NEURAL APPROXIMATION OF EXTENDED PERSISTENT HOMOLOGY ON GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Persistent homology is a widely used theory in topological data analysis. In the context of graph learning, topological features based on persistent homology have been used to capture potentially high-order structural information so as to augment existing graph neural network methods. However, computing extended persistent homology summaries remains slow for large and dense graphs. Inspired by recent success in neural algorithmic reasoning, we propose a novel learning method to compute extended persistence diagrams on graphs. The proposed neural network aims to simulate a specific algorithm and learns to compute extended persistence diagrams for new graphs efficiently. Experiments on approximating extended persistence diagrams and several downstream graph representation learning tasks demonstrate the effectiveness of our method. Our method is also efficient; on large and dense graphs, we accelerate the computation by nearly 100 times.

1 INTRODUCTION

In recent years, much effort has been made to improve the expressiveness of graph neural networks (GNNs). Among these methods, an widely used approach is to explicitly inject other high order information, such as graph topological/structural information, into the GNN models (You et al., 2019; Li et al., 2020). To this end, persistent homology (Edelsbrunner et al., 2000; Edelsbrunner & Harer, 2010), which captures topological structures (e.g., connected components and loops) and encodes them in a summary called *persistence diagram (PD)*, has been injected to machine learning pipelines for various graph learning tasks (Zhao & Wang, 2019; Zhao et al., 2020; Hofer et al., 2020; Carrière et al., 2020; Chen et al., 2021; Yan et al., 2021). In particular, it has been found helpful to use the so-called *extended persistence diagrams (EPDs)* (Cohen-Steiner et al., 2009), 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 (Zhao et al., 2020) or link prediction (Yan et al., 2021), 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.

Our goal is to develop a learning-based framework to compute the EPDs for graphs efficiently. Recent works on algorithmic learning on graphs (Veličković et al., 2019; Xhonneux et al., 2021) showed that sequential algorithms such as Dijkstra and Depth-First Search (DFS) can be approximated by GNNs. Inspired by these works, we rewrite the algorithm to compute EPD into a sequential-like form and propose a novel neural framework to estimate the EPDs whose architecture is aligned with this algorithm. Experiments show that on large and dense graphs, the proposed framework is much faster than the direct computation of EPDs.

Compared with the sequential algorithms proposed in (Xhonneux et al., 2021), the algorithm to compute EPDs requires extra steps. To address these challenges, we propose several modules to approximate these steps. Using these modules, we empirically show that our method achieves a satisfying approximation quality of EPDs on different graphs. To further evaluate the effectiveness of our framework, we perform experiments on two downstream graph representation learning tasks: node classification and link prediction. We show that the deep learning models using the predicted EPDs perform comparably with the architectures using the ground-truth EPDs. In other words, the approximated EPDs do not lose learning power.

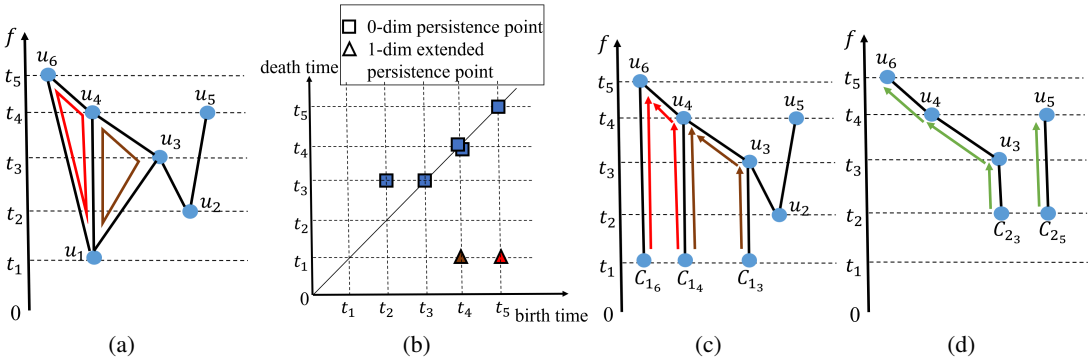

Figure 1: An explanation of extended persistent homology and its computation. (a) The input graph is plotted with a given filter function. (b) the extended persistence diagram of (a). Commonly speaking, the persistence points on the diagonal (uncritical points) should not be plotted. We plot these points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

To the best of our knowledge, we are the first to directly estimate EPDs on graphs with deep learning models. We note that our method is fundamentally different from previous works learning to directly approximate persistence images, a vectorization of persistence diagrams (Som et al., 2020; Montufar et al., 2020). These methods fail to simulate the computation algorithm closely. Furthermore, their training loss does not respect the special metric of EPDs. Therefore their output is not satisfying in both approximation quality and learning power. Finally, we observe that our model can be easily transferred to unseen graphs. This is encouraging as we may now generalize topological computation to various challenging real-world graphs without extra effort.

2 Related Works

Learning with Persistent Homology. Based on the theory of algebraic topology (Munkres, 2018), persistent homology (Edelsbrunner et al., 2000; Edelsbrunner & Harer, 2010) extends the classical notion of homology, and can capture the topological structures (e.g., loops, connected components) of the input data in a robust (Cohen-Steiner et al., 2007) manner. It has already been combined with various deep learning methods including kernel machines (Reininghaus et al., 2015; Kusano et al., 2016; Carriere et al., 2017), convolutional neural networks (Hofer et al., 2017; Hu et al., 2019; Wang et al., 2020; Zheng et al., 2021), transformers (Zeng et al., 2021), connectivity loss (Chen et al., 2019; Hofer et al., 2019). and graph neural networks (Zhao et al., 2020; Chen et al., 2021; Yan et al., 2021; Zhao & Wang, 2019; Hofer et al., 2020; Carrière et al., 2020).

Neural Algorithm Execution. Many works have studied neural execution in different domains before (Zaremba & Sutskever, 2014; Kaiser & Sutskever, 2015; Kurach et al., 2015; Reed & De Freitas, 2015; Santoro et al., 2018; Yan et al., 2020). With the rapid development of GNNs in graph representation learning, learning graph algorithms with GNNs has attracted researchers' attention (Veličković et al., 2019; 2020; Xhonneux et al., 2021). These works exploit GNNs to approximate certain classes of graph algorithms, such as parallel algorithms and sequential algorithms.

3 Extended Persistent Homology and Learning Extended Persistence Diagrams

Extended Persistent Homology. Extended persistent homology captures 0-dimensional (connected components) and 1-dimensionsl (loops) topological structures and summarizes their topological information into the so-called *extended persistence diagram (EPD)*, which is a planar multiset of points, each of which (b, d) corresponds to the information of some homological feature (i.e., components, loops, and their higher dimensional analogs). For example, for the input graph in Figure 1 (a), its output EPD is shown in Figure 1 (b). Further details around extended persistent homology are available in Section A.1 in the appendix.

Computing Extended Persistence Diagrams. Xhonneux et al. (2021) point out that sequential algorithms (e.g., Dijkstra) can be approximated by GNNs. Inspired by their idea, we rewrite the algorithms to compute EPDs in a sequential-like form that is easier for a GNN-like architecture to "simulate". The algorithms are listed in the Appendix.

Note that when we perform the standard persistence algorithm via matrix reduction, every simplex in the input simplicial complex will either be a *creator* (indicating that adding this simplex will create a new family of homology classes/topological features) or a *destroyer* (indicating that adding this simplex will destroy some existing homology classes/topological features). In the end, the persistence algorithm will pair up these creators and destroyers, and their function values give rise to the persistence points in the resulting persistence diagrams. In our context where the input is a graph which can be viewed as a 1-dimensional simplicial complex (consisting of 0-simplices/vertices V and 1-simplices/edges E), the persistence algorithms will pair up these simplices. In other words, to compute EPDs, we will simply find the pairing partners for all edges.

Description of the Sequential algorithm. Here we focus on describing the algorithm to compute 1D EPD. Further details are available in the Appendix. Our sequential algorithm, as shown in the Appendix, exploits the observations from (Agarwal et al., 2006). Specifically, consider a vertex $v_i \in V$, and suppose there are k edges $e_{i_1}, \ldots, e_{i_k}$ incident to v_i with function values higher than v_i (i.e., the function value of the other endpoint of these edges is higher than $f(v_i)$). See Figure 1 (c) for an illustration. For node u_1, $k = 3$, and the three edges are u_1u_3, u_1u_4, and u_1u_6. Now imagine we put each such edge in a different component C_{i_j}, $j \in [1, k]$ – we call this *upper-edges splitting operation* – and start to sweep the graph G in increasing values of a but starting at $f(v_i)$. Then, the first time any two such components merge will give rise to a new persistence point in the 1D EPD. For instance, in Figure 1 (c), C_{1_4} and C_{1_3} first merge at u_4, and this will give rise to the brown loop in Figure 1 (a) with (t_4, t_1) as its persistence point. Intuitively, this pairing captures the so-called thinnest cycle basis (Agarwal et al., 2006; Cohen-Steiner et al., 2009).

Hence to compute the extended persistence pairing induced by v_i, we can call Algorithm 3 to identify the first time when components containing C_{i_j}s are merged. This can be achieved using the union-find-like data structure to track the components. The main difference from the standard union-find data structure is that as we merge components, each component needs to be represented by the minimum (the vertex in this component with the smallest function value) which is not required in a standard union-find data structure. (The same holds for the algorithm to compute the 0D PD.)

Learning Extended Persistence Diagrams. Considering that every edge in the input graph will give rise to either a 0D ordinary persistence point or a 1D extended persistence point, we transfer the learning of EPDs into a link prediction problem. Specifically, our base architecture follows standard link-prediction architectures (Chami et al., 2019; Yan et al., 2021): (1) For a input graph, we first use a specially designed GNN model which later we call PDGNN to obtain the node embedding for all these vertices. (2) Subsequently, a MLP (Multi-layer perceptron) is applied to the node embeddings to obtain the persistent pairing information for each edge.

Our specially designed GNN for estimating persistence diagrams, is called **PDGNN** (Persistence Diagram Graph Neural Network). Compared with the Sequential algorithms proposed in (Xhonneux et al., 2021), extended persistence diagrams need extra care: (1) the Find-Root algorithm needs to return the minimum of the component, (2) edge operations such as upper-edge splitting. Our PDGNN modifies standard GNNs by (1) implementing the root-finding process by a combination of sum aggregation and min aggregation as our message aggregation function; (2) incorporating edge operations such as the upper-edges splitting operation with edge features and edge attention. Further details are available in the Appendix.

4 EXPERIMENTS

In this paper, we thoroughly evaluate the proposed model from 3 different perspectives: approximation quality, transferability, and algorithm efficiency. Due to the page limit, we put the evaluation on approximation quality and transferability to the Appendix, and briefly introduce the results in the following paragraph. In the evaluation on algorithm efficiency. Experiments demonstrate that the proposed method is much faster than the original algorithm, especially on large and dense graphs.

Table 1: 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 (Yan et al., 2021)	0.95	0.39	2.15	362.60	1195.66	5.72	24.14
Gudhi (The GUDHI Project, 2015)	**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**

Datasets. To compute EPDs, we need to set the input graphs and the filter functions. Following the settings of existing state-of-the-art models (Zhao et al., 2020; Yan et al., 2021), for a given graph $G = (V, E)$, we extract the k-hop neighborhoods of all the vertices, and extract $|V|$ vicinity graphs as input graphs. In terms of filter functions, we introduce Ollivier-Ricci curvature Ni et al. (2018), heat kernel signature, and the node degree as the filter function. The datasets include (1) citation networks including Cora, Citeseer, and PubMed (Sen et al., 2008); (2) Amazon shopping datasets including Photo and Computers (Shchur et al., 2018); (3) coauthor datasets including CS and Physics (Shchur et al., 2018). Details are available in the Appendix.

Approximation Quality. In Section A.4.1, we evaluate the approximation error between the predicted diagram and the ground truth diagram and show that the prediction is very close to the original diagram. We also add ablation study to show the effectiveness of all the proposed modules. To understand how much does the approximation error influence downstream tasks, in Section A.4.2, 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. From the two evaluation, we can safely conclude that the predicted diagram is a wonderful substitution of the original EPD in terms of both approximation error and learning power.

Transferability. One appealing feature of our method is its 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; we can apply a pre-trained model to large and dense graphs, on which direct EPD computation is infeasible.

We prove the transferability empirically. In Table 4, we adopt the model pre-trained on Photo to predict the EPDs of CS and Physics, and achieve good approximation performance. We provide comprehensive experiments to evaluate the transferability of our methods. See Table 6 and Table 7 in the Appendix for the approximation quality and learning power evaluation of transferred models.

Algorithm Efficiency. For a fair and complete comparison, we compare with algorithms from Gudhi (The GUDHI Project, 2015) and from (Yan et al., 2021). We select the first 1000 nodes from Cora, Citeseer, PubMed, Photo, Computers, CS, Physics, and then extract their 2-hop neighborhoods as the input vicinity graphs. We then compute the EPDs and report the time (seconds) used to infer these diagrams.

We list the average nodes and edges of these vicinity graphs in the first line of Table 1. As shown in Table 1, 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 extended persistence diagrams on small graphs, and use our model to execute extended persistence diagrams on large graphs. The model can be applied to various graph representation learning works based on persistent homology.

5 Conclusion

We propose a learning algorithm to approximate EPDs. Inspired by recent success on neural algorithm execution, we propose a novel GNN with different technical contributions to simulate the computation of EPDs. Experiments show that our method achieves satisfying approximation quality and learning power while being significantly faster than the original algorithm, especially 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.

REFERENCES

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.

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

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

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

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*, pp. 2786–2796. PMLR, 2020.

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

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*, pp. 2573–2582. PMLR, 2019.

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

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

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

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

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

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

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.

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*, pp. 1633–1643, 2017.

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

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

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*, pp. 2305–2312, 2014.

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

Łukasz Kaiser and Ilya Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

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

Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. *arXiv preprint arXiv:1511.06392*, 2015.

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

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.

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.

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. URL `www.graphlearning.io`.

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

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*, pp. 447–462. Springer, 2018.

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

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

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*, pp. 4741–4748, 2015.

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.

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.

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

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*, pp. 834–835, 2020.

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, pp. 1383–1392. Wiley Online Library, 2009.

The GUDHI Project. *GUDHI User and Reference Manual*. GUDHI Editorial Board, 2015. URL http://gudhi.gforge.inria.fr/doc/latest/.

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

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

Petar Veličković, Lars Buesing, Matthew C Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *arXiv preprint arXiv:2006.06380*, 2020.

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*, pp. 118–136. Springer, 2020.

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.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018.

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.

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*, pp. 11659–11669. PMLR, 2021.

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

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

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

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

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.

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

Songzhu Zheng, Yikai Zhang, Hubert Wagner, Mayank Goswami, and Chao Chen. Topological detection of trojaned neural networks. *arXiv preprint arXiv:2106.06469*, 2021.

A APPENDIX

In the Appendix, we provide (1) Introduction of extended persistent homology and its computation; (2) additional experimental details, including introduction on the datasets, and the experimental settings; (3) further experiments, including the evaluation on approximation quality, the evaluation on transferability, and the influence of training samples; (4) experiments on graph classification datasets.

A.1 EXTENDED PERSISTENT HOMOLOGY

We briefly introduce extended persistent homology and refer the readers to (Cohen-Steiner et al., 2009; Edelsbrunner & Harer, 2010) for more details.

Ordinary Persistent Homology. Persistent homology captures 0-dimensional (connected components) and 1-dimensionsl (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* (Cohen-Steiner et al., 2009)[1] 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 1-dimensional EPD consists of β_1 number of persistent 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 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, persistence diagrams live in an infinite-dimensional space equipped with an appropriate metric structure, such as the so-called p-th Wasserstein distance, or the bottleneck distance (Cohen-Steiner et al., 2007). There have been many works in the literature to vectorize persistence diagrams for downstream analysis. A popular choice is the persistence image (Adams et al., 2017),

A.1.1 COMPUTING EXTENDED PERSISTENCE DIAGRAMS (EPDS)

In this section, we rewrite the algorithms to compute the EPDs for graphs into a form that is easier for GNNs to simulate as inspired by the neural execution work of (Veličković et al., 2019; Xhonneux et al., 2021). The algorithm for computing the 0D EPDs is well-known, using the so-called union-find data structure (Edelsbrunner & Harer, 2010; Dey & Wang, 2022). The algorithm for computing the 1D EPDs has not been reported before, however, the idea follows from existing work (Agarwal et al., 2006). For simplicity of presentation, we assume that all vertices have distinct function values $f : V \to \mathbb{R}$, and our goal is to compute the 0D and 1D EPDs PD_0 and PD_1.

The neural execution of "sequential" algorithms. Xhonneux et al. (2021) point out that sequential algorithms as shown in Algorithm 1 (e.g., Dijkstra) can be approximated by GNNs.

[1]Extended persistence is also closely related to the concept of zigzag persistence (Carlsson & De Silva, 2010).

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**
      $s/l = argmin/argmax(pu.value, pv.value)$
      $l.root = s$
      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

Inspired by their idea, we rewrite the algorithms to compute EPDs in a sequential-like form that is easier for a GNN-like architecture later to "simulate". In particular, the algorithm for computing 1D extended persistence is shown in Algorithm 2; the algorithm for computing 0D persistence, which is shown in Algorithm 4, is the standard union-find strategy.

High-level description of sequential algorithms. Note that when we perform the standard persistence algorithm via matrix reduction, every simplex in the input simplicial complex will either be a *creator* (indicating that adding this simplex will create a new family of homology classes/topological features) or a *destroyer* (indicating that adding this simplex will destroy some existing homology classes/topological features). In the end, the persistence algorithm will pair up these creators and destroyers, and their function values give rise to the birth and death times in the resulting persistence diagrams. In our context where the input is a graph which can be viewed as a 1-dimensional simplicial complex (consisting of 0-simplices/vertices V and 1-simplices/edges E), the persistence algorithms will pair up these simplices.

In particular, for an edge (1-simplex) $e \in E$: (1) it can either serve as a destroyer (killing a connected component), in which case it is paired up with a vertex v, giving a *persistence pairing* (v, e) and a corresponding persistence point $(f(v), f(e))$ in the 0-D PD PD_0. 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. Alternatively, the edge could be creating a 1-cycle (intuitively a loop in the graph) and is a creator during the ascending filtration. This loop will ultimately be killed during the descending filtration when we take relative homology $\mathrm{H}_1(X, X^a)$, when we sweep past some vertex w (and $a = f(w)$). Hence the algorithm will pair up (e, w) which gives rise to a persistence point $(f(e), f(w))$ in the 1-D EPD PD_1.

In other words, to compute PD_0 and PD_1, we will simply find the pairing partners for all edges. We note that in the literature, when the filtration is induced by a filter function, often in the end we output the persistence diagram induced by the lower-star filtration, which roughly means that we ignore all local pairings where a simplex is paired with another one incident on it. Such local pairings correspond to those persistence points with 0 persistence if the function value is distinct on all vertices. We compute all pairings as this has the consequence that later our neural network will essentially compute a "pairing partner" for each edge in the graph as an edge feature. Once such pairings are computed, it is easy to remove those local pairings.

Sequential algorithm to compute PD_1. The algorithm to compute PD_0 is similar to Algorithm 3. Therefore, here we just describe the algorithm to compute PD_1. It turns out that using the observations from (Agarwal et al., 2006), the following procedure will produce the 1D extended persistence capturing loops. Specifically, consider a vertex $v_i \in V$, and suppose there are k edges $e_{i_1}, \ldots, e_{i_k}$ incident to v_i with function values higher than v_i (i.e., the function value of the other endpoint of these edges is higher than $f(v_i)$). See Figure 1 (c) for a simple illustration. For node u_1, $k = 3$, and the three edges are u_1u_3, u_1u_4, and u_1u_6. Now imagine we put each such edge in a different component C_{i_j}, $j \in [1, k]$ – we call this *upper-edges splitting operation* – and start to sweep the graph G in increasing values of a but starting at $f(v_i)$. Then, the first time any two such components merge will give rise to a new persistence point in the 1D extended persistence diagram. For instance, in Figure 1 (c), C_{1_4} and C_{1_3} first merge at u_4, and this will give rise to the brown loop in Figure 1 (a) with (t_4, t_1) as its persistence point. Intuitively, this pairing captures the so-called thinnest cycle basis (Agarwal et al., 2006; Cohen-Steiner et al., 2009).

Hence to compute the extended persistence pairing induced by v_i, we can call Algorithm 3 to identify the first time when components containing C_{i_j}s are merged. This can be achieved using the union-find-like data structure to track the components. The main difference from the standard union-find data structure is that as we merge components, each component needs to be represented by the minimum (the vertex in this component with the smallest function value) which is not required in a standard union-find data structure. (The same holds for the algorithm to compute the 0D persistence diagram PD_0 in the Appendix.) The detailed algorithm/procedures involved can be found in Algorithm 3.

Complexity. The computational complexity of the Union-Find algorithm is $O(|E|\alpha(|E|))$, where $\alpha(\cdot)$ is the inverse Ackermann function. Therefore, we need $O(|V||E|\alpha(|E|))$ time to compute an EPD using the sequential algorithm described in this section. Note this sequential algorithm is not necessarily the most efficient one. In practice, one may use the quadratic algorithm ($O(|V||E|)$) as in (Yan et al., 2021). 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 (Agarwal et al., 2006; Georgiadis et al., 2011). But this algorithm remains theoretical so far.

Algorithm 4 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.2 Learning Extended Persistence Diagrams

In this section, we will introduce our neural network architecture to approximate the EPDs on graphs. As explained in the previous section, we will pair up all simplices (edges/vertices) in the input graph, instead of outputting only those non-local pairing corresponding to the pairing between critical points in the smooth case. The reason is that in this way, every edge is paired and corresponds to a persistence point, therefore the computation of EPD can be transformed into a link prediction problem on graphs. Furthermore, once such pairings are computed, we can easily remove all local pairings.

Basic Framework. As mentioned in Section A.1.1, every edge in the input graph will give rise to either a 0-dim ordinary persistence point or a 1D extended persistence point. Therefore, we transfer the learning of extended persistence diagram into a link prediction problem. Specifically, our base architecture, which is shown in Figure 2, follows standard link-prediction architectures (Chami et al., 2019; Yan et al., 2021): (1) For a 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 the persistent pairing information for each edge $(u, v) \in E$ via $PP_{uv} = W([h_u \bigoplus h_v]) \in R^2$ which is the persistence point associated to this edge (u, v). Here, h_u and h_v denote the node embedding for node u and v, and $\bigoplus$ represents the concatenation of vectors.

Standard GNN. A GNN typically learns the node embedding via an iterative aggregation of local graph neighbors. The k-th iteration (the k-th GNN layer) can be written 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 (Xu et al., 2018), MSG is a MLP followed by an activation function, and AGG is a sum aggregation.

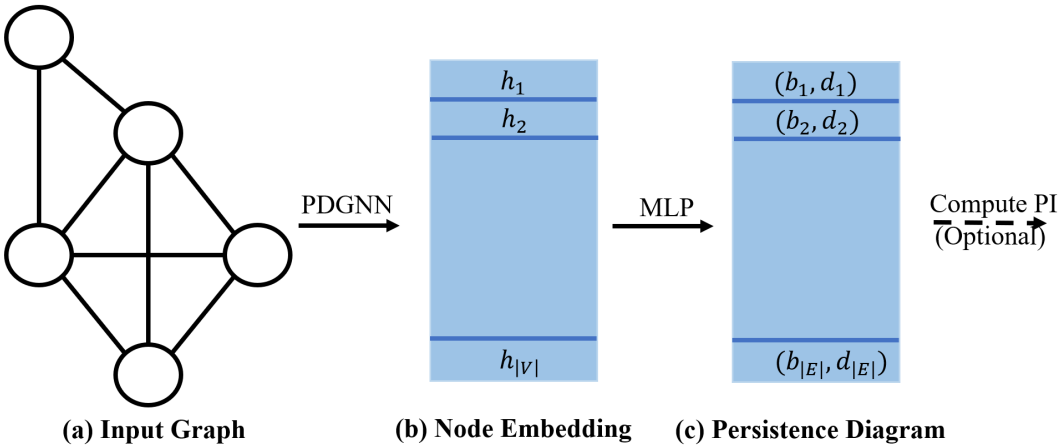

Figure 2: The basic framework.

PDGNN. We now describe our specially designed GNN for estimating persistence diagrams, called **PDGNN** (Persistence Diagram Graph Neural Network). Compared with the Sequential algorithms proposed in (Xhonneux et al., 2021), extended persistence diagrams need extra care: (1) the Find-Root algorithm needs to return the minimum of the component, (2) edge operations such as upper-edge splitting. Our PDGNN modifies standard GNNs with the following modules to handle these steps.

Find-Root Algorithm. 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 (Xu et al., 2018), we implement the root-finding process by a combination of sum aggregation and min aggregation as our AGG function. To be specific:

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

Incorporating edge operations. As shown in (Veličković et al., 2019; Xhonneux et al., 2021), classic GNNs are not effective in "executing" algorithms operating on edges, such as Relax-Edge subroutine. Furthermore, in Algorithm 2, we also need the upper-edge splitting operation for each vertex. To this end, similar to (Xhonneux et al., 2021) we also use edge features and attention to provide bias using edges. Specifically, we propose the following MSG:

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

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 GAT (Veličković et al., 2018) as our edge weight.

Given that the Union-Find-step algorithm should be implemented on all nodes to obtain the EPDs, ideally we need a large GNN model to simulate all the computations. 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. Therefore we expect a limited-size single GNN model can simultaneously approximate all EPD computations. This is confirmed by empirical evidence in Section 4.

A.3 Experimental Details

A.3.1 Datasets.

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

1. Citation networks: Cora, Citeseer, and PubMed (Sen et al., 2008) are standard citation networks where nodes denote scientific documents and edges denote citation links.

Table 2: 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

2. Amazon shopping records: In Photo and Computers (Shchur et al., 2018), 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 (Shchur et al., 2018), nodes denote authors and edges denote that the two authors co-author a paper.

The detailed statistics are available in Table 2.

To compute EPDs, we need to set the input graphs and the filter functions. Existing state-of-the-art models on node classification (Zhao et al., 2020) and link prediction (Yan et al., 2021) mainly focus on the local topological information of the target node(s). Following their settings, for a given graph $G = (V, E)$ (e.g., Cora), 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 (Ni et al., 2018)[2], heat kernel signature with two temprature values (Sun et al., 2009; Hu et al., 2014)[3] and the node degree[4]. For an input vicinity graph, we compute 4 EPDs based on the 4 filter functions, and then vectorize them to get 4 peristence images (Adams et al., 2017). Therefore, we can get $4|V|$ EPDs in total.

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.

A.3.2 Experimental Details

In this section, we mainly present the experimental settings on neural execution, as for the setting in downstream graph representation learning tasks, we are consistent with (Zhao et al., 2020; Yan et al., 2021).

Evaluation metrics. Recall that the input of our model is a graph and a filter function, and the output is the predicted diagram. After obtaining the predicted diagram, we vectorize it with persistence image (Adams et al., 2017) 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 compute extended persistence diagrams 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 all different filter functions.

Other settings. Following the settings in (Zhao et al., 2020; Yan et al., 2021), 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

[2]Following the settings in (Zhao et al., 2020; Yan et al., 2021), we adopt the Ollivier-Ricci curvature as the graph metric, and the distance to target node(s) as the filter function.

[3]Following the settings in (Carrière et al., 2020), we set the temparature $t = 10$ and 0.1 and adopt these two kernel functions as the filter functions.

[4]Node degree is used as the initial filter function in (Hofer et al., 2020).

Table 3: 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 4: 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

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 whole model, 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 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 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. In case of these 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, and report the normal W_2 distance, that is, to let the predicted points pair with the diagonal in the inference stage.

A.4 Additional Experiments

A.4.1 Approximation Quality

In this section, we evaluate the approximation error between the predicted EPDs and the ground truth EPDs.

Table 5: 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)	93.0±0.2	92.8±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**

Evaluation metrics. Recall that the input of our model is a graph and a filter function, and the output is the predicted diagram. After obtaining the predicted diagram, we vectorize it with persistence image Adams et al. (2017) 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 compute extended persistence diagrams 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 AGG function and MSG function proposed in Section A.2. Its strategy is to first predict the EPD, and then convert it into the persistence image. To show the superiority of the strategy, we compare with the strategy from Som et al. (2020); Montufar et al. (2020), 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 Xu et al. (2018) and GAT Veličković et al. (2018) as the backbone GNNs.

To show the effectiveness of the modules proposed in Section A.2, we add other baselines with our proposed strategy. **GAT** denotes GAT as the backbone GNN. **GAT (+MIN)** denotes GAT with the 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 Appendix.

Results. Table 3 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. Comparing PDGNN (w\o ew) and PDGNN, we find the use of the edge weight being helpful. Edge weights help the model focus on the individual Relax-Edge sub-algorithm operated on every edge.

A.4.2 Downstream Tasks

In this section, we evaluate the performance of the predicted diagrams on 2 graph representation learning tasks: node classification and link prediction. To be specific, we replace the ground truth EPDs in state-of-the-art models based on extended persistent homology (Zhao et al., 2020; Yan et al., 2021) 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** (Kipf & Welling, 2016), **GAT** (Veličković et al., 2018) and **HGCN** (Chami et al., 2019). For link prediction, we compare with several state-of-the-art methods such as **SEAL** (Zhang & Chen, 2018) and **P-GNN** (You et al., 2019). Notice that GCN and GAT are not originally designed for link prediction, therefore we follow the settings in (Chami et al., 2019; Yan et al., 2021), that is, to get the node embedding through these models, and use the Fermi-Dirac decoder (Krioukov et al., 2010; Nickel & Kiela, 2017) to predict whether there is a link between the two target nodes. In comparison with the original extended persistence diagram, we also add **PEGN** (Zhao et al., 2020) and **TLC-GNN** (Yan et al., 2021) 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 (Kipf & Welling, 2016; Veličković et al., 2018; Zhao et al., 2020). 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 (Chami et al., 2019; Yan et al., 2021). To be precise, we randomly split existing edges into 85/5/10% for training, validation, and test sets. An equal number of non-existent

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

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 4 and Table 5 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.

A.4.3 Experiments on transferability

In this section, we design experiments to evaluate the transferability of PDGNN in terms of different graph structures. Our aim is that with a pre-trained model, or simply fine-tuning the pre-trained model can let the model predict EPDs on totally unseen graphs. Therefore, we use the models pre-trained on Photo, and report the W_2 distance between the predicted diagrams and ground truth EPDs. Notice that we only use Ollivier-Ricci curvature (Ni et al., 2018) 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-tune, the pre-trained model can achieve almost an equal performance compared with the standard setting. It justifies the fine transferability of PDGNN.

The fine transferability 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. In a totally new environment, instead of training the models without pre-training for many epochs, we can simply fine-tune or even directly use the pre-trained model to predict extended persistence diagrams on new graph structures.

A.4.4 Evaluation on the influence of training samples

In this section, we evaluate the influence of training samples on PDGNN. Our aim is 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 3 and Figure 4.

As shown in Figure 3, the training error tends to converge as the training samples gradually increase. Considering that the W_2 distance and PIE cannot directly reflect the influence as NCA does, we select a vicinity graph in Cora which is hard for PDGNN to learn and visualize in Figure 4. As shown in the figure, as the number of training samples increases, we find that PDGNN can gradually capture

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

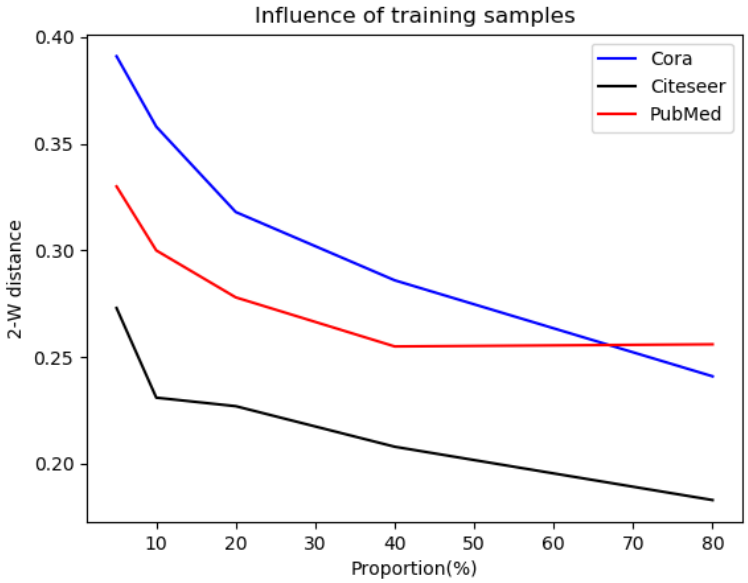

Figure 3: Influence of training samples.

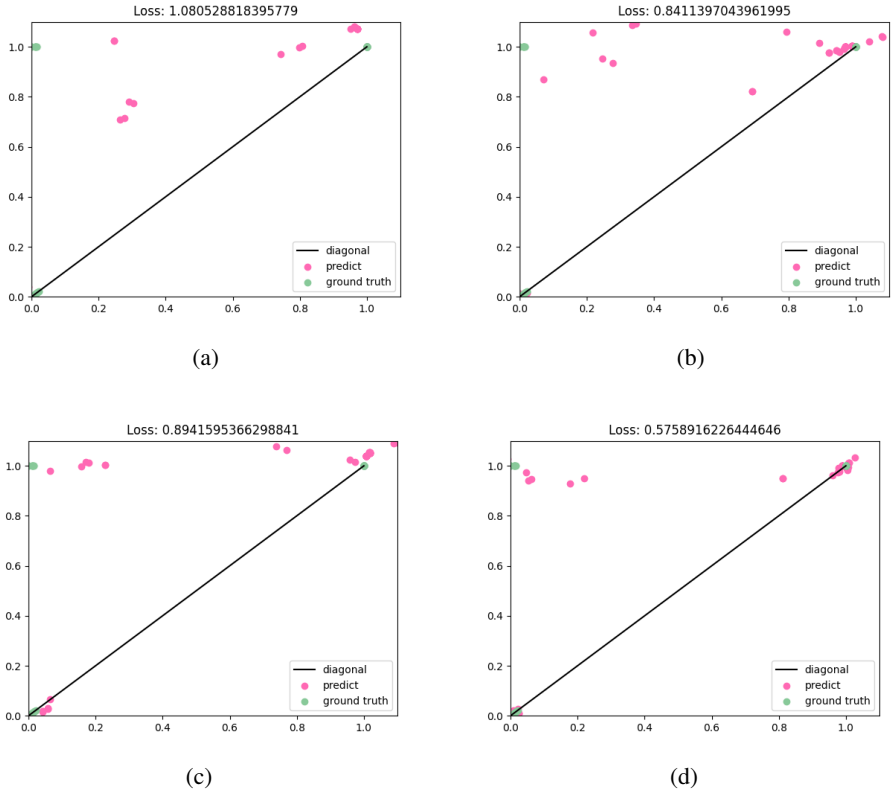

Figure 4: Visualization on the influence of training samples. We select a vicinity graph in Cora with Ollivier-Ricci curvature as the filter function, and plot the influence of training samples the W_2 distance (loss) of EPDs. (a), (b), and (c) denote the prediction of PDGNN with 5/10/20% training samples, (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
IMDB-BINARY	1000	19.8	193.1	0.176	4.13e-4
REDDIT-BINARY	2000	429.6	995.5	0.383	1.92e-4

the ground truth persistent 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 can also observe that in Table 7, PDGNN can reach 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.4.3 and Section A.4.4, we can draw a conclusion that our model can be easily generalized to other frameworks. PDGNN does not need many training samples to reach an acceptable performance, while it can be easily transferred to totally unseen graphs.

A.4.5 Experiments on graph classification datasets.

In the experiment part (Section 4), we only consider predicting EPDs in the k-hop neighborhoods of the original graphs. Even if these vicinity graphs can be large and dense, there are fundametal structural differences between these vicinity graphs and other real-world graphs. For example, in a k-hop neighborhood of node u, the distance between a node u_1 to the other node u_2 is at most $2k$: $d(u_1, u_2) \leq d(u_1, u) + d(u_2, u) \leq k + k = 2k$. However, in a real-world graph, the distance can be very long.

Observation. Assume that our aim is to find a loop generated by two connected components that originate from the same node. An example is shown in Figure 1 (c), in which the two connected components C_{1_4} and C_{1_3} are combined when node u_4 arises. In every message passing iteration, the Union-Find-step algorithm will only run the Relax-Edge function once. If starting from the connected component C_{1_4}, the algorithm needs 1 GNN layer to find node u_4. Similarly, if starting from the connected component u_{13}, the algorithm needs 2 GNN layers to find node u_4. Therefore, the algorithm needs $max(1, 2) = 2$ GNN layers to find node u_4. In conclusion, to find a loop generated by two connected components c_1 and c_2, the algorithm needs $max(d_1, d_2)$ layers to obtain the persistence point, where d_1 (resp. d_2) denotes the distance from c_1 (resp. c_2) to the node that the two connected components are combined. Therefore, we can use a $2k$-layer PDGNN to obtain the persistence points of all the potential loops in a k-hop neighborhood. In the settings of (Zhao et al., 2020; Yan et al., 2021), k is at most 2. Consequently in Section 4, we set the layer of PDGNN to 4 and achieve a promising approximation result.

In this section, we do further experiments on graph classification datasets, in which we approximate the EPDs on the original graph rather than the vicinity graphs. We exploit the datasets from the TU Dortmund University (Morris et al., 2020), 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 Table 4, we observe that the performance on graph classification datasets is generally worse. There are 2 possible explanations: (1) In the graph classification datasets, the training samples can be very small, e.g., there are only 188 graphs in MUTAG, therefore PDGNN has not well approximated the algorithm. (2) As discussed above, the diameter of these graphs can be large, therefore extra GNN layers are needed to execute the algorithm. However, we discover that the increasing of GNN layers only brings little improvement to the approximation error. This can be due to the well-known vanishing gradient problem in deep learning models.

To evaluate the results more clearly, we also visualize some selected examples in Figure 5. As shown in the figure, in most situations, PDGNN can well execute the EPDs on these graphs, and the W_2

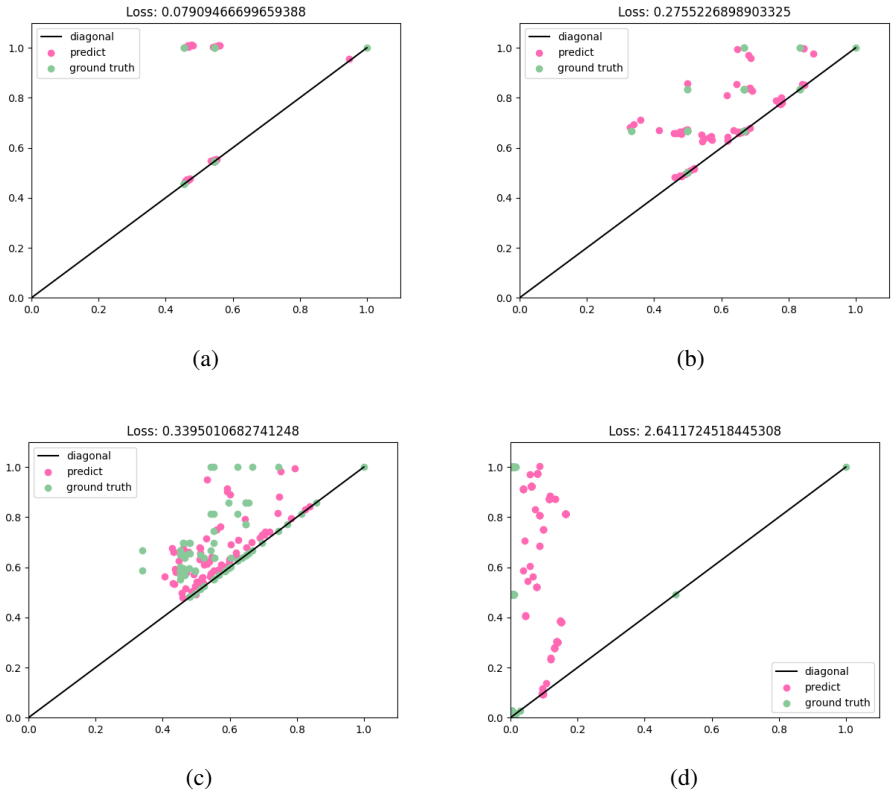

Figure 5: Visualization of graph classification samples. We select samples from IMDB-BINARY, PROTEINS, ENZYMES, and REDDIT-BINARY, respectively, and report the W_2 distance (loss).

distance around 0.3 is generally an acceptable result. However, in certain cases like Figure 5 (d), the model only captures a tendency of the EPD. This can be due to the long diameter of the graph.