University of Bergen
Department of Informatics

Vectorizing Distributed Homology with Deep Set of Set Networks

Author: Sigurd Roll Solberg
Supervisors: Nello Blaser

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

June, 2023

Abstract

Distributed homology, a topological invariant, holds potential as an instrument for uncovering insights into the structural characteristics of complex data. By considering both the density and connectivity of topological spaces, it offers the potential for a more detailed and stable understanding of the underlying structure of data sets. This is particularly beneficial when confronting noisy, real-world data.

Despite its potential, the complexity and unstructured nature of distributed homology pose hurdles for practical use. This thesis tackles these issues by proposing a novel pipeline that fuses distributed homology and supervised learning techniques. The goal is to facilitate the effective incorporation of distributed homology into a wide array of supervised learning tasks.

Our approach is anchored on the DeepSet network, an architecture adept at managing set inputs. Using this, we devise a comprehensive framework specifically designed to handle inputs composed of a set of sets. Furthermore, we present a dedicated architecture for distributed homology, designed to boost robustness to noise and overall performance. This approach shows marked improvements over full persistent homology methods for both synthetic and real data.

While our results may not yet rival state-of-the-art performance on real data, they demonstrate the potential for distributed invariants to enhance the efficiency of topological approaches. This indicates a promising avenue for future research and development, contributing to the refinement of topological data analysis.

Acknowledgements

First and foremost I would like to thank my supervisor Nello Blaser for his great ideas and consistent guidance throughout. You have provided me with invaluable insights and detailed explanations in addition to much appreciated help structuring the work. I would also like to thank Audun Ljone Henriksen, Morten Blørstad, Thorarinn Gunnarsson and the rest my classmates for many great discussions and for providing me with the motivation to keep working. Special thanks to Herman Jangsett Mostein for years of great discourse and collaboration as well as helpful feedback.

I extend my gratitude to my friends and my family as well. In particular, I thank my two sisters for their continued support and encouragement throughout this project, and my parents for inspiring me to embark on it in the first place. Finally I want to give a big thanks to my girlfriend for her love and patience, without whom this journey would not have been possible.

Sigurd Roll Solberg
Wednesday 14th June, 2023

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The basic assumption that motivates the field of Topological Data Analysis (TDA) is that data has shape and that this shape matters [13]. Based on this assumption, identifying descriptive quantities of the shape of a space can be a way to detect relevant, low-dimensional features of potentially high-dimensional spaces. TDA provides with methods of engineering the features of a dimensionality reduction. The relevant domain of these features ranges from geometric ones such as angles and volumes to topological features such as connectivity, holes, and cavities. In the analysis of a space and its structure, we want to identify properties that can allow us to distinguish between distinct spaces, but still remains stable to deformation, hence allowing smooth similarity measures between these structures.

Topological invariants are properties of a space that can be expressed in terms of its topology. This implies their invariance to continuous deformation and provides us with the desired stability. Recently the field of topological data analysis has been heavily focused on the computation or estimation of the full persistent homology [31] of spaces, and the representation of this invariant through persistence diagrams. Some of the most commonly used algorithms for these estimations are extremely computationally expensive [16] and can be sensitive to noise and outliers. As such, estimates of the full persistent homology may not be the most suitable invariant for all data. In [31] it was shown that a collection of persistence diagrams from subsets of a space can be a preferable invariant in many cases. We call this collection distributed persistence. This collection has been

proven to contain relevant information about the shape of a space for some specific cases [31], and in some cases even strictly more than the full persistent homology. However, in general, it is not yet clear exactly what information it embodies.

Machine learning provides a framework for utilizing such hidden information by approximating relations between a vector space and explicitly defined features of its elements. Especially deep learning excels, compared to other approaches, at solving complex, high-dimensional problems due to its ability to learn how to do both feature extraction and feature engineering from its input. Because of these properties, deep learning is also particularly suited for learning tasks on unstructured data such as distributed persistence. PersLay [14] is an established deep-learning approach for facilitating the use of machine learning on persistence diagrams.

In combination, deep learning with PersLay, and distributed persistence may provide both powerful, descriptive features for classification and regression tasks. Building on this foundation, this thesis aims to develop a new approach that leverages the potential of distributed persistence within a deep learning context. By integrating these two elements, we hope to overcome some of the computational and noise sensitivity challenges associated with current methods.

1.2 Objective

The primary goal of this thesis is to develop a novel pipeline that leverages the strengths of distributed homology as a topological invariant and combines it with the power of supervised learning techniques. This pipeline aims to address the challenges posed by the unstructured and complex nature of distributed homology and enable its effective utilization in practical applications. Specifically, the objective is to integrate distributed homology into the DeepSets framework, a deep learning approach designed for handling set inputs. By doing so, we seek to capture the intricate structure and latent information contained within the family of sets that make up the distributed homology.

The key objectives can be summarized as follows:

- Develop a deeper understanding of distributed homology as a topological invariant in practical applications and its potential for providing insights into the topological characteristics of data.

- Establish a framework for handling set of set inputs in supervised machine learning, with a strong theoretical backing to ensure reliable performance and applicability.
- Employ this newly constructed framework to integrate distributed homology effectively into practical applications.
- Evaluate the performance and effectiveness of the proposed pipeline on various supervised learning tasks, demonstrating its ability to reveal intricate patterns and relationships in geometric data sets.

1.3 Thesis Outline

The outline for the rest of this thesis is as follows:

Chapter 2 - Background Theory introduces the necessary background in machine learning, topology, and topological data analysis for this thesis.

Çhapter 3 - Related Work describes the current status of the field and prior research on related topics.

Chapter 4 - Method presents our proposed deep learning framework for integrating distributed topological information along with details on the implementation.

Chapter 5 - Experiment describes the specifics of how our experiments were conducted. Results are also presented here.

Chapter 6 - Discussion reviews the results and analyzes the findings of the experiments. Additionally, we evaluate our approach and make concluding remarks.

Chapter 7 - Future Work outlines ideas for future improvements and experiments.

Chapter 2

Background Theory

2.1 Machine Learning and Deep Neural Networks

The field of machine learning focuses on creating algorithms and statistical models that empower computer systems to extract meaningful insights from data. Different optimization methods are used to learn how to perform tasks such as classification, regression, or dimensionality reduction. The field can be broadly divided into three categories: unsupervised, supervised, and reinforcement learning. In this thesis, the two former will be of interest.

Unsupervised learning involves extracting information from data without prior knowledge of what information to look for. This type of learning focuses on uncovering hidden patterns and structures in unlabeled data. Common learning objectives include dimensionality reduction, clustering, and anomaly detection. In contrast, supervised learning aims to extract information about the relationship between observation data and a specific set of outcomes or features. This is achieved by providing labeled examples that demonstrate the associations between the observations and their properties of interest. The model tries to learn the underlying relationship between these two spaces.

Machine learning has numerous applications across various fields, and holds state-of-the-art solutions for many problems including computer vision, natural language processing, weather forecast, and healthcare. It has enabled the development of intelligent systems that can automate decision-making processes and make predictions on complex problems that were previously thought to require human intelligence.

In the subsequent section, we will introduce supervised learning, with a specific emphasis on deep learning, discussing how these methods can be applied within topological data analysis to address persistent challenges and provide potent tools for data analysis.

2.1.1 Supervised Learning

As mentioned, supervised learning is a method for relating observations to specific properties of those observations. More formally, the task can be formulated as finding a mapping $f : X \longrightarrow Y$ from an input space X (observation space) to an output space Y with features of particular interest. For example, in image classification, we can make observations in the space of all images (input space X in this example) and try to find a mapping to the space $\{True, False\}$ (output space Y in this example) based on whether or not each image contains a cat or not. The underlying assumption is that such a function exists and that it defines the correspondence between elements of X and Y. We call this function $\hat{f}$, **the labeling function**, and its outputs simply the ground truth. The goal of the learning process is ideally to uncover this function. However, finding $\hat{f}$ using conventional methods might prove difficult in many cases, but supervised learning provides methods for approximating $\hat{f}$, by leveraging collected samples of pairs $(x, \hat{f}(x))$ where $x \in X$ and $\hat{f}(x) \in Y$, often referred to as **experience**.

Supervised learning assumes that we can approximate $\hat{f}$ and find a solution to our problem by searching a space of candidate functions F, to find a function f that provides predictions for the samples in the provided experience. Evaluation of f in this search is based on a scalar-valued, non-negative function l, called a **loss-function**. Given a labeled sample (x_i, y_i) where $y_i = \hat{f}(x_i)$, this function measures the error in the prediction $\hat{y_i} = f(x_i)$ and the provided label y_i. We can more formally express the problem of supervised learning as finding the function f^* that minimizes the expected error over the distribution of all pairs $(x, y) \sim D$:

$$f^* = \arg\min_{f \in F} \mathbf{E}_{\sim \mathbf{D}}[l(f(x), y)]. \tag{2.1}$$

Ideally, our objective is to solve equation 2.1, which requires complete information about the distribution D. However, in practice, it is rare to have access to the entirety of the distribution D, making it difficult to obtain an exact solution. But, assuming that

we are provided n points (x_i, y_i) sampled independently from D, we can empirically infer an approximated solution to eq. 2.1 by averaging the loss over our experience.

$$f^* \approx \arg\min_{f \in F} \frac{1}{n} \sum_{i=1}^{n} l(f(x_i), y_i) \tag{2.2}$$

It's important to note that provided a sufficiently complex space F and a finite number of samples, finding a function f that offers an optimal solution, that is $\sum_{i=1}^{n} l(f(x_i), y_i) = 0$ is always possible. At the same time, we are not concerned with the function's ability to correctly map the samples provided, seeing as their labels are already known. The real goal is for the model to **generalize** its performance to new, unseen data, specifically the rest of the space X. To evaluate this property of the candidate functions, we make an approximation based on their performance on a separate set of samples that the model has not yet seen. From this point on, we refer to the data used by a machine learning algorithm for searching the function space F as the **training data** and the data used to evaluate the generalized performance of the candidate function f provided by this search as the **validation data**. Commonly, validation data is utilized to contrast different model types following their training. However, this added selection stage can introduce bias towards the specific data present in the validation set. To mitigate this, it is common to employ a third set of samples to provide an unbiased approximation of the model's generalized performance. This independent data subset is referred to as the **test data**. When a model yields significantly better performance on the training data than the test data, in other words, the generalized performance is much worse than the performance during training, we say that our model is **overfitting**. When both the performance and the generalized performance of the model is bad, we say the model is **underfitting**. Note that there are no exact bounds for when a model is over- or underfitting, and they both are subjective and very situational terms.

2.1.2 Deep Learning and Gradient Decent

Many machine learning algorithms use iterative optimization with the loss function as the objective function to search the space F for the function f^* described in eq 2.2. The efficiency of this process is a crucial part of machine learning, and by making appropriate assumptions about our search space, we can potentially improve how effectively we maneuver through it. In this section, I will introduce deep learning and the most commonly used optimization algorithm called gradient descent.

Deep learning (DL) use neural networks to define the space F. The foundational concept of neural networks draws inspiration from neurobiology [20]. They consist of interconnected components referred to as nodes or neurons, mirroring the structure of the brain [20]. This intricate structure allows for the construction of complex functions through the composition of numerous simple signals. In the human brain, a signal is received in a sensory organ and propagates electrical impulses through the brain. This is analogous to the propagation of input signals in a neural network, however, the impulses are replaced by compositions of linear transformations and non-linearities. The most common type of networks are feed-forward, also called fully connected networks, in which the network structure can be represented as a directed, acyclic graph (DAG). Typically the network consists of layers of nodes, where each node in a layer is connected to all the nodes of the next layer, as seen in figure 2.1. The first and the last layer is commonly referred to as the input- and output layers. Every intermediate layer is called a hidden layer.

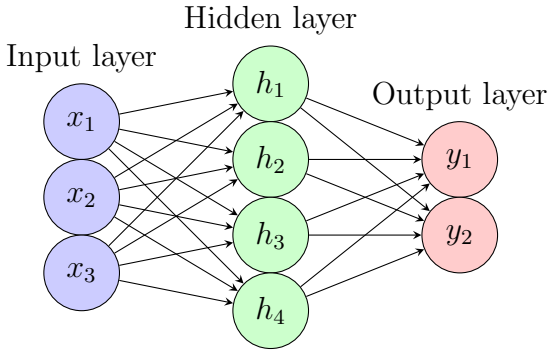


Figure 2.1: Neural Network with one hidden layer.

In order to understand how information flows from the input layer to the outputs/predictions let's look at how a single neuron operates.

Let's consider a neural network with P layers, denoted as the input layer (layer 0), hidden layers (layers 1 to P-1), and the output layer (layer P). The output of a neuron in layer p is computed using a non-linear function, called an activation function, applied to a linear combination of the outputs from the previous layer plus a scalar value called the bias. Mathematically, for a neuron j in layer p, we can express its output as:

$$a_j^{(p)} = \sigma\left(\sum_{k=1}^{n^{(p-1)}} w_{jk}^{(p)} a_k^{(p-1)} + b_j^{(p)}\right)$$

where:

- $a_j^{(p)}$ is the output of neuron j in layer p.
- σ is the activation function applied element-wise.
- $w_{jk}^{(p)}$ is the weight associated with the connection between neuron k in layer $(p-1)$ and neuron j in layer p.
- $a_k^{(p-1)}$ is the output of neuron k in layer $(p-1)$.
- $b_j^{(p)}$ is the bias term associated with neuron j in layer p.
- $n^{(p-1)}$ is the number of neurons in layer $(p-1)$.

The activation function σ introduces non-linearity into the neural network, enabling it to learn complex, non-linear relationships between inputs and outputs. Common choices for activation functions include the sigmoid function, tanh function, and rectified linear unit (ReLU) function. The weights $w_{jk}^{(l)}$ and biases $b_j^{(l)}$ are parameters of the neural network that need to be learned from the training data. The learning process involves adjusting these parameters to minimize a loss function and find the optimal solution f^* in F.

The space F defined by a neural network is the set of all functions that can be expressed by the network given appropriate values for all trainable parameters. By restricting the networks to use only differentiable non-linearities (or almost everywhere differentiable functions like ReLU), we restrict our function space F to differentiable functions, as can be easily seen from the chain rule. This allows the optimization task to be performed using gradient-based optimization algorithms, such as gradient descent. Before we dive into the details of gradient descent, we should note that, when F is given by a neural network, all the functions $f \in F$ can be expressed as the network function N of parameters defined by the vector $\hat{\theta}$, that is $F = \{N(\hat{\theta}) \mid \hat{\theta} \in H \in \mathbb{R}^d\}$ where d is the number of parameters in the network and H is some predefined space. We refer to the function given by $N(\hat{\theta})$ for some $\hat{\theta}$ as $f_{\hat{\theta}}$. Rewriting eq. 2.2, in terms of this new formulation, gives us the equation:

$$\hat{\theta}^* \approx \arg\min_{\mathbb{R}^d} \frac{1}{n} \sum_{i=1}^{n} l(f_{\hat{\theta}}(x_i), y_i) \tag{2.3}$$

Gradient-based optimization means that the gradient $\nabla[\frac{1}{n}\sum_{i=1}^{n} l(f_{\hat{\theta}}(x), y))]$ with respect to the parameters $\hat{\theta}$ of the network provides information of the slope of l along all

dimensions of $\hat{\theta}$. These slopes are used to evaluate the appropriate update for the next step in the search. The rule for updating the parameters of the network at timestep t is:

$$\hat{\theta}_{t+1} = \hat{\theta}_t - \alpha * \nabla[\frac{1}{n}\sum_{i=1}^{n} l(f_{\hat{\theta}}(x_i), y_i))], \tag{2.4}$$

where α is a positive scalar called the **learning rate**. This manually chosen hyperparameter indicates the magnitude of change at each iteration. In a locally convex space, the gradient ∇l will decrease as we approach a local minimum of l. However, even though we can guarantee convergence to a stationary point under certain weak assumptions [28], we can not guarantee that this is the global optimum, nor that it is even a local optimum.

2.1.3 Error Decomposition

Recall from eq. 2.1 that when solving problems, we would like to find a function that minimizes the expected loss over the entire distribution D of possible samples. As described in the previous sections, supervised learning provides a framework for approximating such a solution. In order to make the problem more manageable, we had to make some assumptions to simplify its complexity. Before diving into machine learning, it's crucial to realize that this step is essential. Without knowledge about the problem and its potential solutions, there is no definitive approach that can be considered better than any other in general. This fact is rooted in the **no free lunch theorem**. The no-free-lunch theorem states that if any optimization algorithm is averaged over all possible objective functions, its performance will be the same as a random or blind search. Mathematically this can be expressed in the following way:

$$\sum_{f \in \mathcal{F}} P(f) \cdot L_{alg}(f) = \sum_{f \in \mathcal{F}} P(f) \cdot L_{rand}(f) \tag{2.5}$$

where $\mathcal{F}$ is the set of all possible objective functions, $P(f)$ is the probability distribution over the objective functions, $L_{alg}(f)$ is the expected performance of the optimization algorithm on function f, and $L_{rand}(f)$ is the expected performance of random search on function f. The theorem implies that there is no universally superior optimization algorithm and that the effectiveness of an algorithm is heavily dependent on the specific problem it is applied to. So when handling a problem, any feasible approach will have to

introduce constraints on the scope of the optimization and, by extension, introduce some bias toward the underlying assumption on which the algorithm is based. This bias may increase the error in our final model compared to the optimal one.

Though constraining the problem is a necessary step, balancing between a concentrated and an exploring search is key, and a well-known problem in machine learning often referred to as the bias-variance trade-off [17]. In order to solve this problem, it is important to understand the ways that different constraints introduce errors. One way of summarizing this is by decomposing the error into three main parts based on the introduced bias. This decomposition is based on the work of Bronstein et al. [10].

Firstly, we restrict the candidate functions to be the pre-defined pool of functions F, not guaranteeing that it contains the optimal solution. We call the error caused by this restriction the **approximation error**. Secondly, we use empirical approximation to evaluate our solution's quality which naturally inserts some variance, which we call the **statistical error**. Thirdly, we are generally unable to brute force a complete search through every candidate function in F but instead leverage optimization techniques such as gradient descent to search for a solution. This allows for a discrepancy between the best solution we found and the best solution available, and we call this error the **optimization error**.

The statistical error is closely related to the concept of overfitting since it measures the difference in performance on the samples provided and its generalized performance. The approximation error, on the other hand, measures the difference in generalized performance between the best solution of our candidate space F and the optimal solution. In general, we want to balance the trade-off between the approximation error and the computational complexity of evaluating a large number of candidate functions. The optimization error will be of lesser significance to this thesis. A primary focus, however, will be on the balance between the approximation- and the statistical error, specifically on how to decrease the approximation error without increasing the statistical error through careful construction of the function space F. That is, reducing the chance of overfitting without increasing the degree of underfitting. Figure 2.2 illustrates the intuition behind this decomposition. The level curves indicate the performance of the models with respect to the true data distribution (blue), and the subsampled experience (pink). F_δ is our hypothesis space F. ϵ_{stat} and ϵ_{appr} refers to the statistical and approximation error respectively.

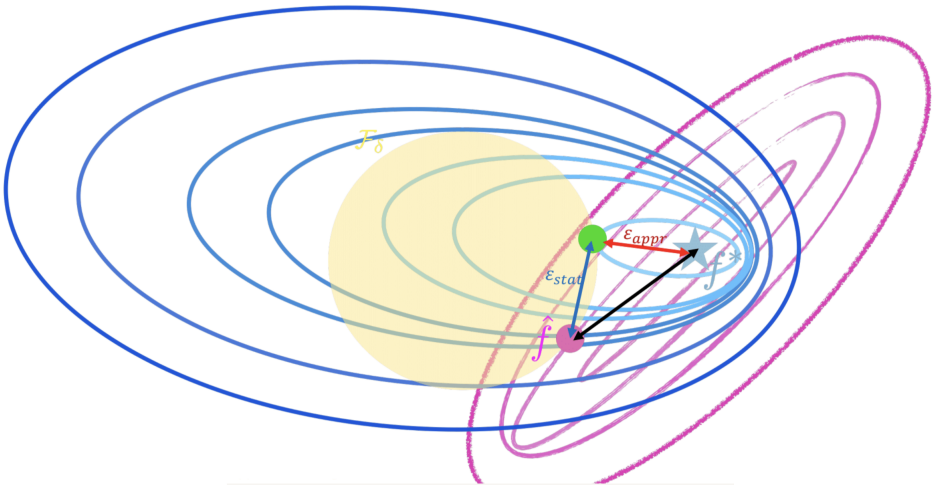


Figure 2.2: Illustration of the error decomposition solution space. The illustration is taken from [10].

2.1.4 Model Complexities and Function Spaces

As mentioned, in supervised learning, we define a pool of functions F, from which we hope to find a suitable function f that approximates the solution to a given problem. It is necessary to ensure that the space F covers a suitable domain of candidate functions for the process of searching through it to be of any value. In this section, we will talk about the construction of F and how this construction balances our approximation and statistical error. Before addressing this issue, remember from section 2.1.2 that we generally only expect to find a locally optimal or close to a locally optimal solution and typically don't expect to find f^*.

One approach to ensure that F contains a good solution is to increase the **complexity** of the neural network. In practice, this means increasing the number of parameters in our model, by increasing the dimensionality d of the vector $\hat{\theta} \in \mathbb{R}^d$, and as such increase the cardinality, or the number of functions in F. One of the greatest strengths of deep learning with neural networks is that we can increase the complexity of our model arbitrarily, thus making the space F include functions that are arbitrarily similar to any function [21]. Although doing this ensures that the space contains a solution as close to optimal as we want, finding it becomes an increasingly complex problem, and will often increase the optimization error. In addition, increasing the complexity allows our model function to adapt to increasingly small data tendencies, which in practice often leads to overfitting, thus increasing our statistical error. Given the assured existence of a good solution in F,

the problem can be tackled from a new viewpoint. That is, finding ways to restrict F without removing potentially great solutions.

In general, there are two ways of introducing such constraints. One is by the removal of functions from F. Another is by introducing a measure of preference to the functions of F. This is often called **regularization**. Regularizing allows us to prioritize searching certain functions or types of functions. In theory, how we choose what functions to prioritize could be done in any way that encodes some preliminary information we might possess about the problem. However, it is common that this regularization bases its preferences on the principle presented in **Occam´s Razor**. Occam's Razor is a guiding principle in machine learning that encourages the preference for simpler models over complex ones [9]. That is, by increasing the preference for less complex functions. In practice, regularization is often introduced as a scalar-valued function $R(\hat{\theta})$, which we add to our loss function, and as such, seek to minimize. Adding this to function to eq. 2.4 we get:

$$\hat{\theta}^* \approx \arg\min_{\hat{\theta} \in \mathbb{R}^d} R(\hat{\theta}) + \frac{1}{n} \sum_{i=1}^{n} l(f_{\hat{\theta}}(x_i), y_i). \tag{2.6}$$

L1-Regularization, often called Lasso regularization, is a commonly used regularization term. It penalizes high absolute values for the parameters in a neural network. For a network $f_{\hat{\theta}}$ with parameters $\hat{\theta}$, the L1 regularization term $R_{L1}(\hat{\theta})$ is introduced as follows:

$$R_{L1}(\hat{\theta}) = \lambda * \sum_{\theta \in \hat{\theta}} |\theta|. \tag{2.7}$$

The λ -parameter controls the strength of the regularization. A higher value of lambda will result in stronger regularization, pushing more weights toward zero and promoting sparsity in the model.

The second option is the removal of functions from F, which is usually a more complex problem than changing their priority. Firstly because it is typically done through the alteration of the neural network, and due to the complexity of this structure, this often requires deep understanding and experience. Secondly, because the removal is a more definitive action than decreasing their preference, it might have a more significant impact on our search. However, some of the greatest successes in machine learning have been

made possible by constructing new network structures that allow us to decrease the size of the function space F without removing relevant functions [23]. An important concept that enables these types of constraints is the concept of symmetries in the data and how to exploit them.

2.1.5 Symmetries

Symmetries is a crucial concept in machine learning, without which supervised learning in high dimensions would be impossible [24]. It can be used as a tool to reduce problems of high complexity, and in fact, knowing all symmetries of the data would make learning the label function trivial. We will revisit this statement, but first we establish a formal definition of symmetry.

Definition 1 *A **symmetry** of an object is a transformation of that object that leaves it unchanged.*

As an example, let's say our object x is an image of a butterfly in the space X of all images of butterflies and define a transformation $g : X \longrightarrow X$ that flips the image along its vertical axis.

Figure 2.3: Image of a butterfly with its symmetric axis. Illustration taken from [1]

We can see that the input is unchanged under the transformation of g. This implies that g is a symmetry of x, and assuming this holds for all the images in X, we can

generalize this by saying that g is a symmetry of X. Several types of symmetries are of interest when doing machine learning. In particular, we are interested in those symmetries that leave the input unchanged with respect to our label function. Formally this is a transformation g of our input space X such that:

$$\hat{f}(g(x)) = \hat{f}(x), \tag{2.8}$$

for all $x \in X$. Revisiting the previous statement that knowing all symmetries would make any classification problem trivial becomes obvious now. Knowing all symmetries would provide us with transformations between all elements within a given class. Thus, we would only need to provide one labeled sample for each class to solve the problem completely. However, this is rarely the case. The more typical concern is to find the optimal use of information obtained from a given set of symmetric transformations.

Provided some data X with a symmetric transformation g, we know that the label function $\hat{f}$ is also invariant to g; thus, it is obvious that our solution f should have the same property. Including this information in our search for f should increase its performance, or at the very least, not decrease it since we can guarantee some similarities between f and $\hat{f}$. Explicitly we can guarantee that if f correctly predicts the label of some $x_k \in X$ then f will correctly label all of $\{x \,|\, x \sim x_k, \, x \text{ and } x_k \in X\}$, the equivalence class of x_k. Here $\sim$ is the equivalence relation given by the transformation g.

In general, there are two ways we can enforce this property on f. The first option is called **data augmentation**. This is a pre-processing step of the data where we use the symmetric transformations of our data to increase the amount of experience available by including all or some of the transformed, equivalent samples provided by these transformations. In this way, the model should be able to learn these invariances itself. This is, however, not necessarily the most efficient approach. Firstly, this does not in general guarantee complete invariance, and secondly, the computational expense is increased proportionally to the number of symmetric transformations. We can think of this as a way of decreasing the statistical error by increasing the amount of experience available.

The second approach relates to section 2.1.4 and is about restricting F to contain only functions that share the invariances of L. This approach can potentially decrease both the optimization- and the statistical error without increasing the approximation error.

For deep learning with neural networks, F is defined through the architecture of the network, so to make this restriction, we need to design networks that enforce the

desired properties. Maybe the most established and successful of such approaches are the **convolutional neural networks** (CNN) introduced in LeCun et al. [23] for image processing and classification that exploits the invariance to translation in images [24].

The primary architectural component in a CNN that harnesses translation symmetry is the convolutional layer. In this layer, a set of learnable filters (also known as convolutional kernels) slide or convolve across the input image. Each filter captures local patterns or features in the image by performing a dot product between its weights and a corresponding spatially local region of the input. By exploiting the symmetry of translation through shared filter weights, the CNN can effectively learn local features that are agnostic to their specific spatial positions in the image. This allows the network to generalize well across different locations and achieve translation invariance. In 2.4

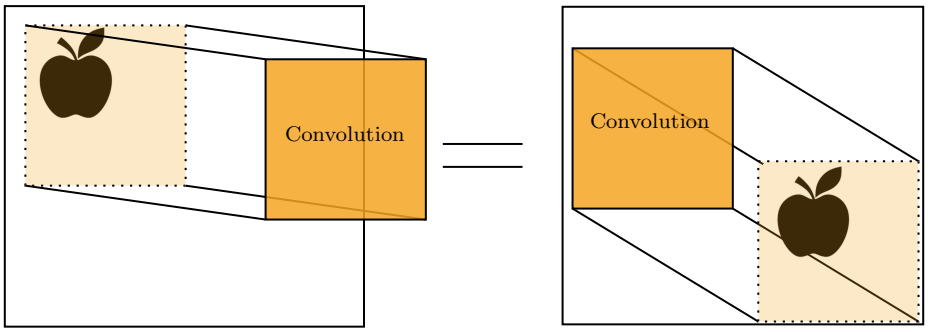


Figure 2.4: Illustration of translation invariance in the convolution operation of CNN.

2.2 Topology and Shapes

Topology is a field in mathematics that studies the properties of geometric objects that remain unchanged under continuous deformations, such as stretching and twisting. These are properties that pertain to the structure and connectivity of space on a global scale. The significance of studying topology lies in the realization that many geometric problems are primarily concerned with the fundamental structural aspects of a space, rather than its exact shape. A good example of such problems appear in fields where problems typically can be represented as graphs such as in network analysis and neuroscience, since graphs are inherently topological objects [19].

In recent years the popularity of topologically focused algorithms such as t-SNE [12], UMAP [25] and Mapper [29] has grown [31], implying the relevancy of evaluating the

structural features in real-world data. A significant concern for the field of TDA has been finding appropriate topological invariants to summarize the relevant topological information of a space. Until quite recently, most approaches were based on deterministic algorithms. These kinds of algorithms do not typically consider the respective probabilities associated with different regions in the feature space of the data, which is often relevant in realistic, noisy data. Hence they may not always be the most reasonable approach, as they can generally struggle with sensitivity to outliers. Stability with respect to different kinds of noise is by and large a desired property in machine learning, and recently new approaches in TDA that try to provide this have been proposed. This will be considered in section 2.3.4. However, in order to analyze and evaluate different methods throughout this thesis, I will first introduce some necessary background in topology and topological data analysis.

Definition 2 *For an arbitrary set X and a set of subsets τ of X, τ is a* ***topology*** *on X if the following holds.*

1. *$\emptyset$ and X is in τ.*
2. *The union of the elements of any subcollection of τ is in τ.*
3. *The intersection of the elements of any finite subcollection of τ is in τ.*

The topology of a space is a definition of its open sets and establishes the fundamental structure that underlies the properties of the space. Typically these open sets, as defined by the topology, are thought of as neighborhoods of elements in a set.

Definition 3 *If X is a set, a* ***basis*** *for a topology on X is a collection B of subsets of X such that:*

1. *For each $x \in X$, there is at least one element of B that contains x.*
2. *If x belongs to the intersection of two basis elements B_1 and B_2, then there is a basis element B_3 containing x such that $B_3 \subset B_1 \cap B_2$.*

By utilizing the basis to define the open sets, we gain the advantage of establishing the topology of a space without the need to explicitly enumerate all open sets. This approach grants us flexibility in representing the space and enables efficient analysis and characterization of its topological properties.

The construction of a topology from a basis is done by taking the unions of sets within the basis and considering them as open sets. These unions, along with the empty set, serve as the foundation for the topology. To ensure that the resulting structure meets the requirements of a topological space, specifically requirement number three in the definition, we also include all possible intersections of these unions.

For machine learning purposes, it is reasonable to limit the scope of relevant spaces to ones where we have a way of assessing the similarity of elements. We will focus on metric spaces, which are spaces that have a metric defined on them.

Definition 4 *A **metric** on a set X is a function d: $X \times X \to R$ having the following properties:*

1. ***Identity**. d(x,x) = 0, for all $x \in X$*
2. ***Non-Negativity**. d(x,y) > 0, whenever $x \neq y$ for all $x, y \in X$.*
3. ***Symmetry**. d(x,y) = d(y,x), for all $x, y \in X$*
4. ***The triangle inequality**. d(x, z) + d(z, y) $\geq$ d(x,y), for all $x, y, z \in X$*

Provided a metric, we can construct the metric topology on a space. The metric topology asserts the neighborhood analogy as it uses the proximity of elements to define the open sets of X. This will be the only type of topology considered in this thesis.

Definition 5 *If d is a metric on a space X. Let $B_d(x, \epsilon) = \{y \mid d(x, y) < \epsilon\}$, called the **$\epsilon$-ball** centered at x. Then the collection of all ϵ-balls for $x \in X$ and $\epsilon > 0$ is a basis for the **metric topology** induced by d.*

A topological invariant of a space is a property that can be expressed in terms of its topology. This implies that as long as the topology is preserved, any deformation and transformation on the space is ignored. It follows that for any space X, other distinct spaces exist that share all topological properties of X(all continuous deformations of X, for example). We say that these spaces are **homeomorphic** to X, which means that from a topological perspective, they are equivalent, implying a one-to-one mapping exists between their open sets.

2.2.1 Homology

A common goal of analyzing the structure or shape of a space is often to be able to distinguish between similar spaces and those that are not. When making such a distinction based on topology, direct comparison of their open sets is often complicated and not done in practice. Instead, we make comparisons based on properties derived from the topology, the most common of which being the **homology**. In this section, I will briefly introduce this concept and give a general overview of its purpose. In section 2.2.2, we will look at this concept in more detail in an applied setting.

Homology is a fundamental tool in algebraic topology that allows us to assign algebraic objects, called homology groups, to topological spaces. These groups quantitatively capture some of the topological structure of a space. Specifically, by associating homology groups to a space, we can detect and quantify the presence of holes, tunnels, voids, or other topological features. These groups provide information about the number and dimensions of these features in a given space. Spaces with different homology groups are topologically distinct and cannot be transformed into one another through continuous deformations. We say that these spaces are not homeomorphic.

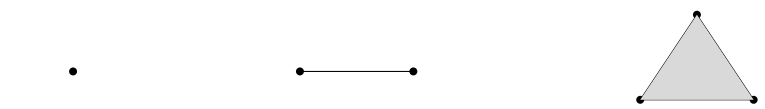
We will now look more formally at the concept of homology, particularly for the cases commonly encountered in machine learning.

2.2.2 Persistent Homology

Many computer science tasks handle large, high dimensional data sets which often contain relevant topological properties [31], and the field of research called **computational topology** overlaps the field of topology and computer science. A particular property that is important to note when transitioning from the theoretical field of topology to machine learning is that these data sets are commonly assumed to be sets sampled from an underlying distribution or a geometric object like a manifold, and it usually is the properties of this underlying structure that are of interest. Figure 2.5 shows a set of point clouds sampled from a geometric object with various kinds of noise.

Figure 2.5: Point cloud sampled for the underlying shape of a chair, with various kinds of noise. Illustration taken from [2]

As mentioned in section 2.2.1, we seek to compute the homology groups of a space in order to compare and distinguish between different spaces. We assume, as mentioned above, that capturing properties of the underlying distribution is the main objective. Computing the homology of this space might not be possible if the full information about the topology of the space is not available, and as such, we resort to computing the **persistent homology** instead. The concept behind persistent homology is to track how the homology groups of a space change over a range of different parameter values. This allows one to identify a space's most persistent or stable topological features across a range of scales or parameters and to distinguish them from transient or spurious features. These parametric values define how we assume the underlying distribution around our samples to be shaped. For this thesis, we will only consider radius-based approaches that use an increasing radius around points to define the expansion of the space. We will now look more formally into the computation and theory of persistent homology and start with its most fundamental building block. A **simplex** is the generalization of triangles and tetrahedrons to arbitrary dimensions. For instance, a 0-simplex is a point; a 1-simplex is a line; a 2-simplex is a triangle, and so on. A k-simplex is defined by a set of k+1 endpoints. For a k-simplex σ we use the notation $\sigma = (v_0, ..., v_k)$, where $(v_0, ..., v_k)$ is the set of endpoints.



Simplices are fundamental building blocks in algebraic topology and are used to construct more complex topological spaces when combined in larger families. When these families are closed under taking subsets, we call them complexes or **simplicial complexes**.

Definition 6 *A **simplicial complex** K is a family of simplexes closed under taking subsets. That is if $\sigma \in K$ and $\tau \subseteq \sigma$, then also $\tau \in K$.*

Simplicial complexes allow us to represent complex topological structures using simple building blocks and create a rigorous mathematical foundation for topological analysis. From these complexes, we can extract chains of elements from different dimensions called k-chains.

Definition 7 *Let K be a simplicial complex. A **k-chain** is the finite formal sum*

$$\sum_{i=1}^{N} c_i \sigma_i$$

where each $c_i \in \{0, 1\}$ and each σ_i is a k-simplex in K.

Intuitively, k-chains represent a collection of k-simplices and are used to evaluate larger k-dimensional regions of a simplicial complex.

Definition 8 *Let K be a simplicial complex and σ a k-simplex $\sigma = (v_0, ..., v_k)$. Then the **boundary** of σ is defined as*

$$\delta\sigma = \sum_{i=0}^{k} (v_0, v_1, ..., v_{i-1}, v_{i+1}, ..., v_k) \tag{2.9}$$

*Similarly, the **boundary** of a k-chain C is*

$$\delta C = \sum_{j=0}^{N} c_i \delta\sigma_i \tag{2.10}$$

Definition 9 *A **cycle** is a k-chain with an empty boundary, that is, $\delta\sigma = 0$ for some k-chain σ.*

Details on the construction of simplicial complexes will be discussed later in this section. We assume that a pre-constructed simplicial complex has been provided to us for now. As mentioned, we wish to analyse this structure using its homology groups, which we now have defined as a sufficient background for expressing more formally.

Definition 10 *The kth* ***homology group*** *of a simplicial complex is the quotient group*

$$H_k = Z_k/B_k,$$

where Z_k are all the k-cycles and B_k are all the k-boundaries.

Intuitively, any k-cycle that is not a boundary of some $(k+1)$-chain in the complex is considered a non-zero element of the homology group. Given some space X, the dimension of $H_k(X)$ is called the kth Betti number, which is a topological invariant of X. For example, the 0-th homology group $H_0(X)$ is isomorphic to the group generated by the connected components of X, because a 0-cycle is a formal linear combination of points. The basic idea of homology is to study the properties of continuous maps between topological spaces in terms of how they transform cycles and boundaries. By looking at how cycles and boundaries are mapped under a continuous function, we can define algebraic maps between homology groups that measure how the topology of the space changes. Roughly speaking, we can say that the homology of a space counts the number of holes a space has in each dimension. A cycle can be thought of as a chain of some dimension that forms a closed loop, and whenever this cycle encapsulates something that is not present in the space, or more precisely, is not a boundary, there is a hole.

Definition 11 *A* ***filtered simplicial complex*** *is a sequence of simplicial complexes* $K_0 \subseteq K_1 \subseteq K_2 ... \subseteq K_N$

Filtered simplicial complexes are a collection of simplicial complexes created by sequentially adding new simplices and all their sub-simplices to an existing complex. The **persistent homology groups** of the space are the homology groups for each of the simplicial complexes. This sequence captures the structural change in the space as we gradually change the value of different parameters.

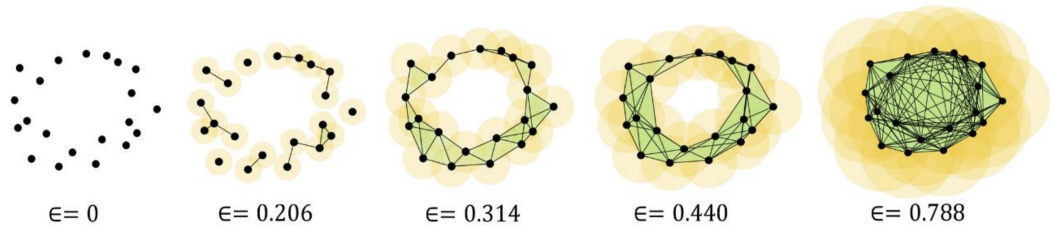


Figure 2.6: A filtered simplicial complex constructed from a radius-based approach. Illustration taken from [22].

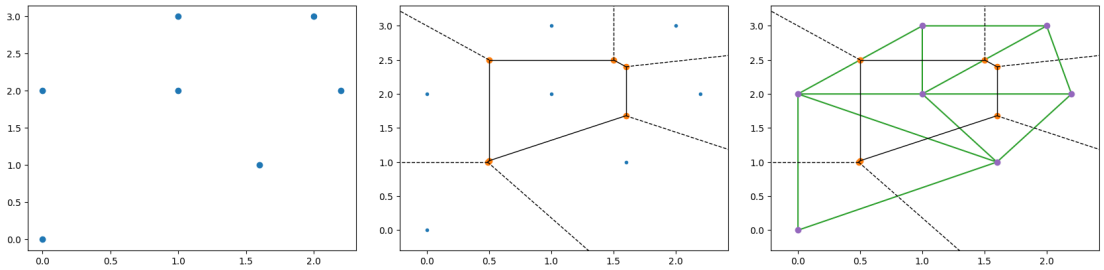
2.2.3 Alpha Complex

For this thesis, we will focus on radius-based approaches, which use a growing radius around each point as the parameter, which we vary in order to construct several simplicial complexes. Specifically, we will consider the **alpha complex** [27].

The alpha complex is constructed from the Delaunay triangulation, and is a subcomplex of the Delaunay-complex, meaning that for any radius δ the alpha-complex $\alpha(S)$ for some point set S is contained in the Delaunay complex. The Delaunay complex is in turn strongly related to the Voronoi cells. For a finite set of points, $S \subset X$, where X is the set of points in some a metric space (X, d), the Voronoi-cells is a segmentation of X into a finite set of regions, such that each region contains exactly one element s from S and all points in X where the distance to s is smaller than to any other point of S. Mathematically the Voronoi region R_k associated to a point s_k in (X, d) can be expressed as:

$$R_k = \{x \in X \mid d(x, s_k) \leq d(x, s_j) \text{ for all } j \neq k\} \tag{2.11}$$

The Delaunay-complex consists of the points S and a k-simplex is included whenever a set of k+1 points have adjacent Voronoi cells. See figure:



(a) Pointset S (b) Voronoi Cells (c) Delaunay Graph over Voronoi Cells

Figure 2.7: Pointset with Voronoi Cells and the corresponding Delaunay Graph.

The alpha-complex uses, as mentioned, an increasing radius δ to create balls around each point in S. The parameter δ can grow arbitrarily, however, the balls around a point are constrained by the Voronoi cells. Whenever the intersection of a set of k+1 balls is non-empty a k-simplex is added. This is typically done for all dimensions up to some pre-determined integer value but could be computed for infinite dimensions. Some natural limitations, however, exist so that this never has to be done. Firstly given some dataset D of size m where the elements are from $\mathbb{R}^d$, we know that there are no subsets larger than m. This implies that the homology groups H_n for $n > m$ are trivial. Secondly, when working with data of dimension d, we know there are no holes of dimensions $n > d$. So we would need to compute the homology groups at max up to a dimension of $min\{m, d\}$.

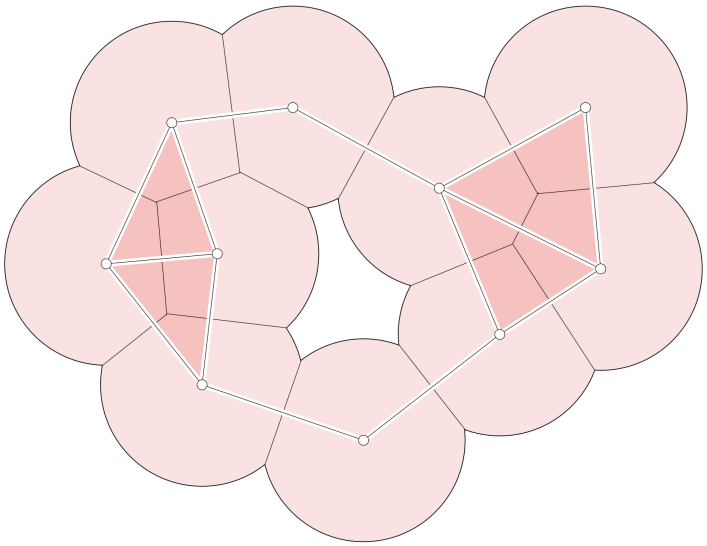


Figure 2.8: Alpha-complex at a given radius. The balls around each point are constrained by its Voronoi cell. Illustration retrieved from Atienza et al. [8]

2.2.4 Persistence Diagrams

As the filtration parameter is changed in our filtered simplicial complex, topological features appear and disappear. These events are encoded in the changing Betti numbers we extract by computing the homology. By recording the appearance and disappearance of these features by their filter values, we can represent the lifetime of different topological features by these two numbers, often called the birth and death values.

The complete information about the change in topological features of the space can be displayed in what is known as a **persistence diagram**. This diagram consists of a set of points in the plane, each representing the lifetime of a topological feature. This forms a set-like structure where each point corresponds to a particular topological feature, and the entire set collectively encodes the topology of the data.

Each point's x-coordinate represents the birth scale, and the y-coordinate corresponds with the death scale. The diagonal line y = x represents features that are born and die at the same scale, while points above this line correspond to longer-lived features.

Importantly, we keep track of the dimension of each topological feature as well as its lifetime. This is done by combining multiple diagrams, one for each dimension of homology, effectively forming a set of persistence diagrams.

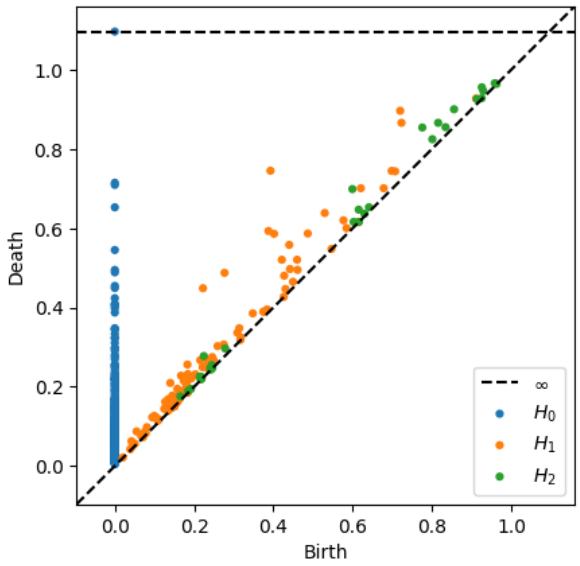


Figure 2.9: Peristence diagram for some point cloud.

Persistence diagrams serve both as a visual tool and a mathematical representation for extracting information about the topological structure of a data set. Importantly, they

are stable with respect to small perturbations in the input data. This stability property ensures that minor changes in the data set do not drastically alter the persistence diagram, making them robust to this kind of noise. However, this is not the only type of noise likely to occur in real data sets. Furthermore, persistence diagrams enable the comparison of different spaces, thereby facilitating the analysis and understanding of their topological characteristics.

However, it's worth noting that the multidimensional set structure of persistence diagrams poses challenges when incorporating them into traditional machine learning models. In the upcoming sections, we'll explore strategies to tackle these challenges, such as vectorization methods and specific deep learning architectures designed to handle such inputs.

2.3 Sets and Vectorizations

For the entirety of this chapter, we have discussed machine learning approaches that focus on mappings for an input space X to an output space Y, where X particularly is a vector space. In practice, most machine learning approaches require an input of fixed dimensions [32]. When working with inputs that form a set, we need to find a suitable, vectorized representation. Furthermore, this **vectorization** should preserve the properties of the set, the most important of which is its **permutation invariance**. Since a set of elements has no natural ordering, any ordered representation of its elements is equivalent. As an example, we can express a permutation invariant function f on a set $\{x_1, x_2, x_3\}$ as any function satisfying:

$$f(x_1, x_2, x_3) = f(x_2, x_3, x_1) = ... = f(x_3, x_2, x_1), \tag{2.12}$$

for every permutation of $\{x_1, x_2, x_3\}$. When working with general data structures, preserving the inherent symmetries of the data is crucial. In the case of sets, the number of symmetric elements has a factorial scaling with respect to the number of elements, making it even more important here.

An additional challenge is the sensitivity to the number of elements in the set. We do not want to place restrictions on the size of the input sets. Thus it is important that even sets of different sizes would have vectorized representations of the same dimensions, since two sets that are embedded in distinct vector spaces are harder to compare.

We call vectorizations that preserve both of these properties for valid **set functions**. To preserve the permutation invariance under a vectorization f, we have the following result from Zaheer et al. [32]

Theorem 2.3.1 *A function $f(X)$ operating on a set X having elements x_i from a countable universe is a valid set function, i.e., invariant to the permutation of instances in X, iff it can be decomposed in the form $\rho(\sum_{i=0}^{n} \phi(x_i))$, for suitable transformations ϕ and ρ.*

which states that any permutation invariant function on a set can be described as a transformation ρ of the sum over each set element after individually applying the transformation ϕ. We call the transformation ϕ the **feature extractor**, and ρ the **downstream transformation**. In the theorem, the sum operator is used for the set aggregation, but we can expand on this restriction with the results in Soelch et al. [30] to show that the summation operator can be exchanged with any sum-isomorphic operator, such as the mean or the max. In the following sections, we will examine some standard manual methods and how to construct trainable vectorizations using neural networks.

2.3.1 Vectorizations

For general sets the most common vectorizations are simple scalar-valued aggregation functions such as the sum-, mean- or max-functions. We will call these functions **summary functions**. These functions summarize a particular statistic of the set but do typically not capture all relevant properties. However, in combination, the summary functions can be expressive and versatile

For vectorizations of persistence diagrams, Ali et al. [7] proposes an approach that uses a collection of these summary functions called **persistence statistics**. Defined as:

Definition 12 *The **persistence statistics** of dimension k persistence diagram μ consists of:*

- *the mean, the standard deviation, the median, the interquartile range, the full range, the 10th, 25th, 75th and 90th percentiles of the births p, the deaths q, the midpoints $\frac{p+q}{2}$ and the lifespans q - p for all intervals $[p, q]$ in μ_k counted with multiplicity.*
- *the total number of points in μ_k.*

- *the entropy of μ_k as defined in [7].*

However, there are some additional properties that we want to preserve in persistence diagrams specifically, which the persistence statistics do not. Particularly we want invariance with respect to points along the diagonal. That is points with persistence 0.

One of the most common vectorizations used, that preserves this property is **persistence images**. They were introduced by Adams et al. [6]. Their construction starts with choosing a grid of pixels that covers the range of birth and death times of the points in the persistence diagram. Each pixel represents a small interval of time, and its value is determined by a weight function that measures the number or intensity of points in the persistence diagram that fall within that time interval. There are several weight functions that can be used to define a persistence image, such as the Gaussian kernel. Typically the weight function aims to capture information about the density of points in the region as well as their persistence.

Once the grid and weight function is chosen, the persistence image is obtained by convolving the weight function with the persistence diagram, essentially projecting the points onto the pixel grid and summing their weights. The resulting image represents a smoothed and continuous version of the persistence diagram.

2.3.2 DeepSet

As seen in theorem 2.2.1, any set function can be described by two transformations, one operating on each element of the set and one operating on the sum of the transformed elements. This fact inspires a general framework for learning set functions called DeepSet, presented by Zaheer et al. [32]. The feature extractor and the downstream network, ρ and ϕ, are replaced with universal approximators which transfer the universality properties to the domain of set functions [32]. Specifically, DeepSets replaces the feature extractor and the downstream function with two independent neural networks of any kind. Although the polling function is a fixed hyper-parameter of the model, the two transformations are trainable and allow for learning set vectorization that is specialized for the given task [32].

$$\rho(\sum_{i=0}^{n} \phi(x_i)) \tag{2.13}$$

This architecture provides a general baseline for designing models for handling set structures. Because of its general structure, many applications use a design based on DeepSet, but with specialized modifications. DeepSets offers several advantages as a vectorization method for sets. Primarily, the replacement of the transformations ϕ and ρ with trainable neural networks allows the model to learn and adapt the set vectorization specifically to the given task at hand. These have been shown to be effective in a variety of applications, including point cloud classification, graph classification, and text classification [32].

2.3.3 PersLay

PersLay is a supervised approach to vectorizing persistence diagrams introduced in Carrière et al. [14]. It is a deep learning method that maps a persistence diagram to a trainable feature vector using Deep Sets in a way that aims to learn a representation of the topological information in the diagrams.

PersLay is for the most part an application of DeepSets to a concrete domain than a new approach. Still, in the paper, some specific architectural design choices are made to display the ability of the DeepSet approach to recreating other vectorizations, such as persistence landscapes and persistence images. For general purposes, however, no significant constraints on the topology of the neural network are added in the PersLay architecture, except for a weighting mechanism. The weighting function is implemented by including a second parallel network in the feature extractor, assigning each point in the diagram a scalar-valued weight. This weight is multiplied by the feature vector.

$$\rho(\sum_{i=0}^{n} \phi(x_i) * w(x_i)) \tag{2.14}$$

The advantage of PersLay over other vectorization methods is that it can capture more complex topological features of the data, as well as noise and perturbations. It is also a fast and scalable method, making it suitable for large datasets. But most importantly, since this method is trainable, it can adapt its representation to the task at hand and is able to learn what features to encode and which to ignore.

2.3.4 Distributed Persistence

Although persistence diagrams are great tools for summarizing and comparing topological structures, they fail to provide a few desired properties. Firstly, as mentioned, persistence diagrams are stable with respect to small perturbations in the data. This is a great property to have when working with such noise. However, this is not the only type of noise likely to appear in typical real-world data. Including additional incorrect or bad samples is arguably a more common problem, and full persistence diagrams do not provide any stability guarantees with respect to this. We can formalize this property slightly by conjecturing that we would like stability with respect to small changes in the underlying probability distribution from which our samples are drawn. This includes a notion of probability to the analysis by assigning probabilities to points or regions in the data and would allow us to overcome the challenges related to sensitivity to outliers. This consideration is something that the computation of full persistence diagrams is missing when using a radius-based complex such as the alpha complex.

Distributed persistence introduced by Solomon et al. [31] is a method for studying the topology of a space which, instead of computing the homology groups of the entire dataset at once uses subsampling and computes the persistent homology or other topological invariants of many smaller subsets. It has been proposed to sample m subsets from the pool of subsets of size k and compute the persistent homology of each of these, for some m and k. Previous work has also been done on other subsampling schemes, such as using all subsets of size at most k. However, this thesis's primary focus will be sampling subsets of a given size. The paper [31] provides a theoretical foundation for this distributed persistence approach's stability, inverse stability and its interpolation between topological and geometric information. However, this foundation is based on the criteria that both the topological invariant and complete information about the samples of each subset are provided. The structure that consists of a labeled set of subsets and their topological invariants is a way more complex structure to handle in a machine learning setting, so the approach we will consider is the one proposed by [15] without labels.

Chapter 3

Related Work

Several articles have explored the topic of distributed persistence and multiple variations have been proposed. One of the earliest works in this area was the distributed algorithm for persistent homology proposed by Boissonnat et al. in 2011. The algorithm is based on the idea of computing the persistent homology of local neighborhoods of the data and then merging them together to obtain the global persistent homology. The authors showed that their algorithm can scale to very large datasets and can be implemented in a distributed and parallelized manner. However, their approach differs from the one considered in this thesis as it uses local subsamples while we consider globally uniformly sampled subsets.

In the paper, Chazal et al. a new distributed algorithm for computing persistent homology of large datasets is proposed. The algorithm is based on the idea of randomly sampling the subsets. It was shown that their algorithm can achieve significant speedup and scalability compared to a serial implementation, while still producing accurate results. In their approach, they consider an aggregate of persistence landscapes computed from a collection of n random subsamples of size m and their Vietoris-Rips-filtrations. They also discuss the theoretical properties of their algorithm, including the stability of the computed persistent homology with respect to perturbations of the data and the trade-offs between communication costs and discriminating power. In Solomon et al. [31] a similar subsampling scheme is applied, but they consider a set of labeled topological invariants instead of aggregating the set of invariants. However, under the condition of including labels for the subsets, they prove the stability of the approach with respect to perturbations and show that it has a globally stable inverse. One of the main contributions made in [31] is also proof that their method interpolates between topological and geometric information.

In Solomon et al. [31] a case study on a simple point cloud dataset was conducted. Here it is highlighted how full persistence computation with radius-based filtrations lacks robustness to noise and is unable to capture information about density in space. They also show how a distributed approach, the one proposed by Chazal et al. [15], overcomes this problem by smoothly interpolating its computed invariants when noise is added. This suggests that this topological invariant also as an unlabeled set contains relevant information for distinguishing and comparing spaces even with noise. Finally, the paper lists a number of open problems for future work in distributed persistence, two of which are closely related to the objectives of this thesis. They mention the open question of finding good vectorized representations of the persistence diagram constructed from the collection of subsamples, and also what information is contained in the unlabeled subsamples.

Vectorization of persistence diagrams has been an important topic of research for many years in topological data analysis. In Ali et al. [7] a survey of multiple such vectorizations is conducted and their performance for various supervised machine learning tasks is compared. In this comprehensive study, it was shown that the persistence statistics proposed by the authors performed very well and better than other commonly used approaches for several of the well-known datasets used, including FMNIST and SHREC14. The paper considered approaches in the context of supervised machine learning and also considers trainable methods, but separates the vectorizations from the supervised learning process which differs from our approach.

In 2017 Zaheer et al. [32] introduced DeepSets, a neural network architecture for learning vectorized representations of sets. Its design ensures the criteria for a valid set function. Later, in 2019, PersLay was introduced by Carrière et al. [14]. This neural network architecture extends the design requirements from the DeepSet to specialize in the transformation of persistence diagrams. The PersLay architecture enforced a method for weighting the elements of the diagrams and continued this functionality which has been commonly used in other vectorizations such as persistence images and landscapes. In their paper Carrière et al. [14] demonstrated its effectiveness in shape recognition and object classification tasks using persistence diagrams obtained from 3D shape data.

In a more recent study, Bubenik et al. [11] (2021) compared PersLay to other persistence diagram vectorization methods in various applications, including image analysis and protein classification, and showed that it outperformed some of the other methods in certain contexts.

Chapter 4

Method

As a topological invariant, distributed homology has shown promise in providing insights into the topological characteristics of complex data [31]. Being a theoretical framework that considers both the density and connectivity of topological spaces, it offers the potential for a more nuanced understanding of the underlying structure of datasets. By taking a probabilistic approach to homology computation, distributed homology has the possibility to reveal intricate patterns and relationships that may not be readily apparent through traditional methods, especially for realistic noisy data.

Despite its proven richness in encoding important information about data with shape, the unstructured and complex nature of distributed homology poses challenges for its effective utilization in practical applications.

In response to this challenge, this thesis proposes a novel pipeline that combines the power of distributed homology with the expressiveness and flexibility of supervised learning techniques. Specifically, we adopt the DeepSets framework, a deep learning approach designed to handle set inputs, to capture the intricate structure of distributed homology. By incorporating DeepSets into our pipeline, we aim to unlock the latent information contained within distributed homology, thereby enabling seamless integration into the domain of supervised learning tasks.

4.1 DSSN

In this section, I introduce DSSN (Deep Set of Set Network), a general framework for inserting inputs of set, where each element of the set is itself a set (a set of sets), into a supervised machine learning pipeline.

In section 2.3.2, we saw that any valid set function could be decomposed into a feature extractor, an aggregate operator, and a downstream function. By simple insertion, we can infer the general structure of any function operating on a set of sets while still satisfying the conditions of a valid set function. Assume you have a family of m sets X, where X is a family of subsets of some point cloud S. Then any valid function for sets of sets can be decomposed as:

$$\rho_1(\sum_{i=0}^{m} \rho_2(\sum_{j=0}^{n_i} \phi(x_{ij}))), \tag{4.1}$$

where ρ_1, ρ_2, ϕ are suitable transformations. The inner summation $\sum_{j=0}^{n} \phi(x_{ij})$ captures the processing of each individual element of the inner sets, while the outer summation $\sum_{i=0}^{m} \rho_2(\cdot)$ combines the processed results for each set in the family. By inserting neural networks for various transformations in this equation we get models capable of learning appropriate functions, and we call this general architecture DSSN. Equation 4.1 gives the universal structure of any valid set of set functions.

The DSSN framework provides a flexible structure that allows for the inclusion of various vectorization methods of persistence diagrams by insertion in place of the inner transformation $\sum_{j=0}^{n} \phi(x_{ij})$. For instance, techniques such as persistence images and persistence statistics can be easily integrated into the DSSN architecture.

4.2 DDPN

We also propose a more specialized architecture, the DDPN (Deep Distributed Persistence Network) for encoding distributed homology specifically. This is a DSSN that is specifically designed for processing a family of persistence diagrams that includes a neural network architecture inspired by PersLay [14]. It copies the idea of a parallel weighting function in the feature extractor for the vectorization of the diagrams, however, some modifications were found that improve performance. Firstly, we assign a scalar valued weight to each dimension of the set embedding individually. Secondly, this weighting is based only on the dimension of the topological feature, not on its birth, death or persistence.

$$\rho_1(\sum_{i=0}^{m} \rho_2(\sum_{j=0}^{n_i} \phi(x_{ij}) \odot w(\dot{x_{ij}}))) \tag{4.2}$$

where ρ_1, ρ_2, ϕ and w are replaced by 4 separate fully connected neural networks. For clarity:

- $w : \{0,1\}^3 \to \mathbb{R}^{q1}$, takes only the one-hot-encoding of the dimension of each topological feature a gives a trainable weighting of each of the output dimensions for a point.
- $\phi : \mathbb{R}^3 \to \mathbb{R}^{q1}$, is the feature extractor that maps the points of the persistence diagrams to a q_1 dimensional embedding.
- $\rho_2 : \mathbb{R}^{q_1} \to \mathbb{R}^{q_2}$, receives the vectorized persistence diagrams embedded in q_1 and acts as the feature extractor on the set of these. The diagrams are mapped to q_2 dimensions.
- $\rho_1 : \mathbb{R}^{q_2} \to \mathbb{R}^c$, takes the aggregated set representation as input. It further processes the aggregated representation and produces the final model output.

The elements of the inner set in, x_{ij}, no longer refer to subsets of S but rather the set of points in the persistence diagram of such a subset. Here $\dot{x_{ij}}$ is the one-hot-encoding of x_{ij} and $\odot$ is the element wise multiplication.

The motivation for this approach is simple. By not putting additional constraints on any of the building blocks, the network can learn any valid set function, including the labeling function. Including the dimension weighting w in the inner set function rather than a more general DeepSet provides a less general network, but more specialized to the specific tasks of processing persistence diagrams. We will discuss the functionality of this w in chapter 6. Doing feature engineering, such as pre-computing the persistence diagram representations using persistence images or statistics as mentioned in section 2.3.1, can greatly simplify the optimization problem and allows us to reduce the size of our hypothesis space of potential solutions. Still, with sufficient domain coverage in the provided experience, learnable approaches should be able to find a solution that is at least as suitable.

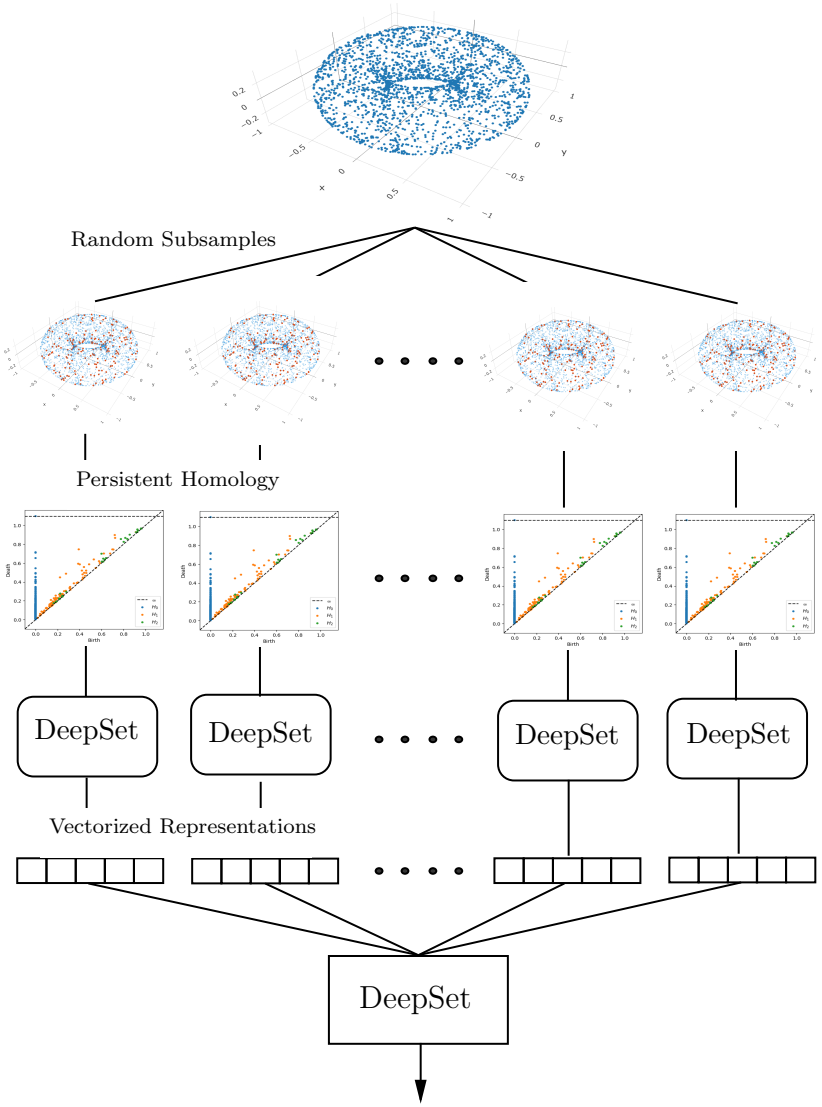


Figure 4.1: Overview of the full DDPN pipeline.

4.2.1 Implementation Details

A significant reason behind the growth of machine learning in recent years is the remarkable increase in computation power and efficiency. This progress has been facilitated by the emergence of specialized computing units such as GPUs, which have accelerated the training process of machine learning systems and enabled the handling of large-scale datasets that were previously impractical or even infeasible to process. Parallelization, in particular, has played a pivotal role in this advancement. Unlike traditional sequential computing, where a single processor performs tasks sequentially, parallelization allows

multiple processors to work simultaneously, leading to significantly faster computation. Neural networks, among other machine learning algorithms, are inherently parallelizable, as their computations can be decomposed into smaller, independent tasks that can be executed concurrently. Popular frameworks like PyTorch support parallelization and GPU computing, with critical components often implemented in high-performance programming languages like C++. Leveraging these implementations is highly advantageous for practical purposes. However, although PyTorch supports set-like inputs, it currently lacks implementations that support parallelizing sets of varying sizes or sets of sets.

One critical aspect of PyTorch implementations is the assumption that inputs are tensors, and computations rely on the computational graph and back-propagation. Tensors require all their corresponding elements to have the same dimension. However, when dealing with sets, each set may have a different size. This introduces a challenge with two possible solutions: either utilizing a dynamic data structure that allows for varying-sized inputs or ensuring that all sets have the same size by inserting or removing elements, a process known as padding.

DDPN V.1

In the initial solution, the model was implemented without padding and utilized Python lists as a data structure allowing variable-size elements. Each batch where represented as a list of samples, where each sample was in itself a list of nested tensors.

This approach, however, sacrificed the benefits of PyTorchs efficient implementation and suffered greatly in terms of both speed and efficiency. Although efforts where made to increase performance using high performance languages like Cython and C, none yielded satisfactory results.

DDPN V.2

A key challenge when using padding to match tensor sizes is that adding new elements to a set through padding can potentially impact the model's output. Ensuring that the model remains invariant to these padded elements, while maintaining its handling of the original elements, is generally a non-trivial problem.

However, as mentioned in 2.2.4, a desired property of any persistence diagram vectorization is invariance to elements along the diagonal. By incorporating this invariance

into the architecture, we can guarantee the preservation of a symmetric property and simultaneously resolve the padding problem.

In the second iteration of DDPN (DDPN V.2), the dynamic data structure was discarded, and the architecture was modified to address the invariance property described above. Padding was also introduced by appending all-zero vectors to the inner sets, ensuring that the size of each inner set matches the size of the largest set in the mini-batch.

In the case of PersLay, which was specifically designed to handle persistence diagrams, a weighting function was introduced. While the underlying reasoning for this design choice is not explicitly mentioned in the paper, including such a weighting function and ensuring that it assigns zero weights to zero elements solves the padding problem. In practice, the weighting function can be implemented as a regular fully connected network with no bias in any layers, ensuring that zero-vectors are not transformed. It is unclear whether the inclusion of the weighting function in the PersLay paper was motivated by the same line of reasoning.

By adopting this new approach and incorporating padding into the input tensors, both computational speed and performance were drastically improved.

Note that this approach can be transfered to the general DSSN structure as well without loss of its generality, trough a binary weighting function.

All experiments in this thesis where conducted using the DDPN V.2.

Chapter 5

Experiment

The experiments in this thesis have two main goals. Firstly, the aim is to investigate the impact of different parameter choices on the distributed persistence computation. To understand the unstructured information within distributed persistence across various parameter values, a series of experiments will be conducted in a supervised setting. The performance of the models will be used as a basis for comparing the input's expressive power and gaining insights into the encoded information.

Secondly, the thesis will evaluate the performance of the DDPN (Distributed Deep Persistence Network) and other DSSN (Deep Set-based Neural Network) architectures for supervised machine learning tasks, specifically point cloud classification. To establish a baseline, the proposed methods will be compared to traditional full persistent homology approaches across all architectures. Furthermore, the trainable approaches utilizing DeepSets, introduced by us, will be compared to non-parametric methods, such as the one tested in Solomon et al. [31].

5.1 Pre-processing

In this section, I will cover the parts of the pipeline that precede the training of the models in the machine learning pipeline. This includes the normalization of the point clouds as well as how persistence diagrams, both for full persistence and distributed persistence, are computed. Additionally, we will cover how the input is represented.

Sampling Scheme

The construction of the distributed persistence used in the experiments is based on the approach first suggested by Chazal et al. [15], which is described in detail in section 2.3.4. The procedure begins with sampling a fixed number m of subsets, each containing a fixed size of k points, uniformly from the given point cloud. For each sampled subset, the filtered sequence of alpha complexes is computed. Subsequently, persistent homology is calculated, and the corresponding persistence diagrams are extracted. All topological computations are handled using the GUDHI [3] library.

Normalization

Normalization refers to the process of scaling the input data so that it falls within a specific range. This can potentially be a very important step in a machine-learning pipeline since many machine-learning algorithms are optimized for data that conform to a normal distribution. For neural networks, as utilized in this thesis, most schemes that are used for the initialization of the network weights assume normalized inputs [18]. Even though they are typically able to learn how to handle inputs that are not normalized, they tend to train and converge faster when this step is included. By normalizing the inputs, we can also remove any bias towards certain input features that might be present in the raw data due to their scale [18].

We will use Min-Max scaling on the point clouds for normalization. Remember that rescaling does not change the topological properties of the clouds, see section 2.2. MinMax-scaling fixes the range of the data to be exactly [0,1]. We are not scaling the inputs directly, but rather the distribution of distances between points in the point cloud. However, since we sample points from the cloud and use distances as filtration values in the alpha complex, by fixing the radius of the clouds to be exactly 1, we can ensure that any distance is in the range [0,2]. Consequently, the birth, death, and persistence values lie within this range as well.

Given a point cloud P of elements from $\mathbb{R}^3$ we do a normalisation of the radius of P with MinMax using the following formula:

$$P_{norm} = \{x/r_{max} \mid x \in P\}, \tag{5.1}$$

where r_{max} is given by $r_{max} = \max_{x \in P} |x|$, when all point clouds are centered in origo. This transformation scales each dimension by the same scalar and as such preserves all angles, making it preferable when evaluating geometrically distinct spaces.

A second approach was considered where each dimension of the point cloud is scaled independently. This is preferable from a topological viewpoint since a greater range of parameter values is utilized, however since we are evaluating the geometric information contents as well, it was discarded.

Input Representation DDPN

The DDPN model learns a representation of the input persistence diagrams during training. However, it is important to clarify how these diagrams are represented as they are fed into the model. Each sample is represented as a family of padded persistence diagrams. These diagrams are represented as vectors, where each vector element consists of the birth, death, and persistence (death - birth) values of a topological feature. Additionally, a one-hot encoding is included to indicate the dimension of the topological feature. We will be considering point clouds in 3D, and remembering from section 2.2.3, there is no need to consider homology groups of higher dimensions, so the one-hot-encoding is three-dimensional as well.

To illustrate, consider an example where a 1-dimensional hole is born at $\delta = 0.2$ and destroyed at $\delta = 0.9$. The representation of this persistence diagram would be as follows:

0.2	0.9	0.7	0	1	0

In this representation, the first two elements (0.2 and 0.9) correspond to the birth and death values of the 1-dimensional hole, respectively. The third element (0.7) represents the persistence value, calculated as the difference between the death and birth values. The remaining elements (0, 1, 0) form the one-hot encoding, indicating that the topological feature is 1-dimensional. This one hot encoding corresponds to the vector $\dot{x_{ij}}$ from eq. 4.2.

Each persistence diagram comprises a set of such vectors. For efficiency reasons, each diagram is padded with zero vectors, ensuring that all diagrams within a mini-batch contain the same number of points. This padding mechanism helps maintain consistency in the input dimensions and allows for streamlined batch processing during training. In section 4.2.1 the background for this is discussed in further detail.

5.2 Pilot

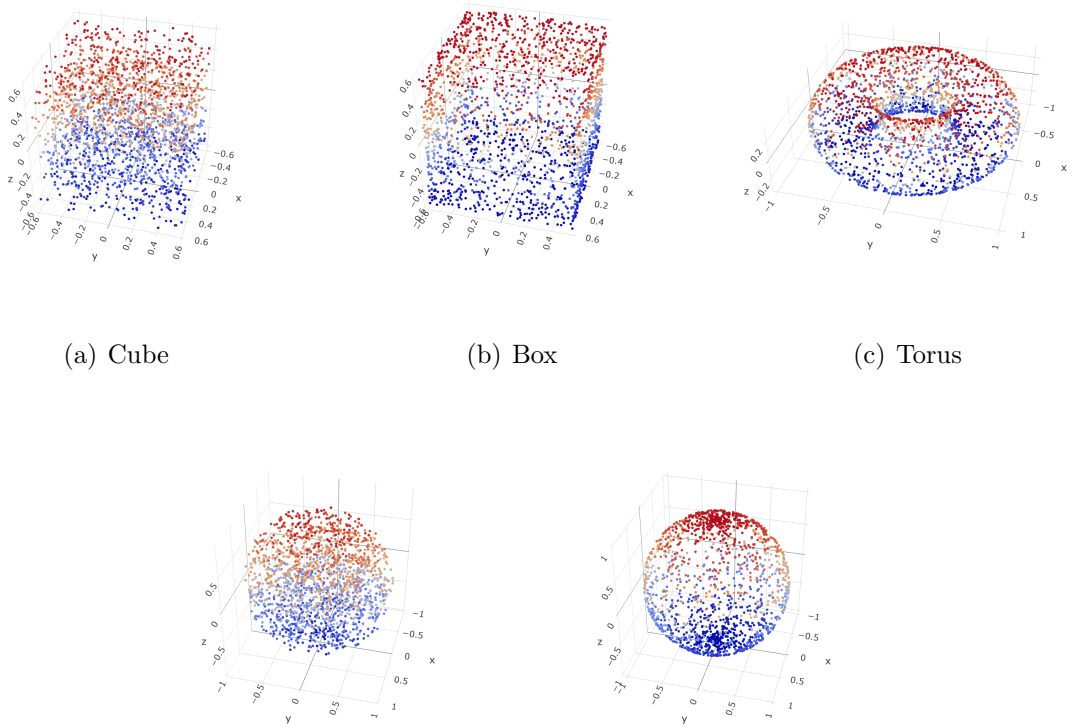
For initial studies, of both the properties and information contents of the distributed persistence approach, as well as the performance of various models, a pilot study was conducted using a synthetic data set. Specifically, this pilot aims to gain insight into how various parameters of the distributed persistence approach affect the overall performance of different models and what kind of information is preserved. The two crucial parameters of the distributed persistence approach are the number of subsets m and the size of each of these subsets k. One initial assumption we want to evaluate is whether or not it is strictly better to increase the parameter m. We compare the resulting performances against a baseline using full persistent homology of the clouds and various models.

Importantly, the goal was also to gain insight into the various hyper-parameters of the DDPN and their effect on the model's final performance. Ultimately we aim to find a well-performing model in order to make a reasonable estimate of the performance of our approach on real data.

5.2.1 Data

For this pilot, we constructed a synthetic data set of 3D point clouds for classification tasks which we will refer to as PointClouds. The data set consists of 750 clouds evenly distributed over 5 classes: filled cube, box (cube surface), torus, ball, and sphere. Each cloud in the data set is created using the tadasets-library [5] by sampling (250-2500) points uniformly from one of these manifolds. Additionally, varying levels of Gaussian noise are added to the samples, but there are corresponding samples in all classes with matching noise levels. This noise is added by setting the noise parameter [5] to $i/150$ for the i-th cloud of each class. Finally, for all sets of corresponding samples, a random re-scaling of the dimensions is applied. The rescaling factor of each dimension is sampled uniformly and independently from [0.5, 2]. The clouds are centered at origo.

The classes were selected to cover a suitable range of topologically and geometrically distinct spaces. A torus contains both a 1-hole and a 2-hole. Both a sphere and a box contain only a 2-hole, but they are clearly distinguishable from a geometric standpoint. The same is true for cubes and spheres which are both simply connected spaces, that is without any holes, but still are geometrically different.



(a) Cube (b) Box (c) Torus

(d) Ball (e) Sphere

Figure 5.1: Sample from every class of the PointClouds dataset without noise. The points are colored by their Z-value for clarity.

5.2.2 Models

Network Architechtures

Equation 4.1 provides a general description of a deep-learning approach using distributed persistence, and the DSSN network embodies this structure without additional constraints. Different models and specialized architectures, can be constructed by inserting well-known vectorizations in replacement for the trainable segments of this architecture, will be tested. Specifically, the component we will modify and experiment with the most in the models is the function for persistence diagram vectorization, $\rho_2(\sum_{j=0}^{n} \phi(x_{ij}))$.

We will experiment with three vectorization methods. Two common deterministic approaches to vectorizing persistence diagrams: persistence images and statistical vectorizations as described in section 2.3.1. Both vectorizations have previously been successfully applied to similar classification problems. Additionally, persistence images were

used to argue for the value of distributed persistence in [31] and are thus included in our study for comparison.

Let v be either of these three vectorizations, and x_i be the set of points of the persistence diagram of the i-th subset, then the structure of the two models used will look like this:

$$\rho_1(\sum_{i=0}^{m} v(x_i)), \tag{5.2}$$

$$\rho_1(\sum_{i=0}^{m} \rho_2(v(x_i))), \tag{5.3}$$

In equation 5.2 we take a simple aggregate of a set of vectorized persistence diagrams, while in equation 5.3 we handle the diagrams as elements in a set for a standard DeepSet model, by applying a feature extractor to the vectorized representations. In addition to these two, we will test the DDPN network using a PersLay [14] inspired feature extractor for the inner vectorization.

Across all experiments and models, the dimension-wise mean operator will be used as the set function for the aggregate of the set of persistence diagrams.

A summary of all model architectures can be seen in table 5.1. Here Ω is the persistence statistics and β is the persistence image vectorization.

	Inner Vectorization	Outer Vectorization	Downstream Network
IMG-M	$\beta(x_i)$	$mean(\cdot)$	$\rho_1^1(\cdot)$
STAT-M	$\Omega(x_i)$	$mean(\cdot)$	$\rho_1^2(\cdot)$
STAT-DS	$\Omega(x_i)$	$mean\left(\rho_2^1(\cdot)\right)$	$\rho_1^2(\cdot)$
DDPN	$sum(\phi(x_{ij}) * w(x_{ij}^{\cdot}))$	$mean\left(\rho_2^2(\cdot)\right)$	$\rho_1^3(\cdot)$

Table 5.1: Model architectures for pilot experiments.

$\phi, w, \rho_1^2, \rho_1^3, \rho_2^1, \rho_2^2$ are fully connected neural networks, and ρ_1^1 is a CNN. All of these neural network architectures are described in detail in Appendix A.

The decision to apply only one model using persistent images was driven by the observation that it exhibited poorer performance compared to the other approaches. This aligned with our initial assumptions and the observations made by [7].

Distributed Persistence Parameters

There are two main parameters of the distributed persistence calculation, and we seek to explore their effects with these experiments. The first of these is the number of random subsamples drawn from a point cloud, that is the size m of the family X of subsets of S for some point set S. Secondly, we use a fixed size for all of these subsets, and this constant is presumably an important parameter that we call k. For each network architecture covered in section 5.2.2 and summarized in table 5.1, we will train a model for each parameter combination in a pre-defined set for distributed persistence.

A complete list of all parameter values used in the experiment is displayed in table 5.2.

Data-parameters	Values
k :	2, 5, 50, 100, 500, FULL
m :	1, 3, 10, 31, 100, 316, 1000

Table 5.2: Range of parameter values for distributed presistence.

We introduce a naming convention for use throughout this chapter for convenience when referring to a specific model and the data on which it was trained. We use subscript as an indication of the parameter m and superscript as a reference to the subset size k. As an example, a DDPN model trained on data with parameters $m = 100$ and $k = 50$ is referred to as DDPN_{100}^{50}

5.2.3 Training, Model Selection, and Hyper-parameter Space

In this section, we will cover the details of the training pipeline as well as the model selection and tuning of certain hyper-parameters.

Fixed Parameters

There are a large set of optimizable hyper-parameters in the training pipeline and the optimal combination may vary significantly for the different models. By fixing some of these across all models it might be easier to isolate the effect of the parameters we aim to evaluate. On the other side, ensuring that these fixed parameters are not better suited for one model than another is hard. Either way, the performance of our models might

be affected in unwanted ways. We decided to hold several parameters constant across as many experiments in consideration of the computational expense.

A simple hold-out validation scheme is used across all experiments. Additionally, all experiments and models use the same train-, validation-, and test split of 250/250/250 samples respectively. Traditionally a larger portion of the data is used for training, however, we found this training size to be sufficient and we saw no reason to compromise the estimates of the generalized performance by reducing their size. As the loss function, we use CrossEntryopyLoss [26] with the Adam optimizer.

Model Selection

For each (architecture, data parameter)-combination we train a set of models covering a small grid search of hyper-parameters, the model with the best validation performance was chosen.

The learning rate starts at $1e^{-2}$ and using a scheduler it is reduced by 5% every fifth epoch. All models are trained for 1000 epochs and validated every 25. The state of the model at the checkpoint with the highest generalized performance is saved.

Hyper-parameter Search

The architectures described in section 5.2.2 are the only ones used. Since the same architecture is used across multiple inputs with varying sizes and variance the general idea was to create an adequately large and complex network to handle all data sets and we experiment with different degrees of regularization, specifically L1 regularization and the λ-parameter described in 2.1.4.

The specific range of parameter values can be seen in the table 5.3.

Hyper-parameters	Values
Batch Size :	16, 64
λ :	0, 0.001, 0.01

Table 5.3: Hyperparamters.

5.2.4 Results

Table 5.4 shows a table of the generalized test performance of all architectures. The optimal hyper-parameters combination found during model selection for all models is displayed in figure A.3. We display the full range of values for the parameter m, but not all parameters in the range. The complete list can be found in Appendix A.

	IMG-M			STAT-M			STAT-DS			DDPN		
k \ m	1	100	1000	1	100	1000	1	100	1000	1	100	1000
2	24.8	36.0	55.6	20.0	37.2	46.4	23.1	48.8	65.2	24.4	54.4	**74.0**
5	23.2	46.4	68.8	35.6	70.0	82.0	24.0	60.8	84.4	22.4	67.2	**87.2**
50	51.2	83.2	90.0	47.2	95.2	94.4	58.8	92.0	94.8	58.0	**96.0**	95.2
100	59.2	91.2	90.4	69.6	93.6	96.4	68.4	93.2	**97.2**	69.2	93.6	92.4
500	74.8	82.0	-	82.4	89.6	-	77.6	86.4	-	80.4	**92.4**	-
FULL	74.0	-	-	72.0	-	-	72.4	-	-	**78.8**	-	-

Table 5.4: Accuracy in percentages of all model architectures for each combination of the parameter values $m = 1$, 100, 1000 and $k = 2$, 5, 50, 100, 500, FULL.

Across all models and subset sizes, there is a clear indication that increasing the parameter m for training and validation data simultaneously is strictly advantageous. This effect is particularly apparent for small subsets combined with the DeepSet-based approaches. It can also be observed that for these instances the DeepSets outperforms the STAT-M models, however, for the opposite cases, where m is small and k is larger, STAT-M tends to perform better.

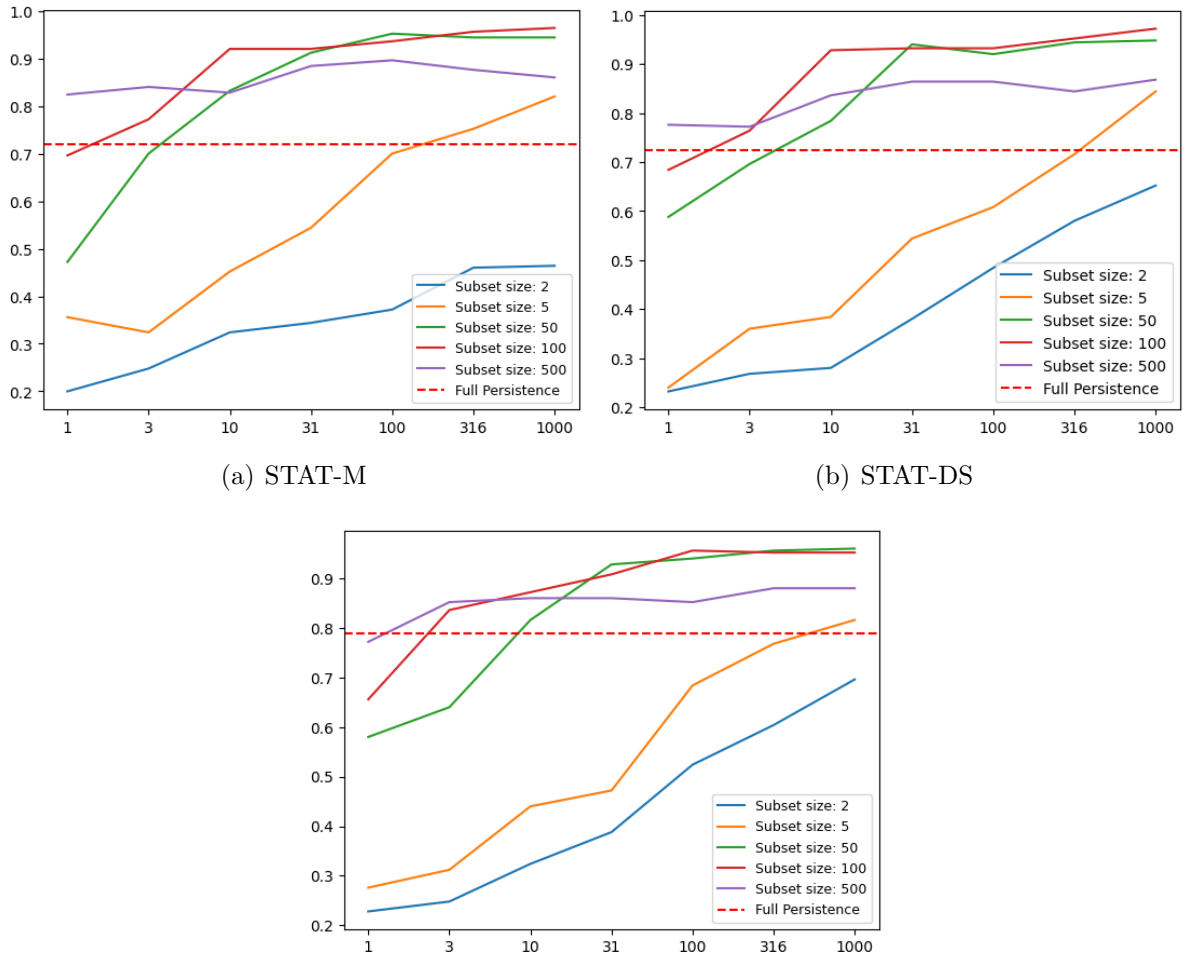
Importantly, for sufficiently large values of both m and k, a distributed approach greatly outperforms full persistence-based models.

It is clear that persistence images do not give similar performances to the other approaches, and for that reason, we will not include them in further analysis.

Table A.3 in appendix A shows that there are no clear patterns in terms of the relationship between the data parameters and the batch size. A tendency towards a lower L1 term for the larger inputs can be observed, implying a smaller need for regularization for these parameters.

Figure 5.2 shows the performance of the STAT-M and the DDPN architecture for various subset sizes k as a function of the number of subsets in the distributed persistence, m. We see as a general trend across all models, that increasing values for m has a positive

impact on model performance. This effect seemingly decreases with the performance of the model.



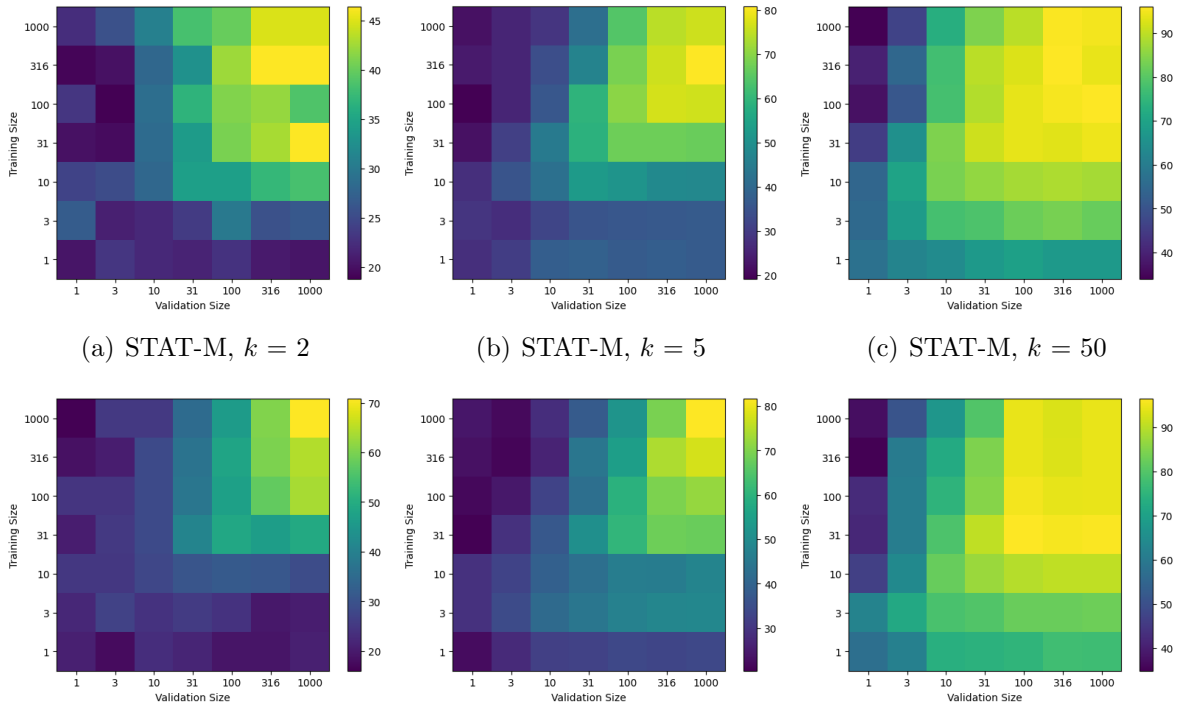
(a) STAT-M

(b) STAT-DS

(c) DDPN

Figure 5.2: Each subfigure displays the accuracy of all models MODEL^k of given architecture as a function of m.

To further analyze the effect of the number of subsets we investigate the resulting performance of various values for m during training and inference independently. We refer to these as m_{train} and m_{val}. In figure 5.3 a series of heatmaps of the generalized performance of each of the models STAT-M^k, STAT-DS^k and DDPN^k when trained on a certain m_{train} and validated against all the other values for m. The heatmaps show similar effects to the results in figure 5.2 giving a clear indication that increased m is advantageous for performance. However, there is a slight shift in terms of optimal performance towards higher validation data sizes. In fact, increasing m_{train} significantly above m_{val} has a negative impact on performance. This is especially true for small values for m_{val} during inference. This effect is present in the results from all architectures.

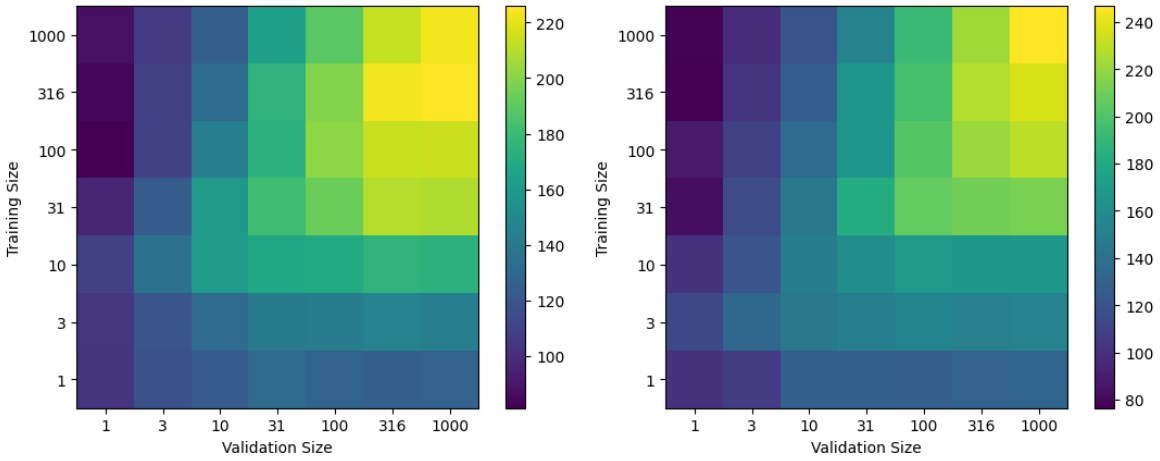


(a) STAT-M, $k = 2$ (b) STAT-M, $k = 5$ (c) STAT-M, $k = 50$

(d) STAT-DS, $k = 2$ (e) STAT-DS, $k = 5$ (f) STAT-DS, $k = 50$

Figure 5.3: Heatmaps showing the relative accuracy on test data for all combinations of the parameters m_{train} and m_{val}.

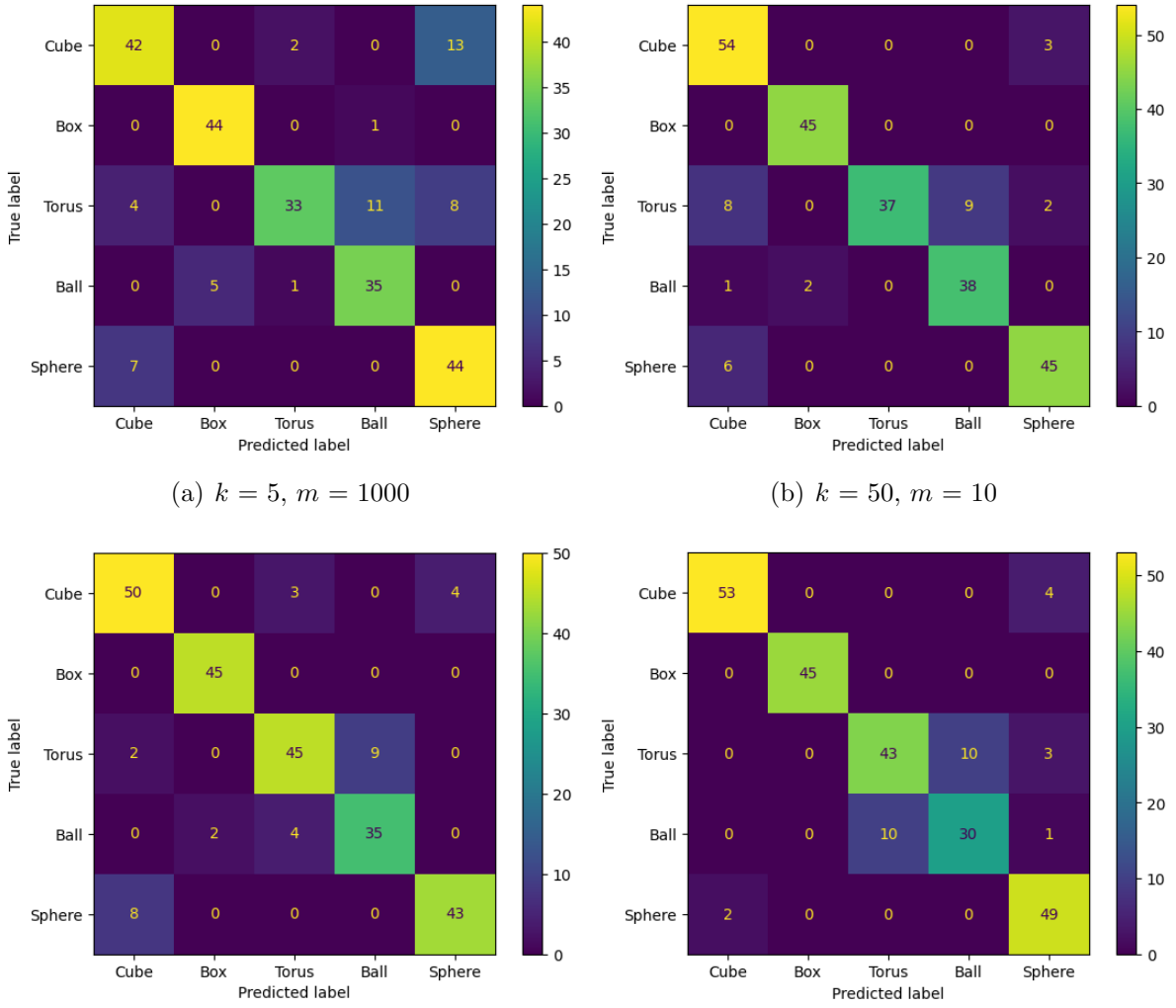
The heatmap shown in figure 5.4 emphasizes the consistency of the observation that increasing m_{val} has a stronger positive effect on performance than during training, across all subset sizes. It displays the average performances of each column in figure 5.3.



(a) STAT-M (b) STAT-DS

Figure 5.4: Shows the normalized sum of the heatmaps from figure 5.3 across all the k-values 2, 5, 50.

In figure 5.5 we display the confusion matrix of a selection of similarly performing models to evaluate the balance of geometric and topological information contents of the data sets they were trained on. All the models are all based on the same architecture in order to reduce variable parameters in the comparisons. There do not seem to be clear patterns supporting our prior assumptions, however, a weak tendency towards better performance on the torus class can be observed. This is also the most topologically complex shape, which corresponds to some degree with the theory from [31].



(a) $k = 5$, $m = 1000$

(b) $k = 50$, $m = 10$

(c) $k = 100$, $m = 10$

(d) $k = 500$, $m = 100$

Figure 5.5: Confusion matrices of various STAT-M models.

5.3 ModelNet10

In the second experiment, we apply the distributed persistence method to a well-known dataset, ModelNet10 [4] and make a comparative study based on the best-performing models developed in the pilot study. ModelNet10 contains a range of different geometric objects for point cloud classification and is commonly used as a baseline problem for shape recognition models. We will compare the performance of our approaches against similar full persistence models.

5.3.1 Data

ModelNet10 is a subset of the ModelNet40 data set and consists of 4,899 CAD (Computer Aided Design) models distributed over 10 classes. The object classes are compiled by a list of the ten most common objects in the world, based on statistics collected from the SUN-database [4]. From these CAD objects a point cloud is sampled uniformly.

It is important to note that many of the underlying geometric objects are actually simply connected spaces from a topological point of view. Meaning that we also in this experiment will be able to evaluate the type of information preserved under various parameters of the distributed homology pipeline.

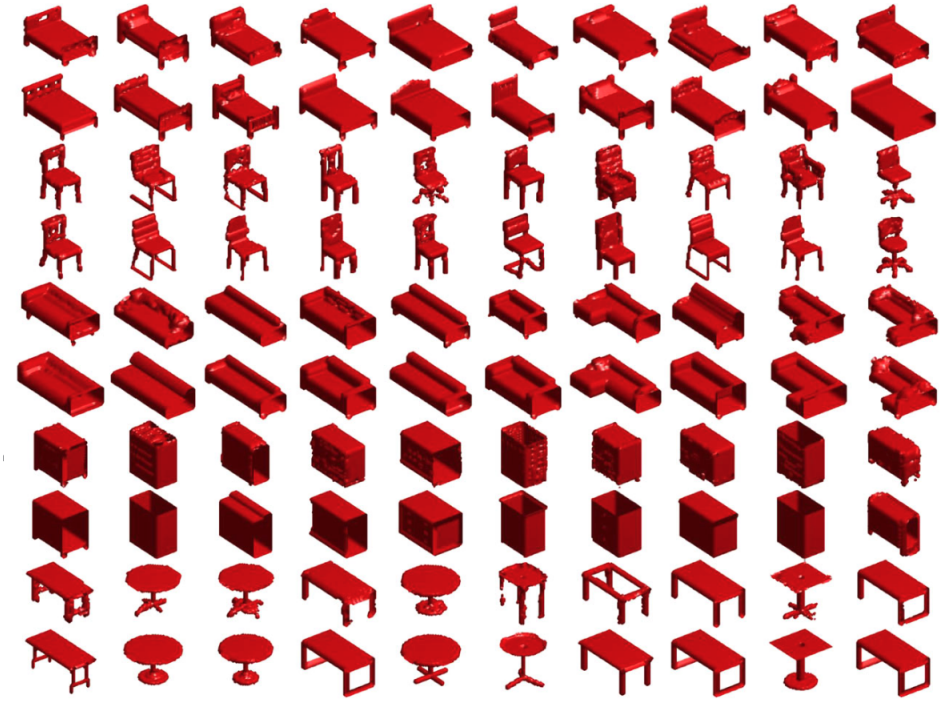


Figure 5.6: Samples of CAD shapes from all classes of the ModelNet10 data set. Illustration taken from [33].

The data is divided into a train/test split of 3991/908. We do not further split the training data into training and validation since no model-selection process was performed on this data. The classes and the object distribution across these classes can be seen in table 5.5.

Class Name	Train	Test	Total
bathtub	106	50	156
bed	515	100	615
chair	889	100	989
desk	200	86	286
dresser	200	86	286
monitor	465	100	565
nightstand	200	86	286
sofa	680	100	780
table	392	100	492
toilet	344	100	444
total	3991	908	4899

Table 5.5: Class distribution of ModelNet10.

For each forward pass, we sample 6000 points from each given in the batch, from which we then compute the distributed persistences.

5.3.2 Models

Because of time restrictions, we will use some of the best-performing models from the pilot study and apply them directly to the ModelNet10 data set. We will in additionally compare each of the networks to the baseline using full persistent homology.

The models used for this experiment are DDPN^{50}_{100}, DDPN^{5}_{1000} and $\text{STAT-DS}^{100}_{100}$. Preferably we would like to evaluate the $\text{STAT-DS}^{100}_{1000}$ but because of computational limitations, we were not able. For comparison we train two baseline models, $\text{STAT-DS}^{\text{FULL}}$ and $\text{DDPN}^{\text{FULL}}$, both selected based on performance in the pilot.

5.3.3 Hyper-parameters

We use the same parameters as used during the pilot, except we run a fixed number of 100 epochs, batch size of 64, and no regularization.

5.3.4 Results

Table 5.6 shows the results of the experiments by the accuracy on the test set. The results align with those from the pilot study, showing clearly that distributed persistence outperforms traditional full persistence.

Model	Accuracy
DDPN^{5}_{1000}	**70.1**
DDPN^{50}_{100}	68.8
$\text{DDPN}^{\text{FULL}}$	55.8
$\text{STAT-DS}^{100}_{100}$	65.2
$\text{STAT-DS}^{\text{FULL}}$	51.9

Table 5.6: Generalized performance, as accuracy in percent, for various models on the ModelNet10 data set.

Although the scope of this experiment is limited and not comprehensive enough to draw definitive conclusions, the results further reinforce the observations made in the pilot study. They highlight a clear discrepancy between full persistence-based approaches and distributed ones. The DDPN networks demonstrate the best overall performance, however, it is worth noting that the DDPN5^{1000} model outperforms the DDPN50^{100} model

on this particular data set, which adds an intriguing aspect to the findings. Knowing the geometric properties of this data set, as discussed in section 5.3.1, this can potentially be considered supportive observations for the theoretical background which shows that smaller k focuses the information content toward geometric features.

It is important to note that there is a noticeable discrepancy between the state-of-the-art performance on this data set and the performance of all our models [4].

Chapter 6

Discussion

Here we analyze our results and present the main findings of our experiments. Additionally, we will evaluate the strengths and weaknesses of both the method in itself, but also our experimental setup and the potential shortcomings of our experiments.

6.1 Main Findings

6.1.1 Analysis

Parameters of distributed persistent homology

The results of our experiments show that our pipeline outperforms the baseline across a wide range of parameters. This holds both the parameters related to distributed persistence as well as the architectures and models employed. Notably, we observe that variations in both m and k have a smooth impact on the model's performance, indicating their stability in a supervised setting.

While increasing m, particularly during inference, appears to consistently improve performance, we achieve optimal results not at the highest values of k, but rather at intermediate values. This observation aligns with the continuous interpolation of information captured by our topological invariant, as shown in [31]. It suggests that in data sets where both geometric and topological features are relevant, the subset sizes yielding

the best performance lie within the intermediate range rather than at the extreme ends of the parameter spectrum.

Furthermore, it is worth noting that even for large subsets where the subset topology closely resembles that of the original point cloud, our approach outperforms the baseline when a sufficient number of subsets is considered. This underscores the effectiveness of a distributed approach in effectively filtering out noise and outliers.

Model Comparison

The results in table 5.4 show that the DDPN model slightly outperforms the other models for almost all subset sizes, making it the highest-performing model for all values of k except $k = 100$. This highlights the flexibility and effectiveness of a trainable approach in capturing the relevant information from the persistence diagrams.

It is worth noting that the performance discrepancy between the DDPN model and the other models primarily exists for the largest values of m. This can be attributed to the requirement for a sufficient amount of training data to effectively train deep learning models compared to pre-constructed vectorizations. Deep learning models typically have a higher capacity to learn complex patterns and representations when provided with a larger amount of training data.

We consider the effects of the various models particularily interesting for the smallest value of k, specifically $k = 2$. In this case, both the DeepSet vectorization of the set of persistence images and each individual diagram vectorization exhibit a significant increase in performance compared to the STAT-M^2 model. This suggests that for the smallest of subsets the persistence statistics are not optimal, and that DeepSets increases the effectiveness of our model in these cases.

For small values of m, we observe that the overall performance of all models becomes quite similar. However, the non-parametric vectorizations of the STAT-M approach demonstrates a slight advantage in this scenario. This indicates that when the number of subsets is small, the simplicity and flexibility of feature engineering methods are effective and comparable to the more complex deep learning models.

Hyper-parameters of DeepSets

We have iterative and manually tested several key parameters of the DDPN architecture as a part of the model evaluation process in the pilot studies, and made several observations to improve performance. We will cover some of the most interesting findings here.

We found that a learning rate of $1e^{-2}$ yielded significantly better results than other values. In particular, we observed that lower learning rates converged quickly to solutions with worse performance on both training and validation data indicating that a sub-optimal local minimum was entered. Typically decreasing the batch size can reduce this problem by increasing the variance of each update, however, this was not true for the DeepSet approaches.

Searching for appropriate sizes for the various networks ϕ, ρ, and w did not yield such clear results, and this is still a poorly understood challenge from our side. We found that the embedding sizes $q1$ and $q2$ were significant parameters, but there was a wide range of well-performing values and a combination of values for all data inputs, which we found surprising. In general, the networks appeared very robust to these mentioned parameters.

We observed that including batch normalization for the downstream networks, across all DSSN models has a significant, positive impact for all data sets.

6.2 Strengths and Weaknesses

In this section, we will give an post-experiment summary of our experimental work and analysis. We aim to discuss the strengths of our analysis and methodology as well as acknowledging the shortcomings and weaknesses of our study.

6.2.1 Weaknesses

DDPN Studies

Firstly, during our experimentation with the DDPN architecture and DeepSets in general, we encountered numerous challenges related to hyper-parameter tuning. These types of

networks have unique requirements that do not necessarily align with our intuition from standard neural networks. Due to the limited research and usage of DeepSets in practical applications there is limited knowledge about the effect of various parameters. As a result, we had to spend considerable time searching for functional parameter combinations specific to these models. This non-standard aspect of hyper-parameter tuning could potentially introduce a weakness in our comparisons, as it may limit the generalizability of our findings. In particular, there is a bias in our methodology with regards to the time spent optimizing the architechtures.

Some of the results of this process are described in section 6.1.1. This is potentially a weakness of our comparisons.

Limited Hyper-parameter Search

An important consideration that extends the discussion on comprehensiveness in the previous section is the limited exploration of hyper-parameters in our model evaluation and selection process. This limitation primarily stems from the substantial time investment required for developing and implementing the DDPN network, as described in the subsequent section (4.2.1).

In our experiments, we trained a total of six models for each architecture on each dataset in the initial experiment. This reduced variance in our performance estimates for each architecture. However, conducting an even more extensive hyper-parameter search would have been desirable. It is worth noting that we tested only one architectural design for each type of the three model types across all data parameters. This is not ideal since this is in general an important parameter, which can greatly increase performance when tuned. Nevertheless, we argue that a model type that demonstrates good performance across diverse inputs holds inherent value, thus justifying the comparisons made in our study.

Furthermore, it is important to acknowledge that we did not perform a dedicated process of model evaluation or selection on the ModelNet10 data set. Instead, our model selection was based on the performance observed on a different data set, with the primary aim of comparing our approach to established baselines using well-known data. Consequently, there is potential for performance improvements across all models when evaluated directly on the ModelNet10 data set. Future studies should consider conducting rigorous model evaluation and selection procedures tailored specifically to the target data set.

ModelNet10

ModelNet10 data set consists of shapes from which we extract point clouds for classification tasks. Remember from 5.3.1 most of the underlying object shapes in ModelNet10 are simply connected spaces, which means that topological approaches, in general, treat them as equivalent. Consequently, ModelNet10 may not be the most suitable data set to benchmark our models specifically from a topological perspective. This hypothesis is strengthened by the fact that a small subset size of $k = 5$ outperformed the other parameters on ModelNet10 while larger values like $k = 100$ and $k = 50$ gave the best results in the pilot study with more topologically distinct classes.

To further strengthen the evaluation of our models, it would have been advantageous to compare their performance on data sets where other topological approaches have been applied. Such data sets would provide a more comprehensive understanding of the models' capabilities and their ability to capture and utilize topological information effectively.

By examining the performance of our models on data sets with diverse shapes and topological structures, we could gain valuable insights into their strengths and weaknesses in handling different types of data. This broader comparison would provide a more robust assessment of our models' performance and their suitability for various topological analysis tasks.

6.2.2 Strengths

End-to-End Learning

The DSSN framework enables end-to-end learning, meaning that the entire model, including the feature extraction and downstream processing components, can be jointly optimized during training. This end-to-end learning approach eliminates the need for manual feature engineering or handcrafted representations, which can be time-consuming and challenging in topological analysis tasks. By learning the optimal representation and processing steps directly from the data, the DSSN approach offers the potential for improved performance and efficiency in capturing and utilizing topological information.

Robustness and Stability

Although evaluating a limited set of networks across several data sets is unlikely to extract the full potential in terms of performance for all cases, models that perform well across a wide range of parameters show robustness which we think is a desirable trait in itself.

The DDPN network provides desired stability with respect to small perturbations in the input data as it only operates on continuous functions. This means that minor changes in the dataset, such as noise or small variations in the point cloud representation, do not drastically alter the persistence diagrams or the model's predictions. This stability property is advantageous in real-world applications where input data can be noisy or subject to small variations. It ensures that the model's predictions remain reliable and consistent even in the presence of such perturbations, enhancing the robustness and reliability of the approach.

Generality

The DSSN framework is built on a strong theoretical foundation, providing a completely general solution, suitable for a wide array of problem domains since it can approximate any valid set of set function. By minimizing restrictions on the solution space, the DSSN framework offers flexibility in handling different types of data and tasks. Additionally, it offers optionally for straightforward insertion of feature engineered, hand-crafted vectorizations that are already established within the TDA field.

6.3 Conclusion

This thesis has demonstrated the potential of distributed persistence as a topological invariant for effectively capturing the characteristics of noisy data in comparison to conventional full persistence methods in practical applications. We have successfully showcased how the distributional approach overcomes sensitivity challenges and provides valuable insights into complex data sets.

Through the introduction and application of our novel framework, DSSN (Deep Set of Set Network), we have illustrated the practical utilization of distributed persistence. While our results on the ModelNet10 data set do not reach the standard set by other

state-of-the-art approaches, we have convincingly outperformed our baselines, indicating the effectiveness of our proposed approach for topological analysis.

By integrating DeepSets with our specialized architecture, DDPN (Deep Distributed Persistence Network), we have successfully improved the performance of our framework. This integration has also demonstrated the potential benefits of employing trainable vectorizations when sufficient data is available.

Furthermore, leveraging the performance of the supervised model, we have gained a deeper understanding of the relationship between model performance and the configuration of distributed persistence.

Chapter 7

Future Work

The results of the experiments conducted in this thesis strongly indicate that a distributed approach to persistent homology computation in a supervised learning framework outperforms the traditional full persistence computation for radius-based methods.

However, additional comparative studies are required in order to establish a strong empirical foundation for this statement. Particularly, we would have liked to evaluate our approach on more topologically separable data sets especially ones where topologically focused approaches have previously been applied successfully. This is both to substantiate the results from our experiments, but also to see if this could establish new state-of-the-art performances on such data sets. Additionally, a more extensive comparison that covers a wider range of hyper-parameters for all architectures would greatly strengthen the conviction of our results.

Furthermore, the possibility of utilizing the strengths of transfer learning is intriguing for these distributed approaches, especially given the observations made around varying the parameter value m during training and validation. Experimenting with how the feature extractors in particular could transfer their learned representations to other models with varying parameters for the distributed persistence could potentially increase the efficiency and generalization ability of this method.

During our work on the experiments of this thesis, it has become apparent that there are many hyper-parameters and architectural features of the DeepSet architecture that are not understood sufficiently. Some of our discoveries are covered in chapter 6, but a detailed survey investigating these in further detail would help progress the work with DeepSets substantially. Especially the embedding sizes of the feature extractors, the

balance of expressivity of the different networks as well as the need for a high learning rate are among the most interesting questions to answer.

Lastly, there is an open question as to how the inclusion of labels for each subset would affect the performance of this approach, especially for smaller subset sizes. After all, the theoretical foundation laid down in [31] was based on labeled subsets rather than the ones considered here. How to introduce this information in practice is a difficult problem, particularly if included as exact labels. However, we conjecture that some useful information about the relationship between subsets can be included through a similarity measure such as the Hausdorf-distance [ref] or the Earth Mover-distance. The relative distance between all subsets would provide enough information to construct a fully connected graph. A Graph Neural Network over a DeepSet extractor would allow for communication between the vectorized persistence diagrams about the similarity of their points. For small subsets in particular this might be useful information as they might often be sampled from very different regions of the underlying point cloud.

Bibliography

[1] Supersimple.com.
URL: https://supersimple.com/article/butterfly-symmetry/.

[2] Pointcloud-c — project page.
URL: https://pointcloud-c.github.io/home.html.

[3] Gudhi library – topological data analysis and geometric inference in higher dimensions.
URL: https://gudhi.inria.fr/.

[4] Princeton modelnet.
URL: https://modelnet.cs.princeton.edu/.

[5] Setup — tadasets 0.1.0 documentation.
URL: https://tadasets.scikit-tda.org/en/latest/.

[6] Henry Adams, Tegan Emerson, Michael Kirby, Rachel Neville, Chris Peterson, Patrick Shipman, Sofya Chepushtanova, Eric Hanson, Francis Motta, and Lori Ziegelmeier. Persistence images: A stable vector representation of persistent homology. *Journal of Machine Learning Research*, 18(8):1–35, 2017.
URL: http://jmlr.org/papers/v18/16-337.html.

[7] Dashti Ali, Aras Asaad, Maria Jose Jimenez, Vidit Nanda, Eduardo Paluzo-Hidalgo, and Manuel Soriano-Trigueros. A survey of vectorization methods in topological data analysis.
URL: https://github.com/dashtiali/vectorisation-app.

[8] N Atienza, L M Escudero, M J Jimenez, and M Soriano-Trigueros. Characterising epithelial tissues using persistent entropy *. 2021.

[9] Anselm Blumer, Andrzej Ehrenfeucht, David Haussler, and Manfred Warmuth. Occam's razor. *Information Processing Letters*, 24(6):377–380, 1987.

[10] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. Geometric deep learning: Grids, groups, graphs, geodesics, and gauges. *CoRR*, abs/2104.13478, 2021.
URL: https://arxiv.org/abs/2104.13478.

[11] Jodi L. Bubenik, Melissa Hale, Ona Mcconnell, Eric T. Wang, Maurice S. Swanson, Robert C. Spitale, and J. Andrew Berglund. Rna structure probing to characterize rna–protein interactions on low abundance pre-mrna in living cells. *RNA*, 27:343, 3 2021. ISSN 14699001. doi: 10.1261/RNA.077263.120.
URL: /pmc/articles/PMC7901844//pmc/articles/PMC7901844/?report=abstracthttps://www.ncbi.nlm.nih.gov/pmc/articles/PMC7901844/.

[12] T Tony Cai and Rong Ma. Theoretical foundations of t-sne for visualizing high-dimensional clustered data. *Journal of Machine Learning Research*, 23:1–54, 2022.
URL: http://jmlr.org/papers/v23/21-B0524.html.

[13] Gunnar Carlsson. Topology and data. *Bulletin of The American Mathematical Society - BULL AMER MATH SOC*, 46:255–308, 04 2009. doi: 10.1090/S0273-0979-09-01249-X.

[14] Mathieu Carrière, Frédéric Chazal, Yuichi Ike, Théo Lacombe, Martin Royer, and Yuhei Umeda. Perslay: A neural network layer for persistence diagrams and new graph topological signatures. 2020.
URL: https://github.com/MathieuCarriere/perslay.

[15] Frédéric Chazal, Brittany Terese Fasy, Fabrizio Lecci, Bertrand Michel, Alessandro Rinaldo, and Larry Wasserman. Subsampling methods for persistent homology. 2014.

[16] Ethan Coldren. On vietoris-rips complexes: the persistent homology of cyclic graphs.

[17] Scott Fortmann-Roe. Understanding the bias-variance tradeoff. 2013.
URL: https://scott.fortmann-roe.com/docs/BiasVariance.html.

[18] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks.
URL: http://www.iro.umontreal.

[19] Jonathan L. Gross and Jay Yellen. *Graph Theory and Its Applications*. CRC Press, 2006.

[20] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[21] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2:359–366, 1 1989. ISSN 0893-6080. doi: 10.1016/0893-6080(89)90020-8.

[22] Suen Wun Ki, Christine Supervisor, Wai Yeung, Yan Moderator, and Bo Wu. Persistent homology: Hole detection in lidar point clouds with topological data analysis. 2021.

[23] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[24] Stéphane Mallat. Understanding deep convolutional networks. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065):20150203, 2016.

[25] Leland Mcinnes, John Healy, and James Melville. Umap: Uniform manifold approximation and projection for dimension reduction. 2020.

[26] PyTorch. Pytorch documentation: torch.nn.crossentropyloss. `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`, 2021. Accessed: May 26, 2023.

[27] Yohai Reani and Omer Bobrowski. A coupled alpha complex. 2021.

[28] Mark Schmidt. Gradient descent progress bound gradient descent convergence rate cpsc 540: Machine learning convergence of gradient descent. 2017.

[29] Gurjeet Singh, Facundo Mémoli, and Gunnar Carlsson. Topological methods for the analysis of high dimensional data sets and 3d object recognition. *Eurographics Symposium on Point-Based Graphics*, 2007.

[30] Maximilian Soelch, Adnan Akhundov, Patrick van der Smagt, and Justin Bayer argmaxai. On deep set learning and the choice of aggregations.

[31] Elchanan Solomon, Alexander Wagner, and Paul Bendich. From geometry to topology: Inverse theorems for distributed persistence.
URL: `https://arxiv.org/abs/2101.12288`.

[32] Manzil Zaheer, Satwik Kottur, Siamak Ravanbhakhsh, Barnabás Póczos, Ruslan Salakhutdinov, and Alexander J Smola. Deep sets.

[33] Haoxu Zhang, Chenchen Qiu, Chao Wang, Bin Wei, Zhibin Yu, Haiyong Zheng, and Juan Li. Learning spectral normalized adversarial systems with stacked structure for high-quality 3d object generation. *Concurrency and Computation: Practice and Experience*, 33(15):e5430, 2021. doi: https://doi.org/10.1002/cpe.5430.
URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5430`. e5430 cpe.5430.

Appendix A

Appendix

A.0.1 Hyper-parameters of models in Pilot

Table A.1: Selected hyper-parameters for STAT-M models in Pilot.

	STAT-M		
k \ m	1	100	1000
2	(64, 0.001)	(64, 0.001)	(16, 0.001)
5	(16, 0.001)	(64, 0.01)	(64, 0.01)
50	(16, 0.0)	(64, 0.001)	(64, 0.001)
100	(64, 0.01)	(64, 0.0)	(64, 0.0)
500	(64, 0.0)	(16, 0.0)	-
FULL	0	-	-

Table A.2: Selected hyper-parameters for STAT-DS models in Pilot.

	STAT-M		
k \ m	1	100	1000
2	(64, 0.0)	(16, 0.0)	(16, 0.0)
5	(16, 0.001)	(64, 0.0)	(64, 0.001)
50	(16, 0.0)	(64, 0.001)	(16, 0.001)
100	(16, 0.001)	(16, 0.0)	(64, 0.0)
500	(64, 0.0)	(64, 0.0)	-
FULL	0	-	-

Table A.3: Selected hyper-parameters for DDPN models in Pilot.

	STAT-M		
k \ m	1	100	1000
2	(16, 0.001)	(16, 0.001)	(16, 0.0)
5	(64, 0.01)	(16, 0.001)	(64, 0.001)
50	(16, 0.001)	(16, 0.0)	-
100	-	-	-
500	-	-	-
FULL	0	-	-

A.0.2 Networks

Table A.4 shows the feature extractor ϕ of the DDPN network.

Layer No.	Nodes	Activation
Linear 1	12	LeakyReLU
Linear 1	16	-

Table A.4: Fully connected network, ϕ.

The w network consists of only a single layer without any non-linearities. Bias is crucially not included as a parameter in this network.

Layer No.	Nodes	Activation
Linear 1	16	-

Table A.5: Fully connected network, w.

The shared downstream network of both STAT approaches, ρ_1^2 is shown in table A.6.

Layer No.	Nodes	Activation
Linear 1	128	ReLU
BatchNorm 1	-	-
Linear 2	64	ReLU
BatchNorm 2	-	-
Linear 3	64	ReLU
BatchNorm 3	-	-
Linear 4	32	ReLU
BatchNorm 4	-	-
Linear 5	5	-

Table A.6: Fully connected network, ρ_1^2.

Table A.7 shows the downstream network of the DDPN architecture.

Layer No.	Nodes	Activation
BatchNorm 1	-	-
Linear 1	32	LeakyReLU
BatchNorm 2	-	-
Linear 3	5	LeakyReLU

Table A.7: Fully connected network, ρ_1^3.

The feature extractor of the STAT-DS network, ρ_2^1:

ρ_2^1

Layer No.	Nodes	Activation
Linear 1	64	LeakyReLU
Linear 2	64	LeakyReLU

Table A.8: Fully connected network, ρ_2^1.

ρ_2^2 acts both as a part of the downstream network of the inner vectorization in the DDPN and as a feature extractor of the outer vectorization.

Layer No.	Nodes	Activation
Linear 1	40	LeakyReLU
Linear 2	64	-

Table A.9: Fully connected network, ρ_2^2.

The details of this CNN architecture can be seen in figure A.10. The inputs are persistence images computed using GUDHI [3] with the following hyper-parameters: bandwidth = 0.1, resolution = 20, range = 2.

Layer No.	Channels/Nodes	Kernel Size	Padding	Stride	Activation
Convolution 1	6	5	2	2	ReLU
BatchNorm 1					
Convolution 2	6	5	1	1	ReLU
BatchNorm 2					
Convolution 3	6	5	0	1	ReLU
BatchNorm 3					
Convolution 4	6	3	0	1	ReLU
Flatten					
Linear 1	32				ReLU
BatchNorm 4					
Linear 2	5				-

Table A.10: CNN architecture for ρ_1^1.