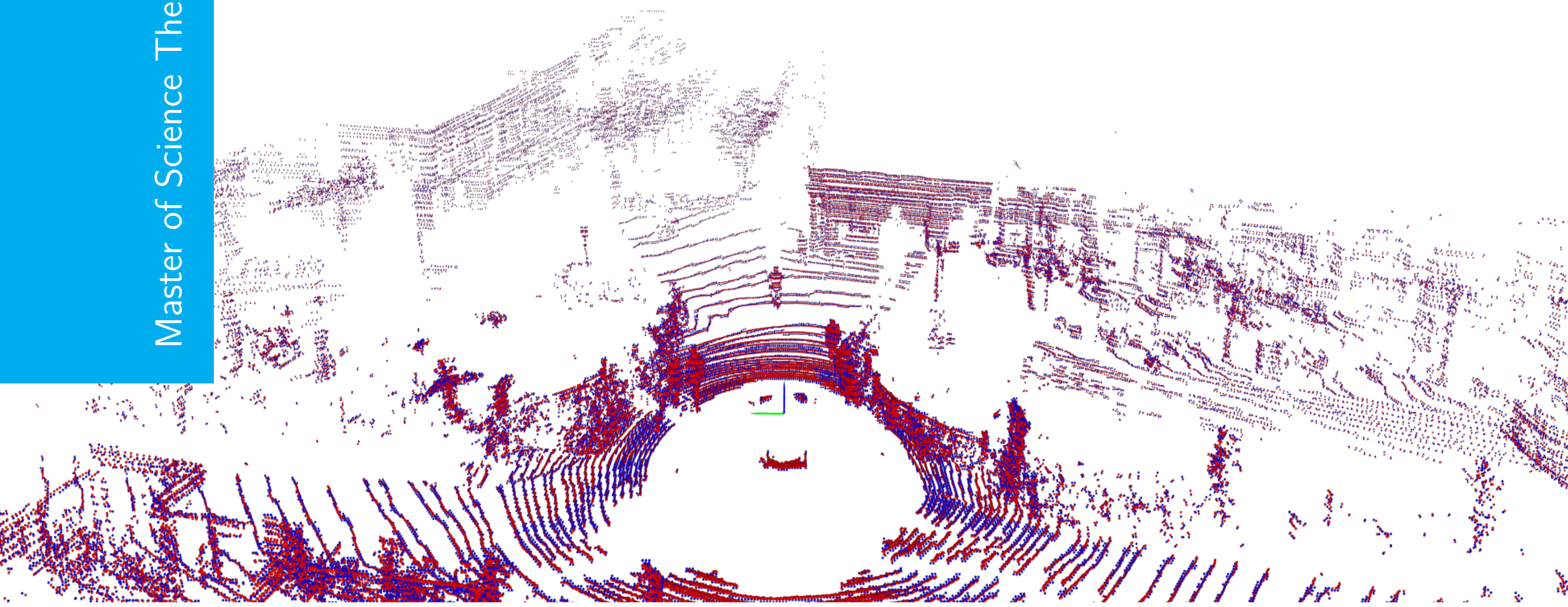
Point Cloud Compression for Automotive LiDAR using Tensor Decomposition Methods

C.V.M.M. Vorage

Master of Science Thesis



Delft University of Technology

Delft Center for Systems and Control

Cognitive Robotics

Point Cloud Compression for Automotive LiDAR using Tensor Decomposition Methods

Master of Science Thesis

For the degree of Master of Science in Systems and Control and Master of Science in Robotics at Delft University of Technology

C.V.M.M. Vorage

August 16, 2024

Faculty of Mechanical Engineering (ME) · Delft University of Technology

Delft University of Technology
Department of
Delft Center for Systems and Control (dcsc) and Cognitive Robotics (CoR)

The undersigned hereby certify that they have read and recommend to the Faculty of Mechanical Engineering (ME) for acceptance a thesis entitled

Point Cloud Compression for Automotive LiDAR using Tensor Decomposition Methods

by

C.V.M.M. Vorage

in partial fulfillment of the requirements for the degree of

Master of Science in Systems and Control and
Master of Science in Robotics

Dated: August 16, 2024

Supervisor(s): ______________________
Dr.ir. K. Batselier

____
Dr. J.F.P. Kooij

Reader(s): ______________________
Dr.ir. K. Batselier

____
Dr. J.F.P. Kooij

____
Dr. N.J. Myers

____
Dr. H. Caesar

Abstract

The training process of machine learning models for self-driving applications suffers from bottlenecks during loading and processing of LiDAR point clouds with large storage complexity. Many studies aim to remedy this problem from an implementation perspective by developing efficient data loading and processing pipelines. This study, on the other hand, explores an alternative approach by augmenting data representations to achieve lower storage complexity known as point cloud compression.

A broad analysis is presented on novel point cloud compression codecs using tensor decomposition methods. Several point cloud representations and tensor decomposition methods are considered over a range of hyperparameter choices and compression values. In order to assess the performance of the presented codecs: the compression rate, quality of the reconstruction, and time complexity is compared to the octree-based baseline model: TMC13.

Compared to the baseline model, the performance of the presented tensor decomposition-based codecs falls short. One of the presented codecs does notably outperform the others. This codec uses synthetic tensorization followed by sorting using z-location and decomposition using the TT-SVD algorithm. Sorting by z-value isolates the ground plane, which is a dominant low-rank feature, which can effectively be decomposed using the TT-SVD algorithm yielding adequate results. Visualizations of all codecs presented in this thesis can be viewed by scanning their respective QR codes, or centrally by clicking this link: https://data.4tu.nl/private_datasets/kIWHX8E3DwOvm0ToRFAYBacKdFIljtDMwk1p0H5lopo

Several limitations of the presented tensor decomposition-based codecs are: the omission of bitwise compression on the factor matrices, and the trade-off between bitwise precision and truncation due to tensor decomposition. Future work could improve in these areas along with considering the use of different heuristics and optimizing the tensor network topology.

Contents

Acknowledgements

I would like to thank my supervisors Dr.ir. K. Batselier and Dr. J.F.P. Kooij for their assistance during the writing of this thesis.

Delft, University of Technology
August 16, 2024

C.V.M.M. Vorage

Chapter 1

Introduction

In recent years, the automotive industry has made numerous advancements towards autonomous driving. Companies like Waymo [31], Cruise [30] and Motional [63] offer taxi rides using their full self-driving vehicles in multiple US states. These advancements in autonomous driving can partially be attributed to developments in sensor technologies such as Light Detection and Ranging (LiDAR). The LiDAR sensor maps the surroundings of the vehicle, enabling it to perceive the road and its users in order to safely navigate traffic. Figure 1-1 shows an example of a LiDAR scan (referred to as point cloud) from the View of Delft (VoD) dataset [65]. The image shows: the vehicle located in the lower-centre of the image, the reference frame of the LiDAR, and various road elements and users.

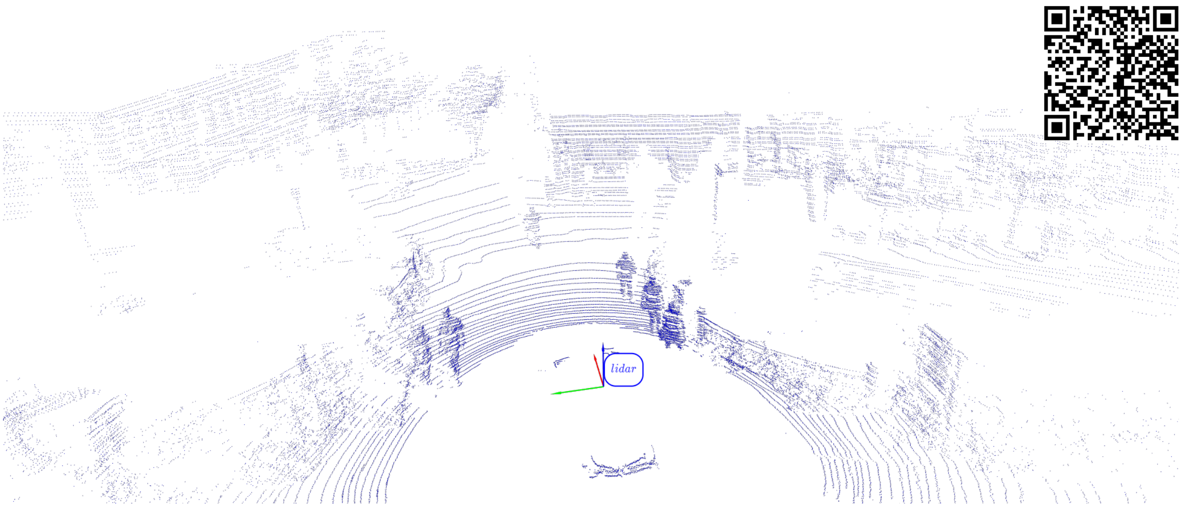


Figure 1-1: LiDAR point cloud of VoD dataset [65]. Scan QR code or click on link for 3D render.

As can be seen from Figure 1-1, LiDAR point clouds contain a vast number of points, which can easily amount up to ~100000 per scan. Unfortunately, the large volume of these point clouds pose significant challenges for efficiently training Machine Learning models for tasks such as object detection and semantic segmentation.

Machine learning models such as deep neural networks undergo a training process, which can roughly be divided into three consecutive steps, performed iteratively [45]. The first is to load a training sample in memory (fetch from SSD/HDD); The second is to process the sample in memory (decode, crop, rotate, normalize, etc.); And the third is to update model parameters (e.g. by gradient updates w.r.t the loss function). These operations are performed in parallel meaning that the overall training speed (sec/epoch) is determined by the throughput of the weakest link. This training process (often called data pipeline) is illustrated in Figure 1-2.

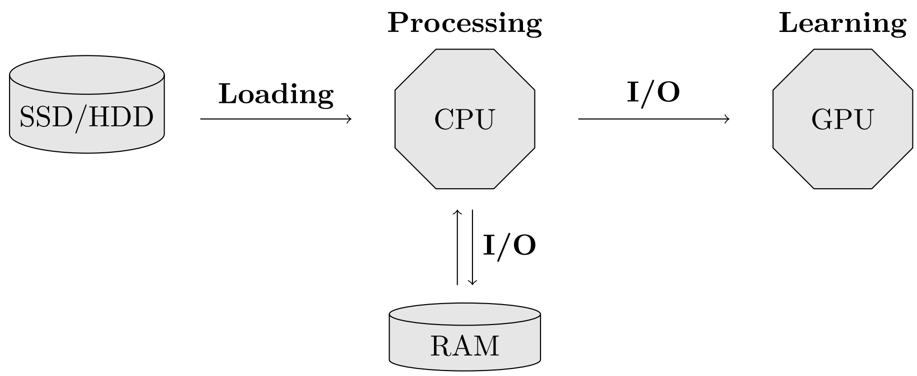


Figure 1-2: Data pipeline of general machine learning model training.

The illustration of the data pipeline shows that in order to improve training efficiency, it is essential to tackle the slowest step. Table 1-1 shows an overview of various studies identifying the bottleneck in machine learning data pipelines.

Publication	**Loading**	**Processing**	**Learning**
Kakaraparthy et al. (2019) [39]	✖	✖	
Zhang et al. (2020) [88]	✖		
Zolnouri et al. (2020) [92]	✖	✖	
Mohan et al. (2020) [58]	✖	✖	
Murray et al. (2021) [60]	✖		
Kuchnik et al. (2022) [42]	✖		
Isenko et al. (2022) [36]	✖	✖	
Leclerc et al. (2023) [45]		✖	

Table 1-1: Scientific publications identifying the bottleneck(s) in the machine learning data pipeline. Bottleneck(s) are marked with a cross (✖).

Of course, different tasks might prefer different hardware, and the specific hardware employed influences the performance on the three distinct steps. For example, using a setup which has enough working memory to load the entire dataset will result in avoiding fetch stalls, while having a surplus of CPU or GPU cores, will result in avoiding prep stalls or GPU stalls respectively [58]. With these considerations in mind, Table 1-1 should be viewed as a general indication of where the bottleneck lies, instead of a direct comparison between studies.

Nevertheless, Table 1-1 clearly shows that the two most important steps to improve are loading and processing, as opposed to learning. The studies mention that limitations are primarily caused by: I/O bottlenecks or available network bandwidth (**loading**), and large preprocessing overhead resulting from required data augmentation (**processing**), which results in unsaturated GPU's.

Many studies have been performed to find remedies to reduce these data pipeline bottlenecks. Mohan et al. (2021) and Kuchnick et al (2022) both developed software to diagnose a data pipeline and identify its bottleneck [42],[58]. Multiple studies developed data loader frameworks which enable faster training due to: exploiting parallelism, caching, asynchronous data transfer, data pre-loading, just-in-time compilation, and many more clever tricks [45],[58],[60],[86],[92]. Others focus on developing tools for finding the sweet spot between I/O-bound and processing-bound by allocating offline and online preprocessing steps between CPU and GPU [36]. An example of this is NVIDIA's data loading library (DALI) [17].

All of the methods mentioned above approach this problem from an implementation perspective, meaning that bottlenecks should be removed by designing better algorithms and more efficient parallel processing [39]. Another option is to approach this problem from a data perspective. Altering data representations in order to obtain lower storage complexity could potentially reduce the time spent on loading, processing, and I/O of data samples. Reconstruction of these compressed data samples would however need to occur before being able to train the model based on the samples. If this reconstruction would be possible on the GPU, workload could be shifted and the overall throughput of the data pipeline in Figure 1-2 could be optimized.

A first step however is to investigate how such a compressed representation can be found. Regarding automotive LiDAR applications this field of research is known as: **Point Cloud Compression (PCC)**.

1-1 Point Cloud Compression

The goal of Point Cloud Compression (PCC) is to find a representation of the data which yields lower storage complexity without significantly compromising the quality of the reconstruction.

This representation can be obtained in various ways such as finding patterns or exploiting redundancies in the data. A LiDAR point cloud consists of a set of P points each with F features. This unordered set of points ($\mathcal{V} = \{\mathbf{v}_i\}_{i=1}^{P}$) can be visualized in tabular format ($\mathcal{V} \in \mathbb{R}^{P \times F}$) like shown in Figure 1-3.

The goal is thus to find a data representation ($\tilde{\mathcal{V}}$), with a storage complexity ($\mathcal{O}$) lower than the original LiDAR data ($\mathcal{V}$), which can be formulated as:

$$\mathcal{O}(\tilde{\mathcal{V}}) < \mathcal{O}(\mathcal{V}) = P \times F. \tag{1-1}$$

This found data representation ($\tilde{\mathcal{V}}$) will have some approximation error with respect to the original LiDAR data. The approximation error can be used as a proxy for the quality of the reconstruction.

Current developments in PCC can roughly be divided into two main categories [69]. On the one hand, there are the conventional methods [5]. These methods are primarily based on efficient space-partitioning techniques such as KD-Trees [73],[48] and Octrees [22],[22]. On the other hand, there are deep learning-based methods [32]. These methods employ various machine learning architectures such as: Auto Encoders [27],[28],[70], and Recurrent Neural Networks [80].

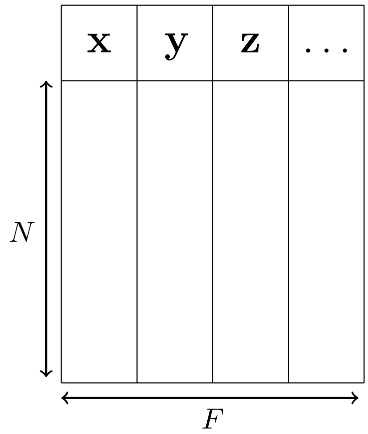


Figure 1-3: LiDAR data in tabular format.

A virtually unexplored approach is to employ **tensor decomposition** methods for PCC. Tensor decomposition methods are powerful tools able to decompose multi-dimensional objects (tensors) into multi-linear products of factor matrices/tensors. These factor matrices capture the latent space of the high-dimensional object, while greatly reducing the amount of storage needed. Reductions in storage complexity differ across methods, but frequently result in the reduction of exponential complexity in the dimensions to linear in the dimensions and quadratic in the rank of the decomposition. Exploiting this reduction might be a vital step to reach unprecedented compression gains with minimal reconstruction loss. On top of that, tensor decomposition methods rely on multi-linear products which could potentially be used to perform fast reconstruction on the GPU relieving the bottleneck in the machine learning data pipeline.

1-2 Research Objective

The main research question of this thesis can thus be phrased as:

Research Question

- Are **tensor decomposition** methods a competitive alternative for **Point Cloud Compression** of automotive LiDAR data?

Employing tensor decompositions on automotive LiDAR data is unprecedented, hence many different aspects of the compression pipeline present research opportunities. These aspects include: the data representation of the LiDAR point cloud, the type of tensor decomposition method employed, and the hyperparameters of the specific tensor decomposition method. This thesis may be considered as an exploratory study with the aim of investigating a broad amount of these aspects.

Regarding the type of tensor decomposition this thesis will explore three prevalent methods: the **Canonical Polyadic Decomposition** [40, 56], the **Tucker Decomposition** [14, 16, 81], and the **Tensor Train Decomposition** [64]. Section 2-3 will cover all relevant details on these specific decompositions and the algorithms employed to obtain them.

In general, a couple of research subquestions can be formulated that apply to all of the novel compression pipelines discussed in this thesis. These subquestions are:

General Research Subquestions

- What LiDAR data representations have an inherent **low-rank structure**, making it applicable for **tensor decomposition** methods?
- What is the performance of the tensor decomposition method in terms of: **reconstruction quality**, **amount of compression**, and **computational complexity**?
- How does the tensor decomposition method compare to the octree-based **baseline model** over a range of **hyperparameter** choices?

As mentioned one important aspect of applying tensor decomposition methods on automotive LiDAR data is the data representation of the point cloud. This thesis will explore three different data representations. These data representations are: **voxel-based** [1, 4], **synthetically tensorized** [61], and **geometry aware tensorized** (novel). Why these representations are chosen and what they entail will be discussed in Chapter 3. All of these representations have their own advantages and disadvantages. Finding out what these are lies central to the representation-specific research subquestions, which are formulated below:

Representation-Specific Research Subquestions

Voxel-Based

- How does the **voxelization** process of a point cloud affect the discretization loss and computational complexity of tensor decomposition methods?

Synthetic Tensorization

- How does the ordering of LiDAR points and the set of **synthetic tensorization** parameters affect the performance of tensor decomposition algorithms?
- What heuristics can be used to improve this ordering?

Geometry Aware Tensorization

- Does a **geometry aware** placement of LiDAR points in the tensor yield a representation which favours tensor decomposition methods?

In order to answer all of the research questions listed above several experiments are conducted. The methodology behind these experiments will be explained in Chapter 3 along with the baseline model. Chapter 4 will cover the experimental setup and present all the results of the experiments. Finally, Chapter 5 will conclude the thesis by answering the research questions posed in this section.

1-3 Relevance of Research

The merits of this research are threefold. Economically, there are big advantages for self-driving car manufacturers. Shorter training times, result in faster deployment of their products and services, yielding a competitive advantage over competitors. Additionally, improved efficiency leads to reduced computing resource and energy consumption, resulting in lower costs and better profit margins for businesses. Environmentally, the implications of this research are substantial. State-of-the-art machine learning models require a massive amount of resources in the form of energy consumption, often causing environmental pollution. OpenAI's GPT-3 has been estimated to have required 552.1 metric tonnes of CO_2 equivalent in training [19],[66]. This is equivalent to the energy consumption of 167 households in the Netherlands for a whole year [10]. Recognizing that these emissions belong solely to the training stage of one model emphasizes the potential impact this research can have on reducing CO_2 emissions within the industry. Lastly, there is the technological advantage. Accelerated training enables faster integration of new technologies in self-driving vehicles. This pushes advancements in the industry leading to safer and more reliable autonomous driving systems.

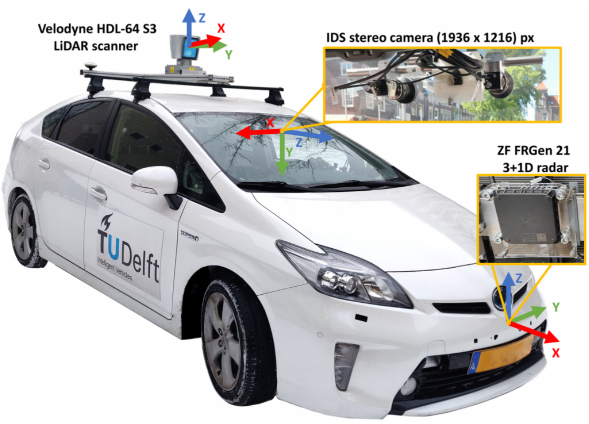
Chapter 2

Related Work and Theoretical Background

This chapter will introduce the related work on the subject of Point Cloud Compression (PCC) and present the theoretical background necessary to understand the tensor decompositions methods employed in Chapter 3 and Chapter 4. Section 2-1 will discuss the various point cloud representations used in automotive applications such as PCC, which will be discussed in Section 2-2. Section 2-3 will introduce the mathematical notation and theory regarding tensor decomposition methods.

2-1 Point Cloud Representations

Light Detection and Ranging (LiDAR) sensors such as the Velodyne displayed in Figure 2-1b are often often mounted on top of a vehicle like shown in Figure 2-1a.



(a) Toyota Prius 2013 with rotating 3D LiDAR sensor. Inteligent Vehicles, TU Delft [65].

(b) Velodyne HDL-64 S3 LiDAR [51].

Figure 2-1: Experimental setup of View of Delft (VoD) dataset [65].

The LiDAR sensor spins around its axis and emits light pulses at various elevation and azimuth angles. Through measuring the time-of-flight of these emitted light pulses the LiDAR is able to map its surroundings, and create a high-resolution 3D point cloud. The Cartesian coordinates of the points in the cloud are obtained using spherical coordinate transformation using the distance ρ, azimuth ϕ and elevation θ as displayed in Figure 2-2 and shown in Equation 2-1.

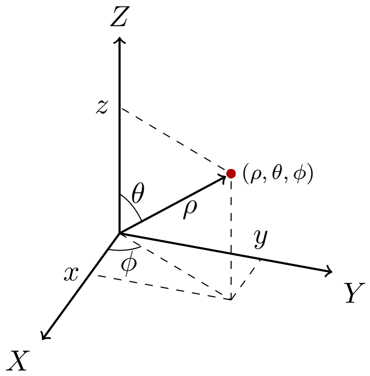


Figure 2-2: Spherical to Cartesian coordinate transformation.

$$
\begin{aligned}
x &= \rho \sin(\theta) \cos(\phi) \\
y &= \rho \sin(\theta) \sin(\phi) \\
z &= \rho \cos(\theta)
\end{aligned}
\tag{2-1}
$$

Tabular By default, LiDAR point clouds are often stored in tabular format, as is the case for the VoD and Karlsruhe Institute of Technology and Toyota Technological Institute (KITTI) datasets [65], [24]. Successively obtained points are stored contiguously in memory, resulting in a long table with as many rows as points in the cloud, and as many columns as attributes per point. Figure 2-3 shows an example of this type of data storage.

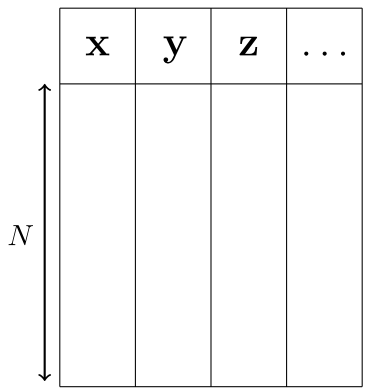


Figure 2-3: LiDAR data in tabular format.

Matricized Another way of representing automotive LiDAR point clouds is the range-view-based representation [1],[4]. This representation is obtained after "matricizing" a LiDAR point cloud. Matricizing a LiDAR point cloud is done using Cartesian to spherical coordinate projection [21]. Equation 2-2 shows this projection, which is the opposite of the projection in Equation 2-1.

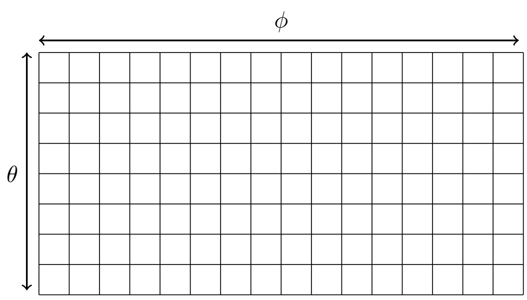


Figure 2-4: Matricization.

$$
\begin{aligned}
\rho &= \sqrt{x^2 + y^2 + z^2} \\
\theta &= \arccos\left(\frac{z}{x^2 + y^2 + z^2}\right) \\
\phi &= \operatorname{sign}(y) \arccos\left(\frac{x}{x^2 + y^2}\right)
\end{aligned}
\tag{2-2}
$$

The obtained projection can be viewed as a matrix (hence the name: matricization), where the rows correspond to a discretized range of the elevation angle (θ), and the columns to a discretized range of the azimuth angle (ϕ). The value within the matrix at a specific location (θ_i, ϕ_j) corresponds to the Euclidean distance (ρ) towards the LiDAR point. Figure 2-4 shows an example of the matricized view. Typical ranges of automotive LiDAR for the azimuth and elevation are $[-180°, 180°]$ and $[-25°, 4°]$ respectively [71]. During matricization multiple LiDAR points can be projected into the same cell. Handling this ambiguity is a design choice. One approach is to take the average of the point attributes.

Figure 2-5 shows an example of a matricized LiDAR point cloud from the VoD dataset [65]. The image depicts a cyclist moving towards the right in front of the vehicle. The image also shows many horizontal dark-blue lines. These lines are missing values in the matrix, caused by matricizing the point cloud at a too small angular resolution. Increasing the angular resolution can be done to remove the missing values, but this also reduces the quality of the picture, which will cause a weaker performance of any trained model.

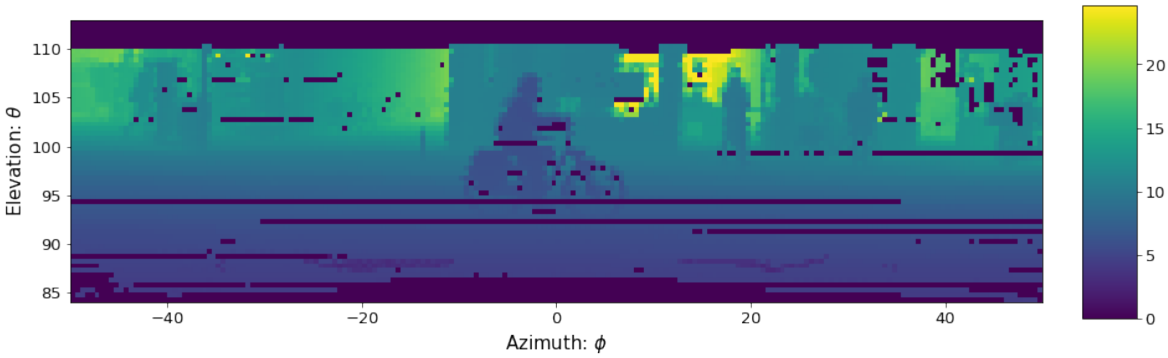


Figure 2-5: Matricized LiDAR point cloud of the VoD dataset [65]. The colorbar on the right denotes the distance of the matricized points with respect to the LiDAR reference frame.

Pillarized A popular representation is the Birds-Eye-View (BEV), obtained after performing pillarization of the LiDAR data [1]. Acclaimed papers such as PointPillars [44] use this technique to drastically improve inference time due to avoiding computationally expensive 3D convolutional layers present in VoxelNet [90].

Figure 2-6 shows an example of pillarization. LiDAR points are projected onto the (x, y)-plane, causing the information in the z-direction to be lost. In this example, equidistant spacing is chosen for the individual pillars in both directions, which is a design choice.

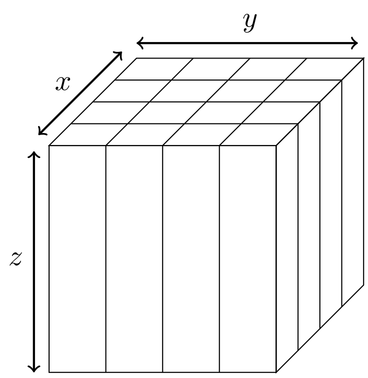


Figure 2-6: Pillarization.

Voxelized Another popular method for representing 3D-spaces is to use voxels [1],[4]. Voxels are three-dimensional boxes which are stacked on top of each other and build up the 3D-scene. Voxelization is the process of taking a point cloud in tabular form, and mapping each point to a specific voxel coordinate.

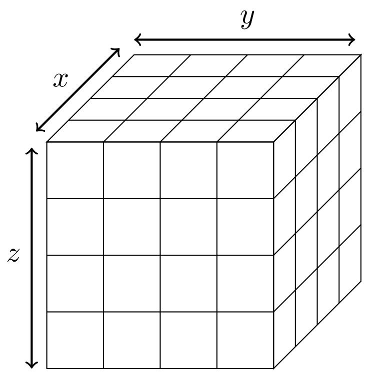


Figure 2-7: Voxelization.

Many acclaimed papers such as VoxelNet [90] use voxelization to preprocess a LiDAR point cloud at the start of their detection pipeline. Figure 2-7 shows an example of voxelizing a three-dimensional space into 64 seperate voxels. The example shows equidistant spacing for all three directions (x,y,z).

An advantage of voxel-based methods compared to projection-based methods (Matricization or Pillarization), is that they retain more information during the discretization process. Pillarization causes all information in the vertical direction to be lost, while matricization results in information loss due to the angular resolution. Because of this, voxel-based implementations are more versatile since they can be employed for BEV-detection tasks, but also for semantic segmentation tasks. They do however often face increased computational cost, due to the expensive 3D convolutions [74].

Octrees Another way of spatially partioning point clouds is to use octrees [5]. In an octree, each node represents a 3D cube. The root node represents the entire 3D region, and is divided into 8 smaller cubes called octants. Each of these octants can be subdivided again resulting into 8 more octants. This process is repeated recursively until the required level of detail is met, or a specific condition is met such as: all LiDAR points are placed in a separate octant.

Figure 2-8 shows an example of partioning 3D-space using octrees. The image shows that a depth of 3 is reached in terms of level of detail. The efficiency in representing 3D-spaces using this method lies in exploiting sparsity of the data. Often entire regions of LiDAR point clouds are empty (for example the sky), which means that using octrees a very large octant can be denoted using a 0, since it does not contain any points.

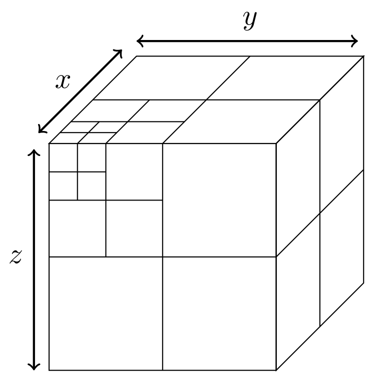


Figure 2-8: Octrees.

Octrees are primarily used for two types of tasks when employed using LiDAR data. The first task is semantic segmentation. Multiple methods use an octree structure to first recursively divide the scene into octants, which is then processed using various methods such as graph-based networks or plane-semengtation [83], [79], [77]. The second task for which octree representations are used is point cloud compression. These methods use octree structures to encode the point cloud in order to reduce spatial redundancy. Various methods exist such as: OctSqueeze [34] and OctAtention [22]. OctSqueeze uses a deep tree-structured entropy model, which uses context information to decrease the entropy of intermediate nodes. OctAtention is a multiple-contexts deep learning framework for lossless encoding which exploits similarities between sibling and ancestor nodes.

KD-Trees Another method to spatially partition 3D point clouds is to use KD-trees [5]. KD-trees are k-dimensional trees, used to partition a k-dimensional space. The construction of the KD-Tree consists of recursively partitioning 3D space, by placing hyperplanes along each axis. Often the hyperplanes are placed in such a way that the amount of points after each division is balanced across the partitions. Figure 2-9 shows an example of a three-dimensional KD-tree. The KD-tree is constructed by first partitioning the y-axis, followed by the z-axis, and finally the x-axis.

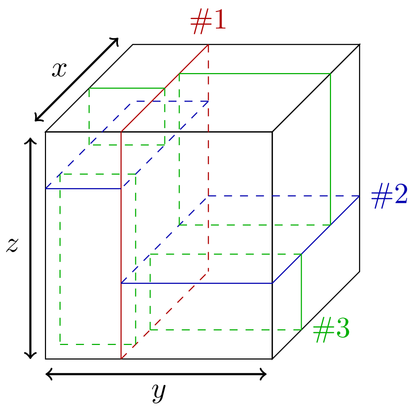


Figure 2-9: KD-trees.

Two different applications of kd-trees on LiDAR data are most common. These are: outlier detection and algorithmic efficiency. Shen et al. (2011) use kd-trees to detect outliers in airborne LiDAR data [73]. They used a combination of elevation histogram analysis to get rid of the obvious outliers, and kd-tree partioning to filter points based on their distance to k-neighbouring points. A similar work by Li et al. (2011) uses a histogram of elevation scales and a multilevel segmentation algorithm to filter outliers in airborne LiDAR data [48].

Two different works by Choi et al. (2012) and Li et al. (2016) employed KD-tree structures to accelerate the Iterative Closest Point (ICP) algorithm, which can be used for reconstructing of 3D surfaces, path planning or localization [13],[50]. Other works employed KD-trees for efficiently processing LiDAR data [7] or data management for visualisation of LiDAR scans [54].

Wavelet An alternative method is to apply a wavelet transform on the LiDAR data [6]. Each point attribute $(x, y, z, \text{reflectance}, \text{etc.})$ is considered as a 1D vector and independently wavelet transformed. The wavelet transform decomposes the signal into different frequency components and calculates the wavelet coefficients, denoting the individual contribution of those components. Often there is a strong correlation between successively obtained points, which generate small and similar wavelet coefficients [6]. Those can either be efficiently compressed or discarded, due to having little impact on the reconstruction performance [84].

Im et al (2010) propose using a Haar wavelet transform on the LiDAR point attributes [38], which is shown in Figure 2-10. Their approach consists of three steps. First the signals are decomposed using Haar Wavelet transform. Second, the wavelet coefficients which are below a prespecified threshold value are zeroed. Third, the signals are reconstructed using the wavelet coefficients. A large compression ratio was achieved, with minimal reconstruction error, due to the strong correlation between successively obtained points.

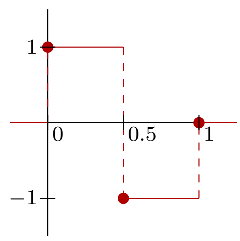


Figure 2-10: The Haar Wavelet.

Table 2-1 shows an overview of the previously mentioned point cloud representations and some noteworthy publications employing this representation.

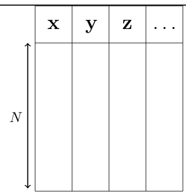
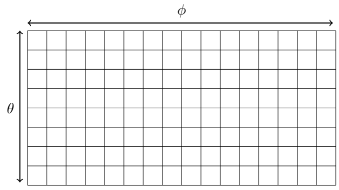
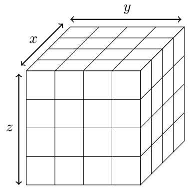
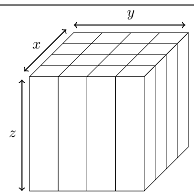
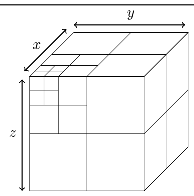
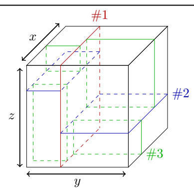
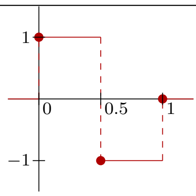
	Format	Publication
Tabular		PointNet [68]
Matricization		TensorMap [71]
Voxelization		VoxelNet [90] CenterPoint [87] VoxelNeXt [11] FSTR [20] LINK [55]
Pillarization		PointPillars [44] CenterPoint [87]
Octrees		OctSqueeze [34] OctAttention [22] Ainala et al. (2016) [2] Schwarz et al. (2019) [72]
KD-Trees		Gandoin et al. (2002) [23] Lien et al. (2010) [52] Shen et al. (2011) [73] Li et al. (2011) [48]
Wavelet		WALZ [84] Joon Im et al. (2010) [38]

Table 2-1: Overview of automotive LiDAR point cloud representations.

2-2 Point Cloud Compression Methods

In an effort to advance development of PCC the MPEG 3D Graphics Coding group put in a call for proposals in January 2017 [26]. This call for proposals eventually resulted in the definition of two research fields in 2020:

- Video-based Point Cloud Compression (V-PCC)
- Geometry-based Point Cloud Compression (G-PCC)

V-PCC is employed for clouds with a relative uniform distribution of points [26], and is generally linked to applications such as: augmented reality, virtual reality, and 3D video streaming. G-PCC is the technique employed for more spare distributions, and thus linked to Surface-PCC and LiDAR-PCC [25].

Over the years the MPEG Group has developed many different coder-decoder (codec) architectures for Surface-PCC and LiDAR-PCC. Codecs which outperformed all previous models were released to the public as the new baseline model under the names TMC1 and TMC3 for Surface-PCC and LiDAR-PCC respectively. Due to the similarity between Surface-PCC and LiDAR-PCC the MPEG Group decided to merge the two baseline models TMC1 and TMC3 into a single platform called TMC13, which is publicly available [59]. TMC13 will be employed as baseline model for this thesis.

The current works on point cloud compression can roughly be divided into two categories [69]. On the one hand, there are the conventional methods [5]. These methods generally employ space partitioning techniques such as Octrees [22],[22] and KD-Trees [73],[48]. The baseline model (TMC13) can be classified as a conventional model, since it employs octrees as space partitioning method. On the other hand, there are deep-learning based alternatives. These methods employ machine learning architectures such as: Auto-Encoders [27],[28],[70] and Recurrent Neural Networks [80].

A virtually unexplored research area is to employ tensor decomposition methods for point cloud compression. Across literature, only one noteworthy paper by Novikov & Oseledets has been identified to attempt this. Their paper describes a novel method employing low-rank Tensor Train (TT) decompositions to efficiently represent point clouds, which enables fast approximate nearest neighbour search [61]. Their method directly tensorizes the point cloud data in tabular form, by partitioning the amount of samples (N) into k dimensions as $N = N_1 \cdot N_2 \cdot \ldots \cdot N_k$. Equation 2-3 shows how matrix Y is reshaped (Definition 2.3) into tensor $\mathbf{Y}$, where the last dimension D denotes the amount of attributes per point.

$$\mathbf{Y} = \texttt{reshape}\left(Y, [N_1, \cdots, N_k, D]\right) \tag{2-3}$$

Although very interesting regarding this research, their work did get rejected for publication in the journal International Conference on Learning Representations (ICLR) 2024.

A possible drawback of employing tensor decompositions on automotive LiDAR data, which could be the reason it is relatively unexplored, is that they are generally not suited well for decomposing sparse tensors. This could limit performance on sparse representations, such as: voxel-based. There are however remedies for this possible limitation, namely there are various works on sparse tensor decomposition methods [41],[49],[67],[78]. The work on PCC using tensor decompositions by Novikov & Oseledets circumvents this drawback by direct tensorization of the raw point cloud data [61].

2-3 Multilinear Algebra & Tensor Decomposition Methods

Before we can start introducing the various tensor decompositions that will be explored, the underlying multilinear algebra needs to be explained. This section will start by introducing the notation and mathematical operations used from multilinear algebra and follow up with discussing several tensor decomposition methods.

2-3-1 Preliminaries

Throughout this thesis, data structures of different dimensions are discussed. The convention used is as follows; scalars are lowercase, vectors are bold lowercase, matrices are uppercase, and tensors are bold uppercase. Many of the mathematical definitions are obtained from Kolda & Bader's (2008) paper on Tensor Decompositions and Applications [40]. Apart from mathematical notation, tensor network diagrams will be used to illustrate various tensor operations. Table 2-2 shows a tensor network diagram for each type of data structure. In these diagrams nodes correspond to a mathematical object, whereas the outgoing edges (also referred to as modes or ways) correspond to the dimensions of the object. For example, the matrix X has two modes I_1 and I_2, which correspond to the length and width of the matrix respectively.

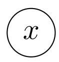
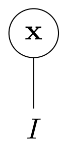
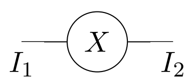
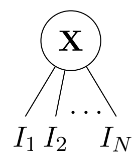
Name	Scalar	Vector	Matrix	Tensor
Mathematical Notation	$x \in \mathbb{R}$	$\mathbf{x} \in \mathbb{R}^{I}$	$X \in \mathbb{R}^{I_1 \times I_2}$	$\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$
Tensor Network Diagram				

Table 2-2: Mathematical and Tensor Network Diagram Notation.

Definition 2.1 denotes the convention used for multi-index notation, which applies to various other definitions that will follow.

Definition 2.1: Multi-Index

A multi-index $\overline{i_1 i_2 \dots i_n}$ can be used to refer to a single element in a tensor $\mathbf{X}$. When vectorizing a tensor $\mathbf{X}(i_1, i_2, \dots, i_N) \xrightarrow{\text{vec}} x_{i_{i_1, i_2, \dots, i_N}}$ the little-endian convention (reverse lexicographic ordering) [14] is used to determine the order of mapping the elements:

$$\overline{i_1 i_2 \dots i_n} = i_1 + (i_2 - 1) I_1 + (i_3 - 1) I_1 I_2 + \dots + (i_n - 1) I_1 \dots I_{n-1} \tag{2-4}$$

2-3-2 Basic Operations

There are several basic operations that can be performed on tensors. These operations are related to augmenting the data structure, and are widely used in many tensor decomposition algorithms [40],[64],[75]. These operations are: permutation, reshaping, and vectorization, which will be explained in Definition 2.2, Definition 2.3, and Definition 2.4 respectively.

Definition 2.2: Permute

Permuting a matrix/tensor changes the order of the modes. For a matrix this is equivalent to taking the transpose, while for tensors the order of the modes needs to explicitly stated.

Example

Permuting tensor $\mathbf{A} \in \mathbb{R}^{2\times3\times4\times5}$ with permutation operator Π and indices $\{2, 1, 4, 3\}$ results in:

$$\Pi_{\{2,1,4,3\}}(\mathbf{A}) = \tilde{\mathbf{A}} \in \mathbb{R}^{3\times2\times5\times4} \tag{2-5}$$

For notational clarity Matlab syntax will be used when performing permutation operations. For the above mentioned permutation this will be denoted as: $\texttt{permute}(\mathbf{A}, [2, 1, 4, 3])$.

Definition 2.3: Reshape

Reshaping a tensor rearranges the elements of the tensor according to newly specified modes. The product of the size of these newly specified modes needs to be equal to the number of elements in the original tensor.

Example

Tensor $\mathbf{A} \in \mathbb{R}^{2\times3\times4\times5}$ can be reshaped into $\tilde{\mathbf{A}} \in \mathbb{R}^{6\times2\times10}$, since the product of the size of their modes is equal to 120 for both tensors.

For notational clarity Matlab syntax will be used to denote a reshape operation. For the example above this will be formulated as: $\texttt{reshape}(\mathbf{A}, [6, 2, 10])$
Note: Reshaping a tensor is dependent on the multi-index convention (Definition 2.1).

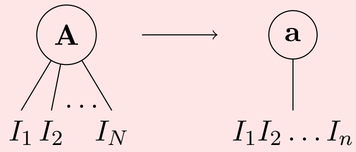
Definition 2.4: Vectorize

Vectorizing a tensor is done by iteratively grabbing the elements contiguously stored according to Definition 2.1 and putting them in a single vector.

Example

Tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ can be vectorized resulting in:

$$\text{vec}(\mathbf{A}) = \mathbf{a} \in \mathbb{R}^{I_1 I_2 \ldots I_N} \tag{2-6}$$



2-3-3 Multilinear Operations

Tensor decomposition methods are based on multilinear operations. One of them is the mode-n product, which is explained in Definition 2.6. This operation is used to multiply a tensor with a matrix across a specific mode. In order to successfully perform this multiplication, the tensor should be matricized along the corresponding mode. This matricization is defined in Definition 2.5.

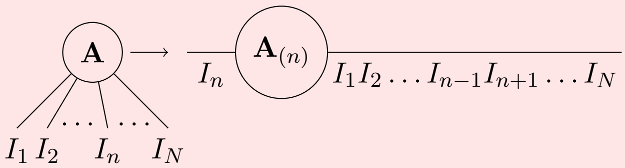
Definition 2.5: Matricize

Mode-n Matrication

A mode-n matricization matricizes a tensor, by taking I_n as its first dimension (length) and $I_1 I_2 \ldots I_{n-1} I_{n+1} \ldots I_N$ as its second dimension (width). This can be done by first permuting and then reshaping the tensor.

Example

Tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N}$ can be mode-n matricized, which results in: $\mathbf{A}_{(n)} \in \mathbb{R}^{I_n \times I_2 \ldots I_{n-1} I_{n+1} \ldots I_N}$

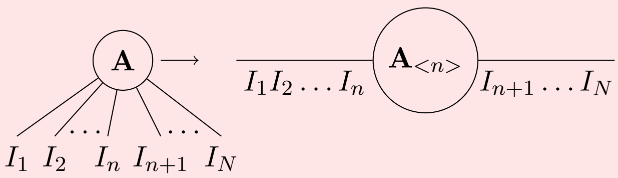


Mode-(1,...,n) Matricization

A mode-(1,...n) matricization matricizes a tensor by taking the product of mode 1 until n as its first dimension (length) and the product of the remaining modes as its second dimension (width).

Example

Tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N}$ can be mode-(1,..,n) matricized resulting in: $\mathbf{A}_{<n>} \in \mathbb{R}^{I_1 I_2 \ldots I_n \times I_{n+1} \ldots I_N}$



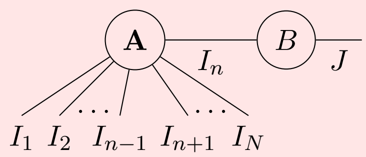
Definition 2.6: Mode-n Product

A mode-n product is the multiplication of a tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_n \times \ldots \times I_N}$ with a matrix $B \in \mathbb{R}^{J \times I_n}$ over mode n, resulting in tensor $\mathbf{C} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N}$ such that $c_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} = \sum_{i_n=1}^{I_n} a_{i_1,i_2,\ldots,i_N} \cdot b_{j,i_n} \; \forall \; \{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N\} \in \{\mathcal{I}_1 \times \ldots \times \mathcal{I}_{n-1} \times \mathcal{J} \times \mathcal{I}_{n+1} \times \ldots \times \mathcal{I}_N\}$.
It can be obtained by first performing a mode-n matricization (Definition 2.5), and subsequently performing a matrix multiplication. In tensor network diagram notation it can be visualized as a contraction over the corresponding modes.

Example

A mode-n product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_n \times \ldots \times I_N}$ and matrix $B \in \mathbb{R}^{J \times I_n}$ results in:

$$\mathbf{A} \times_n B \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_{n-1} \times J \times I_{n+1} \times \ldots \times I_N} \tag{2-7}$$



A series of mode-n products can be represented using the square bracket notation, where each factor matrix $\{B^{(1)},\ldots,B^{(N)}\}$ is multiplied across the corresponding mode of tensor $\mathbf{A}$.

$$\left[\mathbf{A}; B^{(1)},\ldots,B^{(N)}\right] \triangleq \mathbf{A} \times {}_1 B^{(1)} \times_2 \ldots \times_N B^{(N)} \tag{2-8}$$

Definition 2.7 shows the inner product. This is a scalar value which can be used to show the similarity between two tensors. Another operation is the outer product, which is denoted in Definition 2.8. The outer product is used in various tensor decomposition methods, and can be used to construct rank-1 tensors, which is a concept that will be explained in Definition 2.13.

Definition 2.7: Inner Product

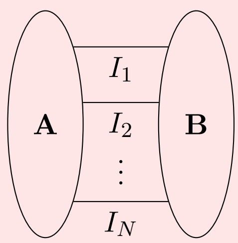
The inner product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ with tensor $\mathbf{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ is the contraction across all modes, resulting in the scalar c such that $c = \sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} a_{i_1,\ldots,i_N} \cdot b_{i_1,\ldots,i_N}$. The sizes of each mode need to be identical for both tensors. Equivalently, both tensors can be vectorized and multiplied accordingly.

Example

The inner product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ can be denoted as:

$$< \mathbf{A}, \mathbf{B} > = \sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} a_{i_1,\ldots,i_N} \cdot b_{i_1,\ldots,i_N} \tag{2-9}$$

$$< \mathbf{A}, \mathbf{B} > = \text{vec}(\mathbf{A})^T \text{vec}(\mathbf{B}) \in \mathbb{R}$$



Definition 2.8: Outer Product

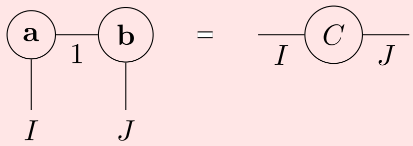
Vector Outer Product

The outer product of vector $\mathbf{a} \in \mathbb{R}^I$ and vector $\mathbf{b} \in \mathbb{R}^J$ is defined to be matrix $C \in \mathbb{R}^{I \times J}$ such that $c_{i,j} = a_i \cdot b_j \ \forall \ \{i,j\} \in \{\mathcal{I} \times \mathcal{J}\}$. The outer product of n vectors is an n-dimensional object.

Example

The outer product of vector $\mathbf{a} \in \mathbb{R}^I$ and vector $\mathbf{b} \in \mathbb{R}^J$ can be calculated as:

$$C = \mathbf{a} \circ \mathbf{b} = \mathbf{a}\mathbf{b}^T \in \mathbb{R}^{I \times J} \qquad (2\text{-}10)$$



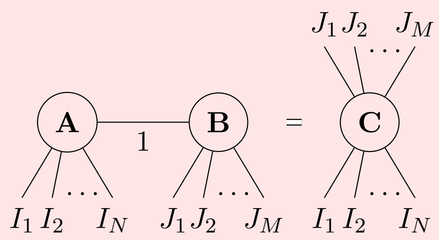
Tensor Outer Product

The outer product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{J_1 \times J_2 \times \ldots \times J_M}$ is defined to be tensor $\mathbf{C} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N \times J_1 \times J_2 \times \ldots \times J_M}$ such that $c_{i_1,i_2,\ldots,i_N,j_1,j_2,\ldots,j_M} = a_{i_1,i_2,\ldots,i_N} \cdot b_{j_1,j_2,\ldots,j_M} \ \forall \ \{i_1,i_2,\ldots,i_N,j_1,j_2,\ldots,j_M\} \in \{\mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \times \mathcal{I}_N \times \mathcal{J}_1 \times \mathcal{J}_2 \times \ldots \times \mathcal{J}_M\}$. In a tensor network diagram this can be illustrated using a rank-1 connection.

Example

The outer product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{J_1 \times J_2 \times \ldots \times J_M}$ can be calculated as:

$$\mathbf{C} = \mathbf{A} \circ \mathbf{B} = \sum_{r=1}^{1} a_{i_1,\ldots,i_N,r} \cdot b_{r,j_1,\ldots,j_M}$$
$$\mathbf{C} = \mathbf{A} \circ \mathbf{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N \times J_1 \times J_2 \times \ldots \times J_M} \qquad (2\text{-}11)$$



The Khatri-Rao, Kronecker, and Hadamard product are all important mathematical operations used in renowned tensor decomposition algorithms such as the: Canonical Polyadic - Alternating Least Squares (CP-ALS) [40] and Tensor Train Singular Value Decomposition (TT-SVD) [64]. These operations are explained in Definition 2.9, Definition 2.10, and Definition 2.11 respectively.

Definition 2.9: Khatri-Rao Product

The Khatri-Rao product of matrix $A \in \mathbb{R}^{I \times R}$ and matrix $B \in \mathbb{R}^{J \times R}$ is defined as the column-wise Kronecker product resulting in matrix $C \in \mathbb{R}^{JI \times R}$ such that $c_{\overline{j,i},r} = a_{i,r} \cdot b_{j,r} \ \forall \ \{\overline{j,i}, r\} \in \{\mathcal{I} \times \mathcal{J} \times \mathcal{R}\}$.

Example

The Khatri-Rao product of matrix $A \in \mathbb{R}^{I \times R}$ and matrix $B \in \mathbb{R}^{J \times R}$ can be calculated as:

$$C = A \odot B = (\mathbf{a}_{:,1} \otimes \mathbf{b}_{:,1}, \ldots, \mathbf{a}_{:,R} \otimes \mathbf{b}_{:,R}) \in \mathbb{R}^{JI \times R} \tag{2-12}$$

Definition 2.10: Kronecker Product

Matrix Kronecker Product

The Kronecker product of matrix $A \in \mathbb{R}^{I \times J}$ and matrix $B \in \mathbb{R}^{K \times L}$ is defined as the element-wise block multiplication resulting in matrix $C \in \mathbb{R}^{KI \times LJ}$ such that $c_{\overline{k,i},\overline{l,j}} = a_{i,j} \cdot b_{k,l} \ \forall \ \{\overline{k,i}, \overline{l,j}\} \in \{\mathcal{I} \times \mathcal{J} \times \mathcal{K} \times \mathcal{L}\}$.

Example

The Kronecker product of matrix $A \in \mathbb{R}^{I \times J}$ and matrix $B \in \mathbb{R}^{K \times L}$ can be calculated as:

$$C = A \otimes B = \begin{pmatrix} a_{1,1}B & \ldots & a_{1,J}B \\ \vdots & \ddots & \vdots \\ a_{I,1}B & \ldots & a_{I,J}B \end{pmatrix} \in \mathbb{R}^{KI \times LJ} \tag{2-13}$$

Tensor Kronecker Product

The Kronecker product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{J_1 \times J_2 \times \ldots J_N}$ is defined as tensor $\mathbf{C} \in \mathbb{R}^{J_I I_1 \times J_2 I_2 \times \ldots \times J_N I_N}$ such that $c_{\overline{j_1,i_1},\ldots,\overline{j_M,i_N}} = a_{i_1,\ldots,i_N} \cdot b_{j_1,\ldots,j_M} \ \forall \ \{\overline{j_1,i_1}, \ldots, \overline{j_M,i_N}\} \in \{\mathcal{I}_1 \times \ldots \times \mathcal{I}_N \times \mathcal{J}_1 \times \ldots \times \mathcal{J}_N\}$.

Example

The Kronecker product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{J_1 \times J_2 \times \ldots J_N}$ is denoted as:

$$\mathbf{C} = \mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{J_I I_1 \times J_2 I_2 \times \ldots \times J_N I_N} \tag{2-14}$$

Definition 2.11: Hadamard Product

The Hadamard product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ is defined as the elementwise product resulting in tensor $\mathbf{C} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ such that $c_{i_1,i_2,\ldots,i_N} = a_{i_1,i_2,\ldots,i_N} \cdot b_{i_1,i_2,\ldots,i_N} \ \forall \ \{i_1, i_2, \ldots, i_N\} \in \{\mathcal{I}_1 \times \mathcal{I}_2 \times \ldots \times \mathcal{I}_N\}$.

Example

The Hadamard product of tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and tensor $\mathbf{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ can be calculated as:

$$\mathbf{C} = \mathbf{A} \circledast \mathbf{B} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N} \tag{2-15}$$

The Frobenius norm is a metric often employed to define the (relative) error of an obtained tensor decomposition [64]. Definition 2.12 shows how the Frobenius norm can be calculated for any tensor.

Definition 2.12: Frobenius Norm

The Frobenius norm of a tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ is defined as the square root of the inner product with itself, which is denoted as: $\|\mathbf{A}\|_F = \sqrt{< \mathbf{A}, \mathbf{A} >}$.

Example

For any tensor $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, the Frobenius norm can be calculated as:

$$\begin{aligned} \|\mathbf{A}\|_F &= \sqrt{< \mathbf{A}, \mathbf{A} >} = \sqrt{\sum_{i_1=1}^{I_1} \cdots \sum_{i_N=1}^{I_N} a_{i_1,\ldots,i_N} \cdot b_{i_1,\ldots,i_N}} \\ \|\mathbf{A}\|_F &= \sqrt{\mathrm{vec}(\mathbf{A})^T \mathrm{vec}(\mathbf{A})} \in \mathbb{R} \end{aligned} \tag{2-16}$$

2-3-4 Notions of Rank

There are multiple notions of rank used in tensor decomposition methods. The Canonical Polyadic Decomposition (CPD) considers the extension of matrix rank to higher dimensions called tensor rank. The Multilinear Singular Value Decomposition (MLSVD) on the other hand employs the multilinear rank, which is the (matrix) rank of its mode-n matricizations.

Definition 2.13: Matrix/Tensor Rank

Matrix Rank

The rank of a matrix is defined as the maximum amount of linearly independent columns. Alternatively this can be viewed as the minimum amount of vector outer products needed to represent the matrix.

Example

Matrix $C \in \mathbb{R}^{2\times 2}$ can be reduced to row-echelon form, which shows only 1 linearly independent column, meaning $\text{rank}(C) = 1$. Alternatively, the matrix C can be represented as a single outer product between vectors $\mathbf{a}$ and $\mathbf{b}$, implying its rank is 1.

$$
\begin{aligned}
C &= \begin{pmatrix} 2 & 1 \\ -4 & -2 \end{pmatrix} \sim \begin{pmatrix} 2 & 1 \\ 0 & 0 \end{pmatrix} \Rightarrow \text{rank}(C) = 1 \\
C &= \begin{pmatrix} 2 & 1 \\ -4 & -2 \end{pmatrix} = \underbrace{\begin{pmatrix} 1 \\ -2 \end{pmatrix}}_{\mathbf{a}} \circ \underbrace{\begin{pmatrix} 2 \\ 1 \end{pmatrix}}_{\mathbf{b}} \Rightarrow \text{rank}(C) = 1
\end{aligned}
\tag{2-17}
$$

Tensor Rank

The concept of rank generalizes to the tensor case, where it corresponds to the minimum amount of vector outer products that are needed to represent the tensor.

Example

Tensor $\mathbf{D} \in \mathbb{R}^{2\times 2\times 2}$ is of rank 1, since it can be written as a single outer product between vectors $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$.

$$
\mathbf{D} = \begin{matrix} & \begin{pmatrix} 4 & 6 \\ 4 & 6 \end{pmatrix} \\ \begin{pmatrix} 2 & 3 \\ 2 & 3 \end{pmatrix} & \end{matrix} = \underbrace{\begin{pmatrix} 1 \\ 1 \end{pmatrix}}_{\mathbf{a}} \circ \underbrace{\begin{pmatrix} 2 \\ 3 \end{pmatrix}}_{\mathbf{b}} \circ \underbrace{\begin{pmatrix} 1 \\ 2 \end{pmatrix}}_{\mathbf{c}} \Rightarrow \text{rank}(\mathbf{D}) = 1 \tag{2-18}
$$

Definition 2.14: Multilinear Rank

The multilinear rank of tensor $\mathbf{X} \in \mathbb{R}^{I_1\times I_2\times\ldots\times I_N}$ is an N-tuple consisting of the dimensions of the vector space spanned by its mode-n fibers. In other words, it is an tuple consisting of the ranks of its mode-n matricizations.

$$
\text{rank}_{ML}(\mathbf{X}) = \left\{\text{rank}\left(\mathbf{X}_{(1)}\right), \text{rank}\left(\mathbf{X}_{(2)}\right), \ldots, \text{rank}\left(\mathbf{X}_{(N)}\right)\right\} \tag{2-19}
$$

2-3-5 Overview of Mathematical Notation

Table 2-3 shows an overview of the mathematical notation introduced in this chapter, which will be used to define three tensor decomposition methods. These are: the CPD, the MLSVD, and the TT-SVD.

Notation	Definition
a	Scalar
$\mathbf{a}$	Vector
A	Matrix
$\mathbf{A}$	Tensor
$\mathbf{A}(i_1, i_2, \ldots, i_D) = a_{i_1,i_2,\ldots,i_D}$	Element of tensor $\mathbf{A}$
$\mathbf{A}_{(n)}$	Mode-n matricization of tensor $\mathbf{A}$
$\mathbf{A}_{<n>}$	Mode-(1,...,n) matricization of tensor $\mathbf{A}$
$\mathbf{A} \times_n B$	Mode-n product of tensor $\mathbf{A}$ with matrix B
$\left[\mathbf{A}; B^{(1)}, \ldots, B^{(N)}\right]$	Sequence of mode-n products on tensor $\mathbf{A}$ using factor matrices $\{B^{(1)}, \ldots, B^{(N)}\}$
$< \mathbf{A}, \mathbf{B} >$	Inner product of tensors $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \circ \mathbf{B}$	Outer product of tensors $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \otimes \mathbf{B}$	Kronecker product of tensors $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \odot \mathbf{B}$	Khatri-Rao product of tensors $\mathbf{A}$ and $\mathbf{B}$
$\mathbf{A} \circledast \mathbf{B}$	Hadamard product of tensors $\mathbf{A}$ and $\mathbf{B}$
$\\|\mathbf{A}\\|_F$	Frobenius norm of tensor $\mathbf{A}$
i_n	Index of dimension n
I_n	Size of dimension n
$\mathcal{I}_n$	Set containing all indices i_n in I_n
$\text{vec}(\mathcal{A})$	Vectorization of tensor $\mathcal{A}$
$\texttt{permute}(\mathbf{A}, [i_1, \ldots, i_n])$	Permuting tensor $\mathbf{A}$ using indices $[i_1, \ldots, i_n]$
$\texttt{reshape}(\mathbf{A}, [I_1, \ldots, I_n])$	Reshaping tensor $\mathbf{A}$ using dimensions $[I_1, \ldots, I_n]$
$\texttt{rank}(\mathbf{A})$	Rank of tensor $\mathbf{A}$
$\texttt{rank}_{ML}(\mathbf{A})$	Multilinear rank of tensor $\mathbf{A}$
$\texttt{diag}(\mathbf{c})$	Matrix with elements of vector $\mathbf{c}$ on its diagonal
$\texttt{diag}(\mathbf{c}, N)$	N-dimensional tensor with elements of vector $\mathbf{c}$ on its superdiagonal
$\texttt{svd}(A)$	Singular Value Decomposition of matrix A
$\texttt{svd}_\delta(A)$	δ-Truncated Singular Value Decomposition of matrix A
$\mathcal{O}(\mathbf{A})$	Storage complexity of tensor $\mathbf{A}$

Table 2-3: Mathematical notation used in tensor decomposition methods.

2-3-6 Canonical Polyadic Decomposition

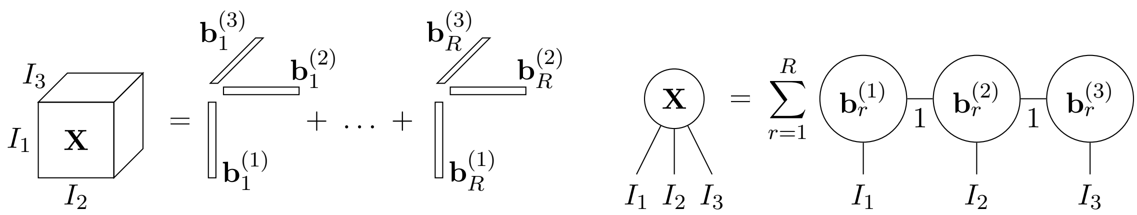
The first tensor decomposition that will be introduced is the CPD, which is denoted in Definition 2.15.

Definition 2.15: Canonical Polyadic Decomposition (CPD)

Any tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ can be represented by a finite sum of vector outer products, where the length of the vectors correspond to the dimensions of the tensor. In mathematical terms this can be denoted as:

$$\mathbf{X} = \sum_{r=1}^{R} \boldsymbol{b}_r^{(1)} \circ \boldsymbol{b}_r^{(2)} \circ \ldots \circ \boldsymbol{b}_r^{(N)} \tag{2-20}$$

For a three-dimensional case, this can be illustrated using the figures below. Figure 2-11a shows a 3D visualization of the summation of rank-1 terms, while Figure 2-11b shows the corresponding tensor network diagram.



(a) 3D visualization of CPD.

(b) Tensor network diagram of CPD.

Figure 2-11: Graphical representations of Canonical Polyadic Decomposition.

The smallest amount of vector outer products that are needed to represent a tensor exactly is called the rank R (Definition 2.13). Finding this rank is however not an easy task, it may be considered an NP-hard problem [35].

The CPD is a representation consisting of the summation of the minimal amount of rank-1 terms. This summation may be considered unique (up to scaling and permutation of indices) if the individual R components are unique [40].

An alternative representation to Equation 2-20 is shown in Equation 2-21, where the vectors $[\mathbf{b}_r^1, \ldots, \mathbf{b}_r^N]$ are normalized to unit length, and the norms are stored in scalar λ_r.

$$\mathbf{X} = \sum_{r=1}^{R} \lambda_r \boldsymbol{b}_r^{(1)} \circ \boldsymbol{b}_r^{(2)} \circ \ldots \circ \boldsymbol{b}_r^{(N)} \tag{2-21}$$

In virtually any real life application noise will be present in the system, meaning an exact computation is not possible or might not exist. Because of this, the problem of finding the CPD of a given tensor should be written as an optimization problem. Equation 2-22 shows the objective function of the optimization problem for an N-dimensional tensor. The goal is to minimize this objective function, which denotes the squared frobenius norm of the error between the original ($\mathbf{X}$) and approximated ($\tilde{\mathbf{X}}$) tensor.

$$
\begin{aligned}
J\left(\lambda_r, \boldsymbol{b}_r^{(1)}, \boldsymbol{b}_r^{(2)}, \dots, \boldsymbol{b}_r^{(N)}\right) &= \left\| \mathbf{X} - \sum_{r=1}^{R} \lambda_r \boldsymbol{b}_r^{(1)} \circ \boldsymbol{b}_r^{(2)} \circ \dots \circ \boldsymbol{b}_r^{(N)} \right\|_F^2 \\
J\left(\boldsymbol{\Lambda}, B^{(1)}, B^{(2)}, \dots, B^{(N)}\right) &= \left\| \mathbf{X} - \underbrace{[\boldsymbol{\Lambda}; B^{(1)}, B^{(2)}, \dots, B^{(N)}]}_{\tilde{\mathbf{X}}} \right\|_F^2
\end{aligned}
\tag{2-22}
$$

The equation shows the factor matrices $[B^{(1)}, B^{(2)}, \dots, B^{(N)}]$ and superdiagonal core tensor $\boldsymbol{\Lambda}$, which are the decision variables of the optimization problem. The superdiagonal core tensor $\boldsymbol{\Lambda}$ stores the norms if the factor matrices have been normalized to unit length, otherwise it will be a superdiagonal identity tensor.

The CPD reduces the storage compexity of tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ from being exponential in the amount of modes N, to being linear in the rank R, as can be seen in Equation 2-23.

$$
\begin{aligned}
\mathcal{O}(\mathbf{X}) &= \prod_{n=1}^{N} I_n \\
\mathcal{O}(\texttt{CPD}(\mathbf{X})) &= R \sum_{n=1}^{N} I_n
\end{aligned}
\tag{2-23}
$$

Canonical Polyadic - Alternating Least Squares (CP-ALS)

There are many ways to solve Equation 2-22, however a popular method is to use the Canonical Polyadic - Alternating Least Squares algorithm shown in Algorithm 1 [40],[56]. This algorithm takes as input a tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, initialization method for factor matrices $\mathbf{B} = [B^{(1)}, \dots, B^{(N)}]$, rank of the decomposition R, allowed relative error ϵ, and maximum amount of iterations i_{max}. It returns the vector λ and factor matrices $\mathbf{B}$ that make up the approximation of tensor $\mathbf{X}$. The approximation $\tilde{\mathbf{X}}$ abides to the defined allowed relative error (ϵ) if the algorithm converged within the maximum amount of steps (i_{max}).

The algorithm makes use of the identity in Equation 2-24, which states that the mode-n matricization of a CPD can be expressed in terms of its factor matrices $[B^{(1)}, B^{(2)}, \dots, B^{(N)}]$ and core Λ in matrix form.

$$
\mathbf{X}_{(n)} = B^{(n)} \Lambda \left(B^{(N)} \odot \dots \odot B^{(n+1)} \odot B^{(n-1)} \odot \dots \odot B^{(1)} \right)^T \tag{2-24}
$$

Substituting this identity into the optimization problem, results in N equivalent optimization problems with a different structure. Using the mode-1 matricization, the first optimization problem can be obtained, which is shown in Equation 2-25.

$$
\min_{B^{(1)}, \dots, B^{(N)}} \| \mathbf{X}_{(1)} - B^{(1)} (B^{(N)} \odot B^{(N-1)} \odot \cdots \odot B^{(2)})^T \| \tag{2-25}
$$

This optimization does not yield an easy overall solution for all factor matrices $[B^{(1)}, B^{(2)}, \dots, B^{(N)}]$, but it does if additional constraints are imposed. Fixing all factor matrices except for $B^{(1)}$ turns the problem into least squares form [40].

$$\min_{B^{(1)}} \|\mathbf{X}_{(1)} - B^{(1)}(B^{(N)} \odot B^{(N-1)} \odot \cdots \odot B^{(2)})^T\| \tag{2-26}$$

The least squares solution can then be obtained like shown in Equation 2-27 [82], where † denotes the Moore–Penrose pseudo-inverse.

$$\overset{*}{B}^{(1)} = \mathbf{X}_{(1)} \left((B^{(N)} \odot B^{(N-1)} \odot \cdots \odot B^{(2)})^T \right)^{\dagger} \tag{2-27}$$

The new estimate for factor matrix $B^{(1)}$ can then be used when solving the next least squares problem, where the optimization variable will be the second factor matrix $B^{(2)}$, like shown in Equation 2-28

$$\min_{B^{(2)}} \|\mathbf{X}_{(2)} - B^{(2)}(B^{(N)} \odot B^{(N-1)} \odot \cdots \odot B^{(3)} \odot B^{(1)})^T\| \tag{2-28}$$

This process is repeated for all N problem formulations, yielding a solution for all factor matrices $[B^{(1)}, \ldots, B^{(N)}]$. The factor matrices are then used to create the approximation of the mode-N matricization $\tilde{\mathbf{X}}_{(N)}$, which is used to compute the relative error of the decomposition in Frobenius norm sense. If the error is below the defined acceptable threshold the algorithm stops. Otherwise, the iterative procedure continues for a specified amount of maximum iterations.

Algorithm 1 Canonical Polyadic - Alternating Least Squares (CP-ALS) [40],[56]

Require: Tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, rank of the decomposition R, allowed relative error ϵ, maximum iterations i_{max}

- $\mathbf{B} \leftarrow$ Initialize factor matrices $B^{(n)} \in \mathbb{R}^{I_n \times R}$
- **while** $e > \epsilon$ and $i < i_{max}$ **do**
  - **for** $n = 1$ to N **do**
    - $V \leftarrow B^{(1)T}B^{(1)} \circledast \cdots \circledast B^{(n-1)T}B^{(n-1)} \odot B^{(n+1)T}B^{(n+1)} \circledast \cdots \circledast B^{(N)^T}B^{(N)}$
    - $B^{(n)} \leftarrow \mathbf{X}_{(n)}(B^{(N)} \odot \cdots \odot B^{(n+1)} \odot B^{(n-1)} \odot \cdots \odot B^{(1)})V^{\dagger}$
    - **for** $r = 1$ to R **do**
      - $\lambda_r \leftarrow \|B^{(n)}_{:,r}\|_F$
      - $B^{(n)}_{:,r} \leftarrow \frac{B^{(n)}_{:,r}}{\mathbf{c}_r}$
    - **end for**
  - **end for**
  - $\tilde{\mathbf{X}}_{(\mathbf{N})} \leftarrow \mathrm{diag}(\lambda_r)B^{(N)}(B^{(N-1)} \odot \cdots \odot B^{(1)})^T$
  - $e \leftarrow \frac{\|\mathbf{x}_{(\mathbf{N})} - \tilde{\mathbf{X}}_{(\mathbf{N})}\|_F}{\|\mathbf{X}_{(\mathbf{N})}\|_F}$
- **end while**
- **return** λ, $\mathbf{B}$

After using the algorithm, the approximation of tensor $\mathbf{X}$ can be obtained by mode-n products of the factor matrices $\mathbf{B} = [B^{(1)}, \ldots, B^{(N)}]$ with the constructed core $\mathbf{C}$ like shown in Equation 2-29.

$$\tilde{\mathbf{X}} = \text{diag}(\mathbf{c}, N) \times_1 B^{(1)} \times_2 \ldots \times_N B^{(N)} = \left[\mathbf{C}; B^{(1)}, \ldots, B^{(N)}\right] \tag{2-29}$$

The factor matrices $\mathbf{B} = [B^{(1)}, \ldots, B^{(N)}]$ can be initialized in various ways. Common practice is to initialize them by drawing values from a normal distribution with unit variance [3]. In mathematical terms this can be denoted like shown in Equation 2-30.

$$b_{i,j} \sim \mathcal{N}(0, 1) \; \forall \; \{i, j\} \in I \times J, \quad B \in \mathbb{R}^{I \times J} \tag{2-30}$$

In order to reduce the chance of getting stuck in a local minima, the CP-ALS is often performed multiple times from different (random) initialization points [29]. Harshman & Lundy (1994) suggest performing 6 separate random initializations drawn from the same distribution. If all 6 solutions agree, the probability of finding a different solution with an equivalent or lower relative error is smaller than 0.05, when drawing a new sample from the same distribution.

2-3-7 Multilinear Singular Value Decomposition (MLSVD)

The second tensor decomposition method that will be discussed is the Multilinear Singular Value Decomposition (MLSVD). This decomposition is an extension of the Singular Value Decomposition (SVD) for matrices to higher dimensions. The following definitions, theorems, and properties lay the groundwork upon which the MLSVD is built.

Definition 2.16: Singular Value Decomposition (SVD) [76]

The SVD of any real matrix $X \in \mathbb{R}^{K \times L}$ of rank n, is defined as the product of three matrices: U, Σ, and V.

$$X = U\Sigma V^T \tag{2-31}$$

Matrix Σ contains the (non-negative) singular values of X in descending order of magnitude on its diagonal, with the possibility of trailing zeros if $\texttt{rank}(X) < \min(K, L)$.

$$\Sigma = \texttt{diag}(\sigma_1, \sigma_2, \ldots, \sigma_n) \tag{2-32}$$

Matrices U and V are orthogonal and contain the left and right singular vectors respectively.

$$\begin{aligned} U &= \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \ldots & \mathbf{u}_n \end{pmatrix}, & V &= \begin{pmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \ldots & \mathbf{v}_n \end{pmatrix} \\ UU^T &= U^T U = I, & VV^T &= V^T V = I \end{aligned} \tag{2-33}$$

The dimensions of the matrices depend on the size of matrix X and on the requested form (full-size or economy-size). For a full rank ($n = K$) square matrix ($K = L$), the decomposition is equivalent to the form shown below.

$$\texttt{svd}(X) = \underbrace{\begin{pmatrix} \vdots & \ldots & \vdots \\ \mathbf{u}_1 & \ldots & \mathbf{u}_K \\ \vdots & \ldots & \vdots \end{pmatrix}}_{U \in \mathbb{R}^{K \times K}} \underbrace{\begin{pmatrix} \sigma_1 & 0 & 0 \\ 0 & \ddots & 0 \\ 0 & 0 & \sigma_K \end{pmatrix}}_{\Sigma \in \mathbb{R}^{K \times K}} \underbrace{\begin{pmatrix} \cdots & \mathbf{v}_1 & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & \mathbf{v}_K & \cdots \end{pmatrix}}_{V^T \in \mathbb{R}^{K \times K}} \tag{2-34}$$

Property 2.1: Uniqueness of Singular Value Decomposition (SVD)

The SVD of any matrix $X \in \mathbb{R}^{I \times J}$ is unique for distinct singular values:

$$\sigma_a \neq \sigma_b \;\forall\; \{a, b\} \in \min(I, J) \times \min(I, J) \tag{2-35}$$

Property 2.2: Rank of Singular Value Decomposition (SVD)

A few consequences of Definition 2.16 and Theorem 2.1 are:

- The number of nonzero singular values equals the rank R of the matrix.
- The first R columns of U are an orthonormal basis for the column space of X.
- The first R rows of V are an orthonormal basis for the row space of X.

Theorem 2.1: Eckart-Young-Mirsky Theorem [18] [57]

The Eckart-Young-Mirsky theorem states that the best low-rank approximation in Frobenius norm sense of any given matrix $X \in \mathbb{R}^{I \times J}$ can be made by discarding the smallest singular values.
For a square matrix in SVD format, this means discarding σ_I by truncating the matrices U, Σ, and V like shown in the equation below.

$$\texttt{SVD}(X) = \left(\begin{array}{ccc|c} \vdots & \cdots & \vdots & \vdots \\ \mathbf{u}_1 & \cdots & \mathbf{u}_{I-1} & \mathbf{u}_I \\ \vdots & \cdots & \vdots & \vdots \end{array}\right) \left(\begin{array}{ccc|c} \sigma_1 & 0 & 0 & 0 \\ 0 & \ddots & 0 & 0 \\ 0 & 0 & \sigma_{I-1} & 0 \\ \hline 0 & 0 & 0 & \sigma_I \end{array}\right) \left(\begin{array}{ccc} \cdots & \mathbf{v}_1 & \cdots \\ \vdots & \vdots & \vdots \\ \cdots & \mathbf{v}_{I-1} & \cdots \\ \hline \cdots & \mathbf{v}_I & \cdots \end{array}\right) \tag{2-36}$$

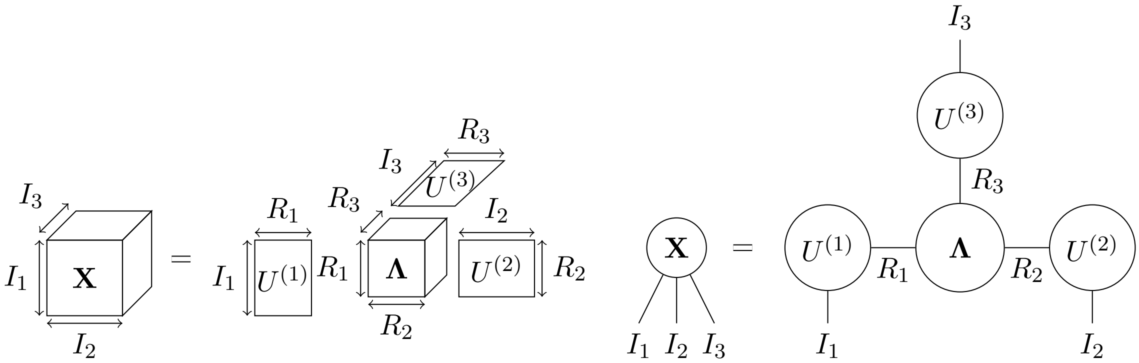
Definition 2.17: Multilinear Singular Value Decomposition (MLSVD)

The MLSVD decomposes a tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ into an all-orthogonal, ordered core tensor $\mathbf{\Lambda} \in \mathbb{R}^{R_1 \times R_2 \times \ldots \times R_N}$ and orthonormal factor matrices $[U^{(1)}, \ldots, U^{(N)}] \in \mathbb{R}^{I_i \times R_i}$ $\forall\, i \in \mathcal{N}$. In mathematical terms this can be denoted like shown below.

$$\begin{aligned} \mathbf{X} &= \mathbf{\Lambda} \times_1 U^{(1)} \times_2 \ldots \times_N U^{(N)} \\ \mathbf{X} &= \sum_{r_1=1}^{R_1} \cdots \sum_{r_n=1}^{R_N} \lambda_{r_1,\ldots,r_N} \mathbf{u}_{r_1}^{(1)} \circ \cdots \circ \mathbf{u}_{r_N}^{(N)} = \left[\mathbf{\Lambda}; U^{(1)}, \ldots, U^{(N)}\right] \end{aligned} \tag{2-37}$$

The MLSVD can be viewed as an unconstrained version of the CPD. The difference between them can be seen in the separate summation for each rank $\{R_1, \ldots, R_N\}$ and the accompanying norm $\lambda_{r_1,\ldots,r_N}$.

Figure 2-12 shows two graphical representations of the MLSVD for a 3rd-order tensor. Figure 2-12a shows a 3D visualization while Figure 2-12b shows the tensor network diagram. The MLSVD consists of a 3D core tensor Λ and factor matrices $[U^{(1)}, U^{(2)}, U^{(3)}]$.



(a) 3D visualization of MLSVD. **(b)** Tensor network diagram of MLSVD.

Figure 2-12: Graphical representations of the Multilinear Singular Value Decomposition.

Algorithm 2 shows the steps that need to be performed to obtain the MLSVD for any tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$. The first step is to compute the factor matrices $[U_1, \ldots, U_N]$ which can be done by performing an SVD on each mode-n matricization like shown in the equation below.

$$\mathtt{svd}(\mathbf{X}_{(n)}) = U^{(n)} \Sigma^{(n)} V^{(n)T} \tag{2-38}$$

The core tensor $\mathbf{\Lambda}$ can then easily be constructed using the multiplications shown in Equation 2-39. The factor matrices $[U^{(1)}, \ldots, U^{(N)}]$ are defined to be orthonormal, meaning $U^{(n)} U^{(n)T} = I \ \forall \ n \implies U^{(n)T} = (U^{(n)})^{-1} \ \forall \ n$.

$$\begin{aligned} \mathbf{X} &= \mathbf{\Lambda} \times_1 U^{(1)} \times_2 \ldots \times_N U^{(N)} \\ \mathbf{X} \times_N (U^{(N)})^{-1} &= \mathbf{\Lambda} \times_1 U^{(1)} \times_2 \ldots \times_{N-1} U^{(N-1)} \\ \mathbf{X} \times_N U^{(N)T} &= \mathbf{\Lambda} \times_1 U^{(1)} \times_2 \ldots \times_{N-1} U^{(N-1)} \\ &\vdots \\ \mathbf{\Lambda} &= \mathbf{X} \times_1 U^{(1)T} \times_2 \ldots \times_N U^{(N)T} \end{aligned} \tag{2-39}$$

Algorithm 2 Multilinear Singular Value Decomposition (MLSVD) [14],[81]

Require: Tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$,
1: $\mathbf{\Lambda} \leftarrow \mathbf{X}$
2: **for** $n = 1$ to N **do**
3: $[U^{(n)}, \Sigma^{(n)}, V^{(n)}] \leftarrow \mathtt{SVD}(\mathbf{X}_{(n)})$
4: $\mathbf{\Lambda} \leftarrow \mathbf{\Lambda} \times_n U^{(n)T}$
5: **end for**
6: **return** $\mathbf{\Lambda}$, $[U^{(1)}, \ldots, U^{(N)}]$

After obtaining the MLSVD using Algorithm 2, the user is able to truncate the decomposition by specifying the ranks $\{R_1, R_2, \ldots, R_N\}$ for each mode. Determining the ranks can be done by inspecting the dominant modes in each mode-n matricization.

The data complexity of the truncated MLSVD (also known as Tucker decomposition) is dependent on the chosen ranks $\{R_1, R_2, \ldots, R_N\}$. Equation 2-40 shows this denoted mathematically, where the complexity is reduced from being exponential in N, to being exponential in R.

$$\begin{aligned} \mathcal{O}(\mathbf{X}) &= \prod_{n=1}^{N} I_n \\ \mathcal{O}(\mathtt{Tucker}(\mathbf{X})) &= \sum_{n=1}^{N} I_n R_n + \prod_{n=1}^{N} R_n \end{aligned} \tag{2-40}$$

The truncated MLSVD using user-defined ranks $\{R_1, R_2, \ldots, R_N\}$ is however not optimal in least-squares sense [40]. Hence, in order to compute an optimal rank-$\{R_1, R_2, \ldots, R_3\}$ decomposition, an ALS-type algorithm such as the Higher-Order Orthogonal Iteration (HOOI) can be used [16].

2-3-8 Tensor Train (TT)

The third tensor decomposition that will be discussed is the Tensor Train (TT) [64]. Similar to the CPD and MLSVD, the tensor train also denotes a tensor as a summation of rank-1 terms.

Definition 2.18: Tensor Train (TT)

The equation below shows the TT denoted in mathematical terms.

$$\mathbf{X} = \sum_{r_1=1}^{R_1} \dots \sum_{r_{N-1}=1}^{R_{N-1}} \mathbf{x}^{(1)}_{:,r_1} \circ \mathbf{x}^{(2)}_{r_1,:,r_2} \circ \dots \circ \mathbf{x}^{(N)}_{r_{N-1},:} \tag{2-41}$$

The summations over an indivual TT-rank $\{R_1, R_2, \dots, R_{N-1}\}$ occur only between consecutive vector outer products, hence the name: Tensor Train.

Figure 2-13 shows the tensor network diagram for the TT. The diagram shows the TT-cores $[U^{(1)}, U^{(2)}, \dots, U^{(N)}]$ and the TT-ranks $\{R_1, R_2, \dots, R_{N-1}\}$.

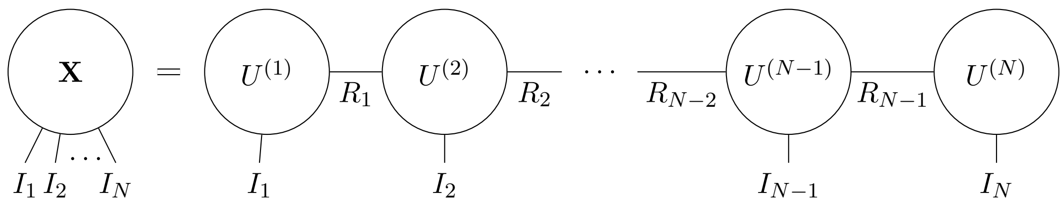


Figure 2-13: Tensor network diagram of Tensor Train Singular Value Decomposition (TT-SVD)

The complexity of the tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ is reduced from being exponential in N, to quadratic in the rank R and linear in I and N, as can be seen in Equation 2-42.

$$\begin{aligned} \mathcal{O}(\mathbf{X}) &= \prod_{n=1}^{N} I_n \\ \mathcal{O}(\texttt{TT-SVD}(\mathbf{X})) &= (N-2)IR^2 + 2IR \leq NIR^2 \end{aligned} \tag{2-42}$$

Tensor Train Singular Value Decomposition (TT-SVD)

A useful algorithm to obtain TT decomposition is the Tensor Train Singular Value Decomposition (TT-SVD) [64]. An important element of this TT-SVD is the δ-truncated SVD explained in Definition 2.19.

Definition 2.19: δ-truncated SVD

The δ-truncated SVD is a low-rank approximation of any matrix $X \in \mathbb{R}^{I \times J}$ based on Theorem 2.1. It uses the Singular Value Decomposition to divide the singular values into two sets: $\{\Sigma_1, \Sigma_2\}$.

$$X = \begin{pmatrix} U_1 & U_2 \end{pmatrix} \begin{pmatrix} \Sigma_1 & \\ & \Sigma_2 \end{pmatrix} \begin{pmatrix} V_1^T \\ V_2^T \end{pmatrix} \tag{2-43}$$

The second set Σ_2 is truncated from the SVD, resulting in the low-rank approximation $\tilde{X} = U_1 \Sigma_1 V_1^T$. The resulting approximation error is equal to the squares of the discarded singular values.

$$\|X - \tilde{X}\|_F^2 = \|\Sigma_2\|_F^2 = \sum_{r=R+1}^{\max(I,J)} \sigma_r^2 \tag{2-44}$$

By design, this approximation error lies below the specified error bound δ.

$$\|X - \tilde{X}\|_F^2 \leq \delta \tag{2-45}$$

Note: For notational convenience, the δ-truncated SVD is denoted by: $\texttt{svd}_\delta(\dots)$.

The Tensor Train Singular Value Decomposition (TT-SVD) is shown in Algorithm 3. It takes as input any tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and an allowed relative error of the decomposition ϵ.

The algorithm starts by calculating the allowed truncation error per mode δ (line 1), creates a placeholder for tensor $\mathbf{X}$ (line 2), and sets rank r_0 to 1 (line 2).

The next steps (line 4-7) are performed for the first $N-1$ modes of tensor $\mathbf{X}$, and can best be explained using tensor network diagrams.

Figure 2-14 shows the operations for the first mode, where $n = 1$. The tensor is reshaped using a mode-1 matricization (line 4) and a δ-truncated SVD is performed on the result (line 5). The truncated orthonormal matrix $U_1^{(1)}$ containing the left singular vectors of the SVD is reshaped using the rank information $\{r_0, r_1\}$ and stored as the first core G_1 (line 6). The product of the remaining matrices $\Sigma_1^{(1)} V_1^{(1)T}$ is stored as C and used for obtaining the next core.

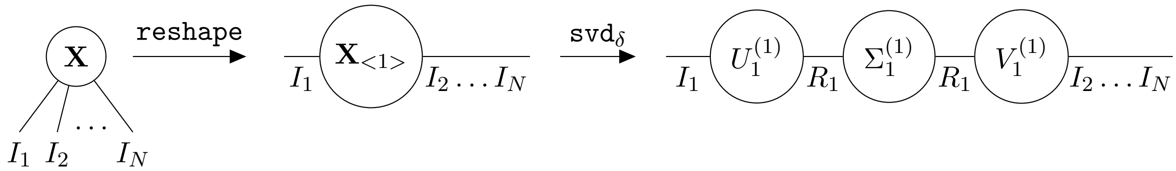


Figure 2-14: Operations for obtaining the first core ($U_1^{(1)}$) of TT-SVD.

Figure 2-15 shows the operations for the second mode, where $n = 2$. The remaining matrix product $\Sigma_1^{(1)} V_1^{(1)T}$ is reshaped, and a δ-truncated SVD is performed on the result. The truncated left singular vector matrix $U_1^{(2)}$ is reshaped and stored as the second core G_2, while the remaining matrix product $\Sigma_1^{(2)} V_1^{(2)T}$ is used to obtain the next core.

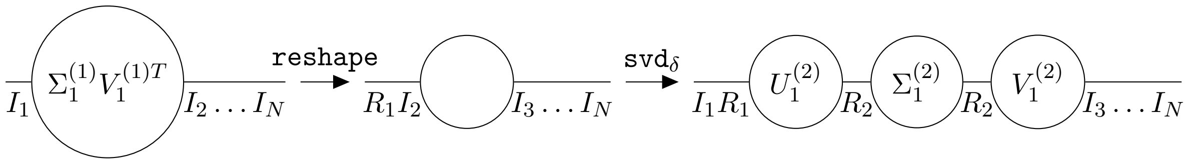


Figure 2-15: Operations for obtaining the second core ($U_1^{(2)}$) of TT-SVD.

Algorithm 3 Tensor Train Singular Value Decomposition (TT-SVD) [64]

Require: Tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, Allowed relative error ϵ

$\delta \leftarrow \frac{\epsilon}{\sqrt{N-1}} \|\mathbf{X}\|_F$

$C \leftarrow \mathbf{X}$, $r_0 \leftarrow 1$

for n=1 to N-1 **do**

  $C \leftarrow \mathtt{reshape}(C, [r_{n-1} I_k, :])$

  $[U, \Sigma, V, r_n] \leftarrow \mathtt{svd}_\delta(C)$

  $G_n \leftarrow \mathtt{reshape}(U, [r_{n-1}, I_n, r_n])$

  $C \leftarrow \Sigma V^T$

end for

$G_N \leftarrow C$

return Tensor $\mathbf{B} = [G_1, \dots G_N]$ in TT-form

This process of reshaping and performing δ-truncated SVD's is done for the first $N-1$ modes. The end result is displayed in Figure 2-16. The decomposition consists of $N-1$ orthonormal cores $[U_1^{(1)}, \dots, U_1^{(N-1)}]$, and a norm-core $\Sigma_1^{(N)} V_1^{(N)T}$ located in the N-th position.

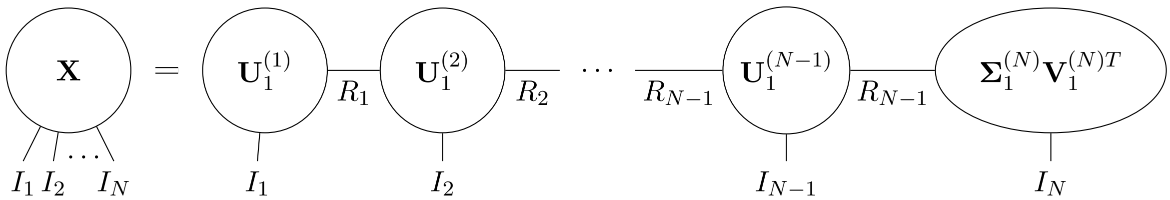


Figure 2-16: Tensor network diagram of Tensor Train Singular Value Decomposition (TT-SVD).

A python implementation of the TT-SVD algorithm is shown in Appendix A-3.

Chapter 3

Methodology

This chapter will introduce the methodology regarding the baseline approach and the 3 novel approaches of applying tensor decomposition methods for PCC. The baseline approach is MPEG's TMC3 [59], which will be covered in Section 3-1. The three novel approaches are voxel-based, synthetic and geometry aware tensor decomposition methods, which will be explained in Section 3-2, Section 3-3 and Section 3-4 respectively.

3-1 Baseline Approach

The baseline approach employed for PCC in this thesis is called TMC3 [59]. Liu et al. (2019) created a schematic overview of the encoder architecture for TMC3, which is shown in Figure 3-1. The overview shows that the geometry and attributes are encoded separately, where the attribute-encoding depends on the geometry.

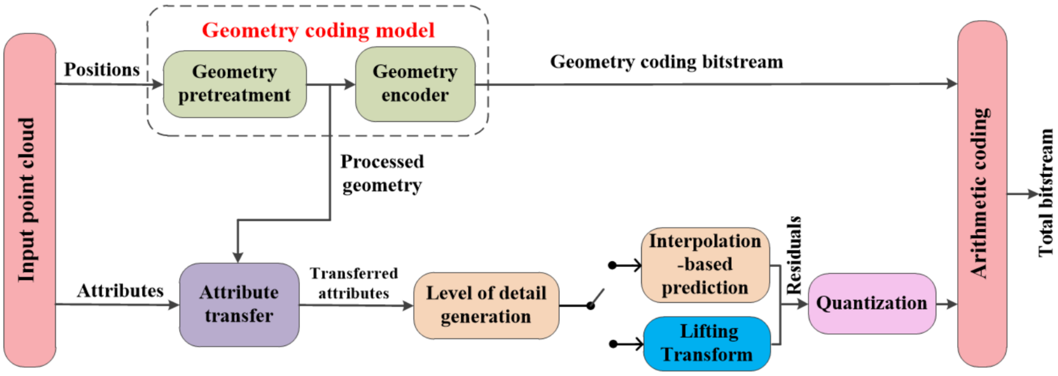


Figure 3-1: TMC3 Encoder Diagram [53].

For this thesis, the most relevant part of TMC3 is the geometry coding model, since LiDAR data mainly consists of geometry features. The to be proposed novel approaches are based on exploiting the geometry of the data, and are thus primarily competitors for the geometry coding model. The geometry coding model consists of two parts. The first part is the geometry pretreatment and the second part is the geometry encoder.

Geometry Pretreatment During the geometry pretreatment, the first step is to convert the 3D world coordinates of the point cloud to frame coordinates. This is done through translation ($[t_x, t_y, t_z]^T$) and scaling (α) of all N points using Equation 3-1.

$$\begin{pmatrix} x_i \\ y_i \\ z_i \end{pmatrix} = \frac{1}{\alpha} \cdot \left(\begin{pmatrix} x_i^{\text{world}} \\ y_i^{\text{world}} \\ z_i^{\text{world}} \end{pmatrix} - \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \right) \ \forall i \in N \tag{3-1}$$

After coordinate conversion the coordinates of all points can be quantized in order to achieve more compression at the expense of some reconstruction loss. This quantization is performed using Equation 3-2. The equation shows that the minimum value over all points $\mathbf{p}_{\min} = [x_{\min}, y_{\min}, z_{\min}]^T$ is substracted from each point, multiplied by the position quantization scale factor q and rounded off to the nearest integer.

$$\breve{\mathbf{p}}_i = \texttt{Round}\left((\mathbf{p}_i - \mathbf{p}_{\min}) \cdot q \right) \ \forall i \in N \tag{3-2}$$

Geometry Encoder The geometry encoder employed in TMC3 uses octree-based decomposition. The first step in octree decomposition is to determine the maximum depth of the octree. This is done by solving the inequality shown in Equation 3-3 for the smallest value of n. The maximum value of the (quantized) frame coordinates is taken over all dimensions (x, y, z) and all points (N), which results in the length (2^n) of the 3D cube that will be used to apply octree decomposition up to depth n.

$$2^n \geq \max\left(\max(x_i), \max(y_i), \max(z_i)\right) \ \forall i \in N \tag{3-3}$$

Figure 3-2 shows a graphical example of an octree decomposition for a fictitious point cloud with only 3 points situated in the top-left front side of the 3D box. The root node of the octree represents the entire 3D region and is divided into 8 smaller cubes called octants. Each of the octants occupied with points is subdivided again resulting into 8 more octants at the next level of depth. This process is repeated recursively until the required level of detail is met, or a specific condition is met such as: all LiDAR points are placed in a separate octant, which occurs at the maximum depth n. The right-hand side of Figure 3-2 shows how the octree decomposition can be efficiently represented as three bytes.

The final step of TMC3 is to encode the bitstream resulting from the geometry encoder using an arithmetic encoder, which gives as output a binary file containing the compressed representation.

The key strength of TMC3 for LiDAR PCC is the combination of octree decomposition with arithmetic encoding. The octree decomposition is well-suited for compressing sparse point clouds, since it effectively skips regions with no occupancy. The resulting bitstream from the octree decomposition can then conveniently be encoded using an arithmetic encoder.

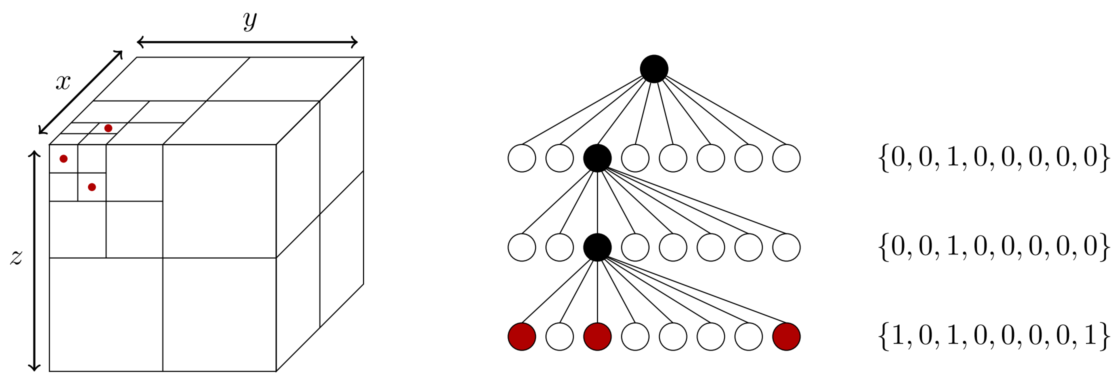


Figure 3-2: Octree Decomposition

Lossy vs. Lossless TMC3 has the built-in functionality to perform lossless compression (w.r.t. the quantization) as well as lossy compression. Lossless compression is achieved by picking a small enough value for the position quantization scale factor q, and generating an octree decomposition with maximum depth. Lossy compression is achieved conversely, for example by decomposing the octree up to depth $n - 1$ or picking a large value for q.

3-2 Voxel-Based Tensor Decompositions for Point Cloud Compression

The first approach that will be discussed is applying tensor decomposition methods on voxel-based representations for point cloud compression. However, before diving into the mathematical formulation let us first discuss what the voxel-based representation entails and why it would be a viable candidate for applying tensor decomposition methods.

Voxelization is defined as the process of discretizing the 3D space into equal sized volumes called voxels. The voxels are small 3D cubes that combined build up the entire 3D space. Figure 3-3 shows an example of a voxelized representation, where the red cubes denote voxels that are occupied with LiDAR points.

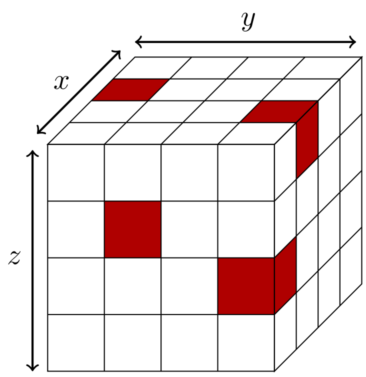
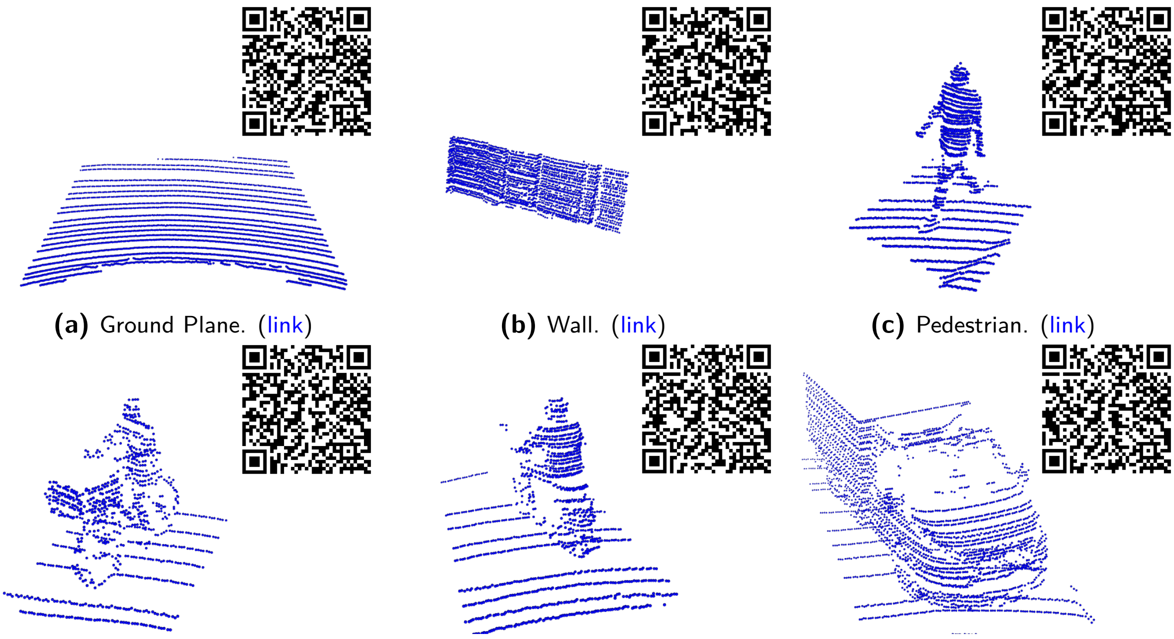


Figure 3-3: Voxel-Based Representation

Figure 3-4 shows LiDAR data from the VoD dataset for several distinct road elements and road users. The road elements are the ground plane and the wall, which can accurately be described as planar surfaces. The road users, which are the pedestrian, cyclist, scooter, and truck can be approximately described by a set of planar surfaces or volumes.



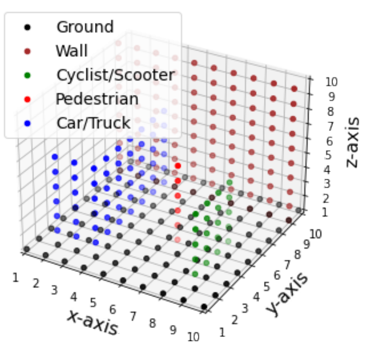
(a) Ground Plane. (link) **(b)** Wall. (link) **(c)** Pedestrian. (link)

(d) Cyclist. (link) **(e)** Scooter. (link) **(f)** Truck. (link)

Figure 3-4: Road Elements and Users. Scan QR Code for 3D Render.

The reason why recognizing that road elements and users can be described as planar objects is interesting is because tensor decomposition methods can effectively describe planar surfaces. This is because planar surfaces are naturally low-rank structures of the 3D space they reside in. Tensor decompositions are powerful methods designed to seek these low-rank structures in order to reduce data complexity.

Figure 3-5 shows an approximation of the road users and road elements shown in Figure 3-4 using a rank-6 CPD. The elements are thus constructed using 6 vector outer products like shown in Equation 3-4, which together build up the entire scene.

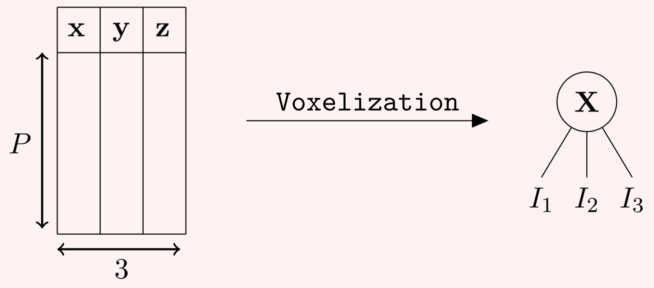


$$\mathbf{X} = \sum_{r=1}^{6} \lambda_r \boldsymbol{b}_r^{(1)} \circ \boldsymbol{b}_r^{(2)} \circ \ldots \circ \boldsymbol{b}_r^{(N)} \tag{3-4}$$

Figure 3-5: Road Users and Elements described as CPD.

Definition 3.1: Voxelization

A point cloud $\mathcal{V} \in \mathbb{R}^{P \times 3}$ consisting of P points and 3 features per point can be voxelized resulting in a tensor $\mathbf{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$, where $\{I_1, I_2, I_3\}$ refer to the size of the $\{x, y, z\}$ dimensions respectively.



The 3D space with range $\{x_r, y_r, z_r\}$ is discretized using a voxel dimension $\{x_v, y_v, z_v\}$.

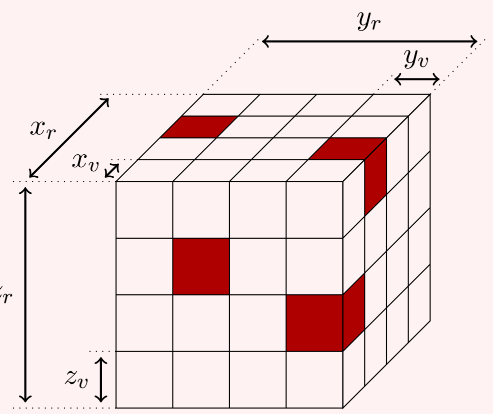


Figure 3-6: Voxelization.

The dimensions of the resulting tensor will be determined by the amount of voxels that fit in each direction.

$$\mathbf{X} \in \mathbb{R}^{\frac{x_r}{x_v} \times \frac{y_r}{y_v} \times \frac{z_r}{z_v}} = \mathbb{R}^{I_1 \times I_2 \times I_2} \tag{3-5}$$

The mapping for each point $\mathbf{p} = [x, y, z]^T$ of $\mathcal{V}$ towards tensor $\mathbf{X}$ is determined by calculating its tensor indices (i_1, i_2, i_3) using the equation below, where "$\lceil\ \rceil$" denotes the `ceil` function, and $\{x, y, z\}_{min}$ is the lowest value of $\{x, y, z\}$ for all points in $\mathcal{V}$.

$$(i_1, i_2, i_3) = \left(\left\lceil \frac{\mathbf{p}_x - x_{min}}{x_v} \right\rceil, \left\lceil \frac{\mathbf{p}_y - y_{min}}{y_v} \right\rceil, \left\lceil \frac{\mathbf{p}_z - z_{min}}{z_v} \right\rceil \right) \ \forall\ \mathbf{p} \in \mathcal{V} \tag{3-6}$$

The resulting tensor $\mathbf{X}$ will either be **binary** in case only the presence of any point is counted, or **occupancy-based** in case the amount of points that reside in a voxel is counted.

Definition 3.1 shows the mathematical formulation, which is used throughout this thesis to describe the process of voxelization. When reconstructing a voxelized representation the inverse mapping is applied, and the presence of points within a voxel is determined by rounding of the tensor element to an integer value.

As mentioned in Definition 3.1, voxelizing a point cloud is performed using two voxelization parameters: the range of the 3D space, and the voxel dimensions. Table 3-1 shows the voxelization parameters for three renowned backbones, which achieved state-of-the-art performance on the nuScenes object detection benchmark.

Publication	CenterPoint [87]	SECOND [85]	VoxelNeXt [11]
Range: $\begin{pmatrix} x_{min} & x_{max} \\ y_{min} & y_{max} \\ z_{min} & z_{max} \end{pmatrix}$	$\begin{pmatrix} -51.2 & 51.2 \\ -51.2 & 51.2 \\ -5 & 3 \end{pmatrix}$	$\begin{pmatrix} -49.6 & 49.6 \\ -49.6 & 49.6 \\ -5 & 3 \end{pmatrix}$	$\begin{pmatrix} -54 & 54 \\ -54 & 54 \\ -5 & 3 \end{pmatrix}$
Voxel Size: $\begin{pmatrix} x_v \\ y_v \\ z_v \end{pmatrix}$	$\begin{pmatrix} 0.1 \\ 0.1 \\ 0.2 \end{pmatrix}$	$\begin{pmatrix} 0.05 \\ 0.05 \\ 0.2 \end{pmatrix}$	$\begin{pmatrix} 0.075 \\ 0.075 \\ 0.2 \end{pmatrix}$

Table 3-1: Configurations of state-of-the-art voxel-based backbones for nuScenes object detection benchmark [62].

As can be seen from the table above, the ranges and voxel dimensions are quite similar. For this thesis the 3D range and voxel dimensions of VoxelNeXT are used for performing voxelizations of point clouds. These voxelizations then result in a tensor of dimension:

$$\mathbf{X} \in \mathbb{R}^{\frac{x_r}{x_v} \times \frac{y_r}{y_v} \times \frac{z_r}{z_v}} = \mathbb{R}^{\frac{108}{0.075} \times \frac{108}{0.075} \times \frac{8}{0.2}} = \mathbb{R}^{1440 \times 1440 \times 40} \tag{3-7}$$

Voxelizing a point cloud discretizes the 3D space, and forces all points to lie onto a 3D integer lattice. This process causes an inherent loss to occur, without any compression gains.

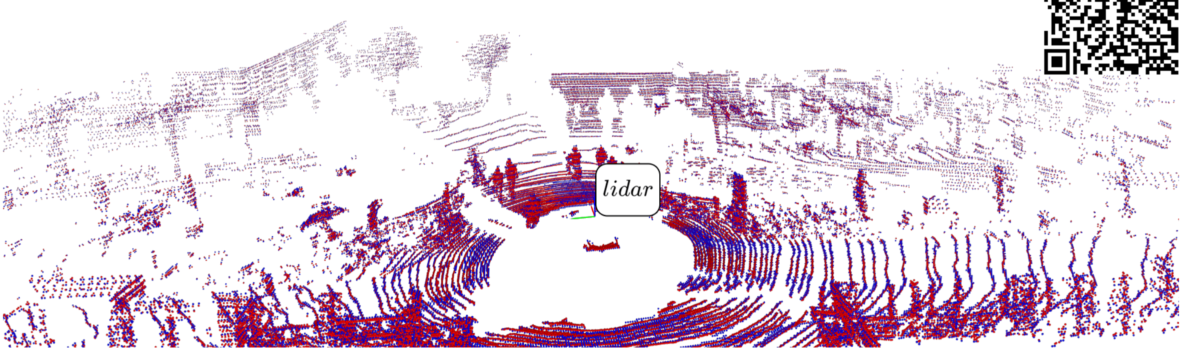


Figure 3-7: Original(red) and Voxelized(blue) LiDAR from VoD dataset using VoxelNeXt Voxel Dimensions [65],[11]. PSNR-NN: 64.20. Scan QR Code or click on link for 3D Render.

Figure 3-7 shows a visualization of a sample from the VoD dataset with the original(red) and voxelized(blue) LiDAR. The voxelization results into a drop from an infinite/perfect PSNR-NN (no voxelization), to a value of: 64.20. This value is already lower (worse) than the

PSNR-NN obtained after applying lossless compression using the baseline method TMC13 onto the same sample. TMC13 obtained a PSNR-NN of: 106.69.

3-2-1 Tensorized Voxelizations

In order to exploit more of the similarities in local geometry voxel-based representations could be tensorized along either of the coordinate axis. Figure 3-8 shows an illustration of what such a tensorization along both the x and y coordinate axes would look like. The image shows how a tensor of dimension $12 \times 12 \times 4$ is reshaped into a tensor of dimension $4 \times 3 \times 4 \times 3 \times 4$.

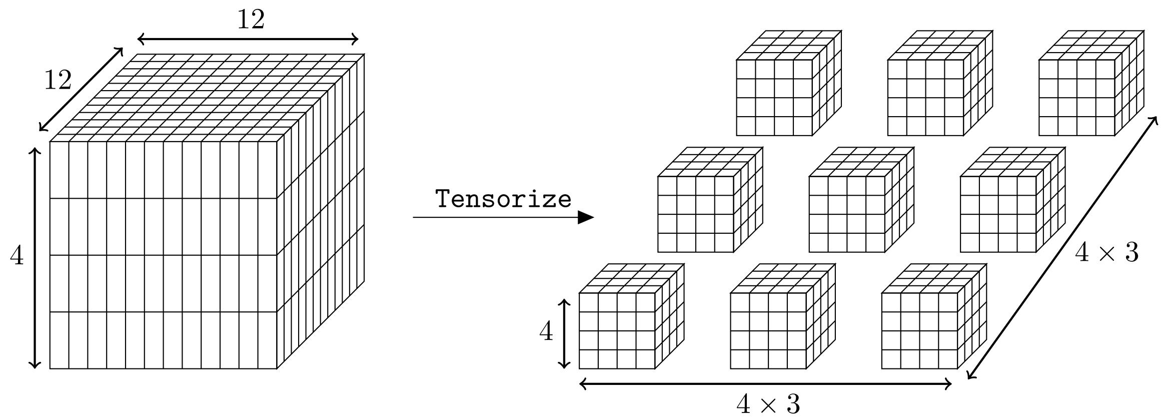


Figure 3-8: Tensorizing a voxel-based representation along the x and y coordinate axes.

The idea is that elements such as walls or road users appear multiple times in various (x,y)-locations in the scene. Possibly, tensorizing the voxelized representations could aid in exploiting this phenomenon. The voxel-based representation employed in this work is of dimension $\mathbf{X} \in \mathbb{R}^{1440 \times 1440 \times 40}$. Equation 3-8 shows how this voxelization is tensorized along the x- and y- coordinate axis, resulting in a reshaped tensor with 5 modes instead of 3.

$$\mathbf{X} \in \mathbb{R}^{1440 \times 1440 \times 40} \xrightarrow{\text{Tensorize}} \mathbf{X} \in \mathbb{R}^{40 \times 36 \times 40 \times 36 \times 40} \tag{3-8}$$

3-3 Synthetic Tensor Decompositions for Point Cloud Compression

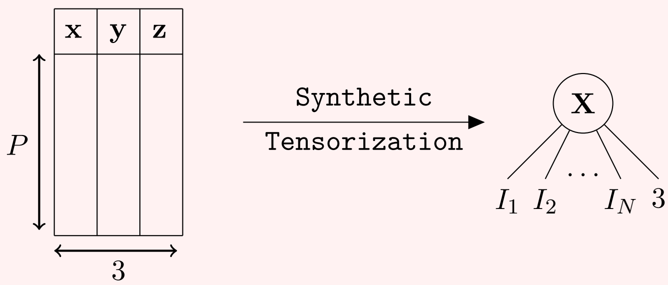
The second approach that will be discussed is synthetic tensorization. This approach circumvents the problem of obtaining a highly sparse and voluminous tensor, which occurs during voxelization of a sparse point cloud. It does this by directly tensorizing the LiDAR data in tabular form. Definition 3.2 introduces this tensorization technique as **synthetic** tensorization, since the modes and their sizes need to be artificially chosen.

Definition 3.2: Synthetic Tensorization

A point cloud $\mathcal{V} \in \mathbb{R}^{P\times 3}$ consisting of P points and 3 features per point can be synthetically tensorized by dividing the P points into N modes such that:

$$P = I_1 \cdot I_2 \cdot \ldots \cdot I_N \tag{3-9}$$

The point cloud $\mathcal{V}$ denoted in tabular form below is reshaped using a synthetic set of dimensions $[I_1, \ldots, I_N]$ resulting into the tensor: $\mathbf{X} \in \mathbb{R}^{I_1\times I_2\times\ldots\times I_N\times 3}$.



Using Matlab notation this operation can be denoted as:

$$\mathbf{X} = \texttt{reshape}(\mathcal{V}, [I_1, \ldots, I_N, 3]) \tag{3-10}$$

Note: Synthetic tensorization uses the little-endian convention (Definition 2.1).

An important consequence of the formulation in Definition 3.2, is that in order to successfully tensorize the LiDAR data, the multiplicative property below needs to hold.

$$P = I_1 \cdot I_2 \cdot \ldots \cdot I_N \tag{3-11}$$

This means that the number of points in the cloud needs to be equal to the product of the sizes in the first N dimensions $[I_1, \ldots I_N]$ of the to be created tensor. In order to satisfy this constraint some LiDAR points will have to be discarded. Choosing which points are discarded is based on their Euclidean distance to the LiDAR reference frame. The points furthest away are discarded first, since points closer to the vehicle are deemed much more valuable in an automotive setting. For example, detecting a pedestrian at close proximity to the vehicle is a much more urgent task, than one located at a large distance.

The modes and the sizes of each mode need to be determined prior to synthetic tensorization. Equation 3-12 shows the synthetic tensorization parameters that will be explored. These parameters can adhere to the constraint in Equation 3-11, since they contain fewer elements than the minimum amount of points across all clouds in the dataset which is: 64273.

$$
\begin{aligned}
&\text{Set } 1 := (3,3,3,3,3,3,3,3,3,3) && \text{since} && 3^{10} &&= 59049 \leq 64273 \\
&\text{Set } 2 := (6,6,6,6,6,6) && \text{since} && 6^{6} &&= 46656 \leq 64273 \\
&\text{Set } 3 := (9,9,9,9,9) && \text{since} && 9^{5} &&= 59049 \leq 64273 \\
&\text{Set } 4 := (15,15,15,15) && \text{since} && 15^{4} &&= 50625 \leq 64273 \\
&\text{Set } 5 := (40,40,40) && \text{since} && 40^{3} &&= 64000 \leq 64273 \\
&\text{Set } 6 := (253,253) && \text{since} && 253^{2} &&= 64009 \leq 64273
\end{aligned} \tag{3-12}
$$

In tensor network diagram notation these synthetic tensorizations can be visualized like shown in Figure 3-9.

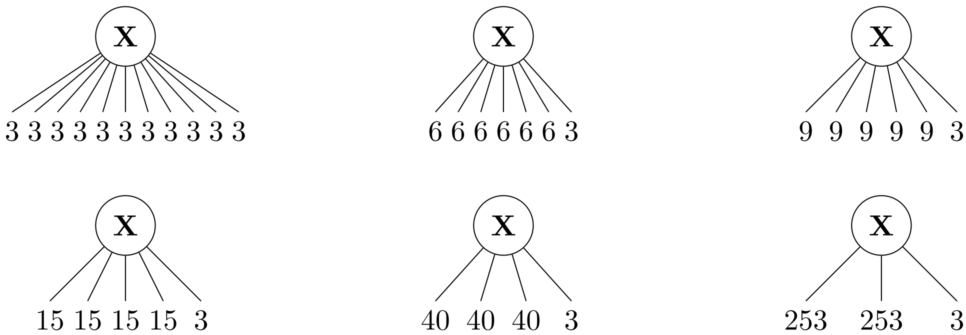


Figure 3-9: 6 Different Synthetic Tensorizations.

3-4 Geometry Aware Tensor Decompositions for Point Cloud Compression

The third approach that is considered is **geometry aware** tensor decomposition for PCC. This approach is in essence a combination of the voxel-based method and the synthetic tensorization-based method. The goal of this tensorization method is to tensorize the LiDAR data in such a way that the location of a point within the tensor ($\mathbf{X}_{(i_1,i_2,i_3)}$) corresponds with its real-world location ($p = [x,y,z]^T$). Definition 3.3 denotes this tensorization method formally.

Definition 3.3: Geometry Aware Tensorization

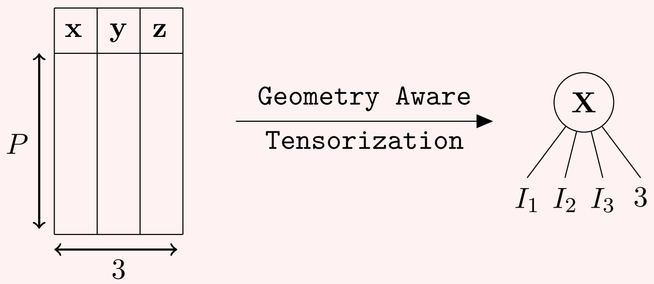
A point cloud $\mathcal{V} \in \mathbb{R}^{P\times 3}$ with Cartesian range $\{x_r, y_r, z_r\}$ consisting of P points and 3 features per point can be tensorized geometry aware by dividing the P points into 3 modes such that:

$$P = I_1 \cdot I_2 \cdot I_3, \qquad (I_1 : I_2 : I_3) \approx (x_r : y_r : z_r) \tag{3-13}$$

The size of each mode is thus in proportion to the range of values points can have in that mode.
The next step is to order the points in the point cloud in such a way that after tensorization, their location in the tensor will (roughly) correspond with their real-world location. This can be done using two approaches:

- Hierarchical Approach Subsection 3-4-1
- Assignment Problem Subsection 3-4-2

The last step is to reshape the ordered point cloud $\mathcal{V}$ using the calculated set of dimensions $[I_1, I_2, I_3]$ resulting into the tensor: $\mathbf{X} \in \mathbb{R}^{I_1\times I_2\times I_3\times 3}$.



Note: Synthetic tensorization uses the little-endian convention (Definition 2.1).

The reason why geometry aware tensorization could be a fruitful method is because it combines the advantages of the voxel-based and synthetic tensorization methods. LiDAR points are being given a location in the tensor ($\mathbf{X}_{(i_1,i_2,i_3)}$) based on their $[x, y, z]$-values. Neighbouring real-world points will thus be given neighbouring tensor indices. This could allow for tensor decomposition methods to exploit the similarity in neighbouring points/tensor indices, since they will likely be part of the same low-rank planar structure in the real world. Contrary to the voxel-based method, this method does not result in a highly sparse and voluminous tensor. Hence, compression gains can instantly be acquired by truncating any singular values using tensor decomposition methods.

3-4-1 Hierarchical Approach

One approach to achieve a geometry aware tensorization of a LiDAR point cloud is to use hierarchic division with sorting. The idea can best be illustrated by viewing Figure 3-10. The first step is to sort the points in the cloud by one of their geometry features $\{x, y, z\}$. In the example this is done using their z-location. The next step is to divide the sorted cloud into level sets. In the example there are 3 z-level sets. The points in these 3 level sets will become the: bottom-, middle-, and top-layer of the tensor. The next step is to order the z-level sets by another geometry feature. In the example this is the x-location. This then allows for creating 7 x-level sets for each z-level set. These x-level sets are the mode-3 fibres of the tensor. The last step is to sort the mode-3 fibres based on their y-location, with as result an hierarchically divided point cloud based on their Cartesian values.

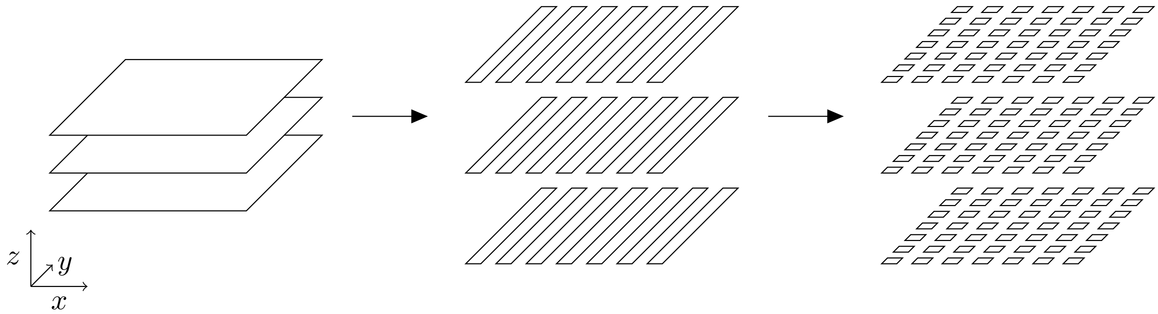


Figure 3-10: Hierarchical approach for geometry aware tensorization.

3-4-2 Assignment Problem

The hierarchical approach is an efficient and relatively easy-to-implement method, that often results in a decent mapping of LiDAR points to tensor indices. It does however not result in an optimal assignment of points in any sense. An alternative method to tackle the problem of assigning points to tensor indices is to formulate it as an assignment problem [9]. A formulation as assignment problem will allow for penalizing the placement of points in the tensor using a cost function. This cost function can for example be the Frobenius error of a point's "ideal" location with respect to its true location in the tensor.

The first step in the assignment problem approach is to create a location tensor $\mathbf{L}$ with the calculated dimensions (I_1, I_2, I_3) using Equation 3-13.

$$\mathbf{L} = \mathbb{R}^{I_1 \times I_2 \times I_3 \times 3} \tag{3-14}$$

The second step is to fill each mode-4 fibre of this location tensor. The values of the mode-4 fibres are calculated by mapping the location tensor indices to real-word Cartesian values. Equation 3-15 shows how the entries of the location tensor are calculated. The parameters $\{x_{box}, y_{box}, z_{box}\}$ represent the size of a voxel when discretizing the range of the scene $\{x_r, y_r, z_r\}$ using parameters $\{I_1, I_2, I_3\}$. The parameters $\{x_{min}, y_{min}, z_{min}\}$ are the minimum values of the scene.

$$\begin{gathered}
\mathbf{L}(i_1, i_2, i_3, :) = \begin{pmatrix} x_{min} + x_{box} \cdot \left(i_1 + \frac{1}{2}\right) \\ y_{min} + y_{box} \cdot \left(i_2 + \frac{1}{2}\right) \\ z_{min} + z_{box} \cdot \left(i_3 + \frac{1}{2}\right) \end{pmatrix} \quad \forall \ \{i_1, i_2, i_3\} \in \{\mathcal{I}_1 \times \mathcal{I}_2 \times \mathcal{I}_3\} \\
x_{box} = \frac{x_r}{I_1}, \quad y_{box} = \frac{y_r}{I_2}, \quad z_{box} = \frac{z_r}{I_3}
\end{gathered} \tag{3-15}$$

The third step is to compute the mode-4 matricization ($\mathbf{L}_{(4)}$) of the tensor resulting in location matrix L like shown in Equation 3-16.

$$L = \mathbf{L}_{(4)} \in \mathbb{R}^{I_1 I_2 I_3 \times 3} \tag{3-16}$$

The location matrix L can then be used to calculate the placement error $e_{j,k}$. The placement error is defined as the Frobenius loss of placing point j in location k in the tensor. This placement error is calculated for all N points and all N locations.

$$e_{j,k} = ||p_j - L_{k,:}||_2^2 \;\; \forall \; \{j,k\} \in \{\mathcal{N} \times \mathcal{N}\} \tag{3-17}$$

All these placement errors can be stored in a single matrix $C \in \mathbb{R}^{N \times N}$, called the cost matrix. In this matrix a row corresponds to a point, and a column to a location in the tensor.

$$C = \begin{pmatrix} e_{1,1} & \cdots & e_{1,N} \\ \vdots & \ddots & \vdots \\ e_{N,1} & \cdots & e_{N,N} \end{pmatrix} \tag{3-18}$$

This cost matrix can then be used to formulate a linear sum assignment problem [15] like shown in Equation 3-19.

$$\begin{aligned} \min \sum_j \sum_k C_{j,k} X_{j,k} \\ \sum_j X_{j,k} = 1 \;\; \forall j \in \mathcal{N} \\ \sum_k X_{j,k} = 1 \;\; \forall k \in \mathcal{N} \end{aligned} \tag{3-19}$$

The Boolean matrix X denotes whether point j is assigned to location k. Summations across all rows and all columns of X will always sum up to 1, since each point can only be assigned once and each location can only occupy one point.

Solving this problem can be done using various methods such as the Hungarian Algorithm [43], or Jonker-Volgenant Method [37]. This thesis uses a modification of the Jonker-Volgenant Method [15] with a guarantee of computational complexity in the order $\mathcal{O}(N^3)$ compared to the Hungarian method which has $\mathcal{O}(N^4)$, where N is the dimension of the cost matrix.

3-4-3 Experiment Overview

Figure 3-11 shows an overview of the proposed experiments as well as the baseline model TMC3, which is displayed in the grey box.

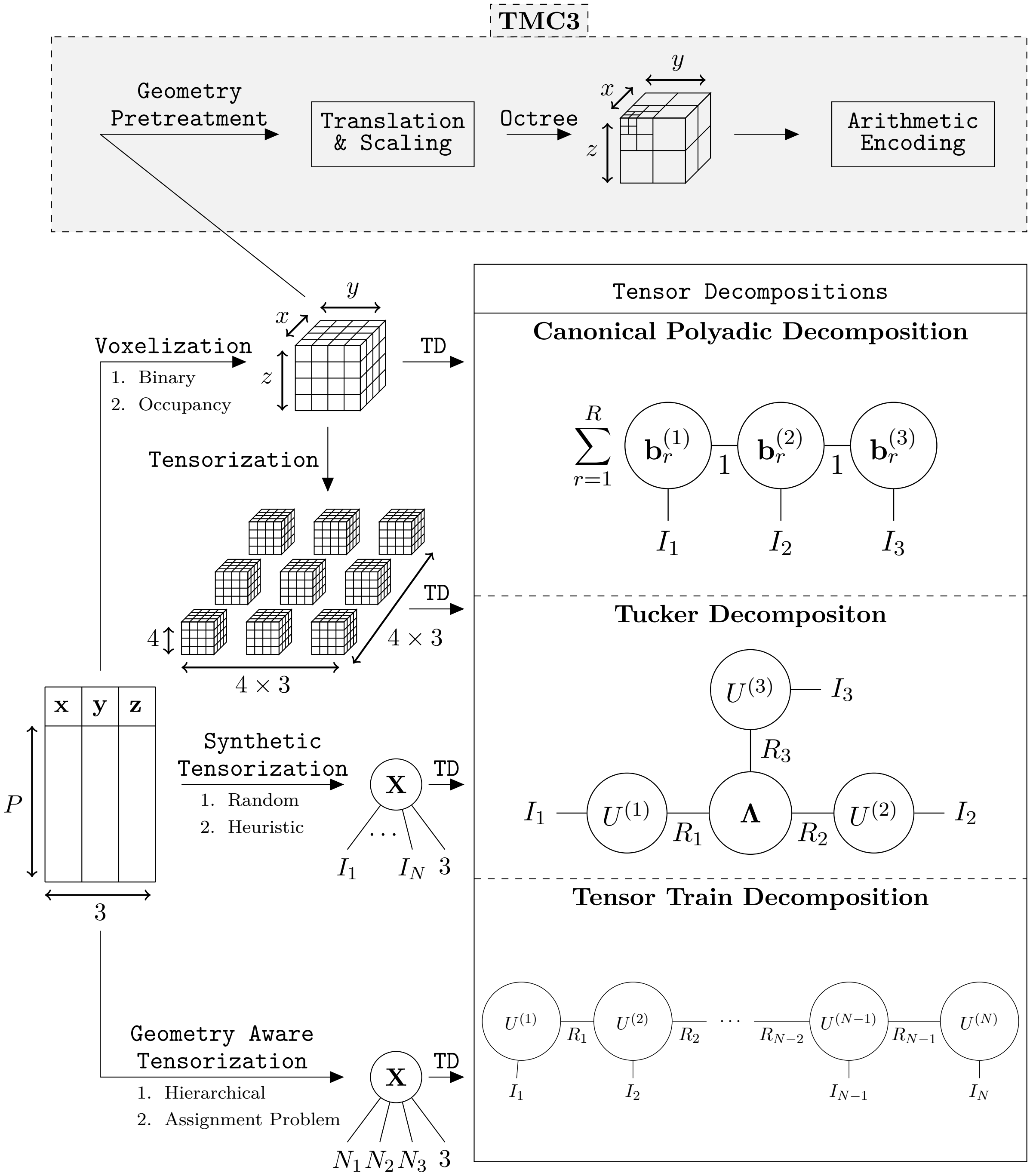


Figure 3-11: Overview of the proposed experiments. The grey box displays the baseline model: TMC3

Chapter 4

Experiments

This chapter will begin by introducing the experimental setup in Section 4-1. Afterwards, the experimental results of the baseline method will be established in Section 4-2. Section 4-3, Section 4-4, and Section 4-5 will discuss the results for the three novel approaches: Voxel-Based, Synthetic Tensorization, and Geometry Aware Tensorization for PCC respectively.

4-1 Experimental Setup

View of Delft Dataset Evaluating the baseline method and all proposed novel methods in Chapter 3 will be done using samples from the View of Delft (VoD) dataset [65]. In order to obtain a fair comparison between all approaches, the same 10 samples will be used for all methods unless stated otherwise. The sample ID's are: $[00000, 01000, 02000, 03000, 04000, 05000, 06400, 07000, 08000, 09000]$. These samples are purposefully chosen to lie far away from each other temporally, as to avoid similarities between samples. When performing qualitative analysis a different sample with ID [01222] will be used for all methods.

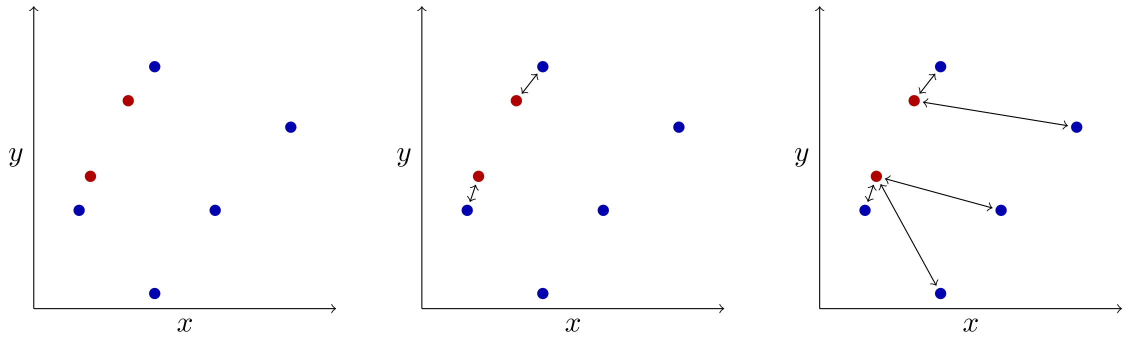
The premise of this thesis is that the geometry of LiDAR point clouds might inherently contain low-rank structures suitable for tensor decompositions. Hence, in order to get the most unbiased comparison to the geometry coding model of the baseline method (TMC3), only the geometry features $[x, y, z]$ of the LiDAR data will be decomposed.

In order to assess the performance of a codec, two objective metrics are required. One metric is needed to determine the quality of the reconstruction, and another to determine the amount of compression achieved. These metrics are the PSNR-NN and BPP respectively.

PSNR-NN The metric chosen to assess the quality of the reconstruction is the two-sided Peak Signal-to-Noise Ratio Nearest Neighbour Loss (PSNR-NN), which is displayed in Equation 4-1.

$$
\begin{aligned}
\text{PSNR-NN} &= 10\log_{10}\left(\frac{\Omega^2}{NN_{loss}^{max}}\right) \\
\Omega &= \max\Big(x_{max}-x_{min},\ y_{max}-y_{min},\ z_{max}-z_{min}\Big) \\
NN_{loss}^{max} &= \max\Big(NN_{loss}(\mathcal{V}_{orig},\mathcal{V}_{rec}),\ NN_{loss}(\mathcal{V}_{rec},\mathcal{V}_{orig})\Big) \\
NN_{loss}(\mathcal{V}_A,\mathcal{V}_B) &= \sqrt{\frac{1}{P}\sum_{\mathbf{p}\in\mathcal{V}_A}\|\mathbf{p}-\mathbf{p}_{Bnn}\|_2^2}
\end{aligned}
\tag{4-1}
$$

The nearest neighbour loss (NN_{loss}) of point cloud $\mathcal{V}_A$ with respect to point cloud $\mathcal{V}_B$ is defined as the square root of the average Euclidean distance between each point in $\mathcal{V}_A$ and its nearest neighbour in $\mathcal{V}_B$. The PSNR-NN takes the maximum over the NN_{loss} of the original point cloud $\mathcal{V}_{orig}$ with the reconstruction $\mathcal{V}_{rec}$ and vice versa. The reason why taking the maximum is important can be explained using Figure 4-1. Figure 4-1a shows a dummy example of two 2D point clouds: $\mathcal{V}_A$ (red) and $\mathcal{V}_B$ (blue). The NN_{loss} of $\mathcal{V}_A$ with respect to $\mathcal{V}_B$ is quite small and is visualized using the arrows displayed in Figure 4-1b. On the contrary, the NN_{loss} of $\mathcal{V}_B$ with respect to $\mathcal{V}_A$ displayed in Figure 4-1c is quite large, since a number of points in $\mathcal{V}_B$ are located at a great distance from the points in $\mathcal{V}_A$.



(a) Two 2D point clouds. **(b)** $NN_{loss}(\mathcal{V}_A,\mathcal{V}_B)$. **(c)** $NN_{loss}(\mathcal{V}_B,\mathcal{V}_A)$.

Figure 4-1: The two-sided nearest neighbour loss for a 2D point cloud.

Interpreting $\mathcal{V}_A$ as the original point cloud and $\mathcal{V}_B$ as its reconstruction allows for concluding that the reconstruction contains a great amount of false positives. These false positives are reflected into a high NN_{loss}, causing a low PSNR-NN. Interpreting $\mathcal{V}_B$ as the original point cloud and $\mathcal{V}_A$ as its reconstruction allows for making a similar argument, but now regarding a high amount of false negatives. Hence, the two-sided NN_{loss} is a great measure for limiting the amount of false predictions.

The parameter Ω is defined as the maximum distance between points across each of the coordinate axes. This parameter allows for comparing performance between two point clouds with a different scale, i.e. a larger NN_{loss}^{max} is permitted for a point cloud with a larger range of (x,y,z)-values in order to obtain a similar PSNR-NN. The PSNR-NN is bounded between 0 and infinity, where a perfect reconstruction results in an infinite PSNR-NN score.

Another type of loss, which is often employed (in tensor decomposition methods) to calculate the error between an original sample and a reconstruction is the Frobenius error shown in Equation 4-2.

$$\text{Frobenius Error} = \frac{\|\mathcal{V}_A - \mathcal{V}_B\|_F}{\|\mathcal{V}_A\|_F} \tag{4-2}$$

The Frobenius error is however not a suitable error metric for point cloud compression. This is because the Frobenius error uses the element-wise difference between points in the cloud. An element-wise difference assumes that the reconstructed points should be found at the exact same index they were in the original point cloud. This is a hard constraint which is unnecessary, since the ordering of points in the point cloud does not affect the reconstruction quality.

BPP In order to asses how much compression each method achieves, a compression metric applicable to both the baseline method and the competing tensor decomposition methods needs to be chosen. This metric is called the Bits Per Point (BPP) and is widely used across PCC literature [5, 53, 91]. The BPP can simply be calculated by dividing the size in bits of an encoded representation by the amount of points in the cloud it encodes, like shown in Equation 4-3. Consequently, the BPP is an effective metric for comparing compression across point clouds of different sizes.

$$\text{BPP} = \frac{\text{Amount of Bits}}{\text{Amount of Points}} \tag{4-3}$$

The baseline method outputs a binary file, which contains the compressed representation of the LiDAR point cloud. Computing the BPP can thus be done by evaluating the size of this file and the amount of points it encodes. For the competing tensor decomposition methods, the amount of bits can be obtained by multiplying the amount of independent elements in the tensor decomposition by the precision in bits used to represent each element. This then raises the question: what precision is needed to effectively represent the tensor decomposition elements?

This thesis investigates applying tensor decompositions on 3 different representations: Voxel-based, Synthetic Tensorization and Geometry Aware Tensorization. Regarding the voxel-based representations, 2 different options are considered: binary voxelization, and occupancy-based voxelization. Although these representations contain binary and integer values respectively that could be effectively represented using a small amount of bits, the tensor decompositions which define these representations do not. They consists of floating point elements, similarly to the tensor decompositions obtained using Synthetic and Geometry-Aware Tensorization.

For all of these representations a precision must thus be picked based on the desired resolution of tensor elements, which in turn depends on the desired resolution in $[x, y, z]$ coordinates. This presents a trade-off in terms of compression by means of precision in bits, and compression resulting from tensor decomposition methods. For this thesis, the half-precision floating-point format (16 bit) is chosen [33]. The reason why is because 16-bit encoding results in an acceptable precision at the edges of the scene. Table 3-1 showed that the maximum geometry values for points is 54, hence using Equation 4-4 the maximum interval of floating point precision can be defined.

$$\begin{aligned}
\text{Interval} &= 2^{\lfloor \log_2(\max(x,y,z)) \rfloor - \text{mantissa}} \\
\text{Interval} &= 2^{\lfloor \log_2(54) \rfloor - 10} \\
\text{Interval} &= 2^{5-10} = 2^{-5} = 0.03125
\end{aligned} \tag{4-4}$$

This resolution interval at the edges of scene is deemed acceptable for automotive applications. The resolution of 0.03125 meters is at least twice as accurate compared to the voxel sizes of automotive applications such as VoxelNeXt [11], which is: $[v_x, v_y, v_z] = [0.075, 0.075, 0.2]$.

Comparing the BPP's for both methods has two limitations. One small limitation is that the binary file obtained using TMC3 holds metadata about the point cloud. This metadata requires storage but does not directly store any points. A rather large limitation on the side of tensor decomposition methods, is that the tensor elements itself are not being bitwise compressed, which does occur in the baseline method. TMC3 uses this bitwise compression in the final stages of the compression pipeline: the arithmetic encoder. Taking these factors into account, the reader is advised to consider the obtained results of the tensor decomposition methods as a proof of concept, which could be improved further by applying bitwise compression techniques onto the decompositions.

Compression Rate Apart from the BPP an additional metric is used, which also denotes the amount of compression achieved. This metric is the compression rate and is defined as:

$$\text{Compression Rate} = \frac{\text{Elements in Tensor Decomposition}}{\text{Elements in Original Data}} = \frac{\mathcal{O}(\text{TD})}{N \times 3}. \tag{4-5}$$

The compression rate is an intuitive metric useful for comparing compression between tensor decomposition methods. It utilizes an element-based view similar to tensor decomposition methods, where compression is achieved by discarding an integer amount of elements.

4-2 Baseline Method: TMC3

Evaluating the baseline model is done by calculating the PSNR-NN and BPP for the 10 different samples taken at distinct timestamps from the VoD dataset. TMC3 has the option to perform lossy compression as well as lossless compression. Table 4-1 shows the performance of TMC3 on the VoD samples. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the samples. Lossy compression is evaluated for six different sets of compression parameters.

The table shows very high PSNR-NN values, which increase as the BPP increases. In other words, a better reconstruction quality means more bits are needed to represent the data. The time complexity increases for higher PSNR-NN values, except when using lossless compression. A caveat regarding lossless compression of TMC3 is that it is lossless with respect to a quantized representation of the data. In other words, the PSNR-NN is infinite (lossless) with respect to the quantized resolution.

Compression Type	PSNR-NN ↑	BPP ↓	Time (sec)
Lossy	52.5 ± 0.2	3.1 ± 0.2	0.09 ± 0.01
	58.4 ± 0.2	3.3 ± 0.2	0.13 ± 0.02
	70.0 ± 0.2	5.0 ± 0.4	0.38 ± 0.05
	75.6 ± 0.2	6.6 ± 0.5	0.59 ± 0.05
	87.4 ± 0.2	12.2 ± 0.7	0.96 ± 0.05
	93.4 ± 0.2	15.5 ± 0.7	1.07 ± 0.06
Lossless	106.6 ± 0.2	22.8 ± 0.7	0.52 ± 0.03

Table 4-1: Performance of TMC3 on samples from VoD dataset. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the 10 samples.

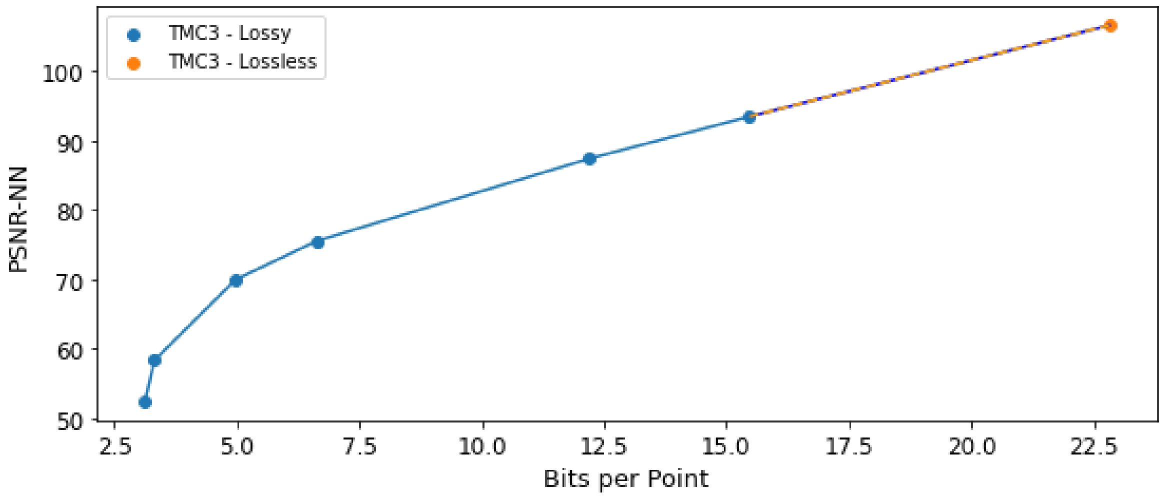


Figure 4-2: Performance curve of baseline model TMC3 on LiDAR samples from the VoD dataset using lossy and lossless compression.

The information in Table 4-1 can also be displayed visually in the form of a performance curve. Figure 4-2 shows this performance curve, which plots the PSNR-NN against the BPP. The further a codec is located in the top-left of this plot the better.

Figure 4-3 shows the result of applying TMC13 on a single LiDAR sample using lossless compression. The image shows the original LiDAR data in red, and the reconstructed LiDAR data in blue. The reconstruction is a strong match with the original data resulting in a high PSNR-NN value. Clicking on the link in the description or scanning the QR code will visualize a 3D render of the scene.

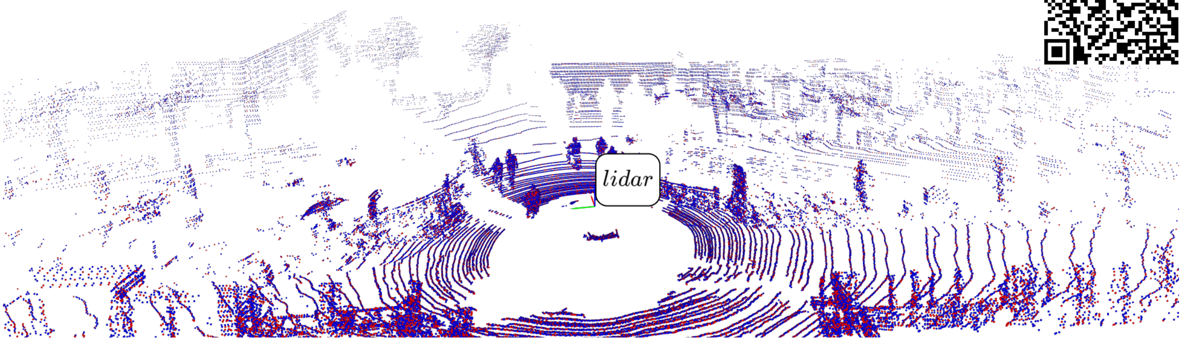


Figure 4-3: Original(red) and reconstructed using TMC3(blue) LiDAR data. PSNR-NN: 106.69. BPP: 23.47. Scan QR Code or click on link for 3D Render.

4-3 Voxel-Based Tensor Decomposition for Point Cloud Compression

4-3-1 Voxel-Based CPD

The first tensor decomposition that will be explored is the CPD obtained using the CP-ALS algorithm (Algorithm 1), visible in Appendix A-1-1. This is an iterative algorithm that computes the least squares solution for each mode-n matricization of the problem in alternating fashion. Because it is an iterative algorithm, it is important to first investigate whether the algorithm converges and (roughly) within how many iterations. Figure 4-4 shows the Frobenius error plotted against the iteration index for 6 different random initialization of the factor matrices $[B^{(1)}, B^{(2)}, B^{(3)}]$. The figure shows that all initializations converge and most do that within 5-10 iterations. They do however not converge to the same value, which indicates some or all runs get stuck in local minima.

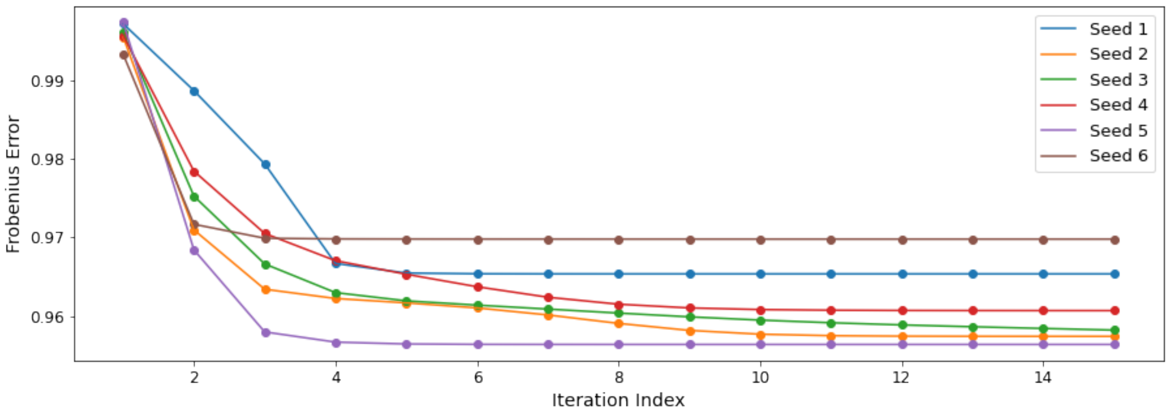


Figure 4-4: Convergence of CP-ALS: Relative frobenius error of the decomposition plotted against the iteration index for 6 random initializations. Most runs converge within 5-10 iterations.

Table 4-2 shows the performance of the CP-ALS algorithm using occupancy-based voxelization on a single sample. The table shows that when we increase the rank of the decomposition, the quality of the reconstruction increases. The CP-ALS algorithm ran for only 5 iterations, which already resulted in large computational times, specifically for high values of the rank. The high computational times can partially be attributed to the cost of devoxelizing the point cloud. This happens during each iteration and is the reason why finding a rank-1 decomposition already takes a long time. On the other hand, the computational time is affected by the requested rank of the decomposition. A larger rank results in larger factor matrices, which results in a larger system of equations that needs to be solved.

Rank	**1**	**5**	**10**	**20**	**50**	**75**
PSNR-NN ↑	18.34	20.94	21.17	21.63	22.37	22.40
Compression Rate ↓	1.08 %	5.41 %	10.81 %	21.61 %	54.03 %	81.05 %
BPP ↓	0.51	2.59	5.16	10.2	25.2	37.4
Time (seconds)	162	183	218	268	400	973

Table 4-2: Performance of CPD using occupancy-based voxelization on a single sample.

Figure 4-5 shows the qualitative result of applying the CP-ALS algorithm on an occupancy-based voxelization of a single sample. The image shows the original LiDAR in red and the reconstruction of the CPD in blue. The reconstruction is heavily centered around the origin, and points far away from this location are for the most part not reconstructed. The reason why reconstruction is favoured around the origin, is believed to be caused by the density of points in that area. The density of points is much higher close to the LiDAR reference frame, due to the nature of how the LiDAR points are collected.

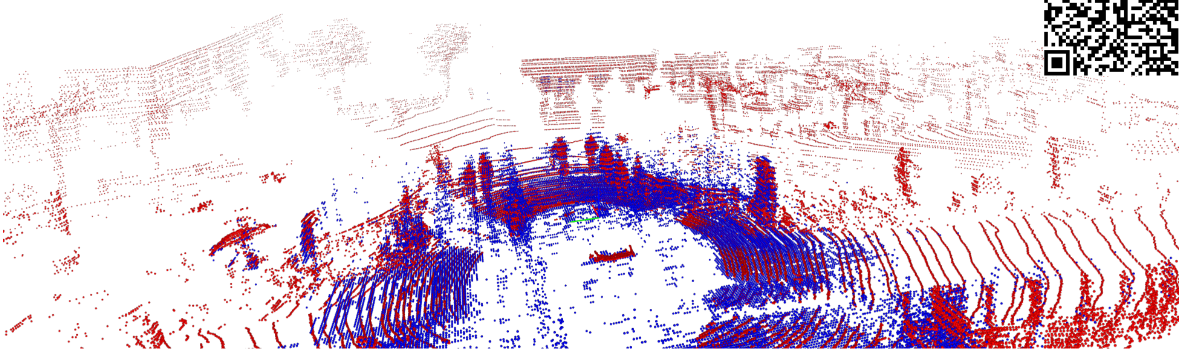


Figure 4-5: Original(red) and CPD(blue) of LiDAR using occupancy-based voxelization. PSNR-NN: 22.40. BPP: 37.42. Compression rate: 81.05%. Scan QR Code or click on link for 3D Render.

4-3-2 Voxel-Based Tucker Decomposition

Verifying MLSVD Algorithm Since the MLSVD algorithm is not an iterative algorithm like the CPD, verifying its implementation can easily be done by visualizing the reconstruction without any truncation. In other words, no singular values are being discarded. The data is merely being transformed into a different format, where the most dominant modes are in leading positions allowing for truncating the insignificant singular values. Figure 4-6 shows this visualization. Up to numerical precision this is an exact reconstruction. This is verified by the PSNR-NN of 64.20, which is exactly the same as the PSNR-NN of the voxelized representation in Figure 3-7.

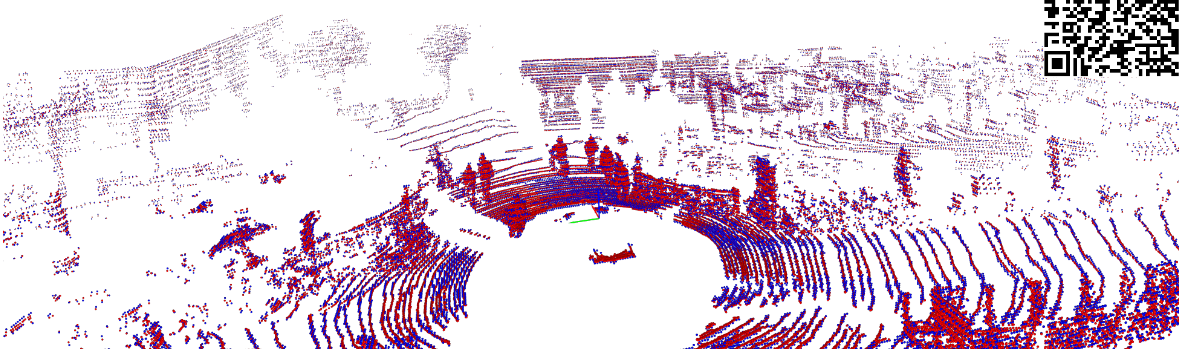


Figure 4-6: Original (red) and Tucker Decomposition without truncation (blue) of LiDAR using occupancy-based voxelization. PSNR-NN: 64.20. Scan QR Code or click on link for 3D Render.

Scree Analysis A useful tool that can be used to quantitatively determine what values to pick for the Tucker ranks is scree analysis [8]. The first step in scree analysis is to generate the scree plot. The scree plot displays the singular values for each mode-n unfolding of the tensor from large to small on a logarithmic scale. According to scree analysis, the relevant components/factors are located to the left of the elbow joint (point of maximum curvature) on a scree plot.

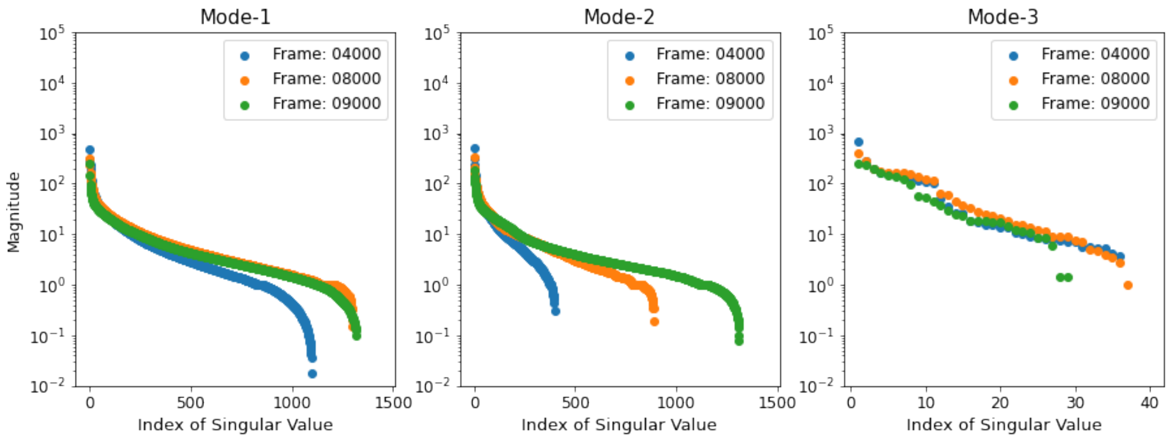


Figure 4-7: Singular Values of the Mode-n unfoldings for 3 Samples (Scree Plot). Singular value decline across samples is similar. The amount of singular values per mode varies per sample.

Figure 4-7 shows the scree plot of the occupancy-based voxelization for 3 different samples. The figure shows that mode-1 and mode-2 corresponding to the x- and y-axes embody relevant components, but also noise which does not contribute to describing the scene. This implies that a Tucker decomposition with truncations specifically in the first two modes could be used to reduce the complexity of the data without sacrificing too much in terms of reconstruction performance.

An interesting observation regarding Figure 4-7 is that the amount of singular values in mode-2 varies across the different point clouds. This phenomenon is directly related to the real world location of the vehicle, and more specifically to the view of the LiDAR sensor. Some streets have tall buildings close to the curb causing the emitted light pulses to directly reflect on the surface. Any objects behind these building are not captured using the LiDAR sensor, and are therefore unrepresented in the point cloud. Figure 4-8 shows the LiDAR point cloud belonging to *Frame: 04000* (•) in Figure 4-7. The image shows that the range of points in the y-direction is limited, causing fewer singular values to be present in the mode-2 matricization.

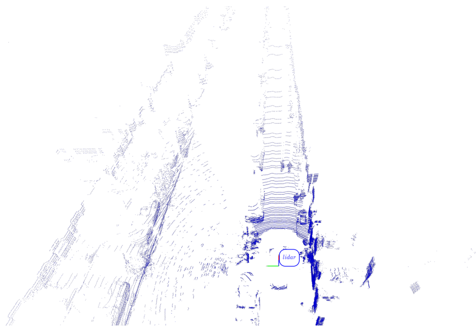


Figure 4-8: Snapshot of (narrow) LiDAR Point Cloud, *Frame: 04000*.

Table 4-3 shows the performance of the Tucker decomposition using binary and occupancy-based voxelization. The Tucker ranks, PSNR-NN, BPP, compression rate, and time needed to acquire the decomposition is shown. The table shows a number of interesting findings.

Tucker Ranks	PSNR-NN ↑		BPP ↓	Compression ↓ Rate	Time (sec)
	Binary	Occupancy			
(1440,1440,40)	64.1 ± 0.3	64.1 ± 0.3	15006 ± 1034	31591 ± 2153 %	37.9 ± 1.0
(1000,1000,35)	60.6 ± 2.8	60.5 ± 2.9	6560 ± 447	13665 ± 932 %	37.4 ± 1.0
(600,600,30)	44.8 ± 6.4	45.5 ± 6.7	2159 ± 147	4497 ± 307 %	38.1 ± 0.6
(300,300,25)	32.5 ± 4.7	34.4 ± 5.8	536 ± 37	1118 ± 76 %	37.9 ± 0.6
(62,62,12)	26.2 ± 3.2	26.7 ± 4.2	39.7 ± 2.7	82.7 ± 5.6 %	38.3 ± 0.8

Table 4-3: Performance of Tucker Decomposition using binary and occupancy-based voxelization. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the 10 samples.

First of all, the occupancy-based voxelization performs a little better compared to the binary voxelization but not by a significant margin. Secondly, the time it took to find each decomposition is roughly the same for all Tucker ranks, which is as expected since the time complexity of the MLSVD algorithm is dependent on the size of the tensor to be decomposed and not on the Tucker ranks (R_1, R_2, R_3). Finally, the most important finding is shown in the compression rate and BPP columns. The columns show that in order to acquire compression (compression rate $< 100\%$ or BPP < 46), the MLSVD needs to be truncated considerably. The Tucker ranks need to be reduced from $(1440, 1440, 40)$ to roughly $(66, 66, 14)$ for at least some compression to occur. The reason why this happens, is because the voxel-based representation of size $\mathbf{X} \in \mathbb{R}^{1440 \times 1440 \times 40}$ is highly voluminous and very sparse. Across the 10 samples the average amount of voxels that are non-empty is only 0.055 %, and the standard deviation is 0.008 %. The calculation of the compression rate is performed with respect to the size of the original LiDAR data which is of size $\mathcal{V} \in \mathbb{R}^{P \times 3}$. Hence, the voxel-based representation needs to truncate a lot of singular values before compression is achieved. Equation 4-6 shows the inequality that needs to be satisfied before compression is reached.

$$\begin{gathered} \mathcal{O}(\text{Tucker Decomposition}) < \mathcal{O}(\mathcal{V}) \\ I_1R_1 + I_2R_2 + I_3R_3 + R_1R_2R_3 < 3P \end{gathered} \tag{4-6}$$

Figure 4-9 shows the result of applying the Tucker decomposition with ranks $(R_1, R_2, R_3) = (62, 62, 12)$ onto the occupancy-based voxelization. A PSNR-NN of 22.27 and a compression rate of 80.48% is achieved. Similar to the results of the CPD, the reconstruction is heavily centered around the origin.

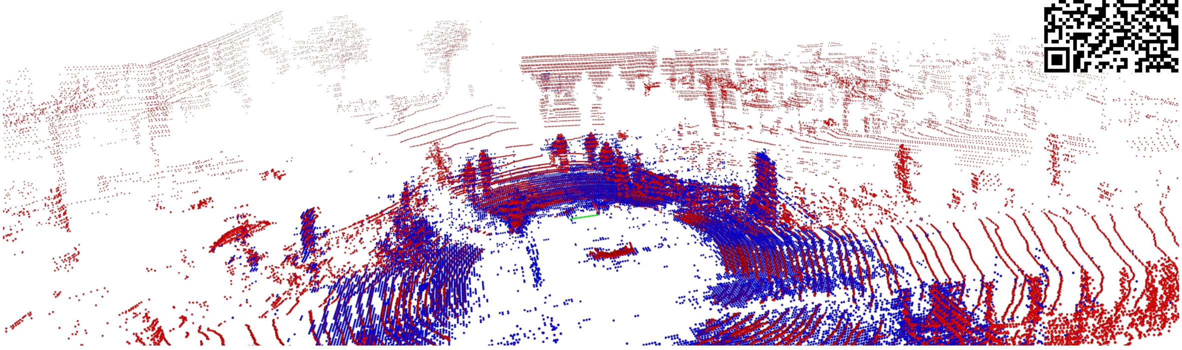


Figure 4-9: Original (red) and Tucker Decomposition with ranks $(R_1, R_2, R_3) = (62, 62, 12)$ of LiDAR using occupancy-based voxelization. PSNR-NN: 22.27. BBP: 38.63. Compression rate: 80.48%. Scan QR Code or click on link for 3D Render.

4-3-3 Voxel-Based Tensor Train Decomposition

Verifying TT-SVD Algorithm Similar to the MLSVD algorithm, the implementation of the TT-SVD algorithm can be verified by selecting a very small value for ϵ, which is the hyperparameter for setting the allowed relative error of the decomposition. With a very small ϵ, the TT-SVD algorithm will not be allowed to truncate any of the singular values in each of the modes. The resulting reconstruction should up to numerical precision be an exact reconstruction. Figure 4-10 verifies that the implementation is correct, since the PSNR-NN value of 62.22 is equal to voxel-based representation shown in Figure 3-7.

Table 4-4 shows the results of applying the TT-SVD algorithm onto binary and occupancy-based voxelizations for the 10 different samples. The table shows a few noteworthy observations. The occupancy-based voxelization performs slightly better than the binary voxelization, except when a high ϵ value is present. Similar to the Tucker decomposition, the TT-SVD algorithm has to truncate a very large amount of singular values before any compression is

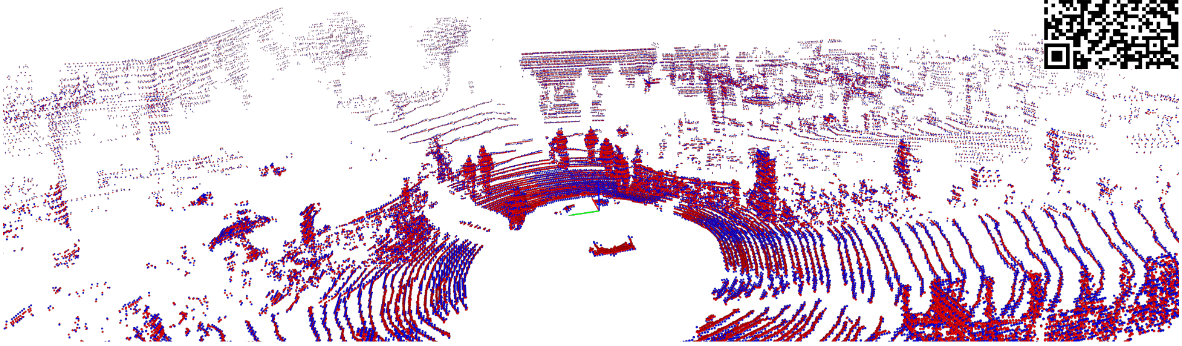


Figure 4-10: Original (red) and TT Decomposition without truncation (blue) of LiDAR using occupancy-based voxelization. PSNR-NN: 64.20. Scan QR Code or click on link for 3D Render.

achieved. This is again related to the highly voluminous and very spare representation of the tensor. Interesting to note is that the time complexity of the TT-SVD algorithm reduces as ϵ increases. This happens because when ϵ is increased, the amount of singular values that get truncated in each mode-n SVD get increased as well. This has a cascading effect, since the algorithm uses the result from the truncated SVD in the current mode as input for the next mode, which will then have fewer elements and therefore take less computational time.

ϵ	**PSNR-NN** $\uparrow$		**Compression Rate** (%) $\downarrow$		**BPP** $\downarrow$		**Time (sec)**
	Binary	**Occup.**	**Binary**	**Occup.**	**Binary**	**Occup.**	
0.01	64.1±0.3	64.1±0.3	24287±4277	24734±4200	11658±2053	11872±2016	39.9±2.2
0.1	63.7±0.3	64.0±0.3	21610±3925	23126±3985	10373±1884	11100±1913	39.6±2.4
1	58.2±2.1	61.7±2.1	14553±3425	18101±3336	6985±1644	8688±1601	38.1±2.2
10	42.3±3.1	49.0±2.7	5122±1594	8357±1674	2458±765	4011±804	35.4±1.8
100	28.6±2.9	30.4±1.7	748±380	559±269	359±182	268±129	32.7±1.2
290	25.4±2.0	19.4±1.6	76.9±3.2	33.1±3.0	36.9±1.6	15.9±1.4	32.8±1.3

Table 4-4: Performance of TT Decomposition using binary and occupancy-based voxelization. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the 10 samples.

Figure 4-11 shows the qualitative result of applying the TT-SVD algorithm on an occupancy-based voxelization. The decomposition achieves a PSNR-NN of: 28.90, and a compression rate of 80.38%. Similar to the CPD and Tucker decomposition, the reconstruction is heavily centered around the origin, which is believed to be caused by the high density of points in that region.

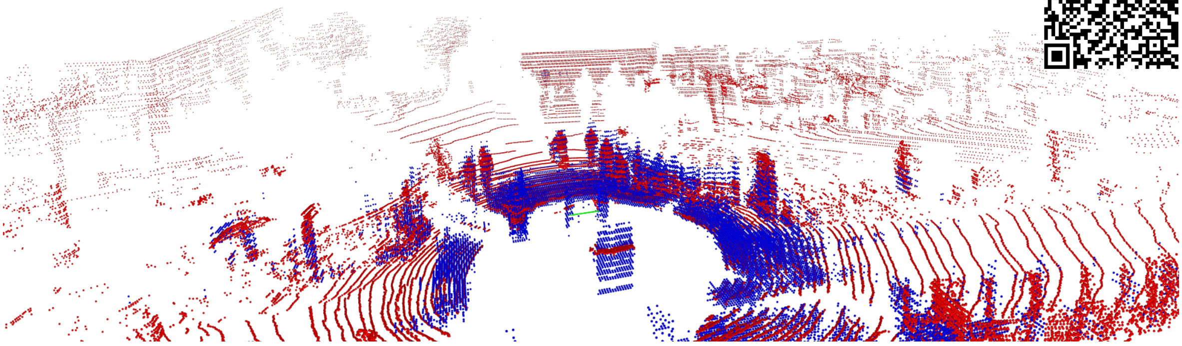


Figure 4-11: Original (red) and TT Decomposition of occupancy-based voxelization. PSNR-NN: 24.27. BPP: 34.21. Compression rate: 71.27%. Scan QR Code or click on link for 3D Render.

4-3-4 Tensorized Voxelizations

A possible solution to the reconstructions which are heavily centered around the origin of the CP, TT, and Tucker decomposition is to tensorize the voxelized representation as described in Subsection 3-2-1. This tensorization of the voxelization could aid in exploiting more of the similarities in local geometry.

Voxel-Based Tensorization of Tucker Decomposition

Table 4-5 shows the performance of applying the MLSVD algorithm onto the tensorization of the voxelized representation for the 10 different samples. The table shows that the performance of the occupancy-based voxelization is slightly better than the binary voxelization.

Tucker Ranks	PSNR-NN ↑		BPP ↓	Compression ↓ Rate	Time (sec)
	Binary	Occupancy			
(12,12,12,12,11)	26.0 ± 2.7	26.3 ± 2.4	41.7 ± 2.8	86.9 ± 5.9 %	38.3 ± 0.8

Table 4-5: Performance of Tucker Decomposition using tensorization of binary and occupancy-based voxelization. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the 10 samples.

Figure 4-12 shows the result of applying the MLSVD algorithm onto the tensorized occupancy-based voxelization. Compared to the regular occupancy-based voxelization (Figure 4-9), an increase in terms of PSNR-NN from 22.27 to 26.59 is obtained. Additionally, the reconstruction is less centered around the origin, but the result is not very signicant.

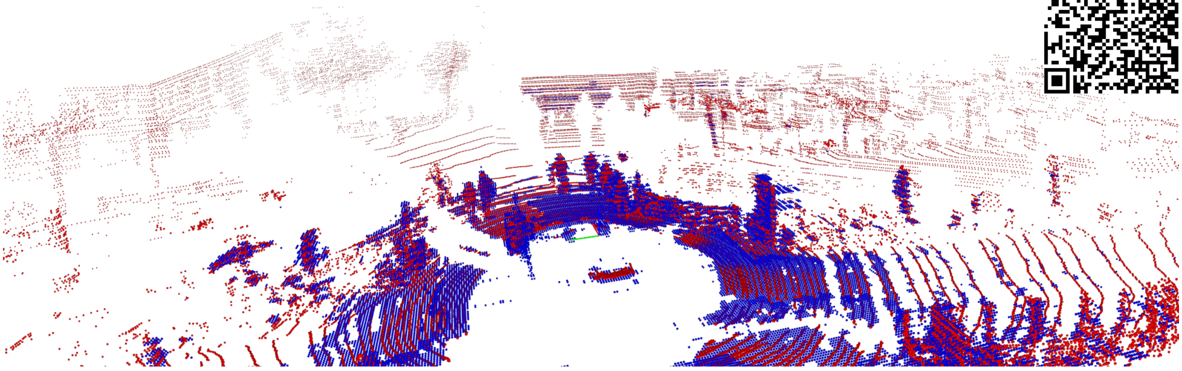


Figure 4-12: Original (red) and Tucker Decomposition with Tucker-ranks $(R_1, R_2, R_3, R_4, R_5) = (12, 12, 12, 12, 11)$ of LiDAR using tensorized occupancy-based voxelization. PSNR-NN: 26.59 Compression rate: 81.89 %. Scan QR Code or click on link.

Voxel-Based Tensorization of Tensor Train Decomposition

Table 4-6 shows the performance of the TT-SVD algorithm using a tensorized binary or occupancy-based voxelization. The table shows a very high standard deviation for the compression rate of both the binary and occupancy-based voxelizations. This is caused by the parameter ϵ, which is quite sample specific. The same value for ϵ applied on two different samples, can result in a large difference in compression ratio.

ϵ	PSNR-NN ↑	BPP ↓	Compression Rate ↓	Time (sec)
		Binary		
205	23.1 ± 2.6	25.9 ± 30.6	53.9 ± 63.7 %	26.4 ± 1.9
		Occupancy		
185	23.3 ± 1.8	25.8 ± 18.6	53.8 ± 38.7 %	24.1 ± 3.0

Table 4-6: Performance of TT Decomposition using tensorization of binary and occupancy-based voxelization. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the 10 samples.

Figure 4-13 shows the result of applying the TT-SVD algorithm onto a tensorized occupancy-based voxelization. The algorithm achieves a compression ratio of 81.09%, and a PSNR-NN of 25.03. Compared to the TT-SVD of the regular occupancy-based voxelization (Figure 4-11) a drop in PSNR-NN of 28.90 to 25.03 is observed.

4-3-5 Discussion

Table 4-7 shows the results of all the discussed tensor decompositions using a voxel-based representation for a comparable compression rate. The table shows a few noteworthy findings. None of the tensor decomposition methods significantly outperform each other based

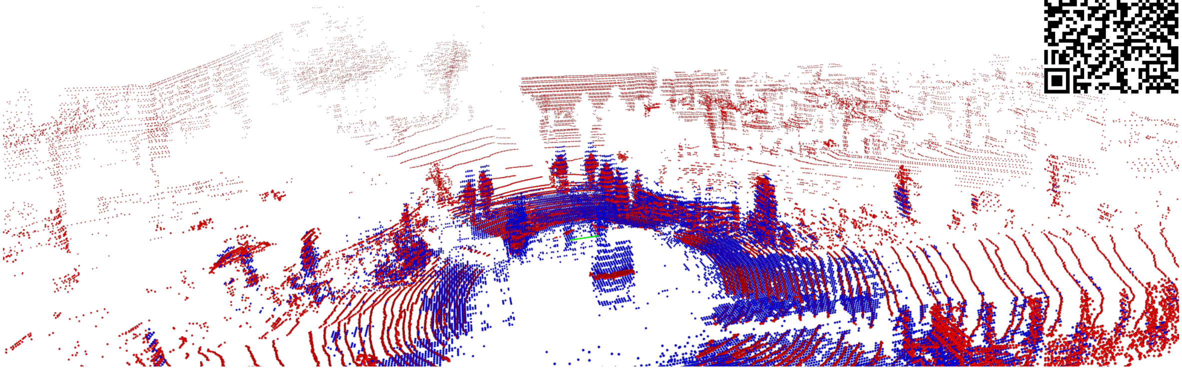


Figure 4-13: Original (red) and TT Decomposition with TT-ranks $(R_1, R_2, R_3, R_4) = (7, 89, 56, 5)$ of LiDAR using tensorized occupancy-based voxelization. PSNR-NN: 25.03. Compression rate: 81.09%. Scan QR Code or click on link for 3D Render.

on PSNR-NN. Binary or occupancy-based voxelization does not yield a significantly different result. The CPD has by far the longest computational time, due to it being an iterative algorithm.

Method	PSNR-NN ↑		Compression Rate ↓		Time (sec)
	Binary	Occupancy	Binary	Occupancy	
CPD	21.9	22.4	81.1 %	81.1 %	973
Tucker	26.2 ± 3.2	$\mathbf{26.7 \pm 4.2}$	82.7 ± 5.6 %	82.7 ± 5.6 %	38.3 ± 0.8
TT	25.4 ± 2.0	19.4 ± 1.6	76.9 ± 3.2 %	33.1 ± 3.0 %	32.8 ± 1.3
Tucker Tens.	26.0 ± 2.7	26.3 ± 2.4	86.9 ± 5.9 %	86.9 ± 5.9 %	38.3 ± 0.8
TT Tens.	23.1 ± 2.6	23.3 ± 1.8	53.9 ± 63.7 %	53.8 ± 38.7 %	24.1 ± 3.0

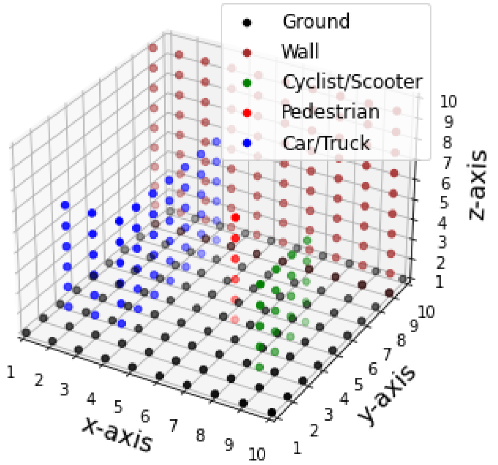
Table 4-7: Performance of Tucker Decomposition using binary and occupancy-based voxelization. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over the 10 samples.

There are multiple possible reasons as to why the performance of the investigated voxel-based approach does not yield satisfactory results. These reasons will be explained below.

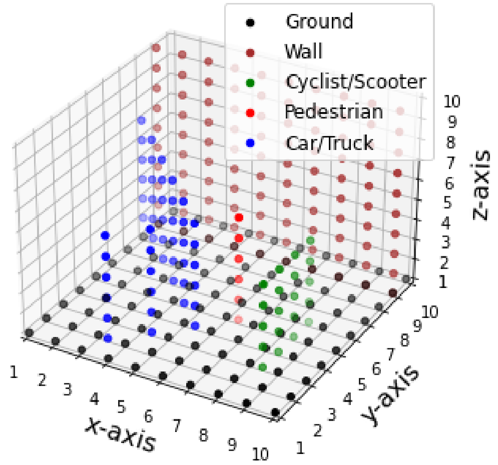
Rotational Variance The idea of applying tensor decomposition methods on a voxel-based representation stems from the analysis presented in Figure 3-5. The analysis showed that a simple rank-6 CPD could be used to describe an abstraction of the 3D scene. The problem with this reasoning is that it assumes road users and road elements are aligned with the coordinate axis. A low-rank CPD could be constructed because of this alignment. In real life however, objects are most often not aligned with the coordinate axis. Hence, finding a low-rank CPD (or any other tensor decomposition) that is an adequate representation of the scene might be impossible.

To test this hypothesis let us revisit the abstraction of the scene depicted in Figure 4-14a.

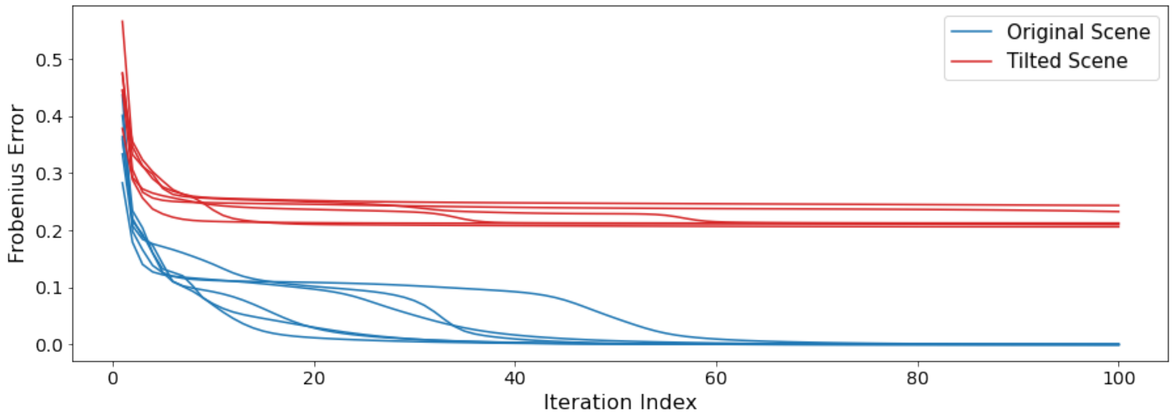
However, instead of aligning all elements with the coordinate axis, consider that the car/truck is rotated like shown in Figure 4-14b. Figure 4-14c shows the frobenius error plotted against the iteration index for 6 different runs of the CP-ALS algorithm applied on the original scene and the scene with the tilted truck. The figure shows that the algorithm is able to find a rank-6 decomposition with negligible small error for the original scene, while it does not for the tilted scene. Alignment with the coordinate axis is thus of vital importance for finding a low-rank CP decomposition which is a valid approximation of the scene.



(a) Original Scene.



(b) Scene with tilted car/truck.



(c) Convergence of CP-ALS algorithm on the original scene and tilted scene when searching for a rank-6 decomposition. The algorithm is not able to find a suitable decomposition for the tilted scene, due to rotation of the car/truck.

Figure 4-14: Rotational Variance of CPD on Voxelized Representation.

Sparse Representation The voxel-based representation is a highly voluminous and very sparse representation. As already mentioned before, the amount of voxels that are occupied across the 10 samples is only 0.055 % on average. Compared to the original size of the point cloud $\mathcal{V} \in \mathbb{R}^{P \times F}$ the voxel-based representation $\mathbf{X} \in \mathbb{R}^{1440 \times 1440 \times 40}$ contains much more elements. In mathematical terms: $1440 \cdot 1440 \cdot 40 >> PF$. This has as an affect that in order to acquire compression a very large amount of singular values needs to be truncated. Effectively, the representation is being blown up by voxelizing the point cloud, causing the amount of truncation needed to increase considerably in order to achieve compression.

There are multiple approaches to tackle the problem of the very sparse representation. One method is to apply sparse tensor decomposition methods [41],[49],[67],[78]. These methods are designed to account for the sparsity in the data. Another approach is to alter the data representation into a different format. Instead of voxelizing the point cloud, direct tensorization can be applied onto the LiDAR data [61]. The next section will discuss this approach. A big advantage of this approach is that compression is directly achieved, even with little truncation of the singular values.

4-4 Synthetic Tensor Decompositions for Point Cloud Compression

Table 4-8 shows the result of applying the CP-ALS, MLSVD, and TT-SVD on 10 different samples tensorized using parameters: $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$. Based off Table 4-8, the performance in terms of PSNR-NN increased with respect to the voxel-based methods, and the computational time of the Tucker and TT have reduced considerably. The TT decomposition achieves the highest average PSNR-NN.

Method	Rank/ϵ	PSNR-NN ↑	BPP ↓	Compression Rate ↓	Time (sec)
CPD	1240	40.8 ± 0.9	37.8	80.1 %	47.2 ± 1.2
Tucker	(37,37,37,4)	36.6 ± 1.1	38.0	79.1 %	0.023 ± 0.004
TT	12	**49.3 ± 0.8**	38.4 ± 2.6	79.9 ± 5.4 %	0.119 ± 0.001

Table 4-8: Performance of tensor decomposition methods using synthetic tensorization parameter $(I_1, I_2, I_3, I_4) = (40, 40, 40, F)$ on 10 different samples.

Table 4-8 does however not show the full story. Figure 4-15 shows a visualization of applying the CP-ALS, MLSVD, and TT-SVD algorithms onto synthetically tensorized LiDAR data using parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$.

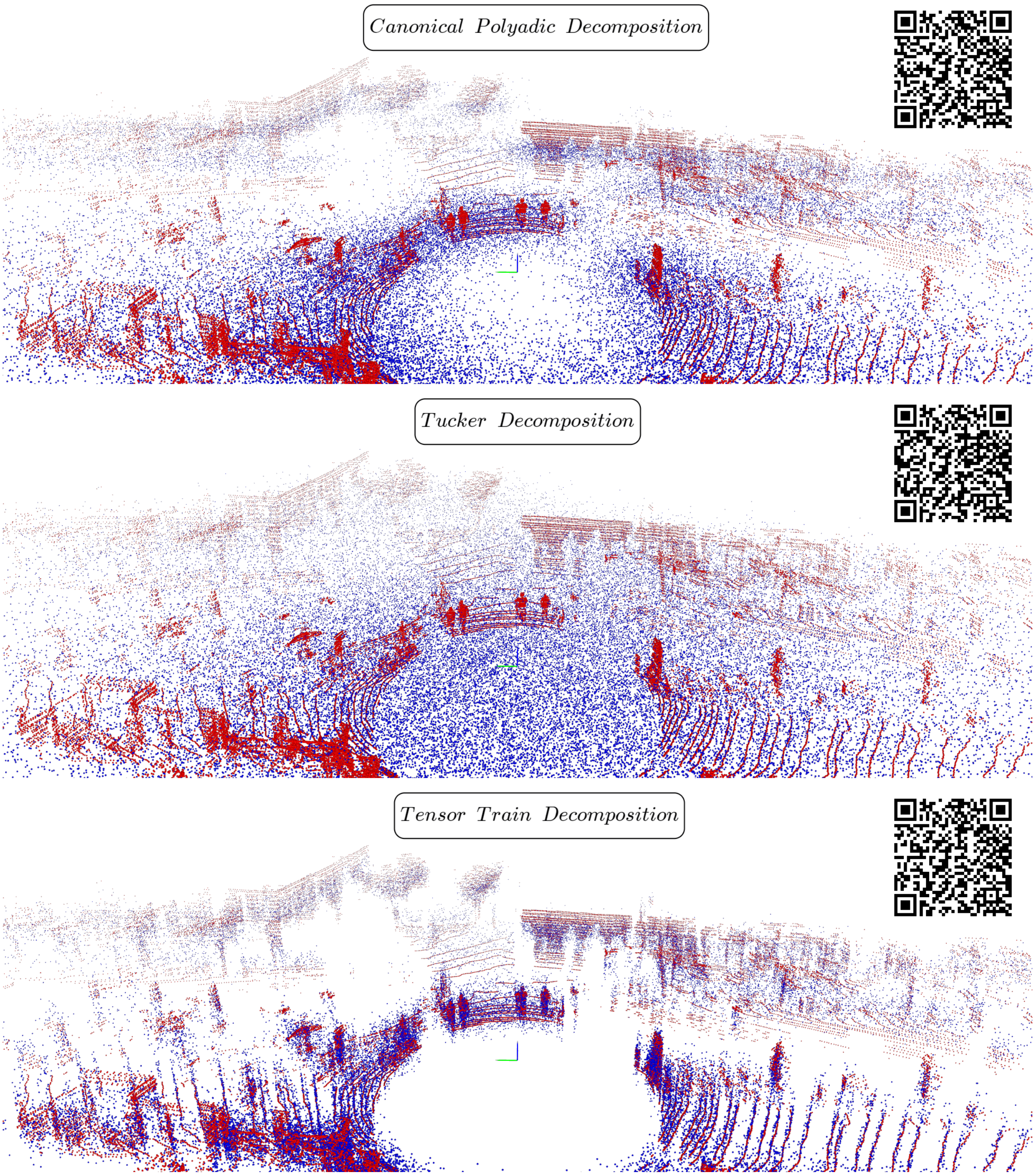


Figure 4-15: Original (red) and Tensor Decomposition (blue) of LiDAR using synthetic tensorization parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, F)$. Scan QR Code or click on the links for a 3D render: CPD, Tucker, TT.

The figure shows that the Tucker decomposition is completely unable to capture the structure of the LiDAR data. The CPD is able to capture some of the local density in points. The TT decomposition performs the best, however it is far from a perfect reconstruction.

The reason why the performance of the CP-ALS, MLSVD, and TT-SVD algorithms is relatively weak, is related to the structure of the (synthethically) tensorized LiDAR data. During tensorization of the LiDAR data, the ordering of the points in the cloud exactly determines which location the point will inhabit in the tensor. In other words, a one-to-one mapping by means of the little-endian convention is used to map elements from the raw LiDAR data to the tensorized representation. The results (both table and figure) above were obtained using this one-to-one mapping. However, the ordering of the points in the cloud was not set. This means that the placement of points in the tensor was effectively done at random. This random placement of points increases the difficulty for tensor decomposition methods to find a low-rank decomposition that is a good approximation of the LiDAR data.

Figure 4-16 shows a scree plot of the synthetically tensorized representation using parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ for 100 different random seeds (used when sorting the LiDAR data at random). One of these seeds is used to obtain the results shown in Figure 4-15.

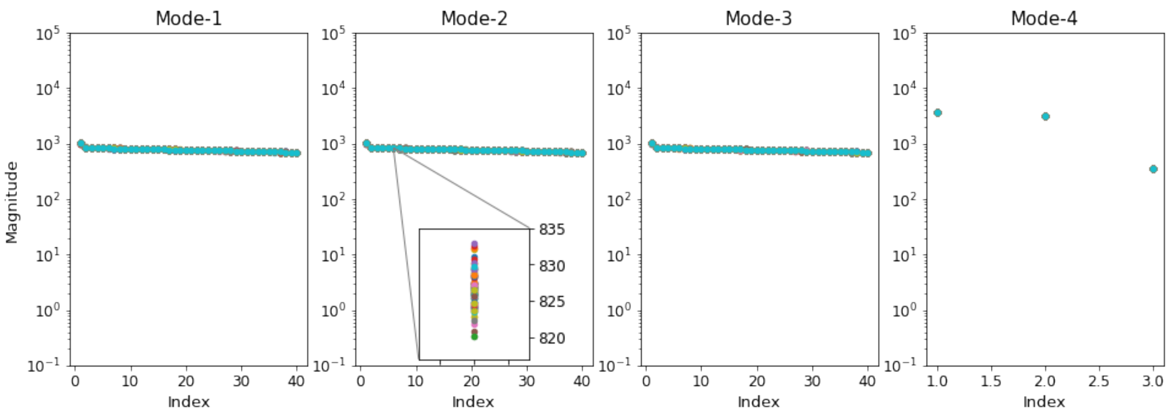


Figure 4-16: Singular values of mode-n unfoldings using synthetically tensorized LiDAR data with parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ for 100 different enumerations of points in the cloud. The difference in singular value decline is virtually zero.

Figure 4-16 corroborates the reasoning presented earlier, since apart from the first singular value all others show negligible difference in magnitude for the first 3 modes, which implies that finding a good low-rank Tucker decomposition will be difficult.

Figure 4-16 shows the scree plot for 100 different enumerations of vectors using the same sample. Figure 4-17 on the other hand, shows the scree plot using a single enumeration for the 10 different samples. The figure demonstrates that the almost flat singular value decrease of the first 3 modes is present across all samples. This finding eliminates the possibility of the behaviour being sample specific.

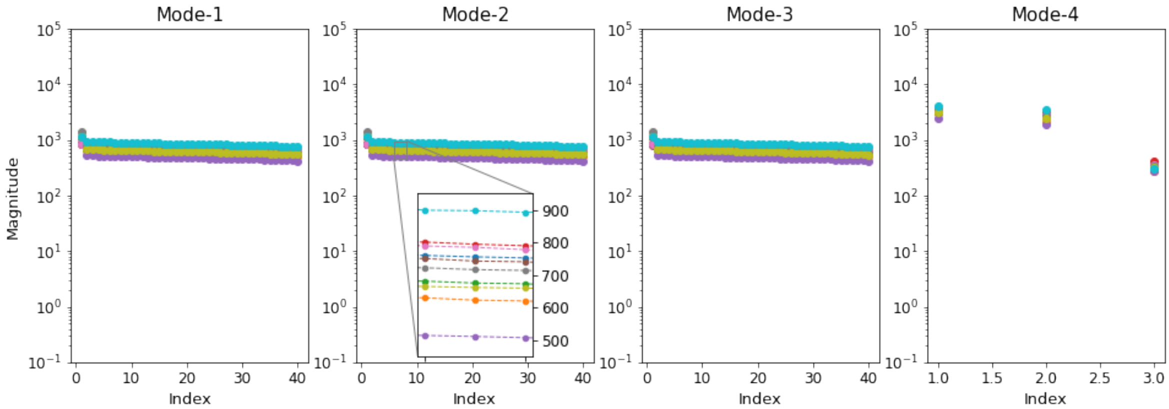


Figure 4-17: Singular values of mode-n unfoldings using synthetically tensorized LiDAR data with parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ for 10 different samples. Singular value decline across samples is highly comparable.

A crucial question now becomes how to define the enumeration of points in the point cloud that results in a tensorized representation which is susceptible to tensor decomposition methods. For a point cloud $\mathcal{V} \in \mathbb{R}^{P \times F}$ with P points there are $P\,!$ possible permutations to order the list of points.

Exploring all the possible permutations is an intractable problem, since point clouds can easily contain ~100000 points, meaning $100000! \approx 2.8 \cdot 10^{456573}$ unique permutations exist. Because of this, alternative approaches must be considered for bringing structure into the LiDAR data during synthetic tensorization.

Common practice when tackling problems where exploring all options is too exhaustive, is to apply some sort of heuristic. This heuristic can be used to sort the LiDAR data prior to synthetic tensorization. By default, a LiDAR point cloud $\mathcal{V} \in \mathbb{R}^{P \times F}$ consists of F features per point. An easy-to-implement heuristic, is to sort the LiDAR point cloud by one of these F features. The VoD dataset contains LiDAR points with 4 features. These are the X-location, Y-location, Z-location, and the reflectance. All of these features are considered as heuristics for sorting the point cloud. Figure 4-18 shows a visualization of these sort methods. The LiDAR points are colored based on their position in the sorted point cloud. A (dark) blue point is situated in the start of the list, while (dark) red points are at the end.

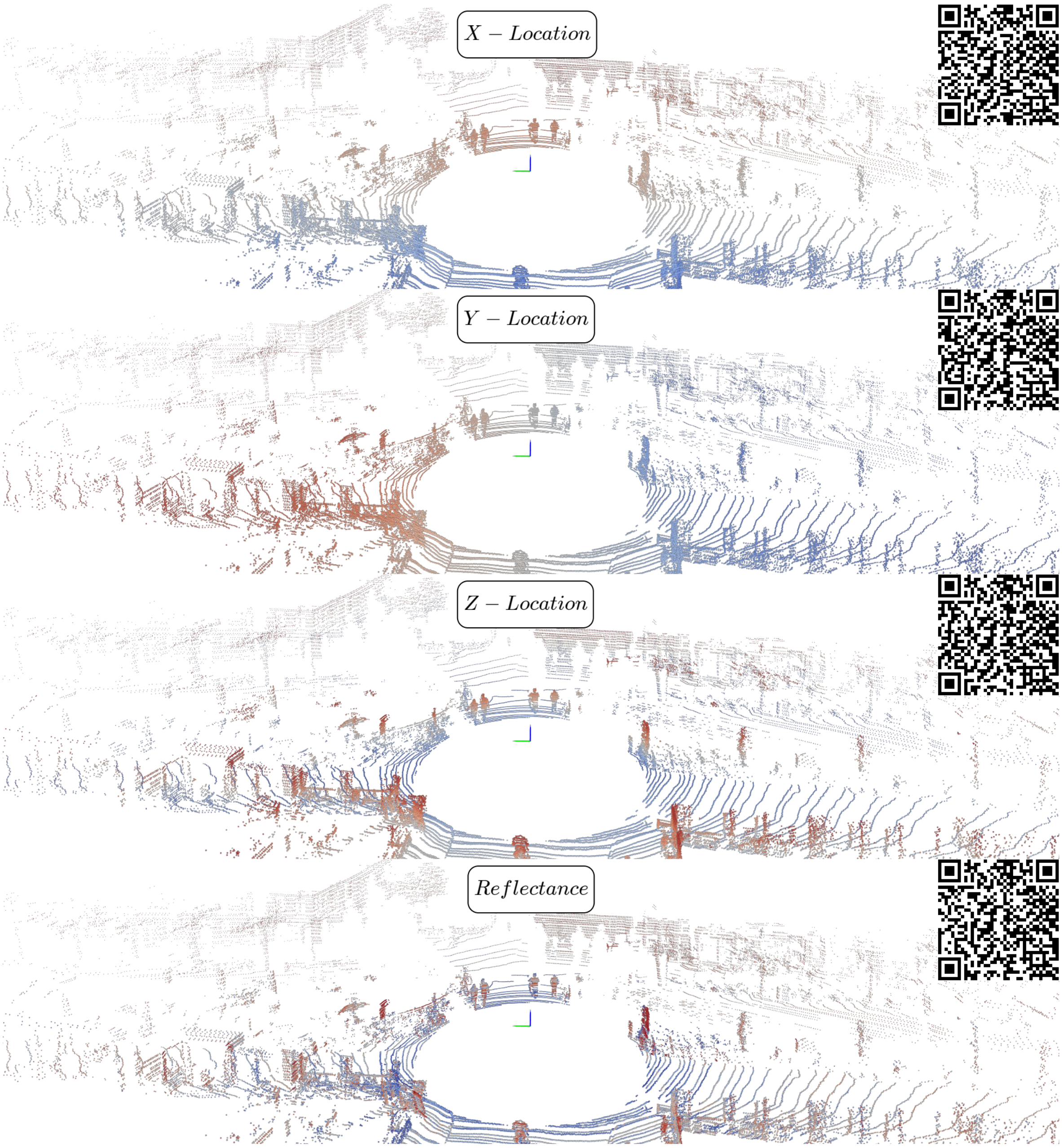


Figure 4-18: Visualizations of the heuristics based on point features employed for sorting the LiDAR point cloud. Scanning the QR code or clicking on the following links will visualize a 3D render: X-Location, Y-Location, Z-Location, Reflectance.

Apart from the existing features, heuristics can also be defined as functions of these features. Equation 4-7 and Equation 4-8 show the elevation (θ) and azimuth (ϕ) angle respectively. Both of these angles are also considered as heuristics.

$$\theta = \arctan\left(\frac{z}{\sqrt{x^2 + y^2}}\right) \tag{4-7}$$

$$\phi = \text{arctan2}(y, x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & \text{if } x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & \text{if } x < 0 \text{ and } y \geq 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & \text{if } x < 0 \text{ and } y < 0 \\ +\frac{\pi}{2} & \text{if } x = 0 \text{ and } y > 0 \\ -\frac{\pi}{2} & \text{if } x = 0 \text{ and } y < 0 \\ \text{undefined} & \text{if } x = 0 \text{ and } y = 0 \end{cases} \tag{4-8}$$

Lastly, Equation 4-9 shows the angle (ψ) between two vectors $\mathbf{v}_a$ and $\mathbf{v}_b$ in 3D space. By treating the points in the cloud as vectors, the angle between each point can be calculated.

$$\psi = \arccos\left(\frac{\mathbf{v}_a^T \mathbf{v}_b}{\mid \mathbf{v}_a \mid \cdot \mid \mathbf{v}_b \mid}\right) \tag{4-9}$$

This then allows for defining two more heuristics. One of them orders all points by means of the angular difference towards a single (initial) point. This method is labelled as: Angular Difference - Single Vector. The other method orders all points by finding the next point which has the smallest angular difference towards the previous point. This is done consecutively for all points, resulting in the smallest step in angular difference (ϕ) between successive points. This method is labelled as: Angular Difference - Consecutive Vectors.

Figure 4-19 shows a visualization for these 4 different heuristics that will be considered. Identical to Figure 4-18, the ordering of points is denoted by the color scale which runs from blue to red.

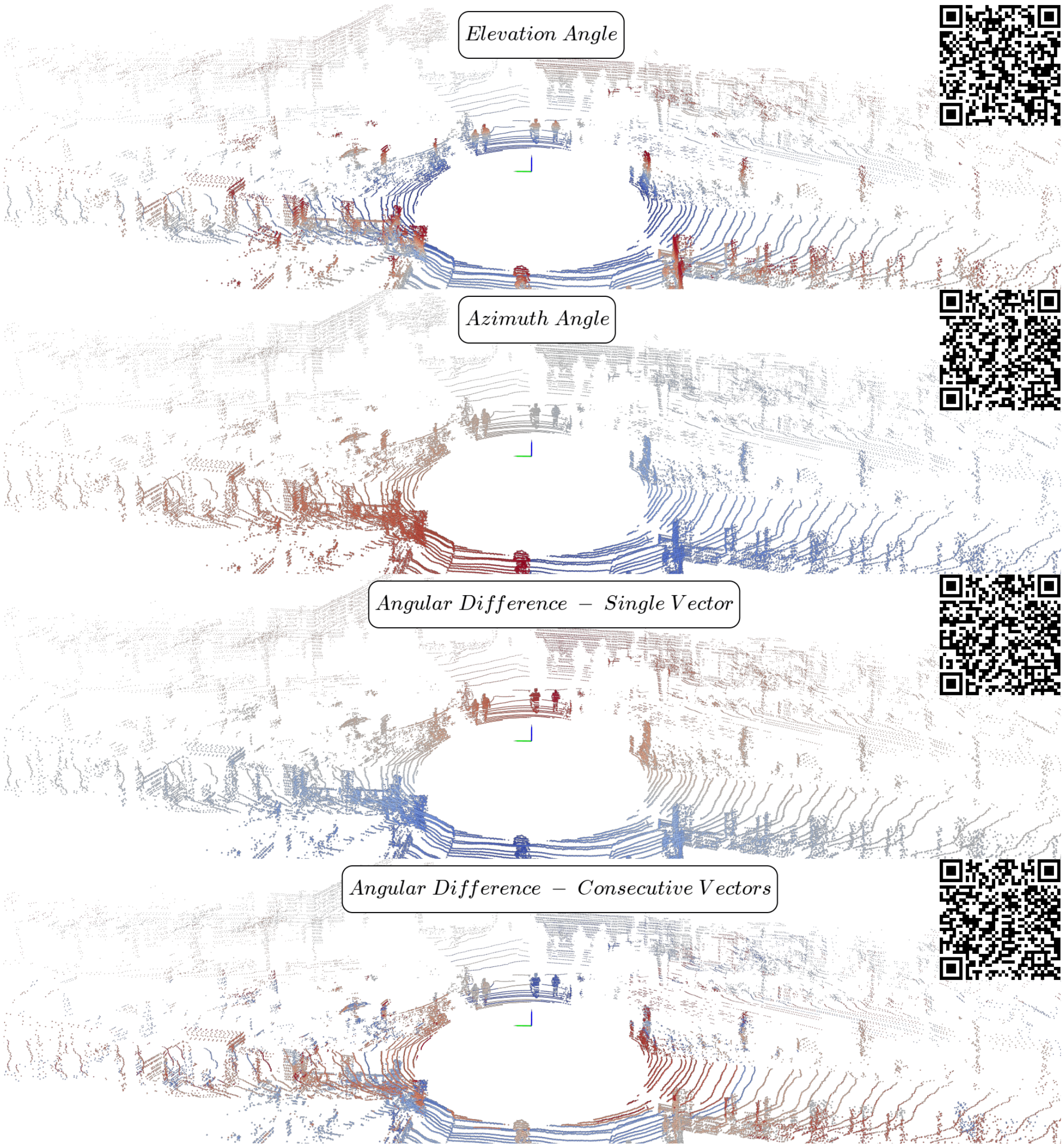


Figure 4-19: Visualizations of the angular heuristics employed for sorting the LiDAR point cloud. Scanning the QR code or clicking on the following links will visualize a 3D render: Elevation Angle, Azimuth Angle, Angular Difference - Single Vector, Consecutive Difference - Single Vector.

Figure 4-20 shows the singular values of the mode-n unfoldings for 9 different sorting methods averaged over the 10 samples. The figure clearly shows that sorting using the angular difference of consecutive vectors (mustard green) results in the highest curvature of the scree plot. Hence, this heuristic would be most promising to apply before synthetically tensorizing the LiDAR data and computing a Tucker decomposition.

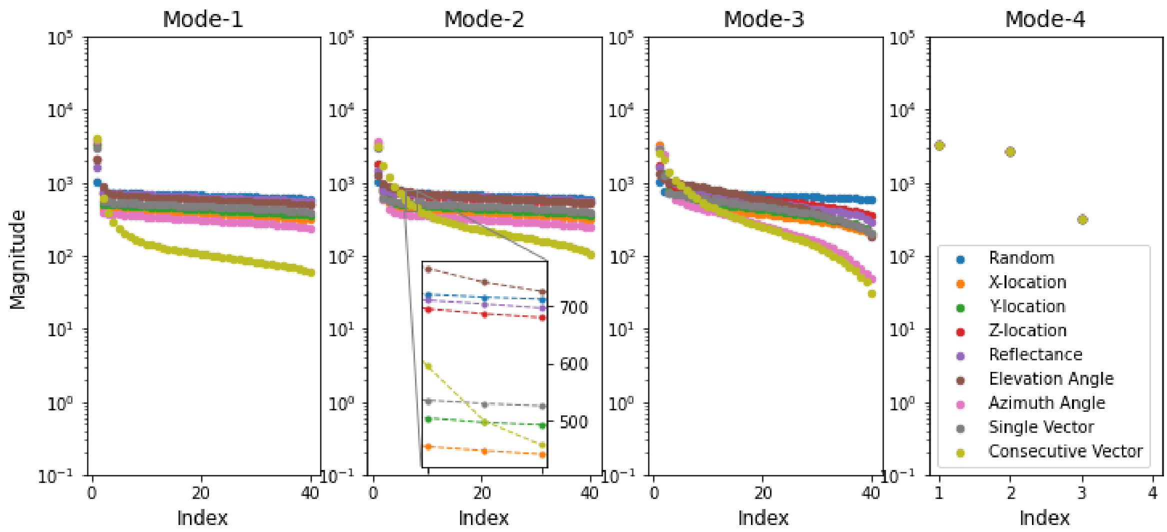


Figure 4-20: Scree plot for 9 different sorting methods averaged over 10 samples. Consecutive Vector method displays the most singular value decline.

Table 4-9 shows the performance of applying the TT-SVD and MLSVD algorithm onto synthetically tensorized LiDAR for all of the mentioned heuristics. The table verifies the findings of the scree analysis in Figure 4-20, since the best performing heuristic for the Tucker decomposition is the Angular Difference - Consecutive Vectors method. The TT decomposition outperforms the Tucker decomposition for all sort methods. The best performing heuristic is sorting by z-value, which results in a PSNR-NN of 110.6 on average. Virtually all sort methods show an improvement in terms of PSNR-NN compared to randomly ordered LiDAR data.

The computational time of each method is quite low except when using the Angular Difference - Consecutive Vectors heuristic. This sort method needs to consecutively find the closest point in terms of angular difference. This means it has to solve a nearest neighbour problem $P-1$ times, where P is the amount of points. The problem does decrease in size every step, since every nearest neighbour that is found will not be considered in the next step.

Figure 4-21 shows a visualization of TT-SVD algorithm applied on synthetically tensorized LiDAR with parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ and sorted using the z-values of the LiDAR points. The image shows a great improvement in terms of reconstruction quality compared to previous methods, which is also reflected in the much higher PSNR-NN: 109.50.

Sort	Method	Rank/ϵ	PSNR-NN ↑	BPP ↓	Compression Rate ↓	Time (sec)
x	Tucker	(37,37,37,3)	40.0 ± 1.0	38.0	79.1 %	1.2 ± 0.1
	TT	0.001	88.1 ± 1.2	36.8 ± 0.3	76.6 ± 0.6 %	1.3 ± 0.1
y	Tucker	(37,37,37,3)	40.9 ± 2.4	38.0	79.1 %	1.2 ± 0.1
	TT	0.0005	92.0 ± 2.6	36.6 ± 0.4	76.2 ± 0.9 %	1.3 ± 0.1
z	Tucker	(37,37,37,3)	37.5 ± 1.3	38.0	79.1 %	1.2 ± 0.1
	TT	0.000005	$\mathbf{110.6 \pm 1.1}$	36.5 ± 0.2	76.0 ± 0.4 %	1.3 ± 0.1
r	Tucker	(37,37,37,3)	36.0 ± 1.1	38.0	79.1 %	1.2 ± 0.1
	TT	12	50.1 ± 0.6	36.9 ± 2.0	76.8 ± 4.1 %	1.3 ± 0.1
θ	Tucker	(37,37,37,3)	37.2 ± 1.2	38.0	79.1 %	1.2 ± 0.1
	TT	1.3	58.0 ± 1.2	36.7 ± 1.8	76.4 ± 3.8 %	1.3 ± 0.1
ϕ	Tucker	(37,37,37,3)	42.4 ± 1.9	38.0	79.1 %	1.2 ± 0.1
	TT	0.7	61.3 ± 1.1	36.9 ± 1.6	76.9 ± 3.3 %	1.3 ± 0.1
ψ_{SV}	Tucker	(37,37,37,3)	39.0 ± 1.3	38.0	79.1 %	1.2 ± 0.1
	TT	6.5	52.5 ± 1.2	36.5 ± 2.6	76.1 ± 5.4 %	1.3 ± 0.1
ψ_{CV}	Tucker	(37,37,37,3)	$\mathbf{47.9 \pm 2.1}$	38.0	79.1 %	140 ± 2
	TT	0.28	64.3 ± 0.9	36.8 ± 1.9	76.6 ± 4.0 %	141 ± 1

Table 4-9: Performance of Tucker and TT decomposition when sorting using angular difference of consecutive vectors prior to synthetic tensorization with parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, F)$. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over 10 different samples.

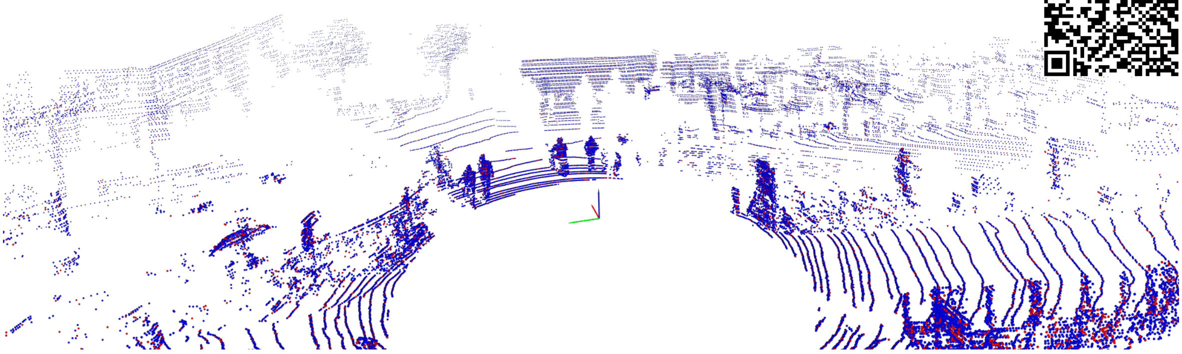


Figure 4-21: Original(red) and TTSVD(blue) of synthetically tensorized LiDAR with parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ and sorted using z-values. PSNR-NN: 109.50. BPP: 36.67. Compression rate: 76.41%. Scan QR Code or click on link for 3D Render.

Table 4-9 showed the performance of the TT and Tucker decomposition for all heuristics using synthetic tensorization parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ evaluated around a compression rate of 78%. This does however not show the full story, since performance should be evaluated over a range of compression values and there are various options for tensorization parameters.

Figure 4-22 shows the performance of the TT-SVD algorithm for all of the heuristics and sets of tensorization parameters evaluated over a range of compression values. The figure also shows the performance of the baseline model (TMC3) in brown. The figure shows a number of interesting findings. Regarding large amount of compression (BPP $\in [0, 30]$), sorting by angular difference of consecutive vectors results in the best performance using a tensor decomposition method. For a relatively small amount of compression (BPP $\in [30, 40]$) sorting by z-values is the best performing tensor decomposition method, which corroborates the findings in Table 4-9. The baseline model outperforms all tensor decomposition methods over its entire range of compression values. The choice for the synthetic tensorization parameters does not show a very large impact with regards to compression performance.

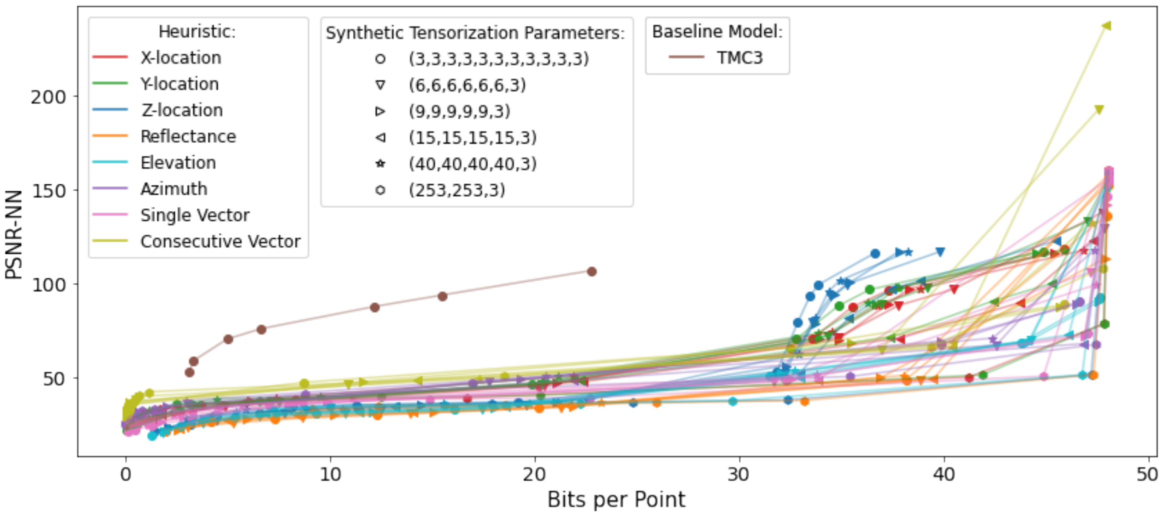


Figure 4-22: Performance curve for the TT-SVD algorithm using various sets of synthetic tensorization parameters and sort heuristics. The baseline model (TMC3) outperforms all tensor decomposition methods over the range of compression values.

As already mentioned the two best performing methods are sorting by z-value for small amounts of compression and by angular difference of consecutive vectors for large amounts of compression. The reasoning why these methods stand out is as follows. Sorting point clouds by z-value causes all points of similar height to be contiguously stored. This makes it easier for tensor decomposition methods to identify and exploit low-rank structures perpendicular to the z-axis. LiDAR point clouds contain one dominant low-rank structure perpendicular to this axis, which is: the ground plane. The ground plane, which consists of a significant amount of points, is thus an ideal target for compression of z-value sorted LiDAR point clouds. Apart from the ground plane, not many low-rank structures exist on the xy-plane. This causes the performance of sorting by z-value to be strong with relatively small amounts of compression compared to large amounts of compression.

In order to validate this train of thought, an experiment is performed. Figure 4-23 shows this experiment. The figure shows the performance curve of the TT-SVD algorithm when sorting by z-value (red) as well as three test scenarios (green, blue and grey). The test scenarios also use the TT-SVD algorithm on z-value sorted data, however they contain one extra modification. Prior to sorting the LiDAR data a homogeneous rotation is applied. The LiDAR point cloud is rotated by a 45 degree angle around the x-axis (green), y-axis (blue) and z-axis (grey). The figure shows that a rotation around the x- and y- axis causes a drop in performance in the low compression region (BPP $\in [32, 42]$), and a small increase in performance in the high compression region (BPP $\in [0, 32]$) This change in performance is not observed when applying a rotation around the z-axis, since the red and grey line coincide. The drop in performance in the low compression region (BPP $\in [32, 42]$) can be attributed to the fact that the LiDAR points belonging to the ground plane are not anymore grouped together. Hence, exploiting redundancies of this low-rank structure has become more difficult for the TT-SVD algorithm.

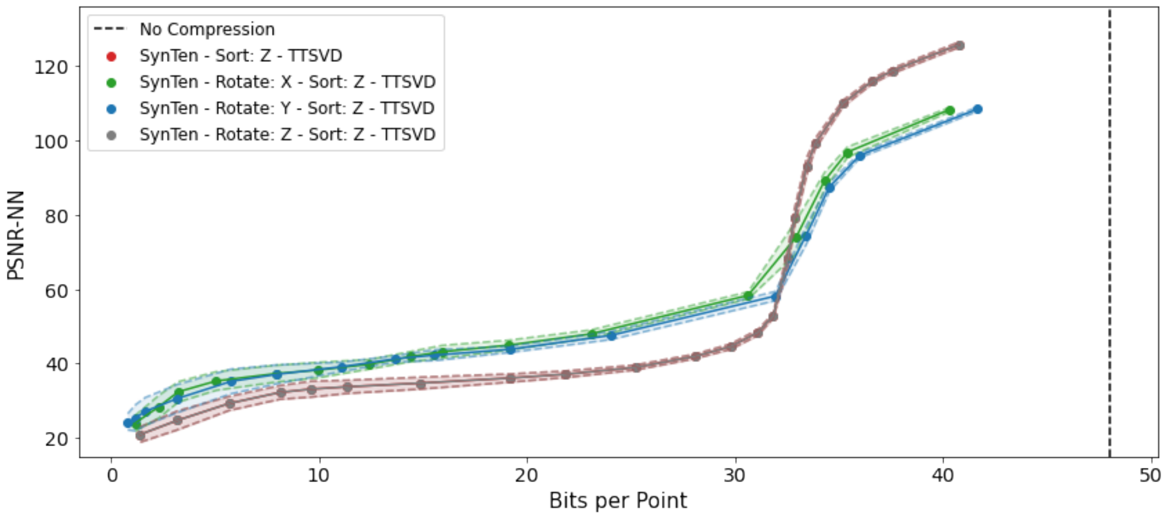


Figure 4-23: Performance curve of best-performing compression methods and baseline model: TMC3. The dashed lines denote the variance of each method in $\mu \pm \sigma$.

A similar argument can be made regarding the performance of sorting by angular difference of consecutive vectors. This sort method finds the nearest neighbour in terms of angular difference between consecutive points. Points with small angular difference will likely originate from the same low-rank structure such as the facade of a house. This means that points belonging to the same low-rank structure will likely be stored contiguously. This grouping of low-rank structures is believed to be the cause for success of this sort method.

Figure 4-22 showed that none of the tensor decomposition methods outperform the baseline model TMC3. In order to see why this occurs, let us view another variant of the performance curve. Figure 4-24 shows the performance of 3 samples decomposed using the TT-SVD algorithm tensorized using parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$ and sorted using angular difference of consecutive vectors. The figure shows that for all 3 samples a very large drop in terms of PSNR-NN occurs, which does not yield a significant contribution in terms of compression gains. This drop in PSNR-NN is caused by the first truncation that occurs when epsilon exceeds a certain value.

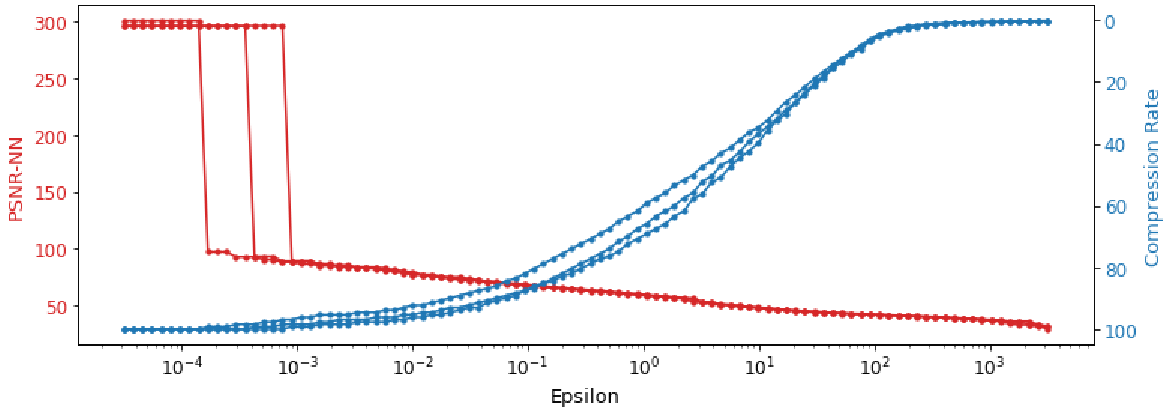


Figure 4-24: PSNR-NN and compression rate for 3 samples decomposed using the TT-SVD algorithm, tensorized using parameters $(I_1, I_2, I_3, I_4) = (40, 40, 40, 3)$, and sorted using angular difference of consecutive vectors. A large drop in PSNR-NN occurs at the first truncation.

4-5 Geometry Aware Tensor Decompositions for Point Cloud Compression

The next approach that will be discussed is geometry aware tensor decompositions. It is a combination of the previous two approaches. The idea is that a LiDAR point cloud is tensorized in such a way that the placement of points within the tensor reflects the real-world location of the points. Within geometry aware tensorization two approaches are considered: Hierarchical and Assignment Problem.

4-5-1 Hierarchical Approach

Figure 4-25 shows the singular values of the mode-n unfoldings for the geometry aware tensorized LiDAR data plotted against random sorted LiDAR and the best sorting method for synthetic tensorization: Angular Difference - Consecutive Vector. The singular values are averaged over the 10 different samples. The figure shows that the geometry aware and consecutive vector approach perform similarly, since the singular value decline is comparable for both methods.

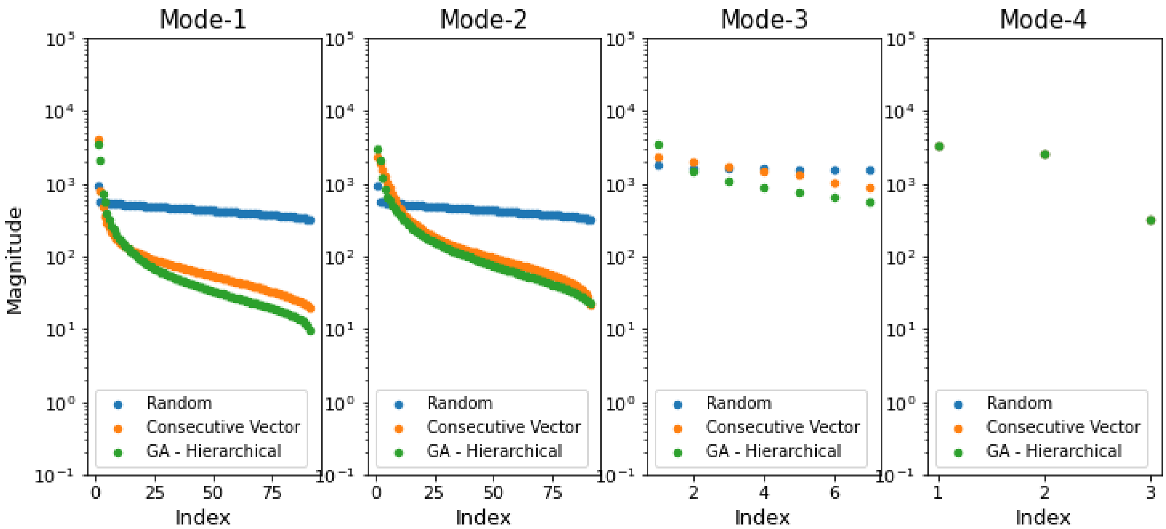


Figure 4-25: Singular values of mode-n unfoldings for several tensorization methods averaged over 10 samples. A small difference is displayed in singular value decline between the Consecutive Vector and Geometry Aware - Hierarchical approach.

Table 4-10 shows the performance of applying the CP-ALS, MLSVD and TT-SVD algorithm onto geometry aware tensorized LiDAR using the hierarchical approach. The best performing method is the TT-SVD, achieving the highest PSNR-NN with minimal computational time.

Method	Ranks / ϵ	PSNR-NN $\uparrow$	BPP $\downarrow$	Compression Rate $\downarrow$	Time (sec)
CPD	764	57.8 ± 1.1	38.4	80.0 %	80 ± 2
Tucker	(81,82,7,3)	53.9 ± 2.3	38.5	80.2 %	1.9 ± 0.1
TT	0.36	$\mathbf{62.5 \pm 0.8}$	38.4 ± 1.6	80.0 ± 3.3 %	1.8 ± 0.1

Table 4-10: Performance of CP-ALS, MLSVD and TT-SVD when using geometry aware tensorization. The information is displayed in the format $\mu \pm \sigma$, where μ is the mean, and σ the standard deviation over 10 different samples.

Figure 4-26 shows the qualitative result of applying the TT-SVD algorithm onto geometry aware tensorized LiDAR. The figure shows a decent reconstruction with a PSNR-NN of: 59.54.

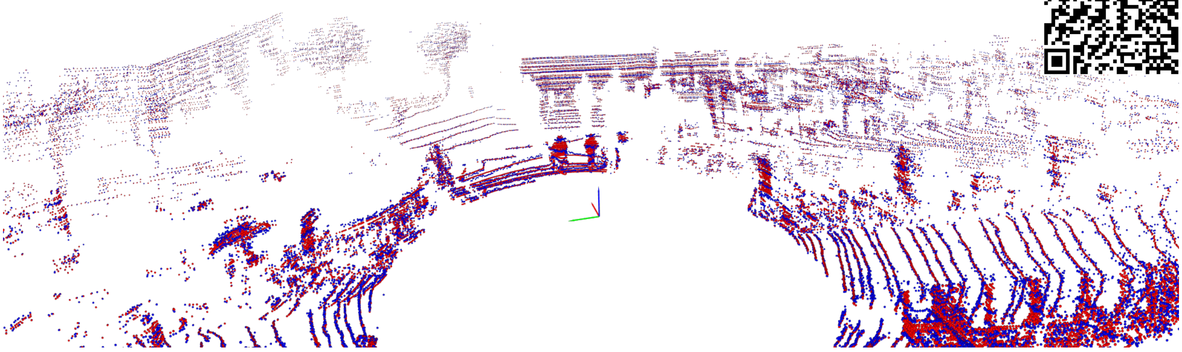


Figure 4-26: Original(red) and TTSVD(blue) of geometry aware tensorized LiDAR using hierarchical approach. PSNR-NN: 59.54. BPP: 36.05. Compression rate: 75.10%. Scan QR Code or click on link for 3D Render.

4-5-2 Assignment Problem Approach

The hierarchical approach for geometry aware tensorization is a relatively easy-to-implement and fast approach for tensorizing the LiDAR data. It does however not give any guarentees regarding the optimality of assigning points to tensor indices. This is where the assignment problem approach steps in. The assignment problem formulation finds the optimal allocation of points with respect to the Frobenius loss of the placement error.

Unfortunately, this optimality comes at a price. The computational complexity increases drastically and the memory requirements exceed 16 GB, which makes the problem unsolvable on many system architectures like the one employed during this thesis. In order to present some analysis regarding this method, the size of the LiDAR point cloud would have to be reduced considerably ($\sim 80\%$ reduction) in order to make the problem tractable. A reduction of this magnitude would make obtained results incomparable with the other proposed methods. Hence, the assignment problem approach is not investigated further.

Chapter 5

Conclusions

This thesis has presented a proof-of-concept for an alternative approach to Point Cloud Compression of automotive LiDAR data. Concluding this thesis can be done by revisiting the research question posed in the introduction:

- Are **tensor decomposition** methods a competitive alternative for **Point Cloud Compression** of automotive LiDAR data.

In order to answer this research question, the following three novel PCC codecs were designed, tested, and evaluated:

1. Voxel-based Tensor Decomposition
2. Synthetic Tensor Decomposition
3. Geometry Aware Tensor Decomposition

All of these three codecs were tested for three tensor network topologies using their most prevalent algorithms: the CP-ALS, the MLSVD, and the TT-SVD. Evaluating the performance of all codecs was done by comparing their obtained PSNR-NN, BPP and time-complexity with the current most widely-used baseline for LiDAR PCC, which is TMC3.

Figure 5-1 shows the PSNR-NN plotted against the BPP for the best-performing implementations of the 3 novel PCC codec's. The figure shows that the baseline model (TMC3) outperforms all proposed methods across the entire range of BPP values. With regard to time complexity, TMC3 also outperforms all of the proposed methods, due to the minimal time needed to find the encoded representation. Answering the research question can thus be done by stating that tensor decomposition methods are not a competitive alternative for point cloud compression of automotive LiDAR data.

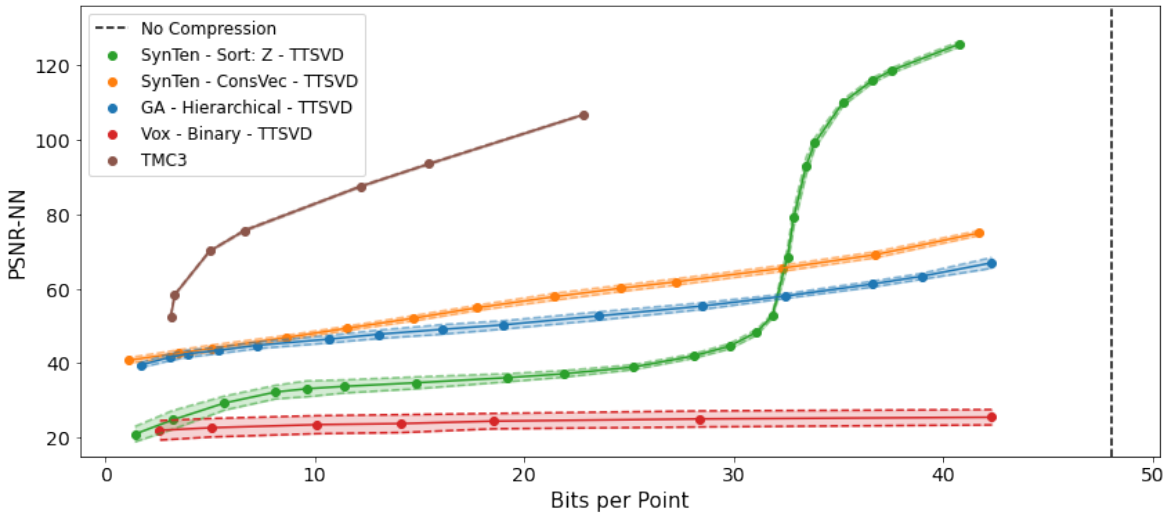


Figure 5-1: Performance curve of best-performing compression methods and baseline model: TMC3. The dashed lines denote the variance of each method in $\mu \pm \sigma$.

Tensor decomposition methods thrive upon data representations, which contain an inherent low-rank structure. Thus, a key part of this thesis was to investigate what LiDAR data representations contain this low-rank structure. Three representations were considered: Voxelized, Synthetically Tensorized and Geometry Aware Tensorized.

The voxel-based representation showed difficulty with finding fitting low-rank decompositions. This was on the one hand attributed to rotational variance of objects in the scene, but also due to a highly voluminous tensor resulting from the voxelization process. This highly voluminous tensor caused the computational complexity of tensor decomposition methods to increase drastically. Additionally, the voxelization process caused a discretization loss to occur, prior to acquiring any tensor decomposition.

Synthetically tensorized LiDAR initially showed little promise, since tensor decomposition methods were applied onto unstructured LiDAR data. Since exploring all possible permutations of LiDAR points in the cloud was intractable, heuristics were designed. These heuristics were used to sort the LiDAR data based on a specific value. Two heuristics outperformed the others. Sorting by angular difference of consecutive vectors was effective for large amount of compression, while sorting by z-value scored best for small amounts of compression. The set of synthetic tensorization parameters, which define the amount and sizes of each mode did not show much impact on compression performance compared to the employed heuristics. A key takeaway is thus the importance of applying a heuristic to structure the LiDAR point cloud, making it more suitable for tensor decomposition methods.

Regarding geometry aware tensorization two methods were proposed: Hierarchical and Assignment Problem. The first method resulted in a good compression performance with a very short computational time compared to other tensor decomposition-based methods. Compared to the baseline model, its performance fell short. Unfortunately, the second method could not be tested due to hardware limitations. The problem of placing points into the tensor was recast as an assignment problem. Solving the assignment problem was however intractable due to the large square cost matrix of size N, which resulted into memory demands by the modified Jonker-Volgenant algorithm which exceeded 16GB [15].

The performance of the presented tensor decomposition methods for PCC fell short compared to the baseline model: TMC3. There are however a number of considerations that can put this result into perspective.

First of all, a big limitation on the side of tensor decomposition methods is that it does not employ bitwise compression, which occurs in the baseline method TMC3. TMC3 uses this bitwise compression in the final stages of the compression pipeline: the arithmetic encoder. The compression performance of tensor decomposition methods could thus possibly be improved by employing bitwise compression on the factor matrices and/or core tensors.

Second of all, a choice was made regarding the precision of the to be compressed elements. For this thesis, the half-precision floating-point format (16 bit) was chosen. Setting this precision presents a trade-off, since compression can either be achieved by means of reducing the precision in bits, or by truncating elements using tensor decomposition methods. Possibly, the performance of the proposed tensor decomposition-based codecs can thus be improved by reducing the chosen precision in bits.

The motivation of this thesis was to reduce the loading and processing bottleneck during training of machine learning models for automotive self-driving applications. This bottleneck could potentially be reduced by shifting workload from the strained CPU to the unsaturated GPU. Tensor decomposition methods which are based on multilinear products could potentially remove this bottleneck by performing fast reconstruction of compressed point clouds on the GPU. Unfortunately, the compression performance of tensor decomposition methods fell short compared to the baseline model TMC3. This means that in order to remove the training bottleneck there are two high-level possibilities. On the one hand, existing well-performing codecs such as TMC3 could potentially be augmented or updated to allow for fast reconstruction of compressed point clouds on the GPU. On the other hand, future work could look into improving the novel tensor decomposition-based PCC codecs presented in this thesis. The next section will elaborate on several research opportunities worth exploring.

5-1 Future Work

This thesis has presented the first work on point cloud compression of automotive LiDAR using tensor decomposition methods. Because of this, many avenues of research are still open to discover. A few of these research oppertunities are listed below.

Tensor Network Topology This thesis has investigated decomposing automotive LiDAR data into three prevalent tensor network topologies: the CPD, Tucker and TT. There are however much more topologies such as the tensor ring [89], tree tensor network [12], but also many more which are not named. Future work could thus take a much broader look into the different types of topologies, and how they perform regarding LiDAR PCC. One method to find the best tensor network topology is to employ genetic algorithms [46],[47]. These algorithms create a population of different topologies, and breed new topologies by mating of successful individuals. The idea is that the offspring of the successful topologies will likely have the same traits of its succesfull parents. This process is repeated until an ideal topology is found.

Synthetic Tensorization - Enumeration of Points This thesis has shown that an important step prior to synthetic tensorization is to introduce structure into the LiDAR data. The enumeration of points in the cloud has a big impact on the performance of tensor decomposition methods. This thesis employed various heuristics in order to sort the LiDAR. Future work could look into various ways of finding the best ordering of LiDAR points in the cloud. This could for example be done by developing new heuristics. Alternatively, an optimization-based approach might be possible, where the ordering of points in the cloud is updated iteratively with respect to some loss function.

Compression across Time This thesis only considered compressing individual LiDAR samples. For some applications however, it might be interesting to compress a group of consecutive LiDAR samples. Consecutive LiDAR samples will most likely contain redundant information since the interval between scans is often around 0.1 seconds. This redundancy of information in consecutive samples could possibly be exploited using tensor decomposition methods.

Appendix A

Code

A-1 Canonical Polyadic Decomposition (CPD)

A-1-1 Canonical Polyadic - Alternating Least Squares (CP-ALS)

```
def CPD(T,R,init,maxIter,relativeErrorThreshold):
    """
    Inputs:
    T: Tensor
    R: Rank
    init: Initialization method
    """

    # Initialize Factor Matrices
    if init == "Random":
        mu = 0
        sigma = 1
        FactorMatrices = []
        for dim in range(T.ndim):
            FactorMatrices.append(np.random.normal(mu, sigma, (T.shape[
                dim],R)))

    # Normalize the columns of the Factor Matrices
    for idx, FactorMatrix in enumerate(FactorMatrices):
        FactorMatrices[idx] = FactorMatrix / np.linalg.norm(FactorMatrix,
            axis=0)

    # Obtain the tensor unfoldings:
    Tn = []
    for idxDim in range(T.ndim):
         Tn.append(mode_n_matricization(T,idxDim+1))

    # Create empty list for storing relative Frobenius and PSNR-NN error
```

```
    relativeErrorList = []
    PSNRNNErrorList = []

    # Reconstruct original LiDAR in tabular form
    T_og_tabular = detensorize_no_frame_data(T, dim_list = T.shape)

    # Loop through amount of ALS iterations:
    for idxIter in range(maxIter):
        # Loop over the dimensions of the tensor:
        for idxDim in range(T.ndim):

            # Create a second loop for iterating over the dimensions:
            secondLoop = list(range(T.ndim))
            secondLoop.remove(idxDim)

            # Initiialize intermediary values: V and KR
            V = np.ones((FactorMatrices[0].T @ FactorMatrices[0]).shape)
            KR = np.ones((1,FactorMatrices[secondLoop[0]].shape[1]))
            for secondIdx in secondLoop:
                V = V*(FactorMatrices[secondIdx].T @ FactorMatrices[
                    secondIdx])
                KR = khatri_rao(FactorMatrices[secondIdx], KR)

            # Update Factor Matrix, obtain norm, and normalize
            FactorMatrices[idxDim] = Tn[idxDim] @ KR @ np.linalg.pinv(V)
            c = np.linalg.norm(FactorMatrices[idxDim], axis=0)
            FactorMatrices[idxDim] = FactorMatrices[idxDim] / c

        # Compute the current estimate of the mode-N unfolding of the
            tensor
        KR_end = np.ones((1,FactorMatrices[0].shape[1]))
        for k in range(T.ndim-1):
            KR_end = khatri_rao(FactorMatrices[k],KR_end)
        TN_est = c*FactorMatrices[T.ndim-1] @ KR_end.T

        # Calculate Relative Error (Frobenius) between the estimate and
            the true mode-N unfolding of the tensor
        TN_error = Tn[T.ndim-1] - TN_est
        norm_error = frob_norm(TN_error) / frob_norm(Tn[T.ndim-1])
        relativeErrorList.append(norm_error)

        ### For calculating PSRN During each iteration, Detensorize
        # Calculate PSNR of NN error
        T_rec = reconstruct_CPD(FactorMatrices = FactorMatrices,
                                norm_vector = c)
        T_rec_tabular = detensorize_no_frame_data(T_rec, dim_list = T.
            shape)
        PSNR = get_PSNR_NN_VoD(points_og = T_og_tabular,
                               points_rec = T_rec_tabular,
                               output = False)
        PSNRNNErrorList.append(PSNR)

        # Check if the stopping criterion has been reached
```

```
        if relativeErrorList[-1] < relativeErrorThreshold:
            break

    # Calculate PSNR of NN error
    T_rec = reconstruct_CPD(FactorMatrices = FactorMatrices,
                            norm_vector = c)
    T_rec_tabular = detensorize_no_frame_data(T_rec, dim_list = T.shape)
    PSNR = get_PSNR_NN_VoD(points_og = T_og_tabular,
                           points_rec = T_rec_tabular,
                           output = False)
    PSNRNNErrorList.append(PSNR)
    return FactorMatrices, c, relativeErrorList, PSNRNNErrorList
```

A-1-2 Reconstruct CPD

```
def reconstruct_CPD(FactorMatrices, norm_vector):
    N = len(FactorMatrices) # Amount of dimensions
    v = len(norm_vector) # Size of each dimension

    RankvTensor = 0
    for r, Lambda in enumerate(norm_vector):
        Rank1Tensor = Lambda
        for Matrix in FactorMatrices:
            Rank1Tensor = np.tensordot(Rank1Tensor, Matrix[:,r], axes=0)
        RankvTensor = RankvTensor + Rank1Tensor
    T_rec = RankvTensor
```

A-2 Multilinear Singular Value Decomposition (MLSVD)

A-2-1 Mode-n Matricization

```
def mode_n_matricization(X,n):
    firstdims = np.arange(0, n-1, 1)
    lastdims = np.arange(n, X.ndim, 1)
    dim_change = np.concatenate(([n-1],firstdims,lastdims))
    X = X.transpose(dim_change)
    X = X.reshape((X.shape[0], -1), order='F')
    return X
```

A-2-2 Mode-n Product

```
def mode_n_product(X,Y,n):
    # MODE_N_PRODUCT takes tensor X and compatible matrix Y and performs
      mode-n product between X and Y.
    # INPUT tensor X, matrix Y.
    # OUTPUT tensor Z.
    # X:= I_1 ... I_n-1 I_n I_n+1 ... I_N
```

```
    # Y:= J x I_n

    # Perform mode-n matricization
    X_matricized = mode_n_matricization(X,n)
    # resulting shape: I_n x I_1 ... I_n-1 x I_n+1 ... I_N

    # Multiply
    Z = Y@X_matricized
    # resulting shape: J x I_1 ... I_n-1 I_n+1 ... I_N

    # Collect dimensions
    dim_J = Y.shape[0]
    dim_I = X.shape
    N = X.ndim

    # Reshape
    dim_change = np.concatenate(([dim_J], dim_I[0:n-1], dim_I[n:N] )).
        astype(int)
    Z = np.reshape(Z, dim_change, order='F')
    # resulting shape: J x I_1 x ... x I_n-1 x I_n+1 x ... x I_N

    # Permute
    firstdims = np.arange(1, n, 1)
    lastdims = np.arange(n, N, 1)
    dim_change = np.concatenate((firstdims,[0],lastdims))
    Z = np.transpose(Z, dim_change)
    # resulting shape: I_1 x ... x I_n-1 x J x I_n+1 x ... x I_N

    return Z
```

A-2-3 MLSVD

```
def MLSVD_ND(T,output):
    from scipy import linalg
    factor_matrices = []
    core = T # Set Core tensor to T
    for i in range(T.ndim):
        Ti = mode_n_matricization(T,i+1)
        Ui, *_ = linalg.svd(Ti, full_matrices=False)
        factor_matrices.append(Ui)
        core = mode_n_product(core,Ui.transpose(),i+1)
        if output:
            print(f"Computed Mode: {i+1}")
    return core, factor_matrices
```

A-2-4 Truncate MLSVD

```python
def truncate_MLSVD_ND(core,factor_matrices,ranks):
    # Create slice list for truncating core
    slice_list = []
    # Create counter
    i = 0
    for matrix in factor_matrices:
        factor_matrices[i]  = matrix[:,:ranks[i]]
        # Append trucation for mode "i" to slice list
        slice_list.append(slice(0,ranks[i]))
        # Increment count
        i += 1
    # Truncate core tensor
    core = core[tuple(slice_list)]
    return core, factor_matrices
```

A-2-5 Reconstruct MLSVD

```python
def rec_MLSVD_ND(core, factor_matrices):
    # Create list which holds tuples for reconstructing
    rec_list = []
    for idx, matrix in enumerate(factor_matrices):
        # Pad factor matrices
        factor_matrices[idx] = np.pad(matrix, ((0,0),(0,matrix.shape[0]-matrix.shape[1])), 'constant', constant_values=0 )
        # Append to rec list
        rec_list.append((0,matrix.shape[0]-matrix.shape[1]))
    # Pad core
    core = np.pad(core, tuple(rec_list), 'constant', constant_values=0)
    # loop through factor matrices
    for idx, matrix in enumerate(factor_matrices):
        # Compute mode-n product
        core = mode_n_product(core,matrix,idx+1)
    # Rename variable core for clarity
    T_rec = core
    return T_rec
```

A-2-6 Plotting Singular Values of Mode-n Unfoldings

```python
def plot_singular_values(T):
    import matplotlib.pyplot as plt
    import numpy as np
    fig, axs = plt.subplots(1,T.ndim, figsize=(T.ndim*3,5))
    fig.suptitle('Singular Values of Mode-n Unfoldings')
    axs[0].set_ylabel('Magnitude')
    for i in range(0, T.ndim):
        Si = np.linalg.svd(mode_n_matricization(T,i+1),full_matrices=False, compute_uv=False)
        print(f"Computed mode-{i+1} matricization")
        print(f"Computed SVD: {i+1}")
        axs[i].set_yscale('log')
```

```
        axs[i].scatter(np.arange(1,len(Si)+1,1), Si)
        axs[i].set_xlabel('Index')
        axs[i].set_title(f'Mode-{1+i}')
        axs[i].set_ylim(0.1, 10**5)
    plt.show()
```

A-3 Tensor Train Singular Value Decomposition (TT-SVD)

A-3-1 TT-SVD

```
def TT_SVD(tensor, epsilon):
    # Create empty list which will contain the TT-cores of the decomposition
    tt_cores = []
    error = []
    # Calculate truncation parameter
    d = tensor.ndim
    n = tensor.shape
    delta = (epsilon / np.sqrt(d-1)) * frob_norm(tensor)
    # Set temporary tensor
    C = tensor
    r = np.ones(d).astype(int)
    for k in range(d-1):
        # Reshape tensor
        C = np.reshape(C, (r[k]*n[k], -1), order='F')
        # Compute SVD
        U,S,V = np.linalg.svd(C, full_matrices=False)
        # Find Truncation index, and trucation error
        trunc_errors = np.cumsum(np.flip(S)**2)
        bound_satisfied = np.where(trunc_errors < delta)[0]
        if bound_satisfied.size == 0: # Check if we can not truncate
            bound_satisfied = [0]
            error.append(0) # Error is 0 if we do not truncate
        else:
            # Append largest error where trunc error bound is still satisfied
            error.append(trunc_errors[bound_satisfied[-1]])
        r[k+1] = len(S) - bound_satisfied[-1]
        # Compute delta-truncated SVD
        Ut = U[:,:r[k+1]]
        St = np.diag(S[:r[k+1]])
        Vt = V[:r[k+1],:]
        # Store newly obtained core (U)
        tt_cores.append(np.reshape(Ut, (r[k],n[k],r[k+1]), order='F') )
        # Keep right side of SVD
        C = St @ Vt
    # Append last (norm) core to TT cores list
    tt_cores.append(C)
    rel_error = np.sqrt(np.sum(error)) / frob_norm(tensor)

    return tt_cores, rel_error
```

A-3-2 Reconstruct TT-SVD

```
def TT_reconstruct(tt):
    tensor = np.ones((1,1))
    og_dims = np.empty(len(tt)).astype(int)
    ranks = np.empty(len(tt)).astype(int)
    for idx, core in enumerate(tt):
        ranks[idx] = core.shape[-1]
        tensor = tensor @ mode_n_matricization(core,1)
        tensor = np.reshape(tensor, (-1, ranks[idx]), order='F')
        og_dims[idx] = core.shape[1]
    # Reshape tensor back to original size
    tensor = np.reshape(tensor, tuple(og_dims), order='F')
    return tensor
```

Bibliography

[1] Rashid Abbasi, Ali Bashir, Hasan Alyamani, Farhan Amin, Jaehyeok Doh, and Jianwen Chen. Lidar point cloud compression, processing and learning for autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, PP:1–18, 01 2022.

[2] Khartik Ainala, Rufael N Mekuria, Birendra Khathariya, Zhu Li, Ye-Kui Wang, and Rajan Joshi. An improved enhancement layer for octree based point cloud compression with plane projection approximation. In *Applications of Digital Image Processing XXXIX*, volume 9971, pages 223–231. SPIE, 2016.

[3] Rasmus Bro. Multiway analysis in the food industry. models, algorithms and applications. *Ph.D. dissertation, University of Amsterdam, Amsterdam*, 08 2001.

[4] Elena Camuffo, Daniele Mari, and Simone Milani. Recent advancements in learning algorithms for point clouds: An updated overview. *Sensors*, 22(4), 2022.

[5] Chao Cao, Marius Preda, and Titus Zaharia. 3d point cloud compression: A survey. In *Proceedings of the 24th International Conference on 3D Web Technology*, Web3D '19, page 1–9, New York, NY, USA, 2019. Association for Computing Machinery.

[6] Chao Cao, Marius Preda, Vladyslav Zakharchenko, Euee S. Jang, and Titus Zaharia. Compression of sparse and dense dynamic point clouds—methods and standards. *Proceedings of the IEEE*, 109(9):1537–1558, 2021.

[7] Van-Hung Cao, KX Chu, Nhien-An Le-Khac, M Tahar Kechadi, Debra Laefer, and Linh Truong-Hong. Toward a new approach for massive lidar data processing. In *2015 2nd IEEE International Conference on Spatial Data Mining and Geographical Knowledge Services (ICSDM)*, pages 135–140. IEEE, 2015.

[8] Raymond B. Cattell. The scree test for the number of factors. *Multivariate Behavioral Research*, 1(2):245–276, 1966. PMID: 26828106.

[9] Dirk G Cattrysse and Luk N Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European journal of operational research*, 60(3):260–272, 1992.

[10] Milieu Centraal. Praktisch over duurzaam. https://www.milieucentraal.nl/klimaat-en-aarde/klimaatverandering/wat-is-je-co2-voetafdruk/. Accessed: 09-01-2024.

[11] Yukang Chen, Jianhui Liu, Xiangyu Zhang, Xiaojuan Qi, and Jiaya Jia. Voxelnext: Fully sparse voxelnet for 3d object detection and tracking. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21674–21683, June 2023.

[12] Song Cheng, Lei Wang, Tao Xiang, and Pan Zhang. Tree tensor networks for generative modeling. *Physical Review B*, 99(15):155131, 2019.

[13] Won-Seok Choi, Yang-Shin Kim, Se-Young Oh, and Jeihun Lee. Fast iterative closest point framework for 3d lidar data in intelligent vehicle. In *2012 IEEE Intelligent Vehicles Symposium*, pages 1029–1034. IEEE, 2012.

[14] Andrzej Cichocki, Namgil Lee, Ivan V. Oseledets, Anh Huy Phan, Qibin Zhao, and Danilo P. Mandic. Low-rank tensor networks for dimensionality reduction and large-scale optimization problems: Perspectives and challenges PART 1. *CoRR*, abs/1609.00893, 2016.

[15] David F. Crouse. On implementing 2d rectangular assignment algorithms. *IEEE Transactions on Aerospace and Electronic Systems*, 52(4):1679–1696, 2016.

[16] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. On the best rank-1 and rank-(r1,r2,...,rn) approximation of higher-order tensors. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1324–1342, 2000.

[17] NVIDIA Developer. Nvidia data loading library (dali). https://developer.nvidia.com/dali. Accessed: 03-01-2024.

[18] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[19] Ahmad Faiz, Sotaro Kaneda, Ruhan Wang, Rita Osi, Parteek Sharma, Fan Chen, and Lei Jiang. Llmcarbon: Modeling the end-to-end carbon footprint of large language models, 2023.

[20] Lue Fan, Feng Wang, Naiyan Wang, and Zhaoxiang Zhang. Fully sparse 3d object detection, 2022.

[21] Lue Fan, Xuan Xiong, Feng Wang, Naiyan Wang, and Zhaoxiang Zhang. Rangedet: In defense of range view for lidar-based 3d object detection. *CoRR*, abs/2103.10039, 2021.

[22] Chunyang Fu, Ge Li, Rui Song, Wei Gao, and Shan Liu. Octattention: Octree-based large-scale contexts model for point cloud compression. *Proceedings of the AAAI Conference on Artificial Intelligence*, 36(1):625–633, Jun. 2022.

[23] Pierre-Marie Gandoin and Olivier Devillers. Progressive lossless compression of arbitrary simplicial complexes. *ACM Trans. Graph.*, 21(3):372–379, jul 2002.

[24] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012.

[25] D. Graziosi, O. Nakagami, S. Kuma, A. Zaghetto, T. Suzuki, and A. Tabatabai. An overview of ongoing point cloud compression standardization activities: video-based (v-pcc) and geometry-based (g-pcc). *APSIPA Transactions on Signal and Information Processing*, 9:e13, 2020.

[26] MPEG 3D Graphics Coding group (3DG). Mpeg point cloud compression. https://mpeg-pcc.org/. Accessed: 15-01-2024.

[27] André FR Guarda, Nuno MM Rodrigues, and Fernando Pereira. Point cloud coding: Adopting a deep learning-based approach. In *2019 Picture Coding Symposium (PCS)*, pages 1–5. IEEE, 2019.

[28] André FR Guarda, Nuno MM Rodrigues, and Fernando Pereira. Deep learning-based point cloud geometry coding: Rd control through implicit and explicit quantization. In *2020 IEEE International Conference on Multimedia & Expo Workshops (ICMEW)*, pages 1–6. IEEE, 2020.

[29] Richard A. Harshman and Margaret E. Lundy. Parafac: Parallel factor analysis. *Computational Statistics & Data Analysis*, 18(1):39–72, 1994.

[30] Andrew J. Hawkins. Cruise is now charging for rides in its driverless vehicles in san francisco. https://www.theverge.com/2022/6/23/23180156/cruise-driverless-vehicle-charge-riders-san-francisco, Jun 2022. Accessed: 03-01-2024.

[31] Andrew J. Hawkins. https://www.theverge.com/2023/12/20/24006712/waymo-driverless-million-mile-safety-compare-human, Dec 2023. Accessed: 03-01-2024.

[32] Reetu Hooda, W. David Pan, and Tamseel M. Syed. A survey on 3d point cloud compression using machine learning approaches. In *SoutheastCon 2022*, pages 522–529, 2022.

[33] David G. Hough. Ieee standard for floating-point arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[34] Lila Huang, Shenlong Wang, Kelvin Wong, Jerry Liu, and Raquel Urtasun. Octsqueeze: Octree-structured entropy model for lidar compression. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[35] Johan HÃ¥stad. Tensor rank is np-complete. *Journal of Algorithms*, 11(4):644–654, 1990.

[36] Alexander Isenko, Ruben Mayer, Jeffrey Jedele, and Hans-Arno Jacobsen. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD/PODS '22. ACM, June 2022.

[37] Roy Jonker and Ton Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. In *DGOR/NSOR: Papers of the 16th Annual Meeting of DGOR in Cooperation with NSOR/Vorträge der 16. Jahrestagung der DGOR zusammen mit der NSOR*, pages 622–622. Springer, 1988.

[38] Im Jeong Joon, Alexander Leonessa, Andrew Kurdila, and Young-Jae Ryoo. A real-time data compression for ground-based 3d lidar data using wavelets and compressive sensing. *SCIS & ISIS*, 2010(0):772–777, 2010.

[39] Aarati Kakaraparthy, Abhay Venkatesh, Amar Phanishayee, and Shivaram Venkataraman. The case for unifying data loading in machine learning clusters. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[40] Tamara G. Kolda and Brett W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009.

[41] Tamara G Kolda and Jimeng Sun. Scalable tensor decompositions for multi-aspect data mining. In *2008 Eighth IEEE international conference on data mining*, pages 363–372. IEEE, 2008.

[42] Michael Kuchnik, Ana Klimovic, Jiri Simsa, Virginia Smith, and George Amvrosiadis. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 33–51, 2022.

[43] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[44] Alex H. Lang, Sourabh Vora, Holger Caesar, Lubing Zhou, Jiong Yang, and Oscar Beijbom. Pointpillars: Fast encoders for object detection from point clouds, 2019.

[45] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Mądry. Ffcv: Accelerating training by removing data bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12011–12020, June 2023.

[46] Chao Li and Zhun Sun. Evolutionary topology search for tensor network decomposition. In *International Conference on Machine Learning*, pages 5947–5957. PMLR, 2020.

[47] Chao Li, Junhua Zeng, Zerui Tao, and Qibin Zhao. Permutation search of tensor network structures via local sampling. In *International Conference on Machine Learning*, pages 13106–13124. PMLR, 2022.

[48] Feng Li, Zhiwei Yu, Bo Wang, and Qianlin Dong. Filtering algorithm for lidar outliers based on histogram and kd tree. In *2011 4th International Congress on Image and Signal Processing*, volume 5, pages 2741–2745, 2011.

[49] Lingjie Li, Wenjian Yu, and Kim Batselier. Faster tensor train decomposition for sparse data. *Journal of Computational and Applied Mathematics*, 405:113972, 2022.

[50] Shihua Li, Jingxian Wang, Zuqin Liang, and Lian Su. Tree point clouds registration using an improved icp algorithm based on kd-tree. In *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, pages 4545–4548, 2016.

[51] Velodynme LiDAR. Velodyne's hdl-64e lidar sensor looks back on a legendary career. https://velodynelidar.com/blog/hdl-64e-lidar-sensor-retires/. Accessed: 09-01-2024.

[52] Jyh-Ming Lien, Gregorij Kurillo, and Ruzena Bajcsy. Multi-camera tele-immersion system with real-time model driven data compression: A new model-based compression method for massive dynamic point data. *The Visual Computer*, 26:3–15, 2010.

[53] Hao Liu, Hui Yuan, Qi Liu, Junhui Hou, and Ju Liu. A comprehensive study and comparison of core technologies for mpeg 3-d point cloud compression. *IEEE Transactions on Broadcasting*, 66(3):701–717, 2020.

[54] Hua Liu, Zhengdong Huang, Qingminga Zhan, and Penga Lin. A database approach to very large lidar data management. *The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Beijing, China*, 37(B1):463–468, 2008.

[55] Tao Lu, Xiang Ding, Haisong Liu, Gangshan Wu, and Limin Wang. Link: Linear kernel for lidar-based 3d perception. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1105–1115, 2023.

[56] Rachel Minster, Irina Viviano, Xiaotian Liu, and Grey Ballard. Cp decomposition for tensors via alternating least squares with qr decomposition. *Numerical Linear Algebra with Applications*, 30(6):e2511, 2023.

[57] L. Mirsky. Symmetric gauge functions and unitarily invariant norms. *The Quarterly Journal of Mathematics*, 11(1):50–59, 1960.

[58] Jayashree Mohan, Amar Phanishayee, Ashish Raniwala, and Vijay Chidambaram. Analyzing and mitigating data stalls in DNN training. *CoRR*, abs/2007.06775, 2020.

[59] MPEGGroup. GitHub - MPEGGroup/mpeg-pcc-tmc13: Geometry based point cloud compression (G-PCC) test model.

[60] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *CoRR*, abs/2101.12127, 2021.

[61] Georgii Sergeevich Novikov and Ivan Oseledets. Tensor-train point cloud compression and efficient approximate nearest neighbor search. 2023.

[62] Nuscenes.org. nuscenes detection task - lidar only. https://www.nuscenes.org/object-detection?externalData=no&mapData=no&modalities=Lidar. Accessed: 20-02-2024.

[63] Motional Operations. Motional expands autonomous testing to san diego. https://motional.com/news/motional-expands-autonomous-testing-san-diego, Jul 2022. Accessed: 03-01-2024.

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

[65] Andras Palffy, Ewoud Pool, Srimannarayana Baratam, Julian F. P. Kooij, and Dariu M. Gavrila. Multi-class road user detection with 3+1d radar in the view-of-delft dataset. *IEEE Robotics and Automation Letters*, 7(2):4961–4968, 2022.

[66] David A. Patterson, Joseph Gonzalez, Quoc V. Le, Chen Liang, Lluis-Miquel Munguia, Daniel Rothchild, David R. So, Maud Texier, and Jeff Dean. Carbon emissions and large neural network training. *CoRR*, abs/2104.10350, 2021.

[67] Eric T Phipps and Tamara G Kolda. Software for sparse tensor decomposition on emerging computing architectures. *SIAM Journal on Scientific Computing*, 41(3):C269–C290, 2019.

[68] Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017.

[69] Maurice Quach, Jiahao Pang, Dong Tian, Giuseppe Valenzise, and Frederic Dufaux. Survey on deep learning-based point cloud compression. *Frontiers in Signal Processing*, 2, 2022.

[70] Maurice Quach, Giuseppe Valenzise, and Frederic Dufaux. Learning convolutional transforms for lossy point cloud geometry compression. In *2019 IEEE international conference on image processing (ICIP)*, pages 4320–4324. IEEE, 2019.

[71] Sirisha Rambhatla, Nikos D. Sidiropoulos, and Jarvis Haupt. Tensormap: Lidar-based topological mapping and localization via tensor decompositions. In *2018 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. IEEE, November 2018.

[72] Sebastian Schwarz, Marius Preda, Vittorio Baroncini, Madhukar Budagavi, Pablo Cesar, Philip A. Chou, Robert A. Cohen, Maja Krivokuća, Sébastien Lasserre, Zhu Li, Joan Llach, Khaled Mammou, Rufael Mekuria, Ohji Nakagami, Ernestasia Siahaan, Ali Tabatabai, Alexis M. Tourapis, and Vladyslav Zakharchenko. Emerging mpeg standards for point cloud compression. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 9(1):133–148, 2019.

[73] Jing Shen, Jiping Liu, Rong Zhao, and Xiangguo Lin. A kd-tree-based outlier detection method for airborne lidar point clouds. In *2011 International Symposium on Image and Data Fusion*, pages 1–4, 2011.

[74] Guangsheng Shi, Ruifeng Li, and Chao Ma. Pillarnet: Real-time and high-performance pillar-based 3d object detection, 2022.

[75] 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.

[76] G. W. Stewart. On the early history of the singular value decomposition. *SIAM Review*, 35(4):551–566, 1993.

[77] Yun-Ting Su, James Bethel, and Shuowen Hu. Octree-based segmentation for terrestrial lidar point cloud data in industrial applications. *ISPRS Journal of Photogrammetry and Remote Sensing*, 113:59–74, 2016.

[78] Will Wei Sun, Junwei Lu, Han Liu, and Guang Cheng. Provable sparse tensor decomposition. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 79(3):899–916, 2017.

[79] Yi-Hsing Tseng and Miao Wang. Automatic plane extraction from lidar data based on octree splitting and merging segmentation. In *Proceedings. 2005 IEEE International Geoscience and Remote Sensing Symposium, 2005. IGARSS '05.*, volume 5, pages 3281–3284, 2005.

[80] Chenxi Tu, Eijiro Takeuchi, Alexander Carballo, and Kazuya Takeda. Point cloud compression for 3d lidar sensor using recurrent neural network with residual blocks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3274–3280. IEEE, 2019.

[81] 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.

[82] Michel Verhaegen and Vincent Verdult. *Filtering and System Identification: A Least Squares Approach*. Cambridge University Press, 2007.

[83] Miao Wang and Yi-Hsing Tseng. Lidar data segmentation and classification based on octree structure. *parameters*, 1(5), 2004.

[84] Hakan Wiman and Yuchu Qin. Fast compression and access of lidar point clouds using wavelets. In *2009 Joint Urban Remote Sensing Event*, pages 1–6, 2009.

[85] Yan Yan, Yuxing Mao, and Bo Li. Second: Sparsely embedded convolutional detection. *Sensors*, 18(10), 2018.

[86] Chih-Chieh Yang and Guojing Cong. Accelerating data loading in deep neural network training. *CoRR*, abs/1910.01196, 2019.

[87] Tianwei Yin, Xingyi Zhou, and Philipp Krähenbühl. Center-based 3d object detection and tracking. *CoRR*, abs/2006.11275, 2020.

[88] Zhen Zhang, Chaokun Chang, Haibin Lin, Yida Wang, Raman Arora, and Xin Jin. Is network the bottleneck of distributed training? In *Proceedings of the Workshop on Network Meets AI & ML*, NetAI '20, page 8–13, New York, NY, USA, 2020. Association for Computing Machinery.

[89] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.

[90] Yin Zhou and Oncel Tuzel. Voxelnet: End-to-end learning for point cloud based 3d object detection. *CoRR*, abs/1711.06396, 2017.

[91] Walter Zimmer, Ramandika Pranamulia, Xingcheng Zhou, Mingyu Liu, and Alois C Knoll. Pointcompress3d–a point cloud compression framework for roadside lidars in intelligent transportation systems. *arXiv preprint arXiv:2405.01750*, 2024.

[92] Mahdi Zolnouri, Xinlin Li, and Vahid Partovi Nia. Importance of data loading pipeline in training deep neural networks. *CoRR*, abs/2005.02130, 2020.

Glossary

List of Acronyms

KITTI	Karlsruhe Institute of Technology and Toyota Technological Institute
ICLR	International Conference on Learning Representations
VoD	View of Delft
LiDAR	Light Detection and Ranging
ICP	Iterative Closest Point
CPD	Canonical Polyadic Decomposition
CP-ALS	Canonical Polyadic - Alternating Least Squares
TT	Tensor Train
TT-SVD	Tensor Train Singular Value Decomposition
MLSVD	Multilinear Singular Value Decomposition
SVD	Singular Value Decomposition
HOOI	Higher-Order Orthogonal Iteration
PCC	Point Cloud Compression
BEV	Birds-Eye-View
V-PCC	Video-based Point Cloud Compression
G-PCC	Geometry-based Point Cloud Compression
codec	coder-decoder
PSNR-NN	Peak Signal-to-Noise Ratio Nearest Neighbour Loss
BPP	Bits Per Point

List of Symbols

ϵ	Relative error

i_{max}	Maximum amount of iterations
R	Rank