Data Engineering and Analytics

Technische Universität München

Master´s Thesis

SWIM in Point Clouds: Sampling Weights for PointNet

Zi Hen Lin

Data Engineering and Analytics

Technische Universität München

Master's Thesis

SWIM in Point Clouds: Sampling Weights for PointNet

Author:	Zi Hen Lin
Examiner:	Prof. Dr. Felix Dietrich
Assistant advisor:	Erik Lien Bolager
Submission Date:	1st October, 2024

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

1st October, 2024 Zi Hen Lin

Acknowledgments

I would like to express my deepest gratitude to my TUM advisor, Erik Bolager, and my Infineon supervisors, Dr. Thomas Ramcke and Dr. Michael Stadler, for their invaluable patience and feedback. I could not have undertaken this journey without the support of Prof. Dr. Felix Dietrich.
I extend my special thanks to my family and partner, Melody Teng, for their unconditional moral support. I also extend my sincere thanks to Ajvi Meçe for the fruitful discussions and her support.

Abstract

3D machine learning has received growing attention from the industries due to the potential to offer significant time gain over the classical numerical approaches. We introduce a sampling approach to construct PointNet using point clouds, with orders of magnitude better efficiency compared to the conventional iterative gradient descent, at the expense of a small performance loss. This sampling is an extension of Sampling Where It Matters (SWIM). We propose two approaches to preserve the geometric properties in point cloud, namely KDTree and Recursive Sampling. For the latter, we leverage quantile and the length-squared distribution of point coordinates to enhance the sampled representations. The research also tackles space complexity by implementing batch-wise weight and bias updates. Our approach is proven effective and efficient under large-scale settings, specifically, sampled PointNet preserves more than 90% of PointNet's accuracy on classifying the standard 3D benchmark ModelNet40 at the costs of less than 10% of PointNet's GPU training time on a CPU, assuming that the point clouds share a standard orientation.

Contents

1 Introduction

In our three-dimensional (3D) world, computational 3D modeling has become integral to industries like semiconductor, automotive, architecture, and medicine, enhancing design and manufacturing processes with improved precision and reduced experimental costs. This iterative approach generates an abundance of 3D data, represented mathematically for computer processing. While classical numerical methods such as finite element analysis have been pivotal in simulating physical properties of complex 3D geometries, their fine-grained approach often results in slow runtimes. The recent success of deep learning in various domains has sparked interest in using neural networks as efficient surrogates for these methods.

In business contexts, the use of 3D machine learning aims to amortize the cost of slow classical methods upfront into training robust deep learning models, potentially offering significant time gains during inference while maintaining good accuracy. A deep learning model learns by optimizing its weights and biases through iterative gradient descent. The training process of the model could be too expensive to justify a switch from classical approaches, due to its iterative nature and the intricate dimensionality of 3D objects. This text is concerned with sampling the weights and biases of a 3D deep learning model using the differences among inputs to significantly speed up the process without sacrificing the performance too much.

Two early examples, Extreme Learning Machines [Rahimi and Recht, 2008] and Random Projection Networks [Huang et al., 2006], have successfully demonstrated the massive speedup gained of randomizing the weights and biases of classical machine learning models. Extending these ideas to deep learning, Giryes et al. [2016] propose the data-agnostic randomized deep neural network. Galaris et al. [2022] highlight the importance of incorporating inputs in the sampling process but report no further than low-dimensional settings. Recently, Bolager et al. [2023] build on this foundation and show that data-driven random sampling of weights and biases are possible in high dimensional ambient space. They also design a probability distribution to facilitate the sampling process instead of using the original uniform distribution. The proposed method, Sampling Where It Matters (SWIM), has shown superior performance against the previous data-agnostic methods and is comparable to modern deep learning methods on several tasks.

The essence of weight construction through SWIM lies in computing the differences between inputs. It is straightforward to identify different 3D objects, however, it is non-trivial to represent the differences numerically due to the inherent irregular hierarchical structures. Fortunately, 3D data in general use cases often adheres to specific formats, which helps ensure accurate geometric representation. By exploiting the underlying geometric properties in these formats, it is possible to compute the differences meaningfully. We consider three common 3D representations: point clouds, meshes, and voxels, as visualized in Figure 1.1. All three are subject to some forms of discretizations: a point cloud contains points sampled from the surfaces of a 3D object; a mesh represents the shape and surfaces using multiple triangles with different sizes; and a voxel fits an object into a cubified space under a chosen resolution. To understand how the differences can be computed, we review the data structures of these representations.

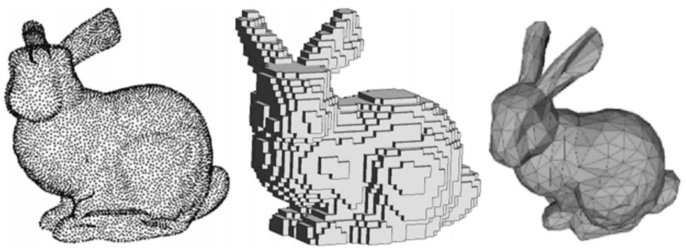


Figure 1.1: This figure is taken from Hoang et al. [2019] to illustrate three different representations of 3D data. **Left:** Point cloud. **Middle:** Voxel. **Right:** Mesh.

Computationally, point clouds are matrices in $\mathbb{R}^{n\times k}$ with the rows representing n points, and the columns for k features, including the coordinates and additional features; meshes are represented with vertex matrices with the coordinates ($\mathbb{R}^{n\times 3}$), face matrices which specify the vertices of the triangles with the index triplets ($\mathbb{R}^{n\times 3}$) and additional feature matrices; since voxels implicitly denote the coordinates, usually tensors in $\mathbb{R}^{x\times y\times z\times k}$ — where the first three axes (x, y, z) shape the coordinate boundaries with the desired granularity, and k is the number of the dimension of the features — are required computationally for this geometric representation. It is technically feasible to compute the differences between the matrices or tensors, but the outcomes might not be geometrically meaningful. Hence, it is of importance to understand the geometry of each representation.

3D meshes are a popular choice for computational modeling because of their flexible triangulated structure, which accurately encodes geometric and material information of a 3D object. However, the number of triangles generated by various meshing algorithms can significantly differ, leading to local inconsistency. 3D mesh inconsistency presents a challenge for general machine learning models: ensuring robustness against the meshing algorithms. Inconsistent representations may mislead models, unless all inconsistencies can be mapped to a single object. This mapping typically requires a large number of examples for accurate registration. Furthermore, this makes computing directions between two meshes more difficult, as the triangulation patterns may not align due to different software implementations.

Voxels are often favored in the machine learning community over meshes, as they are the 3D extension of images — a successful early application in deep learning. 3D voxels have an advantage in geometry: they are locally consistent with the same resolution, simplifying direction computations between two voxel grids. In particular, the local consistency allows for straightforward direction computations, by directly calculating cube-to-cube differences. This is because voxels maintain a uniform resolution, meaning that each cube (voxel) has the same size and orientation. As a result, determining the directional relationship between two voxels is as simple as comparing their positions in the grid, which is computationally less intensive than dealing with the variable triangle counts and structures found in meshes. However, this consistency comes at the cost of disproportionate computational memory requirements, which can be a significant drawback in memory-constrained environments.

Point clouds offer a compelling alternative to both meshes and voxels as a 3D representation. They strike a balance between the flexibility of meshes and the consistency of voxels. Meshes rely on complex triangulation and structural assumptions. These assumptions about connectivity and continuity can can amplify the impact of local variations, affecting the interpretation of the entire representation. Without any inherent connectivity, point clouds are more robust to local inconsistencies. Consequently, point clouds allow for more flexible and intuitive spatial comparisons. Compared to voxels, point clouds can represent 3D objects more efficiently, especially for sparse or detailed structures, as they do not require a fixed grid resolution. This efficiency translates to lower memory requirements and potentially faster processing times. To this end, it makes sense to choose point clouds as our target representation, as we take both performance and computational resources into account.

Once we have chosen a suitable representation, we survey for a fitting model. Classical machine learning methods often struggle with the unordered and variable-sized nature of point clouds. Traditional approaches typically require structured input with a fixed dimensionality, which point clouds do not naturally provide. Moreover, these approaches either require hand-crafted features or consume the raw inputs without respecting the implicit geometric properties, which leads to unnecessary parameters and over-complicated optimization landscape. This is where geometric deep learning architectures like PointNet [Qi et al., 2017a] come into play. PointNet is specifically designed to handle the unique characteristics of point clouds. It uses a series of shared Multi-Layer Perceptrons (MLPs) to process each point identically, followed by a symmetric function to aggregate information across all points. This architecture is invariant to point permutations and can handle input point clouds of varying sizes. PointNet can learn to extract meaningful features directly from the raw point cloud data with significantly fewer parameters, credit to the weight-sharing design.

This thesis explores the application of SWIM to PointNet. We analyze SWIM-PointNet compatibility and extend SWIM to capture geometric properties using KDTree for nearest neighbor calculations. The robustness of sampled PointNet against rotations is evaluated through three distinct approaches. We investigate SWIM's scalability in terms of runtime and performance against increasing data augmentations and point cloud densities. To maintain SWIM's speed advantage, we propose a recursive sampling approach to replace KDTree, addressing the curse of dimensionality. We evaluate SWIM's sampling distribution effectiveness and suggest an alternative using quantiles and length squared distribution of point coordinates. To reduce space complexity, we enable SWIM to update weights and biases with data batches. Finally, we search for the optimal sampled PointNet architecture before evaluating the optimized architecture on a standard 3D deep learning dataset, ModelNet40.

The following sections are organized as such: Section 2 covers the necessary foundations and reviews the relevant literature; Section 3 documents our main research questions with corresponding solutions together with the respective results and interpretations; Section 4 concludes this journey with future outlook.

2 Preliminaries

In this section, we introduce the foundations which we build upon for our main work. Our main work aims to construct PointNet [Qi et al., 2017a] with good performance and reduced training costs using Sampling Where It Matters (SWIM) [Bolager et al., 2023]. We will also discuss high dimensional nearest neighbour search as it is the pillar of our contributions. At the end of this section, we state our problem definition and review the related literature.

2.1 PointNet

PointNet is an instance of deep learning models, in particular, a neural network. A neural network is a machine learning model intended to mimic the biological neural networks in brains. Conceptually, it contains arbitrary number of layers with arbitary number of neurons as shown in Figure 2.1. The architecture of neural network in the figure is called Multi-Layer Perceptron (MLP) with the layer type called Fully-Connected layer [1]. In the figure, its first layer is the input layer with three neurons, which projects the data from input space to a 3D latent space, in which becomes the input space of the second layer, the hidden layer. Similarly, the hidden layer projects the outputs from the input layer to a 4D latent space, before the output layer projects its 4D inputs to the 2D output space. Computationally, these projections between layers are done via matrix multiplication with non-linear activation functions such as sigmoid, tanh, or ReLU. Each column of the matrix is a j-dimensional vector which corresponds to a neuron in the layer and takes in j-dimensional inputs. The number of neurons represents the number of output dimensions.

Using the hidden layer with a single input datum as an example, it is represented as a weight matrix in $\mathbb{R}^{3\times 4}$ with its input being a $\mathbb{R}^3$ vector, and the multiplication of the input vector and the weight matrix produces an output vector in $\mathbb{R}^4$. Mathematically, a layer projection can be denoted as:

$$H^{(l)} = \sigma(W^{(l)}H^{(l-1)} + B^{(l)}), \tag{2.1}$$

where l indicates l-th layer, H is the layer output, W is the layer weights, B is the layer biases

[1]In deep learning literature, the terminology of architecture can have two meanings: the overall design of a neural network model including the number of layers, the number of neurons, and the layer types, or the type of a layer such as Fully-Connected layer. To avoid confusion, in this thesis, we refer architecture to the former, and for the latter we term it neural network operator. Therefore, a typical MLP — regardless of the number of layers and neurons — is a generic architecture consists only more than one Fully-Connected layer.

and $\sigma(\cdot)$ is a non-linear activation function. The weights W and biases B are randomly initialized and optimized via iterative gradient-based methods [Goodfellow, 2016].

Neural networks are often referred to as universal function approximators, as proven by Hornik et al. [1989]. The Universal Approximation Theorem shows that a neural network with sufficient neurons can approximate any continuous function, effectively mapping between two spaces. This greatly enhances the practicality of neural networks and diversifies their application domains. Instead of designing complex algorithms, one can simply train a neural network to map inputs to desired outputs (although without any guarantee that the mapping is good), with the training procedures being largely the same across many different domains, such as classifying protein molecules and semiconductor chips. However, a standard MLP is not an efficient architecture, as it considers all input dimensions individually, leading to a huge number of weights, especially for high-dimensional inputs like $1,000 \times 1,000$ pixel images, which are increasingly common in modern deep learning.

This limitation of standard MLPs has led to the development of several specialized neural network operators, such as Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and Graph Neural Networks (GNNs). These operators are designed to handle different data modalities, such as grids, sequences, and graphs, respectively, by enabling weight-sharing mechanisms and a localized learning approach that considers only the neighboring inputs instead of all inputs (for a collective introduction, see the textbook written by Murphy [2022]). The universal approximation properties of these specialized neural network operators have been proven by Zhou [2020] for CNNs, Schäfer and Zimmermann [2006] for RNNs, and Brüel Gabrielsson [2020] for GNNs. Furthermore, Bronstein et al. [2021] have provided an overarching theoretical framework, known as Geometric Deep Learning, which explains how these neural network operators can exploit the geometric structures of input data to facilitate the learning process in an efficient manner.

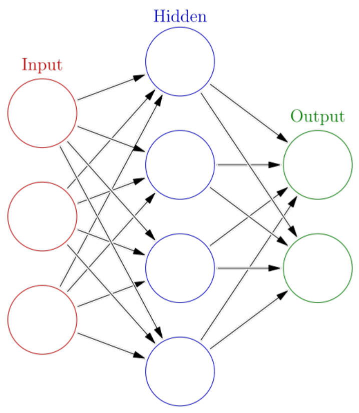


Figure 2.1: This figure visualizes the basic form of neural network — Multi-Layer Perceptron. Taken from Wikipedia [Wikipedia].

2.1.1 Geometric Deep Learning on 3D data

Before the introduction of PointNet by Qi et al. [2017a], there were two main deep learning approaches to learn from 3D data: image-based [Su et al., 2015] and voxel-based [Maturana and Scherer, 2015, Wu et al., 2015]. Both approaches rely heavily on a modern primitive neural network operator, namely convolutional neural network (CNN). The former converts a 3D object into multiple images from different views before feeding into 2D CNNs, while the latter operates on voxelized (or cubified) 3D objects using 3D CNNs. Convolutional-based approaches demand inputs with regular format because this stream of approaches fundamentally partitions the *input space* into grids. The partitioned space establishes a strong assumption to simplify input geometries and enable learning the translational equivariance properties using convolutional kernel [Bronstein et al., 2021]. A g-equivariant function f is defined mathematically as:

$$f([g(x_1), \ldots, g(x_n)]) = [g(f(x_1)), ..., g(f(x_n))], \tag{2.2}$$

where $g(\cdot)$ is an action and x is the input. In the context of translational equivariance, this means that the translated inputs will map to translated outputs with the same translations.

This space-partitions assumption, however, introduces two disadvantage from local and global point of view. Locally, it is analogous to the approximation of integral using Riemann sums, as known as the Rectangle method. The higher the resolution, the more accurate the approximation. This suggests that the space partition turns 3D object representation into an accuracy versus computational resources trade-off, since finer granularity requires more entries given the same space. From the global perspective, regular input format indicates that the "space" has to be a box which can contain the largest value in each axis. The finer the resolution, the more the blank grids, resulting in an inefficient representation with bad scalability due to the curse of dimensionality.

PointNet considers an alternative representation of 3D modality — point clouds, which instances of sets. At its most basic form, a point cloud $\mathbf{X}$ — $\mathbf{X} = \{x_1, x_2, \ldots, x_n\}, \mathbf{X} \in \mathbb{R}^{n \times k}, x_i \in \mathbb{R}^k, \forall i \in \{1, 2, \ldots, n\}$, where n is the number of points in $\mathbf{X}$, k is the number of dimension of a point — is an *unordered* set of 3D coordinates in the Euclidean space. Geometric properties such as normals or domain properties can be added as addition features. In practice, a point cloud is often sampled from the surfaces a mesh to describe the 3D shape using sparse scattered points.

Unlike an image or a voxel, a point cloud does not possess strong geometric assumption, which is a double-edged sword. On one hand, a point cloud cannot presume the coordinates — which are implicitly modeled with grids — and relies on model's capability to discover the underlying structure by including the coordinates as features; on the other hand, this is a very efficient representation as it can describe the irregular shapes of 3D objects precisely, without forcing the shapes into a cubified space. As the volume of 3D data grows, the efficiency of point cloud is increasingly favourable. Given sufficient data, it is feasible to obtain comparable or even better performance without the geometric bias of grids, analogous to end-to-end deep learning emerges as a superior option compared to feature engineering in big data era.

Based on these practical considerations, PointNet was the first modern neural network architecture proposed to specifically train on point clouds. Qi et al. [2017a] propose two key geometric properties of point clouds: unordered and invariant under geometric transformations. The authors tackle the former from the perspective of permutation invariance which disregards the order of inputs by enforcing PointNet to approximate symmetry functions on sets, and the latter with joint alignment networks to embrace the rotational invariance. By capturing these properties, PointNet can be seen as a primitive neural network operator for point clouds, where it concatenates a few layers of Fully-Connected layers with a max pooling function. This max pooling function outputs the maximum values of the input features, effectively aggregating the point features into a single vector.

It is also important to note the work on DeepSets [Zaheer et al., 2017], which was proposed around the same time as PointNet. DeepSets focuses on the theoretical foundations of a similar neural network operator for sets, using summation instead of the max pooling function employed by PointNet. Specifically, Zaheer et al. [2017] characterizes the properties of permutation invariant functions, and Qi et al. [2017a] proves that the permutation invariant operators used in PointNet can perform universal approximation on continuous set functions that are permutation invariant. Next, we carefully review each component of PointNet's architecture to understand their individual contributions to the overall model performance.

2.1.2 Architecture

Figure 2.2 illustrates the architecture of PointNet, which on the top row is pivoted by 5 shared Fully-Connected layers and a MaxPool layer as the operator to extract global features of the point clouds. For classification tasks, one can add a classifier (which the authors use a three-layer MLP) to produce classification logits for the output classes. In case of segmentation tasks, PointNet needs to generate the class logits for each point. Therefore, it concatenates the 64D outputs from the third shared Fully-Connected layer together with n-times 1024D outputs of MaxPool layer before feeding the 1088D concatenated latent features into another MLP consisting 5 shared Fully-Connected layers. For each (shared) Fully-Connected layer, it follows by a ReLU non-linear activation layer and a BatchNorm layer. We purposely leave out the input transform block and feature transform block for now, as these two components are not part of the vanilla PointNet architecture.

Shared Fully-Connected layer is the weight-sharing variant of Fully-Connected layer. In the context of point clouds, a shared Fully-Connected layer would optimize the weights in $\mathbb{R}^{d_{in} \times d_{out}}$ — where d_{in} is the number of input dimensions and d_{out} is the number of output dimensions — for all points, while a typical Fully-Connected layer will have weights in $\mathbb{R}^{n \times d_{in} \times d_{out}}$, that is, each point of n points has its own treatment. Qi et al. [2017a] do not explicitly mention the reason of using shared Fully-Connected layer instead of the regular other than processing each point individually and identically. We believe two additional advantages are the reduced number of parameters, and the flexibility of accepting point clouds with varying number of points. The **MaxPool** layer is a function which takes the maximum of the input along the given dimension, in the case of PointNet, it extracts the maximum values of the 1024 latent features across all points. It is the secret ingredient for PointNet to approximate symmetry functions and achieve permutation invariance.

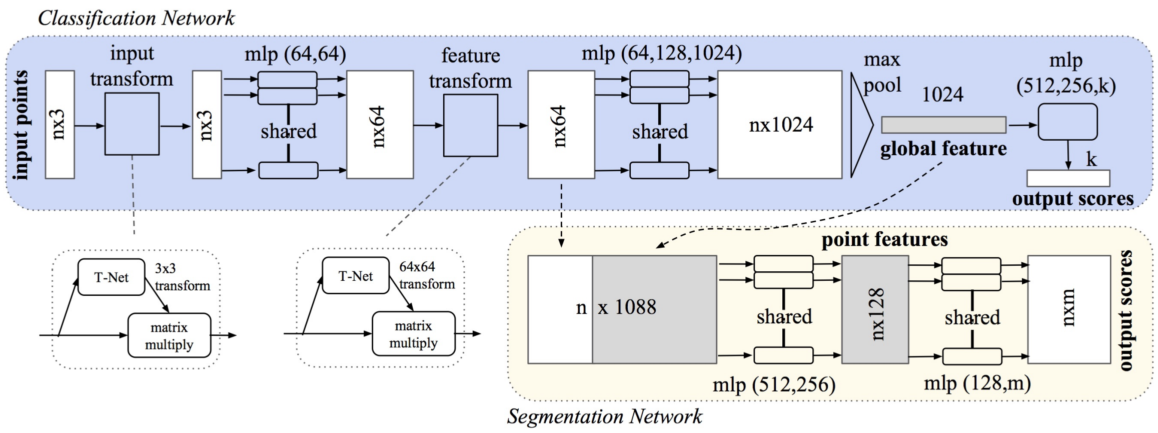


Figure 2.2: The architecture of PointNet. Taken from the literature [Qi et al., 2017a].

ReLU non-linear activation layer can be mathematically formulated as $\max(0, x)$, that is, to preserve the positive values and eliminate the negative values of inputs x. This simple function is important to add a hint of non-linearity to PointNet, since the matrix multiplications of Fully-Connected networks are linear in nature. Therefore, the sandwiches of (shared) Fully-Connected layers with ReLU layers are essentially MLPs, as indicated in Figure 2.2 with the adjacent brackets including corresponding layer widths.

The injections of **BatchNorm** layers stabilize the training process of mini-batch iterative gradient descent by preventing internal covariate shift. It is a common practice for neural network to split datasets into small batches, as known as mini-batch, due to limited computational memory. The distribution of each batch could be vastly different from the rest, causing drastic changes in gradients. Therefore, a BatchNorm layer contains two learnable parameters — which scales and shifts the distribution of outputs as necessary — to rectify the differences in distributions across the batches, leading to stabler gradients and easier training process.

Up to now, we have discussed the components to build a vanilla PointNet. Note that the machine learning literature would sometimes omit ReLU and BatchNorm when describing the architecture, since these two layers are usually coupled with a neural network operator such as a Fully-Connected network. A vanilla PointNet has an encoder consists of 5 layers of shared Fully-Connected layer with sizes of [64, 64, 64, 128,1024] and a MaxPool layer. The architecture of the decoder depends on downstream tasks, as previously mentioned.

A vanilla PointNet can capture the first geometric properties of point cloud, unordered, since the order of the points would not change the outcome due to the identical treatments of shared MLPs and the symmetry of MaxPool layer. However, this does not embrace invariance against geometric transformations. The geometric transformations do not permute the order of the points, but instead, these operations change the values of the features such as coordinates. However, it is an exhausting task to manually neutralize these transformations. In light of this, Qi et al. [2017a] propose an alignment technique to automatically align all inputs in a learned standardized orientation through Transformation Network (T-Net).

A **T-Net** is essentially a reduced vanilla PointNet — three shared Fully-Connected layer with sizes of [64, 128, 1024] and a MaxPool layer as the encoder, and two Fully-Connected layers of width 512 and 256 respectively, as the decoder. It is tasked to predict the inverse of the rotation matrix of a rotated point cloud. Ideally, by applying the 3×3 rotation matrices generated by T-Net, all inputs with random orientations will be aligned. This is shown as the input transform block in Figure 2.2. The authors extend this concept to align the feature space, resulting in the feature transform block which multiplies the outputs of the second shared Fully-Connected layer with the 64×64 (inverse) transformation matrices to align the high dimensional latent features in feature space. It is difficult to discover such transformation matrices, especially in high dimensions. This is where optimization comes into play.

2.1.3 Optimization

In previous sections, we did not mention how PointNet optimizes its randomly initialized weights and biases. In general, neural networks are paired with an objective function during training phase. The objective function, also known as cost function, acts as a guidance in the optimization process. Under the settings of supervised learning, an objective function $\mathfrak{C}(\cdot)$ takes in two parameters: the ground truths and the predicted variables. The optimization problem is usually defined to find the configurations of weights and biases, such that the objective function can be minimized given the inputs and corresponding ground truths.

The training process of neural networks, including PointNet, relies on the gradients of the cost function to update the weights and biases across all layers through backpropagation and the chain rule. Each training iteration comprises two essential phases: the forward pass and the backward pass. During the forward pass, the model performs inference, generating predicted outputs. The cost function then evaluates these predictions against the ground truth labels. Subsequently, the backward pass utilizes the computed gradients to update the weights and biases of the network. This iterative optimization procedure is commonly referred to as gradient descent. It is important to note that in the case of PointNet, the two T-Net modules are integral components of the overall computation graph. Consequently, their weights and biases are updated concurrently with those of the main network body during the backward pass. This simultaneous optimization ensures that all parts of the network, including the input and feature transformation modules, evolve together to improve the model's overall performance.

To solve the instability of the second T-Net, which produces high dimensional transformation matrices, Qi et al. [2017a] insert a regularization term to the cost function of

PointNet:

$$\widetilde{\mathfrak{C}}(\cdot, A) = \mathfrak{C}(\cdot) + ||I - AA^T||_F^2, \tag{2.3}$$

where $\widetilde{\mathfrak{C}}(\cdot, A)$ is the regularized cost function, and A is the 64×64 transformation matrix generated by the second T-Net. The regularization term is the squared Frobenius norm (the counterpart of Euclidean norm for matrix norm) of the difference between an identity matrix, and the matrix multiplications of the transformation matrix A and its transpose. In the ideal case where A is orthogonal, this regularization term will be 0.

Intuitively, this acts as a soft penalty to guide the PointNet to optimize the second T-Net such that it can produce valid transformation matrices which is orthogonal. This is a celebrated technique — joint optimization or indirect supervised learning — tightly coupled with iterative gradient methods and backpropagation. According to Qi et al. [2017a], it takes approximately 3-6 hours for PointNet to converge within 20,000 epochs using gradient descent to learn classification and part-segmentation using the corresponding standard datasets on a GTX1080 GPU. Doing a similar training on a laptop CPU requires 35 days to finish 1,000 epochs. Therefore, we turn our attention to Sampling Where It Matters (SWIM), which can train a model swiftly by orders of magnitude using a CPU under specific caveats.

2.2 Sampling Where It Matters

Sampling is a crucial concept in statistics and machine learning. It involves selecting a subset of samples from a larger population to estimate its characteristics. Various sampling techniques exist, with random sampling being the most common. This method can be uniform, where all members have an equal chance of selection, or weighted, using a specific probability distribution. Stratified sampling, another important technique, divides the population into mutually exclusive subgroups before sampling, ensuring representation across all subgroups and allowing for greater control over sample composition. This approach can be proportional, maintaining the population's distribution, or disproportional, useful for addressing imbalanced datasets. In machine learning, proportional stratified sampling is often employed to create representative train, validation, and test sets, while disproportional stratified sampling can help mitigate the effects of class imbalance in classification tasks.

Sampling could also be a hierarchical process called subsampling. A basic subsampling has two-stages sampling process. The initial process draws samples from the population, and the drawn samples become the sampling frame of the second sampling process. Two stages of sampling could adopt different sampling techniques. For example, a common practice in 3D machine learning is to first sample a dense point cloud with 2048 points from the surfaces of a 3D object, then down-sample to 64 points using a scheme called farthest point sampling, which iteratively samples the farthest point from current point to ensure the entire 3D shape is well represented [Qi et al., 2017a]. SWIM's sampling algorithm — which we will discuss the details in next section — can also be abstractly viewed as a subsampling process. This algorithm is the backbone of SWIM and the main reason behind the huge speedup of runtime compared to iterative gradient descent. Ultimately, sampling is designed to approximate good enough solution with lightweight, one-shot selections while iterative gradient optimization aims to find the best solution available.

2.2.1 Sampling Algorithm

At the time of writing, SWIM can only sample weights and biases for Fully-Connected layers. A sampled Fully-Connected layer is referred as a Dense layer throughout the remaining sections of this thesis for convenience. To construct a neural network using SWIM, we need only four ingredients: inputs, ground truths, Dense layers, and arbitrary linear optimization layers. The non-linear activation layers such as ReLU or tanh are not explicitly needed, because the non-linear activation is already incorporated as part of the weight construction.

To illustrate the training scheme of SWIM, we consider the simple architecture as shown in Figure 2.1, except we optimize the output layer by minimizing a linear least square problem. The sampling procedure for the first two Fully-Connected layers are the same:

1. Given n inputs $x \in \mathbb{R}^{d_{in}}$ where d_{in} is the number of features, layer width d_{out}, n outputs $y \in \mathbb{R}^{d_{gt}}$ where d_{gt} is the ground truth dimension, randomly sample $\max(n, d_{out})$ pairs (x_1, x_2) from the inputs with a uniform distribution to construct a candidate pool $\mathcal{S}$. Two elements in a pair must be different, $x_1 \neq x_2$.

2. Derive the gradient $\frac{||y_1^{(i)} - y_2^{(i)}||_\infty}{\max(||x_1^{(i)} - x_2^{(i)}||_2, \epsilon)}$ — where ϵ is the lower bound constant for distances between pairs — for all pairs $(x_1^{(i)}, x_2^{(i)}) \in \mathcal{S}, i \in \{1, \ldots, \max(n, d_{out})\}$ and normalize into a probability distribution $\mathfrak{P}$.

3. Sample m pairs *with replacement* from $\mathcal{S}$ following the sampling distribution $\mathfrak{P}$ as a collection denoted as $\widetilde{\mathcal{S}}$.

4. Construct m weights $w^{(i)} = s_1 \frac{x_1^{(i)} - x_2^{(i)}}{||x_1^{(i)} - x_2^{(i)}||_2^2}$ — where $w^{(i)} \in \mathbb{R}^{d_{in}}, s_1 \in \mathbb{R}, (x_1^{(i)}, x_2^{(i)}) \in \widetilde{\mathcal{S}}, i \in \{1, 2, \ldots, m\}$ — and m biases $b^{(i)} = -\langle w^{(i)}, x_1^{(i)} \rangle - s_2$, where $b^{(i)} \in \mathbb{R}, s_2 \in \mathbb{R}$.

The choice of non-linear activation functions can be determined by choosing specific s_1 and s_2 as reported by Bolager et al. [2023]. For ReLU, $s_1 = 1$ and $s_2 = 0$; for tanh, $s_1 = 2s_2$ and $s_2 = \frac{ln(3)}{2}$. Step 1 can be considered the second stage of a subsampling process, following the initial stage of sampling the training data from the population. It is worth noting that here we specify the norms — $|| \cdot ||_\infty$ and $|| \cdot ||_2$, both are instances of the L^p-norms — to be consistent with the implementation while Bolager et al. [2023] adopt generic notations for L^p-norms.

Figure 2.3 illustrates the desired samples graphically. Instead of random sampling (first two items from left in the figure), SWIM samples the input pair directions which are located near large gradient (two items from right in the figure), as defined in Step 2 above.

This data-driven approach has been proven effective. Compared to iterative gradient optimization which contains multiple forward and backward pass (materialized by matrix multiplications), the computations of this sampling algorithm are much quicker and mostly linear in terms of time complexity, on top of one (instead of multiple) regular forward pass.

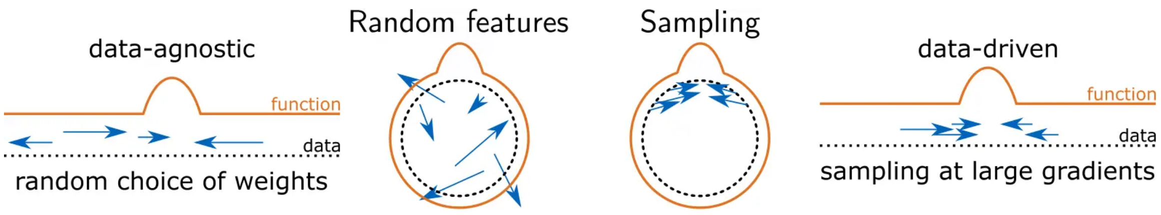


Figure 2.3: The intuition of SWIM sampling weights on large gradient compared to the data-agnostic approach. Taken from the literature [Bolager et al., 2023].

2.2.2 Optimization

For the input layer in Figure 2.1, SWIM constructs the weights and biases using the raw inputs, and this Dense layer (sampled input layer) then projects the raw inputs to establish the sampling frame for next layer, through the sampled weight matrix and sampled biases. The second layer samples from the outputs of the input layer, and projects the embeddings to a 4D latent space in similar fashion. These embeddings, even have the same dimensions as output space, can not provide accurate predictions because they are not projected with the weights optimized against the ground truths. The least square optimization at the output layer in Figure 2.1 is responsible to do the heavy lifting here.

This least square layer is not a mandatory option. We can replace it with any other linear optimizer as necessary, as long as the last layer can optimize weights to project the embeddings to the correct output space. This limits the application of SWIM to only supervised learning settings. Nonetheless, this simple setup works surprisingly well on various popular supervised learning tasks, and its universal approximation properties is also well proven by Bolager et al. [2023]. However, its efficacy on point clouds is untested. Abstractly, point clouds contain two levels: object level and point level. It is non-trivial to compute the distance and direction between two point clouds.

2.3 High Dimensional Nearest Neighbour Search

Unlike the direction, which is not well-defined for point clouds, the distance between two point clouds can be precisely measured as a metric between their respective sets of points. Among the various metrics available, Chamfer Distance and Hausdorff Distance are two widely used and well-defined measures for comparing two sets of points. Formally, given two point clouds, $P_1 = \{x_i \in \mathbb{R}^k\}_{i=1}^{n}$ and $P_2 = \{x_i \in \mathbb{R}^k\}_{i=1}^{m}$, where k is the number of features including the coordinates, n and m are the respective number of points, Chamfer distance and Hausdoff distance are defined as

$$\text{chamfer}(P_1, P_2) = \frac{1}{2} \overbrace{\frac{1}{n}\sum_{i=1}^{n}}^{\text{Mean Aggregation}} |x_i - \text{NN}(x_i, P_2)|, \tag{2.4}$$

$$\text{hausdorff}(P_1, P_2) = \frac{1}{2} \underbrace{\max_{x \in P_1}}_{\text{Max Aggregation}} |x_i - \text{NN}(x_i, P_2)|, \tag{2.5}$$

where $\text{NN}(x, P) = \text{argmin}_{x' \in P} ||x - x'||$ is a nearest neighbour function, which selects the nearest point from another point clouds in terms of distance. We understand these two metrics using a hierarchical view with two layers: point level and object (point cloud) level. To measure the distance at object level, it boils down to choosing an aggregation strategy for the distances computed at point level, where Chamfer distance chooses average, and Hausdoff distance chooses maximum. The distances at point level are trivial to compute, but there is no standard method to pair up points from two point clouds. In this context, both the Chamfer Distance and Hausdorff Distance employ a nearest neighbour function, serving to pair points from one point cloud with their closest counterparts in the other point cloud.

The nearest neighbor function is a fundamental concept in computational geometry and machine learning. It aims to find the closest point in a dataset to a given query point, based on a specified distance metric (commonly Euclidean distance). This function is essential in various applications, including classification, clustering, and, as mentioned in the context of point cloud comparisons, finding corresponding points between two sets of data. The efficiency of nearest neighbor search becomes crucial when dealing with large datasets or when real-time performance is required.

2.3.1 KDTree

One efficient data structure for implementing the nearest neighbor search is the KDTree (K-Dimensional Tree) [Friedman et al., 1977]. As illustrated in Figure 2.4, KDTree is a binary tree that recursively partitions the space along different dimensions, allowing for quick elimination of large portions of the search space. The left part of the figure shows a 2D space with points A through F, while the right part displays the corresponding tree structure. The algorithm works by building a tree where each non-leaf node represents a splitting hyperplane (alternating between x and y axes in this case), and the leaves contain the actual data points. During construction, the space is divided at each level, with points to the left of the splitting plane in the left subtree and points to the right in the right subtree.

The KDTree search process, as illustrated by the black cross in Figure 2.4, efficiently traverses the tree to identify potential nearest neighbors. The search initiates at the root (point A) and descends, making decisions based on which side of each splitting plane

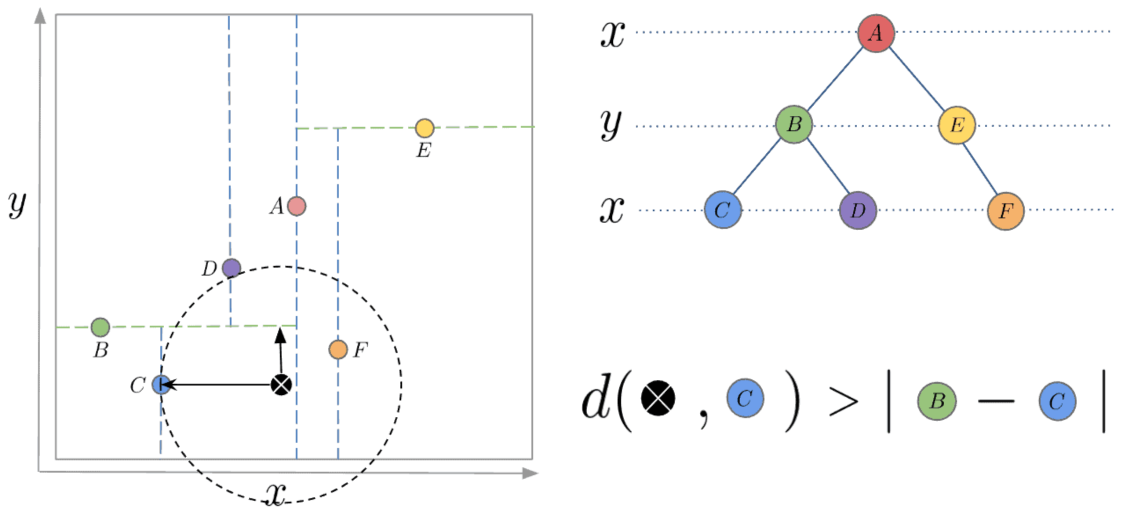


Figure 2.4: The intuition of KDTree construction and query in a 2-dimensional space. Taken from Baeldung [Hristov, 2023].

the query point falls. At each node encountered, the algorithm updates the current best distance if necessary. The dashed circle in the figure represents this current best distance. Upon reaching a leaf node (node C), the algorithm begins to unwind upwards. During this ascent, every non-leaf node undergoes a ”bounds-overlap-ball” test, as proposed by Friedman et al. [1977]. This test determines whether the splitting plane intersects with the sphere defined by the current best distance, indicating the potential need to explore the opposite branch. In our example, the hyperplane of node B intersects with the query point. However, searching the other branch of node B is unnecessary, as node D is farther away and cannot improve the current best distance. The algorithm then ascends to node A, where the ”bounds-overlap-ball” test yields a positive result, prompting exploration of A's right branch. Descending to node F updates the current best distance, after which the algorithm unwinds to node E. The search terminates when the ”bounds-overlap-ball” test at node E fails, indicating that no node on the opposite side of this plane can be closer than the current best distance.

The strength of the KDTree lies in its logarithmic average-case search time complexity. This efficiency makes it significantly faster than brute-force approaches, particularly for low-dimensional data. By strategically dividing the search space and employing the ”bounds-overlap-ball” test, the KDTree algorithm minimizes unnecessary comparisons, resulting in rapid nearest neighbor queries. However, KDTree's performance degrades in high dimensional spaces due to the curse of dimensionality, where the tree structure becomes less effective at pruning the search space. This is because in high dimensions, a random splitting plane is less likely to partitioning the space effectively. Figure 2.4 doesn't show this limitation directly, but imagine extending the concept to many more dimensions — the neat partitioning would become less clear-cut. This weakness often necessitates the use of dimension reduction techniques to maintain efficiency in high dimensional scenarios, paving the way for more advanced methods in complex data spaces.

2.3.2 Dimension Reduction

Dimension reduction is a technique in data analysis and machine learning, which transforms high dimensional data into a lower dimensional representation while preserving essential information. This process is a common approach to manage high dimensional data with a few benefits: it helps in visualizing complex data, reduces computational complexity, mitigates the curse of dimensionality, and can reveal underlying structures in the data.

Dimension reduction techniques can be broadly categorized into linear and non-linear methods. Linear methods, such as Principal Component Analysis, assume that the data lies on or near a linear subspace of the high dimensional space. These methods are computationally efficient and work well when the relationships in the data are indeed linear. However, they may fail to capture important non-linear structures. Non-linear methods can reveal more complex and curved manifolds in the data. These techniques are particularly useful when dealing with datasets that have intricate relationships in the high dimensional space. While non-linear methods can capture more complex structures, they are often more computationally intensive and may be prone to overfitting on small datasets.

Ultimately, the choice between linear and non-linear methods depends on the nature of the data: if the relationships are primarily linear or if computational efficiency is a priority, linear methods may suffice. For datasets with complex structures, non-linear methods are more appropriate, provided there's sufficient data to support their use. Here are some popular dimension reduction techniques along with their brief descriptions:

- **Principal Component Analysis (PCA)** [Pearson, 1901]: A linear technique that identifies the principal components (directions) of maximum variance in high dimensional data. It projects the data onto these components, effectively reducing dimensionality while preserving as much variability as possible.

- **Kernel PCA** [Schölkopf et al., 1997]: An extension of PCA that uses kernel methods to perform nonlinear dimensionality reduction. It first maps the data into a feature space with higher dimensions using an arbitrary kernel function, then applies PCA in this space, allowing it to capture nonlinear relationships in the original data.

- **Locally Linear Embedding (LLE)** [Roweis and Saul, 2000]: A non-linear technique that preserves local relationships between neighboring points. It reconstructs each data point as a weighted sum of its neighbors, then finds a lower dimensional embedding that preserves these local relationships. This is an effective procedure to learn the global structure of non-linear manifolds.

- **Isomap** [Tenenbaum et al., 2000]: A non-linear method that attempts to preserve geodesic distances between points on a manifold. It constructs a neighborhood graph, approximates the geodesic distances using this graph, and then uses multidimensional scaling to find a lower dimensional embedding that preserves these distances.
- **Diffusion Map** [Coifman and Lafon, 2006]: A non-linear technique based on defining a diffusion process on the data. It constructs a graph representing the data, defines a random walk on this graph, and uses the eigenvectors of the resulting operator to embed the data in a lower dimensional space, capturing the intrinsic geometry of the data.

We can see that some non-linear techniques already use a nearest neighbour function (usually also implemented using KDTree) in their procedure. Therefore with these techniques, ideally, the possible use case should be fitting a high dimensional space once and embed the fitted model as part of the data transformation pipeline for repeated queries of KDTree. Otherwise, the entire process (dimension reduction + KDTree) would require more the computational resources than a KDTree partitioning high dimensional space.

2.4 Problem Statement

The introduction thus far spans across 3 most important areas that constitute our work: PointNet, SWIM, and high dimensional nearest neighbour search. Before we begin our journey, it is important to define our problem and objectives clearly.

2.4.1 Problem Definition

In this thesis, we aim to train PointNet with low training time on a CPU and good performance. Specifically, we train PointNet using a state-of-the-art weight construction technique, namely Sampling Where It Matters (SWIM), instead of iterative gradient methods. We define low training time on a CPU numerically as 10% of the training time on a GTX1080 GPU reported by Qi et al. [2017a], which is 18 minutes to 36 minutes for classification using ModelNet40. We also define good performance as retaining more than 90% performance of PointNet.

To obtain good performance, we need to sample the point cloud pairs with steep gradients, in addition to capturing two geometric properties of point clouds: unordered and geometric transformation invariance, which can be re-defined as permutation invariance and rotational invariance on standardized point clouds. Note that a steep gradient refers to two inputs with close proximity and large output differences. Therefore, the final objectives for a sampled PointNet are:

- 18 minutes to 36 minutes training time using ModelNet40,
- Sample point cloud pairs with steep gradient,

- Permutation invariance,
- Rotational invariance.

2.4.2 Validation data

To evaluate the performance of a sampled PointNet during development, we use ModelNet10 [Wu et al., 2015]. ModelNet10 is a dataset of **O**bject **F**ile **F**ormat (OFF) files for 3D object classification, comprising 10 classes. The dataset contains 3,991 training samples and 908 test samples. Our data preprocessing involves several steps. First, we convert the mesh representations to point clouds using the open-source Python library `Trimesh` [Dawson-Haggerty et al.] to load the mesh files. We then sample *64 points* from each mesh, extracting points proportionally to the surface area of each face. The impact of the number of points sampled will be examined in later sections. Finally, the resulting point clouds are standardized to unit spheres before being input to the model.

For our experiments, we use ModelNet10 to test all hypotheses presented in this thesis. To prevent potential data leakage when later evaluating on ModelNet40 (a superset of ModelNet10), we create a custom split of the ModelNet10 training set. This split consists of 2,673 samples for training and 1,318 samples for validation. During development, we evaluate our models exclusively on this validation set. We assess the performance of the Sampled PointNet using three key metrics: accuracy, the one-versus-rest variant of the Area Under the Receiver Operating Characteristic curve (AUCROC), and computation time. Accuracy serves as the default metric for classification tasks, as established in Qi et al. [2017a]. The AUCROC proves particularly valuable during the development phase, ensuring that the model's performance exceeds random guessing.

To evaluate our first objective, we directly measure the computation time of sampled PointNet. For our second objective, we analyze the prediction metrics, which should reflect the quality of the sampled gradients. This approach allows us to assess whether the sampling technique maintains the model's performance while potentially reducing computational costs. To investigate permutation invariance, we examine the consistency of the Sampled PointNet's performance when presented with randomly permuted inputs. This examination occurs at both the object and point levels. A truly permutation-invariant model should maintain consistent performance regardless of the order in which points are presented. To assess rotational invariance, we compare the model's performance on randomly rotated point clouds against its performance on standard inputs. A rotational invariant model should demonstrate comparable performance regardless of the orientation of the input point cloud.

These evaluations collectively provide a comprehensive assessment of the Sampled PointNet's performance, efficiency, and ability to preserve important geometric properties inherent in the original PointNet architecture. By systematically examining these aspects, we can determine the effectiveness of our sampling approach in preserving the key characteristics of PointNet while potentially offering computational advantages.

In the final stages of our study, we will evaluate a more mature version of our sampled PointNet on ModelNet40. This approach allows us to thoroughly test our hypotheses on a smaller dataset before moving to the larger, more comprehensive ModelNet40 for final evaluation.

2.4.3 Related Work

To the best of our knowledge, there exists no prior work on PointNet's weight construction without iteratie gradient methods. It is possible to adapt the data-agnostic methods [Giryes et al., 2016, Rahimi and Recht, 2008] which have worse performance compared to SWIM as reported by Bolager et al. [2023].

Note that in the area of 3D machine learning, Lang et al. [2020] proposed SampleNet, which has a relevant name and based on PointNet. However, SampleNet emphasizes on differentiable point clouds downsampling *instead of sampling the weights and biases* for PointNet. There are more literature similar to this direction, that is, to improve the point cloud inputs — for example, fewer points with equal representation [Lang et al., 2020], or even distribution for points [Lebrat et al., 2021]) — for reduced computation costs with improved performance. It is important to clarify that these methods focus on the inputs instead of the models. In fact, given that SampleNet's architecture consists of two PointNets, it is indeed feasible to utilize our work to construct SampleNet.

3 Sampling Weights for PointNet

The journey started by analyzing the architecture of PointNet, and comparing to what SWIM can offer at the moment — a sampled Fully-Connected layer coupled with a non-linear ReLU/tanh activation layer named **Dense**, and a linear optimization layer, **Linear**, which is a least square optimizer. Here are the components of PointNet along with their compatibility with SWIM:

- **Shared Fully-Connected:** Not available;
- **ReLU:** Integrated in a Dense layer;
- **BatchNorm:** Not available;
- **Fully-Connected:** Integrated in a Dense layer or a Linear layer in case of last layer.

Although T-Net is one of the highlights proposed by Qi et al. [2017a], it can be constructed using these components, hence, we do not discuss it here. Out of the 4 mentioned components, Shared Fully-Connected layer and BatchNorm are not within the capabilities of SWIM. Since BatchNorm is mainly designed for iterative gradient method — to stabilize the gradient during training, we do not need this for a sampled PointNet. Therefore, the only challenge is to develop a Dense layer that shares weights. In the context of PointNet, weight-sharing refers to having the weight matrices in $\mathbb{R}^{d_{in} \times d_{out}}$ — which projects m-point point clouds with input dimensions d_{in} to the desired output dimensions d_{out} — to provide individual and identical treatments, instead of customizing for each point with weights matrices in $\mathbb{R}^{m \times d_{in} \times d_{out}}$.

To develop a sampled shared Fully-Connected layer, we first look at how we can build a Dense layer for point clouds. Recall that SWIM constructs weights and biases using directions between inputs and the norm (distances). However, the directions between two points clouds are not well defined, unlike the distances — for example, Chamfer distance (Equation 2.4) and Hausdorff distance (Equation 2.5). Nonetheless, we can temporarily see the direction computation as an abstraction in our discussion using a materialized function $\text{dir}(\cdot, \cdot)$, and portray the SWIM sampling algorithm for point cloud inputs. The weights for a Dense layer for point clouds with dimensions $(m \times d_{in} \times d_{out})$ can be constructed as followed:

1. Given the materialized point cloud direction function $\text{dir}(\cdot, \cdot) \in \mathbb{R}^{m \times d_{in}}$, n point clouds $X \in \mathbb{R}^{m \times d_{in}}$ where m is the number of points, d_{in} is the number of features, layer width d_{out}, n outputs $y \in \mathbb{R}^{d_{gt}}$ where d_{gt} is the ground truth dimensions, randomly

sample $\max(n, d_{out})$ pairs (X_1, X_2) from the inputs with a uniform distribution to construct a sampling frame $\mathcal{S}$. Two point clouds in a pair must be different, $X_1 \neq X_2$.

2. Derive the gradient $\frac{||y_1^{(i)} - y_2^{(i)}||_\infty}{\max(||\text{dir}(X_1^{(i)}, X_2^{(i)})||_F, \epsilon)}$ — where ϵ is the lower bound constant for distances between pairs — for all pairs $(X_1^{(i)}, X_2^{(i)}) \in \mathcal{S}, i \in \{1, \ldots, \max(n, d_{out})\}$ and normalize into a probability distribution $\mathfrak{P}$.

3. Sample m pairs *with replacement* from $\mathcal{S}$ following the sampling distribution $\mathfrak{P}$ as a collection denoted as $\widetilde{\mathcal{S}}$.

4. Construct d_{out} weights $w^{(i)} = s_1 \frac{\text{dir}(X_1^{(i)}, X_2^{(i)})}{||\text{dir}(X_1^{(i)}, X_2^{(i)})||_F^2}$ — where $w^{(i)} \in \mathbb{R}^{m \times d_{in}}$, $s_1 \in \mathbb{R}$, $(X_1^{(i)}, X_2^{(i)}) \in \widetilde{\mathcal{S}}$, $i \in \{1, 2, \ldots, d_{out}\}$ — and d_{out} biases $b^{(i)} = -\langle w^{(i)}, X_1^{(i)} \rangle - s_2$, where $b^{(i)} \in \mathbb{R}^m$, $s_2 \in \mathbb{R}$.

Here we slightly abuse the notation for s_1 and s_2 to assume that they are two constants which can be broadcasted to match the dimension of operands. For shared Dense layer, one direct approach is to treat the weights with dimension $(d_{in} \times d_{out})$ as an aggregation of m point-wise weights of a Dense layer:

5. Aggregate the weights W on point level to obtain shared weights W_{shared} — $W_{\text{shared}} = \text{aggpoint}(W)$, $W_{\text{shared}} \in \mathbb{R}^{d_{in} \times d_{out}}$, $W \in \mathbb{R}^{m \times d_{in} \times d_{out}}$ — and shared biases B_{shared} — $B_{\text{shared}} = \text{aggpoint}(B)$, $B_{\text{shared}} \in \mathbb{R}^{d_{out}}$, $B \in \mathbb{R}^{m \times d_{out}}$ — using an aggregation function $\text{aggpoint}(\cdot)$.

Chamfer distance and Hausdorff distance have demonstrated that, simple aggregation strategy such as mean or max, can represent the differences between two sets of points well enough. However, fitting the Dense layer followed by an aggregation operation might not work straightaway, because it does not guarantee the sampled point cloud directions are associated with steep gradients.

3.1 Steep Gradients & Permutation Invariance

Computationally, we represent a point cloud using a matrix in $\mathbb{R}^{m \times d_{in}}$. This representation commits to a certain permutation of the order of points, which in reality should not have an impact on the prediction outcome. Qi et al. [2017a] and Zaheer et al. [2017] resolve this by adding a sum operator or a max operator, which both are symmetry functions. The addition of symmetry operator can resolve the same issue for SWIM, however, the permutations of inputs on point level affect SWIM from another perspective. SWIM heavily depends directions with steep gradients, as shown in Figure 2.3. The steep gradient areas of point cloud pairs are approximated by the intersections of the pair, which we can consider as latent representations, as shown in Figure 3.1. The point cloud directions are computed using row-wise subtractions between two m-point point clouds X_1 and X_2: $\text{dir}(X_1, X_2) = [x_1^{(i)} - x_2^{(i)}]$,

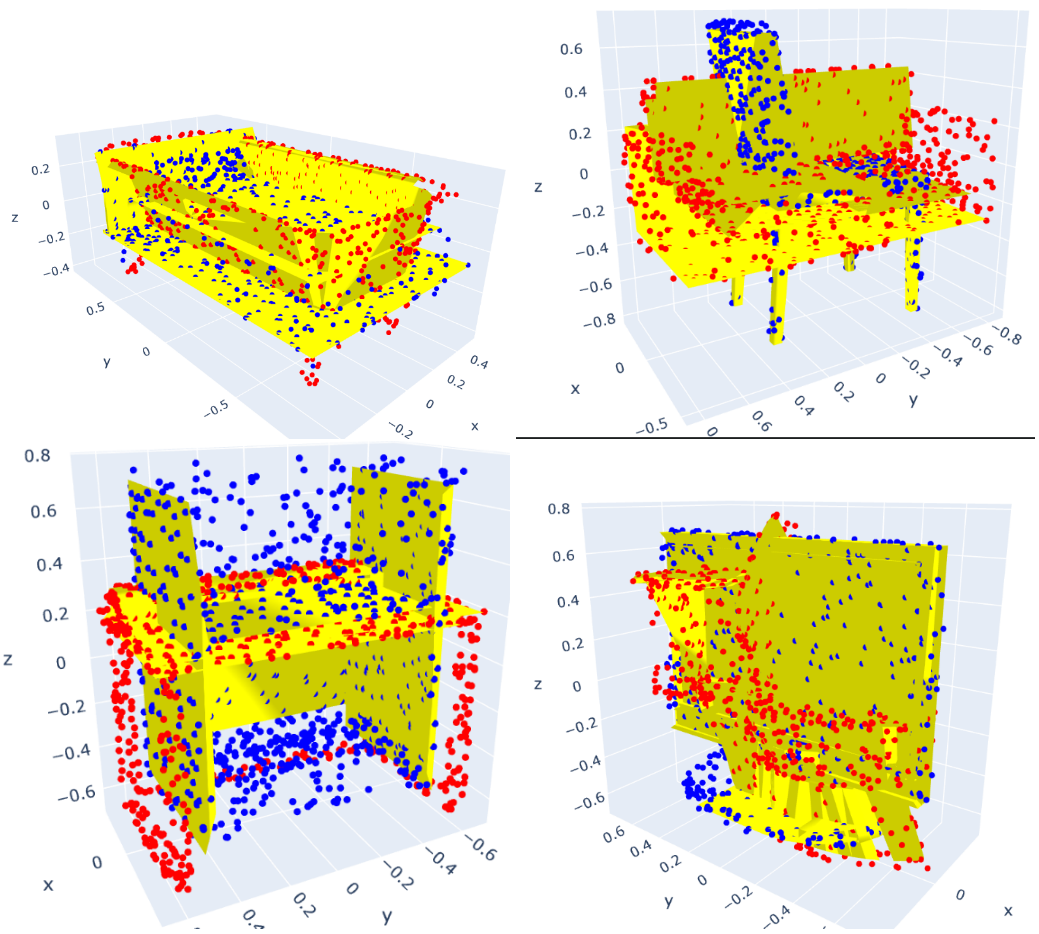


Figure 3.1: This figure shows the steep gradient areas (yellow) in 3D space for four pairs of 1024-point point clouds. The steep gradient areas are loosely approximated by the mesh faces of intersection of both objects. **Top Left:** Bed (blue) and Bathtub (red). **Top Right:** Chair (blue) and Sofa (red). **Bottom Left:** Night Stand (blue) and Table (red). **Bottom Right:** Monitor (blue) and Toilet (red).

where $x_1^{(i)} \in X_1, x_2^{(i)} \in X_2$, and $i \in [1, \ldots, m]$. In this case, the permutations of the points can determine the representation quality of point cloud directions. In other words, given two point clouds, the committed point permutations might contain many "good" point-wise directions, leading to a better representation after aggregation. Conversely, the unhelpful point-wise directions will result in under-representations of the point cloud directions. When the majority of the sampled point cloud directions are under-represented, SWIM cannot perform well. Thus, a solution to sample good point-wise directions is needed.

3.1.1 Sampling using KDTrees

The quality of a direction is determined by the associated gradient: $\frac{||y_1^{(i)}-y_2^{(i)}||_\infty}{\max(||x_1^{(i)}-x_2^{(i)}||_2,\epsilon)}$. Although we have no control over the numerator (which is already fixed as part of data), we can design a machinery to pick denominators with small values. Naturally, the shortest possible distance between two m-point point clouds (X_1 and X_2) will be derived from the shortest pairwise point distances:

$$\text{dir}(X_1, X_2) = [x_1^{(i)} - \text{NN}(x_1^{(i)}, X_2)], x_1 \in X_1, i \in [1, 2, \dots, m], \tag{3.1}$$

where $\text{NN}(x_1, X_2) = \text{argmin}_{x_2 \in X_2} ||x_1 - x_2||$ is a nearest neighbour function, which selects the nearest point of a given point from another point clouds in terms of distance. Finding nearest neighbours is a non-trivial task and KDTree is an efficient tool built for this purpose. Comparing to a brute force search with $\text{O}(n^2)$ run-time, KDTree on average has $\text{O}(kn\log n)$ run-time for construction and $\text{O}(\log n)$ for nearest neighbour search in a k dimensional space. The steps of using KDTrees to compute shortest point-wise directions for each pair of point clouds (X_1, X_2) from the sampling frame $\mathcal{S}$ in a Dense layer are specified below:

1. Given (X_1, X_2), construct a KDTree using X_1: KDTree(X_1).
2. Query KDTree(X_1) for each point in X_2 to obtain the shortest point-wise directions.

To compute shared weight, we choose the mean of the weights as the aggregation function. We name this enhanced Dense layer as DenseKD. The inference of a shared DenseKD layer is equivalent to the following equation:

$$H^{(l)} = \sigma(W^{(l)}_{\text{shared}} H^{(l-1)} + B^{(l)}_{\text{shared}}) \tag{3.2}$$

where l indicates l-th layer, H is the layer output, W_{shared} is the layer weights, B_{shared} is the layer biases and $\sigma(\cdot)$ is a non-linear activation function.

With DenseKD, we can now construct Sampled PointNet. We consider only the architecture of a reduced vanilla PointNet at the moment without T-Net and BatchNorm. Sampled PointNet uses DenseKDs as drop-in replacements for shared Fully-Connected layers, a max operation as MaxPool layer, Dense layers for regular Fully-Connected layers, and lastly a Linear layer wrapping the least square approximation as the linear classifier head. Note that least square approximation is usually deployed for regression task. However, to preserve originality, we follow SWIM to use it for classification. The architecture details are illustrated in form of a list of texts: [shared DenseKD (64), shared DenseKD (128), shared DenseKD (1024), MaxPool, Dense (512), Dense (256), Linear]. Next, we study whether the role of KDTree overlaps with the existing components of SWIM.

3.1.2 KDTree & Sampling Distribution

For Dense layer, the good input pairs are chosen with encouragements using the sampling distribution derived by normalizing the gradients. However, for DenseKD, KDTree can be seen as a strong enforcer to achieve the same purpose. In particular, under a classification setting with one-hot encoded output, the difference in output is either 1 for difference class, or 0 for same class, therefore, the numerator of the gradient does not contribute much. This setting reveals a surprising advantage of computing direction with nearest neighbours: it automatically chooses all points with close proximity, thus under a setting with limited and small output differences, such as classification, sampling points with any probability distribution can give comparative performance. If output labels are no longer needed under this particular setting, it is possible to deploy SWIM for unsupervised or indirect supervised task. However, it is important to notice that this discussion is only within the scope of implementation and there are much more to consider from the theoretical perspectives before applying SWIM on more diverse problem settings.

3.1.3 KDTree & Dimension Reduction

KDTree is not perfect as it can be cursed by the dimensionality of inputs. In the context of a Sampled PointNet, KDTree works well for input layer because the number of input dimensions is typically low, such as 3 or 6. For the projected latent space in Neural Network, the number of dimensions tend to be larger than 20, which is maximum number of dimension to exploit the efficiency of KDTree, as mentioned by `scipy` (Virtanen et al. [2020]) in the documentation for KDTree. A common approach to avoid high dimensional space is by reducing the dimension using dimension reduction techniques. The naive strategy is to first reduce the dimensions, then construct KDTree on the reduced dimension to find nearest neighbours for all points, and finally obtain the distances between the nearest neighbours in original dimension. The insertion of dimension reduction techniques would not complicate the sampling algorithm.

3.1.4 Results

Sampling using KDTrees

To check the effectiveness of DenseKD against point clouds, we compare the performance to a dummy sampled PointNet, which has the same architecture, except that the shared DenseKDs are replaced with shared Dense layers with same layer widths. For simplicity, we denote the former as DenseKD and the dummy as Dense.

Based on Table 3.1, we can notice that the differences in terms of predictive performance are stark. Despite the advantage in computation time, Dense offers little to no predictive power for point clouds. The 50% of AUCROC suggests that Dense is random guessing. Figure 3.2 shows the differences between the point cloud directions computed using random permutation and KDTree. We can notice that the directions computed by KDTree are "disciplined" and pointed towards the steep gradient areas, compared to the messy representation of the random permutation. The accuracy of DenseKD is behind the state-of-the-art by nearly 17 units according to the leaderboard published by Wu et al. [2015],

Methods	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
Dense	$\mathbf{10.29 \pm 2.44}$	$22.23 \pm 2.78\text{e}{-15}$	50.00 ± 0.00
DenseKD	16.40 ± 3.54	$\mathbf{81.25 \pm 8.32\text{e}{-1}}$	$\mathbf{97.17 \pm 3.00\text{e}{-1}}$

Table 3.1: This table illustrates the impact of permutations in computing point cloud directions in terms of time taken, accuracy, and AUC-ROC score. All method were repeated for 30 different seeds. The reported values are *mean±standard deviation* of 30 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **Dense** computes directions by committing to the given permutations of points. **DenseKD** ignores the given permutation by searching the nearest neighbour of each point using KDTree to compute directions.

and only 2 units behind the proposed model, 3DShapeNet [Wu et al., 2015], developed by the composer of the dataset. Overall, this is an encouraging sign considering the huge difference in runtime. Moreover, the leaderboard typically records the results which based on 1024- or 2048-point point clouds, while we are currently using 64-point density.

KDTree & Sampling Distribution

We continue to investigate the necessity of using KDTree together with the original sampling distribution. We experiment using three probability distributions: original, uniform, and length-squared. Original is the sampling distribution is defined by Bolager et al. [2023]; the uniform distribution acts as a non-informative baseline; and the length-squared distribution q — which is frequently used in randomized linear algebra to approximate a matrix — is defined by squared row vector norm over squared matrix norm:

$$q^{(l)}(X_1, X_2 | \{W_j, B_j\}_{j=1}^{(l-1)}) = \frac{||X_1 - X_2||_F^2}{||\mathcal{D}||_F^2}, \tag{3.3}$$

where l denotes a layer of neural network, W and B represents the weights and biases, and $||\cdot||_2$ and $||\cdot||_F$ represent the Euclidean norm and the Frobenius norm respectively. $X_1, X_2 \in \mathcal{X}$ are two point clouds in the input set $\mathcal{X}$, $X_2 - X_1 \in \mathbb{R}^{m \times k}$ is the k-dimensional directions from P_1 to P_2 and materialized as a row in the direction matrix $\mathcal{D}$. In contrast to the original sampling distribution, a length-squared distribution emphasizes the inputs with larger proximity. This is to study the lower bound of impact KDTree by prioritizing the relatively "bad" input pairs after the nearest neighbour search. The sampling algorithm is the same except for the probability distribution to sample input pairs.

From the results in Table 3.2, we can observe that the original sampling distribution might seem superior in terms of performance metrics. However, the discrepancy small enough to be entirely bounded by the standard deviations of the other distributions. Uniform distribution is slightly faster in terms of computation time due to the simplicity in materializing the

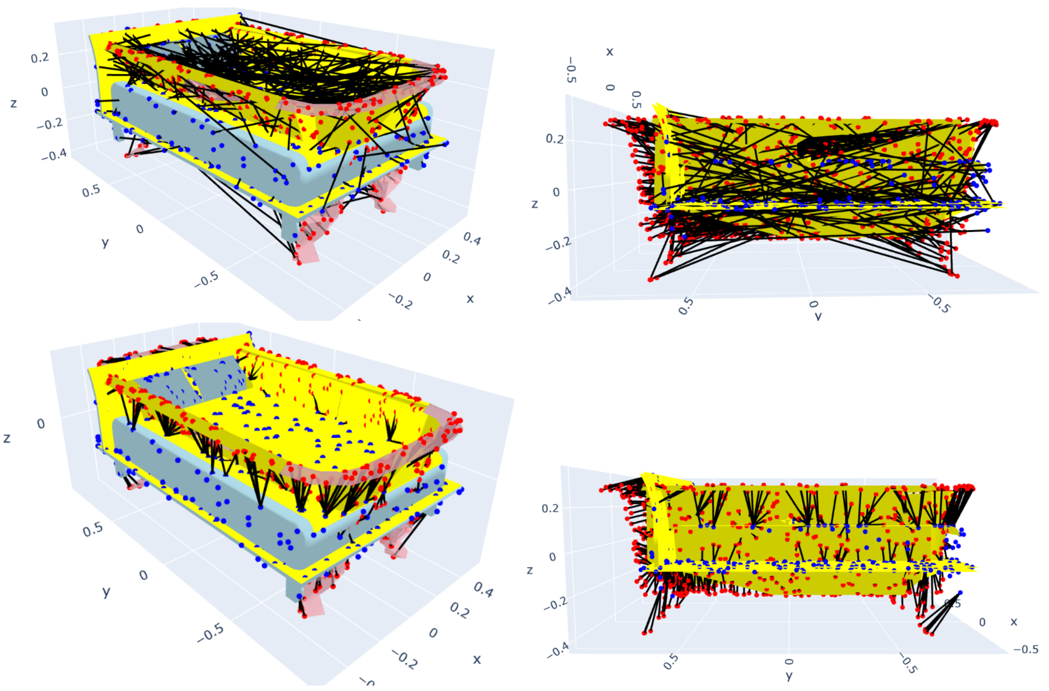


Figure 3.2: This figure shows the differences between the point cloud directions (black lines) between a bed (blue) and a bathtub (red), computed using random permutation and KDTree. To enhance the visualization effect, we use 1024 point per point cloud and 1024 sampled directions. **Top:** Random permutation. **Bottom:** KDTree. **Left:** Isometric view with underlay meshes. **Right:** Side view with gradient area (yellow) only.

probability distribution but also the advantage is also insignificant. The samples based on length-squared distribution in this context constitute an approximation of the distance matrix $\mathcal{D}$. It is naturally for the sampling process with length-squared distribution to result in worse performance on average, as intuitively it encourages input pairs with larger distance, hence contradicts with the theoretical foundation of SWIM. Nevertheless, the results are still comparable since the nearest neighbour function has already set a low upper bound for the distance between input pairs, although the upper bounds of the performance of length-square distribution in both prediction metrics fall short by a short margin to include the upper bounds of that of the original sampling distribution. In general, this experiment has demonstrated that the choice of sampling distribution does not have major impact on sampled PointNet, as KDTree has already done the heavy lifting.

Distribution	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
Original	16.40 ± 3.54	$\mathbf{81.25 \pm 8.32e{-1}}$	$\mathbf{97.17 \pm 3.00e{-1}}$
Uniform	$\mathbf{15.03 \pm 2.46}$	81.03 ± 1.14	$97.14 \pm 3.79e{-1}$
LS	16.17 ± 3.26	80.16 ± 1.12	$96.84 \pm 3.84e{-1}$

Table 3.2: This table illustrates that the choice of sampling distribution is insignificant with nearest neighbours' distance, under supervised classification setting. The experiment for each distribution were repeated for 30 different seeds. The reported values are *mean±standard deviation* of 30 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **Original** defines a probability distribution over the ratio of output finite differences and corresponding input distances. **Uniform** is a uniform distribution, with all points having same sampling probability. **LS** defines length-squared distribution with the sampling probability being the squared row vector norm over the squared matrix norm.

KDTree & Dimension Reduction

Concerning the efficacy of the typical remedies for the curse of dimensionality of KDTree with the use case of Sampled PointNet, we attempt to incorporate several techniques, including Principal Component Analysis (PCA), Kernel PCA, Locally Linear Embedding (LLE), ISOMAP, and diffusion map. The first four are the implementation of `scikit-learn` developed by Pedregosa et al. [2011], and diffusion map is from `datafold` implemented by Lehmberg et al. [2020]. Unfortunately, diffusion map does not work because of `ZeroDivisionError`, which cannot be resolved even by passing a numerical value for the parameter `value_zero_division` of `_symmetric_kernel_division` function through the kernel keyword argument under `datafold.pcfold`. This is because the kernel keyword argument is being fed to the class `DmapKernelFixed` during inference, while the function `_symmetric_kernel_division` is being called during initialization as normalized kernel. Nonetheless, other non-linear dimension reduction methods provided by `scikit-learn` work normally.

Despite the seemingly feasible intuition, the empirical computation time in Table 3.3 proves otherwise due to the runtime complexity as listed in Table 3.4, gathered from Pedregosa et al. [2011]. We can notice that the simplest technique, PCA, is already likely to take more time than KDTree except very few circumstances, for example, the number of points are much larger than the number of dimensions, which will be a rare occasion in our context. The remaining non-linear methods all have worse computational complexity than PCA. We assume the KernelPCA has an additional dominant term n^2 in its complexity for computing the pairwise entries of the kernel. The first and third term of the runtime complexity for Isomap and LLE are the same, which are the nearest neighbour search and partial eigenvalue decomposition. The second term differs as Isomap performs shortest-

Techniques	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
Vanilla	$\mathbf{16.40 \pm 3.54}$	$81.25 \pm 8.32\text{e}{-1}$	$97.17 \pm 3.00\text{e}{-1}$
PCA	21.73 ± 2.92	$81.19 \pm 7.38\text{e}{-1}$	$97.21 \pm 3.13\text{e}{-1}$
PCAVAR	21.30 ± 5.36	$\mathbf{81.29 \pm 8.10\text{e}{-1}}$	$97.21 \pm 3.11\text{e}{-1}$
KernelPCA	29.80 ± 4.73	$81.12 \pm 7.45\text{e}{-1}$	$97.23 \pm 2.95\text{e}{-1}$
Isomap	51.46 ± 3.21	$81.12 \pm 8.83\text{e}{-1}$	$\mathbf{97.24 \pm 3.28\text{e}{-1}}$
LLE	65.07 ± 4.25	79.81 ± 1.23	$96.66 \pm 4.83\text{e}{-1}$

Table 3.3: This table illustrates the impact of different dimension reduction techniques in terms of time taken, accuracy, and AUC-ROC score. All techniques were repeated for 30 different seeds. The reported values are *mean±standard deviation* of 30 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). For **Vanilla**, no dimension reduction technique was used. For **PCAVAR**, it retained the number of components which preserves 99.9% variance. All remaining techniques explicitly reduced the original dimension to 10 components. For **KernelPCA**, a `rbf` kernel was used with default parameters. For **Isomap** and **Locally Linear Embedding (LLE)**, all parameters are default except for `n_components`.

path graph search while LLE constructs weight matrices which is computationally heavy. On top of that, regardless of the chosen dimension reduction techniques, the model has to construct and query KDTree again in the reduced dimension, therefore this approach with the combination of dimension reduction and KDTree cannot optimize the overall runtime complexity. In terms of prediction, the differences are negligible. This is a sensible outcome because these techniques aim to preserve the important information in low dimensional space instead of creating new information.

3.1.5 Discussion

The experiment results in Section 3.1.4 have proved that our conceptual architecture for sampled PointNet works well empirically. In particular, the enhanced Dense layer, DenseKD, enables SWIM to sample directions with steep gradients through using, an efficient implementation of a nearest neighbour function. Since nearest neighbour is absolute regardless of the permutations, in this sense, nearest neighbour function enables the permutation invariance in relation to the representation quality of point cloud directions (which are the aggregations of the corresponding subsampled point-wise directions). The deployment of KDTree also reveals the fact that under certain scenarios such as classification, ground truths and the sampling distribution have only minor contribution for the performance, although the ground truths are still important to optimize the weights of a Linear layer. Lastly, we demonstrate that in the context of a Sampled PointNet, dimension reduction techniques mostly cannot improve the time complexity of KDTree due to their inherent expensive

Techniques	Time Complexity
KDTree	O($kn \log n$)
PCA	O($k^2 n + k^3$)
KernelPCA	O($n^2 + k^2 n + k^3$)
Isomap	O($klog(m) n \log(n) + n^2(m + \log(n) + dn^2$)
LLE	O($k \log(m) n \log(n) + knm^3 + dn^2$)

Table 3.4: The time complexity for each dimension reduction technique compared to KDTree. k is the number of dimensions, n is the number of training points, m is the number of neighbours and d is the number of reduced dimensions.

runtime and the one-time usage of KDTree for each input pair. Overall, we achieve two out of four objectives: permutation invariance (in relation to both the prediction consistency and gradients of input pairs) and sampling the directions with steep gradient. Despite the runtime of DenseKD is remarkably short, our training setting is still not comparable to the standard setting, which uses ModelNet40 with more training samples and 16- or 32-times more points per point cloud.

3.2 Rotational Invariance

The third objective of this thesis, rotational invariance, concerns the robustness of sampled PointNet against rigid body transformations. Although rotations are only a subset of rigid body transformations, it is one of the most common transformations of point clouds. Rotations in 3D Euclidean space about the origin is known as SO(3) (**S**pecial **O**rthogonal group), elaborated by Bronstein et al. [2021]. Most of the classification and regression tasks on point clouds are orientation-agnostic. Intuitively, if we were to classify a bus and an aeroplane, no matter how to rotate the corresponding point cloud, the model should be able to differentiate accurately. Formally, a function defined on inner product space with rotational invariance produces same output regardless of the inputs' orientations. However, finding nearest neighbours — an important operation in computing point cloud directions — is highly subjective to orientation, therefore vulnerable against rigid body transformations. As shown in Figure 3.3, the intersection areas in the standard orientation could be entirely different to that in the rotated orientations, therefore, this could lead to highly inconsistent representations in terms of point cloud directions.

It is important to clarify that, despite Bolager et al. [2023] pointed out SWIM is robust against rigid body transformations, it is only invariant with the entire dataset undergoes same set of transformations, instead of individual heterogeneous transformations for the data points. In this section, we focus on three approaches: T-Net, Data Augmentations, and Spherical Coordinates. T-Net is an unique solution proposed by Qi et al. [2017a]; data augmentations is a common technique in deep learning to improve the robustness of neural network; and spherical coordinates is our attempt to highlight the rotational information through inductive biases. Since Qi et al. [2017a] consider only the rotations along z-axis, which can be considered as a subgroup of SO(3) named SO(2), we follow suit.

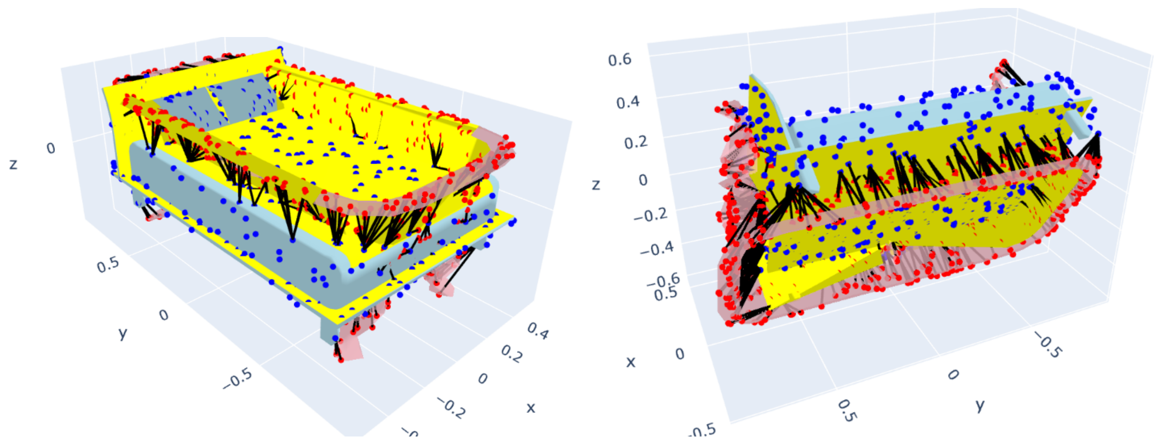


Figure 3.3: This figure illustrates the differences between the sampled directions (black lines) and the steep gradient area (yellow) between a bed (blue) and a bathtub (red). To enhance the visualization effect, we use 1024 points per point cloud and 1024 sampled directions. **Left:** Standard Orientation. **Right:** Randomly Rotated Orientations. The rotated objects are facing from each other in opposite directions.

3.2.1 T-Net: Joint Alignment Network

As one of the highlights for PointNet, Qi et al. [2017a] proposed a joint alignment network, namely T-Net, to enhance the robustness against rigid body transformation. T-Net is essentially a vanilla PointNet with shared Fully-Connected layers, MaxPool layer, and regular Fully-Connected layers. The ultimate PointNet architecture contains two T-Nets to align the input space and feature space, as shown in Figure 2.2. The first T-Net accepts inputs and predicts the respective 3×3 transformation matrices. Then, the product of inputs and their transformation matrices will be fed into the first hidden layer. The transformation matrices aim to align all inputs in the 3D space. The second T-Net accepts the embeddings after first hidden layer as inputs, and produces 64×64 transformation matrices. It aims to align the embeddings in the latent space. The second T-Net would destabilize optimization due to its high dimensionality, therefore, the author added a regularization term on the orthogonality of the feature transformation matrices as a soft constraint.

This elegant idea of leveraging indirect supervised learning to discover unknown transformation matrix stands on the shoulder of backpropagation. Theoretically, backpropagation can recover the transformation matrix using the loss of downstream task, despite not having the ground truth. This unique learning technique exposes the limitation of SWIM, which has superior training speed at the expense of discarding the feedback and guidance from the objective function. Nevertheless, we attempt to construct sampled T-Net that resembles the original. To learn the rotations of point clouds and produce the inverses, we formulate this problem under supervised-learning with synthetic data. Based on the fact that all data of ModelNet10 are precisely aligned, on each point cloud, we multiply a random rotation matrix generated using `provider.rotate_point_cloud` provided by Qi et al. [2017a] in the code repository, and save the transpose of the random rotation matrix as rotation labels. Note that since the rotation matrix is orthogonal, its transpose is equivalent to its inverse. It is much difficult to produce a high-dimensional rotation matrix for high dimensional latent space, therefore we do not consider resembling the second T-Net.

With the synthetic rotated point clouds and rotation labels, we can sample the ultimate PointNet architecture bar the second T-Net. The first T-Net is sampled using the rotated point clouds and the rotation labels, including the original orientation which the rotation label would be an identity matrix. The output of this T-Net is a 3×3 rotation matrix thus this problem is treated as a regression task. Following a similar inference procedure as PointNet, the inputs are multiplied with the predicted transformation matrix by the first T-Net, before entering the first hidden layer for sampling.

The training procedures of a sampled PointNet with embedded sampled T-Net are below:

1. For each point cloud $X \in \mathcal{X}$ from the train set $\mathcal{X}$, apply a rotation matrix $g \in$ SO(2) to obtain a rotated point cloud gX as shown in the top right of Figure 3.3.

2. The training inputs for T-Net is randomly rotated point clouds $\widetilde{\mathcal{X}} = [g^{(i)}X^{(i)}], X \in \mathcal{X}$ with corresponding inverse rotations as ground truths $\widetilde{\mathcal{G}} = [g^{-1(i)}]$, where $i \in [1, \dots, n]$.

3. Construct sampled T-Net using $\widetilde{\mathcal{X}}$ and $\widetilde{\mathcal{G}}$ under regression settings.

4. For each rotated point cloud $\widetilde{X} \in \widetilde{\mathcal{X}}$, apply the rotation matrix predicted by the sampled T-Net, $\widetilde{g} =$ T-Net$(\widetilde{X})$, to obtain an aligned point cloud $\widetilde{g}\widetilde{X}$.

5. Then, sample the weights and biases for a sampled PointNet using the set of aligned point clouds and the ground truths of downstream task.

Clearly, from the training procedures that this implementation of sampled T-Net has plenty of rooms for improvement. However, this should serve as as starting point to inspire more sampling counterparts which are hopefully equivalent to the advance tricks stemmed from iterative gradient methods.

3.2.2 Data Augmentation

Data augmentation is a widely used trick in deep learning. By augmenting the inputs with different transformations, for example, rotations, reflections, translations, the neural network is exposed to more samples, and hence has better prediction robustness. However, the author of PointNet Qi et al. [2017a] did not conduct a thorough ablation study to examine the individual contribution of data augmentation and T-Net towards rotational robustness in spite of their claim.The authors showed only 2% accuracy improvement —which is not significant— with the addition of two T-Nets *on top of* data augmentation. There are also some doubts on their GitHub[1] regarding the contributions of T-Net, which the authors never provide an answer. Therefore, in this thesis we conduct an individual experiment for data augmentation to identify its impact.

For iterative gradient descent, data augmentation only *virtually* increases the volume of train data. This is achievable because of the iterative nature and the gradient feedback. For each iteration, the inputs are randomly rotated in-place without expanding the data size, thus the memory demand remains the same. It is believed that the gradient feedback could automatically optimize the weights to recognize these augmentations. However, SWIM has neither the iterative mechanism nor the gradient feedback. Therefore, the implementation of data augmentation for a sampled net at this stage is crude. We expand the train set by including more rotated examples while preserving the same ground truths, at the expense of computational memory. Here we define the number of augmentations as the number of inputs copies rotated using *different* rotation matrices. For instance, 3-augmentations is equivalent to each point cloud turning into three rotated copies, and therefore 3 times the size of the standard train set (and the computational memory).

3.2.3 Spherical Coordinates

The methods of T-Net and data augmentation, derived from PointNet's original design, typically depend on iterative gradient methods. By intuition, the model should recognize the geometry regardless of the rigid body transformations. Supposed we have infinite data and infinite learning iterations, it is very likely for end-to-end supervised learning to ”learn” such patterns. Although current dataset is small to medium in size, we could scale up later by using ModelNet40 or more data augmentations. Regarding learning iteration, it is now a double-edged blade for SWIM. The advantage of SWIM builds the expense of cutting off backpropagation and learning iterations. It is essentially one-shot learning. Therefore, instead of hoping the model to discover this hidden piece of information naturally, we should instead instill the information explicitly.

The original input features are the spatial coordinates (x, y, z) in Euclidean space. Here we have two approaches to inform the model: firstly, we consider computing the directions with angular information, however, we do not see any suitable point of reference to compute the angular directions between the points in one point cloud and the respective nearest neighbours from another point cloud; secondly, we consider adding the angular information as input features. This is possible via spherical coordinates (r, θ, ϕ), where r is the radial distance between a point and the origin, θ is the polar angle between z-axis and r, and ϕ is the azimuthal angle between the orthogonal projection of r onto the xy-plane and either x- or y-axis.

[1]https://github.com/charlesq34/pointnet

Recall that we only consider SO(2). With SO(2) tranformations, among the input features $(x, y, z, r, \theta, \phi)$, we expect two rotated points which are close to each other in original orientation, will have same values for (z, r, θ), and likely to have only moderate changes in ϕ, thus acting as the counterweight to balance the drastic changes brought by rotations in x and y. However, this approach is not suitable for high dimensions due to computational intractability, thus, is only applicable to input layer similar to sampled T-Net. Nevertheless, we have also conceptually extended this idea and shallowly explored the representation in the hyperbolic space, which is a popular approach in point cloud learning [Lin et al., 2023, Montanaro et al., 2022].

The hyperbolic space comes with different representations and is deemed to be more efficient in terms of capturing hierarchical information compared to Euclidean space, as reported by Sala et al. [2018]. Onghena et al. [2023] proposes a rotation-invariant deep learning method for point cloud segmentation —which is another popular task for point cloud learning — with hyperbolic embeddings. Despite the promising outlook, we decide to leave this direction as future work. This is because most existing work leverage backpropagation to discover hyperbolic embeddings through indirect supervision of loss function, instead of manually converting the euclidean coordinates into other coordinate systems in the hyperbolic space.

3.2.4 Results

To evaluate the rotational robustness of the above approaches: T-Net, data augmentation, Spherical coordinates, we use a sampled PointNet with randomly rotated inputs as baseline. For the rotated inputs without augmentation, it means that the point clouds in train set are randomly rotated, and the size of train set remains the same ($n = 1$). The data augmentation approach is implemented as a sampled PointNet with 3-augmented inputs (hence named Vanilla with $n = 3$ in Table 3.5) by replacing each train point cloud in standard orientation to three randomly rotated copies. Instead of evaluating on a single random orientation, Qi et al. [2017a] recommended to adopt the majority vote of 12 fixed input orientations, evenly spaced from 0 to 2π. Nevertheless, we experiment both with the baseline and investigate the impact.

Table 3.5 shows the influence of two evaluation methods and the effectiveness of three approaches —T-Net, Data Augmentation, and Spherical coordinates — against rotated inputs. We start by comparing the evaluation methods.

Vanilla* evaluates the test set with random individual rotations while Vanilla with $n = 1$ evaluates the augmented test set with 12 fixed rotation angles, as proposed by Qi et al. [2017a]. The proposed evaluation method slightly improves all three metrics because 12-augmented test set with fixed angles is in spirit similar to Monte Carlo or ensemble methods, that is, giving the model more opportunities to predict correct outcomes in the context of classification. However, both prediction results represent significant drops compared to those using standard inputs. This indicates that sampled PointNet is not rotational invariance. In terms of processing time, the difference between Vanilla* and the remaining

Approaches	n	Time Taken (s) ⇓	Accuracy (%) ⇑	AUCROC (%) ⇑
Vanilla*		15.19 ± 1.10	62.69 ± 1.65	87.26 ± 1.07
Vanilla	1	38.08 ± 3.59	66.48 ± 1.72	$91.45 \pm 4.82\text{e}{-1}$
T-Net		84.62 ± 11.39	57.98 ± 1.84	$85.08 \pm 8.36\text{e}{-1}$
Spherical		$\mathbf{30.60 \pm 5.16}$	$53.35 \pm 7.58\text{e}{-1}$	$88.93 \pm 4.63\text{e}{-1}$
Vanilla	3	36.33 ± 4.53	$\mathbf{68.36 \pm 3.65\text{e}{-1}}$	$93.13 \pm 9.34\text{e}{-2}$

Table 3.5: This table illustrates the impact of different approaches in terms of time taken, accuracy, and AUCROC score. All techniques were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **n** is the number of rotated augmentations per point cloud. **Vanilla*** is the vanilla PointNet with a single evaluation for the randomly rotated test set. For the remaining, the test set is rotated 12 times with fixed angles to obtain the majority predicted labels and mean predicted class probabilities. **Vanilla** is the sample PointNet. **T-Net** is sampled PointNet with an embedded sampled T-Net. **Spherical** is the vanilla sampled PointNet with spherical coordinates as additional input features on top of the Euclidean coordinates.

methods are due to the externally affected processing power at the time of execution. For the other approach, the computation time belongs to the same magnitude expect for T-Net, which is approximately doubled of the other approaches because it is a sampled PointNet with regression task embedded in another sampled PointNet.

The remaining approaches — T-Net, Spherical Coordinates and Vanilla ($n = 3$) — are also evaluated with 12 fixed rotation angles. Based on Table 3.5, while the vanilla approach with 3 augmentation is superior with similar computation time, it is obvious that the contributions of T-Net and Spherical coordinates worsen the accuracy and AUCROC when compared to Vanilla (not Vanilla* because of the differences in evaluation methods). For T-Net, this suggests that the predicted rotation matrix fails to align all point cloud in the same orientation, and the inaccuracies turn noises to strengthen the training difficulty. However, it is worth to note that this is a learnable approach thus can be improved with data. Also, sampled T-Net is not an exact replica of the original T-Net as sampled T-Net does not consider the joint optimization of rotation matrices and the downstream tasks unlike the original. Therefore, a future work on sampled neural network for joint optimization or indirect supervision is much needed to replicate the original PointNet architecture perfectly.

Recall that Spherical coordinates incorporate inductive biases by converting spatial coordinates to angular notation, which is a little more invariant against rotations compared to spatial coordinates. Comparing both accuracy and AUCROC between T-Net and Spherical coordinates might cause some confusions because the addition of Spherical coordinates result in worse accuracy at the same time better AUCROC compared to that of T-Net. To understand the reasons behind, we refer to the classification report in Table 3.6 for both approaches. From the table, we can observe that the class distribution in test set is not balanced. Kubat et al. [1997] indicates that accuracy as a metric could be deceitful with imbalanced dataset, namely the accuracy paradox. This phenomena occurs when a simple model ”predicts” only the major classes without any classifying power, yet on paper its accuracy would be decent. Despite a sampled PointNet with an embedded sampled T-Net is by no means a simple model, it suffers from the same problem.

Classes	Precision⇑		Recall⇑		F1-Score⇑		Support
	T-Net	Spherical	T-Net	Spherical	T-Net	Spherical	
0	0.55	0.43	0.62	0.55	0.58	0.48	138
1	0.00	1.00	0.00	0.08	0.00	0.15	**38**
2	0.21	0.35	0.06	0.40	0.10	0.38	**62**
3	0.83	0.63	0.90	0.99	0.86	0.77	299
4	0.28	0.71	0.19	0.22	0.22	0.33	**69**
5	0.53	0.46	0.43	0.12	0.47	0.19	110
6	0.48	0.47	0.77	0.79	0.59	0.59	219
7	0.54	0.67	0.10	0.03	0.17	0.05	**70**
8	0.44	0.60	0.41	0.15	0.43	0.24	121
9	0.56	0.68	0.59	0.46	0.57	0.55	154
Macro Avg	0.44	0.60	0.41	0.38	0.40	0.37	1381
Weight Avg	0.54	0.57	0.58	0.54	0.54	0.48	1381

Table 3.6: This table shows the classification reports for T-Net and Spherical coordinates in terms of Precision, Recall, and F1-score. **Classes** are the corresponding enumerated classes in ModelNet10. **Macro Avg** stands for macro average, which takes a metric average over all classes with uniform weight. **Weight Avg** measures a metric average over all classes by taking class sizes into account. **Support** is the number of data points provided. The bold face supports are the classes with relatively low counts.

In general, T-Net has better performance with more support, and worse performance with low support (the bolded entries in Table 3.6). In particular, both vanilla PointNet (not shown here) and T-Net are likely to omit class 1 in their predictions. This indicates that T-Net has less power to distinguish different classes compared to that of Spherical coordinates. However, we notice that Spherical coordinates tend to have worse recall, known as sensitivity, with moderate class size in spite of its relatively good precision. This might reflect that although Spherical coordinates induces additional geometrical features that raises the floor of its classifying power, that is, the ability to recognize classes with small number of data points, at the same time the inductive biases also hinder its ability to derive high level features for better multi-class classification. This problem could be rectified in the future with features that provides more geometrical information. Nevertheless, it is worth remembering that this explains only the slight discrepancy in terms of accuracy and AUCROC for T-Net and Spherical coordinates. The overall performance still has plenty room for improvement.

3.2.5 Discussion

The detour of rotational invariance unfortunately ends with negative results —- the performance on classifying the rotated inputs is not on par with the performance using inputs with the standard orientation. We implemented two approaches — data augmentation and T-Net — suggested by Qi et al. [2017a], and an additional effort with Spherical coordinates as inductive biases. After all attempts, we find out that there are many doubts [Li et al., 2021, Zhao et al., 2022] whether PointNet is a rotational invariant model as Qi et al. [2017a] claimed to be, which a lot of empirical evidences suggest otherwise. Additionally, in the sequel of PointNet, namely PointNet++ [Qi et al., 2017b], the author excludes T-Net from the architecture, which affirms our observation that T-Net has limited contributions at the expense of training cost with doubled parameters. It is also encouraging to see data augmentation in isolation has an positive impact on rotational invariance.

Nevertheless, our effort exposes a fundamental limitations of SWIM compared to iterative gradient descent for further improvements. Firstly, joint optimization with indirect supervised learning is a popular technique based on gradient descent and backpropagation. However, SWIM gains its massive training time advantage by getting rid of backpropagation. In other words, the only feedback from target variables — which guides weight optimizations — comes from the last least square layer, which requires ground truth. Therefore, it is very tricky to optimize a sub-model such as T-Net without corresponding output or to recover a complex representation such as hyperbolic embedding.

This limitation by no means signifies the end of sampled neural network; in contrast, it encourages more out-of-the-box ideas rather than blindly mimic the mechanisms which based on gradient descent and backpropagation. Since it is difficult for sampled network to learn the implicit rotational information, the way forward is to incorporate the information explicitly, similar the failed attempt with Spherical coordinate. Other than incorporating the information as input features which still have plenty room of improvements, an interesting direction would be developing rotation-invariant weight construction machinery. Li et al. [2021] proposes a PCA-based training scheme using gradient descent, which utilizes the rotational *equivariant* properties of PCA and leverage the geometrical symmetry to reduce possible numbers of rotations down to a feasible counts for learning. This idea develops on the fact that PCA is a special form of rotation guided by the variance of principal axes. In previous section, we also discussed the inclusion of PCA in SWIM which may have partially paved the way to develop a sampled version of this idea.

Despite the extensive experiments thus far, there is an untested hypothesis that 64 points per point cloud could be too sparse to encapsulate sufficient information. Also, the data augmentation approach shines a glimmer of hope that the performance could be further improved by stacking more rotated samples. Therefore, in next section, we explore the scalability of sampled PointNet in terms of point cloud density (number of points per point cloud) and data augmentations with rotations.

3.3 Scale-up Density and Augmentations

Scaling up datasets has proven to be a powerful approach for improving performance across various domains. This is especially evident with the emergence of foundation models. In the context of 3D point cloud processing, increasing the density of point clouds and applying more extensive data augmentations can potentially lead to enhanced model capabilities. By exposing the network to richer, more detailed representations of 3D objects and a wider variety of viewpoints, we aim to improve the model's ability to capture fine-grained geometric features and achieve better generalization. This scaling strategy may result in more robust and accurate predictions. Furthermore, investigating the scalability of sampled PointNet allows us to better understand the trade-offs between computational resources, model complexity, and performance gains, which is crucial for deploying these models in real-world applications with varying computational constraints. To preserve originality of the architecture proposed by Qi et al. [2017a], we do not consider scaling the model parameters here.

3.3.1 Point Cloud Density

As shown in top left of Figure 3.4, 64-point point clouds are hardly recognizable with human vision if the underlaying meshes are not visualized. The density of 512 points improves the overall representation, but 1024-point on the bottom left can establish a more comprehensive presence. Despite point clouds with 2048 points could be more favourable by human, Qi et al. [2017a] shows in their Appendix F that, the model performance is proportional to point cloud density, but quickly saturates when the density approaches 1024. This result suggests an intriguing possibility: a neural network might value sparseness more highly than humans do. We are interested to see whether this phenomenon can be replicated with sample PointNet.

3.3.2 Number of Data Augmentations

In Section 3.2.2, we discussed the benefits of using data augmentation to improve rotational robustness. The empirical results in Table 3.5 not only support our hypothesis that data augmentation has more contributions than T-Net for rotation invariance, they also motivate further investigate to explore the impact of data augmentation at a larger scale. Moreover, data augmentation is not orthogonal to T-Net and Spherical coordinates (introduced in Section 3.2.1 and Section 3.2.3 respectively). It would be interesting to see if there is any positive effect by compounding data augmentation on T-Net and Spherical coordinates to edge towards rotational invariance.

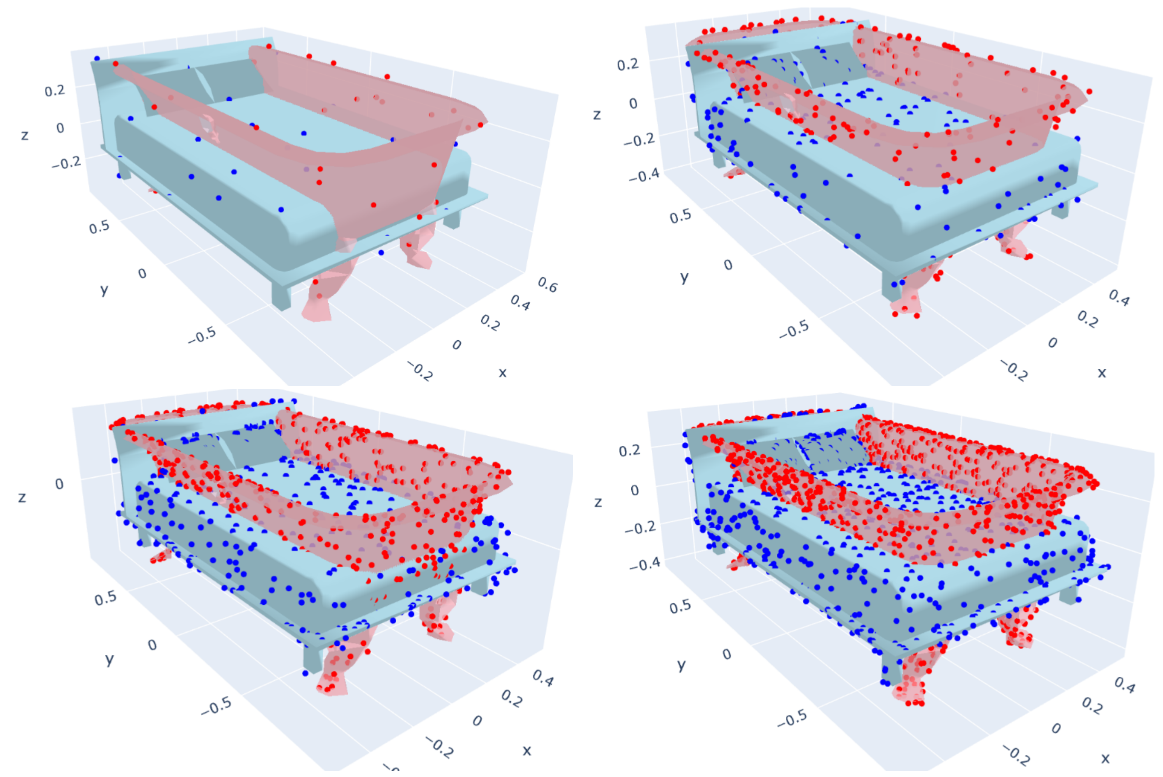


Figure 3.4: An illustration of point cloud pairs in varying densities. The mesh surfaces are underlayed for better visualization. The blue object is a bed, while the red object is a bathtub. **Top Left:** 64 points. **Top Right:** 512 points. **Bottom Left:** 1024 points. **Bottom Right:** 2048 points.

3.3.3 Coupling Effects

There is another dimension we could scale-up for rotational robustness: point cloud density. From human perspective, diverse and well-represented samples will naturally improve the prediction performance. However, this might not be the case for a machine learning model. We are looking for a coupling effect by increasing both the number of points per point cloud and the number of augmentations simultaneously. In the best-case scenario, this could be the key to achieving rotational invariance by simply providing many good examples.

3.3.4 Results

Point Cloud Density

The experiment setup is straightforward with 4 commonly chosen densities: 64, 512, 1024, 2048. Our experiments show that increasing point cloud density from 64 points to 2048

Nr. Points.	Time Taken (s) ⇓	Accuracy (%) ⇑	AUCROC (%) ⇑
64	$\mathbf{39.14 \pm 2.60}$	$81.00 \pm 4.28\text{e}{-1}$	$96.56 \pm 2.15\text{e}{-1}$
512	402.81 ± 70.38	$84.55 \pm 6.61\text{e}{-1}$	$97.12 \pm 2.57\text{e}{-1}$
1024	937.64 ± 149.77	$\mathbf{85.19 \pm 2.00\text{e}{-1}}$	$97.47 \pm 2.17\text{e}{-1}$
2048	1155.06 ± 216.46	$85.13 \pm 8.28\text{e}{-1}$	$\mathbf{97.56 \pm 4.36\text{e}{-1}}$

Table 3.7: This table illustrates that scaling up the density of point cloud can improve performance. The experiment for each distribution were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **Nr. Points** represents the number of points in a point cloud.

points per point cloud can improve performance. However, the improvement becomes insignificant or even slightly decreases from 1024 points to 2048 points. Although Qi et al. [2017a] has presented similar insights, we can offer a different explanation from SWIM's perspective. As we can notice in Figure 3.4, point cloud with 1024 points is representative enough to contain most of the information, such as the peripheral contours and interior details compared to 64 points. 2048 points enhances the details, at the same time, these enhancements can also be regarded as redundant information under the setting of classification, since the task emphasizes global shape recognition rather than intricate structure. Therefore, in our context, it is likely that the increased proportion of redundant information could overshadow the important information or even out during weight aggregation, resulting in non-increasing functional information.

Number of Data Augmentations

The experiment setup is trivial. Similar to Section 3.2.4, we measure the robustness using the aggregated predictions from rotated inputs with 12 fixed angles (12 angles per point cloud) as proposed by Qi et al. [2017a]. We randomly rotate the inputs to obtain {3, 5, 10, 20}-augmentations using SO(2) rotations along the z-axis as the rotated, augmented train sets. In Table 3.8, we first observe that the time taken for training the sample PointNets is proportional to the input sizes: the more the number of augmentations, the more the data points, hence the longer the computation time. To discover the empirical trends of accuracy and AUCROC, we turn our attention to Figure 3.5. In general, increasing the number of augmentation has positive impact for all approaches on all metrics compared to no augmentation at all, that is, the number of augmentation is 1. In spite of the improvements, the additions of T-Net and Spherical coordinates continue to deteriorate the rotational robustness.

For pure data augmentation without additional techniques, we could observe the accuracy peaked at 3-augmentations while its AUCROC continue to rise with more augmenta-

Approaches	n	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
Vanilla	3	$\mathbf{36.33 \pm 4.53}$	$\mathbf{68.36 \pm 3.65e{-1}}$	$93.13 \pm 9.34e{-2}$
T-Net		73.13 ± 6.34	60.50 ± 1.78	88.11 ± 1.11
Spherical		39.02 ± 6.21	$55.74 \pm 4.09e{-1}$	$90.32 \pm 2.82e{-1}$
Vanilla	5	51.78 ± 3.16	$66.90 \pm 6.81e{-1}$	$93.35 \pm 1.68e{-1}$
T-Net		110.50 ± 5.86	$61.81 \pm 4.57{-1}$	$89.38 \pm 3.74e{-1}$
Spherical		49.53 ± 3.73	$55.01 \pm 6.56e{-1}$	$90.61 \pm 1.41e{-1}$
Vanilla	10	112.56 ± 17.71	$67.06 \pm 5.33e{-1}$	$\mathbf{93.42 \pm 2.03e{-1}}$
T-Net		208.37 ± 14.55	62.26 ± 1.56	$90.17 \pm 4.11e{-1}$
Spherical		94.84 ± 4.26	$55.43 \pm 9.21e{-1}$	$90.44 \pm 3.63e{-1}$
Vanilla	20	194.31 ± 6.10	$66.89 \pm 7.22e{-1}$	$93.39 \pm 2.10e{-1}$
T-Net		396.913 ± 23.81	$61.99 \pm 9.06e{-1}$	$90.14 \pm 5.28e{-1}$
Spherical		224.63 ± 37.65	$57.22 \pm 7.61e{-1}$	$91.29 \pm 1.52e{-1}$

Table 3.8: This table illustrates the impact of different approaches with {3, 5, 10, 20}-augmentations in terms of time taken, accuracy, and AUCROC score. All techniques were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **n** is the number of rotated augmentations per point cloud. **Vanilla** is sampled PointNet. **T-Net** is sampled PointNet with an embedded sampled T-Net. **Spherical** is the vanilla PointNet with spherical coordinates as additional input features on top of the Euclidean coordinates.

tions until 10-augmentations. It could be that the model is trying to overcome the accuracy paradox — as discussed previously in Section 3.2.4 — with increased augmentation. The increasing AUCROC indicates that the model can distinguish the classes better rather than blindly optimizes accuracy. Between 3- and 10-augmentations, we can notice that Spherical coordinates experiences similar struggle by prioritizing accuracy or AUCROC alternatively, as demonstrated by the simultaneous improved accuracy and reduced AUCROC or vice versa.

Among the approaches, T-Net has the largest growth — 4.28 units for accuracy and 5.09 unit for AUCROC — as the number of augmentations increases tenfold. If we isolate T-Net from the PointNet, it is intuitive that more rotated samples improve the performance of T-Net, therefore the orientations of point clouds are more aligned. Looking at both the T-Net and the PointNet together, we can suggest that an explanation from the scaling law [Hestness et al., 2017] that the number of parameters of T-Net is the double of that of the original PointNet, therefore it has simply more capabilities to learn. Although it is theoretically possible that given sufficient number of augmentation, the T-Net approach could have better performance than the vanilla, however, the experiment with 20 augmentations shows

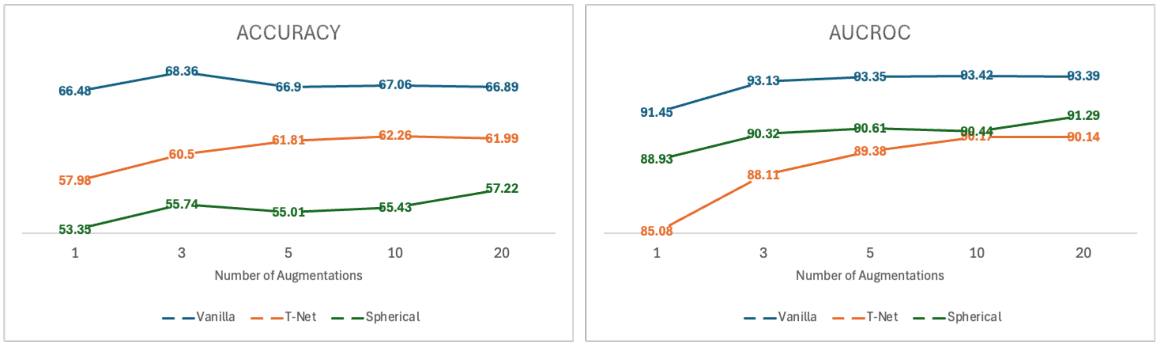


Figure 3.5: The reported results are the average over 5 different seeds. **Left:** The accuracy of three approaches as the number of augmentations increases. **Right:** The AUCROC of three approaches as the number of augmentations increases.

worse performance in all metrics on top of long computation time. This indicates that scaling up in terms of number of augmentation might not result in an emergent properties.

Emergent properties is one of the reasons behind the popularity of large language models, which were trained using gradient descent approach. Judging from our experiment results, we argue that it is difficult for a sampled net to achieve such effect. The underlying reason could be attributed to the fundamental difference between sampled network and gradient descent, where the formal can only *assign* weights on a discrete landscape while the latter *updates* weight in continuous domain. In future, the optimization landscape might converge, as weight quantization for LLM is under active development at the time of writing. Nonetheless, it is not feasible to experiment with large number of augmentation — for example, 100 or 1000 times augmentations — without a mini-batch approach, since it is computationally intractable with the humongous computational memory demand. On a side note, we can also observe that small number of augmentations does not improve the precision of predictions as reflected by the fluctuations in standard deviations as shown in Table 3.8. Overall, we can conclude that scaling up with more number of augmentations might not worth the computation resources given the marginal improvements and fundamental deficiency in fine-tuning.

Coupling Effects

By scaling up both aspects separately, performance improves but only up to a certain point, akin to a bottleneck, meaning that growth is not infinite. In view of this, running experiments for all of the cross products of the number of data augmentations and the number of points per point cloud is not necessary. We therefore selected 4 new configurations to investigate our hypotheses: (3, 512), (3,1024), (5,512), (10,512). The first elements in the tuples are the number of data augmentation, and the second elements are the point cloud density. We specifically choose these configurations to observe two scaling trends without spending too much computation power.

Using (3,64) as a control group, we first study whether the improvement contributed by point cloud density can persist with SO(2) domain with small influence from data augmentation by fixing the number of augmentations to 3 while scaling the point cloud density from 512 to 1024. For the second trend, we hold the number of points at 512 and increasing the number of rotated copies from 3 to 10. The chosen configurations should be indicative for both cases. If there are coupling effects, the growth rate will be different from what we observed in Table 3.7 and Table 3.8.

Approaches	Aug.	Pts.	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
Vanilla	3	64	36.33 ± 4.53	68.36 ± 3.65e−1	93.13 ± 9.34e−2
T-Net			112.13 ± 6.34	60.50 ± 1.78	88.11 ± 1.11
Spherical			39.02 ± 6.21	55.74 ± 4.09e−1	90.32 ± 2.82e−1
Vanilla	3	512	540.27 ± 32.83	70.61 ± 1.88	94.39 ± 4.44*z*e−1
T-Net			1025.44 ± 71.39	67.85 ± 1.58	91.38 ± 6.49e−1
Spherical			457.37 ± 18.44	64.72 ± 8.85e−1	92.53 ± 3.54e−1
Vanilla	3	1024	1484.00 ± 74.75	70.23 ± 2.35	94.45 ± 5.37e−1
T-Net			3001.79 ± 154.36	67.88 ± 1.37	90.78 ± 4.69e−1
Spherical			1345.08 ± 76.82	64.86 ± 1.04	91.78 ± 1.78e−1
Vanilla	5	512	886.88 ± 31.17	**71.43 ± 1.33**	94.50 ± 3.15e−1
T-Net			1640.05 ± 93.13	67.42 ± 1.10	91.22 ± 8.88e−1
Spherical			838.80 ± 33.98	64.83 ± 7.09e−1	92.11 ± 2.01e−1
Vanilla	10	512	2167.77 ± 102.78	70.79 ± 1.48	**94.54 ± 3.58e−1**
T-Net			4026.26 ± 185.30	69.04 ± 8.53e−1	92.58 ± 5.41e−1
Spherical			1979.17 ± 162.49	64.69 ± 8.78e−1	92.71 ± 4.36e−1

Table 3.9: This table illustrates the classification results of scaling up the number of points per point cloud and the number of augmentations on ModelNet10. The experiment for each distribution were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **Aug.** and **Pts.** stand for the number of augmentations and the number of points respectively. **Vanilla**, **T-Net**, and **Spherical** represent vanilla sampled PointNet, sampled PointNet with sampled T-Net, and sampled PointNet with spherical coordinates as additional input features on top of the Euclidean coordinates.

From Table 3.9, the records of time taken suggest that point cloud density is the dominant factor. This is because KDTree needs to consider more points for both the construction and query process. Using 512-point density from Table 3.7 as a reference point, the approaches with 3-augmentations with same density share similar computation time except for the

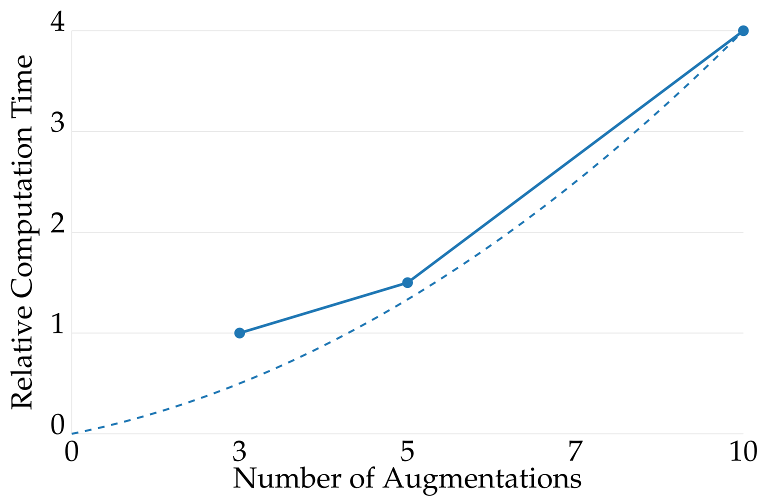


Figure 3.6: The relative computation time versus number of augmentations based on Table 3.9.

T-Net approach, which is technically two sampled PointNets (a sampled PointNet with an embedded sampled T-Net); 5-augmentations costs around 1.5 times of computation time; and 10-augmentations requires approximately fourfold. As visualized in Figure 3.6, when the number of augmentation is large, the computation time is approaching a quadratic growth. To further investigate the trends in terms of prediction metrics, we look at Figure 3.7.

In the figure, we annotate the results of new configurations on top of the charts in Figure 3.5. The results with 512 points are plotted with dotted line, while the results with 1024 are annotated as data points with the corresponding labels on their left. By increasing the density from 64 points to 512 points (compare the data points vertically), the results with 3-, 5-, and 10-augmentations show clear improvements, especially Spherical coordinates which increase the accuracy by nearly 10 units. The gaps between the approaches are smaller in terms of accuracy compared to that in terms of AUCROC, in particular, T-Net with 512 points and 10-augmentations achieves 1.75 units difference in accuracy compared to the pure data augmentation approach of same configuration (10, 512). The results with 1024 points have negligible difference compared to that of 512 points. We can accept the hypothesis that the impact of point cloud density with standard inputs is transferable to SO(2) domain.

We measure the coupling effect of the number of augmentations and the number of points per point cloud by assessing the horizontal trend of new configurations in Figure 3.7. Ideally, we would see a more positive *growth rate* of the dotted lines as the number of augmentations increases, compared to their counterparts with 64 points, indicating that scaling up data

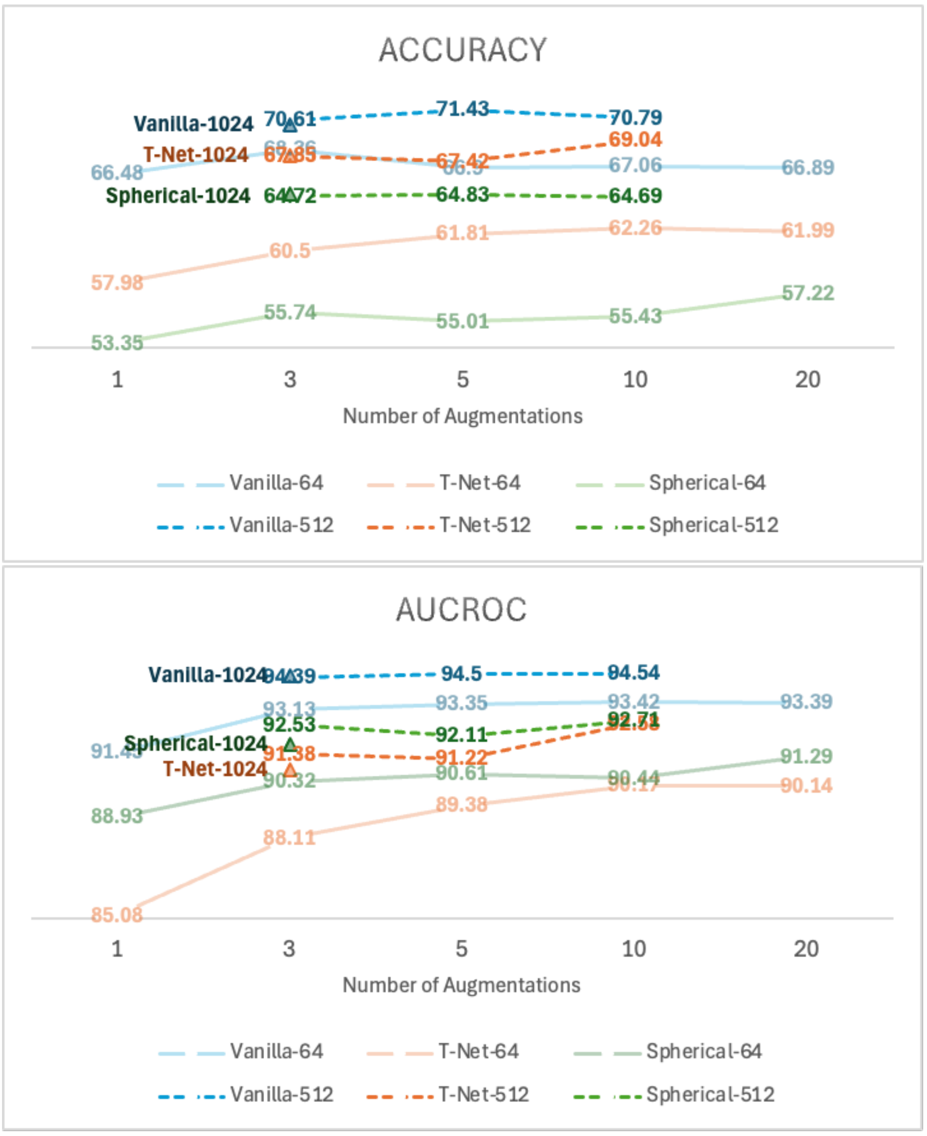


Figure 3.7: The reported results are the average over 5 different seeds. **Top:** The accuracy of three approaches as the number of augmentations increases. **Bottom:** The AUCROC of three approaches as the number of augmentations increases. In both diagrams, the dotted lines are the corresponding (same colors) results of 512 points density. The results of 1024 points are annotated as triangle markers with respective labels on their left.

augmentations and point cloud density together can produce interaction effects. However, the empirical results suggest that this is not the case. T-Net from 5 augmentations to 10 augmentations is the only configuration with relatively noticeable gains in growth rate in terms of accuracy and AUCROC. The other configurations show small fluctuations with similar magnitudes, which is the same phenomena in previous results when we only scaled up the number of augmentations.

Since the new configurations have shown a similar trend, in addition to previous analysis — that the nature of discrete optimization of sampled neural network hampers emergent properties — we decide against conducting further experiments with more number of augmentations and more points. Nonetheless, not only that the computation efficiency will be further diminished as projected in Figure 3.6, the memory demand will also be prohibited with the available computing infrastructure.

3.3.5 Discussion

In this section, we observe that scaling point cloud density and data augmentations can improve the performance of sampled PointNet. By comparing both strategies, scaling up point cloud density results in a steeper growth at the expense of longer computation time. Unfortunately, both strategies have their corresponding bottlenecks and no coupling effects are discovered. For point cloud density, the bottleneck could be different subject to the representation quality of 3D objects. Objects with more intricate structures might require more points to preserve the geometric information. For data augmentations, it is important to clarify that we measure the scaling effect solely in terms data volume. The performance might improve further if we also consider the scaling effect of model parameters, that is, adding more neurons or layers. This is because SWIM can sample more directions if given more weights as placeholders.

The lack of online learning capabilities forces sampled network to process all training data in one-go, which is not scalable. Despite we can always keep the memory demand within the given computational resources using random sampling, with this approach, sampled network can never achieve the emergent properties which is shown by recent advances with big data and foundation models. This poses interesting philosophical questions on the role of sampled network in pragmatic use cases and the direction of development: whether to go large or stay small; lastly, even with data augmentation, sampled network can only pick the best possible input pairs to construct weights. This mechanism offers no solution to edge towards the optimal continuously beyond what the input pairs can offer. Consequently, the heavy lifting falls on the last least square layer, which is a linear optimization with limited predictive power. The growth rate of performance with increasing number of data augmentation suggests the current mechanism might not be a good universal learning method.

Despite computational memory has been the bottleneck to study the effects of larger scale, computation time has approximately increased from 40 seconds to more than 30 minutes. Although it is just a fraction of the required runtime for gradient descent (which requires 3 to 6 hours according to Qi et al. [2017a]), at this rate, sampled PointNet will soon lose its advantage if the inputs continue to scale up (for example, with the standard dataset ModelNet40). A sampled PointNet with DenseKD layers has three components with heavy machinery: nearest neighbour function with KDTree, matrix multiplication, and least square optimization. All three components are implemented and optimized by experienced developer in C programming language. Among these components, matrix multiplication is the irreplaceable backbone of neural network. The only possible runtime optimization for this operation is by running on GPU, therefore we consider this option out of scope; for least square optimization, there is no replacement with significantly lighter computational load, and the convergence rate of most optimization methods varies against different datasets. Therefore, in the following section, we attempt to replace KDTree to ease the computations.

3.4 Recursive Sampling

Table 3.4 lists the dominant runtime complexity of a KDTree as O($kn \log n$). The irreducible runtime is O(n), that is, computing the distance for n points and their corresponding neighbours. The remaining O($k \log n$) complexity is the consequence of KDTree, which we can optimize by resorting to an approximation algorithm. Inspired by the Johnson-Lindenstrauss Lemma [Johnson and Lindenstrauss, 2001], which was also the early inspiration of this input pairs weight construction idea [Galaris et al., 2022], we inject probability to break the curse of dimensionality albeit we aim to reduce computation time by ignoring input dimension instead of reducing the input dimension. Note that here we assume that sampled neural network does not require the *exact* nearest neighbours — which KDTree is already the one of the fastest algorithms — for the points. We also do not need the data structure of KDTree and the accompanying advantages such as fast insertion or fast deletion, since we construct and query the KDTree only once to obtain the nearest neighbours before discarding it.

3.4.1 Naive Approach

Given the assumption that the input dimension can be ignored, we can adopt a randomized approach. However, we need a guiding heuristic to encourage the model to sample points with close proximity.

Intuitively, the SWIM sampling mechanism, which operates at the object level, already functions as a soft nearest neighbour approach in a classification setting. As long as two point clouds belong to different classes, the numerator will be 1, and the likelihood of sampling this pair is solely determined by their proximity. From this perspective, SWIM constructs weights with a high probability of using the nearest neighbours of the inputs.

It is important to note that SWIM without permutation invariance does not actually select the nearest neighbours during direction computation. Instead, it randomly samples a pool of n input pairs for direction computation with O(n) complexity, and then establishes a sampling distribution that encourages pairs with closer proximity to be selected with higher probability, using the computed directions and corresponding ground truth differences. In contrast, KDTree includes selecting nearest neighbours as an integral part of its functionality. Therefore, Table 3.2 demonstrates that a DenseKD layer is nearly agnostic to sampling distributions, as the KDTree overlaps with the functionality of the sampling distribution.

In this case, instead of using KDTree, we can deploy the sampling strategy of SWIM at the point level as the direction function $\text{dir}(\cdot, \cdot)$. Given two point clouds (X_1, X_2), the procedures for direction computation using this sampling approach (Naive Recursive Sampling) are as follows:

1. Given m points $x_1 \in X_1$, m points $x_2 \in X_2$, d_{in} point features, layer width d_{out}, ground truths Y_1, Y_2, randomly sample $\max(m, d_{out})$ pairs (x_1, x_2) from X_1 and X_2 respectively with a uniform distribution to construct a candidate pool $\mathcal{S}$.

2. Derive the gradient $\frac{||Y_1 - Y_2||_\infty}{\max(||x_1^{(i)} - x_2^{(i)}||_2, \epsilon)}$ — where ϵ is the lower bound constant for distances between pairs — for all pairs $(x_1^{(i)}, x_2^{(i)}) \in \mathcal{S}, i \in \{1, \ldots, \max(m, d_{out})\}$ and normalize into a probability distribution $\mathfrak{P}$.

3. Sample m directions $x_1 - x_2$ *with replacement*, $(x_1, x_2) \in \mathcal{S}$ following the sampling distribution $\mathfrak{P}$ as a collection denoted as $\widetilde{\mathcal{D}}$.

4. Return $\widetilde{\mathcal{D}}$ as the point cloud directions between X_1 and X_2.

After computing the point cloud directions between all selected input pairs, the model can then proceed to Step 2 of the weight construction procedure for a Dense layer to derive the gradients for all pairs, as outlined at the beginning of Section 3. This direction computation algorithm is largely similar to the sampling procedure described in Section 2.2.1. Moreover, this approach can be easily extended to other data modalities with deeper nested structures, by recursively applying the direction computation algorithm across multiple levels. This algorithm also preserves the flexibility to use different sampling distributions, depending on the specific use case. To distinguish this layer from SWIM's Dense layer and DenseKD, we refer to the one with this recursive sampling method as DenseR.

However, DenseR samples point pairs that lie close to the gradient with a relatively low probability compared to DenseKD (see Figure 3.8). Most pairs sampled using DenseR tend to overshoot, with few lying close to the gradient. This raises two questions:

1. Why are these long pairs being selected despite the discouragement from the probability distribution?

2. How can we select more short pairs?

By inspecting the probability distribution, as illustrated in the bottom left of Figure 3.9, it indicates that the difference in proximity will be increasingly reduced during normalization as the number of pairs increases. The y-axis of the plots are set to range [0,1] on purpose, to emphasize the *insignificance* of differences of probabilities when the sample size grows.

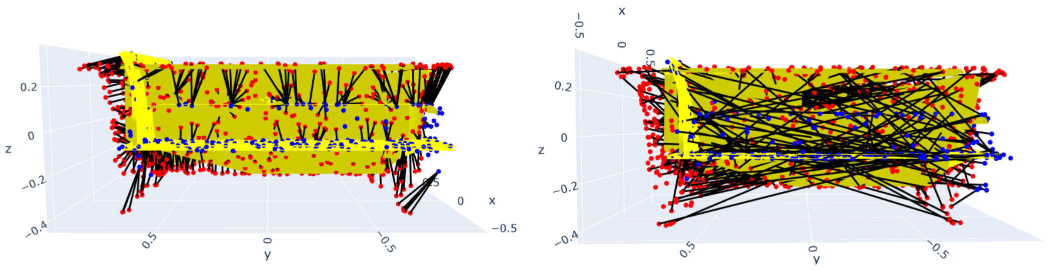


Figure 3.8: This figure shows the differences between the point cloud directions (black lines) between a bed (blue) and a bathtub (red), computed using KDTree and Naive Recursive Sampling. To enhance the visualization effect, we use 1024 point per point cloud and 1024 sampled directions. **Left:** KDTree. **Right:** Naive Recursive Sampling.

The nearly uniform probability distribution cannot serve as a strong guidance to select point pairs because the number of point pairs with long proximity is always more than that of short proximity. In general, there is no good way to *sample* more close proximity pairs without additional complexity. Since previous results already show that sampled neural network does not need optimal short distance pairs, we turn our attention the next possible optimization to answer the second question, namely pair selection.

3.4.2 Sampling Good Pairs

Given the sampled pool of point pairs, we can further refine the selection by choosing more pairs with shorter distances. We have already computed the directions and distances between pairs to derive the gradient, so it is straightforward to exclude the undesired pairs based on distance or gradient by selecting an arbitrary quantile. In the bottom right of Figure 3.9, we can see the probabilities of the 13 point pairs that are in the top 20% in terms of gradient. This sampling distribution with fewer pairs is more "opinionated" than the distribution shown in the bottom left. Compared to the naive approach, this method with a specified quantile α has an extra step as follows:

1. Given m points $x_1 \in X_1$, m points $x_2 \in X_2$, d_{in} point features, layer width d_{out}, ground truths Y_1, Y_2, randomly sample $\max(m, d_{out})$ pairs (x_1, x_2) from X_1 and X_2 respectively with a uniform distribution to construct a candidate pool $\mathcal{S}$.

2. Derive the gradient $g^{(i)} = \frac{||Y_1 - Y_2||_\infty}{\max(||x_1^{(i)} - x_2^{(i)}||_2, \epsilon)}$ — where ϵ is the lower bound constant for distances between pairs — for all pairs $(x_1^{(i)}, x_2^{(i)}) \in \mathcal{S}, i \in \{1, \ldots, \max(m, d_{out})\}$.

3. Select $\mathcal{G}_\alpha = \{g | g \in \{g^{(i)}\}, g > \text{quantile}(\{g^{(i)}, \alpha)\}\}$, where $i \in \{1, \ldots, \max(m, d_{out})$ and α is the given quantile threshold. Normalize $\mathcal{G}_\alpha$ into a probability distribution $\mathfrak{P}_\alpha$.

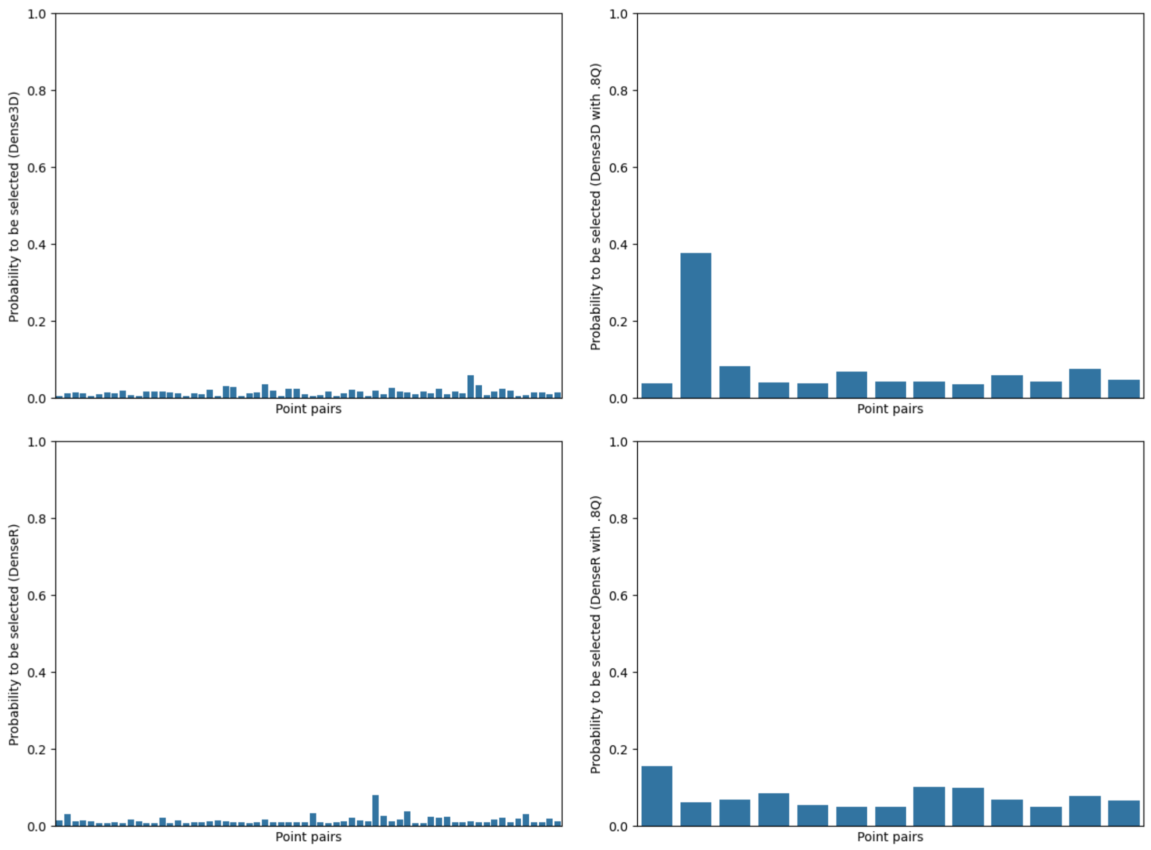
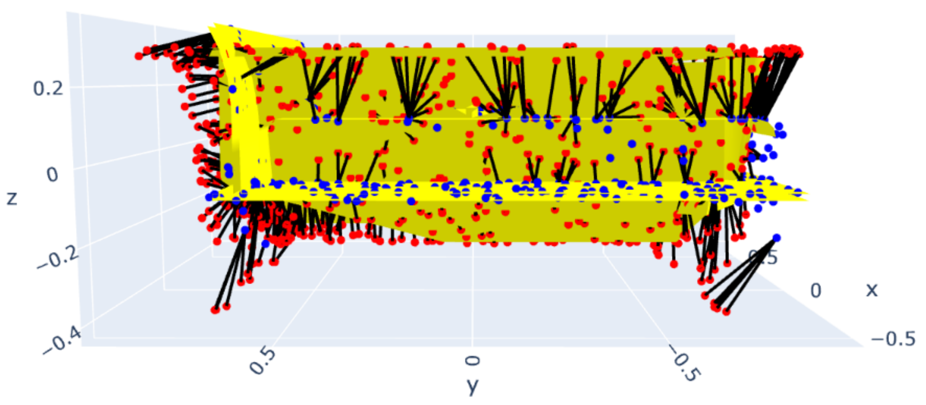


Figure 3.9: All figures show the probability to be selected for the point pairs. The top row are the point pairs chosen by KDTree. The bottom row are chosen by recursive sampling. The left column contains 64 point pairs while the right contain the point pairs above 80% quantile in terms of inverse distance (gradient).

4. Denote $\mathcal{S}_\alpha$ as a shrunk candidate pool which contains the pairs correspond to the elements in $\mathcal{G}_\alpha$. Sample m directions $x_1 - x_2$ *with replacement*, $(x_1, x_2) \in \mathcal{S}_\alpha$ following the sampling distribution $\mathfrak{P}_\alpha$ as a collection denoted as $\widetilde{\mathcal{D}}$. Return $\widetilde{\mathcal{D}}$ as the point cloud directions between X_1 and X_2 to compute gradients on object level (Step 2 of the procedure in Section 3).

In Figure 3.10, we visualize the differences of the sampled directions between KDTree and Sampling Good Pairs (using quantile). Since the quantile filter shrinks the sample pool, it is expected that the number of directions at the bottoom is fewer that that of the top. Despite the directions of KDTree point towards the steep gradient areas in yellow, however, that of Sample Good Pairs are closer to the steep gradient areas, despite the orientations are not as neat as the former.



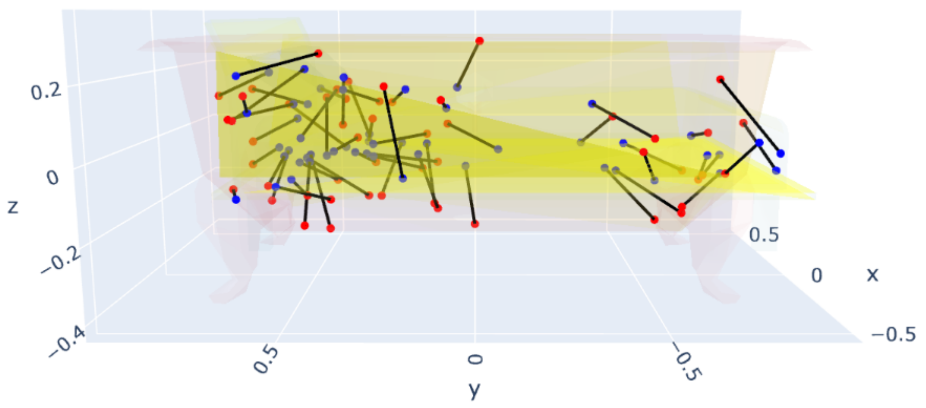


Figure 3.10: This figure shows the differences of the sampled directions (blacklines) between KDTree and Sampling Good Pairs. For better visibility, the figure displays only the sampled directions. The yellow region is the approximated steep gradient areas. **Top:** KDTree. **Bottom:** Sample Good Pairs.

This approach raises a concern about the variance of the selected pairs, as applying a quantile filter effectively shrinks the sampled pool. Furthermore, the quantile filter has no mechanism to control which gradient areas (if there are more than one) are sampled. Consider a scenario where there are two gradient areas between two point clouds. If the corresponding magnitudes for all filtered gradients are similar, regardless of the gradient area, then the gradient area with a higher proportion of gradients in the filtered pool will naturally have more pairs selected. Conversely, if the magnitudes for gradients from one area are always higher than the other, this will also lead to an imbalance in the representation of the sampled pairs. These abstract conjecture are visualized in on the left of Figure 3.11. The sampled pairs mostly concentrated to at the top of the object, which has a relatively larger contact/intersection area, while the distant pairs have relatively thin lines, representing low counts in occurrence. Consequently, this approach cannot always improve the overall representation, as it only passively increases the selection chances of good pairs from different gradient areas, without any active control over the sampling process.

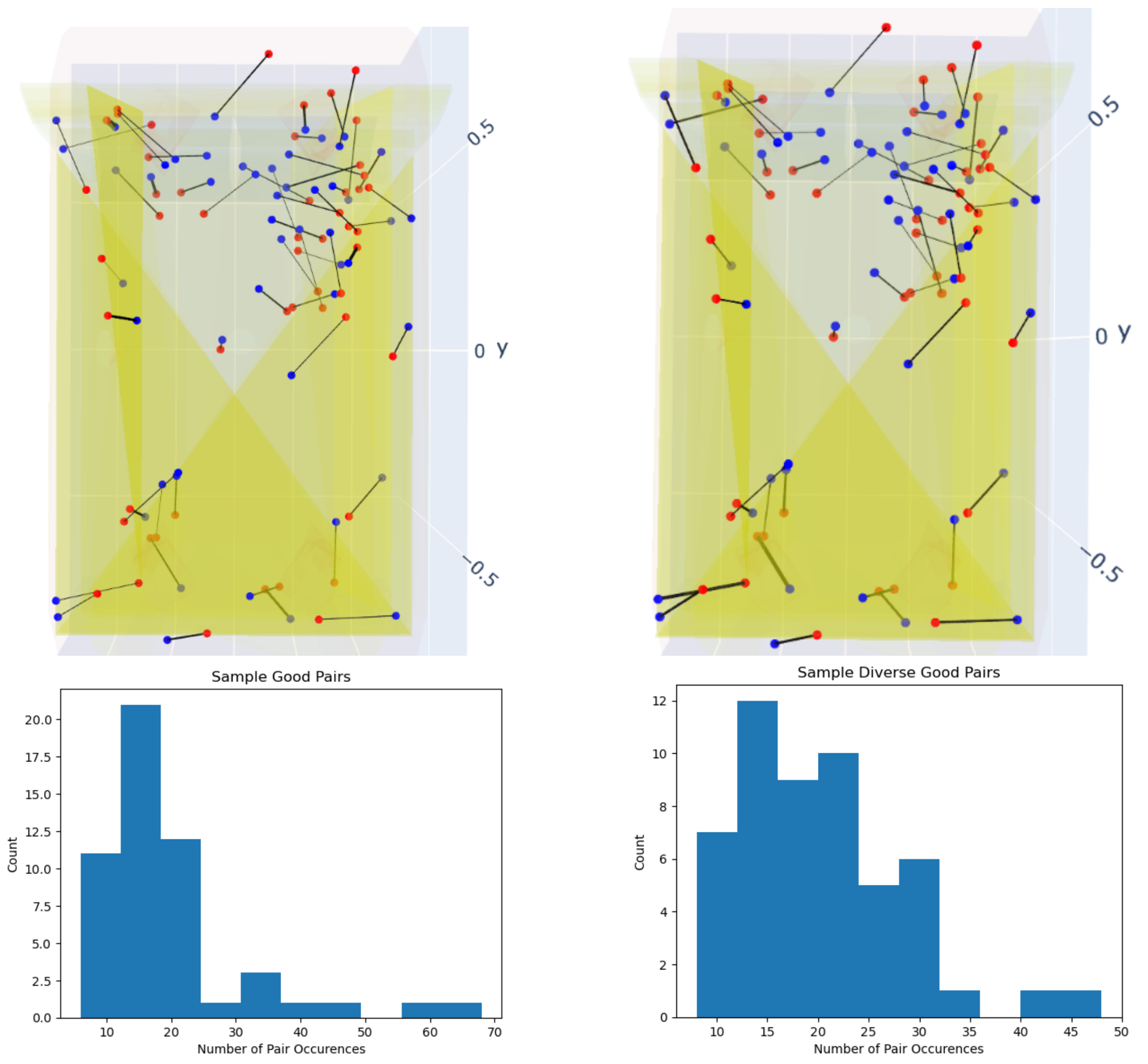


Figure 3.11: The figure shows the differences between Sampling Good Pairs and Sampling Diverse Good Pairs. **Top:** Top view of a bed and a bathtub with yellow intersection areas. For better visibility, only the selected pairs are displayed. The thickness of the black lines is the standardized number of pair occurrence amplified using a constant multiplier of 100. **Bottom:** The histograms of pair occurrence. **Left:** Sampling Good Pairs. **Right:** Sampling Diverse Good Pairs.

3.4.3 Sampling Diverse Good Pairs

The challenge is that we do not know the gradient areas in advance, so we must rely on an unsupervised approach. Logically, a minor gradient area will have fewer point pairs compared to a major gradient area, as the point clouds are sampled proportionally to the face areas of the mesh. Consequently, a minor gradient area will be sparser, meaning a point from one point cloud will have fewer close neighbours from the same point cloud in that area.

Following the intuition that points in minor gradient areas have fewer close neighbours compared to major gradient areas, we can approximate the gradient area of a point by aggregating the distances to all its neighbours. Specifically, a point from a minor gradient area would have a larger aggregated distance to all neighbours. Conversely, points in major gradient areas would have smaller aggregated distances to their neighbours. These aggregated distances can then serve as indicators of the gradient areas to differentiate if a gradient area is minor or major.

While the time complexity to compute the pairwise distances of an m-point point cloud is O(m^2), which is expensive compared to the O(m) sampling approach, we can optimize this step. Our objective is to sample diverse, good pairs, so we only need to know if the selected good pairs are relatively far apart from each other. After filtering the point pairs using a quantile, the number of remaining pairs, m_α, is often much smaller than m, otherwise the filter cannot effectively exclude the bad pairs. Therefore, we can afford to compute the $m_\alpha \times m_\alpha$ distance matrix of the points from one point cloud and sum up the columns to obtain the gradient area indicators.

With the indicators, we want to prioritize the pairs from under-represented areas. To do this, we can select the points with larger aggregated distances, as these are more likely to be from minor gradient areas. The original SWIM sampling distribution prioritizes pairs with close proximity, which have already been selected by the quantile filter. Since we now know all the selected pairs are good, we need a different sampling distribution that prioritizes large distances between the source points of the pairs.

This is where the length squared distribution, slightly different the definition in Equation 3.3, comes into play. Here, we compute the distribution by normalizing the pairwise squared distance matrix of the source points of the good pairs, rather than the distances of the pairs themselves. This approach ensures that the minor gradient areas, which naturally have fewer good pairs, are more likely to be prioritized in the sampling process. However, since the major gradient areas have a higher proportion of good pairs, they will still be well-represented, even though the minor areas are given higher priority. The inherent low number of pairs in the minor areas means the major areas will also get selected, preventing them from being under-represented or overshadowed. A visual aid of this approach can be found on the right of Figure 3.11, which contains more balanced distribution of the sampled pairs compared to that of the left of the figure. The detail procedures are as follows:

1. Given m points $x_1 \in X_1$, n points $x_2 \in X_2$, d_{in} point features, layer width d_{out},

ground truths Y_1, Y_2 randomly sample $\max(m, d_{out})$ pairs (x_1, x_2) from X_1 and X_2 respectively with a uniform distribution to construct a candidate pool $\mathcal{S}$.

2. Derive the gradient $g^{(i)} = \frac{||Y_1 - Y_2||_\infty}{\max(||x_1^{(i)} - x_2^{(i)}||_2, \epsilon)}$ — where ϵ is the lower bound constant for distances between pairs — for all pairs $(x_1^{(i)}, x_2^{(i)}) \in \mathcal{S}, i \in \{1, \ldots, \max(m, d_{out})\}$.

3. Select $\mathcal{G}_\alpha = \{g | g \in \{g^{(i)}\}, g > \text{quantile}(\{g^{(i)}, \alpha\}), i \in \{1, \ldots, \max(m, d_{out})\}$ and α is the given quantile threshold.

4. Denote $\mathcal{S}_\alpha$ as a shrunk candidate pool which contains the pairs correspond to the elements in $\mathcal{G}_\alpha$. Compute length squared distribution $\mathfrak{P}_{\text{LS}}$, which is defined as:

$$\mathfrak{P}_{\text{LS}}(x_1^{(i)}) = \frac{\sum_{x_1^{(i)} \neq x_1^{(j)}} ||x_1^{(i)} - x_1^{(j)}||_2^2}{\sum_{x_1^{(\tilde{i})}} \sum_{x_1^{(\tilde{i})} \neq x_1^{(j)}} ||x_1^{(\tilde{i})} - x_1^{(j)}||_2^2}, \quad x_1^{(i)}, \forall x_1^{(\tilde{i})}, \forall x_1^{(j)} \in \mathcal{S}_\alpha.$$

5. Sample m directions $x_1 - x_2$ *with replacement*, $(x_1, x_2) \in \mathcal{S}_\alpha$ following the sampling distribution $\mathfrak{P}_{\text{LS}}$ as a collection denoted as $\widetilde{\mathcal{D}}$. Return $\widetilde{\mathcal{D}}$ as the point cloud directions between X_1 and X_2 for gradient computation.

3.4.4 Results

To recapitulate, the sampling algorithm for a DenseKD or DenseR is as follows:

1. Given the materialized point cloud direction function $\text{dir}(\cdot, \cdot) \in \mathbb{R}^{m \times d_{in}}$, n point clouds $X \in \mathbb{R}^{m \times d_{in}}$ where m is the number of points, d_{in} is the number of features, layer width d_{out}, n outputs $y \in \mathbb{R}^{d_{gt}}$ where d_{gt} is the ground truth dimensions, randomly sample $\max(n, d_{out})$ pairs (X_1, X_2) from the inputs with a uniform distribution to construct a sampling frame $\mathcal{S}$. Two point clouds in a pair must be different, $X_1 \neq X_2$.

2. Derive the gradient $\frac{||y_1^{(i)} - y_2^{(i)}||_\infty}{\max(||\text{dir}(X_1^{(i)}, X_2^{(i)})||_F, \epsilon)}$ — where $\text{dir}(\cdot, \cdot)$ could be one of KDTree, Naive Recursive Sampling, Sampling Good Pairs, or Sampling Diverse Good Pairs; and ϵ is the lower bound constant for distances between pairs — for all pairs $(X_1^{(i)}, X_2^{(i)}) \in \mathcal{S}, i \in \{1, \ldots, \max(n, d_{out})\}$ and normalize into a probability distribution $\mathfrak{P}$.

3. Sample m pairs *with replacement* from $\mathcal{S}$ following the sampling distribution $\mathfrak{P}$ as a collection denoted as $\widetilde{\mathcal{S}}$.

4. Construct d_{out} weights $w^{(i)} = s_1 \frac{\text{dir}(X_1^{(i)}, X_2^{(i)})}{||\text{dir}(X_1^{(i)}, X_2^{(i)})||_F^2}$ — where $w^{(i)} \in \mathbb{R}^{m \times d_{in}}$, $s_1 \in \mathbb{R}$, $(X_1^{(i)}, X_2^{(i)}) \in \widetilde{\mathcal{S}}$, $i \in \{1, 2, \ldots, d_{out}\}$ — and d_{out} biases $b^{(i)} = -\langle w^{(i)}, X_1^{(i)} \rangle - s_2$, where $b^{(i)} \in \mathbb{R}^m, s_2 \in \mathbb{R}$.

Nr. Points.	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
64	11.34 ± 0.36	69.98 ± 3.03	92.12 ± 1.61
512	73.09 ± 6.50	$\mathbf{76.31 \pm 1.79}$	$94.83 \pm 8.83\text{e}{-1}$
1024	109.26 ± 12.78	73.81 ± 1.32	$93.59 \pm 7.15\text{e}{-1}$
2048	180.31 ± 6.48	75.92 ± 3.14	$\mathbf{94.85 \pm 9.44\text{e}{-1}}$

Table 3.10: This table illustrates the performance of DenseR on ModelNet10 using Naive Recursive Sampling, with {64, 512, 1024, 2048} points per point cloud. The experiment for each distribution were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better). **Nr. Points** represents the number of points in a point cloud.

Naive Recursive Sampling

Table 3.10 presents a comprehensive comparison of DenseR and DenseKD in terms of training time and performance metrics on ModelNet10. The corresponding visualizations for time taken and accuracy are shown in Figure 3.12, which includes the same metrics from Table 3.7 for comparison.

The results affirm our intuition that replacing KDTree with the recursive sampling approach can significantly shorten the training time. The time complexity of KDTree is O($n \log n$), which is referred to as quasilinear or linearithmic time. As n becomes large, the additional $\log n$ term tends to become a constant, making linearithmic time equivalent to linear time with a multiplier. The dotted lines in Figure 3.12 represent the linear trendlines with annotated equations. The trendline for DenseR fits perfectly, while that for DenseKD has some residuals. The gradient of DenseKD's trendline is approximately 7 times that of DenseR, indicating that the runtime of DenseKD will grow 7 times faster than DenseR as the point cloud density increases. This finding is consistent with our complexity approximation, where the multiplier is 7.

The promising gain in runtime comes at the expense of prediction performance, with an average decrease of 11 units across all 4 densities. This significant drop indicates that sampling as an approximate nearest neighbour function cannot be a direct replacement for KDTree, which performs exact nearest neighbour search. The accuracy trend of DenseR, which peaked at 512 points, differs from that of DenseKD. To further explain the trend in Table 3.7, we discuss the impact of the soft nearest neighbour function.

Given two point clouds with m points each, there are m^2 possible point pairs, of which m pairs have the closest proximity (top-1 nearest neighbour). The remaining $m^2 - m$ pairs are likely not close, based on the SWIM geometric intuition. Using random sampling, the probability of selecting a closest pair is $\frac{1}{m}$, which approaches 0 as m increases. This explains the drop in performance, as shown in Figure 3.8, where the Naive Recursive Sampling can hardly select the closest pairs. However, the high AUCROCs in Table 3.10 suggest that

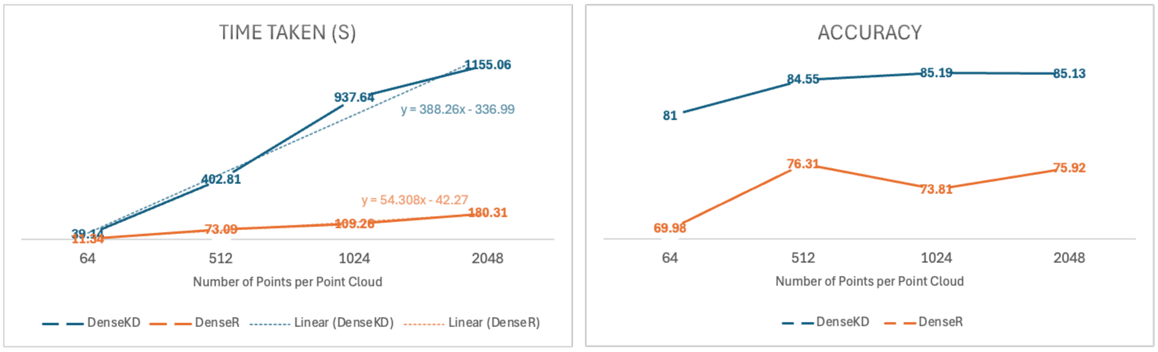


Figure 3.12: The reported results are the average over 5 different seeds. **Left:** The time taken for DenseKD and DenseR to finish training when the number of points per point cloud increases. The dotted lines are the linearly projected trendline with annotated equations in same color. **Right:** The accuracy of DenseKD and DenseR when the number of points per point cloud increases.

the model is not random guessing. Therefore, on the bright side, the assumption that the sampled neural network does not require exact nearest neighbours holds true. Next, we evaluate the effectiveness of quantile filter in sampling good pairs from the inputs.

Sampling Good Pairs

Figure 3.13 presents the time taken and the accuracy of ModelNet10 classification of sampling from top 40%, 20%, and 5% quantile range. The results from Figure 3.12 are plotted with faded effect for visual comparison.

In the figure, it is evident that the additional operations of selecting quantile values and filtering undesired point pairs do not significantly impact the overall runtime. DenseR with three different quantiles exhibits a linear growth rate. The empirical results confirm our intuition with significant accuracy gains of approximately 5 to 10 units, as shown on the right of Figure 3.13.

The trends observed across all quantiles allow for several plausible interpretations. It's important to recall that the sample size for point pairs is determined by the maximum between point cloud density and layer width. Moreover, as quantile is a rank-based statistic, it ensures consistency in the shrunk sampled pool size. Consider the first layer of sampledz PointNet, which has a width of 64 and a density of 64 points. In this case, the sampled pool sizes for top quantile ranges of 40%, 20%, and 5% are 26, 13, and 4, respectively. The indistinguishable results across all quantiles for 64-point point clouds suggest that the differences between two point clouds can be effectively represented in a 64-row direction matrix constructed using as few as four fundamental directions, each repeating a random number of times.

Remember that the sampling space for the subsequent layer is determined by the weights,

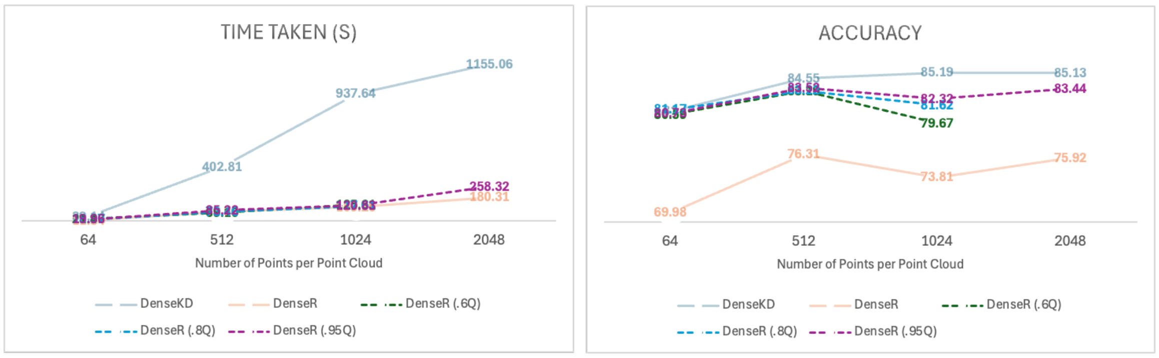


Figure 3.13: DenseR samples from top quantile ranges. The reported results are the average over 5 different seeds. **Left Dotted:** The time taken for DenseR with {0.6,0.8,0.95}-quantile range selection to finish training when the number of points per point cloud increases. **Right Dotted:** The accuracy of DenseR with {0.6,0.8,0.95}-quantile range selection when the number of points per point cloud increases.

which are derived from the point cloud differences. An accuracy of 80.73% with the top 5% quantile and 64 points density indicates that the first layer can construct a decent sample space for subsequent layers using only four 'basis' directions for each point cloud pairs. We adopt a probabilistic perspective to quantify out the "quality" of these four directions. The probability of sampling 4 good pairs out of all possible combinations follows a hypergeometric distribution:

$$p(N_{\text{good}}, M, M_{\text{good}}, N) = \frac{\begin{pmatrix} M_{\text{good}} \\ N_{\text{good}} \end{pmatrix} \begin{pmatrix} M - M_{\text{good}} \\ N - N_{\text{good}} \end{pmatrix}}{\begin{pmatrix} M \\ N \end{pmatrix}}, \tag{3.4}$$

where N_{good} is the number of sampled good pairs, M is the total number of all possible pairs, M_{good} is the total number of good pairs, and N is the number of sampled pairs. In our case, we do not know the exact number of good pairs M_{good}. However, it is reasonable to assume that, on average, a point can form j good pairs, which gives us an approximate total number of good pairs $M_{\text{good}} = 64j$. The probability of having at least 4 good pairs $p(N_{\text{good}} \geq 4)$ with $j = \{3, 5, 10\}$, $M = 64^2$, and $N = 64$ is $\{0.3526, 0.7485, 0.9937\}$ respectively.

Under the classification setting, we can simplify the definition of a good direction (equivalent to steep gradient) to close proximity. It is intuitive that the larger the true extent of close proximity, the more good pairs can be formed. Our analysis shows that the standard deviation of the top 5% quantile range of 64 points is 1.21 units of accuracy across 5 different seeds, suggesting a high likelihood of having at least 4 good pairs. Given the sparseness of point clouds with only 64 points, we can conclude that the threshold for a distance to qualify as a good pair should be relatively low.

When we feed the model with 512-point point clouds, we observe negligible accuracy differences across the three given quantile ranges. This observation leads to two important insights: firstly, it suggests that sampling more good pairs is beneficial for model performance. Secondly, it indicates that SWIM exhibits a degree of robustness against noise, as 512-point clouds are more likely to include additional bad pairs compared to sparser representations. The improved performance with 512 points can be attributed to the increased availability of good pairs at higher densities. However, we see a performance drop when further increasing to 1024 points, suggesting an optimal density sweet spot for best performance.

We also observe the impact of quantiles on performance varies across different densities. For instance, the top 40% quantile range of 512 points (204 pairs) outperforms the top 5% quantile range of 1024 points (51 pairs) in terms of accuracy. This is counterintuitive. Given that the 512-point scenario, with four times as many pairs, inevitably includes more bad pairs than the top 5% quantile range of 1024 points, thus, we cannot attribute the entire performance drop to noise alone.

When we consider the rich representation offered by 1024 points, it gives rise to another reason for the decrease in performance. It is possible that uneven ratios of gradient areas in the point cloud lead to disproportional pair selections. In this scenario, good pairs from minor gradient areas might be overshadowed by those from major gradient areas, resulting in an overall under-representation of certain features. The top 20% quantile range of 1024-point density contains 204 pairs, matching the number in the top 40% quantile range of 512 points. Its slightly worse performance supports our hypothesis of under-representation, where dominant but redundant information from major gradient areas overshadows minor features.

Given that the top 5% quantile range has consistently yielded the best performance for both 512 and 1024 points, it is logical to focus on this range when examining 2048-point clouds, rather than experimenting with the remaining two quantile ranges. The top 5% quantile range of 2048 points (comprising 105 pairs) leads to better accuracy than that of 1024 points. This suggests that under-representation can be mitigated once the number of points in minor gradient areas exceeds a certain threshold. The increased point density allows for better sampling across all gradient areas, including those previously under-represented. It is worth noting that the slight accuracy difference between 512 points and 2048 points is statistically insignificant, as both values fall within the standard deviation of each other. This observation implies that beyond a certain point density, the benefits of increasing the number of points may plateau, which is consistent with our findings in Section 3.3.1.

Sampling Diverse Good Pairs

The results of this approach are plotted in Figure 3.14, showing both training time and accuracy. Notably, there are significant differences in training time among the three quantiles, confirming that computing the distance matrix is indeed computationally intensive for

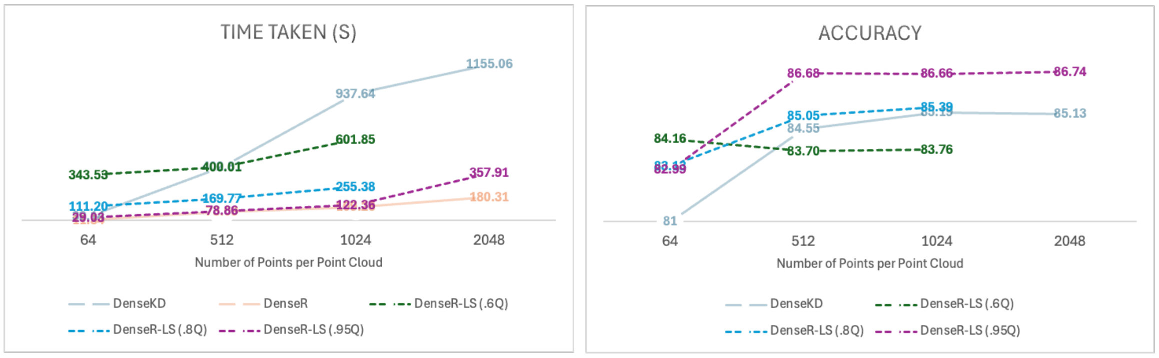


Figure 3.14: DenseR samples from top quantile range with Length Squared distribution of the pairwise distance of source points. The reported results are the average over 5 different seeds. **Left Dotted:** The time taken for DenseR with {0.6,0.8,0.95}-quantile range selection to finish training when the number of points per point cloud increases. **Right Dotted:** The accuracy of DenseR with {0.6,0.8,0.95}-quantile range selection when the number of points per point cloud increases.

larger sampled pools of good pairs. Fortunately, we need not consider large quantile ranges due to their negative impact on performance. The run time for the top 5% quantile range appears to scale linearly, except for the 2048-point density, which is already considered too dense to contribute meaningfully.

The accuracy results are encouraging and support our intuition. This new approach outperforms its predecessor (DenseR + Quantile) across all configurations. Moreover, only two configurations with the top 40% quantile range show worse performance compared to DenseKD. These two configurations, using 512 and 1024 point densities (with similar or worse performance expected for 2048 points with the same quantile), likely include a substantial number of bad pairs. This hypothesis is supported by the relatively significant differences (at least one standard deviation) among the quantiles of 512-point density, compared to its predecessor.

Without the length-squared distribution of source point distances, the selected pairs are likely dominated by good pairs from major gradient areas, leading to the indistinguishable results shown on the right of Figure 3.13. In contrast, our current approach uses the quantile filter to retain good pairs while blindly selecting those with far distances from others. This method, however, is not without risk. If bad pairs manage to pass through the quantile filter — for example, those with source points located on outer edges rather than in intersections (gradient areas) — they are likely to be selected due to their large proximity to the source points of good pairs in the gradient areas.

The accuracy discrepancies across three selected distance quantiles for 64-point density in Figure 3.14 offer valuable insights into the relative importance of good pairs versus gradient diversity. Based on our previous assumption that each point from a 64-point cloud can form 10 good pairs, the probability of sampling four good pairs out of 640 possible pairs in

64 draws approaches certainty. Selecting these four pairs with the top 5% quantile range provides a solid foundation, achieving around 80% accuracy with the original sampling distribution.

Switching to the length-squared distribution of pairwise source points' distances yields a two unit accuracy improvement with a 0.6-unit standard deviation. This suggests that the proportions of these four pairs in the 64-row direction matrix are significant. The top 40% quantile of 64-point density further improves accuracy by two units with a two units standard deviation. The 26 selected pairs from top 40% quantile in this case are certainly more representative than 4 pairs from top 5%, raising the performance ceiling. However, this ceiling is challenging to reach. Using Equation 3.4, the probability of sampling all 26 good pairs out of 192, 320, 640 pairs — corresponding to the assumption of 3, 5, 10 expected pairs per point — is nearly impossible, with probabilities of $\{7.77\text{e}{-16}, 2.41\text{e}{-13}, 1.10\text{e}{-6}\}$ respectively.

As density increases, we observe the negative impact of long-proximity pairs as gradient diversity saturates. The 0.6 and 0.8 quantiles, which contain more bad pairs, show worse performance. This is possibly due to the influence of bad pairs outweighing that of good pairs in shaping the representation. The performance with the 0.95 quantile reaches a plateau — surpassing the exact nearest neighbours from DenseKD — after increasing the density from 64 to 512 points. This indicates that the diversity of directions saturates with good proportions of good and bad pairs. These results successfully verify the importance of gradient diversification in improving model performance.

3.4.5 Discussion

The journey of searching a more efficient replacement for KDTree is fruitful. Using basic probability and statistics, we found a linear run time approach with comparable performance with KDTree. We would like to highlight that the integration of sampling top quantile range and length squared distribution of pairwise source points' distances is also applicable to DenseKD and will likely outperform DenseR, since it is clear from Figure 3.10 that the pair selection of DenseKD can still be optimized by choosing the optimal distances among exact nearest neighbours with the probability distributions shown in the top row of Figure 3.11. However, taking scalability into consideration, it might be a good idea to stick with DenseR for dense point cloud. It is also intriguing to study the impact of adding quantile filter and gradient diversification on the object level, which can be generalized to all SWIM applications. We leave that for future work. For the latter sections, we use DenseR with Sampling Diverse Good Pairs to construct sampled PointNet, since it has the best trade-off between computation time and performance.

3.5 Discrete Online Learning

Online learning is a powerful paradigm in machine learning where models are trained sequentially on incoming data, enabling for training with large datasets which could not be fitted entirely into computational memory. At the heart of many online learning algorithms lies the concept of batching, a technique that processes data in small, manageable chunks rather than all at once. Batching strikes a crucial balance between computational efficiency and statistical accuracy, making it particularly effective when combined with gradient descent optimization. Unlike some optimizers — such as least square — that require a global view of the data, gradient descent can make meaningful progress with just a subset of examples. By using batches, we can approximate the true gradient of the entire dataset, enabling more frequent model updates and often leading to faster convergence. This synergy between online learning, batching, and gradient descent is especially powerful in scenarios with large datasets or streaming environments where data arrives sequentially, allowing models to continuously learn and adapt to changing patterns.

In previous sections, we mentioned in several occasions that computation memory hinders SWIM to process large datasets. After reducing the time complexity using DenseR, space complexity is the only obstacle left to improve SWIM's scalability. Given the rise of stochastic gradient descent, online learning should also be the way forward for SWIM. However, due to the weight construction mechanism and lack of gradient feedback, SWIM cannot update the weights and biases continuously using the feedback. In light of this, we attempt to redefine online learning from SWIM's perspective by accommodating its unique features.

3.5.1 Weight Evolution

The fundamental idea of online learning is the ability to update weights and biases after every inference (the forward pass). The current state of SWIM resets all weights and biases for each inference, and fully adapts to the new inputs. In other words, SWIM has no memory. To enable SWIM with memory, we need to design a mechanism to *retain* and *update* the weights and biases. In other words, we want to replace some constructed weights and biases if the model can find better weights and biases from the new inputs.

A natural follow-up would be determining better weights and biases, with the only reference point for comparison being the existing weights and biases. Fortunately, comparing the old weights and biases with potential new ones is akin to comparing input pairs from a batch of inputs. Weights and biases are constructed using directions and distances, which serve as operands for comparisons. Intuitively, for every inference, we mix selected directions with the newly sampled directions from the new inputs to form a new sampling frame. By normalizing the respective sampling logits — for the original SWIM sampling distribution, the gradient; for length-squared sampling distribution, the features of selected source points — we obtain a sampling distribution that contextualizes the competitiveness of old weights and biases with new inputs.

Using the sampling distribution, a Dense or DenseKD or DenseR layer can follow the same procedure to construct updated weights and biases. This updating process is akin to an evolution to keep the good weights and discard the bad weights following a pre-defined heuristic (the chosen sampling distribution). Note that this mechanism does not concern the point cloud direction function $\mathrm{dir}(\cdot,\cdot)$, which was the focus in Section 3.1 and Section 3.4. It consists of only a few extra steps between directions computation and forming the sampling distribution (assuming the original SWIM distribution) as detailed below:

1. Compute point cloud directions and ground truth differences of new inputs.
2. Concatenate the directions and corresponding ground truth differences of old weights with that of new inputs.
3. Establish the new sampling distribution using the concatenated directions and ground truths.
4. Sample directions to compute new weights and biases.

To use a different sampling distribution, simply substitute the directions and ground truths with corresponding sampling logits for concatenations. However, we could not perform the same mechanism for Linear layer, in particular a least square optimizer, due to its inherent incapability to retain and update weights. Nonetheless, we might not need to update the chosen optimizer multiple times. This is because the responsibility of the linear optimizer is only to project the embeddings from a latent space constructed by sampled directions, to the output space. Conceptually, the process of weight updates is also the process to perfect the constructed latent space; with every weight replacement, the sampled directions establish a latent space with better representation. Thus, it makes sense to optimize the last layer only when the latent spaces are perfected, since the previous weights and biases of last layer will be reset anyway.

To this end, the training scheme will be a little different with the addition of online learning, since we want to update the Dense layers and its variants for multiple times and only need to optimize the least square optimizer for once:

1. For the first batch, constructs initial weights and biases for all Dense layers and its variants.
2. For the subsequent batches, updates the weights and biases for all Dense layers and its variants.
3. Lastly, sample a representative batch from train data, make a forward pass through all Dense layers and its variants, and optimize the least square layer.

With this training scheme in place, the computational memory should no longer be an issue for training on large dataset. To see if this mechanism can raise the ceiling of the performance, we consider these two follow-up questions :

- How can we ensure that the updated weights can produce a latent space with better representation?
- How can we sample a representative batch for the Linear layer?

We attempt to answer these questions by investigating the batching configurations.

3.5.2 Batching Matters

The batching configurations in our context contain three key factors: method, batch size, and the number of iterations. We consider two methods of batching, namely sequential and random. Given a batch size, sequential batching divides the train data into chunks following the given permutations; random batching is essentially subsampling the train data. We hypothesize that sequential batching might lead to worse performance, because it reduces the number of combinations of input pairs — inputs from different batches can never pair up. With random batching, while inputs may be resampled repeatedly, each batch is likely to have different input combinations, thereby broadening the "horizon" of a sampled network. With more choices for point cloud directions, we speculate that the sampled network can produce a latent space with a better representation.

The second component, batch size, also has an impact for iterative gradient methods. Typically, the rule of thumb is to choose a batch size as large as the memory can accommodate. The larger the batch size, the stabler the training. In the context of SWIM, a smaller batch size determines the number of input pairs, resulting in fewer possible combinations. Moreover, it influences the nature of the linear system — under-determined, well-determined, or over-determined — for the least square optimizer to solve. With fewer samples, a smaller batch size also provides fewer optimization opportunities for the least square method to refine its weights. Therefore, we should use larger batch size so that a representative batch can be sampled for the optimization.

Finally, we discuss third key component: the number of iterations. This factor represents the number of opportunities for a sampled neural network to update its weights during the training process. A common initial choice for the number of iterations is the floor of the quotient obtained by dividing the total number of data points in the training set by the batch size. However, this approach is akin to one-shot learning, as the model will only encounter the dataset once. Instead, one can either fix an arbitrary number of iterations or implement an early stop mechanism to identify the optimal number of iterations for training the neural network effectively. This way, the model can make the best use of its learning opportunities and improve its weights accordingly.

3.5.3 Results

We design three experiments to investigate the impact of these three key factors: batching method, batch size, and the number of iterations. We use DenseR with quantiles and length-squared distributions of the sampled source points to construct the shared Fully-Connected layers. The inputs have 512-point density.

Batching Methods

Methods	Time Taken (s) $\Downarrow$	Accuracy (%) $\Uparrow$	AUCROC (%) $\Uparrow$
Sequential	106.47 ± 19.71	$\mathbf{72.84 \pm 2.93}$	$\mathbf{89.26 \pm 1.34}$
Random	$\mathbf{90.07 \pm 6.50\text{e}{-1}}$	$72.15 \pm 8.74\text{e}{-1}$	88.22 ± 1.44

Table 3.11: This table illustrates the performance of sequential batching and random batching on ModelNet10 with 256 batch size, 512 point cloud density, and 10 iterations. The experiment for each distribution were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better).

From Table 3.11, both methods have similar performance with the sequential batching being slightly superior. Despite the advantage being insignificant, our empirical results reject the hypothesis that sequential batching would deteriorate the sampled representations due to fewer possible combinations between input pairs. Using a small batch size of 256, we observe that random batching yields smaller standard deviations for both computation time and accuracy compared to sequential batching. This may be due to random batching resulting in more consistent batch distributions, which allows the least square optimizer to converge at a similar rate.

Batch Size

The results from Figure 3.15 suggest that the number of parameters of the least square optimizer has a big impact on the performance. A 256 batch size reduces the accuracy by around 13 units compared to training without batching, which has an accuracy of 86.68% (shown in Figure 3.14). As the batch size is approximately one-tenth of the size of the train set, it is likely that the batch is not representative. Additionally, recall that in the current architecture of sampled PointNet, the last layer prior to the least square layer has a width of 256. Therefore, a 256 batch size is possible to result in any of the under-determined, well-determined, or over-determined system.

For 1024 and 2048 batch sizes, the inputs are more likely to form an over-determined system with only 256 features. Furthermore, both batch sizes are large enough to enable good representation of the train set. Thus, their performances are only slightly reduced compared to that of training without batching (86.68%), and the discrepancy in accuracy between the two sizes is insignificant. Having addressed the computational memory demand for SWIM to train on large datasets, we proceed to investigate whether increasing the number of iterations would lead to improved predictive power.

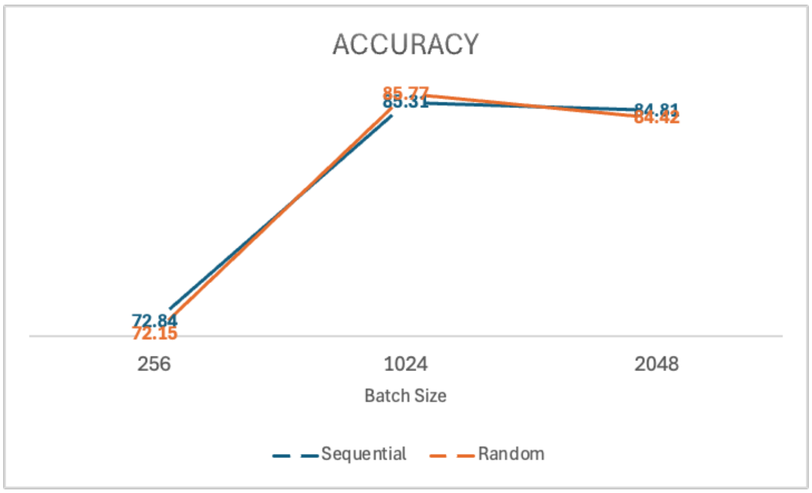


Figure 3.15: The accuracy for the batch sizes of {256, 1024, 2048} with 512-point point clouds and {10, 2, 1} iteration(s). The reported results are the average over 5 different seeds.

Number of Iterations

N. Iter.	Time Taken (s) ⇓	Accuracy (%) ⇑	AUCROC (%) ⇑
2	$\mathbf{98.27 \pm 13.61}$	85.77 ± 1.39	$97.05 \pm 3.89\text{e}{-1}$
5	133.33 ± 2.53	$85.69 \pm 7.66\text{e}{-1}$	$96.89 \pm 5.47\text{e}{-1}$
10	242.26 ± 16.18	$\mathbf{86.39 \pm 1.35}$	$97.12 \pm 5.97\text{e}{-1}$
50	1446.99 ± 16.54	85.75 ± 1.41	$\mathbf{97.18 \pm 5.76\text{e}{-1}}$

Table 3.12: This table illustrates the performance in relation to the number of iterations {2, 5, 10, 50} on randomly batched ModelNet10 with 1024 batch size, 512 point cloud density. The experiment for each distribution were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better).

In the absence of feedback, having more chances for the sampled PointNet's weight choices does not necessarily lead to significantly more good decisions, even with the assistance of sampling distributions. Our results from Table 3.12 confirm this conjecture, as we observe four indistinguishable performances across the chosen number of iterations: 2, 5, 10, and 20. Note that the table shows the results of random batching, therefore we cannot ensure that the model processes through the entire dataset. Based on our previous findings in last experiment with batch sizes and Section 3.3.1, a batch size of 1024 with a point density of 512 should enable good representations on both point cloud and point level, therefore we can rule out the possibility of information deficits as the primary cause.

To further investigate this phenomenon, we analyze it under two scenarios: flat and sharp sampling distributions on point cloud level. As previously noted, on point cloud level, the sampled PointNet uses the original SWIM distribution, which prioritizes close proximity between input pairs under classification settings. In the case of a flat sampling distribution, the differences among the distances of input pairs are overshadowed by the number of input data, leading to small sampling probabilities for each input pair. This is equivalent to randomly sampling new weights for the updates, which can no longer guarantee improvements. This hypothetical scenario highlights the need of implementing a function similar in spirit to the quantile filter on point level, as proposed in Section 3.4.2, to sample the good pairs.

In the latter scenario, the sharp peaks can serve as heuristics to guide the weight updates. However, regardless of the batching method, no mechanism prevents the sampled PointNet from repeatedly choosing the same input pairs (or the pairs from the same two classes). As a result, the weights become increasingly monotonous, leading to a similar problem, as discussed in Section 3.4.3, where the weights under-represent the data distribution due to the lack of diversity. This observation suggests that a gradient diversification approach, similar to the one introduced in Section 3.4.3 at the point level, could be a promising direction for future work.

3.5.4 Discussion

With discrete online learning, SWIM can now process large datasets. Given a large enough batch size, discrete online learning yields comparable performance with a much smaller memory footprint. For a sampled PointNet, the empirical results suggest that the batch size has relatively more influence on the least square optimizer than the Dense layers and their variants. The input size should allow the least square optimizer to solve an over-determined linear system. Additionally, the input should be representative.

At the current stage, having more iterations does not enable better performance, despite multiple encounters with the datasets. We acknowledge that due to the fundamental limitations of SWIM, it is difficult to for discrete online learning to strike the optimum, unlike the conventional online learning for iterative gradient method. We speculate that this situation can be improved by using similar strategies as in Section 3.4: selecting more diverse good pairs. However, defining good pairs and their diversity on the object level becomes more challenging.

One feasible direction is to re-implement the strategies on a point level on the object level. Instead of computing the distance matrix of the selected source points, we can compute the point cloud distances among the selected source point clouds. However, this approach would require additional point cloud direction computations, potentially slowing down the runtime. Another potential future work is to design a scalar signature that encodes context, including point cloud direction and corresponding class pairs between two point clouds. With the scalar signatures, we can perform lightweight sampling to ensure that all class pairs are well represented in the weight constructions. Nevertheless, after determining how to design our inputs with batching, we turn our attention to studying the optimal architecture of sampled PointNet for ModelNet.

3.6 Architecture Optimality

Neural network architecture optimization is a crucial aspect of developing effective machine learning models. The process involves determining the ideal structure and configuration of a neural network to achieve optimal performance on a given task. This includes decisions about the number of layers (depth), the number of neurons in each layer (width), the types of layers to use, and the choice of activation functions. Finding the optimal architecture is often a complex and iterative process that requires a combination of domain knowledge, experimentation, and sometimes automated techniques. The goal is to strike a balance between model complexity and performance. In previous discussions, we have mentioned on several occasions that some parts of the original PointNet architecture are designed to leverage iterative gradient methods, and thus should not be included in sampled PointNet. However, the architecture of sampled PointNet should not be too distinct from PointNet for the sake of fair comparisons.

3.6.1 Results

To preserve the originality of PointNet, we only consider changing the depth and width of the shared DenseR layers, which constitute the encoder, and the Dense layers, which are the decoder. A MaxPool layer will be inserted after the encoder, while a Linear layer with least square optimizer will follow either the MaxPool layer, or the decoder if available. We start by finding the depth configurations which give rise to the best performance. Then, using the best configurations, we investigate whether the model capacity in terms of layer width or the amount of information embedded in the inputs are the limitations. Finally, we question the use of mean function as the weight aggregation strategy for shared Dense layers.

Depth

From the perspective of depth, we investigate three key factors: the number of layers in the encoder, the impact of the decoder, and the number of layers in the decoder. For the encoder, we follow the architecture proposed by Qi et al. [2017a], which consists of at most five layers of shared DenseR: [64, 64, 64, 128, 1024]. For simplicity, we respect the sequential order of the architecture. In particular, we do not randomly omit a middle layer such as [64, 64, 1024]. The layer combinations of encoder are chosen to assess the functionality of first three layers with 64 neurons, and the effects of having differnet widths as last layer of the encoder. As Qi et al. [2017a] mention, the decoder can be any classifier. The original design uses three Fully-Connected layers (including the output layer) as the classifier. We investigate the influence of this designated depth using two Dense layers and one least square layer. Furthermore, we also attempt to justify the necessity of including two Dense layers prior to the least square optimizer.

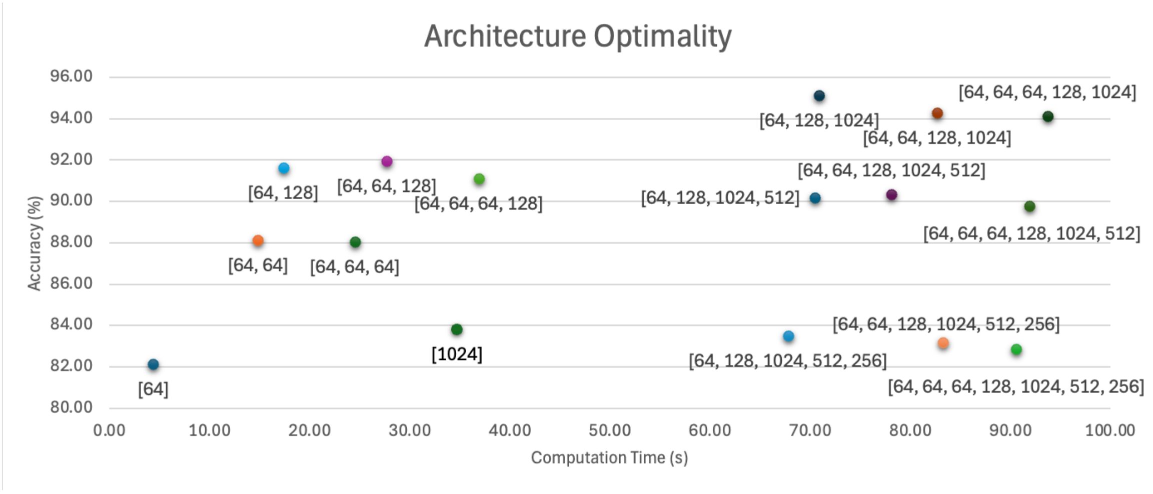


Figure 3.16: The figure shows the computation time and accuracy for different architecture with 512-point density. The reported results are the average over 5 different seeds.

From Table 3.16, we are surprised to see that only one shared DenseR layer with 64 neurons can already achieve a decent accuracy of 80%, despite adding more similar layers resulting in a big boost to 88% accuracy at the expense of slightly longer computation time. However, for the remaining combinations, we can notice that having more than one 64-width layers — for example, [64,128], [64,64,128], [64,64,64,128] — does not significantly help the predictions and may slightly worsen the accuracy. This reflects that one layer with 64 neurons is likely underfitted due to too few parameters.

It is interesting to observe that, although [64,128] and [64,64,64] share the same number of parameters for the encoder, the former has more parameters in its least square optimizer, resulting in 4 units more accurate than the latter. This suggests that the capacity of the least square optimizer plays an important role in getting good predictions. The remaining configurations, grouped by the width of the last layer, affirm that the number of least square's parameters are the dominant factor in relation to predictive power. The [1024] configuration of shared DenseR layer is specifically tested to assess whether large number of least square's parameters with a non-linear transformation brought by the shared DenseR layer is sufficient to achieve best performance. Since the accuracy is only marginally better than that with 64 neurons, this indicates that a certain level of depth is still required to achieve best performance.

Out of all configurations, the groups with a 1024 width as the last layer of the encoders have the best performances at around 94% to 95%, which is only three to four units of accuracy behind the state-of-the-art, with a training time of slightly beyond one minute. This can be attributed to the enhanced modelling capacity of the least square optimizer with two to four times the parameters. Compared to the original sampled PointNet architecture [64, 128, 1024, 512, 256], the improvement of [64, 128, 1024] is more than 10 units of accuracy. It is also interesting to find that the former with 799,370 parameters has shorter computation time than the latter (150,922 parameters), despite the additional parameters of the decoder. This indicates that the least square optimizer has more expensive run time complexity compared to the sampling process and the matrix multiplications. However, the stark difference in the number of parameters suggests that the encoder-only architecture is more efficient in terms of space complexity.

In general, the addition of Dense layers — with width designs of [512] and [512, 256] — as a decoder does not show a positive contribution. The performances are similar to those encoders with small last layer widths, despite two to ten times the computation time. We can conclude that sampled PointNet cannot excel with deep architectures, which is one of the limitations of SWIM as reported by Bolager et al. [2023]. One plausible reason is that the current sampling mechanism cannot harvest information fully during the forward pass of inference. This problem is akin to the gradient vanishing or exploding challenges of iterative gradient methods, but in the reverse direction. Therefore, it is worth investigating the feasibility of adapting specific remedies, such as skip connections, from iterative gradient methods.

Width and Density

To evaluate the impact of width, we use the depth configurations — grouped by the width of the last layer — which have the best performance, that is, [64,128,1024], [64, 64, 128, 1024], and [64, 64, 64, 128, 1024]. Then, we double the parameters for each layer. We also conduct separate runs to inspect whether the size of the model parameters has been a limitation to capture more information. For these runs, we increase the density of point clouds from 512 points to 1024 points.

In Figure 3.17 (note that it is on a different scale compared to Figure 3.16), the results reflect that having denser point cloud from 512-point to 1024-point can marginally improve the performance. Deeper architectures have relatively larger improvements, which are however under one unit of accuracy. Using point clouds with 1024 density, doubling the width of [64, 128, 1024] configuration lead to a slight improvement of roughly 0.15 unit of accuracy. However, the same treatment on the deeper architectures deteriorate the prediction power. It could be that deeper architecture with doubled parameters overfitted to the train set.

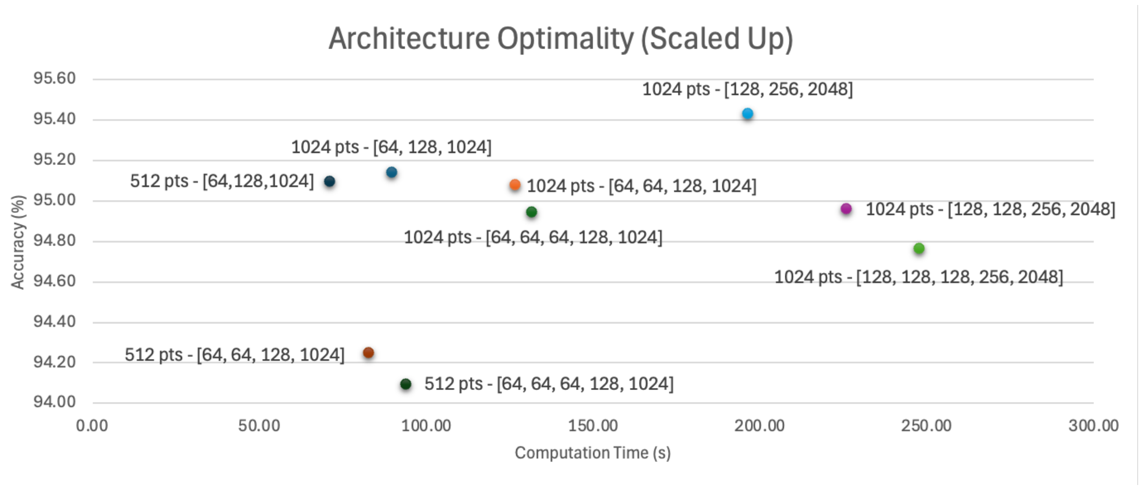


Figure 3.17: The figure shows the computation time and accuracy for different architecture with scaled up density and parameters. The reported results are the average over 5 different seeds.

Weight Aggregation

For the experiments thus far, we have been using the average of weights and biases at point level as the strategy to implement the concept of parameter-sharing. However, this setup might not be optimal. To assess the optimality of weight aggregation strategy, we train sampled PointNets using another two aggregation strategies, namely maximum and median. Maximum is a popular symmetric function, which is employed in Hausdorff distance (Equation 2.5) and in PointNet as a pooling layer. Median is a rank-based statistic which is more robust against outliers and often used as an alternative of mean.

Strategy	Time Taken (s) ⇓	Accuracy (%) ⇑	AUCROC (%) ⇑
Mean	70.92 ± 2.93	$\mathbf{95.10 \pm 7.74e{-2}}$	$\mathbf{99.37 \pm 0.11e{-1}}$
Max	$\mathbf{52.61 \pm 2.28}$	$55.74 \pm 1.56e{-1}$	$84.73 \pm 2.57e{-2}$
Median	55.48 ± 1.27	$93.88 \pm 6.68e{-1}$	$99.11 \pm 1.24e{-1}$

Table 3.13: This table illustrates the performance in relation to different weight aggregation strategies on ModelNet10 with 512 point cloud density and [64, 128, 1024] architecture. The experiment for each distribution were repeated for 5 different seeds. The reported values are *mean±standard deviation* of 5 results. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better).

Table 3.13 shows that using mean function as the aggregation strategy for shared DenseR layers can lead to superior performance compared to maximum and median. Maximum has the worst performance out of all strategies. In the context of SWIM, a maximum function extracts the largest difference in every dimension between two point clouds, which could undermine the latent representation, since SWIM relies on the directions with close proximity. For using median as the aggregation strategy, the sampled PointNet has a slightly worse performance which lies out of the standard deviations of the accuracy of using mean. This reveals that the underlying distributions are *not* normal distribution (otherwise the median is also the mean). On top of that, it shows that using the rank-based statistic would result in under-representation compared to that of mean, which considers all points. However, we argue that the discrepancy of performance between using mean or median will vary depends on point cloud density.

3.6.2 Discussion

It is important to remember that the optimal architecture shown in this section is only optimized against ModelNet. Our empirical findings suggest that sampled PointNet should not blindly follow the original architecture of PointNet. SWIM requires a good depth to perform well. For every layer, SWIM constructs weights which establish a new sampling space for next layer. However, a deep architecture might result in corrupted sampling spaces — akin to gradient vanising or exploding for iterative gradient methods — which lead to bad performance. It is certainly an interesting future work to counter these effects.

The width of a layer determines the number of placeholders, which set the upper bound of the capacity of layer to encode information. With more placeholders, a Dense layer has more capacity to contain diverse directions, but could also lead to overfitting. Lastly, weight sharing mechanism plays an important role in enhancing the efficiency of a neural network. Unlike iterative gradient methods, SWIM cannot fine tune the shared weights through aggregated gradient feedback. Therefore, using basic weight aggregation strategy is a provisional approach to move forward. In future, a more sophisticated strategy should be developed to better encode the rich information between two point clouds using one vector. With the optimal architecture, we can finally evaluate our work thus far on the standard dataset, ModelNet40.

3.7 ModelNet40

ModelNet40, curated by Wu et al. [2015], is a widely used benchmark for 3D objects classification. It includes 12311 CAD models from 40 categories that are split into 9843 for training and 2468 for testing. Since ModelNet40 is the superset of ModelNet10, for all experiments thus far, we evaluated only on the validation set, which is split from the train set of ModelNet10 to prevent data leakage. Since we now have our optimized architecture and input configurations, we no longer need to extract a validation set from the train set of ModelNet40. We train on all 9843 samples, and measure the performance on 2468 test samples in terms of computation time and accuracy.

We create the point cloud dataset from ModelNet40 with density of 512 points. The points are sampled from the surface of the meshes proportionally to the surface area. Additionally, the train set is augmented by 3 times with random rotations to evaluate rotational robustness of sampled PointNet on ModelNet40. The chosen configuration of point cloud density and data augmentation has the best accuracy-efficiency tradeoff on ModelNet10, as investigated in Section 3.3.1 and Section 3.3.2. These points are then centralized and normalized into an unit sphere. For sampled PointNet, we use DenseR with Sampling Diverse Good Pairs and the other common components to implement the architecture of [64, 128, 1024].

Given that the entire train set can be accommodated within the computational memory, and batching does not bring further advantages at this stage, we opt to train two sampled PointNets without batching. The first model uses standard inputs, and its performance is evaluated directly on the test set. For the second model, we train with 3-augmentations rotated inputs and obtain the majority predictions of the test set across 12 fixed rotation angles, as proposed by Qi et al. [2017a] and implemented in Section 3.2.4. To differentiate between these models, we refer to the first as the Sampled PointNet, and the second as the Sampled PointNet (Rotation) in Table 3.14.

To assess the efficacy of our work, we include a comparison of the computation time and accuracy of several models: PointNet, PointNet (Rotation), and the current state-of-the-art, RotationNet [Kanezaki et al., 2019]. Note that the original authors of PointNet [Qi et al., 2017a] only report the accuracy of PointNet — which was trained with rotated inputs — on the 12-rotation evaluation, therefore we temporarily name it as PointNet (Rotation). We obtain the accuracy of PointNet on standard inputs and direct evaluation from Yan [2019], and refer this result as PointNet. For RotationNet, it was trained and tested on multi-view images of ModelNet40.

3.7.1 Results

Models	Nr. Pts	Time Taken $\Downarrow$	Accuracy (%) $\Uparrow$
RotationNet	N/A	$3-5$ days	$\mathbf{97.37}$
PointNet (Rotation)	1024	$3-6$ hours	89.20
Sampled PointNet (Rotation)	512	719.71 ± 11.84 seconds	$65.89 \pm 9.44\text{e}{-1}$
PointNet	1024	$3-6$ hours	90.60
Sampled PointNet	512	$\mathbf{185.65} \pm \mathbf{1.63}$ seconds	$84.35 \pm 3.67\text{e}{-1}$

Table 3.14: This table illustrates the performance of sampled PointNet on ModelNet40 with 512 points per point cloud. **Nr. Pts** stands for number of points, which is not applicable to RotationNet, as it adopts an imaged-based approach. The reported values for sample PointNet and sampled PointNet (Rotation) are *mean±standard deviation* of 5 results using 5 different seed. Bold faces represent the best results of corresponding columns (differentiated by standard deviation in case of ties, the lower the better).

Table 3.14 shows the evaluation results on ModelNet40. Sampled PointNet achieves comparable accuracy compared to PointNet using inputs with standard orientation — at most 6.25 units drop in accruracy, within the boundaries of 10% performance loss. The performance of sampled PointNet (Rotation) against rotated inputs is lackluster, with a 23.31 units accuracy gap behind PointNet (Rotation). Compared to the state-of-the-art RotationNet, sampled PointNet (Rotation) and PointNet (Rotation) have plenty of rooms for improvement. It is worth remembering that RotationNet uses multi-view images instead of point clouds, therefore the column that indicates number of points (Nr. Pts) is not applicable.

In terms of computation time, we consider the longer training time of sampled PointNet (Rotation) on a CPU, which is at most 13 minutes. This time spent is 7.72% and 0.3% of what it takes to train PointNet (as well as PointNet(Rotation)) and RotationNet respectively on a single GPU. This means that we have achieved our objective of using only 10% computation time of PointNet, which ranges from 18 minutes to 36 minutes.

3.7.2 Discussion

The evaluation suggests that the Sampled PointNet is not perfect, and it's only comparable to PointNet under the standard orientation setting. It's worth noting that, as reported by Kanezaki et al. [2019], RotationNet was not trained from scratch, but fine-tuned based on selected pre-trained models, which have access to extra datasets. The difference in the number of parameters across all models is significant: RotationNet has 11.6M, 24.2M, 60.2M, and 102.2M parameters, depending on the pre-trained models. In contrast, PointNet with two T-Nets has 3.5M parameters, and the sampled PointNets has approximately 0.15M parameters. Therefore, Sampled PointNet is the most efficient approach in terms of predictive power, without considering rotational invariance. The weaker performance against rotated inputs could be attributed to the fundamental limitations of SWIM. Given the success of RotationNet, it would be interesting to develop a Sampled PointNet based on pre-trained models.

4 Conclusion

In this thesis, we train PointNet using a state-of-the-art weight construction technique, namely Sampling Where It Matters (SWIM), instead of iterative gradient methods. We design and implement the product, sample PointNet, with the following objectives in mind:

1. 18 minutes to 36 minutes training time using ModelNet40,
2. Sample point cloud pairs with steep gradient,
3. Permutation invariance,
4. Rotational invariance.

In particular, we aim to improve the computation time, accepting a potential 10% drop in accuracy compared to PointNet. In summary, we have successfully achieved three out of four objectives. Our success with the second and third objectives has rectified the performance drop to 6.90% (achieving 84.35% accuracy) using standard inputs. However, our inability to conquer rotational invariance has resulted in a 26.13% reduction in accuracy using rotated inputs.

We achieve the second and third objectives in Section 3.1 using KDTree to compute the point cloud directions. Then, we attempt to preserve rotational invariance using various approaches — T-Net, data augmentation, Spherical coordinates, scaling up density and augmentations — in Section 3.2, Section 3.3.2, and Section 3.3.3, but unfortunately, to no avail. During the process, we are aware that the time complexity brought by KDTree needs to be further optimized in order to achieve the first objective. We propose a scalable approach, namely recursive sampling in Section 3.4. Optimizing this approach reveals the importance of sampling diverse good input pairs on point level. Despite not being part of the objectives, we also equip SWIM with discrete online learning in Section 3.5 to process large datasets with controllable computational memory, unlocking the feasibility of training sampled neural networks using portable devices. Lastly, before we evaluate sampled PointNet in Section 3.7, we optimize the architecture of sampled PointNet against ModelNet10 in Section 3.6. The final evaluation shows that we over-achieve the first objective with four minutes training time using regular inputs, and 13 minutes on the augmented dataset, which is three times larger in size.

With these encouraging results, the journey should not stop here. Compared to the state-of-the-art in deep learning, there are many opportunities for sampled PointNet to improve. While it is impossible to exhaust the list of potential improvements here, we highlight two which are the most relevant to our work: a mechanism should be designed to sample diverse good pairs on the object level (we did it on the point level only); the weights and biases should become a more comprehensive representation — which can encode the more than one direction meaningfully — to fully utilize the advantage of data augmentation and online learning. This may allow SWIM to fundamentally advance beyond the discrete optimization landscape, which is constrained by the input combinations.

Bibliography

Erik Lien Bolager, Iryna Burak, Chinmay Datar, Qing Sun, and Felix Dietrich. Sampling weights of deep neural networks. 2023.

Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Veličković. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges, May 2021. URL http://arxiv.org/abs/2104.13478. arXiv:2104.13478 [cs, stat].

Rickard Brüel Gabrielsson. Universal function approximation on graphs. *Advances in neural information processing systems*, 33:19762–19772, 2020.

Ronald R Coifman and Stéphane Lafon. Diffusion maps. *Applied and computational harmonic analysis*, 21(1):5–30, 2006.

Dawson-Haggerty et al. trimesh. URL https://trimesh.org/.

Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

Evangelos Galaris, Gianluca Fabiani, Ioannis Gallos, Ioannis Kevrekidis, and Constantinos Siettos. Numerical Bifurcation Analysis of PDEs From Lattice Boltzmann Model Simulations: a Parsimonious Machine Learning Approach. *Journal of Scientific Computing*, 92(2):34, June 2022. ISSN 1573-7691. doi: 10.1007/s10915-022-01883-y. URL https://doi.org/10.1007/s10915-022-01883-y.

Raja Giryes, Guillermo Sapiro, and Alex M Bronstein. Deep neural networks with random gaussian weights: A universal classification strategy? *IEEE Transactions on Signal Processing*, 64(13):3444–3457, 2016.

Ian Goodfellow. Deep learning, 2016.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. *arXiv preprint arXiv:1712.00409*, 2017.

Long Hoang, Suk-Hwan Lee, Oh-Heum Kwon, and Ki-Ryong Kwon. A deep learning method for 3d object classification using the wave kernel signature and a center point of the 3d-triangle mesh. *Electronics*, 8(10):1196, 2019.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989.

Hristo Hristov. Introduction to k-d trees. https://www.baeldung.com/cs/k-d-trees, 2023. Accessed: 2024-09-12.

Guang-Bin Huang, Qin-Yu Zhu, and Chee-Kheong Siew. Extreme learning machine: theory and applications. *Neurocomputing*, 70(1-3):489–501, 2006.

William B Johnson and Joram Lindenstrauss. Basic concepts in the geometry of banach spaces. In *Handbook of the geometry of Banach spaces*, volume 1, pages 1–84. Elsevier, 2001.

Asako Kanezaki, Yasuyuki Matsushita, and Yoshifumi Nishida. Rotationnet for joint object categorization and unsupervised pose estimation from multi-view images. *IEEE transactions on pattern analysis and machine intelligence*, 43(1):269–283, 2019.

Miroslav Kubat, Stan Matwin, et al. Addressing the curse of imbalanced training sets: one-sided selection. In *Icml*, volume 97, page 179. Citeseer, 1997.

Itai Lang, Asaf Manor, and Shai Avidan. Samplenet: Differentiable point cloud sampling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7578–7588, 2020.

Léo Lebrat, Rodrigo Santa Cruz, Clinton Fookes, and Olivier Salvado. MongeNet: Efficient Sampler for Geometric Deep Learning, April 2021. URL http://arxiv.org/abs/2104.14554. arXiv:2104.14554 [cs].

Daniel Lehmberg, Felix Dietrich, Gerta Köster, and Hans-Joachim Bungartz. datafold: data-driven models for point clouds and time series on manifolds. *Journal of Open Source Software*, 5(51):2283, 2020. doi: 10.21105/joss.02283. URL https://doi.org/10.21105/joss.02283.

Feiran Li, Kent Fujiwara, Fumio Okura, and Yasuyuki Matsushita. A closer look at rotation-invariant deep point cloud analysis. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 16218–16227, 2021.

Fangzhou Lin, Yun Yue, Songlin Hou, Xuechu Yu, Yajun Xu, Kazunori D Yamada, and Ziming Zhang. Hyperbolic chamfer distance for point cloud completion. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 14595–14606, October 2023.

Daniel Maturana and Sebastian Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 922–928. IEEE, 2015.

Antonio Montanaro, Diego Valsesia, and Enrico Magli. Rethinking the compositionality of point clouds through regularization in the hyperbolic space. *Advances in Neural Information Processing Systems*, 35:33741–33753, 2022.

Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

Pierre Onghena, Leonardo Gigli, and Santiago Velasco-Forero. Rotation-invariant hierarchical segmentation on poincare ball for 3d point cloud. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV) Workshops*, pages 1765–1774, October 2023.

Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

Charles R. Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation, April 2017a. URL `http://arxiv.org/abs/1612.00593`. arXiv:1612.00593 [cs].

Charles Ruizhongtai Qi, Li Yi, Hao Su, and Leonidas J Guibas. Pointnet++: Deep hierarchical feature learning on point sets in a metric space. *Advances in neural information processing systems*, 30, 2017b.

Ali Rahimi and Benjamin Recht. Uniform approximation of functions with random bases. In *2008 46th annual allerton conference on communication, control, and computing*, pages 555–561. IEEE, 2008.

Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. *science*, 290(5500):2323–2326, 2000.

Frederic Sala, Chris De Sa, Albert Gu, and Christopher Ré. Representation tradeoffs for hyperbolic embeddings. In *International conference on machine learning*, pages 4460–4469. PMLR, 2018.

Anton Maximilian Schäfer and Hans Georg Zimmermann. Recurrent neural networks are universal approximators. In *Artificial Neural Networks–ICANN 2006: 16th International Conference, Athens, Greece, September 10-14, 2006. Proceedings, Part I 16*, pages 632–640. Springer, 2006.

Bernhard Schölkopf, Alexander Smola, and Klaus-Robert Müller. Kernel principal component analysis. In *International conference on artificial neural networks*, pages 583–588. Springer, 1997.

Hang Su, Subhransu Maji, Evangelos Kalogerakis, and Erik Learned-Miller. Multi-view convolutional neural networks for 3d shape recognition. In *Proceedings of the IEEE international conference on computer vision*, pages 945–953, 2015.

Joshua B Tenenbaum, Vin de Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Wikipedia. Neural network (machine learning). https://en.wikipedia.org/wiki/Neural_network_(machine_learning), 2024. Accessed: 2024-09-11.

Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3d shapenets: A deep representation for volumetric shapes. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1912–1920, 2015.

Xu Yan. Pointnet/pointnet++ pytorch. https://github.com/yanx27/Pointnet_Pointnet2_pytorch, 2019.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep Sets. In *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/hash/f22e4747da1aa27e363d86d40ff442fe-Abstract.html.

Chen Zhao, Jiaqi Yang, Xin Xiong, Angfan Zhu, Zhiguo Cao, and Xin Li. Rotation invariant point cloud analysis: Where local geometry meets global topology. *Pattern Recognition*, 127:108626, 2022.

Ding-Xuan Zhou. Universality of deep convolutional neural networks. *Applied and computational harmonic analysis*, 48(2):787–794, 2020.