Neural Approximation of Graph Topological Features

Zuoyu Yan
Wangxuan Institute of Computer Technology
Peking University
yanzuoyu3@pku.edu.cn

Tengfei Ma
IBM T. J. Watson Research Center
tengfei.ma1@ibm.com

Liangcai Gao
Wangxuan Institute of Computer Technology
Peking University
glc@pku.edu.cn

Zhi Tang
Wangxuan Institute of Computer Technology
Peking University
tangzhi@pku.edu.cn

Yusu Wang
Halıcıoğlu Data Science Institute
University of California
yusuwang@ucsd.edu

Chao Chen*
Department of Biomedical Informatics
Stony Brook University
chao.chen.1@stonybrook.edu

Abstract

Topological features based on persistent homology capture high-order structural information so as to augment graph neural network methods. However, computing extended persistent homology summaries remains slow for large and dense graphs and can be a serious bottleneck for the learning pipeline. Inspired by recent success in neural algorithmic reasoning, we propose a novel graph neural network to estimate extended persistence diagrams (EPDs) on graphs efficiently. Our model is built on algorithmic insights, and benefits from better supervision and closer alignment with the EPD computation algorithm. We validate our method with convincing empirical results on approximating EPDs and downstream graph representation learning tasks. Our method is also efficient; on large and dense graphs, we accelerate the computation by nearly 100 times.

1 Introduction

Graph neural networks (GNNs) have been widely used in various domains with graph-structured data [55, 35, 29, 52, 7]. Much effort has been made to understand and to improve graph representation power [57, 39, 3, 36]. An intuitive solution is to explicitly inject high order information, such as graph topological/structural information, into the GNN models [62, 34]. To this end, persistent homology [19, 18], which captures topological structures (e.g., connected components and loops) and encodes them in a summary called *persistence diagram (PD)*, have attracted the attention of researchers. Indeed, persistence has already been injected to machine learning pipelines for various graph learning tasks [66, 67, 21, 5, 9, 61]. In particular, it has been found helpful to use the so-called *extended persistence diagrams (EPDs)* [11], which contain richer information than the standard PDs.

Despite the strong learning power of PDs and EPDs, their computation remains a bottleneck in graph learning. In situations such as node classification [67] or link prediction [61], one has to compute EPDs on vicinity graphs generated around all the nodes or all possible edges in the input graph. This can be prohibitive especially for large and dense graphs. Take the Amazon Computers dataset [47]

*Correspondence to Chao Chen, Yusu Wang, and Liangcai Gao

36th Conference on Neural Information Processing Systems (NeurIPS 2022).

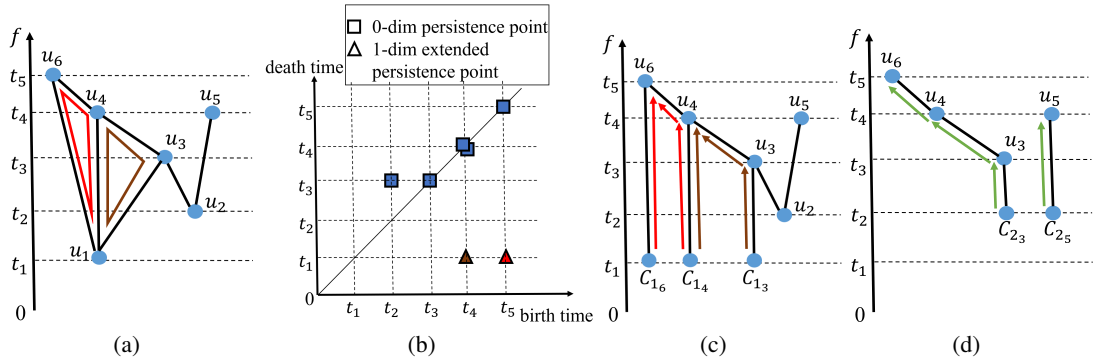


Figure 1: An explanation of extended persistent homology and its computation. The height-function-based filtration is only for illustration purposes. (a) The input graph is plotted with a given filter function. (b) the extended persistence diagram of (a). Commonly speaking, the persistence points on the diagonal (uncritical points) should not be plotted. We plot these points for a clearer illustration. (c) and (d) are examples of finding the loops in the input graph.

as an example. To compute EPDs on vicinity graphs take several seconds on average, and there are 13381 nodes. So to compute all EPDs with a single CPU can take up to a day. This is not surprising as a practical algorithm for computing EPDs is quadratic to the graph size [61].

These computational difficulties raise the question: *can we approximate the expensive computation of EPDs using an efficient learning-based approach?* This is a very challenging question due to the highly complex and unusual mathematical machinery behind the original algorithm. First, the algorithm involves a reduction algorithm of the graph incidence matrix. Each step of the algorithm is a modulo-2 addition of columns that can involve edges and nodes far apart. Such algorithm can hardly be approximated by a direct application of the black-box deep neural networks.

The second challenge comes from the supervision. The output EPD is a point set with an unknown cardinality. The distance between EPDs, called the *Wasserstein distance* [10, 12], involves a complex point matching algorithm. It is highly nontrivial to design a deep neural network with variable output and to support supervision via such Wassserstein distance. Previous attempts [48, 37] directly use black-box neural networks to generate fixed-length vectorization of the PDs/EPDs and use mean squared error or cross-entropy loss for supervision. The compromise in supervision and the lack of control make it hard to achieve high-quality approximation of PDs/EPDs.

In this paper, we propose a novel learning approach to approximate EPDs on graphs. Unlike previous attempts, we address the aforementioned challenges through a carefully designed learning framework guided by several insights into the EPD computation algorithm.

In terms of model output and supervision, we observe that the computation of EPDs can be treated as an edge-wise prediction instead of a whole-graph prediction. Each edge in the graph is paired with another graph element (either vertex or edge), and the function values of the pair are the coordinates of a persistence point in the EPD. This observation allows us to compute EPDs by predicting the paired element for every edge of the graph. The Wasserstein distance can be naturally decomposed into supervision loss for each edge. This element-wise supervision can significantly improve learning efficiency compared with previous solutions, which treat PDs/EPDs as a whole-graph representation and have to use whole-graph representation pooling.

Another concern is whether and how a deep neural network can approximate the sophisticated EPD algorithm. To this end, we redesign the algorithm so that it is better aligned with algorithms that are known to be learnable by neural networks. Recall we observe that computing EPDs can be decomposed into finding pairing for each edge. We show that the decomposition is not only at the output level, but also at the algorithm level. The complex standard EPD computation algorithm can indeed be decomposed into independent pairing problems, each of which can be solved exactly using a classic *Union-Find algorithm* [13]. To this end, we draw inspiration from recent observations that neural networks can imitate certain categories of sequential algorithms on graphs [53, 56]. We propose a carefully designed graph neural network with specific message passing and aggregation mechanism to imitate the Union-Find algorithm.

Decomposing the algorithm into Union-Find subroutines and approximating them with a customized GNN provide better alignment between our neural network and the EPD algorithm. A better alignment guarantees better performance [58]. Empirically, we validate our method by quantifying its approximation quality of the EPDs. On two downstream graph learning tasks, node classification and link prediction, we also show that our neural approximations are as effective as the original EPDs. Meanwhile, on large and dense graphs, our method is much faster than direct computation. In other words, the approximated EPDs do not lose accuracy and learning power, but can be computed much more efficiently. Finally, we observe that our model can be easily transferred to unseen graphs, perhaps due to the close imitation of the Union-Find subroutine. This is encouraging as we may generalize topological computation to various challenging real-world graphs without extra effort.

In summary, we propose an effective learning approach to approximate EPDs with better supervision and better transparency. The technical contributions are as follows.

- We reformulate the EPD computation as an edge-wise prediction problem, allowing better supervision and more efficient representation learning.
- We show that the EPD computation can be decomposed into independent pairing problems, each of which can be solved by the Union-Find algorithm.
- Inspired by recent neural algorithm approximation works [53, 56], we design a novel graph neural network architecture to learn the Union-Find algorithm. The closer algorithmic alignment ensures high approximation quality and transferability.

2 Background: Extended Persistent Homology

We briefly introduce extended persistent homology and refer the readers to [11, 18] for more details.

Ordinary Persistent Homology. Persistent homology captures 0-dimensional (connected components) and 1-dimensional (loops) topological structures and measures their saliency via a scalar function called *filter function*. Given a input graph $G = (V, E)$, with node set V and edge set E, we call all the nodes and edges *simplices*. Denote by $X = V \cup E$ the set of all simplices. We define a filter function on all simpices, $f : X \to \mathbb{R}$. Often, f is induced by a node-valued function (e.g., node degrees), and further defined on edges as $f(uv) = max(f(u), f(v))$.

Denote by X_a the *sublevel set of* X, consisting of simplices whose filter function values $\leq a$, $X_a = \{x \in X | f(x) \leq a\}$. As the threshold value a increases from $-\infty$ to ∞, we obtain a sequence of growing spaces, called an *ascending filtration* of X: $\emptyset = X_{-\infty} \subset ... \subset X_{\infty} = X$. As X_a increases from $\emptyset$ to X, new topological structures gradually appear (born) and disappear (die). For instance, the blue square persistence point at (t_2, t_3) in Figure 1 (b) indicates that the connected component u_2 appears at X_{t_2} and is merged with the whole connected component at X_{t_3}.

Applying the homology functor to the filtration, we can more precisely quantify the birth and death of topological features (as captured by homology groups) throughout the filtration, and the output is the so-called *persistence diagram (PD)*, which is a planar multiset of points, each of which (b, d) corresponds to the birth and death time of some homological feature (i.e., components, loops, and their higher dimensional analogs). The lifetime $|d - b|$ is called the *persistence* of this feature and intuitively measures its importance w.r.t. the input filtration.

Extended Persistent Homology. In the ordinary persistent homology, topology of the domain (e.g., the graph) will be created at some time (has a birth time), but never dies (i.e., with death time being equal to $+\infty$). Hence we cannot capture their importance. In the context of graphs, the importance of loops are not captured via the ordinary persistence. To this end, *extended persistence* [11][2] introduces a *descending filtration* : $\emptyset = X^{\infty} \subset ... \subset X^{-\infty} = X$, where $X^a = \{x \in X | f(x) \geq a\}$ is a *superlevel set*. This induces a sequence of homology groups of the form $\mathrm{H}(X) = \mathrm{H}(X, X^{\infty}) \to \cdots \to \mathrm{H}(X, X^a) \to \cdots \to \mathrm{H}(X, X^{-\infty}) = \emptyset$.

When the input domain is a graph, the 1D EPD consists of β_1 number of persistence points, capturing the birth and death of independent loop features. Here β_1 is the rank of the first homology group; for a connected graph, it is simply the number of independent loops, $\beta_1 = |E| - |V| + 1$. A loop will be created during the ascending filtration, but killed during the descending filtration. The birth and death times of the feature correspond to the threshold value a's when these events happen. In

[2]Extended persistence is also closely related to the concept of zigzag persistence [4].

general, the death time for such loop feature is smaller than the birth time. For example, the red triangle persistence point in Figure 1 (b) denotes that the red cycle in Figure 1 (a) appears at X_{t_5} in the ascending filtration and appears again at X^{t_1} in the descending filtration.

Finally, PDs live in an infinite-dimensional space equipped with an appropriate metric structure, such as the so-called p-th Wasserstein distance [12] or the bottleneck distance [10]. They have been combined with various deep learning methods including kernel machines [44, 32, 6], convolutional neural networks [23, 27, 54, 68], transformers [64], connectivity loss [8, 22], and GNNs [67, 9, 61, 66, 21, 5]. During learning, there have been many works in the literature to vectorize persistence diagrams for downstream analysis. Among these works a popular choice is the persistence image [1].

3 Algorithm Revision: Decomposing EPD into Edge-Wise Paring Predictions

In this section, we provide algorithmic insights into how the expensive and complex computation of EPDs can be decomposed into pairing problems for edges. And each pairing problem can be solved exactly using a Union-Find algorithm. The benefit is two-folds. First, the decomposition makes it possible to train the neural network through edge-wise supervision and use the original Wasserstein distance of EPDs. This allows us to adopt the popular and effective edge-prediction GNN for the goal. Second, we observe the similarity between the Union-Find and sequential algorithms which are known to be learnable by neural networks. This gives us the opportunity to design a special graph neural network to imitate the algorithm accurately, and to approximate EPDs accurately.

Decompose the EPD Computation into Pairing Computations. Recall that our goal is to compute the 0D and 1D EPDs PD_0 and PD_1. We observe that on these EPDs, each point corresponds to a unique pairing of graph elements (vertex-edge pair for 0D EPD, edge-edge pair for 1D EPD). Each pair of elements are essentially the "creator" and "destroyer" of the corresponding topological feature during the filtration. And their filtration values are the birth and death times of the topological feature. For example, the persistence point located at (t_2, t_3) in Figure 1 (b) denotes that the edge u_2u_3 is paired with u_2. Indeed, these pairings cover all elements in the graph; every edge/node belongs to a unique pairing. We note that this is not in conflict with the fact that the PDs/EPDs are often sparse. Many pairings are local and only pair adjacent elements. They correspond to zero-persistence points living in the diagonal of the EPDs.

This pairing view gives us the opportunity to transform the computation of EPDs into a pairing prediction problem: for every edge in the graph, we predict its pairing element. This will be the foundation of our design of the GNN in Sec. 4. Meanwhile, we observe that *the decomposition is not only at the output level*. The original algorithm of EPD, a sequential modulo-2 matrix reduction algorithm, can indeed be rewritten into a set of independent algorithm subroutines, each for the computation of one pairing. Each subroutine is a Union-Find algorithm. This new decomposed EPD algorithm has not been reported before, although the idea follows from existing work [2]. For completeness, we will provide a proof of correctness of the algorithm.

Description of Algorithm 2. The pseudocode for 1D EPD computation is shown in Algorithm 2. We leave the algorithm for 0D to the supplementary material[3]. For simplicity of presentation, we assume that all vertices have distinct function values $f : V \to \mathbb{R}$[4]. Therefore finding the persistence value equals to finding the pairing. To compute the EPD, we traverse all nodes in the vertex set and find their extended persistence pairing. Combining the persistence pair from all nodes, we can obtain the final EPD. The algorithm complexity analysis is provided in the supplementary material.

Finding persistence pairing for nodes. For node $u_i \in V$, we can call Algorithm 3 to identify the corresponding persistence pair. In particular, the algorithm first sorts the graph elements according to an input scalar function, then does the edge operation by finding the roots of the corresponding nodes and merging these nodes. See Figure 1(c) for a simple illustration. For node u_1, there are three upper edges: u_1u_3, u_1u_4, and u_1u_6. We put each such edge u_iu_j in a different component C_{i_j}, – we call this *upper-edge splitting operation* – and start to sweep the graph in increasing values starting at $f(u_i)$. Then, the first time any two such components merge will give rise to a new persistence point in the 1D EPD. For instance, C_{1_4} and C_{1_3} first merge at u_4, and this will give rise to the brown

[3]The 0D algorithm needs a single run of Union-Find [18, 16], and is very similar to Algorithm 3 which is a subroutine used by Algorithm 2.

[4]We can add jitter to the original filter function. The output EPDs will only have minor changes [10]

Algorithm 1 Sequential algorithm

Input: graph $G = (V, E)$, filter function f.
Initialise-Nodes(V, f)
$Q =$ Sort-Queue(V)
while Q is not empty **do**
  $u = Q$.pop-min()
  for $v \in G$.neighbors(u) **do**
    Relax-Edge(u, v, f)
  end for
end while

Algorithm 2 Computation of 1D EPD

Input: filter function f, input graph $G = (V, E)$
$V, E =$ sorted(V, E, f)
$PD_0 =$ Union-Find(V, E, f), $PD_1 = \{\}$
for $i \in V$ **do**
  $C_i = \{C_{i_j} | (i, j) \in E, f(j) > f(i)\}$, $E_i = E$
  for $C_{i_j} \in C_i$ **do**
    $f(C_{i_j}) = f(i)$, $E_i = E_i - \{(i, j)\} + \{(C_{i_j}, j)\}$
  end for
  $PD_1^i =$ Union-Find-step($V + C_i - \{i\}, E_i, f, C_i$)
  $PD_1 += PD_1^i$
end for
Output: PD_0, PD_1

Algorithm 3 Union-Find-step (Sequential)

Input: V, E, f, C_i
$PD_1^i = \{\}$
for $v \in V$ **do**
  $v.value = f(v)$, $v.root = v$
end for
$Q =$ Sort(V), $Q = Q - \{v | f(v) < f(i)\}$
while Q is not empty **do**
  $u = Q$.pop-min()
  for $v \in G$.neighbors(u) **do**
    $pu, pv =$ Find-Root(u), Find-Root(v)
    if $pu \neq pv$ **then**
      $s = argmin(pu.value, pv.value)$
      $l = argmax(pu.value, pv.value)$
      $l.root = s$
      if $pu \in V_i$ and $pv \in V_i$ **then**
        $PD_1^i + \{(u.value, l.value)\}$
      end if
    end if
  end for
end while
Function: Find-Root(u)
$pu = u$
while $pu \neq pu.root$ **do**
  $pu.root = (pu.root).root$, $pu = pu.root$
end while
Return: pu

loop in Figure 1(a) with (t_4, t_1) as its persistence point. While in Figure 1 (d), the two connected components, C_{2_3} and C_{2_5} (originated from u_2) will not be united. Therefore, node u_2 will not lead to any persistence point in the EPD.

Correctness. The following theorem states that the decomposed algorithm is correct. We provide a sketch of the proof, leaving the complete proof to the supplementary material.

Theorem 3.1. *Algorithm 2 outputs the same 1D EPDs as the standard EPD computation algorithm.*

Proof sketch. To compute the 1D EPDs, we simply need to find the pairing partner for all edges. Therefore, to prove that the two algorithms output the same 1D EPDs, we need to prove that the output pairing partners are the same (or share the same filter value). We prove this by showing that both the standard EPD computation algorithm and Algorithm 2 find the "thinnest pair", i.e., the paired saddle points are with the minimum distance in terms of filter value, for all edges.

Neural Approximation of Union-Find. In the previous paragraph, we showed that the computation of EPDs can be decomposed into the parallel execution of Union-Find algorithms, which share a similar sequential behavior. This gives us the opportunity to approximate these Union-Find algorithms well, and consequently approximate EPDs well.

Approximating algorithms with neural networks is a very active research direction [63, 28, 31, 43, 45, 59]. Within the context of graph, GNNs have been proposed to approximate parallel algorithms (e.g., Breadth-First-Search) and sequential algorithms (e.g., Dijkstra) [53, 51, 56]. Particularly relevant to us is the success in approximating the category of sequential algorithms such as Dijkstra. These sequential algorithms, as generally defined in Algorithm 1, sort graph elements (vertices and edges) according to certain function, and perform algorithmic operations according to the order. As described in previous paragraphs, the Union-Find algorithm also contains these steps, and can be expressed in a sequential-like form (Algorithm 3). Therefore we propose a framework to simulate the algorithm.

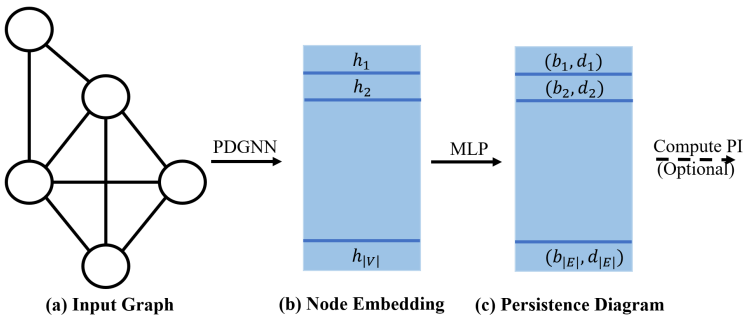


Figure 2: The basic framework.

4 A Graph Neural Network for EPD Approximation

Previous section establishes the algorithm foundation by showing that we can decompose EPD computation into edge pairing prediction problems, each of which can be solved using a Union-Find algorithm. Based on such algorithmic insights, we next introduce our neural network architecture to approximate the EPDs on graphs. Our main contributions are: (1) we transform the EPD computation into an edge-wise prediction problem, and solve it using a GNN framework, inspired by the GNN for link prediction; (2) we design a new backbone GNN model **PDGNN** to approximate the Union-Find algorithm, with specially designed pooling and message passing operations.

4.1 EPD computation as a edge-wise prediction problem

We have established that computing PD_0 and PD_1 can be reduced into finding the pairing partners for all edges. We transfer the problem into an edge-wise prediction problem. We predict the persistence pairing for all edges. This is very similar to a standard link prediction problem [7, 61], in which one predicts for each node pair of interest whether it is a real edge of the graph or not.

Inspired by standard link-prediction GNN architectures [7, 61], we propose our model (see Figure 2) as follows. (1) For an input graph $G = (V, E)$ and a filter function f, we first obtain the initial filter value for all the nodes: $X = f(V) \in R^{|V|*1}$, and then use a specially designed GNN model which later we call PDGNN $\mathcal{G}$ to obtain the node embedding for all these vertices: $H = \mathcal{G}(X) \in R^{|V|*d_H}$. (2) Subsequently, a MLP (Multi-layer perceptron) W is applied to the node embeddings to obtain a two dimensional output for each edge $(u, v) \in E$, corresponding to its persistence pairing. Formally, we use $PP_{uv} = W([h_u \bigoplus h_v]) \in R^2$ as the persistence pair. Here, h_u and h_v denote the node embedding for node u and v, and $\bigoplus$ represents the concatenation of vectors.

Given that the Union-Find-step algorithm should be implemented on all edges to obtain the EPDs. Ideally we need a large GNN model to simulate all Union-Finds at the same time. However, these Union-Find-step algorithms are running in parallel on the same graph. There are many overlapping or similar computational steps between computations on different nodes. Besides, the node ordering are exactly the same. Therefore we expect a limited-size single GNN model can simultaneously approximate all EPD computations. This is confirmed by empirical evidence in Section 5.

4.2 PDGNN

In this section, we explain how to design the backbone GNN to approximate the Union-Find algorithm. Note the Union-Find is similar to known sequential algorithms but with a few exceptions. We design specific pooling and message passing operations to imitate these special changes. These design choices will be shown to be necessary in the experiment section.

Recall a typical GNN learns the node embedding via an iterative aggregation of local graph neighbors. Following [57], we write the k-th iteration (the k-th GNN layer) as:

$$h_u^k = AGG^k(\{MSG^k(h_v^{k-1}), v \in N(u)\}, h_u^{k-1}) \tag{1}$$

where h_u^k is the node features for node u after k-th iterations, and $N(u)$ is the neighborhood of node u. In our setting, $h_u^0 = x_u$ is initialized to be the filter value of node u. Different GNNs have different MSG and AGG functions, e.g., in GIN [57], the message function MSG is a MLP followed by an activation function, and the aggregation function AGG is a sum aggregation function.

Table 1: Approximation error on different vicinity graphs

Dataset	Cora		Citeseer		PubMed		Photo		Computers	
Evaluation	W_2	PIE	W_2	PIE	W_2	PIE	W_2	PIE	W_2	PIE
GIN_PI	—	5.03e-1	—	2.17e-1	—	4.08e-1	—	5.53	—	2.70
GAT_PI	—	1.43e-1	—	1.95e-1	—	1.60	—	20.98	—	44.50
GAT	0.655	2.46e-2	0.431	4.04e-2	0.697	3.5e-1	1.116	1.09	1.145	2.21
GAT (+MIN)	0.579	1.53e-2	0.344	1.02e-2	0.482	4.60e-2	0.820	1.35	0.834	0.64
PDGNN (w/o ew)	0.692	2.77e-2	0.397	2.24e-2	0.666	9.01e-2	2.375	6.47	18.63	27.35
PDGNN	**0.241**	**4.75e-4**	**0.183**	**4.43e-4**	**0.256**	**8.95e-4**	**0.224**	**4.33e-3**	**0.220**	**6.20e-3**

Table 2: Classification accuracy on various node classification benchmarks

Method	Cora	Citeseer	PubMed	Computers	Photo	CS	Physics
GCN	81.5±0.5	70.9±0.5	79.0±0.3	82.6±2.4	91.2±1.2	91.1±0.5	92.8±1.0
GAT	**83.0±0.7**	**72.5±0.7**	79.0±0.3	78.0±19.0	85.1±20.3	90.5±0.6	92.5±0.9
HGCN	78.0±1.0	68.0±0.6	76.5±0.6	82.1±0.0	90.5±0.0	90.5 ± 0.0	91.3±0.0
PEGN (True Diagram)	82.7±0.4	71.9±0.5	**79.4±0.7**	86.6±0.6	**92.7±0.4**	**93.3±0.3**	**94.3±0.1**
PEGN (GIN_PI)	81.8±0.1	65.7±2.1	77.7±0.9	82.4±0.5	88.3±0.7	92.6±0.3	93.7±0.5
PEGN (PDGNN)	82.0±0.5	70.8±0.5	78.7±0.6	**86.7±0.9**	92.2±0.2	93.2±0.2	94.2±0.2

We now describe our specially designed GNN, called **PDGNN** (Persistence Diagram Graph Neural Network). Compared with the Sequential algorithms (Algorithm 1) [56], our Union-Find algorithm (Algorithm 3) differs in: (1) the Find-Root algorithm which needs to return the minimum of the component, (2) additional edge operations such as upper-edge splitting. To handle these special algorithmic needs, our PDGNN modifies standard GNNs with the following modules.

A new aggregation due to the Find-Root function. Finding the minimum intuitively suggests using a combination of several local min-aggregations. Considering that the sum aggregation can bring the best expressiveness to GNNs [57], we implement the root-finding process by a concatenation of sum aggregation and min aggregation as our aggregation function. To be specific:

$$AGG^k(.) = SUM(.) \bigoplus MIN(.) \tag{2}$$

Improved edge operations. As shown in [53, 56], classic GNNs are not effective in "executing" Relax-Edge subroutines. Furthermore, in Algorithm 2, we also need the upper-edge splitting operation for each vertex. In other words, the information of the separated components C_{i_j} are formed by the information from both nodes u_i and u_j. To this end, we use edge features and attention to provide bias using edges. Specifically, we propose the following message function in the k-th iteration:

$$MSG^k(h_v^{k-1}) = \sigma^k[\alpha_{uv}^k(h_u^{k-1} \bigoplus h_v^{k-1})W^k] \tag{3}$$

where σ^k is an activation function, W^k is a MLP module, and α_{uv}^k is the edge weight for uv. We adopt $PRELU$ as our activation function, and the edge weight proposed in [52] as our edge weight.

Training PDGNN. We use the 2-Wasserstein distance between the predicted diagram and the ground truth EPD as the loss function. Through optimal matching, the gradient is passed to each predicted persistence pair. Since we have established the one-to-one correspondence between pairs and edges, the gradient is then passed to the corresponding edge, and contributes to the representation learning.

Table 3: AUC-ROC score on various link prediction benchmarks

Method	Cora	Citeseer	PubMed	Photo	Computers
GCN	90.5± 0.2	82.6±1.9	89.6±3.7	91.8±0.0	87.8±0.0
GAT	72.8± 0.2	74.8±1.5	80.3±0.0	92.9±0.3	86.4±0.0
HGCN	93.8±0.1	**96.6±0.1**	96.3±0.0*	95.4±0.0	93.6±0.0
P-GNN	74.1±2.4	73.9±2.6	79.6±0.5	90.9±0.7	88.3±1.0
SEAL	91.3±5.7	89.8±2.3	92.4±1.2	97.8±1.3	96.8±1.5
TLC-GNN (True Diagram)	94.9±0.4	95.1± 0.7	**97.0±0.1**	98.2±0.1	97.9±0.1
TLC-GNN (GIN_PI)	93.5±0.2	93.3±0.6	96.3 ± 0.2	95.8± 1.0	96.2±0.3
TLC-GNN (PDGNN)	**95.0±0.3**	95.6±0.4	**97.0±0.1**	**98.4±0.6**	**98.2±0.3**

Table 4: Time evaluation on different datasets (seconds)

Dataset	Cora	Citeseer	PubMed	Photo	Computers	CS	Physics
Avg. N/E	38/103	16/43	61/190	797/16042	1879/47477	97/431	193/1315
Fast [61]	0.95	0.39	2.15	362.60	1195.66	5.72	24.14
Gudhi [50]	**0.44**	**0.21**	**1.00**	583.55	8585.50	**3.00**	26.58
Ours	5.21	4.72	4.78	**6.67**	**7.32**	5.18	**5.42**

5 Experiments

In this section, we thoroughly evaluate the proposed model from 3 different perspectives. In Section 5.1, we evaluate the approximation error between the predicted diagram and the original diagram and show that the prediction is very close to the ground truth. Even with a small approximation error, we still need to know how much does the error influence downstream tasks. Therefore, in Section 5.2, we evaluate the learning power of the predicted diagrams through 2 downstream graph representation learning tasks: node classification and link prediction. We observe that the model using the predicted diagrams performs comparably with the model using the ground truth diagrams. In Section 5.3, we evaluate the efficiency of the proposed algorithm. Experiments demonstrate that the proposed method is much faster than the original algorithm, especially on large and dense graphs. Source code is available at https://github.com/pkuyzy/TLC-GNN.

Datasets. To compute EPDs, we need to set the input graphs and the filter functions. Existing state-of-the-art models on node classification [67] and link prediction [61] mainly focus on the local topological information of the target node(s). Following their settings, for a given graph $G = (V, E)$, we extract the k-hop neighborhoods of all the vertices, and extract $|V|$ vicinity graphs.

In terms of filter functions, we use Ollivier-Ricci curvature [41], heat kernel signature with two temprature values [49, 25] and the node degree[5]. For an input vicinity graph, we compute 4 EPDs based on the 4 filter functions, and then vectorize them to get 4 peristence images [1]. Therefore, we can get $4|V|$ EPDs in total. The input graphs include (1) citation networks including Cora, Citeseer, and PubMed [46]; (2) Amazon shopping datasets including Photo and Computers [47]; (3) coauthor datasets including CS and Physics [47]. Details are available in the supplementary material.

We note that in the literature, EPDs often contain the extended persistence point of the whole connected component. In our setting, we remove the point because that (1) no edge is paired with the whole connected component; (2) the value is easy to obtain, and does not need an extra prediction.

5.1 Approximation Quality

In this section, we evaluate the approximation error between the prediction and the original EPDs.

Evaluation metrics. Recall that the input of our model is a graph and a filter function, and the output is the predicted EPD. After obtaining the predicted EPD, we vectorize it with persistence image [1] and evaluate (1) the 2-Wasserstein (W_2) distance between the predicted diagram and the ground truth EPD; (2) the total square error between the predicted persistence image and the ground truth image (persistence image error, denoted as PIE). Considering that our aim is to estimate EPDs on graphs rather than roughly approximating persistence images, we use the W_2 distance as the training loss, while the PIE is only used as an evaluation metric. Given an input graph (e.g., Cora, Citeseer, etc.) and a filter function, we extract the k-hop neighborhoods of all the vertices and separate these vicinity graphs randomly into 80%/20% as training/test sets. We report the mean W_2 distance between diagrams and PIE on different vicinity graphs and 4 different filter functions.

Baseline settings. **PDGNN** denotes our proposed method, that is, the GNN framework with the proposed AGG function and MSG function. Its strategy is to first predict the EPD, and then convert it to the persistence image. To show its superiority, we compare with the strategy from [48, 37], i.e., directly approximate the persistence image of the input graph, as a baseline strategy. **GIN_PI** and **GAT_PI** denote the baseline strategy with GIN [57] and GAT [52] as the backbone GNNs.

To show the effectiveness of the modules proposed in Section 4, we add other baselines with our proposed strategy. **GAT** denotes GAT as the backbone GNN. **GAT (+MIN)** denotes GAT with the

[5]Following the settings in [67, 61], we adopt the Ollivier-Ricci curvature as the graph metric, and the distance to target node(s) as the filter function; Following the settings in [5], we set the temparature $t = 10$ and 0.1 and adopt these two kernel functions as the filter functions; Node degree is used as the initial filter function in [21].

new AGG function. Compared with PDGNN, it exploits the original node feature rather than the new edge feature in the MSG function. **PDGNN (w/o ew)** denotes PDGNN without edge weight. Further experimental settings can be found in the supplementary material.

Results. Table 1 reports the approximation error, we observe that PDGNN outperforms all the baseline methods among all the datasets. The comparison between GAT and GAT_PI shows the benefit of predicting EPDs instead of predicting the persistence image. Comparing GAT and GAT (+MIN), we observe the advantage of the new AGG function, which shows the necessity of using min aggregation to approximate the Find-Root algorithm; Comparing GAT (+MIN) and PDGNN, we observe the effectiveness of using the new MSG function to help the model capture information of the separated connected components. The comparison between PDGNN (w/o ew) and PDGNN shows that edge weights help the model focus on the individual Relax-Edge sub-algorithm operated on every edge.

5.2 Downstream Tasks

In this section, we evaluate the performance of the predicted diagrams on 2 graph representation learning tasks: node classification and link prediction. We replace the ground truth EPDs in state-of-the-art models based on persistence [67, 61] with our predicted diagrams and report the results.

Baselines. We compare our method with various state-of-the-art methods. We compare with popular GNN models including **GCN** [29], **GAT** [52] and **HGCN** [7]. For link prediction, we compare with several state-of-the-art methods such as **SEAL** [65] and **P-GNN** [62]. Notice that GCN and GAT are not originally designed for link prediction, therefore we follow the settings in [7, 61], that is, to get the node embedding through these models, and use the Fermi-Dirac decoder [30, 42] to predict whether there is a link between the two target nodes. In comparison with the original EPD, we also add **PEGN** [67] and **TLC-GNN** [61] as baseline methods. Furthermore, to show the benefit of directly predicting EPDs, we also add the baseline methods **PEGN (GIN_PI)** and **TLC-GNN (GIN_PI)**, which replace the original persistent homology feature with the output from GIN_PI.

Evaluation metrics. For node classification, our setting is the same as [29, 52, 67]. To be specific, we train the GNNs with 20 nodes from each class and validate (resp. test) the GNN on 500 (resp. 1000) nodes. We run the GNNs on these datasets 10 times and report the average classification accuracy and standard deviation. For link prediction, our setting is the same as [7, 61]. To be precise, we randomly split existing edges into 85/5/10% for training, validation, and test sets. An equal number of non-existent edges are sampled as negative samples in the training process. We fix the negative validation and test sets, and randomly select the negative training sets in every epoch. We run the GNNs on these datasets 10 times and report the mean average area under the ROC curve (ROCAUC) scores and the standard deviation.

Results. Table 2 and Table 3 summarize the performance of all methods on node classification and link prediction. We observe that PEGN (PDGNN) and TLC-GNN (PDGNN) consistently perform comparably with PEGN and TLC-GNN, showing that the EPDs approximated by PDGNN have the same learning power as the true EPDs. Furthermore, PEGN using the approximated EPDs achieve better or comparable performance with different SOTA methods.

We also discover that PEGN (GIN_PI) and TLC-GNN (GIN_PI) perform much inferior to the original models using the true EPDs. It demonstrates that the large approximation error from GIN_PI lose much of the crucial information which is preserved in PDGNN.

Transferability. One appealing feature of our method is its transferability. Training on one graph, our algorithm can estimate EPDs well on another graph. This makes it possible to apply the computationally expensive topological features to a wide spectrum of real-world graphs; we can potentially apply a pre-trained model to large and dense graphs, on which direct EPD computation is infeasible. The experiments are provided in the supplementary material.

5.3 Algorithm Efficiency

In this section, we evaluate the efficiency of our proposed model. For a fair and complete comparison, we compare with algorithms from Gudhi [50] and from [61]. We select the first 1000 nodes from Cora, Citeseer, PubMed, Photo, Computers, CS, Physics, and then extract their 2-hop neighborhoods. With Ollivier-Ricci curvature as the filter function, we compute the EPDs and report the time (seconds) used to infer these diagrams.

Results. We list the average nodes and edges of these vicinity graphs in the first line of Table 4. As shown in Table 4, although our model is slower on small datasets like Cora or Citeseer, it is much faster on large and dense datasets. Therefore we can simply use the original algorithm to compute the EPDs on small graphs, and use our model to estimate EPDs on large graphs. The model can be applied to various graph representation learning works based on persistent homology.

6 Conclusion

Inspired by recent success on neural algorithm execution, we propose a novel GNN with different technical contributions to simulate the computation of EPDs on graphs. The network is built on algorithmic insights, and benefits from better supervision and closer alignment with the EPD computation algorithm. Experiments show that our method achieves satisfying approximation quality and learning power while being significantly faster than the original algorithm on large and dense graphs. Another strength of our method is the transferability: training on one graph, our algorithm can still approximate EPDs well on another graph. This makes it possible to apply the computationally expensive topological features to a wide spectrum of real-world graphs.

Acknowledgements. We thank all anonymous reviewers for their constructive feedback very much. This work of Zuoyu Yan, Liangcai Gao, and Zhi Tang is supported by the projects of National Key R&D Program of China (2019YFB1406303) and National Natural Science Foundation of China (No. 61876003), which is also a research achievement of Key Laboratory of Science, Technology and Standard in Press Industry (Key Laboratory of Intelligent Press Media Technology).

References

[1] 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, 2017.

[2] Pankaj K Agarwal, Herbert Edelsbrunner, John Harer, and Yusu Wang. Extreme elevation on a 2-manifold. *Discrete & Computational Geometry*, 36(4):553–572, 2006.

[3] Cristian Bodnar, Fabrizio Frasca, Yu Guang Wang, Nina Otter, Guido Montufar, Pietro Liò, and Michael M Bronstein. Weisfeiler and lehman go topological: Message passing simplicial networks. In *ICML*, 2021.

[4] Gunnar Carlsson and Vin De Silva. Zigzag persistence. *Foundations of computational mathematics*, 10(4):367–405, 2010.

[5] 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. In *International Conference on Artificial Intelligence and Statistics*, pages 2786–2796. PMLR, 2020.

[6] Mathieu Carriere, Marco Cuturi, and Steve Oudot. Sliced wasserstein kernel for persistence diagrams. In *International conference on machine learning*, pages 664–673. PMLR, 2017.

[7] Ines Chami, Zhitao Ying, Christopher Ré, and Jure Leskovec. Hyperbolic graph convolutional neural networks. *Advances in neural information processing systems*, 32:4868–4879, 2019.

[8] Chao Chen, Xiuyan Ni, Qinxun Bai, and Yusu Wang. A topological regularizer for classifiers via persistent homology. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 2573–2582. PMLR, 2019.

[9] Yuzhou Chen, Baris Coskunuzer, and Yulia Gel. Topological relational learning on graphs. *Advances in Neural Information Processing Systems*, 34, 2021.

[10] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. Stability of persistence diagrams. *Discrete & computational geometry*, 37(1):103–120, 2007.

[11] David Cohen-Steiner, Herbert Edelsbrunner, and John Harer. Extending persistence using poincaré and lefschetz duality. *Foundations of Computational Mathematics*, 9(1):79–103, 2009.

[12] David Cohen-Steiner, Herbert Edelsbrunner, John Harer, and Yuriy Mileyko. Lipschitz functions have l p-stable persistence. *Foundations of computational mathematics*, 10(2):127–139, 2010.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[14] Matija Čufar and Žiga Virk. Fast computation of persistent homology representatives with involuted persistent homology. *arXiv preprint arXiv:2105.03629*, 2021.

[15] Thibault de Surrel, Felix Hensel, Mathieu Carrière, Théo Lacombe, Yuichi Ike, Hiroaki Kurihara, Marc Glisse, and Frederic Chazal. Ripsnet: a general architecture for fast and robust estimation of the persistent homology of point clouds. In *ICLR 2022 Workshop on Geometrical and Topological Representation Learning*, 2022.

[16] Tamal K. Dey and Yusu Wang. *Computational Topology for Data Analysis*. Cambridge University Press, 2022.

[17] Vijay Prakash Dwivedi, Chaitanya K Joshi, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

[18] Herbert Edelsbrunner and John Harer. *Computational topology: an introduction*. American Mathematical Soc., 2010.

[19] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *Proceedings 41st annual symposium on foundations of computer science*, pages 454–463. IEEE, 2000.

[20] Loukas Georgiadis, Haim Kaplan, Nira Shafrir, Robert E Tarjan, and Renato F Werneck. Data structures for mergeable trees. *ACM Transactions on Algorithms (TALG)*, 7(2):1–30, 2011.

[21] Christoph Hofer, Florian Graf, Bastian Rieck, Marc Niethammer, and Roland Kwitt. Graph filtration learning. In *International Conference on Machine Learning*, pages 4314–4323. PMLR, 2020.

[22] Christoph Hofer, Roland Kwitt, Marc Niethammer, and Mandar Dixit. Connectivity-optimized representation learning via persistent homology. In *International Conference on Machine Learning*, pages 2751–2760. PMLR, 2019.

[23] Christoph Hofer, Roland Kwitt, Marc Niethammer, and Andreas Uhl. Deep learning with topological signatures. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pages 1633–1643, 2017.

[24] Paul W Holland, Kathryn Blackmond Laskey, and Samuel Leinhardt. Stochastic blockmodels: First steps. *Social networks*, 5(2):109–137, 1983.

[25] Nan Hu, Raif M Rustamov, and Leonidas Guibas. Stable and informative spectral signatures for graph matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2305–2312, 2014.

[26] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

[27] Xiaoling Hu, Fuxin Li, Dimitris Samaras, and Chao Chen. Topology-preserving deep image segmentation. *Advances in Neural Information Processing Systems*, 32:5657–5668, 2019.

[28] Lukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[29] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[30] Dmitri Krioukov, Fragkiskos Papadopoulos, Maksim Kitsak, Amin Vahdat, and Marián Boguná. Hyperbolic geometry of complex networks. *Physical Review E*, 82(3):036106, 2010.

[31] Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[32] Genki Kusano, Yasuaki Hiraoka, and Kenji Fukumizu. Persistence weighted gaussian kernel for topological data analysis. In *International Conference on Machine Learning*, pages 2004–2013. PMLR, 2016.

[33] Panagiotis Kyriakis and Iordanis Fostiropoulos. Learning hyperbolic representations of topological features. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.

[34] Pan Li, Yanbang Wang, Hongwei Wang, and Jure Leskovec. Distance encoding: Design provably more powerful neural networks for graph representation learning. *Neural Information Processing Systems (NeurIPS)*, 2020.

[35] Yao Ma and Jiliang Tang. *Deep learning on graphs*. Cambridge University Press, 2021.

[36] Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 2156–2167, 2019.

[37] Guido Montufar, Nina Otter, and Yu Guang Wang. Can neural networks learn persistent homology features? In *NeurIPS 2020 Workshop on Topological Data Analysis and Beyond*, 2020.

[38] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond (GRL+ 2020)*, 2020.

[39] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[40] James R Munkres. *Elements of algebraic topology*. CRC press, 2018.

[41] Chien-Chun Ni, Yu-Yao Lin, Jie Gao, and Xianfeng Gu. Network alignment by discrete ollivier-ricci flow. In *International Symposium on Graph Drawing and Network Visualization*, pages 447–462. Springer, 2018.

[42] Maximillian Nickel and Douwe Kiela. Poincaré embeddings for learning hierarchical representations. *Advances in neural information processing systems*, 30:6338–6347, 2017.

[43] Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[44] Jan Reininghaus, Stefan Huber, Ulrich Bauer, and Roland Kwitt. A stable multi-scale kernel for topological machine learning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4741–4748, 2015.

[45] Adam Santoro, Ryan Faulkner, David Raposo, Jack Rae, Mike Chrzanowski, Theophane Weber, Daan Wierstra, Oriol Vinyals, Razvan Pascanu, and Timothy Lillicrap. Relational recurrent neural networks. *Advances in Neural Information Processing Systems*, 31:7299–7310, 2018.

[46] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

[47] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *arXiv preprint arXiv:1811.05868*, 2018.

[48] Anirudh Som, Hongjun Choi, Karthikeyan Natesan Ramamurthy, Matthew P Buman, and Pavan Turaga. Pi-net: A deep learning approach to extract topological persistence images. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pages 834–835, 2020.

[49] Jian Sun, Maks Ovsjanikov, and Leonidas Guibas. A concise and provably informative multi-scale signature based on heat diffusion. In *Computer graphics forum*, volume 28, pages 1383–1392. Wiley Online Library, 2009.

[50] The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2015.

[51] Petar Veličković, Lars Buesing, Matthew Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *Advances in Neural Information Processing Systems*, 33:2232–2244, 2020.

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

[53] Petar Veličković, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2019.

[54] Fan Wang, Huidong Liu, Dimitris Samaras, and Chao Chen. Topogan: A topology-aware generative adversarial network. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*, pages 118–136. Springer, 2020.

[55] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[56] Louis-Pascal Xhonneux, Andreea-Ioana Deac, Petar Veličković, and Jian Tang. How to transfer algorithmic reasoning knowledge to learn new algorithms? *Advances in Neural Information Processing Systems*, 34, 2021.

[57] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[58] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S Du, Ken-ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2020.

[59] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines. *Advances in Neural Information Processing Systems*, 33, 2020.

[60] Zuoyu Yan, Tengfei Ma, and Chao Chen. Cycle representation learning for inductive relation prediction. In *International Conference on Machine Learning*, 2022.

[61] Zuoyu Yan, Tengfei Ma, Liangcai Gao, Zhi Tang, and Chao Chen. Link prediction with persistent homology: An interactive view. In *International Conference on Machine Learning*, pages 11659–11669. PMLR, 2021.

[62] Jiaxuan You, Rex Ying, and Jure Leskovec. Position-aware graph neural networks. In *International Conference on Machine Learning*, pages 7134–7143. PMLR, 2019.

[63] Wojciech Zaremba and Ilya Sutskever. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

[64] Sebastian Zeng, Florian Graf, Christoph Hofer, and Roland Kwitt. Topological attention for time series forecasting. *Advances in Neural Information Processing Systems*, 34, 2021.

[65] Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in Neural Information Processing Systems*, 31:5165–5175, 2018.

[66] Qi Zhao and Yusu Wang. Learning metrics for persistence-based summaries and applications for graph classification. *Advances in Neural Information Processing Systems*, 32:9859–9870, 2019.

[67] Qi Zhao, Ze Ye, Chao Chen, and Yusu Wang. Persistence enhanced graph neural network. In *International Conference on Artificial Intelligence and Statistics*, pages 2896–2906. PMLR, 2020.

[68] Songzhu Zheng, Yikai Zhang, Hubert Wagner, Mayank Goswami, and Chao Chen. Topological detection of trojaned neural networks. *Advances in Neural Information Processing Systems*, 34, 2021.

[69] Chi Zhou, Zhetong Dong, and Hongwei Lin. Learning persistent homology of 3d point clouds. *Computers & Graphics*, 102:269–279, 2022.

Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope? [Yes]
   (b) Did you describe the limitations of your work? [Yes] It is discussed in the supplementary material.
   (c) Did you discuss any potential negative societal impacts of your work? [No] The work does not seem to have any negative societal impact.
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]
2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]
3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [No] The data and the experimental details are provided in Section 5 and the supplementary material. We will release the code once accepted.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] It is illustrated in Section 5 in the paper and in the supplementary material.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] Table 2 and Table 3 are examples.
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] It is provided in the supplementary material.
4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes]
   (b) Did you mention the license of the assets? [Yes] It is mentioned in the supplementary material.
   (c) Did you include any new assets either in the supplemental material or as a URL? [No] We do not release new assets.
   (d) Did you discuss whether and how consent was obtained from people whose data you're using/curating? [N/A]
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [N/A]
5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [N/A]

(b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [N/A]

(c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [N/A]

A Supplementary material

In the supplementary material, we provide (1) the related work; (2) the complexity and the correctness of the introduced algorithm; (3) the Union-Find algorithm in a sequential format; (4) additional experimental details, including the introduction of the datasets, and the experimental settings; (5) further experiments, including the evaluation on transferability, the influence of training samples, experiments on other datasets, experiments on other attributes of the model and the limitation of the paper.

A.1 Related Works

Learning with Persistent Homology. Based on the theory of algebraic topology [40], persistent homology [19, 18] extends the classical notion of homology, and can capture the topological structures (e.g., loops, connected components) of the input data in a robust [10] manner. It has already been used in various deep learning domains including kernel machines [44, 32, 6], convolutional neural networks [23, 27, 54, 68], transformers [64], connectivity loss [8, 22], and graph representation learning [67, 9, 61, 66, 21, 5, 33]. Some following works propose persistence-inspired frameworks on other tasks such as knowledge graph completion [60].

Neural Algorithm Execution. Many works have studied neural execution in different domains before [63, 28, 31, 43, 45, 59]. With the rapid development of GNNs in graph representation learning, learning graph algorithms with GNNs has attracted researchers' attention [53, 51, 56]. These works exploit GNNs to approximate certain classes of graph algorithms, such as parallel algorithms (e.g., Breadth-First-Search) and sequential algorithms (e.g., Dijkstra). Although the computation of extended persistence diagrams can be written in a sequential-like form, it needs extra steps and considerations. In our framework, we propose different modules to approximate these steps and achieve satisfying practical performance.

Accelerating Extended Persistent Homology. In general, computing extended persistent homology relies on the well-known matrix reduction algorithm [11]. Much effort has been made to accelerate the computation, but it still takes matrix multiplication time [50, 16, 14]. For the specific case where the input is a function on a graph $G = (V, E)$, it turns out that one can compute it in $O(|E| \log |V|)$ time [2, 20]. Nevertheless, this algorithm remains theoretical, and in practice, often a quadratic $O(|V||E|)$ time algorithm is used for its simplicity [61]. Recently, some works have been proposed to accelerate the computation in a data-driven manner [48, 37, 69, 15]. However, these works try to estimate the persistence image [1], a coarsened topological feature rather than the persistence diagram itself, leading to much worse performance in both approximation error and downstream tasks. Compared with previous works, we propose a novel framework that directly predicts extended persistence diagrams on graphs. As shown in the experiment, the proposed model has achieved a satisfying approximation error while remaining a high efficiency as well.

A.2 Complexity and Correctness of Algorithm 2

In this section, we show the complexity and the correctness of Algorithm 2 (which is restated as Algorithm 5 in the supplementary material.).

A.2.1 Complexity

The computational complexity of the Union-Find algorithm is $O(|E|\alpha(|E|))$ [13], where $\alpha(\cdot)$ is the inverse Ackermann function. Therefore, we need $O(|V||E|\alpha(|E|))$ time to compute an EPD using Algorithm 5. Note this sequential algorithm is not necessarily the most efficient one. In practice, one may use the quadratic algorithm ($O(|V||E|)$) as in [61]. We also note that although not formally published, the best known algorithm for EPD computation is quasilinear, $O(|E| \log |V|)$, using the data structure of mergeable trees [2, 20]. But this algorithm remains theoretical so far.

Algorithm 4 Sequential algorithm

Input: graph $G = (V, E)$, filter function f.
Initialise-Nodes(V, f)
$Q =$ Sort-Queue(V)
while Q is not empty **do**
  $u = Q$.pop-min()
  for $v \in G$.neighbors(u) **do**
    Relax-Edge(u, v, f)
  end for
end while

Algorithm 5 Computation of 1D EPD

Input: filter function f, input graph $G = (V, E)$
$V, E =$ sorted(V, E, f)
$PD_0 =$ Union-Find(V, E, f), $PD_1 = \{\}$
for $i \in V$ **do**
  $C_i = \{C_{i_j} | (i, j) \in E, f(j) > f(i)\}$, $E_i = E$
  for $C_{i_j} \in C_i$ **do**
    $f(C_{i_j}) = f(i)$, $E_i = E_i - \{(i, j)\} + \{(C_{i_j}, j)\}$
  end for
  $PD_1^i =$ Union-Find-step($V + C_i - \{i\}, E_i, f, C_i$)
  $PD_1 += PD_1^i$
end for
Output: PD_0, PD_1

Algorithm 6 Union-Find-step (Sequential)

Input: V, E, f, C_i
$PD_1^i = \{\}$
for $v \in V$ **do**
  $v.value = f(v)$, $v.root = v$
end for
$Q =$ Sort(V), $Q = Q - \{v | f(v) < f(i)\}$
while Q is not empty **do**
  $u = Q$.pop-min()
  for $v \in G$.neighbors(u) **do**
    $pu, pv =$ Find-Root(u), Find-Root(v)
    if $pu \neq pv$ **then**
      $s = argmin(pu.value, pv.value)$
      $l = argmax(pu.value, pv.value)$
      $l.root = s$
      if $pu \in V_i$ and $pv \in V_i$ **then**
        $PD_1^i + \{(u.value, l.value)\}$
      end if
    end if
  end for
end while
Function: Find-Root(u)
$pu = u$
while $pu \neq pu.root$ **do**
  $pu.root = (pu.root).root$, $pu = pu.root$
end while
Return: pu

A.2.2 Correctness

Formally, we restate the theorem below (The theorem is named Theorem 3.1 in the main paper). For a clear statement, we present the standard EPD computation algorithm in Algorithm 7. The detailed description of Algorithm 7 is beyond the scope of the paper. We only introduce the needed information, and refer the readers to [11, 18] for details.

Theorem A.1. *Algorithm 5 outputs the same 1D EPDs as Algorithm 7.*

As stated in Section 2 and Section 3 in the paper, for an edge (1-simplex) $e \in E$, it is either paired with a vertex or an edge. In the former case, the edge, defined as a negative edge, kills a connected component, and gives rise to a 0D persistence point. In the latter case, the edge, defined as a positive edge (in the ascending filtration), creates a loop during the ascending filtration. The loop will ultimately be killed by another edge during the descending filtration (defined as a positive edge in the descending filtration). Hence the positive edge in the ascending filtration is paired with a positive edge in the descending filtration, and gives rise to a 1D extended persistence point. For simplicity, we will call the positive edges in the ascending filtration as *ascending positive edges*, and the positive edges in the descending filtration as *descending positive edges*.

In other words, to compute the 1D EPDs, we can simply find the pairing partner for all positive edges. In the following paragraphs, we show that Algorithm 5 produces the same extended persistence pair as the standard EPD computation algorithm. We first present a definition of the "thinnest pair":

Thinnest pair. Given a filter function $f : X \to \mathcal{R}$, the pair of edges (e_1, e_2) with $f(e_1) < f(e_2)$ is defined as the thinnest pair if the following condition is satisfied: (1) there is a cycle C having e_1 as the lowest edge, and e_2 as the highest edge; (2) for any other cycle with e_1 as the lowest edge, if its highest edge e_2 satisfies that $f(e_3) \neq f(e_2)$, then $f(e_3) > f(e_2)$. Symmetrically, among all

cycles having e_2 as the highest edge, e_1 is the lowest edge in a cycle such that this lowest value is the highest possible.

Lemma A.2. *For every ascending positive edge, Algorithm 5 finds its "thinnest pair".*

Proof. Algorithm 5 decomposes the 1D extended persistence pair finding for all edges into pair-finding among all nodes. In particular, for a given node u, it uses Algorithm 6 to find the pair for its upper edges. There are two cases:

Case 1. If the upper edge is an ascending negative edge, then it will kill a connected component, and will not influence the 1D extended persistence pairing.

Case 2. If the upper edge is an ascending positive edge, it will be paired with the loop once the loop is created in the union-find process. The edge, called e, is the lowest edge in the loop, called C. Recall that C is also the first loop that appears in the union-find process with e as the lowest edge. Therefore C is guaranteed to contain the highest value which is the lowest possible. According to the definition, this will lead to the "thinnest pair"[6]. □

Algorithm 7 The standard EPD computation algorithm

Input: filter funtion f, input graph G
$EPD = \{\}$
M = build reduction matrix(f, G), where M is a $2m * 2m$ binary matrix.
for $j = 1$ **to** $2m$ **do**
  while $\exists k < j$ with $low_M(k) = low_M(j)$ **do**
    add column k to column j
  end while
  add $(f(low_M(j)), f(j))$ to EPD
end for
Output: EPD

Lemma A.3. *In the descending filtration of Algorithm 7, an edge e is paired if a loop C has already appeared, with e as its lowest edge.*

Proof. Every column/row of the binary matrix M shown in Algorithm 7 corresponds to a simplex (node/edge) in the input graph G. For simplicity, we replace the index in M with the simplex it represents in the rest of the paper. For an edge e, $low_M(e)$ denotes its lowest row e_1, with $M[e, e_1] = 1$. After the matrix reduction process, e and e_1 will form an extended persistence pair. It has been shown in [11, 61] that for a loop, its highest edge and lowest edge form its extended persistence pair. In other words, e and e_1 are the lowest and highest edges of the loop they form. Assume that a loop C has already aroused with e as its lowest edge, then there are two cases for the highest edge e_1 in C:

Case 1. If there does not exist an edge e_2, that appears before e_1 in the descending filtration, with $low_M(e_2) = e = low_M(e_1)$, then e will be paired with e_1 in Algorithm 7.

Case 2. If there exists an edge e_2, that appears before e_1 in the descending filtration, with $low_M(e_2) = e = low_M(e_1)$, then e will be paired with e_2 or even other edges that appears earlier than e_2. Among all possibilities, e is paired before C appears in Algorithm 7.

In other words, e will be paired with e_1 or before e_1 in Algorithm 7.

□

Lemma A.4. *Algorithm 7 finds the "thinnest pair" for all positive edges.*

Proof. During the descending filtration, when an edge $e = u_i u_j$ appears, there are two cases:

Case 1. e is a negative edge, and will kill a connected component. This will not influence the 1D extended persistence pair.

[6] We note that once a loop appears in the union-find process, it will consist of two different upper edges. Considering that the two upper edges share the same filter value in the ascending filtration, the output persistence point will not change no matter which edge is paired with the loop.

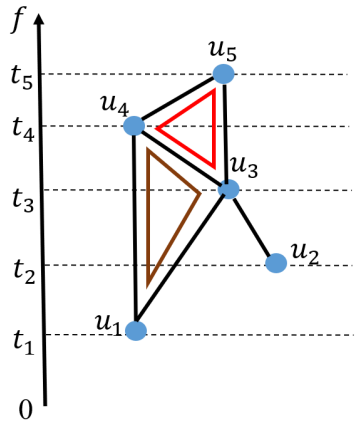


Figure 3: A toy example for Lemma A.4.

Case 2. e is a positive edge, and will be paired with an ascending positive edge $e_1 = u_a u_b$. Assume that e_1 does not construct the "thinnest pair" with e, then there exists an edge $e_2 = u_c u_d$, that forms the "thinnest pair" with e. We can observe a loop $C = u_a \to u_c u_d \to u_b$, in which all edges born earlier than e in the descending filtration, and e_1 is the latest edge in the ascending filtration. A toy example is shown in Figure 3, where $e = u_1 u_3$, $e_1 = u_4 u_5$, $e_2 = u_3 u_4$, and C is the red loop. Then according to Lemma A.3, e_1 will be paired no later than C appears. In other words, it has already been paired before e appears. Therefore, the assumption is wrong, and Algorithm 7 will find the "thinnest pair" for all positive edges.

□

According to Lemma A.2 and Lemma A.4, Algorithm 5 will produce the same 1D extended persistence pair as Algorithm 7. Therefore, they output the same 1D EPD.

A.3 Union-Find Algorithm

In this section, we rewrite the well-known Union-Find algorithm [13] in a sequential format. The algorithm is listed in Algorithm 8. Therefore we can use the proposed framework to estimate the 0D EPDs.

Algorithm 8 Union-Find (Sequential)

Input: V, E, f
$PD_0 = \{\}$
for $v \in V$ **do**
  $v.value = f(v)$, $v.root = v$
end for
$Q = \text{Sort}(V)$
while Q is not empty **do**
  $u = Q.\text{pop-min}()$
  for $v \in G.\text{neighbors}(u)$ **do**
    $pu, pv = \text{Find-Root}(u), \text{Find-Root}(v)$
    if $pu \neq pv$ **then**
      $s/l = argmin/argmax(pu.value, pv.value)$
      $l.root = s$
      $PD_0 + \{(l.value, u.value)\}$
    end if
  end for
end while
Function: $Find - Root(u)$
$pu = u$
while $pu \neq pu.root$ **do**
  $pu.root = (pu.root).root$, $pu = pu.root$
end while
Return: pu

A.4 Experimental details

A.4.1 Datasets.

In this paper, we exploit real-world datasets including:

Table 5: Statistics of the node classification datasets

Dataset	Classes	Nodes	Edges	Features	Avg degree
Cora	7	2708	5429	1433	2.00
Citeseer	6	3327	4732	3703	1.42
PubMed	3	19717	44338	500	2.25
CS	15	18333	100227	6805	5.47
Physics	5	34493	282455	8415	8.19
Computers	10	13381	259159	767	19.37
Photo	8	7487	126530	745	16.90

1. Citation networks: Cora, Citeseer, and PubMed [46] are standard citation networks where nodes denote scientific documents and edges denote citation links.
2. Amazon shopping records: In Photo and Computers [47], nodes represent goods, edges represent that two goods are frequently brought together, and the node features are bag-of-words vectors.
3. Coauthor datasets: In CS and Physics [47], nodes denote authors and edges denote that the two authors co-author a paper.

The detailed statistics are available in Table 5.

A.4.2 Experimental Details

In this section, we mainly present the experimental settings on neural estimation, as for the setting in downstream graph representation learning tasks, we are consistent with [67, 61].

Following the settings in [67, 61], we extract 2-hop neighborhoods of all the nodes in Cora, Citeseer, PubMed and 1-hop neighborhoods of all the nodes in Photo, Computers, Physics, and CS. In the training process, we only adopt the W_2 distance between the predicted diagram and the ground truth diagram as the loss function, while the PIE between the predicted persistence image and the ground truth persistence image only serves as an evaluation metric.

We adopt Adam as the optimizer with the learning rate set to 0.002 and weight decay set to 0.01. We build a 4-layer GNN framework with dropout set to 0. In the training process, we set the batch size to 10, and the training epoch to 20. In this paper, we also exploit a 2-layer MLP to transform the node embedding obtained by the GNN to the persistence points on edges. In the framework, PRELU is adopted as the activation function, the dimension of hidden layers is set to 32, and the dimension of the output persistence image is 25. All the experiments are implemented with two Intel Xeon Gold 5128 processors,192GB RAM, and 10 NVIDIA 2080TI graphics cards.

Notice that in the normal computation of Wasserstein distance between PDs, the persistence points can be paired to the diagonal or the persistence points in the other diagram. However, in the experiments, we observe that with this loss function as the supervision, the model may converge to local minima, e.g., all the predicted persistence points are paired to diagonal. Therefore, the predicted points all converge to the diagonal and contain no topological information. To avoid such situations, we force the predicted points to pair with the persistence points in the ground truth diagram rather than the diagonal in the training stage. In the reference stage, we report the normal W_2 distance between persistence diagrams, that is, to let the predicted points pair with the diagonal.

A.4.3 About the assets we used

Our model is experimented on benchmarks from [38, 26, 17, 46, 47] provided under MIT license.

A.5 Additional Experiments

A.5.1 Experiments on transferability

In this section, we design experiments to evaluate the transferability of PDGNN in terms of different graph structures. Our aim is to evaluate whether the pre-trained model can estimate EPDs on totally unseen graphs. Therefore, we evaluate the models pre-trained on Photo on other datasets, and report

Table 6: Transferability in terms of different graph structures (W_2 distance.)

Pre-train	Cora	Citeseer	PubMed	Photo	Computers
Pre-train	0.392	0.279	0.444	0.379	0.404
Fine-tune	0.348	0.259	0.360	0.380	0.381
Standard	0.354	0.267	0.344	0.379	0.377

Table 7: Influence of training samples on PDGNN

Dataset	Cora			Citeseer			PubMed		
Proportion	W_2	PIE	NCA	W_2	PIE	NCA	W_2	PIE	NCA
5%	0.391	2.51e-3	81.3±0.6	0.273	3.12e-3	70.0±0.7	0.330	4.35e-3	78.0±0.4
10%	0.358	1.88e-3	81.6±0.7	0.231	3.01e-3	70.5±0.5	0.300	2.36e-3	78.5±0.4
20%	0.318	6.99e-4	81.8±0.8	0.227	1.63e-3	70.6±0.5	0.278	1.03e-3	78.3±0.3
40%	0.286	9.79e-4	81.6±0.6	0.208	9.98e-4	70.9±0.6	0.255	1.34e-3	78.8±0.5
80%	0.241	4.75e-4	82.0±0.5	0.183	4.43e-4	70.8±0.5	0.256	8.95e-4	78.7±0.6

the W_2 distance between the predicted diagrams and ground truth EPDs. Notice that we only use Ollivier-Ricci curvature [41] as the filter function. The results are shown in Table 6.

In Table 6, “Pre-train" is to directly predict the EPDs with the pre-trained model, and “Fine-tune" is to fine-tune an epoch on the new datasets, and then predict the EPDs. As shown in Table 6, directly predicting the EPDs with the pre-trained model perform comparably with the standard settings among datasets. We also observe that with only a one-epoch fine-tuning, the pre-trained model can achieve almost an equal performance compared with the standard setting. It justifies the fine transferability of PDGNN. Therefore, in a totally new environment, instead of training the uninitialized models for many epochs, we can simply fine-tune or even directly use the pre-trained model to estimate EPDs on new graph structures.

A.5.2 Evaluation on the influence of training samples

In this section, we evaluate the influence of training samples on PDGNN. We aim to show that the model can reach an acceptable performance with only a small number of training samples.

Recall that for a given graph, we extract the k-hop neighborhoods of all the nodes and randomly select 80% of these vicinity graphs to train PDGNN. For a thorough evaluation, we train PDGNN with 5/10/20/40% vicinity graphs in this experiment and report the W_2 distance of persistence diagrams, the PIE of persistence images, and the node classification accuracy (NCA) in Table 7. We also visualize the influence in Figure 4 and Figure 5.

As shown in Figure 4, the training error tends to converge as the training samples gradually increase. Considering that the W_2 distance and PIE cannot directly reflect the learning power as NCA does, we select a vicinity graph in Cora which is hard for PDGNN to learn and visualize in Figure 5. As shown in the figure, as the number of training samples increases, we find that PDGNN can gradually capture the ground truth persistence points in the up y-axis and the up-right diagonal with much less noise. The number of training samples may help the model learn the hard samples better.

We also observe that in Table 7, PDGNN reaches a comparable performance on NCA with much fewer training samples. The observation shows that a little perturbation on the persistence image will not influence its structural information very much.

Combining the observation in Section A.5.1 and Section A.5.2, we can safely conclude that our model can be easily generalized to other frameworks. PDGNN does not need many training samples to reach an acceptable performance, and it can be easily transferred to totally unseen graphs.

A.5.3 Experiments on graph classification datasets.

In the experiment part, we only consider predicting EPDs of the k-hop neighborhoods of the original graphs. Even if these vicinity graphs can be large and dense, there can be structural differences between these vicinity graphs and other real-world graphs. In this section, we do further experiments on graph classification datasets, in which we approximate the EPDs of the

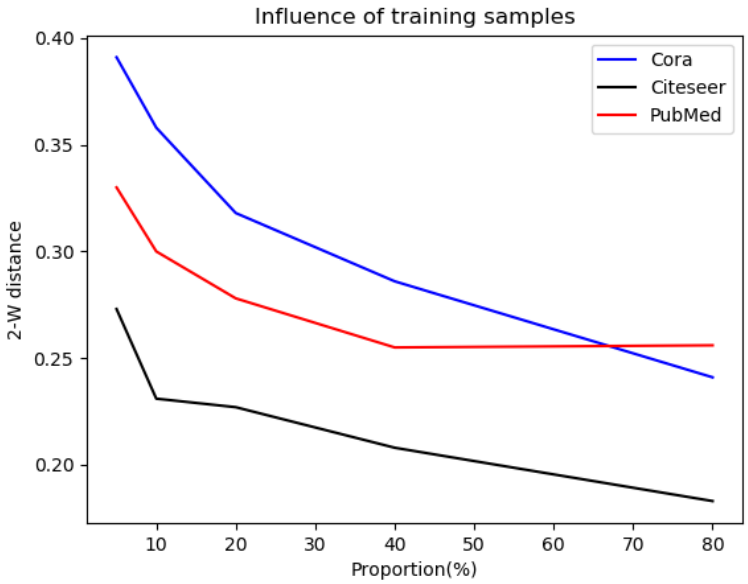
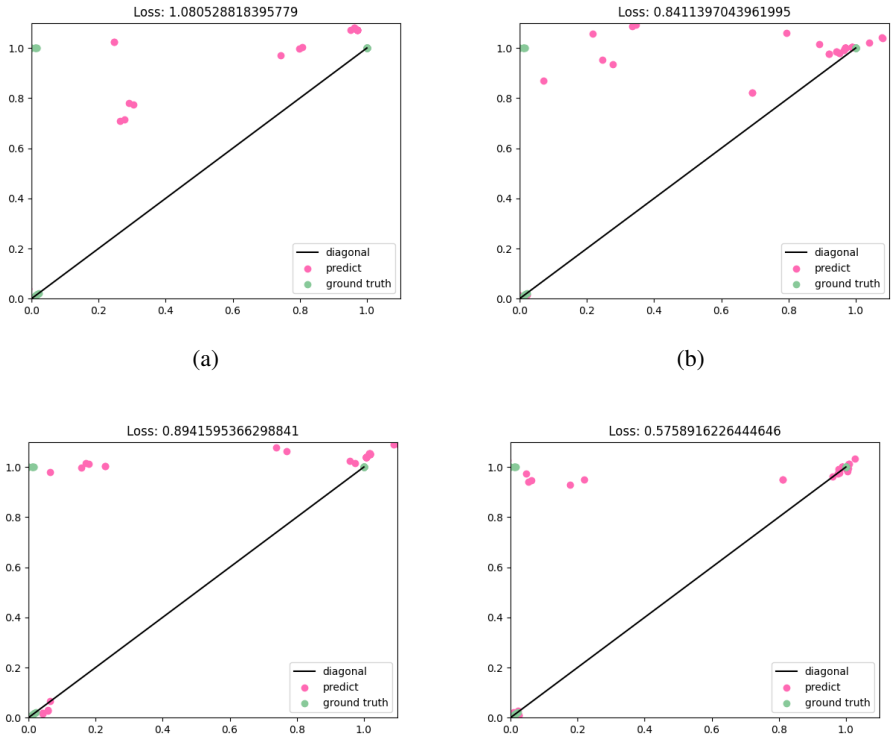


Figure 4: Influence of training samples.



(a) (b) (c) (d)

Figure 5: Visualization on the influence of training samples. We select a vicinity graph in Cora with Ollivier-Ricci curvature as the filter function, and plot the influence of training samples on the W_2 distance (loss) of EPDs. (a), (b), and (c) denote the prediction of PDGNN with 5/10/20% training samples, (d) denotes the prediction of PDGNN with the standard setting.

Table 8: Statistics and approximation error on the graph classification datasets

Dataset	Graphs	Avg Nodes	Avg Edges	W_2	PIE
MUTAG	188	17.9	39.6	0.300	3.06e-4
ENZYMES	600	32.6	124.3	0.299	3.72e-3
PROTEINS	1113	39.1	145.6	0.194	8.30e-4
COLLAB	5000	74.5	4914.4	0.346	3.25e-2
IMDB-BINARY	1000	19.8	193.1	0.176	4.13e-4
REDDIT-BINARY	2000	429.6	995.5	0.383	1.92e-4
ZINC (subset)	12000	23.2	49.8	0.089	1.52e-5
OGBG-MolHIV	41127	25.5	27.5	0.104	4.96e-5

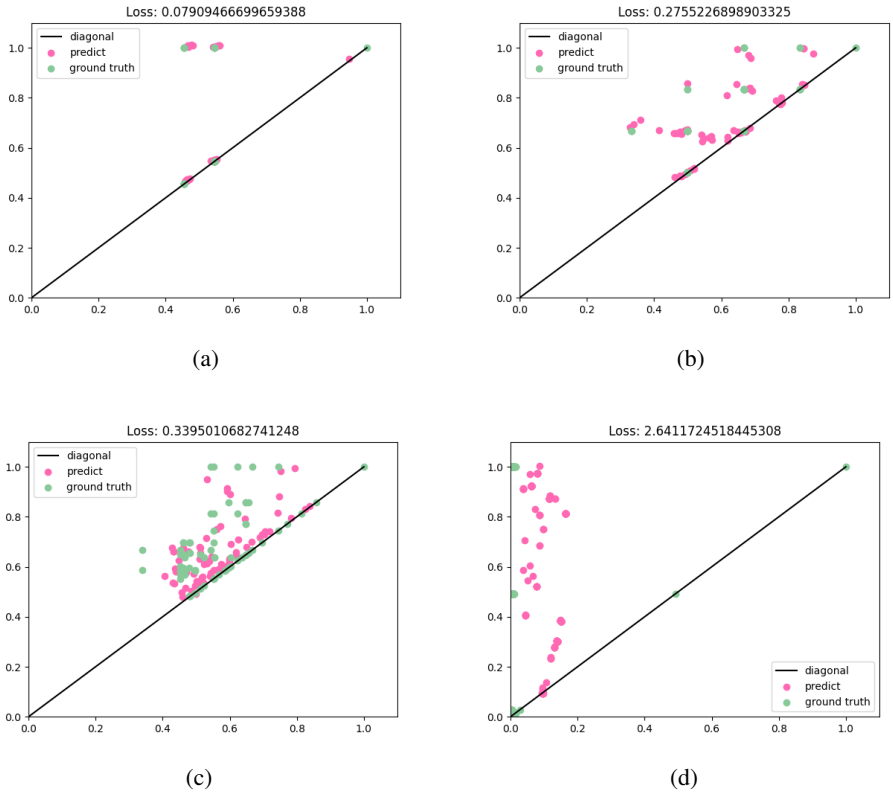
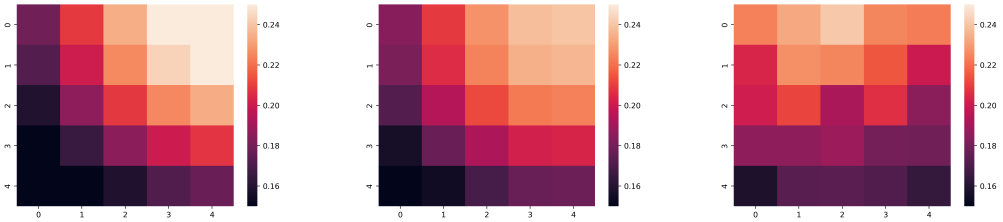


Figure 6: Visualization of graph classification samples. We select samples from IMDB-BINARY, PROTEINS, ENZYMES, and REDDIT-BINARY, respectively, and report the W_2 distance (loss).

real-world graphs rather than the vicinity graphs. We exploit various datasets from the TU Dortmund University [38], benchmarking-GNN [17], and OGB [26]. The detailed information of these datasets and the approximation error are all available in Table 8.

Notice that we do not add Ollivier-Ricci curvature as the filter function here, because computing the filter function on all the graphs will bring too much computational cost. Comparing the results from Table 8 and the results on vicinity graphs, we observe that the performance on graph classification datasets is slightly worse than the performance on vicinity graphs. This may be due to the fact that in graph classification datasets, the training samples can be very small, e.g., there are only 188 graphs in MUTAG, therefore the training is under-fit. On the contrary, the satisfying approximation quality on OGBG-MolHIV and ZINC can be due to their large number of training samples.

To evaluate the results more clearly, we also visualize some selected examples in Figure 6. As shown in the figure, in most situations, PDGNN can well estimate the EPDs on these graphs, and the W_2 distance around 0.3 is generally an acceptable result.



(a) (b) (c)

Figure 7: Examples to explain why not directly approximating PIs.

Table 9: Experiments on large and sparse datasets.

Dataset	Cora	Citeseer	PubMed
Node	2485	2120	19717
Edge	5069	3679	44324
Fast [61]	0.184	0.068	1.816
Gudhi [50]	0.045	0.023	1.696
PDGNN	**0.006**	**0.005**	**0.007**

A.5.4 Why not directly approximating PIs

We believe directly estimated PIs will lose important structural information that can be crucial for downstream tasks. PI is only an approximation of the persistence diagram. The L2 distance between PIs does not accurately reflect the true Wasserstein distance between diagrams. Therefore, using an L2-distance-based loss to directly learn the PI may lead to the loss of important structural information carried by a diagram. An example is provided in Figure 7. For a sample vicinity graph from Cora, we compare the ground truth PI (computed from the ground truth diagram), the PI computed from the diagram estimated by our method PDGNN, and the PI directly estimated by GIN. Both estimated PIs have similar L2 distances from the ground truth PI. But we observe that the PI estimated by PDGNN has a very similar spatial distribution to the ground truth PI. This structural property, however, is not preserved by the directly estimated PI. Such loss of structural information of directly estimated PIs, although not captured by the L2 error, partially explains their worse representation power. In Table 2 and Table 3 in the main paper, the directly estimated PIs (PEGN(GIN_PI) and TLC-GNN(GIN_PI)) perform worse in the downstream task.

This choice of estimating diagrams instead of PIs is a part of the overarching theme of our paper. Note that the main contribution of our paper is to transfer a complicated and uncontrollable learning process to a controllable process with algorithmic insight. This general principle also applies to our learning algorithm. We decompose the diagram computation algorithm into sub-algorithms, which can be approximated well by a GNN. This is the reason PDGNN approximates the diagrams much better than other baselines in Table 1 in the main paper.

A.5.5 Experiments on large and sparse datasets.

In the experiments in the main paper, the input is the k-hop vicinity graphs. On citation graphs, the vicinity graph remains small. On these small graphs, the exact sequential algorithm like Gudhi has less overhead, and thus is unsurprisingly faster.

Indeed, on large and sparse graphs, our method outperforms strong baselines like Gudhi significantly. In Table 9, we compare the running time (in seconds) on popular citation networks including Cora, Citeceer, and PubMed. For each graph, we run experiments on the largest connected subgraph. We also report the number of nodes/edges of the selected subgraph.

Table 10: Experiments on the choice of filter functions.

Dataset	Cora		Citeseer		PubMed	
Filter	clustering	centrality	clustering	centrality	clustering	centrality
Evaluation on approximation error						
W_2	0.392	0.332	0.178	0.237	0.267	0.322
PIE	1.53e-3	4.14e-4	7.17e-4	4.65e-4	2.5e-3	3.75e-4
Evaluation on Time (s)						
Fast [61]	2.10	2.21	1.16	1.31	38.07	39.05
Gudhi [50]	0.98	1.00	0.59	0.63	16.79	16.24
PDGNN	11.25	11.30	13.61	13.62	66.19	67.29

Table 11: Experiments on the threshold value.

Node	80	84	88	92	96	100	104	108	112	116	120
Edge	515	585	660	713	759	820	943	1012	1060	1152	1231
Fast [61]	6.8e-3	8.0e-3	8.9e-3	9.6e-3	1.0e-2	1.1e-2	1.3e-2	1.4e-2	1.4e-2	1.6e-2	1.7e-2
Gudhi [50]	**2.5e-3**	**3.2e-3**	**3.6e-3**	**3.9e-3**	**4.0e-3**	4.8e-3	5.5e-3	6.0e-3	6.4e-3	6.6e-3	6.8e-3
PDGNN	4.5e-3	4.5e-3	4.6e-3	4.6e-3	4.6e-3	**4.6e-3**	**4.7e-3**	**4.7e-3**	**4.7e-3**	**4.7e-3**	**4.8e-3**

A.5.6 Experiment on the choice of filter functions/other graph metrics.

In Table 10, we set degree centrality and clustering coefficient as the filter function, follow the settings in Table 1 and report the approximation error on Cora, Citeseer, and PubMed. We also report computation time following the setting in Table 4. The only difference is that below we report the time to generate all vicinity graphs (rather than 1000 graphs as in Table 4).

We observe that (1) the filter function only has a minor influence on inference/computation speed, for both the sequential algorithm and ours; (2) the filter function does influence the approximation error. The reason is that different filter functions have different ranges; functions with larger ranges tend to have larger approximation errors, especially on PIE. This is another evidence that the distance function on PIs is not very robust for learning.

A.5.7 Experiments on the threshold value of average node/edge to decide which method is the fastest to compute/estimate EPDs.

To find the threshold, we use the well-known Stochastic Block Model (SBM) [24] to generate synthetic graphs. We set the number of nodes in these synthetic graphs from 200 to 300, with 10 as the step. In these graphs, we randomly generate 5 different clusters, and set the probability of edges intra-cluster to 0.4, and the probability of edges inter-cluster to 0.1. In this way, we can obtain 11 graphs with different nodes and edges. We set node degree as the filter function, and add experiments on the largest connected components of these 11 graphs. The information of the selected connected graphs and the running time (second) are listed in Table 11. As shown in the Table, the threshold is around 100 nodes / 820 edges.

We also evaluate the influence of density. We fix the node number of the SBM model to 250, and set the probability of edges intra-cluster from 0.5 to 0.7, and the probability of edges inter-cluster from 0.05 to 0.15. The steps for intra-cluster and inter-cluster are 0.02 and 0.01, respectively. In this

Table 12: Experiments on the threshold value.

Node	100	100	100	100	100	100	100	100	100	100	100
Edge	489	529	595	652	766	842	968	1011	1082	1231	1307
Fast [61]	7.0e-3	7.4e-3	8.3e-3	9.0e-3	1.1e-2	1.2e-2	1.3e-2	1.4e-2	1.4e-2	1.6e-2	1.7e-2
Gudhi [50]	**2.8e-3**	**2.9e-3**	**3.0e-3**	**4.1e-3**	**4.2e-3**	5.1e-3	5.5e-3	5.9e-3	6.2e-3	6.3e-3	6.7e-3
PDGNN	4.1e-3	4.1e-3	4.2e-3	4.2e-3	4.3e-3	**4.4e-3**	**4.7e-3**	**4.7e-3**	**4.8e-3**	**4.8e-3**	**4.8e-3**

way, we can obtain 11 graphs with the same nodes and different edges. We set node degree as the filter function, and add experiments on the largest connected components of these 11 graphs. The information of the selected connected graphs and the running time (second) are also listed in Table 12. As shown in the Table, the threshold is around 100 nodes / 766 edges.

A.5.8 Limitation of the paper.

First, in certain cases like Figure 6 (d), the model only captures a tendency of the EPD. This can be because that the distribution of the EPD of the selected graph is seldom in the training samples. Therefore, it is hard for the model to estimate these EPDs correctly.

Second, topological features are just one side of the data. In many cases, only using topological features such as EPDs to represent the information of graphs is not enough. A better way is to introduce other information such as the semantic information of graphs as complementary.