# TENSOR-TRAIN POINT CLOUD COMPRESSION AND EFFICIENT APPROXIMATE NEAREST-NEIGHBOR SEARCH

**Novikov Georgii**
Skolkovo, Moscow, Russia
AIRI, Moscow, Russia
georgii.novikov@skoltech.ru

**Alexander Gneushev**
Moscow Institute of Physics and Technology
Dolgoprudny, Moscow Region, Russia

**Alexey Kadeishvili**
Moscow Institute of Physics and Technology
Dolgoprudny, Moscow Region, Russia

**Ivan Oseledets**
Skolkovo, Moscow, Russia
AIRI, Moscow, Russia
I.Oseledets@skoltech.ru

## ABSTRACT

Nearest-neighbor search in large vector databases is crucial for various machine learning applications. This paper introduces a novel method using **tensor-train** (TT) low-rank tensor decomposition to efficiently represent point clouds and enable fast approximate nearest-neighbor searches. We propose a probabilistic interpretation and utilize density estimation losses like Sliced Wasserstein to train TT decompositions, resulting in robust point cloud compression. We reveal an inherent hierarchical structure within TT point clouds, facilitating efficient approximate nearest-neighbor searches. In our paper, we provide detailed insights into the methodology and conduct comprehensive comparisons with existing methods. We demonstrate its effectiveness in various scenarios, including out-of-distribution (OOD) detection problems and approximate nearest-neighbor (ANN) search tasks.

## 1 INTRODUCTION

Nearest-neighbor search in large vector databases (clouds of high-dimensional points) constitutes a fundamental component in numerous computer science applications, like local or global image matching, semantic search, out-of-distribution and anomaly detection, among others.

In this paper, we propose to use *tensor-train* (TT) Oseledets (2011) low-rank tensor decomposition to represent point clouds and to facilitate fast approximate nearest-neighbor search. The TT decomposition method has gained prominence in deep learning, primarily to compress large tensors of neural network parameters. When representing point clouds as matrices, we artificially order vectors and thus direct application of TT to compress this matrix with standard tensor approximation algorithms (such as TT-SVD Oseledets (2011); Cichocki et al. (2016), TT-cross approximation Oseledets & Tyrtyshnikov (2010) or ALS Holtz et al. (2012)) would result in very poor quality. To address this issue, we propose probabilistic interpretation of the point cloud. Specifically, we suggest training TT to compress a point cloud using loss functions from density estimation, notably Sliced Wasserstein loss, through standard gradient descent method: Section 2.

Moreover, we observe the ability of the TT point cloud to implicitly contain inner hierarchical structure of points: we can consider it as a hierarchical KMeans clustering and TT format allows fast computation of centroids of different levels. This observation allows us to build a highly efficient approximate nearest-neighbors search algorithm on top of the compressed TT point cloud: Section 2.

Due to the nature of the proposed method, points inside the compressed TT point cloud do not have a direct correspondence with points in original point cloud. However, in out-of-distribution (OOD) detection methods that use distance to the normal point cloud as a score, this is not a problem. The state-of-the-art method that builds upon this methodology is PatchCore Roth et al. (2022). Patchcore utilizes greedy coreset subsampling to reduce redundancy in the point cloud and required storage memory. We demonstrate how to use TT point cloud compression in this setting in Section 2 and

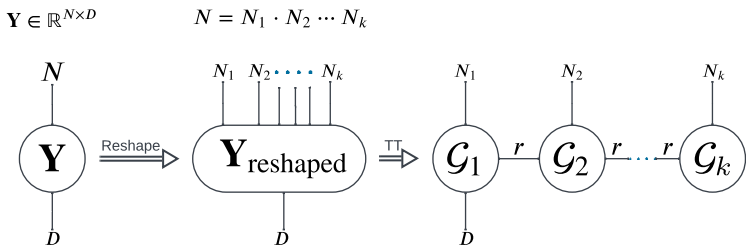



Figure 1: Diagram of the TT point cloud in Penrose graphical notation. Each tensor is depicted as a vertex, and each vertex has as many edges as the dimensionality of the corresponding tensor. Two tensors are connected with a common edge if these two tensors are contracted along the corresponding dimension.

show that our method can significantly outperform coreset-based point cloud subsampling in Sections 2 and 3.2.

For the case of nearest-neighbor search, where the output of the search should be from the original point cloud we suggest to use TT as a search index structure as a replacement for indexes like FAISS Johnson et al. (2017), IMI Babenko & Lempitsky (2015), (G)NO-IMI Babenko & Lempitsky (2016), as detailed in Sections 2 and 3.3.

## 2 METHOD

**Point Cloud Tensorization.** In the context of the OOD detection based on normal (in-distribution) points or ANN problems requiring k-nearest-neighbor search, we begin by considering a set of $D$-dimensional vectors $\mathcal{Y} = \left\{\boldsymbol{y}_i \in \mathbb{R}^D\right\}_{i=1}^N$. It is a natural approach to store this set as a matrix $\boldsymbol{Y} \in \mathbb{R}^{N \times D}$, such that $\boldsymbol{y}_i$ corresponds to the $i$-th row of matrix $\boldsymbol{Y}$.

Low-rank tensor methods are widely popular in tensor compression and representation. We chose *tensor-train* (TT) decomposition due to its efficiency and computational synergies with our method. TT decomposition of a matrix $\boldsymbol{Y}$ starts with matrix tensorization, which involves reshaping the matrix into a high-dimensional tensor. In our specific case, we factorize rows dimension (samples dimension) of matrix $\boldsymbol{Y}$: $N = N_1 \times N_2 \times \cdots \times N_k$ while leaving the columns dimension (features dimension) $D$ unchanged:

$$\boldsymbol{\mathcal{Y}}_{\text{reshaped}} = \text{reshape}(\boldsymbol{Y}, [N_1, \cdots, N_k, D]) \tag{1}$$

Low-rank decomposition of tensor $\boldsymbol{\mathcal{Y}}_{\text{reshaped}}$ is parameterized by $k$ 3-dimensional tensors, referred to as tensor-train cores. The first core, $\boldsymbol{\mathcal{G}}_1 \in \mathbb{R}^{D \times N_1 \times r_1}$, factorizes out feature dimension $D$ and the first sample dimension $N_1$. The remaining $k-1$ cores, $\boldsymbol{\mathcal{G}}_i \in \mathbb{R}^{r_{i-1} \times N_i \times r_i}$, separate all remaining sample dimensions $N_2, \cdots, N_k$:

$$\boldsymbol{\mathcal{Y}}_{\text{reshaped}}[i_1, \cdots, i_k, d] = \boldsymbol{\mathcal{G}}_1[d, i_1, :]\boldsymbol{\mathcal{G}}_2[:, i_2, :] \cdots \boldsymbol{\mathcal{G}}_k[:, i_k, :]. \tag{2}$$

Here, $(i_1, \cdots, i_k)$ represents a multi-index of a sample, which replaces the standard index $i$ after reshape. The dimensions $r_1, \cdots, r_k$ are known as *tensor-train ranks* (TT-ranks) and are subject to the constraint that $r_k = 1$. While TT-ranks are hyperparameters and in general can be different, in our work, we take $r_1 = \cdots = r_{k-1} = r$. The entire TT construction is depicted in Figure 1 using Penrose notation. In the subsequent sections, we denote the matrix obtained as TT with cores $\boldsymbol{\mathcal{G}} = (\boldsymbol{\mathcal{G}}_1, \cdots, \boldsymbol{\mathcal{G}}_k)$ as $\boldsymbol{Y}_{\text{TT}}(\boldsymbol{\mathcal{G}})$ or simply $\boldsymbol{Y}_{\text{TT}}$.

Utilizing the TT point cloud representation allows for a substantial reduction in memory consumption, shifting from $\mathcal{O}(ND)$ to $\mathcal{O}(DN_1r_1 + \sum_i r_{i-1}N_ir_i)$. By defining $N_{\max}$ as $N_{\max} = \max_i N_i$ this can be expressed as $\mathcal{O}(DN_1r + kr^2N_{\max})$, showcasing a potential exponential memory advantage depending on the hyperparameters $\{N_i\}_{i=1}^k$ and $r$. Achieving accurate training for such point cloud compression poses a significant challenge: there exist efficient methods to obtain a TT approximation with a given rank for a given matrix (such as SVD-based methods Oseledets (2011) and optimization-based methods Oseledets & Tyrtyshnikov (2010); Holtz et al. (2012)), the

immediate application of such methods to the matrix $\boldsymbol{Y}$ can yield variable results. This is because rearranging the rows of matrix $\boldsymbol{Y}$ can influence its TT-rank. This implies that for some row orders (enumerations of vectors in the set $\mathcal{Y}$), a more accurate compression of the matrix $\boldsymbol{Y}$ into TT format with a low rank may be achieved, while for other row orders, compression can be highly inaccurate.

To address the sensitivity of TT approximation to row ordering in the matrix $\boldsymbol{Y}$, we propose a new, probabilistic interpretation of the point cloud compression. We consider matrix $\boldsymbol{Y}$ (set $\mathcal{Y}$) as the first point cloud, with elements drawn from a probability distribution $p_{\boldsymbol{Y}}$. The vectors, encoded in TT format within the matrix $\boldsymbol{Y}_{\mathrm{TT}}$, represent the second point cloud, defining a discrete finite distribution $p_{\boldsymbol{Y}_{\mathrm{TT}}}$. Now, we interpret compression of original point cloud $\mathcal{Y}$ as approximation of distribution $p_{\boldsymbol{Y}}$ with distribution $p_{\boldsymbol{Y}_{\mathrm{TT}}}$, where the latter depends on parameters of TT decomposition – cores $(\boldsymbol{\mathcal{G}}_1, \cdots, \boldsymbol{\mathcal{G}}_k)$. This way thus, approximation no longer depend on the particular numbering order of the points in the original cloud when structuring it as a matrix. To closely approximate the distribution $p_{\boldsymbol{Y}}$, we adapt density estimation losses and optimize them using standard SGD methods. In our work, we employ a combination of Sliced Wasserstein Loss Deshpande et al. (2018) and Nearest-Neighbor Distance Loss Hennig & Latecki (2003).

**Sliced Wasserstein Loss.** In our work, we utilize the Sliced Wasserstein Loss to train TT parameters. The Wasserstein Distance, defined between two probability distributions, exhibits many appealing properties, making it suitable for scenarios involving discrete distributions and distributions with different supports, which is particularly important in our case. However, for the general $D$-dimensional case, analytical computation of Wasserstein Distance becomes intractable. In the one-dimensional case, Wasserstein Distance has a simple closed-form solution and can be easily calculated for distributions defined by samples:

$$W_1(\{x_i\}_{i=1}^N, \{y_j\}_{j=1}^N) = \sum_k |x_{i_k} - y_{j_k}|, \tag{3}$$

where $i_k$ and $j_k$ are the indices that sort the sequences $x_i$ and $y_j$, correspondingly:

$$x_{i_k} < x_{i_{k+1}}; \quad y_{j_k} < y_{j_{k+1}}. \tag{4}$$

The Sliced Wasserstein Loss is Wasserstein Distance calculated for random one-dimensional projection of the data. This approach combines the best from both worlds by providing a tractable distance measure for high-dimensional distributions.

$$\mathcal{L}_{\mathrm{SW}}(\{\boldsymbol{x}_i\}_{i=1}^N, \{\boldsymbol{y}_i\}_{j=1}^N) = \tag{5}$$

$$\mathop{\mathbb{E}}_{\boldsymbol{u} \in U[S^{D-1}]} W_1(\{\boldsymbol{u}^T \boldsymbol{x}_i\}_{i=1}^N, \{\boldsymbol{u}^T \boldsymbol{y}_j\}_{j=1}^N). \tag{6}$$

Here, $\{\boldsymbol{x}_i\}_{i=1}^N$ and $\{\boldsymbol{y}_j\}_{j=1}^N$ are samples from two distributions. $S^{D-1}$ denotes a unit sphere in the D-dimensional space, and $U[S^{D-1}]$ is a uniform distribution on this unit sphere. During training, we estimate the expectation on the right-hand side of Equation (5) using Monte Carlo estimation.

SW loss works very effectively in the initial optimization stages, capturing the overall similarity in the shapes of two point clouds and achieving strong alignment at a macroscopic level. However, it may struggle to establish precise point-to-point correspondence between the two point clouds at smaller scales. For this reason, we use it in combination with another loss: Nearest-Neighbor Distance loss.

**Nearest-Neighbor Distance** loss is a classical loss commonly used in applications like KMeans clustering. In this loss, for each point in the first cloud, we find its nearest neighbor in the second cloud and then summat such distances:

$$\mathcal{L}_{\mathrm{NN}} = \sum_{i=1}^{N} \min_j \|\boldsymbol{Y}[i] - \boldsymbol{Y}_{\mathrm{TT}}[j]\|^2. \tag{7}$$

In this way, each point in $\boldsymbol{Y}_{\mathrm{TT}}$ moves towards the centroid of its nearest neighbors in $\boldsymbol{Y}$. Since the point cloud $\boldsymbol{Y}$ can be very large, making the computation of $L_{NN}$ computationally intensive, we use an unbiased estimation of it by considering only a random subset of $\boldsymbol{Y}$ (indexed as $i$ in Equation (7)). We do not subsample $\boldsymbol{Y}_{\mathrm{TT}}$ (indexed as $j$ in Equation (7)).

Points in $\boldsymbol{Y}_{\mathrm{TT}}$ that do not have any assigned nearest neighbors (corresponding clusters are empty) are not be affected by the loss. While these points are covered by the Sliced Wasserstein loss, we can consider the inverse situation: for each point in $\boldsymbol{Y}_{\mathrm{TT}}$, we find its nearest neighbor among $\boldsymbol{Y}$ and calculate the average of such distances:

$$\mathcal{L}_{\mathrm{NN}}^{\mathrm{inv}} = \sum_{j=1}^{N} \min_{i} \|\boldsymbol{Y}[i] - \boldsymbol{Y}_{\mathrm{TT}}[j]\|^2 \tag{8}$$

and take linear combination of this two losses with some coefficient $\alpha$:

$$\mathcal{L}_{\mathrm{NN}}^{\mathrm{total}} = \alpha \mathcal{L}_{\mathrm{NN}} + (1-\alpha)\mathcal{L}_{\mathrm{NN}}^{\mathrm{inv}}. \tag{9}$$

**OOD Detection with TT Point Cloud.** TT point cloud compression is well-suited for OOD detection methods that determine the in-distribution or out-of-distribution status of a query point based on its distance to the nearest neighbor in the normal point cloud (a databank of in-distribution samples). That means, that if the normal point cloud originates from some unknown probability distribution $p_{\text{true}}$, then replacing it with another point cloud that comes from the same (or very close) distribution will not degrade the quality of OOD detection. Therefore, replacing the normal point cloud with TT-compressed point cloud in this setting is straightforward. In comparison to standard techniques where the normal cloud is simply subsampled, either randomly or using methods like coresets, memory gain is achieved due to a more parameter-efficient point cloud representation.

**Approximate Nearest-Neighbor Search.** TT-point clouds cannot be directly applied to the approximate nearest-neighbor search (ANN) problem, as the points contained within them do not have a direct correspondence with the original vectors. Instead, they are "resampled" from an approximation of the unknown underlying distribution $p_{\text{true}}$. To efficiently address the ANN problem, many approaches first construct an *index* structure on top of the vector database. These index structures serve as sparsifications with a significantly lower number of vectors. All vectors from the database are organized into buckets corresponding to the nearest point from the index structure. For each query vector, one must first find the $K$ nearest neighbors from the index structure and then perform an exhaustive search within their respective buckets.

We propose adapting this approach and using TT-compressed point clouds as an index structure. This approach allows for a much larger in-memory index with better coverage of the vector database. Consequently, this results in more evenly distributed bucket sizes, fewer empty buckets, and overall improved search performance.

**Hierarchical Structure.** The TT format of point cloud representation allows rapid computation of mean vectors along any suffix of indices, forming a hierarchical clustering structure. This hierarchical structure can be represented as a tree of depth $k$ with $N_1 \times \cdots \times N_i$ vertices at level $i$. All $N_1 \times \cdots \times N_k$ leaves in that tree are points from the TT point cloud, and non-leaf vertices are centroids (mean vectors) of the corresponding subtrees. This allows for a beam search-like greedy approach to ANN, where we iteratively select the top $K$ nearest neighbors from cluster centroids at each level, progressively refining the search candidates. The detailed explanation and the complete algorithm, including pseudocode, are provided in Appendix B.

## 3 Experiments

### 3.1 Toy Examples

To provide some intuition about TT point clouds, consider the toy examples shown in Figure 2. Each toy point cloud consists of 8192 vectors, and we compress it with a TT representation consisting of two cores with sample dimensions 64 and 128 and a TT-rank of 8. For each toy point cloud, you can observe the centroids of the first and second levels (see Equation (13)) in the first and second columns, respectively. In the third column of Figure 2, you can see the probability density of the original point cloud, and in the fourth column, the probability density of the TT point cloud. The trained vectors reconstruct the original point clouds very well, capturing all important details. They are able to approximate both one- and two-dimensional manifolds: the first point cloud consists of a one-dimensional semicircle, the second point cloud consists of 16 one-dimensional circles arranged

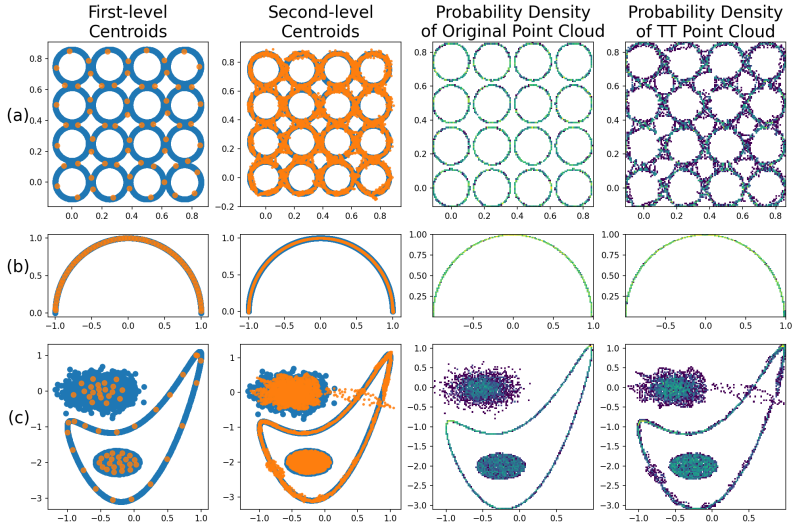



Figure 2: Three toy point clouds (blue points) consisting of 8192 vectors each, and its compressed TT-point cloud approximation (orange points).

in a two-dimensional grid, and the third point cloud consists of three sub-clouds: one with a uniform distribution, one with a normal distribution, and one with a one-dimensional curve featuring a non-uniform distribution (see details in Appendix D).

### 3.2 MVTEC DATASET

MVTec AD Bergmann et al. (2019) is an anomaly detection benchmark that contains 15 different datasets. Each contains training set of anomaly-free images and a test set that contains both images with and without anomalies. Each anomaly has a mask, so per-pixel metrics can be evaluated.

To reduce the memory consumption of the feature databank, the PatchCore method employs coreset subsampling, which greedily solves the following optimization problem over subsampled indices $\mathcal{I} = \{i_k\}_{k=1}^{N'}$ for databank $\mathcal{Y} = \{\boldsymbol{y}_i\}$:

$$\arg\min_{\mathcal{I}} \left\{ \max_{\boldsymbol{y} \in \mathcal{Y}} \min_{i \in \mathcal{I}} \|\boldsymbol{y} - \boldsymbol{y}_i\| \right\}.$$

Here, $N' = cN$, where $N$ is the size of the original feature databank, and $c$ is a subsampling factor. Instead of using coreset subsampling, we compress the point cloud $\mathcal{Y}$ into a TT Point Cloud $\boldsymbol{Y}_{\mathrm{TT}}$. We use TT representation with two cores: the first one of size $1024 \times N_1 \times r$ and the second of size $r \times N_2$. In all our experiments, for a given subsample factor $c$, we choose hyperparameters for the TT point cloud, namely the number of sample factors $N_1$ and $N_2$ and the rank $r$, so that the total number of parameters in the TT $(1024 N_1 r + N_2 r)$ equals to the total parameters in the coreset-subsampled feature databank $(1024cN)$. We choose the same hyperparameters for all MVTec datasets to avoid possible hyperparameter overfitting and to demonstrate the stability of our method, which can be highly desirable in real-world use cases where the method may be applied to related but previously unseen domains.

The standard comparison metrics include pixel-level and image-level AUROC. While we report these values, we found that they are not sufficiently representative. For most of the subdatasets in the MVTec benchmark, the AUROC values are extremely close to 1, often exceeding 0.99. Instead, we also report the area under the precision-recall curve and a specific value, $P@R90$, which represents precision when recall is 90%. This metric is much more challenging for the MVTec benchmark, as some datasets show $P@R90$ values as low as 0.19 for the full (not subsampled) PatchCore feature databank.

In Table 1, we present results for 1% subsampling (corresponding to a 100x compression ratio) and 0.1% subsampling (corresponding to a 1000x compression ratio). All pixel-level metrics heavily favor TT Point Cloud, and as the compression ratio increases, the gap only widens. Image-level metrics are quite competitive with coreset subsampling, with TT Point Cloud showing a slight advantage.

### 3.3 Approximate Nearest Neighbors

ANN is a very well-developed and competitive field, with highly optimized solutions, both algorithmically and in terms of implementation. There are many state-of-the-art libraries and frameworks like FAISS git (a) or NGT git (b), which have undergone extensive low-level optimizations. We acknowledge that it is challenging to compete with such high-end solutions. Instead, we propose a proof-of-concept solution for using TT point cloud as an index structure. In our evaluation, we focus on indirect characteristics such as the quality of dataset coverage, rather than providing actual queries-per-second values.

We conduct our experiments on the Deep1B Babenko & Lempitsky (2016) – dataset, that contains one billion 96-dimensional vectors, that was produced as an outputs from last fully-connected layer of ImageNet-pretrained GoogLeNet Szegedy et al. (2015) model, then compressed by PCA to 96 dimensions and $l_2$-normalized. In our experiments we use only subset of size $10M$ vectors.

We build proof-of-concept ANN system based on TT point cloud, where we use TT point cloud as an indexing tree structure, which plays a role of first-stage database filtration. During training step, each vector in the database is distributed to the bucket of the closest to it point from TT point cloud. During inference, for the query vector $\mathbf{q}$ we first search for $K$ nearest points from TT point cloud, using efficient hierarchical search technique, described in Section 2. Then look for nearest neighbor only inside short-list, that consists of the corresponding buckets.

We compare with similar technique (G)NO-IMI Babenko & Lempitsky (2016). It uses the same idea with two-level indexing tree structure of the form

$$\{\mathbf{S}_i + \alpha_{i,j}\mathbf{T}_j\}\,, \tag{10}$$

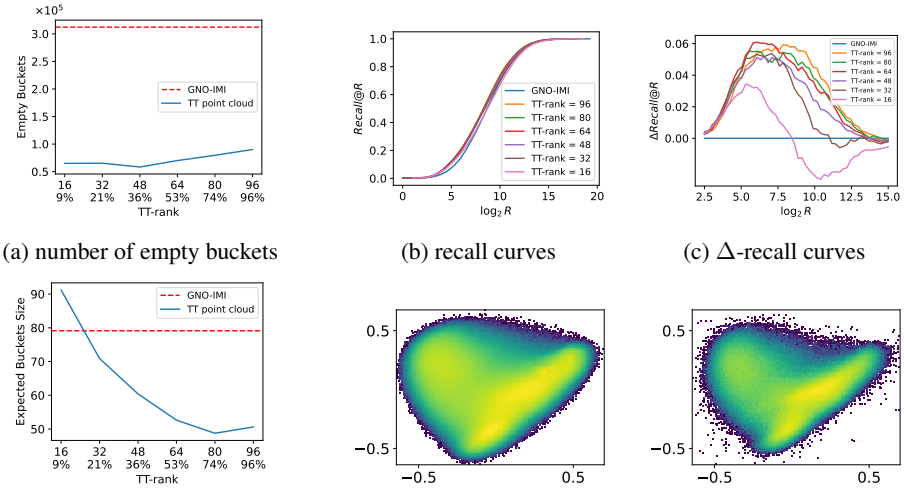
where $\mathbf{S}_i$ and $\mathbf{T}_j$ represent two sets of vectors, first and second-level, respectively. $\alpha$ is an additional weight matrix that is used only in GNO-IMI modification (for NO-IMI $\alpha_{i,j} = 1 \forall i, j$), that adds more flexibility for the construction with the cost of more memory. In the greedy search approach of GNO-IMI, the algorithm initially seeks the top $K$ candidates among $\mathbf{S}_i$ and subsequently explores the best $K$ candidates among $\mathbf{S}_i + \alpha_{i,j}\mathbf{T}_j$ for the indices $i$ determined in the preceding step. In their work Babenko & Lempitsky (2016), it is suggested to utilize sets $\{\mathbf{S}_i\}$ and $\{\mathbf{T}_j\}$ of equal size $2^{14}$. However, given that we are working with a subset of the Deep1B dataset, we have adjusted this quantity to $2^{10}$. This adaptation results in a total of $2^{20}$ vectors in the index structure, encompassing $2 \cdot 96 \cdot 2^{10} + 2^{20} = 1.2$ million parameters.

We compare this structure with a TT point cloud containing three cores, with sample dimensions of 64, 64, and 256, resulting in the same number of buckets. To match the memory consumption of the TT point cloud and GNO-IMI, we can use a TT-rank as large as 96.

The complexity of ANN search with an indexing structure consists of two factors. First, how fast and efficiently we can search for bucket candidates, and second, how good these buckets are. Even if the indexing structure somehow manages to store many vectors using a small amount of memory, if most of the corresponding buckets are empty and only a few of them contain the full vector database, then the second stage of the search will still be required to exhaustively search through a large proportion of the database. If the indexing structure has $M$ buckets, with $i$-th bucket of size $N_i$, then the minimum number of points that need to be searched to find the nearest neighbor of vector $\mathbf{q}$ is given by:

$$\sum_{i=1}^{M} p_i N_i, \tag{11}$$

where $p_i$ is the probability that the nearest neighbor of $\mathbf{q}$ is among the $i$-th bucket. We can estimate $p_i$ based on the size of the $i$-th bucket itself as $\frac{N_i}{N}$, where $N = \sum_{i=1}^{M} N_i$ is the total number of




(a) number of empty buckets (b) recall curves (c) $\Delta$-recall curves

(d) expected bucket sizes (e) original point cloud (f) TT-point cloud

Figure 3: (a, b, c, d): Comparison of different characteristics of the index search structure for GNO-IMI and TT-point cloud with varying TT-ranks. On the x-axis, we plot TT-rank (upper value) and the fraction of parameters of the TT point cloud relative to the number of parameters in GNO-IMI (bottom value). (e, f): Two-dimensional PCA projections of the original point cloud (e) and TT-point cloud (f).

vectors in the database. Then, the expected number of points to be searched through is:

$$\sum_{i=1}^{M} p_i N_i = \sum_{i=1}^{M} \frac{N_i}{N} N_i \propto \sum_{i=1}^{M} N_i^2 \tag{12}$$

On Figures 3e and 3f you can see two-dimensional PCA projections of the original point cloud (which has shape $10\text{M} \times 96$) and TT-compressed point cloud (representing point cloud of size $1M \times 96$ but taking only 671K parameters).

Figure 3d compares dependence of Equation (12) and Figure 3a compares number of empty buckets for GNO-IMI and TT with different ranks. TT point cloud has significantly less empty buckets: almost $\times 6$ less for all ranks. Expected bucket size Equation (12) is lower already with TT-rank $= 32$ (only 21% of GNO-IMI size).

Figure 3b illustrates $Recall@R$ for different values of $R$ – rate of queries for which true nearest neighbor is present in a short-list of length $R$. As the curves are quite similar, we built a plot of curve difference with GNO-IMI as a baseline: we plot original $Recall@R$ curve minus GNO-IMI $Recall@R$ curve on Figure 3c. All TT-ranks starts with quite high advantage over GNO-IMI and the larger the rank – the longer and bigger this advantage continues. Although at the end for recalls very close to one there is a small period of time when GNO-IMI is better.

## 4 Related Work

TT Point Cloud is built upon tensor-train low-rank decomposition, which has a rich history of applications in Machine Learning and Deep Learning. Tensor-Train Matrix decomposition, a specialized form of TT decomposition, shares common principles with our decomposition approach. It has been employed to compress large weight matrices of linear classification layers in fully-convolutional networks Novikov et al. (2015), vocabulary embedding layers in language models Hrinchuk et al. (2020), and linear layers in transformer-based networks Chekalina et al. (2023) and others Phan et al. (2020). TT decomposition has demonstrated its efficiency in parameterizing high-dimensional distributions Novikov et al. (2021); Kuznetsov et al. (2019).

(a) $\times 100$ compression ratio.

| dataset | pixel p@r=90 Our | pixel p@r=90 Coreset | pixel AUROC Our | pixel AUROC Coreset | pixel AUPRC Our | pixel AUPRC Coreset | image p@r=90 Our | image p@r=90 Coreset | image AUROC Our | image AUROC Coreset | image AUPRC Our | image AUPRC Coreset |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bottle | ***0.780*** | 0.777 | ***0.990*** | 0.988 | ***0.904*** | 0.901 | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** |
| Cable | ***0.545*** | 0.396 | ***0.986*** | 0.971 | ***0.830*** | 0.792 | ***0.993*** | 0.991 | ***0.995*** | 0.994 | ***0.997*** | 0.996 |
| Capsule | ***0.366*** | 0.339 | ***0.988*** | 0.979 | ***0.661*** | 0.646 | 0.993 | ***0.994*** | ***0.990*** | 0.988 | ***0.998*** | 0.998 |
| Carpet | ***0.496*** | 0.474 | ***0.986*** | 0.985 | ***0.805*** | 0.802 | ***1.000*** | 0.995 | ***0.982*** | 0.981 | ***0.995*** | 0.994 |
| Grid | ***0.398*** | 0.294 | ***0.986*** | 0.976 | ***0.757*** | 0.734 | ***1.000*** | ***1.000*** | 0.987 | ***0.990*** | 0.996 | ***0.997*** |
| Hazelnut | ***0.634*** | 0.610 | ***0.989*** | 0.987 | ***0.843*** | 0.834 | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** |
| Leather | 0.478 | ***0.479*** | ***0.995*** | 0.995 | ***0.735*** | 0.734 | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** | ***1.000*** |
| Metal Nut | ***0.809*** | 0.710 | ***0.981*** | 0.965 | ***0.929*** | 0.909 | ***1.000*** | ***1.000*** | ***0.999*** | 0.998 | ***1.000*** | 1.000 |
| Pill | ***0.365*** | 0.265 | ***0.967*** | 0.952 | ***0.810*** | 0.792 | ***0.991*** | 0.990 | 0.975 | ***0.975*** | 0.996 | ***0.996*** |
| Screw | ***0.155*** | 0.075 | ***0.984*** | 0.961 | 0.562 | ***0.574*** | 0.959 | ***0.963*** | 0.952 | ***0.962*** | 0.979 | ***0.985*** |
| Tile | ***0.645*** | 0.641 | ***0.972*** | 0.970 | 0.801 | ***0.810*** | ***1.000*** | ***1.000*** | ***1.000*** | 1.000 | ***1.000*** | 1.000 |
| Toothbrush | ***0.541*** | 0.459 | ***0.991*** | 0.981 | ***0.812*** | 0.785 | ***0.923*** | 0.908 | ***0.929*** | 0.902 | ***0.967*** | 0.953 |
| Transistor | ***0.260*** | 0.135 | ***0.949*** | 0.896 | ***0.673*** | 0.630 | ***1.000*** | 0.997 | 0.993 | ***0.994*** | 0.991 | ***0.992*** |
| Wood | ***0.386*** | 0.367 | ***0.956*** | 0.953 | ***0.706*** | 0.706 | ***1.000*** | 0.999 | 0.988 | ***0.989*** | 0.996 | ***0.997*** |
| Zipper | ***0.520*** | 0.512 | ***0.987*** | 0.985 | 0.784 | ***0.786*** | ***1.000*** | 0.996 | ***0.996*** | 0.993 | ***0.999*** | 0.998 |
| Mean | ***0.492*** | 0.435 | ***0.980*** | 0.970 | ***0.774*** | 0.762 | ***0.991*** | 0.989 | ***0.986*** | 0.984 | ***0.994*** | 0.994 |

(b) $\times 1000$ compression ratio.

| dataset | pixel p@r=90 Our | pixel p@r=90 Coreset | pixel AUROC Our | pixel AUROC Coreset | pixel AUPRC Our | pixel AUPRC Coreset | image p@r=90 Our | image p@r=90 Coreset | image AUROC Our | image AUROC Coreset | image AUPRC Our | image AUPRC Coreset |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Bottle | ***0.755*** | 0.627 | ***0.989*** | 0.976 | ***0.896*** | 0.864 | ***1.000*** | ***1.000*** | ***1.000*** | 1.000 | ***1.000*** | 1.000 |
| Cable | ***0.375*** | 0.151 | ***0.976*** | 0.925 | ***0.769*** | 0.641 | ***0.941*** | 0.913 | ***0.969*** | 0.955 | ***0.982*** | 0.975 |
| Capsule | ***0.280*** | 0.065 | ***0.984*** | 0.927 | ***0.618*** | 0.538 | 0.969 | ***0.973*** | 0.956 | ***0.958*** | 0.990 | ***0.992*** |
| Carpet | ***0.478*** | 0.428 | ***0.986*** | 0.981 | ***0.805*** | 0.792 | ***0.998*** | 0.991 | ***0.981*** | 0.978 | ***0.995*** | 0.994 |
| Grid | ***0.259*** | 0.092 | ***0.978*** | 0.938 | ***0.712*** | 0.625 | 0.970 | ***0.981*** | ***0.976*** | 0.970 | ***0.992*** | 0.990 |
| Hazelnut | ***0.572*** | 0.415 | ***0.986*** | 0.979 | ***0.830*** | 0.786 | ***1.000*** | 0.998 | ***1.000*** | 0.999 | ***1.000*** | 1.000 |
| Leather | ***0.489*** | 0.468 | ***0.995*** | 0.994 | ***0.751*** | 0.723 | ***1.000*** | ***1.000*** | ***1.000*** | 1.000 | ***1.000*** | 1.000 |
| Metal Nut | ***0.644*** | 0.400 | ***0.970*** | 0.933 | ***0.900*** | 0.818 | ***1.000*** | 0.996 | ***0.992*** | 0.976 | ***0.998*** | 0.995 |
| Pill | ***0.314*** | 0.136 | ***0.962*** | 0.920 | ***0.785*** | 0.708 | ***0.976*** | 0.965 | ***0.959*** | 0.946 | ***0.993*** | 0.990 |
| Screw | ***0.064*** | 0.028 | ***0.962*** | 0.913 | ***0.298*** | 0.252 | 0.792 | ***0.800*** | 0.735 | ***0.774*** | 0.894 | ***0.916*** |
| Tile | ***0.635*** | 0.583 | ***0.970*** | 0.962 | ***0.797*** | 0.792 | ***1.000*** | ***1.000*** | ***1.000*** | 0.999 | ***1.000*** | 1.000 |
| Toothbrush | ***0.350*** | 0.165 | ***0.983*** | 0.956 | ***0.777*** | 0.661 | ***0.902*** | 0.792 | ***0.918*** | 0.816 | ***0.964*** | 0.931 |
| Transistor | ***0.156*** | 0.083 | ***0.903*** | 0.805 | ***0.587*** | 0.495 | ***0.977*** | 0.961 | 0.956 | ***0.969*** | 0.965 | ***0.971*** |
| Wood | ***0.390*** | 0.335 | ***0.957*** | 0.948 | ***0.709*** | 0.692 | ***1.000*** | ***1.000*** | 0.989 | ***0.990*** | 0.997 | ***0.997*** |
| Zipper | ***0.457*** | 0.310 | ***0.984*** | 0.968 | ***0.751*** | 0.734 | 0.984 | ***0.999*** | 0.986 | ***0.993*** | 0.996 | ***0.998*** |
| Mean | ***0.415*** | 0.286 | ***0.972*** | 0.942 | ***0.732*** | 0.675 | ***0.967*** | 0.958 | ***0.961*** | 0.955 | ***0.984*** | 0.983 |

Table 1: Metrics for the MVTec AD benchmark comparing TT point cloud-based feature databank subsampling and coreset-based subsampling for Patchcore feature vectors. Values are averaged across 16 runs.

Coreset subsampling Agarwal et al. is a well-established technique with applications in various contexts. In essence, a coreset is a subset $\mathcal{S} \subset \mathcal{A}$ that allows for a good approximation of the solution to a problem over the entire set $\mathcal{A}$.

Patchcore Roth et al. (2022) extracts pixel-level features from a set of normal images and make a decision whether a query point belongs to the in-distribution or out-of-distribution class based on the distance to the nearest neighbor among this databank. To reduce memory consumption of the databank and speed up NN search, Patchcore utilizes a specific type of coreset that is particularly well-suited for optimizing nearest-neighbor search performance Sener & Savarese (2018). Other methods that use pixel-level feature databank are SPADE Cohen & Hoshen (2021) and PaDiM Defard et al. (2020). But they do not use the full set of vectors during inference, as SPADE incorporates a distinct initial image-level filtering procedure to operate with relatively small sets of feature vectors, and PaDiM learns a parametric probabilistic representation of the normal class. Notably, Patchcore exhibits significant improvements over similar methods, underscoring the effectiveness of using the full feature databank and the importance of its efficient representation.

ANN search is a crucial problem with various solutions, employing different techniques, including locally-sensitive hashing Jafari et al. (2021), graph-based approaches Wang et al. (2021). IVFADC Jégou et al. (2011) constructs a one-level indexing tree structure, effectively performing k-means clustering. IMI Babenko & Lempitsky (2015) utilizes product quantization with two components instead of vector quantization for the bucket centroids, resulting in an index of quadratic size when compared to the IVFADC approach. (G)NO-IMI Babenko & Lempitsky (2016) builds upon this concept and develops a two-level indexing structure with the capability of approximate nearest neighbor search within it, achieving a significantly better balance between retrieval speed and recall.

## 5 Conclusion

The use of large vector databases in modern machine learning raises the question of their efficient storage and fast vector retrieval. We propose using tensor-train low-rank tensor decomposition to represent large point clouds with a small number of parameters. We introduce a probabilistic interpretation of point cloud approximation to achieve a compression method that is invariant to vector reordering in point cloud. Minimization of Sliced Wasserstein Equation (5) and Nearest-Neighbor Distance Loss Equation (9) allows to interpret points from TT point cloud as a newly

generated points, sampled from the same distribution as points in the original point cloud. TT point clouds are particularly suitable for methods interested in the underlying distribution rather than the vectors themselves, such as out-of-distribution detection based on feature databanks. We tested the proposed OOD feature databank compression method and compared it to coreset subsampling on MVTec AD benchmark. In the crucial area of approximate nearest-neighbor search, where TT point clouds cannot be directly applied, we proposed a proof-of-concept solution: using TT Point Cloud as an indexing structure for the initial filtering of a small number of vectors, in which the nearest neighbors can then be searched through by exhaustive search. We demonstrated the advantages of this approach on the Deep1B dataset in comparison to another indexing method, GNO-IMI. All experiments and implementation of the proposed method can be found in `https://github.com/PgLoLo/tt_point_cloud`.

## REFERENCES

GitHub - facebookresearch/faiss: A library for efficient similarity search and clustering of dense vectors. — github.com. `https://github.com/facebookresearch/faiss`, a. [Accessed 25-09-2023].

GitHub - yahoojapan/NGT: Nearest Neighbor Search with Neighborhood Graph and Tree for High-dimensional Data — github.com. `https://github.com/yahoojapan/NGT`, b. [Accessed 25-09-2023].

Pankaj K Agarwal, Sariel Har-Peled, and Kasturi R Varadarajan. Geometric Approximation via Coresets.

Artem Babenko and Victor Lempitsky. Efficient Indexing of Billion-Scale Datasets of Deep Descriptors. pp. 2055–2063, 2016. URL `https://openaccess.thecvf.com/content_cvpr_2016/html/Babenko_Efficient_Indexing_of_CVPR_2016_paper.html`.

Artem Babenko and Victor S. Lempitsky. The inverted multi-index. *IEEE Trans. Pattern Anal. Mach. Intell.*, 37(6):1247–1260, 2015. doi: 10.1109/TPAMI.2014.2361319. URL `https://doi.org/10.1109/TPAMI.2014.2361319`.

Paul Bergmann, Michael Fauser, David Sattlegger, and Carsten Steger. Mvtec AD - A comprehensive real-world dataset for unsupervised anomaly detection. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pp. 9592–9600. Computer Vision Foundation / IEEE, 2019. doi: 10.1109/CVPR.2019.00982. URL `http://openaccess.thecvf.com/content_CVPR_2019/html/Bergmann_MVTec_AD_--_A_Comprehensive_Real-World_Dataset_for_Unsupervised_Anomaly_CVPR_2019_paper.html`.

Viktoria Chekalina, Georgii S. Novikov, Julia Gusak, Ivan V. Oseledets, and Alexander Panchenko. Efficient GPT model pre-training using tensor train matrix representation. *CoRR*, abs/2306.02697, 2023. doi: 10.48550/arXiv.2306.02697. URL `https://doi.org/10.48550/arXiv.2306.02697`.

Andrzej Cichocki, Namgil Lee, Ivan V. Oseledets, Anh Huy Phan, Qibin Zhao, and Danilo P. Mandic. Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions. *Found. Trends Mach. Learn.*, 9(4-5):249–429, 2016. doi: 10.1561/2200000059. URL `https://doi.org/10.1561/2200000059`.

Niv Cohen and Yedid Hoshen. Sub-Image Anomaly Detection with Deep Pyramid Correspondences, February 2021. URL `http://arxiv.org/abs/2005.02357`. arXiv:2005.02357 [cs].

Thomas Defard, Aleksandr Setkov, Angelique Loesch, and Romaric Audigier. PaDiM: a Patch Distribution Modeling Framework for Anomaly Detection and Localization, November 2020. URL `http://arxiv.org/abs/2011.08785`. arXiv:2011.08785 [cs].

Ishan Deshpande, Ziyu Zhang, and Alexander G. Schwing. Generative modeling using the sliced wasserstein distance. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pp. 3483–3491. Computer Vision Foundation / IEEE Computer Society, 2018. doi: 10.1109/CVPR.

2018.00367. URL http://openaccess.thecvf.com/content_cvpr_2018/html/Deshpande_Generative_Modeling_Using_CVPR_2018_paper.html.

Christian Hennig and Longin Jan Latecki. The choice of vantage objects for image retrieval. *Pattern Recognit.*, 36(9):2187–2196, 2003. doi: 10.1016/S0031-3203(02)00326-6. URL https://doi.org/10.1016/S0031-3203(02)00326-6.

Sebastian Holtz, Thorsten Rohwedder, and Reinhold Schneider. The alternating linear scheme for tensor optimization in the tensor train format. *SIAM J. Sci. Comput.*, 34(2), 2012. doi: 10.1137/100818893. URL https://doi.org/10.1137/100818893.

Oleksii Hrinchuk, Valentin Khrulkov, Leyla Mirvakhabova, Elena Orlova, and Ivan V. Oseledets. Tensorized embedding layers. In Trevor Cohn, Yulan He, and Yang Liu (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pp. 4847–4860. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.436. URL https://doi.org/10.18653/v1/2020.findings-emnlp.436.

Omid Jafari, Preeti Maurya, Parth Nagarkar, Khandker Mushfiqul Islam, and Chidambaram Crushev. A survey on locality sensitive hashing algorithms and their applications. *CoRR*, abs/2102.08942, 2021. URL https://arxiv.org/abs/2102.08942.

Hervé Jégou, Romain Tavenard, Matthijs Douze, and Laurent Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, May 22-27, 2011, Prague Congress Center, Prague, Czech Republic*, pp. 861–864. IEEE, 2011. doi: 10.1109/ICASSP.2011.5946540. URL https://doi.org/10.1109/ICASSP.2011.5946540.

Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs, February 2017. URL http://arxiv.org/abs/1702.08734. arXiv:1702.08734 [cs].

Maksim Kuznetsov, Daniil Polykovskiy, Dmitry P. Vetrov, and Alexander Zhebrak. A prior of a googol gaussians: a tensor ring induced prior for generative models. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 4104–4114, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/4cb811134b9d39fc3104bd06ce75abad-Abstract.html.

Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry P. Vetrov. Tensorizing neural networks. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pp. 442–450, 2015. URL https://proceedings.neurips.cc/paper/2015/hash/6855456e2fe46a9d49d3d3af4f57443d-Abstract.html.

Alexander Novikov, Maxim V. Rakhuba, and Ivan V. Oseledets. Automatic differentiation for riemannian optimization on low-rank matrix and tensor-train manifolds. *SIAM J. Sci. Comput.*, 44(2): 843, 2022. doi: 10.1137/20m1356774. URL https://doi.org/10.1137/20m1356774.

Georgii S. Novikov, Maxim E. Panov, and Ivan V. Oseledets. Tensor-train density estimation. In Cassio P. de Campos, Marloes H. Maathuis, and Erik Quaeghebeur (eds.), *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence, UAI 2021, Virtual Event, 27-30 July 2021*, volume 161 of *Proceedings of Machine Learning Research*, pp. 1321–1331. AUAI Press, 2021. URL https://proceedings.mlr.press/v161/novikov21a.html.

Ivan Oseledets and Eugene Tyrtyshnikov. TT-cross approximation for multidimensional arrays. *Linear Algebra and its Applications*, 432(1):70–88, January 2010. ISSN 00243795. doi: 10.1016/j.laa.2009.07.024. URL https://linkinghub.elsevier.com/retrieve/pii/S0024379509003747.

Ivan V. Oseledets. Tensor-train decomposition. *SIAM J. Sci. Comput.*, 33(5):2295–2317, 2011. doi: 10.1137/090752286. URL https://doi.org/10.1137/090752286.

Anh Huy Phan, Konstantin Sobolev, Konstantin Sozykin, Dmitry Ermilov, Julia Gusak, Petr Tichavský, Valeriy Glukhov, Ivan V. Oseledets, and Andrzej Cichocki. Stable low-rank tensor decomposition for compression of convolutional neural network. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm (eds.), *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXIX*, volume 12374 of *Lecture Notes in Computer Science*, pp. 522–539. Springer, 2020. doi: 10.1007/978-3-030-58526-6_31. URL https://doi.org/10.1007/978-3-030-58526-6_31.

Karsten Roth, Latha Pemula, Joaquin Zepeda, Bernhard Schölkopf, Thomas Brox, and Peter Gehler. Towards Total Recall in Industrial Anomaly Detection. Technical Report arXiv:2106.08265, arXiv, May 2022. URL http://arxiv.org/abs/2106.08265. arXiv:2106.08265 [cs] type: article.

Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018. URL https://openreview.net/forum?id=H1aIuk-RW.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pp. 1–9. IEEE Computer Society, 2015. doi: 10.1109/CVPR.2015.7298594. URL https://doi.org/10.1109/CVPR.2015.7298594.

Mengzhao Wang, Xiaoliang Xu, Qiang Yue, and Yuxiang Wang. A comprehensive survey and experimental comparison of graph-based approximate nearest neighbor search. *Proc. VLDB Endow.*, 14(11):1964–1978, 2021. doi: 10.14778/3476249.3476255. URL http://www.vldb.org/pvldb/vol14/p1964-wang.pdf.

# A Appendix

# B Hierarchical Structure

Each row index of the matrix $\boldsymbol{Y}_{\mathrm{TT}}$ corresponds to some multi-index $(i_1, \cdots, i_k)$ of the tensor $\boldsymbol{\mathcal{Y}}_{\mathrm{TT}}$. The TT structure enables rapid computation of marginalization along a suffix of indices in the multi-index. Let's define the mean vector of tensor $\boldsymbol{\mathcal{Y}}_{\mathrm{TT}}$ along some suffix of indices $(i_{a+1}, \cdots, i_k)$ as $\boldsymbol{y}^{(a)}_{i_1,\cdots,i_a}$:

$$\boldsymbol{y}^{(a)}_{i_1,\cdots,i_a} = \frac{1}{N_{a+1}\cdots N_k} \sum_{i_{a+1},\cdots,i_k} \boldsymbol{\mathcal{Y}}_{\mathrm{TT}}[i_1, \cdots, i_k]. \tag{13}$$

The TT representation allows for the efficient evaluation of such sums. Specifically, marginalization along the last index $i_k$ has its own TT structure:

$$\boldsymbol{y}^{(k-1)}_{i_1,\cdots,i_{k-1}} = \boldsymbol{Y}_{\mathrm{TT}}(\boldsymbol{\mathcal{G}}_1, \cdots, \widetilde{\boldsymbol{\mathcal{G}}}_{k-1})[i_1, \cdots, i_{k-1}], \tag{14}$$

$$\widetilde{\boldsymbol{\mathcal{G}}}_{k-1} = \sum_{\alpha=1,i=1}^{r,N_k} \frac{1}{N_k} \boldsymbol{\mathcal{G}}_{k-1}[:,:,\alpha]\boldsymbol{\mathcal{G}}_k[\alpha, i]. \tag{15}$$

Note that the last two cores, $\boldsymbol{\mathcal{G}}_{k-1}$ and $\boldsymbol{\mathcal{G}}_k$, have been contracted together to form a new core, $\widetilde{\boldsymbol{\mathcal{G}}}_{k-1}$. All other cores remain unchanged. The process of marginalization over a suffix of any length can be obtained by induction:

$$\boldsymbol{y}^{(a)}_{i_1,\cdots,i_a} = \boldsymbol{Y}_{\mathrm{TT}}(\boldsymbol{\mathcal{G}}_1, \cdots, \widetilde{\boldsymbol{\mathcal{G}}}_a)[i_1, \cdots, i_a], \tag{16}$$

$$\widetilde{\boldsymbol{\mathcal{G}}}_a = \sum_{\alpha=1,i=1}^{r,N_{a+1}} \frac{1}{N_{a+1}} \boldsymbol{\mathcal{G}}_a[:,:,\alpha]\widetilde{\boldsymbol{\mathcal{G}}}_{a+1}[\alpha, i]. \tag{17}$$

These marginals essentially serve as centroids of clusters, where each cluster is defined by fixing some prefix of indices in $\boldsymbol{Y}_{\mathrm{TT}}$ and together organizes into a hierarchical clustering structure. It is

important to note that there are no guarantees that this clustering is inherently "good", meaning that each cluster is well-localized and does not intersect with other clusters, however, if it is, efficient approximate nearest-neighbor search can be employed. We consider a beam search-like greedy search approach. For a query vector $\mathbf{q}$, we select the $K$ nearest neighbors from among the cluster centroids at the top level:

$$\mathcal{J}_1 \coloneqq \left\{i_{1j}\right\}_{j=1}^{K} = \text{K-}\arg\min_{i_1} \left\|\mathbf{q} - \boldsymbol{y}_{i_1}^{(1)}\right\| \tag{18}$$

Then, in an iterative fashion, we continue to descend in the hierarchical clustering structure while maintaining a set of the $K$ best candidates for the current centroids level.

$$\mathcal{J}_2 \coloneqq \left\{\left(i_{1j}, i_{2j}\right)\right\}_{j=1}^{K} = \text{K-}\arg\min_{\substack{i_1, i_2 \\ \text{s.t.} \\ i_1 \in \mathcal{J}_1}} \left\|\mathbf{q} - \boldsymbol{y}_{i_1,i_2}^{(a)}\right\| \tag{19}$$

$$\cdots \tag{20}$$

$$\mathcal{J}_k \coloneqq \left\{\left(i_{1j}, \cdots, i_{kj}\right)\right\}_{j=1}^{K} = \text{K-}\arg\min_{\substack{i_1, \cdots, i_k \\ \text{s.t.} \\ (i_1, \cdots, i_{k-1}) \in \mathcal{J}_{k-1}}} \left\|\mathbf{q} - \boldsymbol{y}_{i_1,\cdots,i_k}^{(a)}\right\|. \tag{21}$$

The resulting approximate nearest-neighbor is the best candidate among $\mathcal{J}_k$. Both operations, marginalization, and indexing, required for the fast ANN algorithm, can be efficiently implemented for TT in such a way that on each iteration of Equation (19), the algorithm works with only one core of TT representation. The complete algorithm is described in detail with a code listing in Appendix C.

## C BEAM SEARCH

**Algorithm 1:** Beam search ANN for TT-point cloud

**Data:** Parameters of TT-point cloud $\boldsymbol{\mathcal{G}}_1, \cdots, \boldsymbol{\mathcal{G}}_k$. Precomputed marginalization cores $\widetilde{\boldsymbol{\mathcal{G}}}_1, \cdots, \widetilde{\boldsymbol{\mathcal{G}}}_k$ Equation (16). Queary vector $\mathbf{q}$. Parameter $K$ that determines how many nearest neighbors should be found.

**Result:** Set $\mathcal{J} = \left\{(i_{1j}, \cdots, i_{kj})\right\}_{j=1}^{K}$ of approximate nearest neighbors of vector $\mathbf{q}$ among vectors of $\boldsymbol{\mathcal{Y}}_{\text{TT}}$

Take first-level centroids $\boldsymbol{y}_{i_1}^{(1)} = \widetilde{\boldsymbol{\mathcal{G}}}_1[i_1, :]$ ;

Calculate distances between $\mathbf{q}$ and $\boldsymbol{y}_{i_1}^{(1)}$ : $d_i = \left\|\boldsymbol{y}_i^{(1)} - \mathbf{q}\right\|^2$ ;

Take indices of $K$ closest first-level centroids: $\mathcal{J}_1 \leftarrow \left\{i_{1j}\right\}_{j=1}^{K} = \text{K-}\arg\min_i d_i$ ;

Take $K$ slices of first TT-core $\boldsymbol{\mathcal{G}}_1$ to be able to efficiently calculate higher-level centroids with fixed prefix of indices: $\mathbf{L}_j \leftarrow \boldsymbol{\mathcal{G}}_1[:, i_{1j}, :] \forall j = 1..K$;

**for** *current centroids level* $l \leftarrow 2$ **to** $k$ **do**

  Evaluate $l$-th level centroids with indices from $\mathcal{J}_{l-1}$: $\boldsymbol{y}_{j,i_l} \leftarrow \boldsymbol{y}_{i_{1j},\cdots,i_{l-1j},i_l}^{(l)} = \mathbf{L}_{:,j,:}\widetilde{\boldsymbol{\mathcal{G}}}_l[:, i_l]$ ;

  Calculate distances between $\mathbf{q}$ and $\boldsymbol{y}_{j,i_l}$ : $d_{j,i_l} \leftarrow \left\|\boldsymbol{y}_{j,i_l} - \mathbf{q}\right\|^2$ ;

  Take top-K best candidates: $\mathcal{J}' = \{(j_a, i_{l_a})\} = \text{K-}\arg\min_{j,i_l} d_{j,i_l}$ ;

  Update set with indices of current top-$K$ candidates: $\mathcal{J}_l = \left\{(i_{1j}, \cdots, i_l) | (j, i_l) \in \mathcal{J}'\right\}$ ;

  Update state matrices $\mathbf{L}_1, \cdots, \mathbf{L}_K$: $\mathbf{L}_a \leftarrow \mathbf{L}_{j_a}\boldsymbol{\mathcal{G}}_l[:, i_{l_a}, :]$ s.t. $(j_a, i_{l_a}) \in \mathcal{J}'$ ;

**return** $\mathcal{J}_k$;

Efficient implementation of beam search over TT point cloud consists of two components:

1. Fast and efficient marginals calculation to be able to obtain centroids of various levels
2. Fast and efficient maintaining of such centroids, indices of which are already chosen by beam search.

For the first component, we need to precalculate surrogate TT-cores $\widetilde{\boldsymbol{\mathcal{G}}}_1, \cdots, \widetilde{\boldsymbol{\mathcal{G}}}_k$ as described in Equation (16). This has to be done only once and then can be reused for any number of queries.

For the second component, consider calculating $\boldsymbol{y}^{(l)}_{i_1,\cdots,i_l}$ for fixed $i_1, \cdots, i_{l-1}$ and all possible $i_l$:

$$\boldsymbol{y}^{(a)}_{i_1,\cdots,i_l}[d] = \boldsymbol{Y}_{\mathrm{TT}}(\boldsymbol{\mathcal{G}}_1, \cdots, \widetilde{\boldsymbol{\mathcal{G}}}_l)[i_1, \cdots, i_l] \tag{22}$$

$$= \boldsymbol{\mathcal{G}}_1[d, i_1, :]\boldsymbol{\mathcal{G}}_2[:, i_2, :] \cdots \widetilde{\boldsymbol{\mathcal{G}}}_l[:, i_l] \tag{23}$$

$$= \mathbf{L}[d, :]\widetilde{\boldsymbol{\mathcal{G}}}_l[:, i_l], \tag{24}$$

where

$$\mathbf{L}[d, :] = \boldsymbol{\mathcal{G}}_1[d, i_1, :] \cdots \boldsymbol{\mathcal{G}}_{l-1}[:, i_2, :]. \tag{25}$$

In line 1 we maintain all $K$ such auxiliary matrices $\mathbf{L}_1, \cdots, \mathbf{L}_L$: first they are initialized in line 4 and updated on each for-loop iteration in line 10.

Complexity of the beam search is as follows:

1. For the first-level centroids we need to calculate distances in line 2 in $\mathcal{O}(DN_1)$ time and then initialize matrices $\mathbf{L}$ in line 4 in $\mathcal{O}(KDr)$ time.
2. For each other level centroids, we need to calculate centroids in line 6 in $\mathcal{O}(kdrn)$ time, where $n = \max_{i=2..k-1} N_i$. Then update matrices $\mathbf{L}$ in line 10 in $\mathcal{O}(KDr^2)$ time.

The total asymptotic of Alg.1 is $\mathcal{O}(DN_1 + kKDrn + kKDr^2)$ time.

# D Experiment Setups

## D.1 Toy Examples

First toy point cloud is 16 uniform distributions over circles, arranged in 4-by-4 grid. Each circle is chosen uniformly.

Second toy point cloud is a uniform distribution over one half of a circle.

Thirds toy point cloud is a mixture of three distributions with equal weights:

1. non-uniform distribution over 1-dimensional closed loop
2. uniform distribution over disc (blob inside closed loop)
3. normal distribution (blob above closed loop)

## D.2 MVTec AD

For all datasets in MVTec AD benchmark we used same hyperparameters.

We followed original implementation[1] of Patchcore for feature extraction and coreset subsampling. We trained TT point cloud via riemannian stochastic gradient descent Novikov et al. (2022) for $2^{13}$ iterations with initial learning rate equals to $10^3$ and exponential decay of $\frac{1}{3}$ every $256$ iterations. Before multiplying by learning rate, we scale gradient to 1 to achieve some kind of adaptivity for riemannian SGD.

Parameters of the loss function:

1. SW loss with coefficient 1 and batch size 32
2. NN Distance loss with $\alpha = 0.1$ and coefficient $0.1$, with random $2^{11}$ samples from $\boldsymbol{Y}$ on each iteration.

[1] https://github.com/amazon-science/patchcore-inspection

### D.3 APPROXIMATE NEAREST-NEIGHBOR

To train TT point cloud as an ANN indexing structure we performed two-stage optimization. On the first stage we used Riemannian SGD Novikov et al. (2022) with combination of SW loss with coefficient $1$ and batch size 32, and NN Distance loss with $\alpha = 0.1$ and coefficient $0.1$ with random $2^{15}$ samples from $\boldsymbol{Y}$ on each iteration. Initial learning rate $10^2$ with exponential decay of $\frac{1}{3}$ every $2^{11}$ iterations. Number of iterations $2^{13}$.

On the second stage we used ALS Holtz et al. (2012) with only NN Distance loss with $\alpha$ equals to $0.001$ for 256 iterations.