffusion process. The figure shows the 5 largest (top) and 5 smallest (bottom) values from the latent space representations for each frame. Red lines indicate the original latent space values obtained by projecting the ground truth frames onto the latent space, while blue lines indicate the denoised latent space values obtained by performing the diffusion process using ground truth frames as input. Finally, the yellow lines indicate the denoised latent space values obtained by performing the diffusion process using the generated frames as input. Both methods are able to track the overall dynamics of the latent space.

results of this investigation, where we compare the frames generated by the diffusion model trained on the full RGB dataset (600 episodes) with the frames generated by the diffusion model trained on the full latent dataset (600 episodes). As it can be seen from the figure, the playing field starts to bleed into the score field during the autoregressive generation process for the RGB diffusion model, leading to artifacts. The model trained on RGB frames can carry on with generating frames for a while, however the artifacts get worse as the game time progresses, leading to the total loss of the score field and the frames suddenly diverge to pure static noise at around $t = 300$. On the other hand, the latent diffusion model is able to track the dynamics of the game until the end of the episode, without any artifacts or instability issues. This example demonstrates that the use of tensor network embeddings can help improving the stability and robustness of generative models, especially in high-dimensional data spaces.

5.4 Summary

In this chapter we have explored the use of incremental tensor decompositions for extracting a low-dimensional representation of image and video data. In Section 5.1, we have demon-



(a) $t = 0$ (b) $t = 75$ (c) $t = 150$ (d) $t = 225$ (e) $t = 300$

Figure 5.22: Bleeding within the frames during autoregressive frame generation process. The figure shows the denoised frames generated by the diffusion models at different time steps. The top row shows frames from the diffusion model trained on the full RGB dataset (600 episodes), while the bottom row shows frames from the diffusion model trained on the full latent dataset (600 episodes). The frames are generated using a context window of 4 previous frames and actions. The image generation process starts well for the RGB model but then begins to produce artifacts, such as bleeding of the playing field to the score field. The issue gets worse as the game time progresses, leading to the total loss of the score field and the frames suddenly diverge to pure static noise. This behavior is not observed in the latent diffusion model, which is able to track the dynamics of the game until the end of the episode. The columns correspond to different time steps in the autoregressive generation process denoted in their respective captions.

strated the effectiveness of HT-RISE for compressing video streams and high-resolution images from Unreal Engine 5 environments, achieving compression rates commensurate with JPEG encoding, while preserving the ability to reconstruct high-quality images and videos. The results show that HT-RISE can achieve compression rates of 25.5× on high-resolution video streams while maintaining a peak signal-to-noise ratio (PSNR) of 30.45 dB. The results also show that HT-RISE can achieve compression rates of 9.8× without sacrificing much from the structure of the images, achieving a SSIM of 0.76. Furthermore we have demonstrated that through quantization of the latent representations obtained through HT-RISE, we can challenge JPEG-level compression on disk. This demonstrates the effectiveness of tensor decompositions in learning the underlying structure of the data, leading to efficient compression without significant loss of quality. Furthermore, we have shown that the full-to-latent mapping learned by HT-RISE can generalize well beyond the domain of the training data, allowing for accurate reconstruction of unseen data. This is particularly important for applications where the data distribution may change over time, as it allows for the model to adapt to new data without requiring significant retraining.

Additionally, in Section 5.2 we have explored using tensor networks as a method for learning a low-dimensional representation of image data for behavioral cloning. In particular, we compare the computational speed of using TT against AEs and VAEs for extracting this low-dimensional representation, finding a 2.6× to 9.6× speedup compared to AEs or VAEs. Furthermore, the TT-encoding can be efficiently trained on CPU as well, which leads to comparable or better training times than the AE and VAE trained on GPU (0.9× to 5.4× faster). This speedup comes at no cost in terms of behavioral cloning accuracy or the gameplay scores achieved by the behaviorally cloned agents. As such, the TT-based approach enables learning while having limited access to GPUs, as it allows for the expensive dimensionality reduction step to be performed on CPUs at a comparable or faster time than using an AE or VAE on GPUs. Given the large computational requirements of learning directly from video data, future work will aim to extend this methodology to tackling larger game environments for which the large input space is a computational bottleneck.

Finally, we have explored the use of tensor network embeddings in generative modeling tasks, specifically in the context of denoising diffusion models. In Section 5.3, we have demonstrated that the low-dimensional embeddings obtained through HT-RISE can be effectively used to train denoising diffusion models. The preliminary results indicate that performing the denoising diffusion process in the latent space leads to more stable results compared to performing it in the RGB space. The results also indicate that we can track the dynamics of the latent space representations over time even in the autoregressive setting, which is a promising direction for future work.

Future work in this field can focus on exploring architectures that can leverage the low-dimensional representations obtained through tensor networks. One good argument for this is that the low-dimensional embeddings obtained through tensor networks do not have any explicit spatial correlation between the neighboring features in the latent space. This makes the use of convolutional layers in the UNet architecture less effective, as the convolutional layers are designed to exploit spatial correlations in the data. As such, future work can explore architectures that are better suited for working with low-dimensional representations, such as fully connected networks or transformer-based architectures.

Another interesting direction for future work would be to explore the use of tensor networks in the context of generative modeling tasks, i.e., replacing the UNet architecture with a tensor-network based one for the denoising diffusion process. This would allow exploring other training paradigms such as ALS instead of performing **S**tochastic **G**radient **D**escent (SGD) based optimization.

5.5 Acknowledgements

We acknowledge support from the Automotive Research Center (ARC) in accordance with Cooperative Agreement W56HZV-19-2-0001 with U.S. Army DEVCOM Ground Vehicle Systems Center. The research in this chapter was supported in part through computational resources and services provided by Advanced Research Computing at the University of Michigan, Ann Arbor.

CHAPTER 6

Conclusion and Future Work

This thesis has explored the use of tensor network embeddings for data compression, supervised learning, generative modeling, and Bayesian inference. We have developed several algorithms that enable efficient low-rank tensor decompositions for large-scale streaming data, and we have demonstrated their effectiveness in various applications, including scientific data compression, behavioral cloning, and generative modeling.

6.1 Discussion of Contributions and Insights

This thesis has introduced novel methods for the incremental and batch processing of data using tensor decompositions, leading to significant advances in handling large-scale and streaming data. The key contributions and insights are:

- **The TT-ICE Algorithm for Incremental Tensor Train Decomposition:** The development of the TT-ICE algorithm demonstrates a significant advancement in the ability of computing a low-rank approximation of large-scale or streaming data in the TT format. The key insight is that by leveraging the orthogonality of the TT cores and using efficient update rules inspired by incremental SVD techniques, we can incrementally add missing basis vectors to each TT-core as new data becomes available, resulting in state-of-the-art performance across incremental tensor train decomposition algorithms as demonstrated in Section 3.1. This allows for fast updates to the existing approximation without needing to re-process the entire dataset. This opens the door to applications in online learning and real-time monitoring where data arrives sequentially.

- **The Batch Hierarchical Tucker Format and BHT-l2r Algorithm for Batches of Tensors:** The introduction of the batch hierarchical Tucker format and the BHT-l2r algorithm addresses the challenge of processing a batch of tensors received from the same data source. The underlying insight is that if the tensors in the batch are sampled

from the same underlying distribution, they are likely to share common features that can be exploited for compression. This approach also paves the way for developing incremental algorithms for the hierarchical Tucker format, which is a more suitable format for data with multi-scale structures.

- **The HT-RISE Algorithm for Incremental Hierarchical Tucker Decomposition:** The HT-RISE algorithm extends the idea of performing incremental updates to the hierarchical Tucker format, which is better suited for high-dimensional data with multi-scale structures. The key insight is that by leveraging the orthogonality of the hierarchical Tucker cores and using efficient update rules, we can incrementally add new data to the existing approximation. This overcomes some of the structural limitations of the TT format and provides a powerful tool for streaming high-dimensional data, such as that from video or complex scientific simulations as demonstrated in Section 3.3.

- **Practical Viability and Validation in Scientific Domain:** By applying these methods to scientific data compression and Bayesian inference, this thesis has demonstrated the practical effectiveness of incremental tensor decompositions and the latent space representations obtained through them. The key insight is that the theoretical benefits of tensor decompositions—namely, their ability to break the curse of dimensionality for low-rank data—translate into tangible performance gains on real-world problems, preserving essential features while significantly reducing data size. The suitability of tensor networks for scientific data manifests in their ability to achieve high compression ratios (up to $300,000\times$) while incurring negligible loss in the quality of reconstructions, as well as enabling efficient Bayesian inference, as shown in Section 4.1 and Section 4.2. This highlights the potential of tensor networks as a powerful tool for scientific computing and data analysis.

- **Versatility in Machine Learning and Computer Vision:** The exploration of tensor network embeddings for image and video data showcases their flexibility beyond traditional scientific computing. This thesis has demonstrated that the same principles of low-rank approximation can be effectively used to compress high-fidelity, high-resolution image streams, almost to the extent of JPEG compression, while preserving the ability to reconstruct high-quality images and videos, as shown in Section 5.1. Furthermore, the latent representations obtained through tensor networks can be effectively used for downstream tasks such as behavioral cloning and generative modeling. The key insight is that tensor networks can serve as a powerful alternative

to other dimensionality reduction techniques, such as autoencoders, providing comparable performance with significantly reduced training times (up to 9.6× faster), as demonstrated in Section 5.2 and Section 5.3. This highlights the potential of tensor networks to be integrated into machine learning pipelines.

6.2 Limitations

Despite their strengths, the methods developed in this thesis have limitations that define the scope of their applicability and suggest areas for future research.

- **The Low-Rank Assumption:** The most significant limitation of all tensor decomposition methods, including those presented here, is their reliance on the assumption that the data can be well-approximated by a low-rank tensor. This assumption holds for many types of structured data, such as solutions to certain PDE or video sequences from structured environments, but it may not be valid for highly complex or noisy data. When the data does not have a low-rank structure, the compression ratio will be poor, and the computational benefits will diminish.

- **Computational Complexity:** While tensor decompositions are designed to mitigate the curse of dimensionality, their computational cost can still be substantial. The complexity of the algorithms depends polynomially on the tensor ranks and mode sizes. For data that requires high ranks to achieve a good approximation, the computational and memory costs can become prohibitive, especially in real-time streaming applications. This is a particular challenge for the TT-ICE algorithm, where the growth of the TT-ranks depends on all preceding ranks.

- **Numerical Stability:** The numerical stability of the proposed algorithms can be a concern, especially when using a lower precision arithmetic (e.g., single precision). This can lead to issues such as loss of orthogonality in the tensor cores and accumulation of rounding errors, which can result in degraded approximation quality over time. While techniques such as re-orthogonalization can mitigate these issues, they add to the computational overhead.

- **Hyperparameter Selection:** The performance of the proposed algorithms is sensitive to the choice of hyperparameters, particularly the tensor reshaping and the ordering of the dimensions, or error tolerance used to determine the ranks. Finding the optimal tensor reshaping and dimension ordering is a non-trivial task that often requires domain knowledge and empirical tuning. While heuristics such as selecting

dimensions with strong correlations to be adjacent and picking dimensions of order $O(10)$ can help, there is no one-size-fits-all solution. This can limit the accessibility of these methods to practitioners who may not have deep expertise in tensor methods.

6.3 Future Work

The insights and limitations discussed above naturally lead to several avenues for future research. We group the potential areas for future work into two main categories: algorithmic developments and applications in machine learning and data science.

6.3.1 Algorithmic Developments

A key area for future research is the extension of the proposed algorithms for different real-world scenarios. These improvements would also pave the way for the adoption of tensor network embeddings in various applications.

One important requirement for that is the development of user-friendly software packages that integrate the proposed algorithms into existing machine learning frameworks. Providing end-to-end pipelines can facilitate their adoption in the community and enable researchers and practitioners to leverage tensor network embeddings for their applications.

6.3.1.1 Distributed and Federated Learning

Developing algorithms for incremental decompositions on federated datasets is a promising research direction. This would involve extending the proposed algorithms to handle distributed data sources while preserving privacy and ensuring efficient communication between nodes. Algorithms that can adaptively learn from streaming data in a federated setting, where data is distributed across multiple devices, can enable scalable machine learning systems that learn from large-scale datasets without centralized storage. This would also allow for the adoption of the proposed algorithms in edge computing scenarios, where data is generated and processed on devices like IoT sensors or mobile phones.

6.3.1.2 Data-Aware Compression

Future research can also explore more data-aware compression approaches that adaptively modify tensor network embeddings based on data characteristics. This includes developing algorithms that dynamically adjust the compression rates, ranks, and structures of the tensor networks based on the data distribution and noise levels. Additionally, algorithms that learn

from the data to optimize the compression process can lead to more efficient and effective tensor network embeddings.

An immediate research direction in this category is the extension of the batch-HT format and the HT-RISE algorithm to support n-ary splits for the dimension tree. This would allow for more tailored decompositions that can cater to specific data characteristics, leading to more efficient compression.

6.3.1.3 Multi-Modal Data Processing

Investigating algorithms for processing multi-modal data is another promising research area. This would involve developing tensor network embeddings that can effectively capture the relationships between different modalities (e.g., images, text, audio). Developing algorithms that can integrate multi-modal data into a unified framework for compression and analysis can enable novel applications in areas such as multi-modal learning, cross-modal retrieval, and multi-modal generative modeling. This could also involve exploring the use of tensor networks for multi-modal data fusion, where information from different modalities is combined to improve the overall representation and analysis.

6.3.2 Applications in Machine Learning and Data Science

The applications of tensor network embeddings can be further explored in various domains of machine learning and data science. For example, they can be applied to tasks such as anomaly detection, clustering, and recommendation systems, where the ability to efficiently compress and represent high-dimensional data can lead to improved performance and interpretability. Additionally, integrating tensor network embeddings with deep learning models can enable the development of novel architectures that leverage the strengths of both approaches, leading to improved performance in tasks such as image classification, natural language processing, and generative modeling.

6.3.2.1 Bayesian Optimal Experimental Design

The use of tensor network embeddings in Bayesian optimal experimental design can be further explored to improve the efficiency and effectiveness of this process. This could involve developing algorithms that leverage the properties of low-rank tensor networks in the Bayesian optimization loop to reduce computational requirements. Furthermore, we can explore compression-aware optimization techniques that make informed decisions about the design procedure, accounting for the inherent but bounded error introduced by compression.

An immediate research direction in the Bayesian realm is to combine the generalization capabilities of tensor networks showcased in Section 5.1 with their Bayesian inference capabilities demonstrated in Section 4.2 to perform image inpainting. This could become a powerful tool for tasks such as image restoration for autonomous driving, where reconstructing missing or corrupted parts of images can significantly improve the performance of perception systems.

6.3.2.2 Generative Modeling

In generative modeling, tensor network embeddings can be used to improve the efficiency and effectiveness of generative models. This could involve developing algorithms that leverage the low-dimensional representations from tensor network embeddings to generate high-quality samples, such as images, videos, and text. Additionally, integrating tensor network embeddings with models like denoising diffusion models can enable more efficient and effective generative models that capture complex data distributions. This could also involve exploring tensor networks for tasks such as image synthesis, video generation, and text generation, where efficient representation of high-dimensional data can lead to improved performance and interpretability.

APPENDIX A

Aditional Comparisons for TT-ICE Algorithm

This appendix includes additional results on the performance of the TT-ICE algorithm on other Atari games and a comparison of TT-ICE with TT-FOA algorithm.

A.1 Other Atari Games

This section includes the results of the experiments with the other ATARI games and their discussion.

TT-ICE and TT-ICE* proved their performance in the first experiments with Ms.Pac-Man frames. Next, we investigated the performance of algorithms across different hardware combinations.

For these experiments, we focus our attention on TT-ICE, TT-ICE*, and ITTD and to understand performance across different hardware combinations, we tested the algorithms on several machines. For each experiment, we detail the machine on which it was conducted. The machines are *Dell Precision 7820* workstation with Xeon Silver 4110 processor and 32 GB memory, *Dell XPS-15 9570* laptop with i7-8750H processor and 16 GB memory, and *Dell Precision 7820* workstation with Xeon Silver 4110 processor and 16 GB memory. The machines are referred to in Table A.1 as M1, M2, and M3, respectively. Table A.1 indicates improvements between 44.5× to 56.7× in compression and between 8.2× to 23.5× in time over ITTD5.

In Table A.1, TT-ICE and TT-ICE* complete the compression of streams faster than ITTD5. Furthermore, our proposed algorithms achieve at least an order of magnitude higher compression ratio than ITTD5. In Table A.1, ITTD5 fails without an exception to compress the entire stream due to insufficient memory when $\varepsilon_{des} = 0.01$. This is caused by the high computational requirements of the rounding step in ITTD5 and becomes prohibitive even with $\varepsilon_{des} = 0.1$ for games Beamrider and Enduro. The only case where all three methods fail to compress the entire stream is Enduro with $\varepsilon_{des} = 0.01$ due to the limited memory of

the machine. However, even in that case, we see that TT-ICE can compress at least 3 times more frames than ITTD5.

An impressive point from Table A.1 is the difference in number of frames used between TT-ICE and TT-ICE*. By subselecting frames and skipping the update of the first d TT-cores, TT-ICE* reduces the number of frames used in updates by up to 97% (Pong with $\varepsilon_{des} = 0.1$) without any loss in mean RRE. The reduction in time is also parallel with the reduction in the frames used. TT-ICE* achieves at least 48% reduction (Seaquest with $\varepsilon_{des} = 0.01$) in execution time over TT-ICE.

In Table A.1, Beamrider with $\varepsilon = 0.01$ is another insightful example, which shows that implementing heuristics can be influential on the success of TT-ICE. In that case, TT-ICE* successfully completes the decomposition of the stream, whereas TT-ICE fails to complete the same task.

The additional experiments using ATARI data show that TT-ICE and TT-ICE* can successfully compress streams across different hardware combinations. Furthermore, they provide proof that the decrease in compression time over ITTD is not hardware dependent.

To provide a qualitative perspective on what each of the ε_{des} levels corresponds, we provide reconstructions of the frames compressed both with $\varepsilon_{des} = 0.1$ (Figure A.1) and $\varepsilon_{des} = 0.01$ (Figure A.2). In Figure A.1 we include ITTD5 reconstructions in addition to TT-ICE reconstructions. Figure A.1 shows that there are visual distortions on each reconstructed frame regardless of the algorithm used. These artifacts caused at the error tolerance of $\varepsilon_{des} = 0.1$ can be as unobtrusive as discolored parts/objects (Breakout, Ms.Pac-Man) but also can be completely disruptive as invisible agents (Pong, Q*bert).

As the error tolerance is reduced to $\varepsilon_{des} = 0.01$, the visual artifacts in the reconstruction disappear. In Figure A.2, we see that for $\varepsilon_{des} = 0.01$ the frames can be reconstructed with no visible disruptions. This also provides evidence that TT-cores trained with TT-ICE and TT-ICE* can be used as a method of storage. Unfortunately at this error tolerance level, we can not provide reconstructions of ITTD, as ITTD5 fails due to insufficient memory for all games.

A.1.1 Comparison with TT-FOA

This section includes the comparison study between TT-ICE* and TT-FOA along with its discussion. In regimes where we found TT-FOA effective to run. We were not able to deploy TT-FOA on our problems of interest because it required significantly greater storage and computational requirements than TT-FOA or ITTD due to large-scale auxiliary matrices. For example, the TT-ranks used to approximate the tensors in [1] are very low

Table A.1: Summary statistics for the other ATARI games. Game: name of the game, Machine: code-name of the computer that conducted the experiments, Method: decomposition algorithm used (TT-ICE*: Algorithm 2, TT-ICE: Algorithm 1, ITTD: [161, Alg 3]), ε: the desired relative error bound for the given decomposition method, Frames: total number of the frames compressed in the TT-cores, Used: number of frames used to update the TT-cores, Time: total execution time of the algorithm, mean RRE: final mean RRE at the end of the experiment, CR: compression ratio. For each experiment, the best performance is shown in bold for Time, mean RRE, and CR. All failures are due to insufficient memory.

Game	Machine	Method	ε	Frames	Used	Time	mean RRE	CR
Breakout	M2	TT-ICE	0.1	9295	9295	212.2	0.078	2748.1
		TT-ICE*	0.1	9295	2376	**58.9**	**0.084**	**3525.9**
		ITTD 5	0.1	9295	9295	801.2	0.083	66.55
Breakout	M2	TT-ICE	0.01	9295	9295	572.7	0.0075	9.7
		TT-ICE*	0.01	9295	4997	**198.5**	**0.0078**	**10.1**
		ITTD 5	0.01	436	436	Fails	Fails	Fails
Beamrider	M2	TT-ICE	0.1	42473	42473	2689.1	**0.089**	**130.7**
		TT-ICE*	0.1	42473	4017	**193.0**	0.087	122.3
		ITTD 5	0.1	22606	22606	Fails	Fails	Fails
Beamrider	M2	TT-ICE	0.01	26257	26257	Fails	Fails	Fails
		TT-ICE*	0.01	42473	22693	**2334.3**	**0.008**	**4.3**
		ITTD 5	0.01	4937	4937	Fails	Fails	Fails
Enduro	M1	TT-ICE	0.1	132880	132880	11158.8	**0.096**	**246.6**
		TT-ICE*	0.1	132880	10043	**940.1**	0.095	225.6
		ITTD 5	0.1	96329	96329	Fails	Fails	Fails
Enduro	M1	TT-ICE	0.01	49819	49819	Fails	Fails	Fails
		TT-ICE*	0.01	76396	40482	Fails	Fails	Fails
		ITTD 5	0.01	13289	13289	Fails	Fails	Fails
Q*bert	M3	TT-ICE	0.1	18742	18742	1063.6	0.089	817.7
		TT-ICE*	0.1	18742	3993	**180.7**	**0.094**	**968.4**
		ITTD 5	0.1	18742	18742	1477.5	**0.094**	18.0
Q*bert	M3	TT-ICE	0.01	18742	18742	1740.4	0.006	13.6
		TT-ICE*	0.01	18742	5272	**470.0**	**0.008**	**15.8**
		ITTD 5	0.01	2286	2286	Fails	Fails	Fails
Seaquest	M1	TT-ICE	0.1	36805	36805	1793.8	0.091	1176.8
		TT-ICE*	0.1	36805	3183	**173.1**	**0.096**	**1703.7**
		ITTD 5	0.1	36805	36805	2435.0	0.096	30.4
Seaquest	M3	TT-ICE	0.01	36805	36805	3907.0	0.007	3.1
		TT-ICE*	0.01	36805	21194	**2024.6**	**0.008**	**3.4**
		ITTD 5	0.01	6667	6667	Fails	Fails	Fails
Pong	M1	TT-ICE	0.1	96302	96302	1566.0	**0.066**	**99835.9**
		TT-ICE*	0.1	96302	2738	**70.0**	**0.066**	**99835.9**
		ITTD 5	0.1	96302	96302	1644.6	0.065	2234.6
Pong	M1	TT-ICE	0.01	96302	96302	12895.3	**0.009**	**48.3**
		TT-ICE*	0.01	96302	17219	**1283.2**	**0.009**	47.2
		ITTD 5	0.01	46015	46015	Fails	Fails	Fails

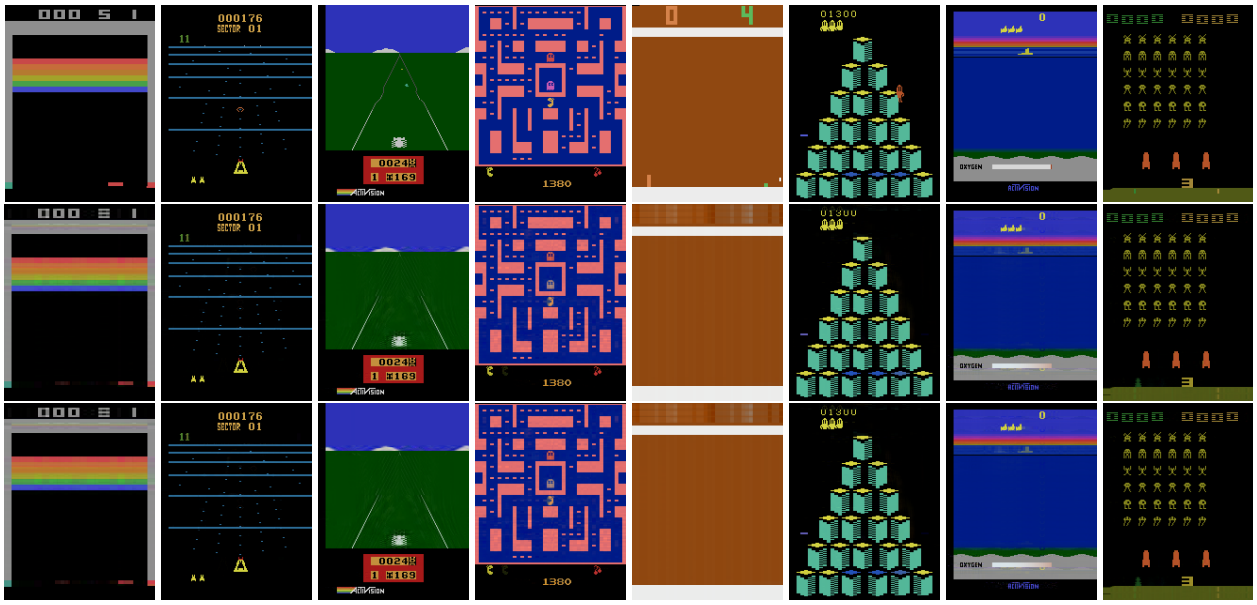


Figure A.1: Original frames from the Atari dataset (top) along with their respective reconstructions from the TT-cores compressed using TT-ICE (middle) and ITTD5 (bottom) with $\varepsilon_{des} = 0.1$. There are visible visual artifacts on some of the frames. There is no visible difference between TT-ICE and ITTD5 reconstructions. Left-to-right: Breakout, Beam Rider, Enduro, Ms.Pac-Man, Pong, Q*bert, Seaquest, Space Invaders

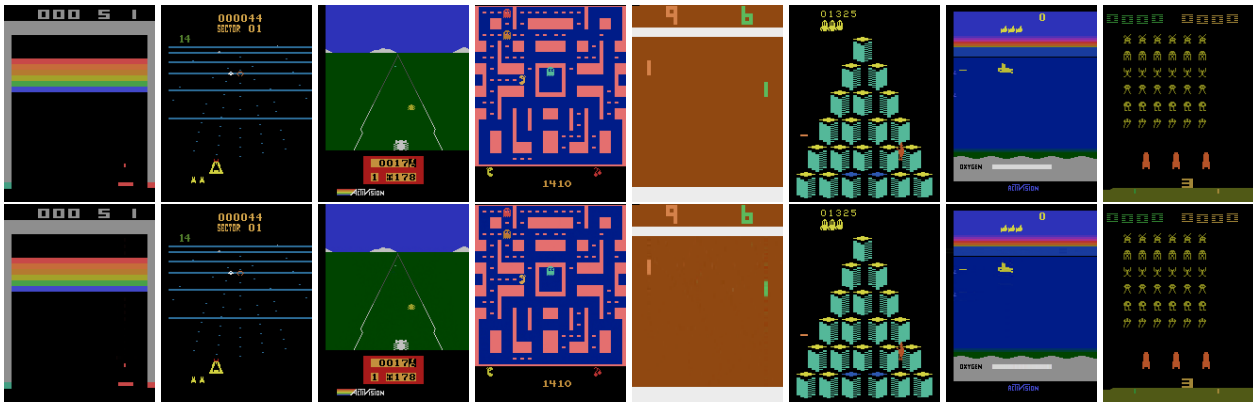


Figure A.2: Original frames from the Atari dataset (top) along with their respective reconstructions (bottom) from the TT-cores compressed using TT-ICE with $\varepsilon_{des} = 0.01$. There are no visible artifacts on the frames. ITTD5 reconstructions are not included here since ITTD5 fails at all games for $\varepsilon_{des} = 0.01$. Left-to-right: Breakout, Beam Rider, Enduro, Ms.Pac-Man, Pong, Q*bert, Seaquest, Space Invaders

in comparison to the final TT-ranks that we obtain in our numerical experiments. As a result, we performed a comparison using a similar testbed to that in the TT-FOA paper. To be able to provide a fair comparison between TT-ICE* and TT-FOA, we implemented the

Matlab script provided by authors of [1] in python[1] and benchmarked against TT-ICE* using a synthetic 4-dimensional tensor with size $10 \times 15 \times 20 \times 500$ with TT-ranks $[1, 2, 3, 5, 1]$, which is same as the study conducted in [1].

We considered 3 scenarios for TT-FOA: 1) TT-FOA initialized with the true TT-ranks, 2) TT-FOA initialized with underestimated TT-ranks, and 3) TT-FOA initialized with overestimated TT-ranks. On the other hand, for TT-ICE* we set $\varepsilon_{des} = 10^{-8}$ since the synthetic tensor is exactly low rank. We then constructed 20 random 4-dimensional tensor streams having increments of size $10 \times 15 \times 20$.

Figure A.3 shows the relative error of the approximation at each increment step averaged over 20 repetitions of the experiment. A comparison of the compression ratio of methods is not provided here since the TT-ranks are assumed known from the beginning for TT-FOA.

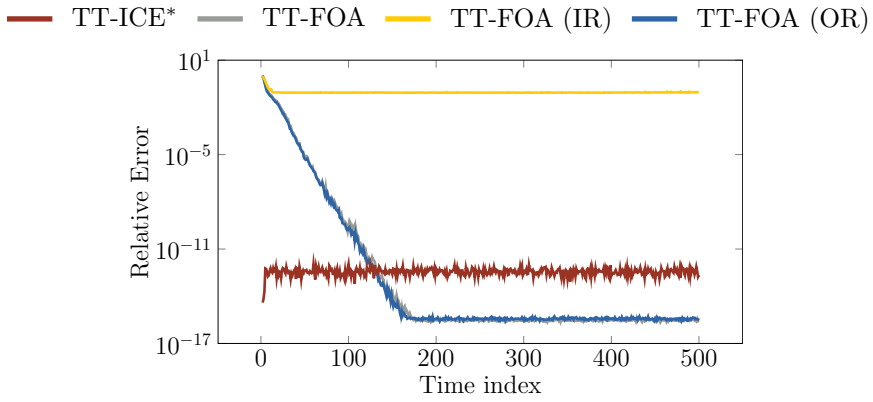


Figure A.3: Comparison of relative errors of TT-ICE* and TT-FOA[1] averaged over 20 repetitions. TT-ICE*: TT-ICE* with $\varepsilon_{des} = 10^{-8}$, TT-FOA: TT-FOA initialized with correct TT-ranks, TT-FOA (IR): TT-FOA initialized with insufficient TT-ranks, TT-FOA (OR): TT-FOA initialized with overestimated TT-ranks. Unlike TT-FOA, TT-ICE* achieves a low relative error without the need for a convergence period and discovers the correct underlying TT-ranks. In case of underestimated TT-ranks, TT-FOA fails to achieve low relative error.

Figure A.3 shows that TT-FOA requires a convergence period before reaching a stable relative error even when it is initialized with correct TT-ranks. In the case of underestimated TT-ranks, TT-FOA converges to a higher relative error depending on the difference in the estimated and actual TT-ranks. TT-FOA converges to the same level of relative error as the correctly estimated case when the TT-ranks are overestimated. This behavior is expected for this set of experiments since we have an exactly low-rank tensor stream. On the other hand,

[1]The accuracy of the python implementation was verified by comparing the results of our python implementation and the published Matlab script using the same 4-dimensional tensor streams. To time TT-FOA fairly in the following experiments, we removed the relative error computation that was included in the Matlab implementation of the algorithm and computed the relative error separately.

TT-ICE* neither requires estimation of the TT-ranks beforehand, nor needs a convergence period to achieve the desired level of relative error.

The difference between both methods becomes evident when the time to compress the streams is considered. TT-FOA performs Kronecker products and matrix inversions, and the size of the matrices involved in those operations plays a decisive role in the speed of the algorithm. Therefore, there is a notable difference in computation time between different TT-rank estimations. When the correct TT-ranks are estimated ([1,2,3,5,1]), compression of the streams takes on average $1.8070s$. This time becomes $1.5543s$ when TT-ranks are underestimated as [1,1,2,3,1] and increases to $4.1290s$ when TT-ranks are overestimated as [1,3,5,10,1]. TT-ICE* compresses the entire stream in $0.5806s$ on average.

When the same experiment is repeated with a tensor of the same size but with TT-ranks [1,4,10,30,1], TT-FOA requires $314.53s$ on average with correct TT-ranks. This time can get as low as $14.965s$ when TT-FOA is initialized with underestimated TT-ranks [1,3,5,20,1] and can get as high as $628.70s$ with overestimated TT-ranks [1,6,15,40,1]. On the other hand, TT-ICE* scales very well in this scenario and compresses the entire stream in $0.6656s$ on average.

APPENDIX B

Additional Performance Analysis of the HT-RISE Algorithm

This appendix provides a comparison between the HT and Batch-HT formats, which are used for storing and processing data in high-dimensional tensor formats. The HT format is designed for efficient storage and computation of tensors, while the Batch-HT format extends this concept to handle batches of tensors.

B.1 Effect of the Dimension Tree on the Compression

As discussed in Section 3.3, the construction of the dimension tree has a direct impact on the compression performance of the hierarchical Tucker decomposition. In this section, we investigate the effect of various alternatives of the dimension tree and their effect on the compression outcomes. Appendix B.1.1 investigates reshaping the initial data to higher (or lower) dimensions, and Appendix B.1.2 investigates the effect of changing the axis reordering.

B.1.1 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 data set with $\varepsilon_{rel} = 0.10$ and no normalization. We consider four different reshaping of the tensor. The baseline reshaping is denoted as $8\times8\times8\times8\times8\times8\times5\times21\times1$. 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 B.1 shows the compression time and relative test error of the algorithms using various reshaping. Figure B.2 shows the comparison of compression time and RTE of the algorithms using the considered reshapings. Table B.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 3.3.2.2

Table B.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 data set with $\varepsilon_{rel} = 0.10$. Time: Compression time in seconds, CR: Compression ratio, RR: Reduction ratio, RTE: Relative test error. ‡ denotes that the algorithm failed to compress the data set 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
$64 \times 64 \times 64 \times 5 \times 21 \times 1$	$5289.4^{\ddagger}$	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

In Figure B.1 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×) and RR (33.87×) among the investigated reshaping. The $64 \times 64 \times 64 \times 5 \times 21 \times 1$ reshaping results in the lowest CR (18.86×) 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 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 data set.

Figure B.2 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 ($11,339$s). 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 data set was successfully compressed.

Another observation from Figure B.2 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 likely due to the fact that the algorithm fails to compress the entire data set 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.

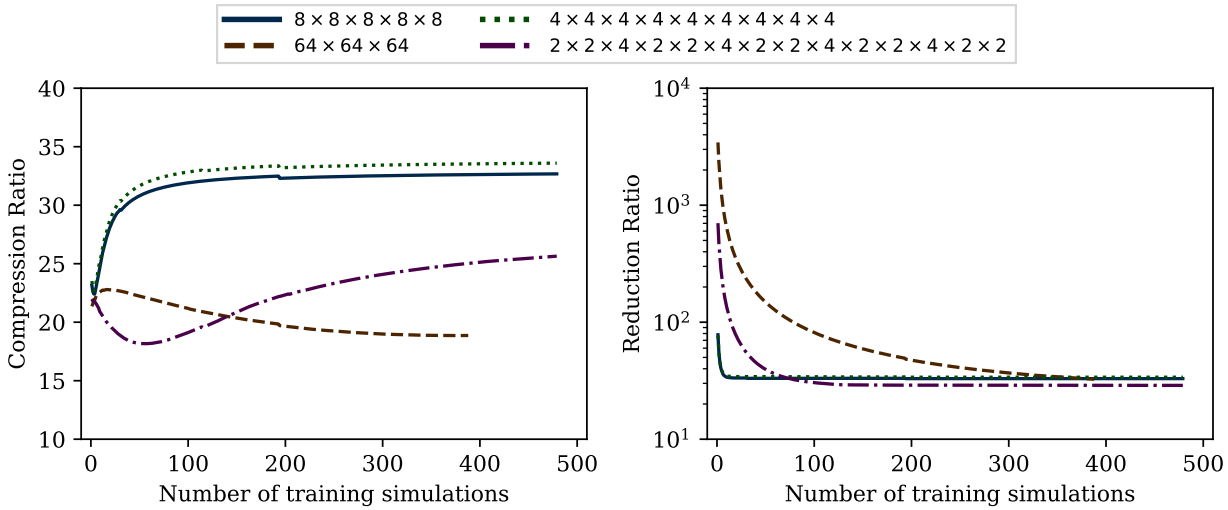


Figure B.1: Comparison of various tensor reshapings in terms of compression ratio (CR - left) and reduction ratio (RR - right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using no normalization. The reshaping $4 \times 4 \times 4 \times 4 \times 4 \times 4 \times 4 \times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

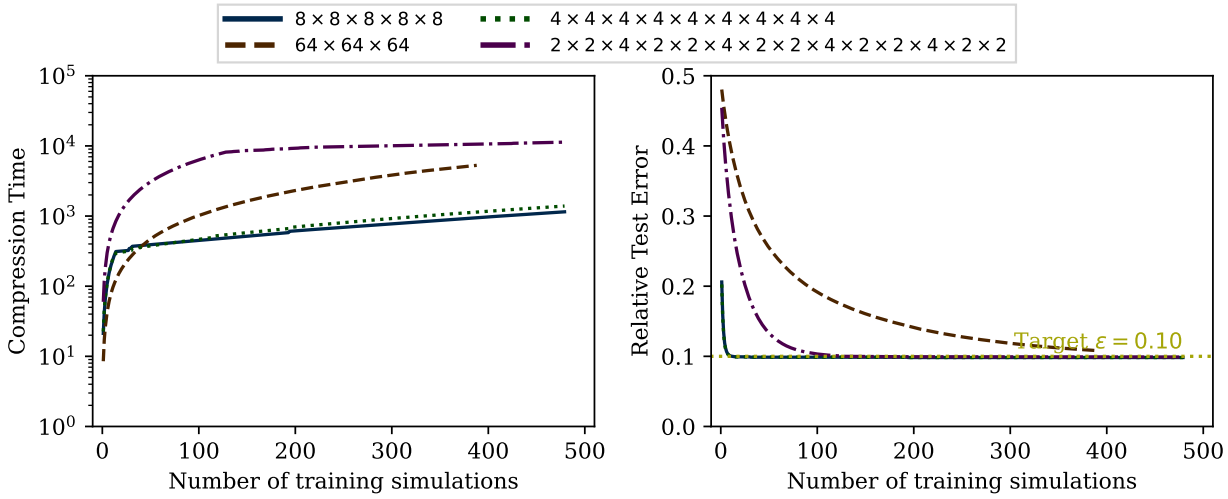


Figure B.2: Comparison of various tensor reshapings in terms of compression time (left) and relative test error (RTE - right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using no normalization. The reshaping $2 \times 2 \times 4 \times 2 \times 2 \times 4 \times 2 \times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

B.1.2 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 interaction 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 data set 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 B.3 shows the compression time and relative test error of the algorithms using various axis reorderings. Figure B.4 shows the comparison of compression time and mean validation error of the algorithms using various axis reorderings. Table B.2 presents the axis ordering, shapes of the corresponding tensors, and the resulting compression time, CR, RR, as well as RTE.

Table B.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 data set 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: Reduction 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

Figure B.3 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× CR and 32.94× RR, whereas the second-best axis ordering, Transpose A, achieves 26.24× CR and 26.41× RR. CR and RR drop down to 18.93× and 19.06× for the worst performing axis ordering, Transpose B.

Figure B.4 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.

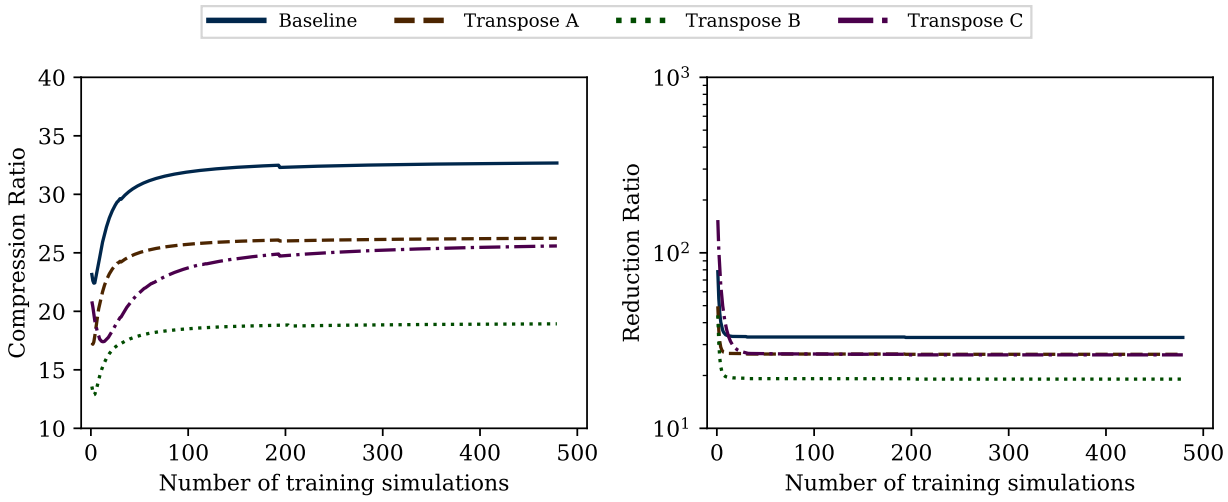


Figure B.3: Comparison of various axis orderings in terms of compression ratio (CR - left) and reduction ratio (RR - right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using no normalization. The baseline axis ordering results in the highest CR and RR among the investigated 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.

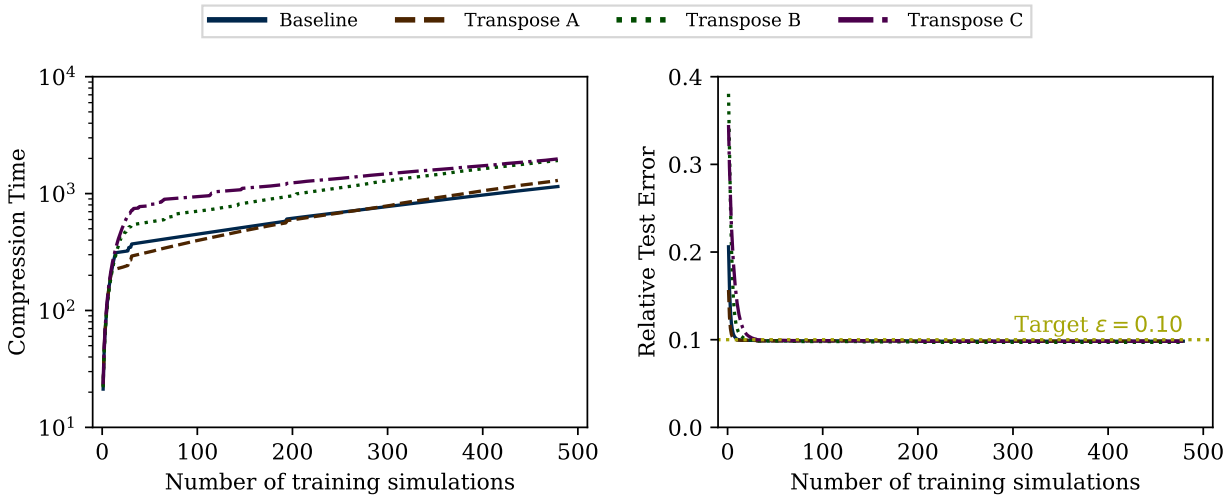


Figure B.4: Comparison of various axis orderings in terms of compression time (left) and relative test error (RTE - right) on the PDEBench 3D Navier-Stokes data set with $\varepsilon_{rel} = 0.10$ using no normalization. The baseline axis ordering results in the lowest compression time among the investigated axis reorderings. Changing the axis ordering of the tensor results in up to 1.72× the compression time of best performing axis ordering. All axis reorderings result in almost the same RTE.

APPENDIX C

Proofs for HT-RISE and BHT-l2r Algorithms

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

C.1 Proofs of correctness for 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.

Proof of Lemma 1 – BHT-l2r layerwise approximation error. This is a straight forward result following the application of [90, Lemma 3.8] to each node in $\mathbf{T}_\ell$. In Equation (3.42), 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 [90]. 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 $E_{\ell,i}$. □

Proof of Theorem 4 – Total approximation error [90, Lemma 3.10]. This is a direct generalization of Lemma 1 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. □

Proof of Corollary 2 – BHT-l2r approximation error. The proof follows from Theorem 4. If we set $\|E_{\ell,i}\| \leq \varepsilon_{nw}$ for all ℓ and i, then using (3.47) the expression for

the overall approximation error becomes

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

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 (3.47) and therefore completes the proof. □

Proof of Claim 1 – Orthogonal reconstructions. 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 $\mathcal{Y}^k = \mathcal{Y}_\parallel^k + \mathcal{Y}_\perp^k$, where $\mathcal{Y}_\parallel^k$ is the projection of $\mathcal{Y}^k$ onto the HT-cores $\mathcal{G}^{k-1}$ and $\mathcal{Y}_\perp^k$ is the orthogonal component of $\mathcal{Y}_\parallel^k$ with respect to the HT-cores of the accumulation $\mathcal{X}_{\mathbf{H}}^{k-1}$. Then, using the properties of the Frobenius inner product we can write $\langle \mathcal{Y}^k, \tilde{\mathcal{Y}}^k\rangle_F = \langle \mathcal{Y}_\parallel^k, \tilde{\mathcal{Y}}^k\rangle_F + \langle \mathcal{Y}_\perp^k, \tilde{\mathcal{Y}}^k\rangle_F$. Since $\tilde{\mathcal{Y}}^k$ is the reconstruction of $\bar{\mathcal{C}}_1^k$ 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 $\mathcal{Y}_\perp^k$ 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 $\|\tilde{\mathcal{Y}}^k\|_F^2 = \|\bar{\mathcal{C}}_1^k\|_F^2$. The reconstruction of $\tilde{\mathcal{Y}}^k$ from $\bar{\mathcal{C}}_1^k$ 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, $\tilde{\mathcal{C}}_2^k(:,:,:,:,i)$[1], can be shown as a reshaping of

$$\sum_{p,q=1}^{r_{1,1},r_{1,2}} \bar{\mathcal{C}}_1^k(p,q,i)\mathcal{G}_{1,1}^{k-1}(:,:,p) \otimes \mathcal{G}_{1,2}^{k-1}(:,:,q),$$

where $\otimes$ denotes the Kronecker product. Note that $\mathcal{G}_{\ell,j}^t(:,:,i)$ 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 $\|\tilde{\mathcal{C}}_2^k(:,:,:,:,i)\|_F^2 = \|\bar{\mathcal{C}}_1^k(p,q,i)\mathcal{G}_{1,1}^{k-1}(:,:,p) \otimes \mathcal{G}_{1,2}^{k-1}(:,:,q)\|_F^2 = \|\bar{\mathcal{C}}_1^k(p,q,i)\|_F^2$ and $\|\tilde{\mathcal{C}}_2^k\|_F^2 = \|\bar{\mathcal{C}}_1^k\|_F^2$.

Since the orthonormality of $\mathcal{G}_{\ell,j}^{k-1}(:,:,i)$ 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., $\|\tilde{\mathcal{Y}}^k\|_F^2 = \|\bar{\mathcal{C}}_1^k\|_F^2$. This completes the proof. □

Proof of Theorem 5 – Approximation error guarantees for HT-RISE. The proof uses Theorem 4 and is direct. Let $\bar{\mathcal{C}}_\ell^k$ (for any $\ell = p,\ldots,1$ and $\bar{\mathcal{C}}_p = \mathcal{Y}^k$) be the intermediate tensor obtained at the ℓ-th layer while processing the streamed tensor $\mathcal{Y}^k$ 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 (3.50) to (3.52), we can write $\bar{\mathcal{C}}_\ell^k$ as

$$\bar{C}_{\ell,(j)}^k = U_{\ell,j}^{k-1}(U_{\ell,j}^{k-1})^T\bar{C}_{\ell,(j)}^k + U_{R_{\ell,j}}^k\Sigma_{R_{\ell,j}}^k(V_{R_{\ell,j}}^k)^T + E_{R_{\ell,j}}, \tag{C.2}$$

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 (C.2) to resemble Lemma 1 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 Lemma 1 and accumulate the layerwise approximation upper bounds to obtain a total approximation error using Theorem 4. □

Proof of Theorem 6 – Error guarantees for the past stream for HT-RISE. 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 [7, 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

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

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 (3.53) and becomes

$$\bar{C}_1^{t,m} = \begin{bmatrix} \bar{C}_1^{k,m} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} \end{bmatrix}, \tag{C.3}$$

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 (C.3) 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

$$\tilde{\mathcal{C}}_1^{t,m}(i_1, i_2, i_3, i_4) = \sum_{\alpha=1}^{r_{1,1}^t} \sum_{\beta=1}^{r_{1,2}^t} \bar{C}_1^{t,m}(\alpha, \beta) \mathcal{G}_{1,1}^t(i_1, i_2, \alpha) \mathcal{G}_{1,2}^t(i_3, i_4, \beta). \tag{C.4}$$

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 (C.4) 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 (3.53), 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

$$\begin{aligned} \tilde{\mathcal{C}}_2^{t,m}(i_1, \ldots, i_8) &= \sum_{\alpha=1}^{r_{2,1}^t} \sum_{\beta=1}^{r_{2,2}^t} \sum_{\gamma=1}^{r_{2,3}^t} \sum_{\theta=1}^{r_{2,4}^t} \tilde{\mathcal{C}}_1^{t,m}(\alpha, \beta, \gamma, \theta) \mathcal{G}_{2,1}^t(i_1, i_2, \alpha) \cdots \mathcal{G}_{4,1}^t(i_7, i_8, \alpha), \\ &= \sum_{\alpha=1}^{r_{2,1}^k} \sum_{\beta=1}^{r_{2,2}^k} \sum_{\gamma=1}^{r_{2,3}^k} \sum_{\theta=1}^{r_{2,4}^k} \tilde{\mathcal{C}}_1^{k,m}(\alpha, \beta, \gamma, \theta) \mathcal{G}_{2,1}^t(i_1, i_2, \alpha) \cdots \mathcal{G}_{4,1}^t(i_7, i_8, \alpha). \end{aligned} \tag{C.5}$$

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

layer p, the reconstruction is performed as

$$\begin{aligned}\tilde{\mathcal{X}}_m^t = \tilde{\mathcal{C}}_p^{t,m}(i_1,\ldots,i_d) &= \sum_{\alpha=1}^{r_{p,1}^t}\cdots\sum_{\zeta=1}^{r_{p,d}^t}\tilde{\mathcal{C}}_{p-1}^{t,m}(\alpha,\ldots,\zeta)\mathcal{G}_{p,1}^t(i_1,\alpha)\cdots\mathcal{G}_{p,d}^t(i_d,\zeta),\\ &= \sum_{\alpha=1}^{r_{p,1}^k}\cdots\sum_{\zeta=1}^{r_{p,d}^k}\tilde{\mathcal{C}}_{p-1}^{k,m}(\alpha,\ldots,\zeta)\mathcal{G}_{p,1}^t(i_1,\alpha)\cdots\mathcal{G}_{p,d}^t(i_d,\zeta).\end{aligned} \tag{C.6}$$

Equations (C.4) to (C.6) 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 D

Auxiliary Subroutines and Additional Information on HT-RISE Analyses

This appendix provides additional information on the auxiliary subroutines used in the HT-RISE algorithm and an alternative approach to determine an upper bound to the approximation error for BHT-l2r and HT-RISE algorithms. The auxiliary subroutines are essential for maintaining the dimensional consistency of the hierarchical Tucker cores during the update process, while the alternative approach allows for a more adaptive error tolerance throughout the algorithms.

D.1 Auxiliary Subroutines for HT-RISE Algorithm

This section contains pseudocodes for the supplementary algorithms that are used in Algorithms 3 and 4.

- `updateIndexSet`: During the update process with HT-RISE (Algorithm 4), 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 6) 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. The `expandCore` (Algorithm 7) 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 8.

Algorithm 6 `updateIndexSet`: Updates the index set of a given object

Input
  $\mathbf{I}_{\Gamma_\alpha}$ index set of the object Γ
  Γ_α object to update the index set
Output
  $\mathbf{I}_{\Gamma_\alpha}$ updated index set
if Γ_α is a node **then** ▷ α is a tuple with layer and node indices
  $\ell, j \leftarrow \alpha$
  $\mathbf{I}_{\Gamma_\alpha} \leftarrow \begin{cases} \{n_{d_{\ell,j}}, r_{\ell,j}\} & \text{if } \Gamma_\alpha \text{ is a leaf} \\ \left\{ \bigcup_{(\ell+1,p)\in\mathbf{S}_{\ell,i}} r_{\ell+1,p} \right\} \cup \{r_{\ell,i}\} & \text{if } \Gamma_\alpha \text{ is a transfer node.} \end{cases}$
else if Γ is a layer **then** ▷ α is the layer index
  $\mathbf{I}_{\Gamma_\alpha} \leftarrow \bigcup_{j=1}^{|\Gamma_\alpha|} \begin{cases} n_{d_{\alpha,j}} & \text{if } \Gamma_{\alpha,j} \text{ is a leaf} \\ \prod_{(\alpha+1,m)\in\mathbf{S}_{\alpha,j}} r_{\alpha+1,m} & \text{if } \Gamma_{\alpha,j} \text{ is a transfer node.} \end{cases}$ ▷ $\Gamma_{\alpha,j}$ is the j-th node on layer α
end if

Algorithm 7 `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 $\mathcal{G}_{\ell,j}$
  $\mathbf{I}_{\mathbf{N}_{\ell,j}} \leftarrow \texttt{updateIndexSet}(\mathbf{I}_{\mathbf{N}_{\ell,j}}, \mathbf{N}_{\ell,j})$ ▷ Update the index set of the node to reflect the new rank
else if $\mathbf{N}_{\ell,j}$ is a transfer node **then**
  $\alpha \leftarrow \frac{1}{r_{\ell,j}} \prod_{\gamma\in\mathbf{I}_{\mathbf{N}_{\ell,j}}} \gamma$
  $\mathcal{G}_{\ell,j} \leftarrow \texttt{reshape}(\mathcal{G}_{\ell,j}, [\alpha, r_{\ell,j}])$ ▷ $\mathcal{G}_{\ell,j}$ is reshaped to an orthonormal matrix before appending U
  $\mathcal{G}_{\ell,j} \leftarrow \begin{bmatrix} \mathcal{G}_{\ell,j} & U \end{bmatrix}$
  $\mathbf{I}_{\mathbf{N}_{\ell,j}} \leftarrow \texttt{updateIndexSet}(\mathbf{I}_{\mathbf{N}_{\ell,j}}, \mathbf{N}_{\ell,j})$ ▷ Update the index set of the node to reflect the new rank
  $\mathcal{G}_{\ell,j} \leftarrow \texttt{reshape}(\mathcal{G}_{\ell,j}, I_{\mathbf{N}_{\ell,j}})$ ▷ Fold $\mathcal{G}_{\ell,j}$ back to 3D core
end if

Algorithm 8 `padWithZeros`: Pad a hierarchical Tucker core with zeros

Input
- $\mathcal{G}_{\ell,j} \in \mathbb{R}^{\alpha\times\beta\times r}$ — HT-core to be padded
- t — index of the node to be padded among the children of its parent node
- r' — size of the zeros to be padded

Output
- $\mathcal{G}_{\ell,j}$ — padded HT-core

$\mathbf{0} \in \mathbb{R}^{r'\times\beta\times r}$ ▷ Assume $t = 1$ for simplicity

$\mathcal{G}_{\ell,j} \leftarrow \mathcal{G}_{\ell,j} \oplus^t \mathbf{0}$ ▷ $\mathcal{G}_{\ell,j} \in \mathbb{R}^{(\alpha+r'\times\beta\times r)}$

D.2 Alternative approach to determine an upper bound to the approximation error for HT-RISE and BHT-l2r algorithms

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 ßHT-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

$$\varepsilon_\ell = \sqrt{\|\mathcal{Y}\|_F^2 - \|\bar{\mathcal{C}}_\ell\|_F^2}, \tag{D.1}$$

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

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

In addition to updating the remaining error budget, we also update the nodewise error

tolerance ε_{nw} accordingly as

$$\varepsilon_{nw} = \frac{\varepsilon_{rem}}{\sqrt{\texttt{svd}_{rem}}}, \tag{D.3}$$

with $\texttt{svd}_{rem}$ 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 Corollary 2 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.

D.3 Further information on datasets used

This section provides additional information on the image-based datasets used in the experiments. We provide additional details on the Basalt MineRL competition dataset and the BigEarthNet dataset.

D.3.1 Basalt MineRL Competition Dataset

The videos from the Basalt MineRL competition dataset are composed of 640×480 pixel RGB images. As part of the competition, there are 4 distinct tasks that the agents are trying to demonstrate. These tasks are:

- **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).

Each sequence begins with the agent spawning in the appropriate biome for the task. The agent is given a set of tools and items relevant to their tasks in their inventory. The agent is also given a set of instructions on how to complete the task and a maximum time frame within which they are expected to complete.

Recall that the data set contains approximately 20,000 gameplay episodes in total. This translates to approximately 160 GB of video demonstrations *per task*. Each video demonstration is accompanied with the relevant keystrokes that the agents take during the gameplay. However, we only use the RGB images in our compression experiments.

D.3.2 BigEarthNet Dataset

The images from the BigEarthNet dataset are multispectral images captured by Sentinel-2 satellites. Each image contains 12 spectral bands, which are listed in Table D.1. Recall that images have size 120×120 pixels.

Table D.1: 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

D.4 Additional results

This section contains the additional results for the experiments conducted in Section 3.1.3. In addition to those, this section further presents the comparison of the batch hierarchical Tucker format and the hierarchical Tucker format.

D.4.1 Additional PDEBench Results

This section presents the detailed results discussed in Section 3.3.2.2. Similar to Table 3.3, we present experiments on the PDEBench 3D turbulent Navier-Stokes simulations with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.05$. In contrast to Section 3.3.2.2, here we provide results for all normalization methods used in the experiments. Figures D.1 and D.2 show the results for experiments with $\varepsilon_{rel} = 0.10$ and Figures D.3 and D.4 show the results for experiments with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.05$.

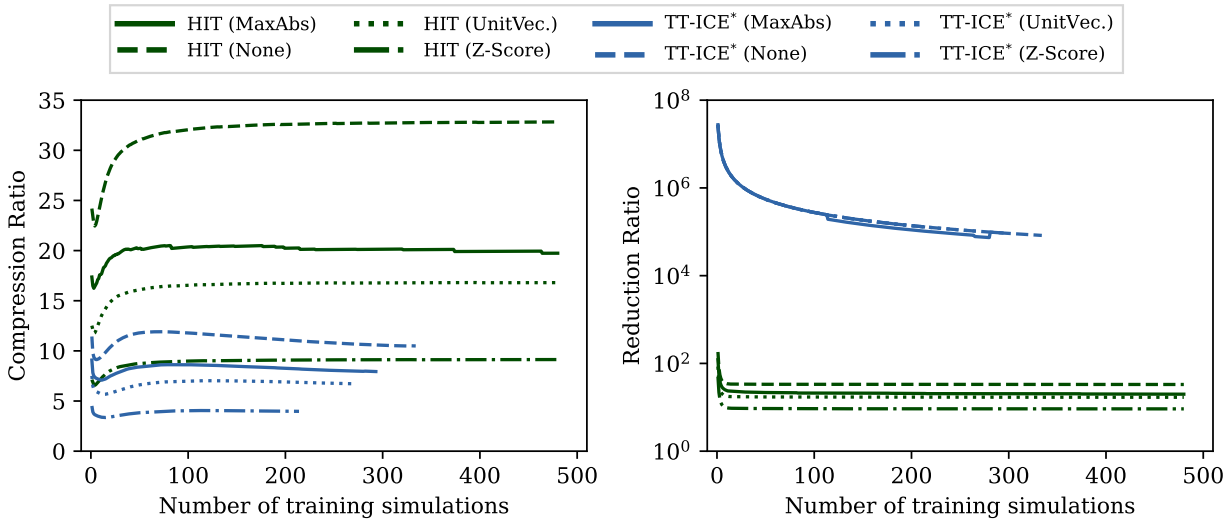


Figure D.1: Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes dataset with $\varepsilon_{rel} = 0.10$ and using various normalization methods. `TT-ICE`* offers orders of magnitude higher reduction ratio while yielding less compression 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 3.17, Figure D.1 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.2. 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.

A similar trend is observed in Figure D.3 and Figure D.4 for the experiments with $\varepsilon_{rel} = 0.05$. `TT-ICE`* offers orders of magnitude higher reduction ratio while yielding less

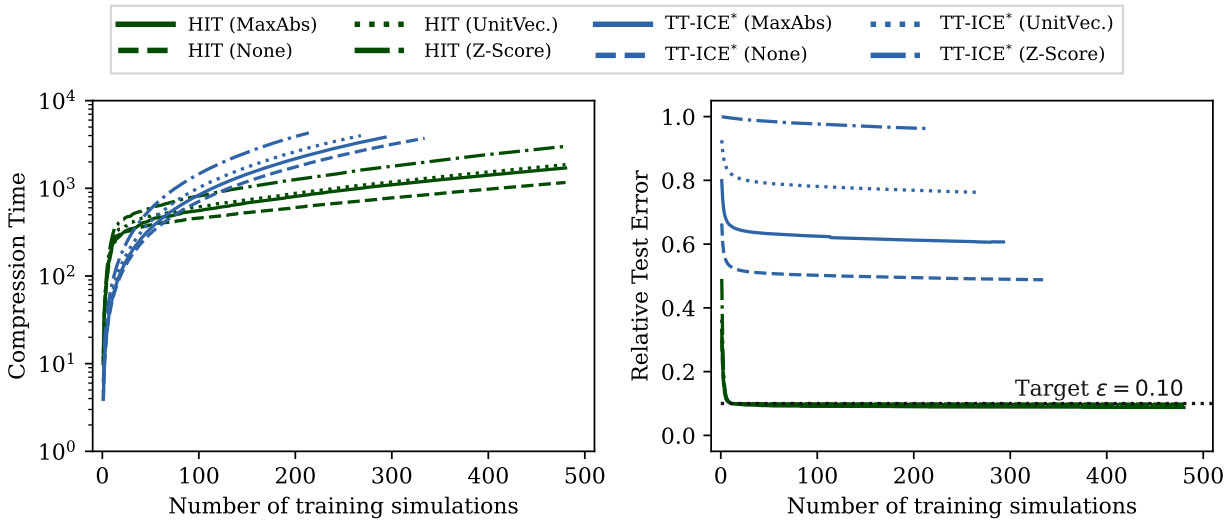


Figure D.2: Comparison of compression time and mean validation error of algorithms using the PDEBench 3D Navier-Stokes dataset with $\varepsilon_{rel} = 0.10$ and using various normalization methods. TT-ICE* fails to reduce the approximation error on the test set whereas HT-RISE achieves below ε_{rel} 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.1 for the normalization methods. The results are averaged over 5 seeds.

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.4.2 Additional Self-Oscillating-Gel Simulations Results

This section presents the detailed results discussed in Section 3.3.2.1. Similar to Table 3.2, 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 normalization in addition to experiments without any normalization. Figures D.5 and D.6 show the results for experiments with $\varepsilon_{rel} = 0.10$ and Figures D.7 and D.8 show the results for experiments with $\varepsilon_{rel} = 0.01$.

Figure D.5 shows the compression ratio and reduction ratio of the algorithms on the self-oscillating gel data set 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

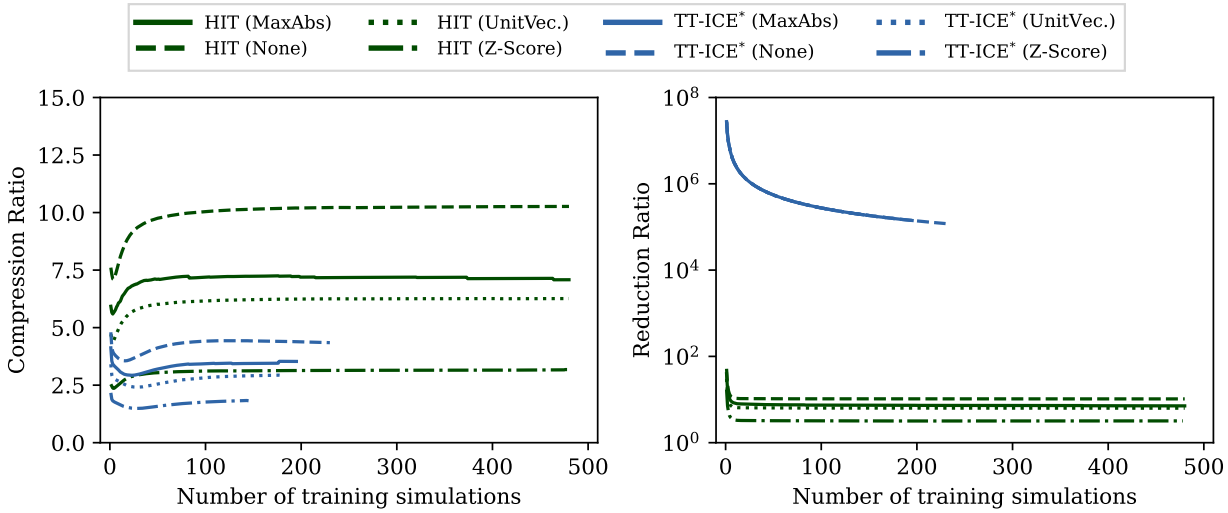


Figure D.3: Compression time (left) and Relative Test Error (right) of the algorithms on the PDEBench 3D turbulent Navier-Stokes dataset with $\varepsilon_{rel} = 0.05$ and using various normalization methods. `TT-ICE`* offers orders of magnitude higher reduction ratio while yielding less compression 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.1 for the normalization methods. The results are averaged over 5 seeds.

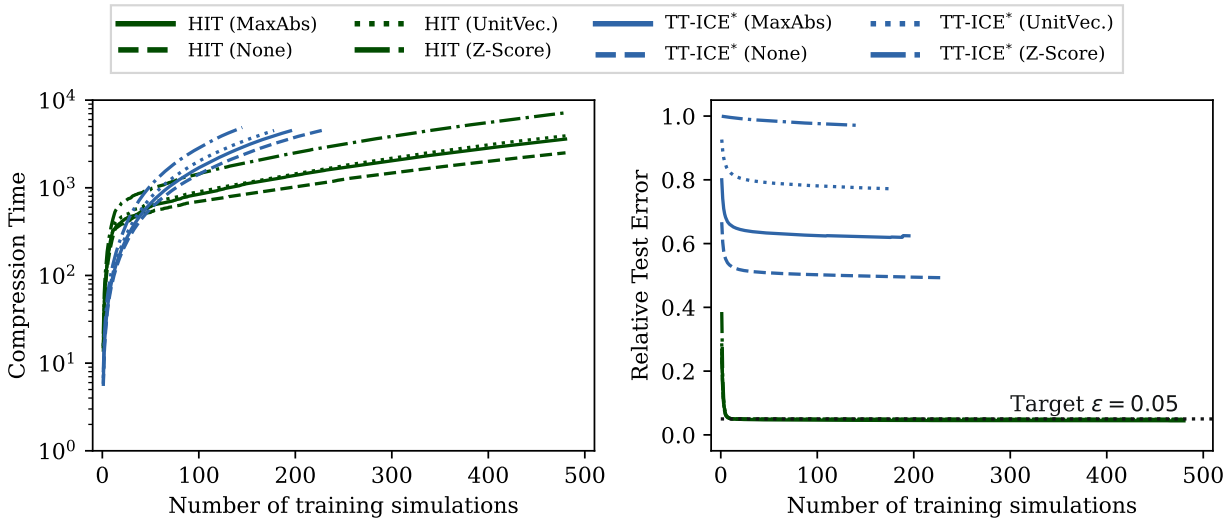


Figure D.4: Comparison of compression time (left) and Relative Test Error (right) of algorithms using the PDEBench 3D Navier-Stokes dataset with $\varepsilon_{rel} = 0.05$ and using various normalization methods. TT-ICE* fails to reduce the approximation error on the test set whereas HT-RISE achieves 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.1 for the normalization methods. The results are averaged over 5 seeds.

HT-RISE.

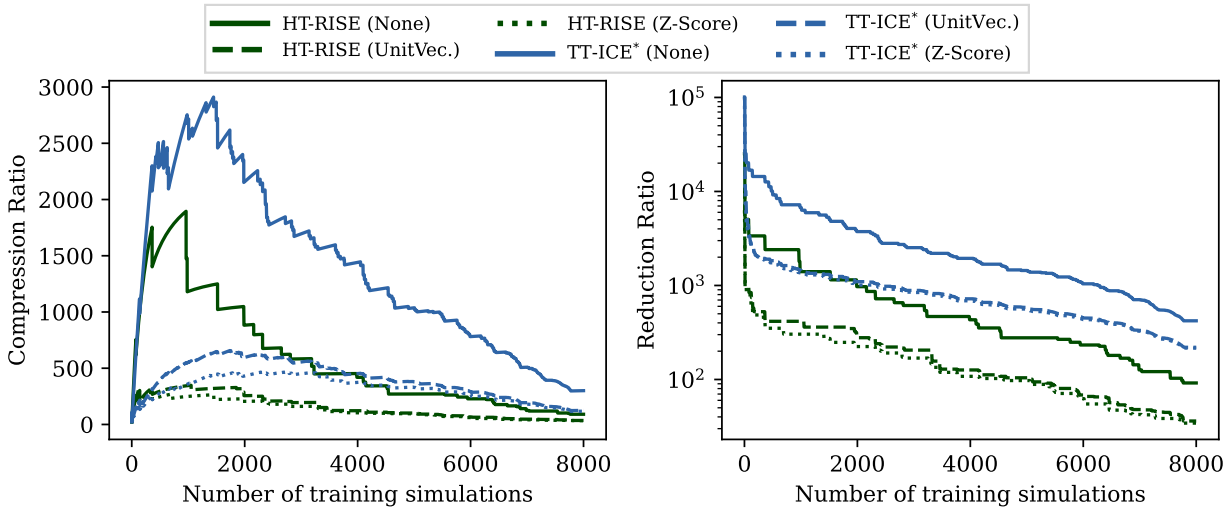


Figure D.5: Compression ratio (CR - left) and Reduction ratio (RR - right) of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.10$ and using various normalization methods. `TT-ICE`* offers orders of magnitude higher reduction ratio while yielding less compression ratio compared 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.6 shows the compression time and relative test error of the algorithms on the self-oscillating gel data set 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 data set.

Figure D.7 shows the compression ratio and reduction ratio of the algorithms on the self-oscillating gel data set 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.8 shows the compression time and relative test error of the algorithms on the self-oscillating gel data set 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 when no normalization is employed. In addition to that, HT-RISE was able to cross the

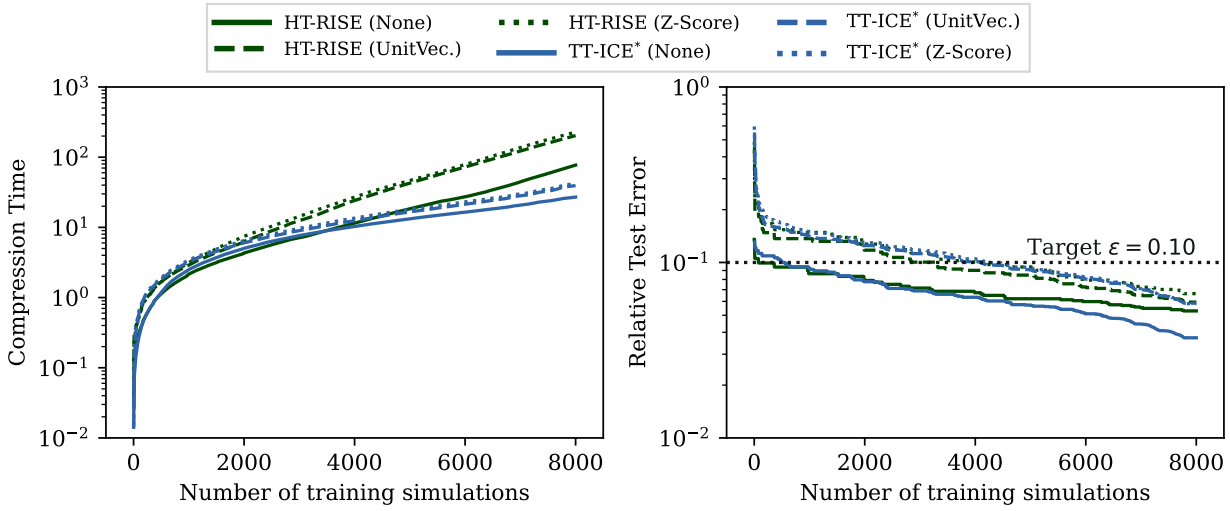


Figure D.6: Compression time (left) and Relative Test Error (right) of the algorithms on the self-oscillating gel data set with $\varepsilon_{rel} = 0.10$ and using various normalization methods. TT-ICE* offers orders of magnitude higher reduction ratio while yielding less compression ratio compared to HT-RISE. All experiments with `TT-ICE*` terminates prematurely due to timeout. Please refer to the caption of Figure D.5 for normalization methods. The results are averaged over 5 seeds.

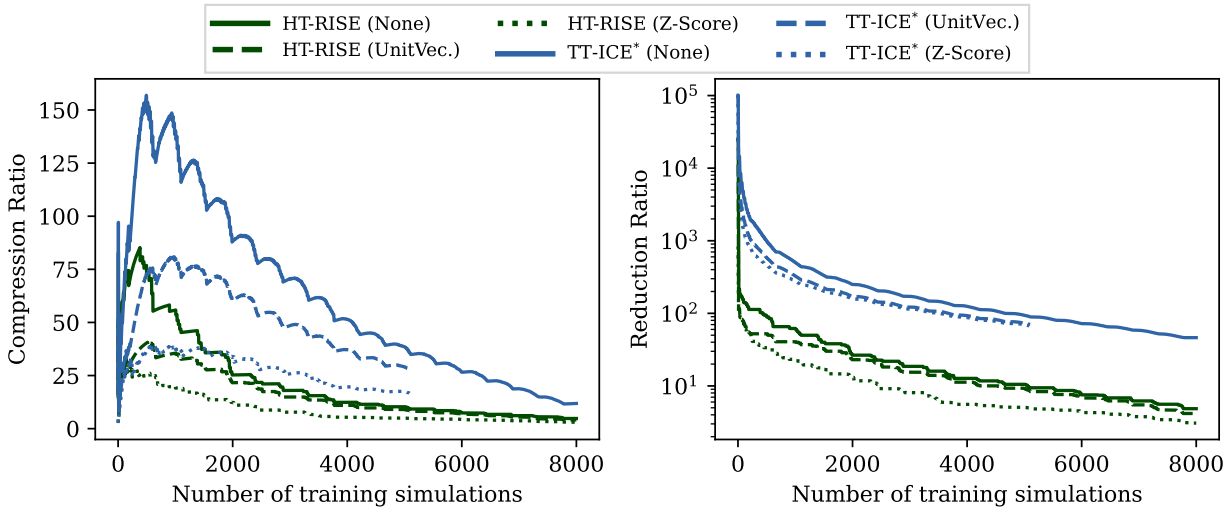


Figure D.7: Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the self-oscillating gel dataset with $\varepsilon_{rel} = 0.01$ and using various normalization methods. None: No normalization, UnitVec: Unit vector normalization, ZScore: Z-score normalization. TT-ICE* offers an order of magnitude higher reduction ratio while yielding comparable compression ratio compared to HT-RISE.

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 data set due to the maximum walltime limit at two of the normalization methods. However, it is worth noting that in Figure D.8 `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.

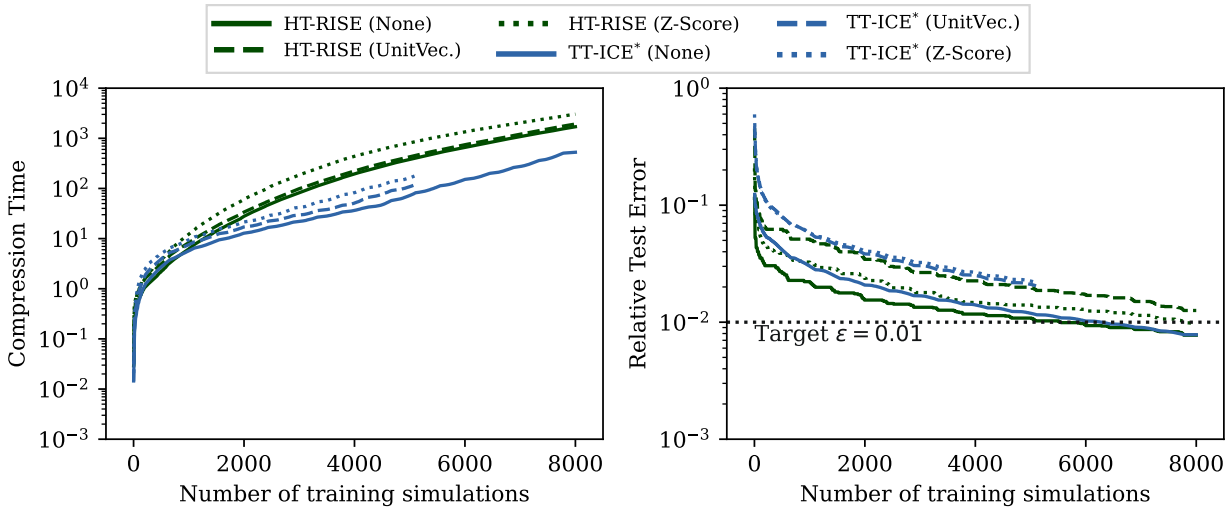


Figure D.8: Comparison of compression time (left) and Relative test error (right) of algorithms using the Self-oscillating gel simulation dataset with $\varepsilon_{rel} = 0.01$ and using various normalization methods. Both methods were able to reduce the approximation error below the target ε_{rel} 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.7 for the normalization methods.

D.4.3 Additional BigEarthNet Results

This section presents the detailed results discussed in Section 3.3.3.2. Similar to Table 3.5, we present experiments on the BigEarthNet data set with $\varepsilon_{rel} = 0.05 - 0.30$. Figure D.9 presents the results of the experiments on compression ratio and reduction ratio of the algorithms whereas Figure D.10 shows the results of the experiments considering compression time and relative test error of the algorithms.

Figure D.9 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

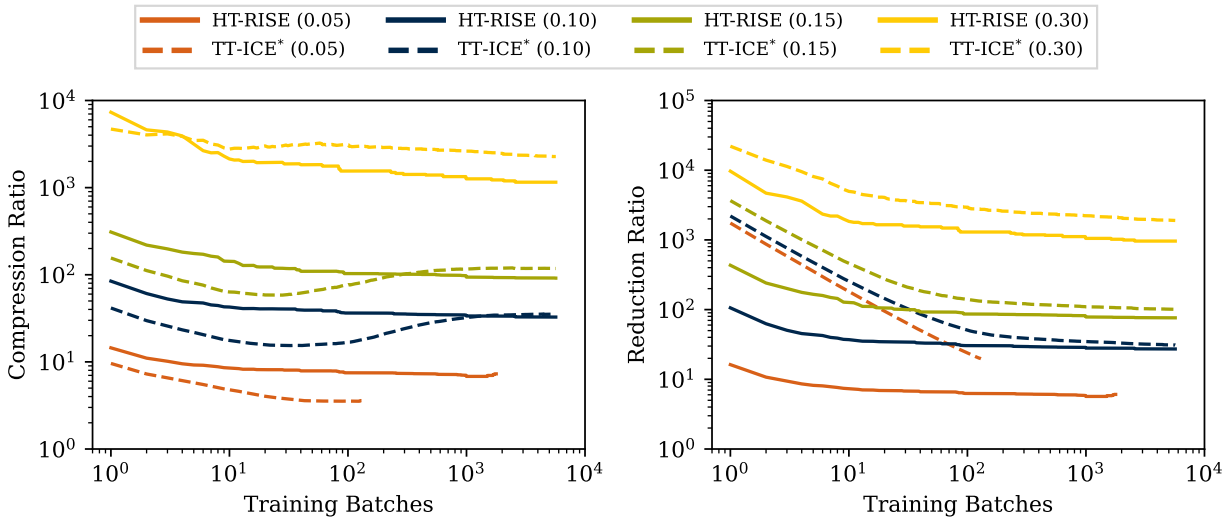


Figure D.9: Compression ratio (CR - left) and reduction ratio (RR - right) of the algorithms on the BigEarthNet dataset with $\varepsilon_{rel} = 0.05 - 0.30$. `TT-ICE*` offers higher CR ($1.1 - 2\times$ that of HT-RISE) for all target relative error levels but $\varepsilon_{rel} = 0.05$. At $\varepsilon_{rel} = 0.05$, 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.

and $100\times$ RR whereas HT-RISE achieves $91\times$ CR and $76\times$ RR. At $\varepsilon_{rel} = 0.10$, the performance of the two algorithms become even more comparable, where `TT-ICE*` achieves $35\times$ CR and $31\times$ RR whereas HT-RISE achieves $32\times$ CR and $27\times$ RR. At these target relative error levels, neither algorithm stuggles to compress the data set. However at $\varepsilon_{rel} = 0.05$, `TT-ICE*` runs into maximum walltime timeout and therefore fails to compress the entire data set and HT-RISE runs into maximum memory limit and therefore fails to compress the entire data set. The portion of the data set 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 data set that `TT-ICE*` is able to compress is significantly smaller than the portion of the data set that HT-RISE is able to compress.

Figure D.10 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 data set in 2482s whereas `TT-ICE*` compresses the entire data set in 6239s. At $\varepsilon_{rel} = 0.15$ the difference becomes less pronounced as HT-RISE compresses the entire data set in $14,896$s whereas `TT-ICE*` compresses the entire data set in $18,866$s. At $\varepsilon_{rel} = 0.10$ `TT-ICE*` becomes slightly faster as HT-RISE compresses the entire data set in $49,408$s and `TT-ICE*` compresses the entire data set in $44,687$s. As discussed above, at $\varepsilon_{rel} = 0.05$ neither algorithm is able to compress the entire data set. At $\varepsilon_{rel} = 0.05$, `TT-ICE*` compresses 129 batches in 1720s whereas in the same amount of time HT-RISE compresses

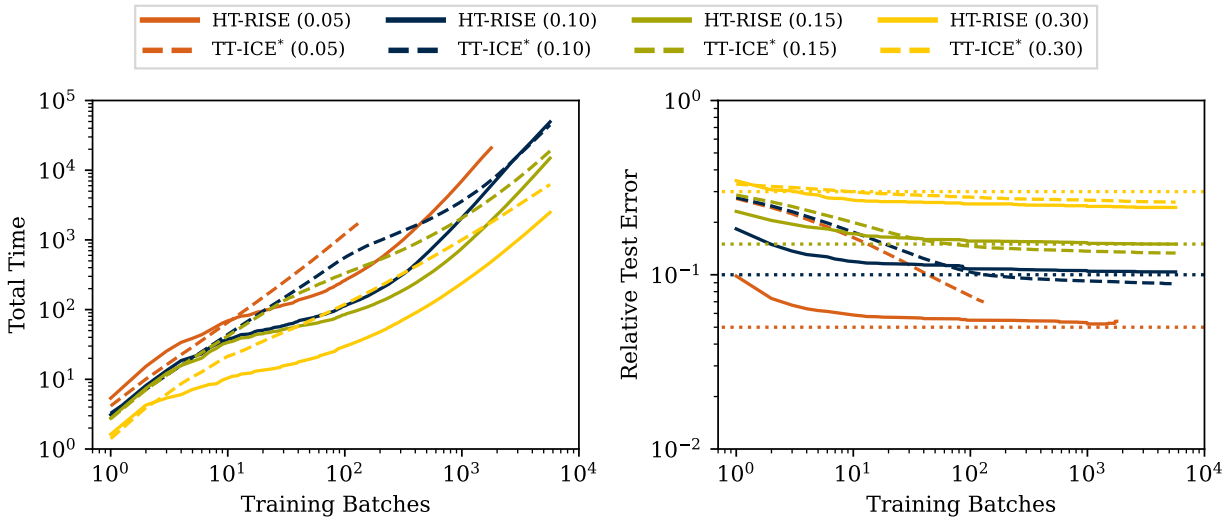


Figure D.10: Compression time (left) and relative test error (RTE - right) of the algorithms on the BigEarthNet dataset with $\varepsilon_{rel} = 0.05 - 0.30$. For all ε_{rel} levels, HT-RISE results in lower, if not comparable, compression time compared to `TT-ICE`*. For all ε_{rel} levels, HT-RISE 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.

443 batches. At the point of failure, HT-RISE compresses 1781 batches in $20,880$s.

D.4.4 Qualitative MineRL Results

This section presents a qualitative comparison of the compressed frames using reconstructions. Figure D.11 shows the reconstructed video frames from the Basalt MineRL competition data set using `TT-ICE`* and HT-RISE algorithms with $\varepsilon_{rel} = 0.10$ and $\varepsilon_{rel} = 0.30$. In Figure D.11, 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 reconstructed 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 $1.85\times$.



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

Figure D.11: Reconstructed video frames from Basalt MineRL competition data set 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 3.4, and Figures 3.20 and 3.21 for quantitative results.

APPENDIX E

Behavioral Cloning Framework

This appendix provides additional details on the behavioral cloning framework presented in Section 5.2, including architectural details of the autoencoders and variational autoencoders used, additional comparisons of behavioral cloning performance, and further analysis of the effect of changing the size of the latent space.

E.1 Additional Details on Training

Architectural details for the encoding part of our AEs and VAEs are laid out in Table E.1. The decoding portion of these networks mirrors the encoding portion, with convolutional layers replaced by transposed convolutional layers.

Table E.1: Architecture for the encoding part of AEs and VAEs. After each intermediate layer we also apply batch norm followed by LeakyReLU - these are excluded from the table for simplicity. The asymmetric padding in the first layer is to account for the shape of the input.

Layer #	Layer Type	Inp. Ch.	Out. Ch.	Kernel Size	Stride	Padding
1	Conv	3	32	3	2	(4, 1)
2	Conv	32	64	3	2	(1, 1)
3	Conv	64	64	3	2	(1, 1)
4	Conv	64	16	1	1	(0, 0)
5	Flatten	-	-	-	-	-
6	Linear	-	-	-	-	-

The AEs and VAEs are hyperparameter tuned individually for each game over initial learning rates of $10^{-2}, 10^{-3}$, and 10^{-4} and L2 weight decay penalties of $1, 10^{-2}$, and 10^{-4}, after which the best model is selected based on validation error and used for encoding. The networks are trained using the Adam-W optimizer [165] with cosine learning rate decay without restarts [164] for fifty epochs in the low data case and five epochs in the moderate

data case. The AE is trained to minimize reconstruction error measured via mean squared error while the VAE is trained using the standard ELBO loss [134]. An additional 1 or 20 trajectories (in the limited and moderate data regimes, respectively) are used as a validation set in order to hyperparameter tune the networks. Note that the TT does not require this step of validation and does not use this additional data.

E.2 Additional Comparisons of Behavioral Cloning Performance

E.2.1 Breakdown of Speedups

In Figures E.1 and E.2, we further break down the timings for training the AE, VAE, and TT-based predictors as follows:

- ***Training the encoder:*** For the AE and VAE, this time is the total time required to perform a hyperparameter optimization by training nine AEs / VAEs with different network hyperparameters and selecting one based on the validation loss measured on a separate validation set. Recall that the TT encoder does not require such a procedure. For the TT, this is the time required to perform the TT decomposition (using `TT-ICE`*) to extract the latent space representation of the input data.
- ***Encoding data:*** Time to pass all the training and validation data through the AE, VAE, and TT in order to extract latent space representations.
- ***Training the action predictor:*** Time to train the action predictor on the encodings extracted in the previous step. The architecture of the predictor is same across all three methods. For each model and for each game, nine predictors are trained using different initial learning rates (10^{-2}, 10^{-3}, and 10^{-4}) and L2 regularization penalties (1, 10^{-2}, and 10^{-4}). The predictor yielding the lowest cross-entropy action prediction loss measured on another separate validation set is selected.

Note that the most time-intensive task is training the encoding. As such, reducing the time it takes to train the encoding by using a TT instead of an AE or VAE leads to a significant reduction in overall training time.

E.2.2 Effect of Changing Epsilon

We conduct experiments testing the effect of changing the size of the latent space for the TT, AE, and VAE. In our experiments, the latent size of the AE and VAE is set up to match

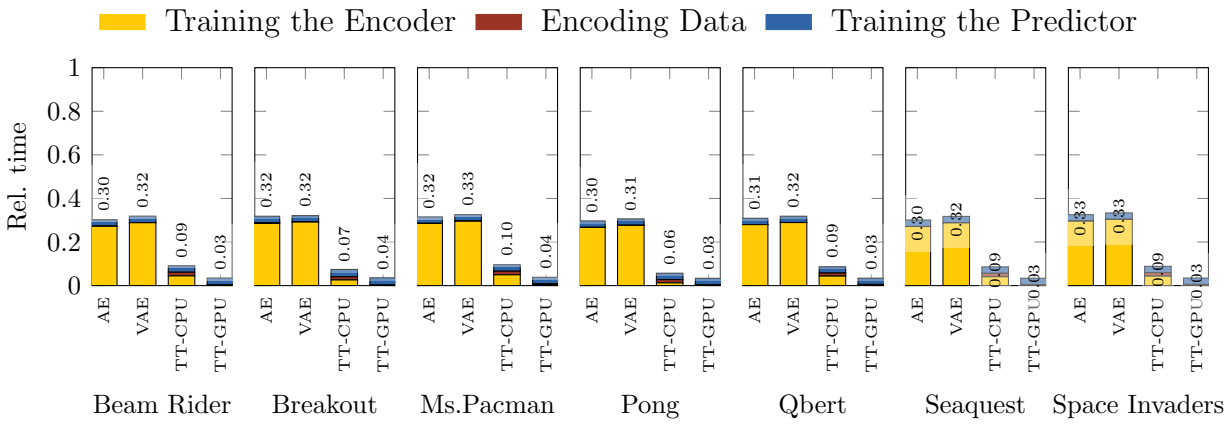


Figure E.1: Time for end-to-end training procedure in the limited data regime, averaged across all experiments, as a fraction of the time required to train the full-frame predictor. Extracting a latent space before training a predictor reduces end-to-end training time in all cases, but using the TT-GPU leads to a 26.1× to 29.7× reduction in training time. Compared to the AE / VAE, using the TT-GPU leads to a 8.2× to 9.6× speedup.

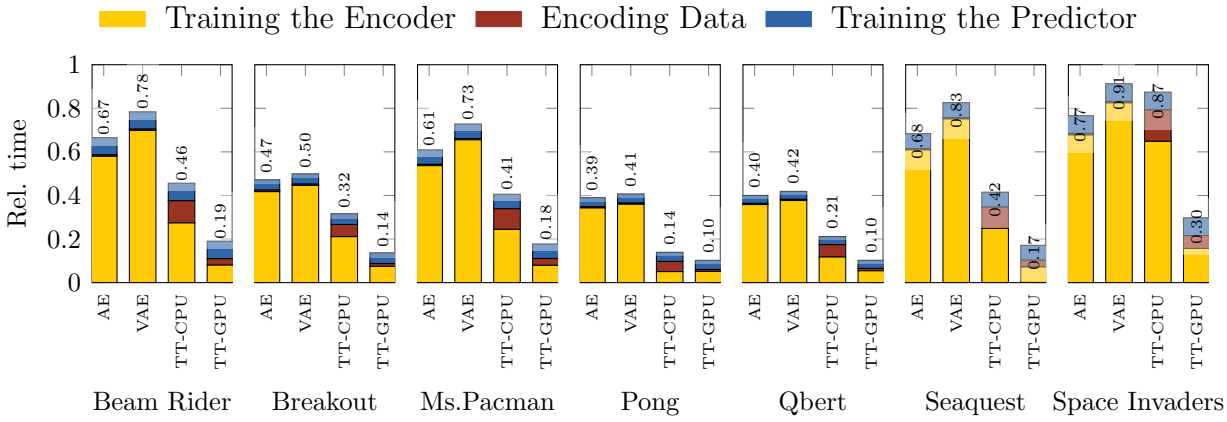


Figure E.2: Time for the end-to-end training procedure in the moderate data regime, averaged across all experiments, as a fraction of the time it takes to train a predictor using the full RGB frames. Extracting a latent space reduces end-to-end training time in all cases, but using TT-GPU leads to a 3.4× to 9.8× reduction in training time compared to no encoding and a 2.6× to 4.8× speedup relative to the AE / VAE.

the latent size of the TT. This experiment is conducted using 160 trajectories as training data, with five training epochs for the AE and VAE. In Table E.2, the latent size as epsilon is changed is displayed.

In Figure E.3, the effect of changing epsilon / the latent size on prediction accuracy is displayed. The general pattern appears to be that decreasing epsilon / increasing the latent size leads to a higher prediction accuracy, as expected. However, for some games, increasing the latent space size past a certain point leads to small or negligible improvements

Table E.2: Latent size as epsilon for the TT is changed. The reported numbers are an average across three seeds (which use different training data).

Epsilon	0.01	0.02	0.05	0.10	0.15
BeamRider	14645	9867	3324	297	72
Breakout	3561	957	50	18	3
MsPacman	11962	6421	867	15	2
Pong	1287	138	4	1	1
Qbert	2267	1560	308	43	10
Seaquest	15163	8170	1338	37	1
SpaceInvaders	17191	13795	8035	3994	2097

in prediction accuracy. The effect of changing the latent space size on timings is shown in Figure E.4. Using a larger latent space size tends to increase the end-to-end training time of the models. These results are consistent with our findings in the main text - at the different epsilon levels tested, the TT-based predictor yielded similar prediction accuracies as the AE and VAE, while being faster to train.

E.2.3 Gameplay Performance of RL Agent

This section provides some statistics on the RL agent used to generate the demonstration data.

Table E.3 provides the gameplay scores achieved by the RL agent compared to an agent that acts randomly by uniformly selecting amongst available actions and a human player. The human player scores are taken from [182]. The RL agent achieves higher scores than an agent acting randomly and achieves human-level performance in 2 of the 7 games. Table E.4 provides the gameplay lengths (number of steps between the beginning of a game to the end) achieved by the RL agent versus an agent that acts randomly. Note that gameplay lengths were not available from [182]. This table indicates that the RL agent has learned behavior that differs from a randomly acting agent.

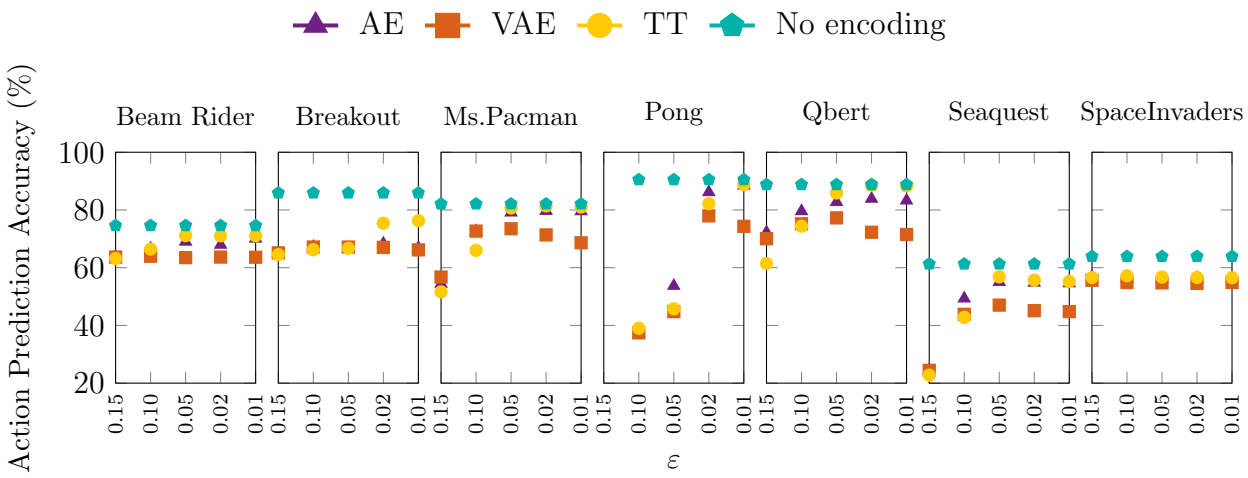


Figure E.3: Action prediction accuracy of behavioral cloning frameworks using different encoding methods at varying ε levels. As the latent space size is increased for the encoding-based predictors, the action prediction accuracy of the predictors tends to increase. The speedup offered by TT does not cause any performance deterioration. Furthermore, the nonlinearity offered by AE/VAE does not result in an improved latent space representation that achieves higher action prediction accuracy with a smaller latent space. Since Pong with $\varepsilon = 0.15$ and $\varepsilon = 0.10$ returns the same latent size, the accuracy for $\varepsilon = 0.15$ is omitted. Please refer to Table E.2 for latent space sizes corresponding to each ε level for each game.

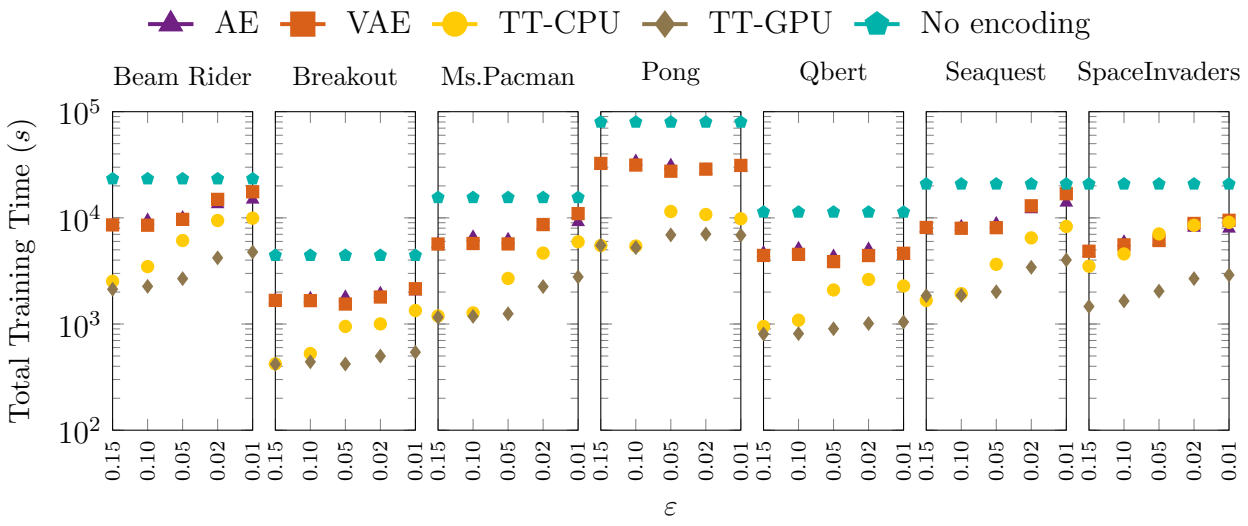


Figure E.4: Total time elapsed to train behavioral cloning frameworks using different encoding methods at varying ε levels. As the latent space size increases, the total training time for the various encoding-based predictors increases. Across all games and ε levels, TT-GPU encoding offers the fastest end-to-end training time. Please refer to Table E.2 for latent space sizes corresponding to each ε level for each different game.

Table E.3: Comparison of the gameplay scores achieved by the RL agent compared to an agent that acts randomly and a human player. The human gameplay scores are taken from [182].

	Random	RL	Human (Mnih et al., 2015)
BeamRider	366.1	1012.3	5775.0
Breakout	1.4	64.0	31.8
MsPacman	244.8	1883.4	15693.0
Pong	-20.3	21.0	9.3
Qbert	167.5	5181.0	13455.0
Seaquest	67.2	1962.0	20182.0

Table E.4: Comparison of the gameplay lengths (number of time steps from the beginning of the game to the end) achieved by the RL agent compared to an agent that moves randomly. Note that gameplay lengths for a human player were not available from [182].

	Random	RL
BeamRider	1289.5	1714.3
Breakout	153.2	982.1
MsPacman	489.7	1025.0
Pong	926.8	1570.7
Qbert	308.8	1390.0
Seaquest	492.2	2414.9
SpaceInvaders	488.7	660.5

APPENDIX F

Additional Results on Particle in Cell Simulation Compression

This appendix contains additional results on the compression of the particle-in-cell simulation snapshots from the TURF simulations.

F.1 Effect of normalization method and target relative error

The normalization methods and target relative error are an important design parameter in the compression pipeline for high-quality data reconstructions. Some of the states, such as particle number density, have large magnitude ranges over the computational domain, and the normalization method helps to reduce the range of these states when computing incremental updates to the TT-approximation. In this section we explore their effects on neutral number density contours. For many of the normalization methods and relative target errors tested, the reconstructed ion number contours resulted in satisfactory qualitative accuracy. The neutral number contours are more difficult to approximate due to wide range of density magnitudes in the domain and less diffuse plume. To see more discernible variations in reconstructions and for a more nuanced discussion on the effects of the compression pipeline parameters, we highlight the neutral number density contours.

As seen in Figure F.1, the square root normalization with target error $\varepsilon = 10^{-1}$ leads to an extreme CR of $53,384\times$, but the higher target relative error results in a less accurate overall TT-ICE approximation, as expected, and numerical artifacts appear. This is in contrast to Figure 4.5 which has lower target relative error of $\varepsilon = 10^{-2}$. The TT-approximation overestimates the number density in the backflow region. In Figure F.2, the logarithmic normalization with base 10 $\varepsilon = 10^{-1}$ introduces statistical noise in the plume.

Table F.1: Compression metrics for number density states using various normalization methods and target relative errors ε.

	Size	Time (s)	ε	CR	TT-ranks	Latent size
Original	5.5 GB	-	-	-	-	-
square root	133 KB	29.697	10^{-1}	53,384	1, 6, 25, 23, 26, 9, 7, 4, 2, 1	2
logarithmic base-10	333 MB	102.156	10^{-1}	16.53	1, 6, 59, 564, 2623, 1299, 254, 49, 25, 1	25
logarithmic-logarithmic	3.9 GB	454.1	10^{-2}	1.38	1, 6, 60, 600, 3000, 11767, 2500, 450, 150, 1	150
unitvector	1.1 MB	36.83	10^{-2}	5287.30	1, 6, 40, 133, 90, 21, 10, 5, 2, 1	2

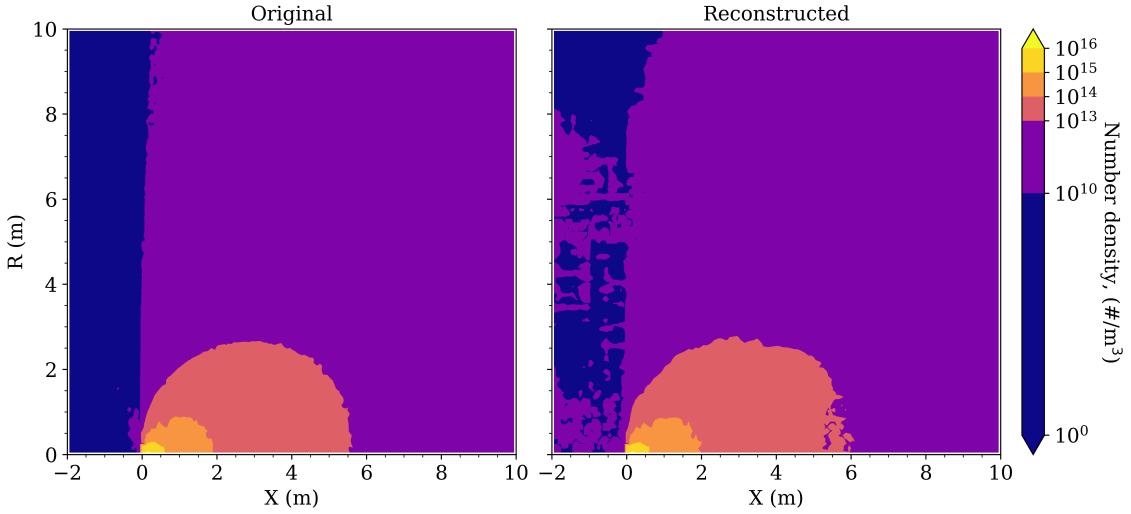


Figure F.1: (Right) Neutral number density contours are shown for reconstructions using a square root normalization, $\varepsilon = 10^{-1}$ target relative error, and where particle number density fields are compressed together. Increasing the input target relative error in results in extreme compression with a CR $53,384\times$, however, decreases contour resolution and fails to capture the correct backflow phenomena when compared to the reconstruction seen in Figure 4.5. Here, the TT approximation overestimates the number density in the backflow region.

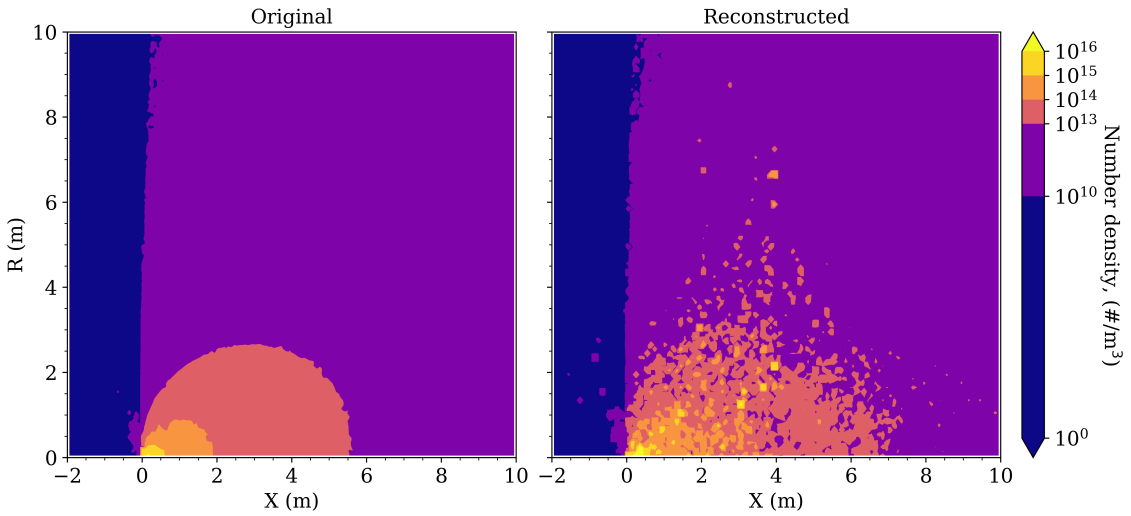


Figure F.2: (Right) Neutral number density contours are shown for reconstructions using a logarithmic normalization with base 10, $\varepsilon = 10^{-1}$ target relative error, and where particle number density fields are compressed together. Although the CR is 16× here, the reconstructed dataset has decreased resolution and more statistical noise in the plume contours for X $<$ 4 m when compared to Figure 4.5. This is likely due to the normalization method used; the base-10 logarithm shrinks the range of values aggressively and makes the reconstruction sensitive to numerical errors.

As seen in Figure F.3, applying the same normalization method twice does not necessarily provide accurate contour reconstructions. The second application of the normalization maps the span of the number density to a very small range and makes the reconstruction extremely sensitive to the changes in the reconstruction prior to normalization. Another possible explanation may be how the .vts files are read into NumPy arrays. Since the reconstruction quality of the plume degrades after X$>$ 4m, a highly likely explanation to this is that the subdomains are directly written into a separate dimension of a reshaped NumPy array, rather than being written into their correct spatial locations.

On the other hand, some normalization methods result in the compression favoring regions with higher number densities over sparsely populated regions while computing a low-rank approximation. As seen in Figure F.4, unit vector normalization with a target error of $\varepsilon = 10^{-2}$ fails to capture the backflow region behavior. Nevertheless, since this combination achieves extreme compression of 5287.3×, the accuracy around the plume might be useful if only the plume is of interest.

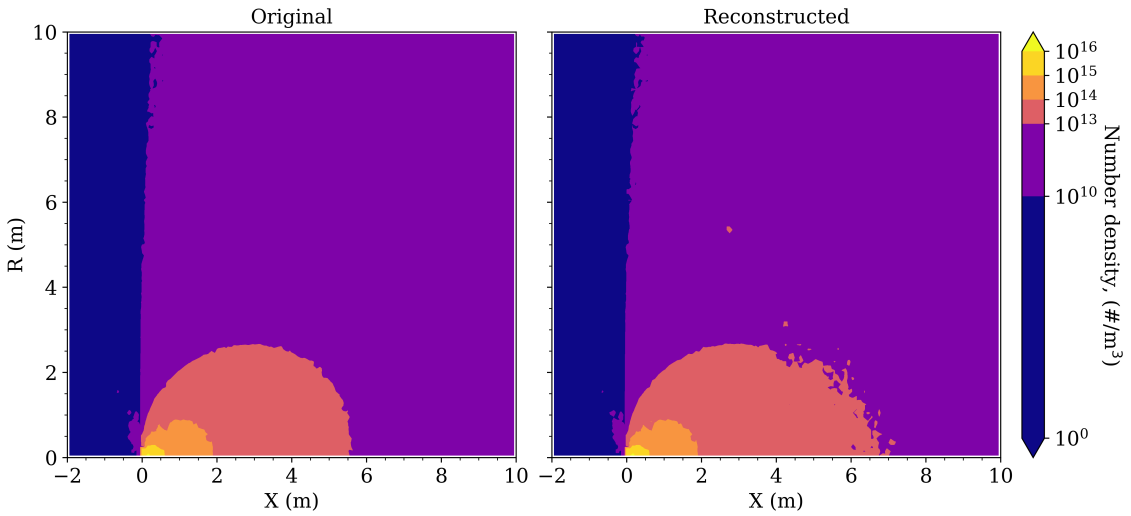


Figure F.3: (Right) Neutral number density contours are shown for reconstructions using a logarithmic-logarithmic normalization, $\varepsilon = 10^{-2}$ relative error, and where neutral and ion number density fields are compressed together. Cascading the same normalization method two times resulted in accurate contour modeling for X < 4 m but decreased resolution in the 10^{14} contour. CR $= 1.38\times$.

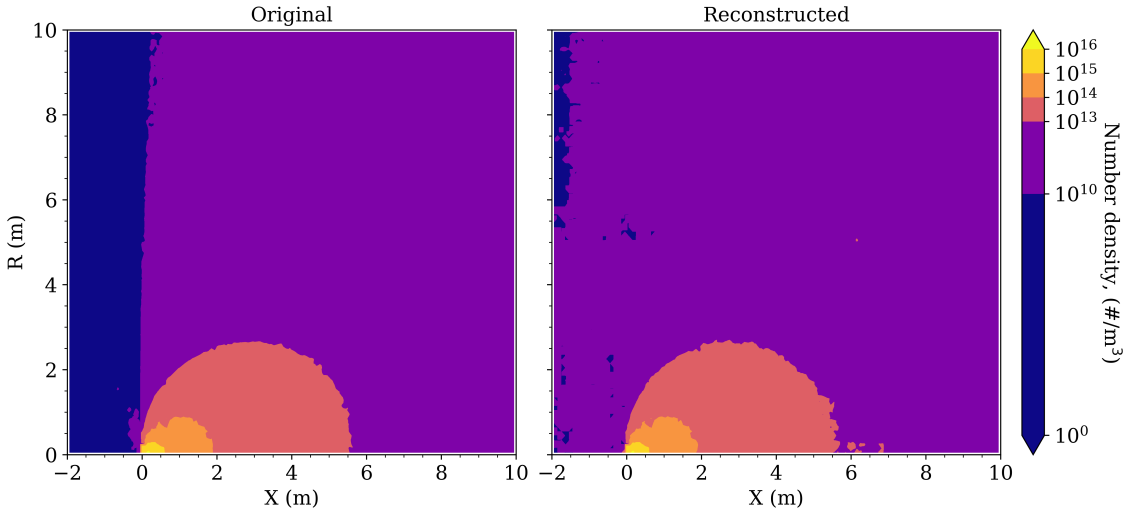


Figure F.4: (Right) Neutral number density contours are shown for reconstructions using a unit vector normalization, $\varepsilon = 10^{-2}$ target relative error, and where particle number density fields are compressed together. Although we have a CR of $5287\times$, we see an inaccurate approximation of the backflow region.

F.2 Effect of individual state compression

Each of the volume mesh states from the TURF simulation used for analysis (i.e. neutral number density, three types of ion number densities, ion current density, electric field components per cell) can be compressed separately or together. A separate compression pipeline for each state allows for flexibility in choosing a normalization method that exploits the state's data structures effectively and potentially yield a higher CR and/or a less statistically noisy approximation. As in Appendix F.1, in this section, we present mainly neutral number density contours to see more discernible variations in reconstructions to observe the relationship between TT-ranks and quality of reconstructions.

The neutral number density contours in Figure F.5, are reconstructed using a z-score normalization with a relative error of $\varepsilon = 10^{-2}$, with the neutral number and ion number density fields compressed together resulting in a CR of 5213×. However, the reconstruction suffers from reduced accuracy, especially in the far-plume region.

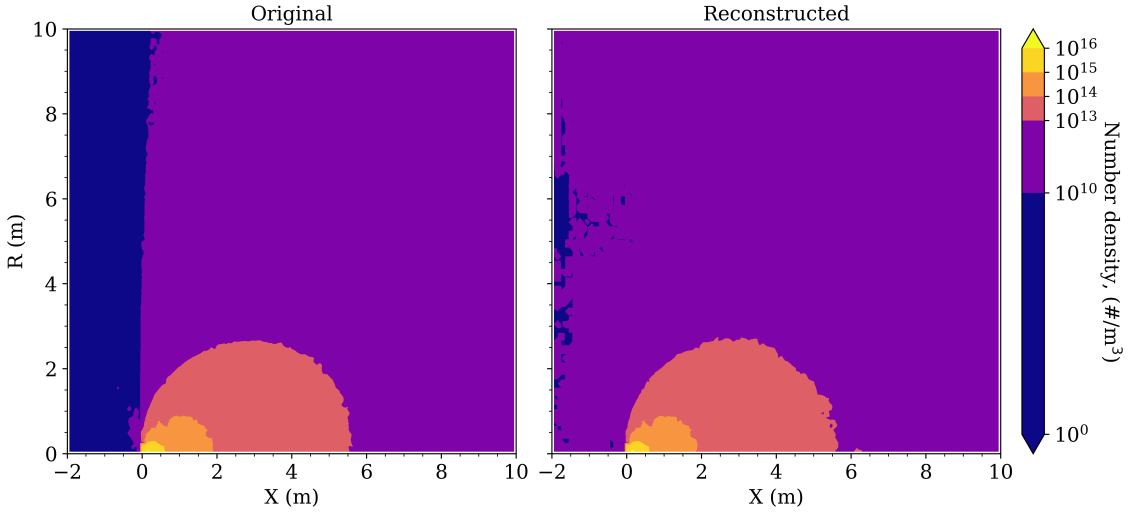


Figure F.5: (Right) Neutral number density contours are shown for reconstructions using a z-score normalization, $\varepsilon = 10^{-2}$ target relative error, and where neutral and ion number density fields are compressed together. This results in an approximation with a CR of 5213×.

When neutral atoms are compressed separately from the ions using the same compression parameters, there is a significant increase in the CR to $24,563\times$. As seen in Figure F.6, the resolution in the plume is not degraded significantly. For scenarios where the priority is respecting data storage limitations, this compression schema becomes highly attractive.

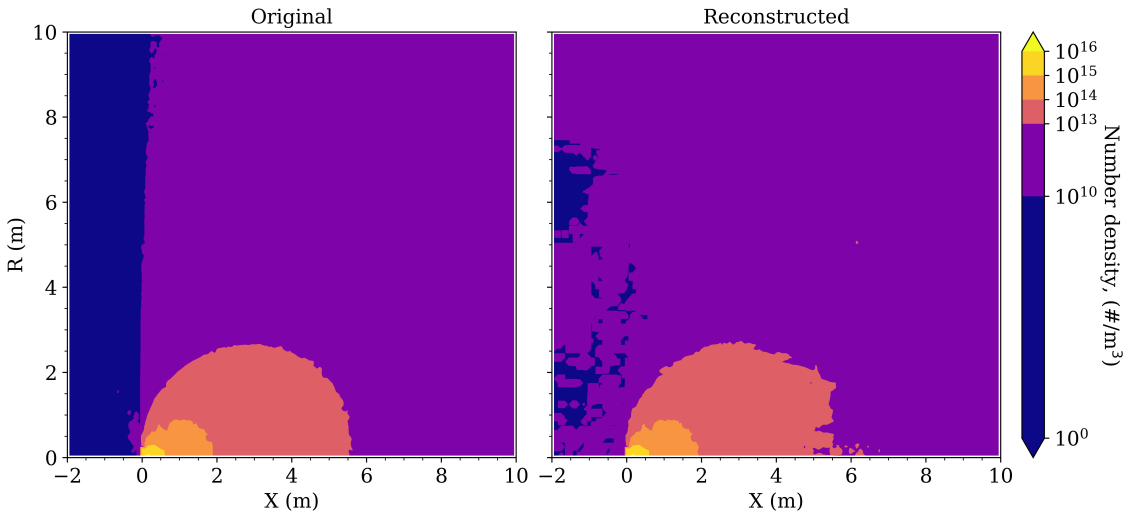


Figure F.6: The reconstructed dataset used the same compression parameters as in Figure F.5, however, here, the neutral atoms are compressed separately from the other fields. Compressing separately increased the CR to 24,563×.

Similarly, in the case of using a logarithmic normalization and a $\varepsilon = 10^{-2}$ relative error, compressing the neutral state separately from the other states results in an increased CR from 1.37×, seen in Figure F.7, to a CR of 1.91× seen in Figure F.8. This reconstruction dataset exhibited a similar resolution and a marginal increase in CR. Separating the compression of the neutral number state from the other states seems to offer a balance of maintaining reconstruction resolution, with a slightly improved CR.

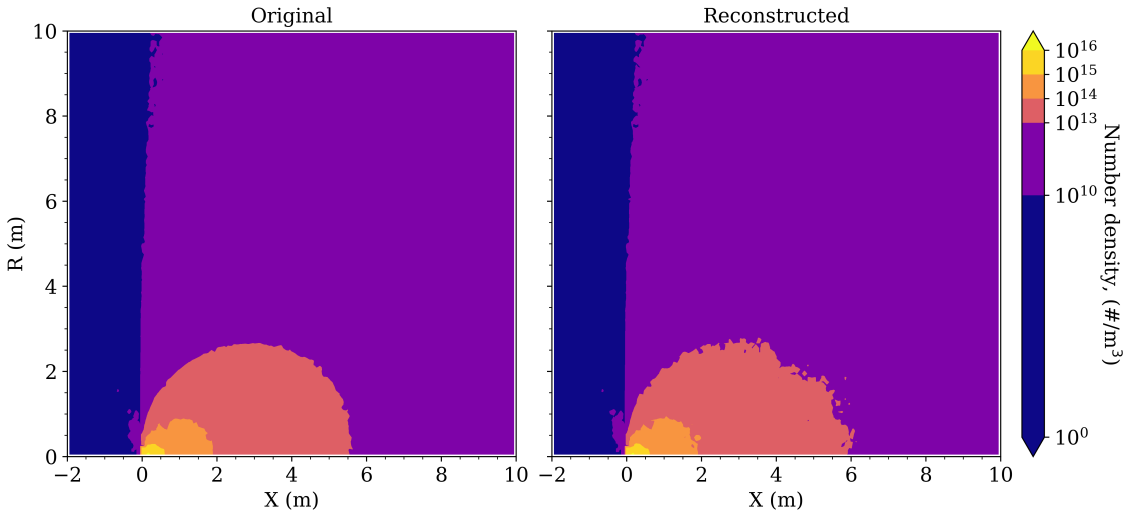


Figure F.7: Neutral number density contours are shown for reconstructions using a logarithmic normalization, $\varepsilon = 10^{-2}$ target relative error, and where neutral and ion number density fields are compressed together. The CR is 1.37×.

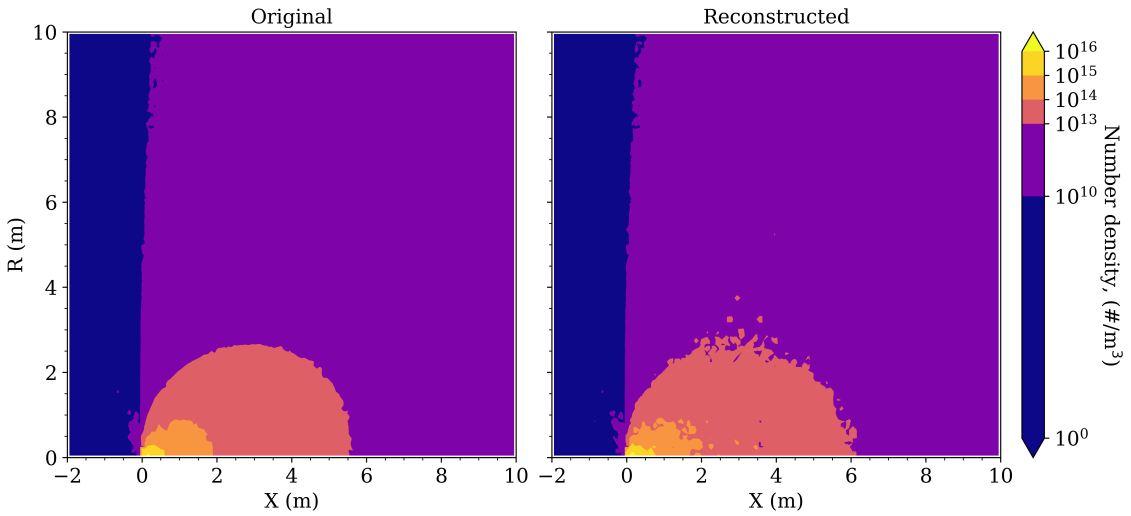


Figure F.8: (Right) The reconstructed dataset has a CR of 1.91× and the same compression input parameters as in Figure F.7, however, the neutral atom field is compressed separately from the other species. We see similar resolution of the reconstruction contours as in Figure F.7 and a marginally increased CR.

Table F.2: Quantitative compression performance comparison between compressing number density states together vs compressing neutral number density separately. Compressing number density separately increases the CR. However, the amount of increase depends heavily on the normalization method preferred. Note that we approximate the size corresponding to the neutral atom density by dividing the total size of the compression by 4. All other metrics reported for number densities compressed together are for the entire compression.

	Norm	Figure #	Size	Time (s)	ε	CR	TT-ranks	Latent size
z-score								
Original	-	-	1.3 GB		-	-	-	-
Together	z-score	F.5	275 KB[1]	36.12	10^{-2}	5213	1, 6, 40, 133, 91, 22, 10, 6, 2, 1	2
Separately	z-score	F.6	71 KB	4.78	10^{-2}	24,563	1, 5, 18, 15, 22, 7, 4, 2, 2, 1	2
logarithmic								
Original	-	-	1.3 GB		-	-	-	-
Together	logarithmic	F.7	1 GB[b]	606.2	10^{-2}	1.371	1, 6, 60, 600, 2999, 11891, 2523, 450, 150, 1	150
Separately	logarithmic	F.8	719 MB	112.08	10^{-2}	1.910	1, 6, 60, 599, 2984, 2569, 585, 148, 148, 1	148

The ion particles are modeled as three different types in TURF: 1) slow-moving ions resulting from MEX collisions, fast-moving ions resulting from MEX collisions, and 3) slow-moving ions from CEX collisions. The total ion number density is a sum of these three types. Figure F.9 shows a reconstruction where the three ion states are compressed separately, each with its own normalization combination and target relative error input (please refer to Table F.3 for details on compression settings). This flexible approach to different states

[1]We approximated this value by dividing the corresponding compression's size on disk by 4.

results in ion number density contours that are very close to the originals, indicating that fine-tuning the compression parameters for each species or state can be a strategy to improve the accuracy of the reconstruction and have a nontrivial compression ratio.

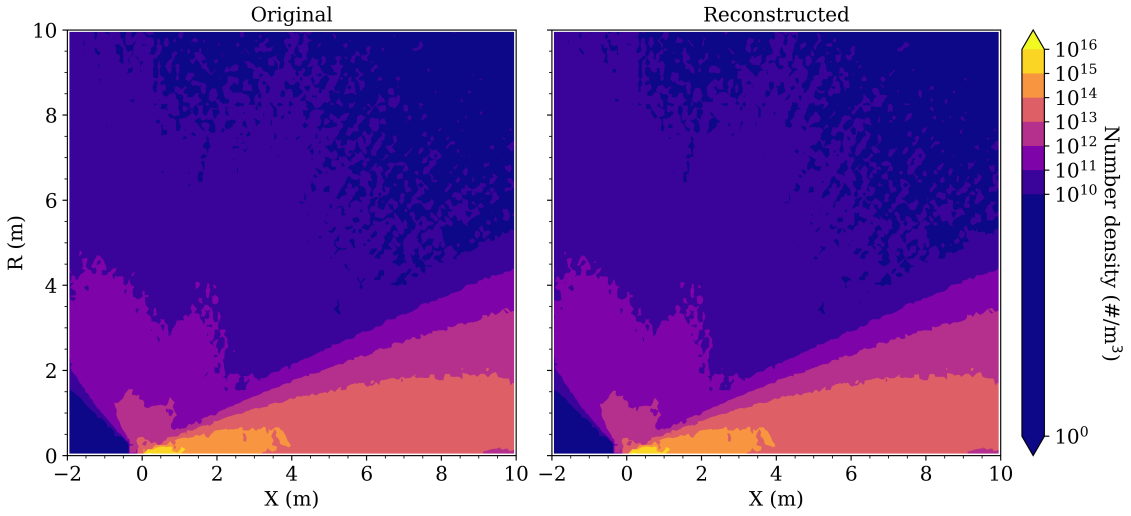


Figure F.9: (Right) The reconstructed ion number density contours here correspond to when the neutral atoms and three types of ions are all compressed separately. Please refer to Table ßF.3 for the details of compression parameters. The reconstructed ion number density contours are very close to the original contours while having an average CR of 2.88×.

Table F.3: Compressed data metrics for ion number density states compressed separately. By compressing states separately, we can achieve extreme compression up to $553,846\times$. Note that the total CR is severely impacted by the compression of slow-moving CEX ions and can be greatly improved by finding a more performant compression setting.

	Normalization 1	Normalization 2	ε	CR
Slow-moving MEX ions	logarithmic	-	10^{-1}	553,846
Fast-moving MEX ions	z-score	-	10^{-4}	14.17
Slow-moving CEX ions	logarithmic	z-score	10^{-2}	1.03
Total CR				**2.881**

F.3 Effect of spatial contextualization

Another parameter to consider is how different subdomains within the same snapshot are inputted into the compression pipeline. They can be inputted in a way to preserve locality and spatial relations between each other.

To parallelize the simulation in TURF, we partition the larger computational (-2, 0, 0) m $\times$ (10, 10, 10) m domain into 2 along each direction, resulting in 8 subdomains. In Section 5.2.6, Appendix F.1, and Appendix F.2, snapshots of size $60 \times 50 \times 50 \times n_s$ are accumulated along a new dimension. To determine the order in which each subdomain is written, this method only considered the binary codes (e.g., `0_0_0`, see Table F.4 for more details) in the names of the files. In this section, we investigate the effect of "stitching" the subdomains together in a spatially-aware way before preprocessing and sending to the compression pipeline. The subdomains are stitched to their respective positions in physical space and passed into the algorithm. Coordinates for the two domain input structures are defined in Table F.4. As a result of this, we write the 8 subdomain snapshots having n_s states corresponding to the same iteration into a NumPy array of shape $120 \times 100 \times 100 \times n_s$. We then reshape this 4-dimensional array into a shape $10 \times 12 \times 10 \times 10 \times 10 \times 10 \times n_s \times 1$ similar to all previous cases and increment over the last dimension.

Table F.4: Domain input structures for compression. The eight subdomains can be fed into the algorithm in a binary order which does not preserve the spatial relation between the subdomains in physical space, i.e (x_{min}, x_{max}, y_{min}, y_{max}, z_{min}, z_{max}). We investigate inputting a single spatially-aware domain where the subdomain respective positions in physical space are considered.

Binary subdomain ordering	**Spatially-aware subdomain ordering**
1. `0_0_0`: $(0, 60, 0, 50, 0, 50)$	1. `0_0_0`: $(0, 60, 0, 50, 0, 50)$
2. `0_0_1`: $(0, 60, 0, 50, 50, 100)$	2. `1_0_0`: $(60, 120, 0, 50, 0, 50)$
3. `0_1_0`: $(0, 60, 50, 100, 0, 50)$	3. `0_1_0`: $(0, 60, 50, 100, 0, 50)$
4. `0_1_1`: $(0, 60, 50, 100, 50, 100)$	4. `0_0_1`: $(0, 60, 0, 50, 50, 100)$
5. `1_0_0`: $(60, 120, 0, 50, 0, 50)$	5. `0_1_1`: $(0, 60, 50, 100, 50, 100)$
6. `1_0_1`: $(60, 120, 0, 50, 50, 100)$	6. `1_0_1`: $(60, 120, 0, 50, 50, 100)$
7. `1_1_0`: $(60, 120, 50, 100, 0, 50)$	7. `1_1_0`: $(60, 120, 50, 100, 0, 50)$
8. `1_1_1`: $(60, 120, 50, 100, 50, 100)$	8. `1_1_1`: $(60, 120, 50, 100, 50, 100)$

Figures F.10 and F.11 show the neutral and ion number contours when the subdomains are input into the compression algorithm as a single stitched domain, taking into account the spatial relation between the subdomains. Z-score normalization and a target relative error of $\varepsilon = 10^{-2}$ are used. For the neutral number densities, the contours of the near-

field plume are representative of the original dataset, but the backflow region is not well approximated. There is decreased resolution in the ion number density contours, as if the simulation was run with fewer particles. However, the CR slightly increases from 5213.6× to 5287.6× by taking into account spatial relations. Providing spatially-aware input to the compression pipeline is another setting to consider based on analysis interest, resolution desired, and computational resource requirements. Compression metrics for this compression input schema are summarized in Table F.4.

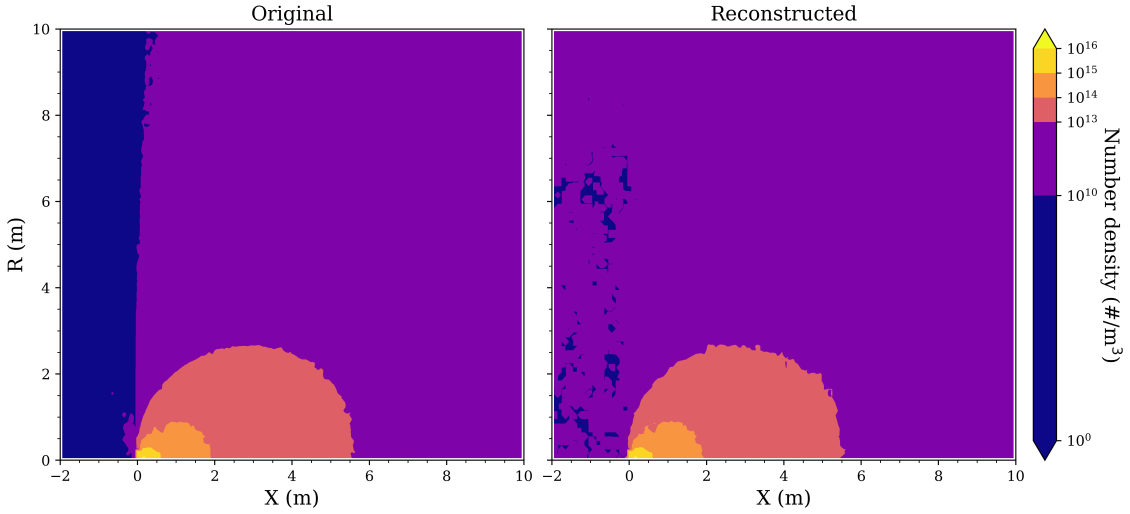


Figure F.10: (Right) Neutral number density contours obtained through compression with binary ordering of subdomains, z-score normalization, and a target relative error of $\varepsilon = 10^{-2}$. This combination achieves a high CR of 5213.6×.

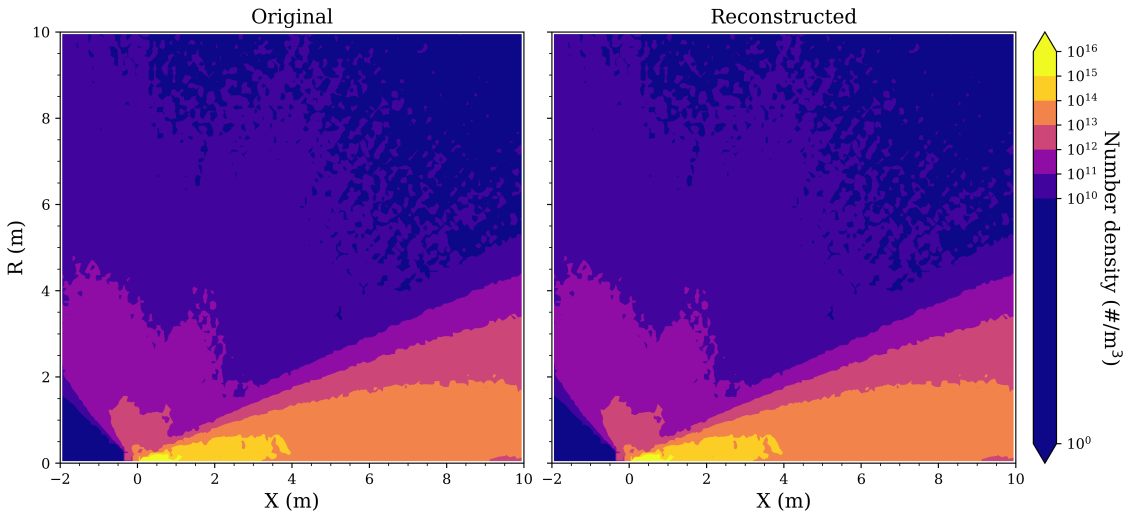


Figure F.11: (Right) Ion number density contours obtained through compression with a spatially-aware domain input, z-score normalization, and a target relative error of $\varepsilon = 10^{-2}$. The CR is increases to 5287.6×.

Figure F.12 and Figure F.13 show the effect of compression without and with considering spatial relations of the subdomains on the neutral number density contours, respectively. In both cases, square root normalization with $\varepsilon = 10^{-2}$ is used and one can see that the contours are near exact to those from the original dataset in both cases. Spatially-aware inputs do not significantly increase resolution over binary ordered inputs, and compression ratios between the two input schemas are comparable.

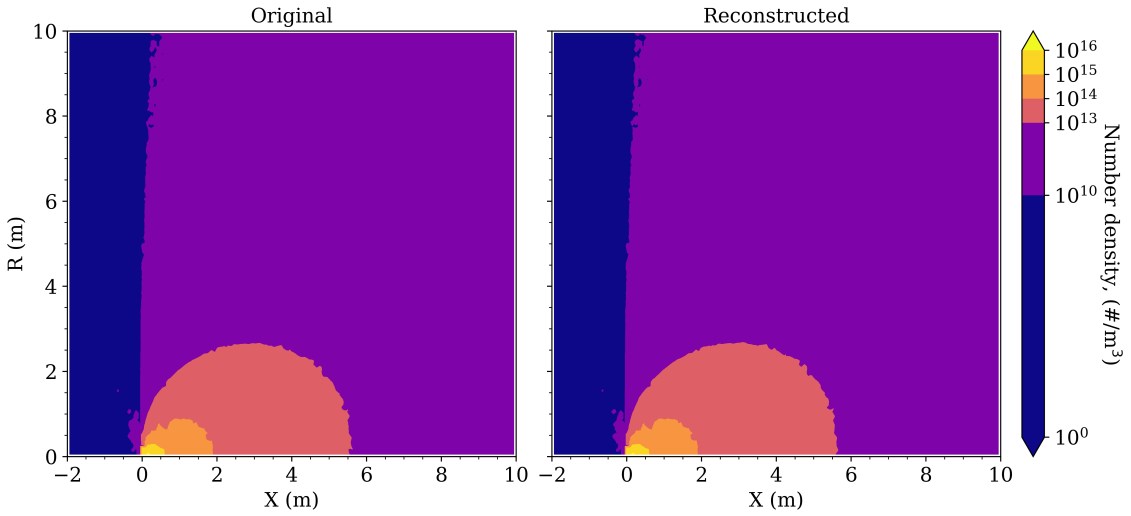


Figure F.12: Neutral number density contours square root normalization and $\varepsilon = 10^{-2}$ using a binary ordering of subdomains in compression, yielding a CR of 8.07×

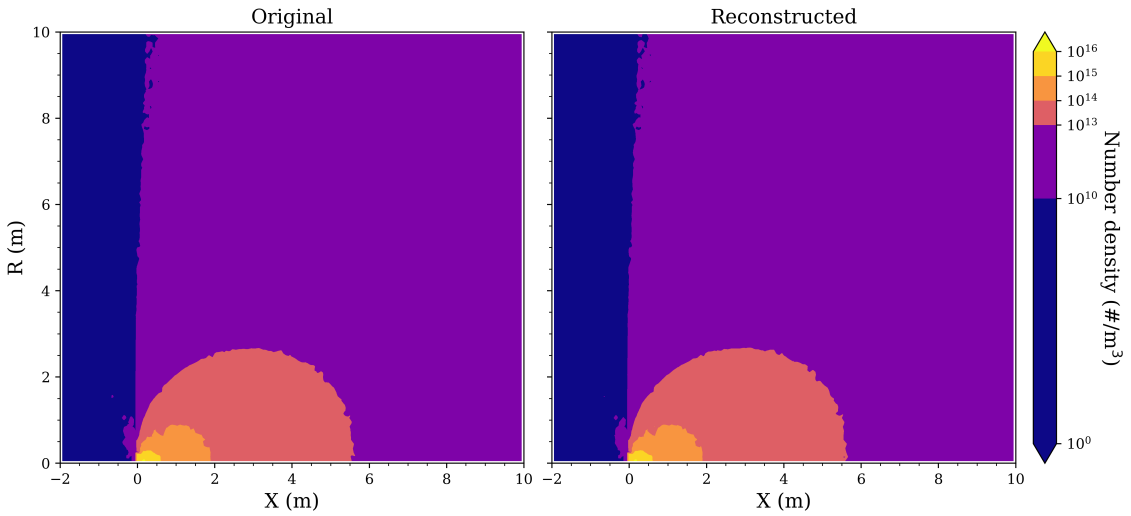


Figure F.13: Neutral number density contours square root normalization and $\varepsilon = 10^{-2}$ using the spatially-aware domain input to compression. This input schema still yields accurate reconstructions to the original with reasonable compression ratio of 7.23×.

As in the neutral number density contours, the ion contours are near exact to those from the original dataset in both compression with and without considering the relations between subdomains, seen in Figure F.14 and Figure F.15 respectively.

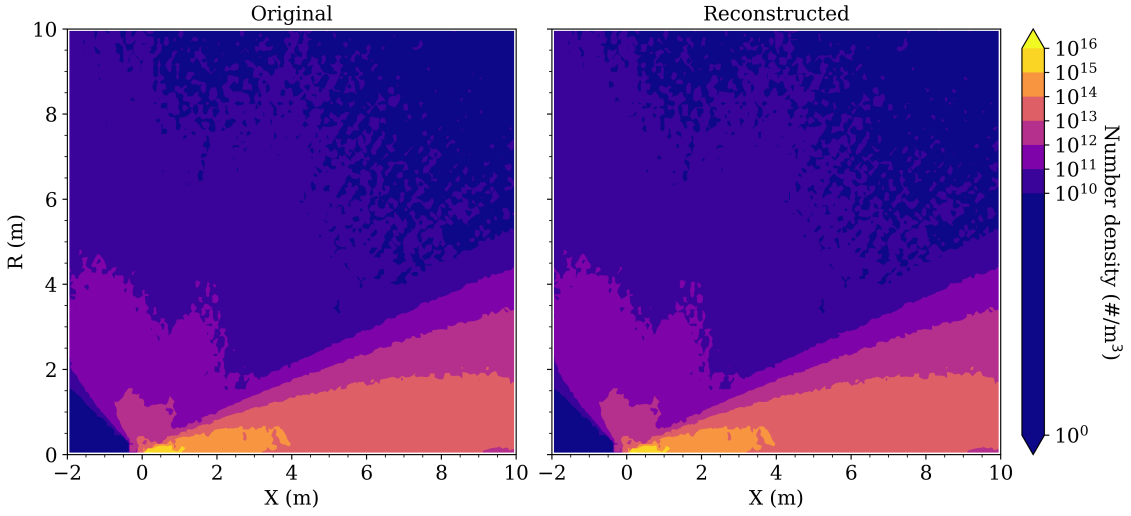


Figure F.14: Ion number density contours square root normalization and $\varepsilon = 10^{-2}$ using the binary ordering of subdomains into compression, yielding a CR of 8.07×.

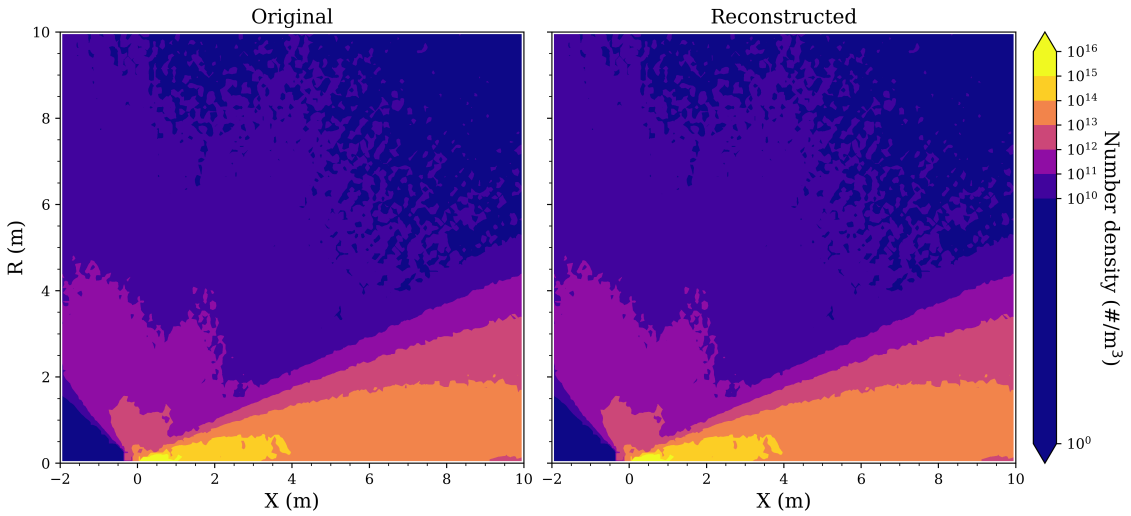


Figure F.15: Ion number density contours square root normalization and $\varepsilon = 10^{-2}$ using the spatially-aware domain input to compression. This yields accurate reconstructions while having a nontrivial CR of 7.23×.

Table F.5: Compression metrics for number density states using binary ordering and spatially-aware inputs. Using spatially-aware subdomain inputs results in accurate reconstructions, but not significantly increased resolution over binary ordered inputs. Further, the compression ratios between the two input schemas are comparable.

	Norm 1	Figure #	Size	Time (s)	ε	CR	TT-ranks	Latent size
z-score, 10^{-2}								
Original	-	-	5.5 GB	-	-	-	-	-
Binary	z-score	F.5	1.1 MB	36.31	10^{-2}	5213.6	1, 6, 40, 133, 91, 22, 10, 6, 2, 1	2
Spatially-aware	z-score	F.10,F.11	1.1 MB	28.25	10^{-2}	5287.6	1, 10, 57, 97, 64, 17, 5, 2, 1	2
square root, 10^{-2}								
Original	-	-	5.5 GB	-	-	-	-	-
Binary	square root	F.12,F.14	681 MB	135.63	10^{-2}	8.07	1, 6, 60, 586, 2829, 2590, 535, 71, 34, 1	34
Spatially-aware	square root	F.13,F.15	755 MB	127.17	10^{-2}	7.23	1, 12, 119, 1143, 5416, 650, 69, 34, 1	34

BIBLIOGRAPHY

[1] Karim Abed-Meraim, Nguyen Linh Trung, Remy Boyer, et al. Adaptive algorithms for tracking tensor-train decomposition of streaming tensors. In *2020 28th European signal processing conference (EUSIPCO)*, pages 995–999. IEEE, 2021.

[2] David H Ackley, Geoffrey E Hinton, and Terrence J Sejnowski. A learning algorithm for boltzmann machines. *Cognitive science*, 9(1):147–169, 1985.

[3] Ian Affleck, Tom Kennedy, Elliott H Lieb, and Hal Tasaki. Valence bond ground states in isotropic quantum antiferromagnets. *Communications in Mathematical Physics*, 115(3):477–528, 1988.

[4] D. Aksoy and D. Kim. Self-oscillating gel simulation snapshot dataset, 2025. Dataset contains simulation snapshots of hexagonal gel sheets using a triangular lattice structure, with both training (19.7 GB) and test (123 GB) data components.

[5] 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.

[6] Doruk Aksoy and Alex A Gorodetsky. Incremental hierarchical tucker decomposition. *arXiv preprint arXiv:2412.16544*, 2024.

[7] 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, 2024.

[8] 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. In *The 38th International Electric Propulsion Conference*, 2024.

[9] 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.

[10] Alen Alexanderian. Optimal experimental design for infinite-dimensional bayesian inverse problems governed by pdes: A review. *Inverse Problems*, 37(4):043001, 2021.

[11] Eloi Alonso, Adam Jelley, Vincent Micheli, Anssi Kanervisto, Amos J Storkey, Tim Pearce, and François Fleuret. Diffusion for world modeling: Visual details matter in atari. *Advances in Neural Information Processing Systems*, 37:58757–58791, 2024.

[12] Alexander Amini, Wilko Schwarting, Guy Rosman, Brandon Araki, Sertac Karaman, and Daniela Rus. Variational autoencoder for end-to-end control of autonomous driving with novelty detection and training de-biasing. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 568–575. IEEE, 2018.

[13] Pia Anttila, Pentti Paatero, Unto Tapper, and Olli Järvinen. Source identification of bulk wet deposition in finland by positive matrix factorization. *Atmospheric Environment*, 29(14):1705–1718, 1995.

[14] Samuel J. Araki. Fast computation of high energy elastic collision scattering angle for electric propulsion plume simulation. *AIP Conference Proceedings*, 1786(1):170004, 2016.

[15] Samuel J. Araki. Multiscale coupling of spacecraft charging model with electric propulsion plume simulation. *IEEE Transactions on Plasma Science*, 47(11):4898–4908, 2019.

[16] Samuel J Araki and Alexander Barrie. Electric propulsion plume simulation coupled with spacecraft charging. In *2018 Joint Propulsion Conference*, page 4906, 2018.

[17] Samuel J. Araki and Robert S. Martin. Dynamic load balancing with over decomposition in plasma plume simulations. *Journal of Parallel and Distributed Computing*, 163:136–146, 2022.

[18] Samuel J. Araki and Richard E. Wirz. Ion–neutral collision modeling using classical scattering with spin-orbit free interaction potential. *IEEE Transactions on Plasma Science*, 41(3):470–480, 2013.

[19] Scott Atchley, Christopher Zimmer, John Lange, David Bernholdt, Veronica Melesse Vergara, Thomas Beck, Michael Brim, Reuben Budiardja, Sunita Chandrasekaran, Markus Eisenbach, et al. Frontier: exploring exascale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–16, 2023.

[20] 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.

[21] Nathan Baker, Frank Alexander, Timo Bremer, Aric Hagberg, Yannis Kevrekidis, Habib Najm, Manish Parashar, Abani Patra, James Sethian, Stefan Wild, et al. Workshop report on basic research needs for scientific machine learning: Core technologies for artificial intelligence. Technical report, USDOE Office of Science (SC), Washington, DC (United States), 2019.

[22] Laura Balzano and Stephen J Wright. On grouse and incremental svd. In *2013 5th IEEE international workshop on computational advances in multi-sensor adaptive processing (camsap)*, pages 1–4. IEEE, 2013.

[23] Tatyana V Bandos, Lorenzo Bruzzone, and Gustavo Camps-Valls. Classification of hyperspectral images with regularized linear discriminant analysis. *IEEE Transactions on Geoscience and Remote Sensing*, 47(3):862–873, 2009.

[24] Muthu Baskaran, M Harper Langston, Tahina Ramananandro, David Bruns-Smith, Tom Henretty, James Ezick, and Richard Lethin. Accelerated low-rank updates to tensor decompositions. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2016.

[25] Casey Battaglino, Grey Ballard, and Tamara G Kolda. A practical randomized cp tensor decomposition. *SIAM Journal on Matrix Analysis and Applications*, 39(2):876–901, 2018.

[26] Etienne Becht, Leland McInnes, John Healy, Charles-Antoine Dutertre, Immanuel W H Kwok, Lai Guan Ng, Florent Ginhoux, and Evan W Newell. Dimensionality reduction for visualizing single-cell data using umap. *Nature Biotechnology*, 37(1):38–44, December 2018.

[27] Florian Becker, Mari Nygård, Jan Nygård, Age Smilde, and Evrim Acar. Phenotyping of cervical cancer risk groups via generalized low-rank models using medical questionnaires. In *Symposium of the Norwegian AI Society*, pages 94–110. Springer International Publishing Cham, 2022.

[28] Mikhail Belkin and Partha Niyogi. Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6):1373–1396, 2003.

[29] Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemyslaw Debiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, et al. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[30] Çağdaş Bilen, Alexey Ozerov, and Patrick Pérez. Audio declipping via nonnegative matrix factorization. In *2015 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pages 1–5. IEEE, 2015.

[31] Iain D Boyd and Rainer A Dressler. Far field modeling of the plasma plume of a hall thruster. *Journal of Applied Physics*, 92(4):1764–1774, 2002.

[32] Matthew Brand. Incremental singular value decomposition of uncertain data with missing values. In *Computer Vision—ECCV 2002: 7th European Conference on Computer Vision Copenhagen, Denmark, May 28–31, 2002 Proceedings, Part I 7*, pages 707–720. Springer, 2002.

[33] Matthew Brand. Fast online svd revisions for lightweight recommender systems. In *Proceedings of the 2003 SIAM international conference on data mining*, pages 37–46. SIAM, 2003.

[34] Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear algebra and its applications*, 415(1):20–30, 2006.

[35] Paul Breiding and Nick Vannieuwenhoven. A riemannian trust region method for the canonical tensor rank approximation problem. *SIAM Journal on Optimization*, 28(3):2435–2465, 2018.

[36] Rasmus Bro and Henk AL Kiers. A new efficient method for determining the number of components in parafac models. *Journal of Chemometrics: A Journal of the Chemometrics Society*, 17(5):274–286, 2003.

[37] Eli Bronstein, Mark Palatucci, Dominik Notz, Brandyn White, Alex Kuefler, Yiren Lu, Supratik Paul, Payam Nikdel, Paul Mougin, Hongge Chen, Justin Fu, Austin Abrams, Punit Shah, Evan Racah, Benjamin Frenkel, Shimon Whiteson, and Dragomir Anguelov. Hierarchical model-based imitation learning for planning in autonomous driving. In *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8652–8659. IEEE, 2022.

[38] Daniel Brown, Russell Coleman, Ravi Srinivasan, and Scott Niekum. Safe imitation learning via fast bayesian reward inference from preferences. In *International Conference on Machine Learning*, pages 1165–1177. PMLR, 2020.

[39] James R Bunch and Christopher P Nielsen. Updating the singular value decomposition. *Numerische Mathematik*, 31(2):111–129, 1978.

[40] Xiaoyu Cai, Qiongli Jing, Bo Peng, Yuanyuan Zhang, Yuting Wang, Ju Tang, et al. Automatic traffic state recognition based on video features extracted by an autoencoder. *Mathematical Problems in Engineering*, 2022, 2022.

[41] 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.

[42] Raymond B. Cattell. "parallel proportional profiles" and other principles for determining the choice of factors by rotation. *Psychometrika*, 9(4):267–283, 1944.

[43] Raymond B Cattell. The three basic factor-analytic research designs—their interrelations and derivatives. *Psychological bulletin*, 49(5):499, 1952.

[44] Dimitris G Chachlakis, Mayur Dhanaraj, Ashley Prater-Bennette, and Panos P Markopoulos. Dynamic l1-norm tucker tensor decomposition. *IEEE Journal of Selected Topics in Signal Processing*, 15(3):587–602, 2021.

[45] 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.

[46] T Charoy, J P Boeuf, A Bourdon, J A Carlsson, P Chabert, B Cuenot, D Eremin, L Garrigues, K Hara, I D Kaganovich, A T Powis, A Smolyakov, D Sydorenko, A Tavant, O Vermorel, and W Villafana. 2d axial-azimuthal particle-in-cell benchmark for low-temperature partially magnetized plasmas. *Plasma Sources Science and Technology*, 28(10):105010, oct 2019.

[47] 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.

[48] Brian Chen. Atari games dataset, 2023. Data set.

[49] 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.

[50] Brian Chen, Siddhant Tandon, David Gorsich, Alex Gorodetsky, and Shravan Veerapaneni. Behavioral cloning in atari games using a combined variational autoencoder and predictor model. In *2021 IEEE Congress on Evolutionary Computation (CEC)*, pages 2077–2084. IEEE, 2021.

[51] Jacqueline H Chen, Alok Choudhary, Bronis De Supinski, Matthew DeVries, Evatt R Hawkes, Scott Klasky, Wei-Keng Liao, Kwan-Liu Ma, John Mellor-Crummey, Norbert Podhorszki, et al. Terascale direct numerical simulations of turbulent combustion using s3d. *Computational Science & Discovery*, 2(1):015001, 2009.

[52] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *International conference on machine learning*, pages 1597–1607. PmLR, 2020.

[53] Xin Chen, Sam Toyer, Cody Wild, Scott Emmons, Ian Fischer, Kuang-Huei Lee, Neel Alex, Steven H Wang, Ping Luo, Stuart Russell, et al. An empirical investigation of representation learning for imitation. *arXiv preprint arXiv:2205.07886*, 2022.

[54] Zhaomin Chen, Chai Kiat Yeo, Bu Sung Lee, and Chiew Tong Lau. Autoencoder-based network anomaly detection. In *2018 Wireless Telecommunications Symposium (WTS)*, page 1–5. IEEE, April 2018.

[55] Dehua Cheng, Richard Peng, Yan Liu, and Ioakeim Perros. Spals: Fast alternating least squares via implicit leverage scores sampling. *Advances in neural information processing systems*, 29, 2016.

[56] Lei Cheng, Zhongtao Chen, Qingjiang Shi, Yik-Chung Wu, and Sergios Theodoridis. Towards flexible sparsity-aware modeling: Automatic tensor rank learning using the generalized hyperbolic prior. *IEEE Transactions on Signal Processing*, 70:1834–1849, 2022.

[57] Zhengxue Cheng, Heming Sun, Masaru Takeuchi, and Jiro Katto. Deep convolutional autoencoder-based lossy image compression. In *2018 Picture Coding Symposium (PCS)*, page 253–257. IEEE, June 2018.

[58] Maria Chierichetti and Fatemeh Davoudi. Optimal sensor location along a beam using machine learning. In *AIAA SCITECH 2022 Forum*, page 1465, 2022.

[59] Seungjin Choi, Andrzej Cichocki, Hyung-Min Park, and Soo-Young Lee. Blind source separation and independent component analysis: A review. *Neural Information Processing-Letters and Reviews*, 6(1):1–57, 2005.

[60] Andrzej Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *arXiv preprint arXiv:1403.2048*, 2014.

[61] P Coche and L Garrigues. A two-dimensional (azimuthal-axial) particle-in-cell model of a hall thruster. *Physics of Plasmas*, 21(2), 2014.

[62] Pierre Comon. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314, 1994.

[63] Hussam Al Daas, Grey Ballard, and Peter Benner. Parallel algorithms for tensor train arithmetic. *SIAM Journal on Scientific Computing*, 44(1):C25–C53, 2022.

[64] John M Dawson. Particle simulation of plasmas. *Reviews of modern physics*, 55(2):403, 1983.

[65] Saibal De, Eduardo Corona, Paramsothy Jayakumar, and Shravan Veerapaneni. Tensor-train compression of discrete element method simulation data. *Journal of Terramechanics*, 113:100967, 2024.

[66] 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.

[67] Lieven De Lathauwer. Decompositions of a higher-order tensor in block terms—part ii: Definitions and uniqueness. *SIAM Journal on Matrix Analysis and Applications*, 30(3):1033–1066, 2008.

[68] 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, 2000.

[69] 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, 2000.

[70] Lieven De Lathauwer and Joos Vandewalle. Dimensionality reduction in higher-order signal processing and rank-(r1, r2,..., rn) reduction in multilinear algebra. *Linear Algebra and its Applications*, 391:31–55, 2004.

[71] Karthik Devarajan. Nonnegative matrix factorization: an analytical and interpretive tool in computational biology. *PLoS computational biology*, 4(7):e1000029, 2008.

[72] Renwei Dian, Shutao Li, and Leyuan Fang. Learning a low tensor-train rank representation for hyperspectral image super-resolution. *IEEE transactions on neural networks and learning systems*, 30(9):2672–2683, 2019.

[73] Alex Diaz-Papkovich, Luke Anderson-Trocmé, and Simon Gravel. A review of umap in population genetics. *Journal of Human Genetics*, 66(1):85–91, October 2020.

[74] Sergey V Dolgov, Boris N Khoromskij, and Ivan V Oseledets. Fast solution of parabolic problems in the tensor train/quantized tensor train format with initial application to the fokker–planck equation. *SIAM Journal on Scientific Computing*, 34(6):A3016–A3038, 2012.

[75] Qian Du and James E Fowler. Hyperspectral image compression using jpeg2000 and principal component analysis. *IEEE Geoscience and Remote sensing letters*, 4(2):201–205, 2007.

[76] Hala M Ebied. Feature extraction using pca and kernel-pca for face recognition. In *2012 8th International Conference on Informatics and Systems (INFOS)*, pages MM–72. IEEE, 2012.

[77] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[78] Joshua Eckels, Collin B Whittaker, Benjamin Jorns, and Alex Gorodetsky. Optimal experimental design to learn reduced-fidelity models for porous electrosprays. In *AIAA SCITECH 2023 Forum*, page 0066, 2023.

[79] Joshua D. Eckels, Thomas A. Marks, Doruk Aksoy, Sruti Vutukury, and Alex A. Gorodetsky. Dynamic mode decomposition for particle-in-cell simulations of a hall thruster and plume. In *IEPC 2024*, 2024.

[80] 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.

[81] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2):179–188, 1936.

[82] J Kevin Ford, Robert C MacCallum, and Marianne Tait. The application of exploratory factor analysis in applied psychology: A critical review and analysis. *Personnel psychology*, 39(2):291–314, 1986.

[83] Nicholas Galioto and Alex Arkady Gorodetsky. Bayesian system id: optimal management of parameter, model, and measurement uncertainty. *Nonlinear Dynamics*, 102(1):241–267, 2020.

[84] Timur Garipov, Dmitry Podoprikhin, Alexander Novikov, and Dmitry P. Vetrov. Ultimate tensorization: compressing convolutional and FC layers alike. *CoRR*, abs/1611.03214, 2016.

[85] Gene Golub and William Kahan. Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis*, 2(2):205–224, 1965.

[86] Gene H Golub and Christian Reinsch. Singular value decomposition and least squares solutions. In *Handbook for automatic computation: volume II: linear algebra*, pages 134–151. Springer, 1971.

[87] Alex Gorodetsky, Sertac Karaman, and Youssef Marzouk. High-dimensional stochastic optimal control using continuous tensor decompositions. *The International Journal of Robotics Research*, 37(2-3):340–377, 2018.

[88] Alex A Gorodetsky, Patrick D Mullen, Aditya Deshpande, Joshua C Dolence, Chad D Meyer, Jonah M Miller, and Luke F Roberts. Thermal radiation transport with tensor trains. *arXiv preprint arXiv:2503.18056*, 2025.

[89] Genevieve Gorrell. Generalized hebbian algorithm for incremental singular value decomposition in natural language processing. In *11th conference of the European chapter of the association for computational linguistics*, pages 97–104, 2006.

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

[91] Ekta Gujral, Ravdeep Pasricha, and Evangelos E Papalexakis. Sambaten: Sampling-based batch incremental tensor decomposition. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, pages 387–395. SIAM, 2018.

[92] Chenfeng Guo and Dongrui Wu. Canonical correlation analysis (cca) based multi-view learning: An overview. *arXiv preprint arXiv:1907.01693*, 2019.

[93] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[94] 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.

[95] 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.

[96] Ali Hassani, Amir Iranmanesh, and Najme Mansouri. Text mining using nonnegative matrix factorization and latent semantic analysis. *Neural Computing and Applications*, 33:13745–13766, 2021.

[97] Johan Håstad. Tensor rank is np-complete. *Journal of algorithms*, 11(4):644–654, 1990.

[98] Hans Hersbach, Bill Bell, Paul Berrisford, Shoji Hirahara, András Horányi, Joaquín Muñoz-Sabater, Julien Nicolas, Carole Peubey, Raluca Radu, Dinand Schepers, et al. The era5 global reanalysis. *Quarterly journal of the royal meteorological society*, 146(730):1999–2049, 2020.

[99] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. Stable baselines. https://github.com/hill-a/stable-baselines, 2018.

[100] Christopher J Hillar and Lek-Heng Lim. Most tensor problems are np-hard. *Journal of the ACM (JACM)*, 60(6):1–39, 2013.

[101] Geoffrey E Hinton. Training products of experts by minimizing contrastive divergence. *Neural computation*, 14(8):1771–1800, 2002.

[102] Geoffrey E Hinton and Sam Roweis. Stochastic neighbor embedding. *Advances in neural information processing systems*, 15, 2002.

[103] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

[104] Frank L. Hitchcock. The expression of a tensor or a polyadic as a sum of products. *Journal of Mathematics and Physics*, 6(1-4):164–189, 1927.

[105] Frank L Hitchcock. Multiple invariants and generalized rank of a p-way matrix or tensor. *Journal of Mathematics and Physics*, 7(1-4):39–79, 1928.

[106] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[107] Sebastian Holtz, Thorsten Rohwedder, and Reinhold Schneider. The alternating linear scheme for tensor optimization in the tensor train format. *SIAM Journal on Scientific Computing*, 34(2):A683–A713, 2012.

[108] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[109] Harold Hotelling. Relations between two sets of variates. *Biometrika*, 28(3-4):321–377, December 1936.

[110] Anthony Hu, Lloyd Russell, Hudson Yeo, Zak Murez, George Fedoseev, Alex Kendall, Jamie Shotton, and Gianluca Corrado. Gaia-1: A generative world model for autonomous driving. *arXiv preprint arXiv:2309.17080*, 2023.

[111] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022.

[112] Weiming Hu, Xi Li, Xiaoqin Zhang, Xinchu Shi, Stephen Maybank, and Zhongfei Zhang. Incremental tensor subspace learning and its applications to foreground segmentation and tracking. *International Journal of Computer Vision*, 91:303–327, 2011.

[113] Shengyu Huang, K Selçuk Candan, and Maria Luisa Sapino. Bicp: block-incremental cp decomposition with update sensitive refinement. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pages 1221–1230, 2016.

[114] Benjamin Huber, Reinhold Schneider, and Sebastian Wolf. A randomized tensor train singular value decomposition. In *Compressed Sensing and its Applications: Second International MATHEON Conference 2015*, pages 261–290. Springer, 2018.

[115] Ahmed Hussein, Mohamed Medhat Gaber, Eyad Elyan, and Chrisina Jayne. Imitation learning: A survey of learning methods. *ACM Computing Surveys (CSUR)*, 50(2):1–35, 2017.

[116] Aapo Hyvärinen and Peter Dayan. Estimation of non-normalized statistical models by score matching. *Journal of Machine Learning Research*, 6(4), 2005.

[117] Aapo Hyvärinen and Erkki Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4-5):411–430, 2000.

[118] Ozan Irsoy and Ethem Alpaydin. Unsupervised feature extraction with autoencoder trees. *Neurocomputing*, 258:63–73, October 2017.

[119] Google is using YouTube videos to train its AI video generator. Google is using youtube videos to train its ai video generator. *CNBC*, 2025.

[120] Zubayer Islam, Mohamed Abdel-Aty, Qing Cai, and Jinghui Yuan. Crash data augmentation using variational autoencoder. *Accident Analysis & Prevention*, 151:105950, March 2021.

[121] Gordana Ivosev, Lyle Burton, and Ron Bonner. Dimensionality reduction and visualization in principal component analysis. *Analytical chemistry*, 80(13):4933–4944, 2008.

[122] Rudolf Jaffé, Kaelin M Cawley, and Youhei Yamashita. Applications of excitation emission matrix fluorescence with parallel factor analysis (eem-parafac) in assessing environmental dynamics of natural dissolved organic matter (dom) in aquatic environments: a review. *Advances in the physicochemical characterization of dissolved organic matter: impact on natural and engineered systems*, pages 27–73, 2014.

[123] Christopher J James and Christian W Hesse. Independent component analysis for biomedical signals. *Physiological measurement*, 26(1):R15, 2004.

[124] Jun-Gi Jang and U Kang. Static and streaming tucker decomposition for dense tensors. *ACM Transactions on Knowledge Discovery from Data*, 17(5):1–34, 2023.

[125] Antonius P. A. Janssen, Sebastian H. Grimm, Ruud H. M. Wijdeven, Eelke B. Lenselink, Jacques Neefjes, Constant A. A. van Boeckel, Gerard J. P. van Westen, and Mario van der Stelt. Drug discovery maps, a machine learning model that visualizes and predicts kinome–inhibitor interaction landscapes. *Journal of Chemical Information and Modeling*, 59(3):1221–1229, October 2018.

[126] Miguel Jaques, Michael Burke, and Timothy M Hospedales. Newtonianvae: Proportional control and goal identification from pixels via physical latent spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4454–4463, 2021.

[127] Sham M Kakade and Dean P Foster. Multi-view regression via canonical correlation analysis. In *International Conference on Computational Learning Theory*, pages 82–96. Springer, 2007.

[128] Anssi Kanervisto, Joonas Pussinen, and Ville Hautamäki. Benchmarking end-to-end behavioural cloning on video games. In *2020 IEEE conference on games (CoG)*, pages 558–565. IEEE, 2020.

[129] Azam Karami, Mehran Yazdi, and Grégoire Mercier. Compression of hyperspectral images using discerete wavelet transform and tucker decomposition. *IEEE journal of selected topics in applied earth observations and remote sensing*, 5(2):444–450, 2012.

[130] Henk A. L. Kiers. Towards a standardized notation and terminology in multiway analysis. *Journal of Chemometrics*, 14(3):105–122, 2000.

[131] Heecheol Kim, Yoshiyuki Ohmura, and Yasuo Kuniyoshi. Transformer-based deep imitation learning for dual-arm robot manipulation. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 8965–8972. IEEE, 2021.

[132] Seung Wook Kim, Yuhao Zhou, Jonah Philion, Antonio Torralba, and Sanja Fidler. Learning to simulate dynamic environments with gamegan. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1231–1240, 2020.

[133] Yong-Deok Kim, Eunhyeok Park, Sungjoo Yoo, Taelim Choi, Lu Yang, and Dongjun Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[134] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *International Conference on Learning Representations (ICLR)*, 2014.

[135] Durk P Kingma, Shakir Mohamed, Danilo Jimenez Rezende, and Max Welling. Semi-supervised learning with deep generative models. *Advances in neural information processing systems*, 27, 2014.

[136] Dmitry Kobak and Philipp Berens. The art of using t-sne for single-cell transcriptomics. *Nature Communications*, 10(1), November 2019.

[137] Tamara G Kolda and Brett W Bader. Tensor decompositions and applications. *SIAM review*, 51(3):455–500, 2009.

[138] Tamara G Kolda, Grey Ballard, and Woody Nathan Austin. Parallel tensor compression for large-scale scientific data. Technical report, Sandia National Lab.(SNL-CA), Livermore, CA (United States), 2015.

[139] Jean Kossaifi, Aran Khanna, Zachary Lipton, Tommaso Furlanello, and Anima Anandkumar. Tensor contraction layers for parsimonious deep nets. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 26–32, 2017.

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

[141] 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.

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

[143] 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.

[144] Joseph B Kruskal. Three-way arrays: rank and uniqueness of trilinear decompositions, with application to arithmetic complexity and statistics. *Linear algebra and its applications*, 18(2):95–138, 1977.

[145] Liwei Kuang, Fei Hao, Laurence T Yang, Man Lin, Changqing Luo, and Geyong Min. A tensor-based approach for big data representation and dimensionality reduction. *IEEE transactions on emerging topics in computing*, 2(3):280–291, 2014.

[146] Ludmila I Kuncheva and William J Faithfull. Pca feature extraction for change detection in multidimensional unlabeled data. *IEEE transactions on neural networks and learning systems*, 25(1):69–80, 2013.

[147] NASA Jet Propulsion Laboratory. Russian stationary plasma thrusters (spt), 1997. Public Domain.

[148] NASA Jet Propulsion Laboratory. Hall thruster for psyche operating at jpl. https://photojournal.jpl.nasa.gov/catalog/PIA23879, 2020. Public Domain.

[149] T. Lafleur, S. D. Baalrud, and P. Chabert. Theory for the anomalous electron transport in hall effect thrusters. ii. kinetic model. *Physics of Plasmas*, 23:11101, 5 2016.

[150] Hugo Larochelle and Yoshua Bengio. Classification using discriminative restricted boltzmann machines. In *Proceedings of the 25th international conference on Machine learning*, pages 536–543, 2008.

[151] William H Lawton and Edward A Sylvestre. Self modeling curve resolution. *Technometrics*, 13(3):617–633, 1971.

[152] V Lebedev, Y Ganin, M Rakhuba, I Oseledets, and V Lempitsky. Speeding-up convolutional neural networks using fine-tuned cp-decomposition. In *3rd International Conference on Learning Representations, ICLR 2015-Conference Track Proceedings*, 2015.

[153] Daniel D Lee and H Sebastian Seung. Learning the parts of objects by non-negative matrix factorization. *nature*, 401(6755):788–791, 1999.

[154] Wentian Li, Jane E. Cerise, Yaning Yang, and Henry Han. Application of t-sne to human genetic data. *Journal of Bioinformatics and Computational Biology*, 15(04):1750017, August 2017.

[155] Weichao Liang, Zhiang Wu, Zhe Li, and Yong Ge. Crimetensor: Fine-scale crime prediction via tensor learning with spatiotemporal consistency. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 13(2):1–24, 2022.

[156] Yitao Liang, Marlos C Machado, Erik Talvitie, and Michael Bowling. State-of-the-art control of atari games using shallow reinforcement learning. *arXiv preprint arXiv:1512.01563*, 2015.

[157] Amaro Lima, Heiga Zen, Yoshihiko Nankaku, Chiyomi Miyajima, Keiichi Tokuda, and Tadashi Kitamura. On the use of kernel pca for feature extraction in speech recognition. *IEICE TRANSACTIONS on Information and Systems*, 87(12):2802–2811, 2004.

[158] Chien-Liang Liu, Wen-Hoar Hsaio, Chia-Hoang Lee, and Fu-Sheng Gou. Semi-supervised linear discriminant clustering. *IEEE Transactions on cybernetics*, 44(7):989–1000, 2013.

[159] Chihuang Liu, Joseph JaJa, and Luiz Pessoa. Leica: Laplacian eigenmaps for group ica decomposition of fmri data. *NeuroImage*, 169:363–373, April 2018.

[160] Haifeng Liu, Zhaohui Wu, Xuelong Li, Deng Cai, and Thomas S Huang. Constrained nonnegative matrix factorization for image representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1299–1311, 2011.

[161] 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.

[162] Xin Liu et al. Locally linear embedding (lle) for mri based alzheimer's disease classification. *Neuroimage*, 83:148–157, 2013.

[163] Zhiyong Liu. Facial expression recognition based on isomap. In *2011 International Conference on Electrical and Control Engineering*, pages 6591–6593. IEEE, 2011.

[164] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016.

[165] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[166] Li Ma, Melba M Crawford, and Jinwen Tian. Anomaly detection for hyperspectral images based on robust locally linear embedding. *Journal of Infrared, Millimeter, and Terahertz Waves*, 31:753–762, 2010.

[167] Duncan MacRae. Direct factor analysis of sociometric data. *Sociometry*, 23(4):360–371, 1960.

[168] Michael W Mahoney et al. Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224, 2011.

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

[170] Krithika Manohar, Bingni W Brunton, J Nathan Kutz, and Steven L Brunton. Data-driven sparse sensor placement for reconstruction: Demonstrating the benefits of exploiting known patterns. *IEEE Control Systems Magazine*, 38(3):63–86, 2018.

[171] Thomas A. Marks and Alex A. Gorodetsky. Hall thruster simulations in Warp-X. In *38th International Electric Propulsion Conference*, Toulouse, France, June 2024.

[172] R. Martin and J. Koo. Thermophysics universal research framework - infrastructure release. Air Force Research Laboratory (AFRL/RQRS), April 2015.

[173] Leland McInnes, John Healy, Nathaniel Saul, and Lukas Großberger. Umap: Uniform manifold approximation and projection. *Journal of Open Source Software*, 3(29):861, 2018.

[174] Robert L. Merlino. Understanding Langmuir probe current-voltage characteristics. *American Journal of Physics*, 75(12):1078–1085, 12 2007.

[175] Kai A Metzger, Peter Mortimer, and Hans-Joachim Wuensche. A fine-grained dataset and its efficient semantic segmentation for unstructured driving scenarios. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 7892–7899. IEEE, 2021.

[176] Sebastian Mika, Bernhard Schölkopf, Alex Smola, Klaus-Robert Müller, Matthias Scholz, and Gunnar Rätsch. Kernel pca and de-noising in feature spaces. *Advances in neural information processing systems*, 11, 1998.

[177] Ioannis G. Mikellides and Alejandro Lopez Ortega. Growth of the lower hybrid drift instability in the plume of a magnetically shielded Hall thruster. *Journal of Applied Physics*, 129(19):193301, 05 2021.

[178] 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.

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

[180] L Mirsky. Symmetric gauge functions and unitarily invariant norms. *Q. J. Math.*, 11(1):50–59, 1960.

[181] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[182] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[183] Kathleen R Murphy, Colin A Stedmon, Daniel Graeber, and Rasmus Bro. Fluorescence spectroscopy and multi-way techniques. parafac. *Analytical methods*, 5(23):6557–6566, 2013.

[184] Alexander Quinn Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models. In *International conference on machine learning*, pages 8162–8171. PMLR, 2021.

[185] Alexander Novikov, Dmitrii Podoprikhin, Anton Osokin, and Dmitry P Vetrov. Tensorizing neural networks. *Advances in neural information processing systems*, 28, 2015.

[186] Ryosuke Okuta, Yuya Unno, Daisuke Nishino, Shohei Hido, and Crissman Loomis. Cupy: A numpy-compatible library for nvidia gpu calculations. In *Proceedings of workshop on machine learning systems (LearningSys) in the thirty-first annual conference on neural information processing systems (NIPS)*, volume 6, 2017.

[187] Noriyasu Omata, Daiwa Satoh, Seiji Tsutsumi, Kaname Kawatsu, and Masaharu Abe. Model-based supervised sensor placement optimization to detect propellant leak in a liquid rocket engine. *Acta Astronautica*, 195:234–242, 2022.

[188] Francisco José Orts Gómez, Gloria Ortega López, Ernestas Filatovas, Olga Kurasova, and Gracia Ester Martın Garzón. Hyperspectral image classification using isomap with smacof. *Informatica*, 30(2):349–365, 2019.

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

[190] Ivan Oseledets and Eugene Tyrtyshnikov. Tt-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, 2010.

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

[192] 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.

[193] Wieslaw Ostachowicz, Rohan Soman, and Pawel Malinowski. Optimization of sensor placement for structural health monitoring: A review. *Structural Health Monitoring*, 18(3):963–988, 2019.

[194] Pentti Paatero. A weighted non-negative least squares algorithm for three-way parafac factor analysis. *Chemometrics and Intelligent Laboratory Systems*, 38(2):223–242, 1997.

[195] Pentti Paatero and Unto Tapper. Positive matrix factorization: A non-negative factor model with optimal utilization of error estimates of data values. *Environmetrics*, 5(2):111–126, 1994.

[196] Pentti Paatero, Unto Tapper, Pasi Aalto, and Markku Kulmala. Matrix factorization methods for analysing diffusion battery data. *Journal of Aerosol Science*, 22:S273–S276, 1991.

[197] Jyothish Pari, Nur Muhammad, Sridhar Pandian Arunachalam, Lerrel Pinto, et al. The surprising effectiveness of representation learning for visual imitation. *Robotics: Science and Systems XVIII*, 2021.

[198] Simone Parisi, Aravind Rajeswaran, Senthil Purushwalkam, and Abhinav Gupta. The unsurprising effectiveness of pre-trained vision models for control. In *international conference on machine learning*, pages 17359–17371. PMLR, 2022.

[199] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[200] J. Perales-Díaz, A. Domínguez-Vázquez, P. Fajardo, E. Ahedo, F. Faraji, M. Reza, and T. Andreussi. Hybrid plasma simulations of a magnetically shielded hall thruster. *Journal of Applied Physics*, 131(10):103302, 03 2022.

[201] David Perez-Garcia, Frank Verstraete, Michael M Wolf, and J Ignacio Cirac. Matrix product state representations. *arXiv preprint quant-ph/0608197*, 2006.

[202] Marilena Pillati, Cinzia Viroli, et al. Supervised locally linear embedding for classification: an application to gene expression data analysis. In *Proceedings of 29th Annual Conference of the German Classification Society (GfKl 2005)*, pages 15–18, 2005.

[203] Emmanuel Pintelas, Ioannis E Livieris, and Panagiotis E Pintelas. A convolutional autoencoder topology for classification in high-dimensional noisy image datasets. *Sensors*, 21(22):7731, 2021.

[204] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: Improving latent diffusion models for high-resolution image synthesis. *arXiv preprint arXiv:2307.01952*, 2023.

[205] Marsela Polic, Ivona Krajacic, Nathan Lepora, and Matko Orsag. Convolutional autoencoder for feature extraction in tactile sensing. *IEEE Robotics and Automation Letters*, 4(4):3671–3678, 2019.

[206] HS Prasantha, HL Shashidhara, and KN Balasubramanya Murthy. Image compression using svd. In *International conference on computational intelligence and multimedia applications (ICCIMA 2007)*, volume 3, pages 143–145. IEEE, 2007.

[207] Jun Qi, Chao-Han Huck Yang, Pin-Yu Chen, and Javier Tejedor. Exploiting low-rank tensor-train deep neural networks based on riemannian gradient descent with illustrations of speech processing. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 31:633–642, 2023.

[208] Antonin Raffin. Rl baselines zoo. https://github.com/araffin/rl-baselines-zoo, 2018.

[209] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research*, 22(268):1–8, 2021.

[210] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 1(2):3, 2022.

[211] C Radhakrishna Rao. Large sample tests of statistical hypotheses concerning several parameters with applications to problems of estimation. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 44, pages 50–57. Cambridge University Press, 1948.

[212] C Radhakrishna Rao. The utilization of multiple measurements in problems of biological classification. *Journal of the Royal Statistical Society. Series B (Methodological)*, 10(2):159–203, 1948.

[213] Ferran Reverter, Esteban Vegas, and Pedro Sánchez. Mining gene expression profiles: an integrated implementation of kernel principal component analysis and singular value decomposition. *Genomics, proteomics & bioinformatics*, 8(3):200–210, 2010.

[214] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[215] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, pages 234–241. Springer, 2015.

[216] Stéphane Ross and Drew Bagnell. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 661–668. JMLR Workshop and Conference Proceedings, 2010.

[217] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

[218] Subhayan Saha, Giovanni Barbarino, and Nicolas Gillis. Identifiability of nonnegative tucker decompositions–part i: Theory. *arXiv preprint arXiv:2505.12713*, 2025.

[219] Ruslan Salakhutdinov, Andriy Mnih, and Geoffrey Hinton. Restricted boltzmann machines for collaborative filtering. In *Proceedings of the 24th international conference on Machine learning*, pages 791–798, 2007.

[220] Hadi Salehi, Alex Gorodetsky, Roya Solhmirzaei, and Pengcheng Jiao. High-dimensional data analytics in civil engineering: A review on matrix and tensor decomposition. *Engineering Applications of Artificial Intelligence*, 125:106659, 2023.

[221] Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Nonlinear component analysis as a kernel eigenvalue problem. *Neural computation*, 10(5):1299–1319, 1998.

[222] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[223] Ulrich Schollwöck. Matrix product state algorithms: Dmrg, tebd and relatives. In *Strongly Correlated Systems: Numerical Methods*, pages 67–98. Springer, 2013.

[224] Pierre Sermanet, Corey Lynch, Jasmine Hsu, and Sergey Levine. Time-contrastive networks: Self-supervised learning from multi-view observation. In *2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 486–487. IEEE, 2017.

[225] Rutav Shah and Vikash Kumar. Rrl: Resnet as representation for reinforcement learning. *arXiv preprint arXiv:2107.03380*, 2021.

[226] Farial Shahnaz, Michael W Berry, V Paul Pauca, and Robert J Plemmons. Document clustering using nonnegative matrix factorization. *Information Processing & Management*, 42(2):373–386, 2006.

[227] Manish Sharma, Panos P Markopoulos, Eli Saber, M Salman Asif, and Ashley Prater-Bennette. Convolutional auto-encoder with tensor-train factorization. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 198–206, 2021.

[228] Hao Shen, Weikang Wan, and He Wang. Learning category-level generalizable object manipulation policy via generative adversarial self-imitation learning from demonstrations. *IEEE Robotics and Automation Letters*, 7(4):11166–11173, 2022.

[229] Lei Shi, Aryya Gangopadhyay, and Vandana P Janeja. Stensr: Spatio-temporal tensor streams for anomaly detection and pattern discovery. *Knowledge and Information Systems*, 43(2):333–353, 2015.

[230] Nicholas D Sidiropoulos, Lieven De Lathauwer, Xiao Fu, Kejun Huang, Evangelos E Papalexakis, and Christos Faloutsos. Tensor decomposition for signal processing and machine learning. *IEEE Transactions on signal processing*, 65(13):3551–3582, 2017.

[231] Jaqueline Silveira, Germain García, Afonso Paiva, Marcelo Nery, Sergio Adorno, and Luis Gustavo Nonato. Tensoranalyzer: identification of urban patterns in big cities using non-negative tensor factorization. *arXiv preprint arXiv:2210.02623*, 2022.

[232] 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.

[233] P Smolensky. Information processing in dynamical systems: foundations of harmony theory. In *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 194–281. The MIT Press, 1986.

[234] Andrews Sobral, Christopher G Baker, Thierry Bouwmans, and El-hadi Zahzah. Incremental and multi-feature tensor subspace learning applied for background modeling and subtraction. In *Image Analysis and Recognition: 11th International Conference, ICIAR 2014, Vilamoura, Portugal, October 22-24, 2014, Proceedings, Part I 11*, pages 94–103. Springer, 2014.

[235] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014.

[236] Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *arXiv preprint arXiv:2010.02502*, 2020.

[237] Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

[238] Vladislav Sovrasov. Ptflops: A flops counting tool for neural networks in pytorch framework. *GitHub repository*, 2018.

[239] Charles Spearman. General intelligence, objectively determined and measured. *The American Journal of Psychology*, 15(2):201–292, 1904.

[240] Suvrit Sra and Inderjit Dhillon. Generalized nonnegative matrix approximations with bregman divergences. *Advances in neural information processing systems*, 18, 2005.

[241] David W Stewart. The application and misapplication of factor analysis in marketing research. *Journal of marketing research*, 18(1):51–62, 1981.

[242] Adam Stooke, Kimin Lee, Pieter Abbeel, and Michael Laskin. Decoupling representation learning from reinforcement learning. In *International conference on machine learning*, pages 9870–9879. PMLR, 2021.

[243] Andrew M Stuart. Inverse problems: a bayesian perspective. *Acta numerica*, 19:451–559, 2010.

[244] 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.

[245] 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.

[246] Jimeng Sun, Spiros Papadimitriou, and S Yu Philip. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *Sixth International Conference on Data Mining (ICDM'06)*, pages 1076–1080. IEEE, 2006.

[247] Jimeng Sun, Dacheng Tao, and Christos Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 374–383, 2006.

[248] 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.

[249] Zhongkai Sun, Prathusha K Sarma, William Sethares, and Erik P Bucy. Multimodal sentiment analysis using deep canonical correlation analysis. *arXiv preprint arXiv:1907.08696*, 2019.

[250] F Taccogna, Savino Longo, M Capitelli, and R Schneider. Particle-in-cell simulation of stationary plasma thruster. *Contributions to Plasma Physics*, 47(8-9):635–656, 2007.

[251] 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.

[252] Rashish Tandon and Suvrit Sra. Sparse nonnegative matrix approximation: new formulations and algorithms. Technical report, Max Planck Institute, 2010.

[253] Albert Tarantola. *Inverse problem theory and methods for model parameter estimation*. SIAM, 2005.

[254] Joshua B Tenenbaum, Vin de Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[255] Bruce Thompson. Exploratory and confirmatory factor analysis: Understanding concepts and applications. *Washington, DC*, 10694(000):3, 2004.

[256] Fan Tian, Misha E Kilmer, Eric Miller, and Abani Patra. Tensor bm-decomposition for compression and analysis of video data. *arXiv preprint arXiv:2306.09201*, 2023.

[257] David Tskhakaya, Konstantin Matyash, Ralf Schneider, and Francesco Taccogna. The particle-in-cell method. *Contributions to Plasma Physics*, 47(8-9):563–594, 2007.

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

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

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

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

[262] Arash Vahdat, Karsten Kreis, and Jan Kautz. Score-based generative modeling in latent space. *Advances in neural information processing systems*, 34:11287–11302, 2021.

[263] Dani Valevski, Yaniv Leviathan, Moab Arar, and Shlomi Fruchter. Diffusion models are real-time game engines. *arXiv preprint arXiv:2408.14837*, 2024.

[264] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[265] Stefan Van der Walt, Johannes L Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D Warner, Neil Yager, Emmanuelle Gouillart, and Tony Yu. scikit-image: image processing in python. *PeerJ*, 2:e453, 2014.

[266] 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.

[267] M Alex O Vasilescu and Demetri Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *Computer Vision—ECCV 2002: 7th European Conference on Computer Vision Copenhagen, Denmark, May 28–31, 2002 Proceedings, Part I 7*, pages 447–460. Springer, 2002.

[268] Frank Verstraete and J Ignacio Cirac. Renormalization algorithms for quantum-many body systems in two and higher dimensions. *arXiv preprint cond-mat/0407066*, 2004.

[269] Frank Verstraete, Valentin Murg, and J Ignacio Cirac. Matrix product states, projected entangled pair states, and variational renormalization group methods for quantum spin systems. *Advances in physics*, 57(2):143–224, 2008.

[270] Guifré Vidal. Class of quantum many-body states that can be efficiently simulated. *Physical review letters*, 101(11):110501, 2008.

[271] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, pages 1096–1103, 2008.

[272] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *nature*, 575(7782):350–354, 2019.

[273] Udo Von Toussaint. Bayesian inference in physics. *Reviews of Modern Physics*, 83(3):943–999, 2011.

[274] Guoqing Wang, Qingzhu Ding, and Zhenyu Hou. Independent component analysis and its applications in signal processing for analytical chemistry. *TrAC Trends in Analytical Chemistry*, 27(4):368–376, 2008.

[275] Mengdi Wang, Jing Yu, Lijuan Niu, and Weidong Sun. Unsupervised feature extraction for hyperspectral images using combined low rank representation and locally linear embedding. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1428–1431. IEEE, 2017.

[276] Tianyu Wang, Nikhil Karnwal, and Nikolay Atanasov. Latent policies for adversarial imitation learning. *arXiv preprint arXiv:2206.11299*, 2022.

[277] Xiaokang Wang, Laurence T Yang, Yihao Wang, Lei Ren, and M Jamal Deen. Adtt: A highly efficient distributed tensor-train decomposition method for iiot big data. *IEEE Transactions on Industrial Informatics*, 17(3):1573–1582, 2020.

[278] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[279] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 1398–1402. IEEE, 2003.

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

[281] Philipp Wolf, Scott Moura, and Miroslav Krstic. On optimizing sensor placement for spatio-temporal temperature estimation in large battery packs. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 973–978. IEEE, 2012.

[282] Keyi Wu, Peng Chen, and Omar Ghattas. An efficient method for goal-oriented linear bayesian optimal experimental design: Application to optimal sensor placement. *arXiv preprint arXiv:2102.06627*, 2021.

[283] Keyi Wu, Peng Chen, and Omar Ghattas. An offline-online decomposition method for efficient linear bayesian goal-oriented optimal experimental design: Application to optimal sensor placement. *SIAM Journal on Scientific Computing*, 45(1):B57–B77, 2023.

[284] Zhong-Cheng Wu, Ting-Zhu Huang, Liang-Jian Deng, Hong-Xia Dou, and Deyu Meng. Tensor wheel decomposition and its tensor completion application. *Advances in Neural Information Processing Systems*, 35:27008–27020, 2022.

[285] Houping Xiao, Fei Wang, Fenglong Ma, and Jing Gao. eotd: An efficient online tucker decomposition for higher order tensors. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 1326–1331. IEEE, 2018.

[286] Wilson Yan, Yunzhi Zhang, Pieter Abbeel, and Aravind Srinivas. Videogpt: Video generation using vq-vae and transformers. *arXiv preprint arXiv:2104.10157*, 2021.

[287] Xuewen Yang, Yingru Liu, and Xin Wang. Reformer: The relational transformer for image captioning. In *Proceedings of the 30th ACM International Conference on Multimedia*, pages 5398–5406, 2022.

[288] Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *International conference on machine learning*, pages 3891–3900. PMLR, 2017.

[289] Yongqiang Yin, Rui Yuan, Yong Lv, Hongan Wu, Hewenxuan Li, and Weihang Zhu. Low-rank tensor train dynamic mode decomposition: An enhanced multivariate signal processing method for mechanical fault diagnosis. *IEEE Transactions on Instrumentation and Measurement*, 2025.

[290] Duwaraka Yoganathan, Sekhar Kondepudi, Balaji Kalluri, and Sumanth Manthapuri. Optimal sensor placement strategy for office buildings using clustering algorithms. *Energy and buildings*, 158:1206–1225, 2018.

[291] Jianbo Yu and Xingkang Zhou. One-dimensional residual convolutional autoencoder based feature learning for gearbox fault diagnosis. *IEEE Transactions on Industrial Informatics*, 16(10):6347–6358, 2020.

[292] Rose Yu, Stephan Zheng, Anima Anandkumar, and Yisong Yue. Long-term forecasting using higher order tensor rnns. *arXiv preprint arXiv:1711.00073*, 2017.

[293] Rose Yu, Stephan Zheng, Anima Anandkumar, and Yisong Yue. Long-term forecasting using tensor-train RNNs, 2018.

[294] Xiyu Yu, Tongliang Liu, Xinchao Wang, and Dacheng Tao. On compressing deep models by low rank and sparse decomposition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7370–7379, 2017.

[295] Zhecheng Yuan, Zhengrong Xue, Bo Yuan, Xueqian Wang, Yi Wu, Yang Gao, and Huazhe Xu. Pre-trained image encoder for generalizable visual reinforcement learning. *Advances in Neural Information Processing Systems*, 35:13022–13037, 2022.

[296] Xiang Yue, Zhen Wang, Jingong Huang, Srinivasan Parthasarathy, Soheil Moosavinasab, Yungui Huang, Simon M Lin, Wen Zhang, Ping Zhang, and Huan Sun. Graph embedding on biomedical networks: methods, applications and evaluations. *Bioinformatics*, 36(4):1241–1251, October 2019.

[297] Jaime Zabalza, Jinchang Ren, Jiangbin Zheng, Huimin Zhao, Chunmei Qing, Zhijing Yang, Peijun Du, and Stephen Marshall. Novel segmented stacked autoencoder for effective dimensionality reduction and feature extraction in hyperspectral imaging. *Neurocomputing*, 185:1–10, 2016.

[298] 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.

[299] 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.

[300] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[301] Chun-Hou Zheng, Bo Li, Lei Zhang, and Hong-Qiang Wang. Locally linear discriminant embedding for tumor classification. In *2008 International Conference on Intelligent Computing and Intelligent Systems*, volume 2, pages 1093–1100. IEEE, 2008.

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

[303] Wuyang Zhou, Yu-Bang Zheng, Qibin Zhao, and Danilo Mandic. Tensor star tensor decomposition and its applications to higher-order compression and completion. *arXiv preprint arXiv:2403.10481*, 2024.

[304] Yassine Zniyed, Remy Boyer, Andre LF De Almeida, and Gérard Favier. A tt-based hierarchical framework for decomposing high-order tensors. *SIAM Journal on Scientific Computing*, 42(2):A822–A848, 2020.