PyTorch Geometric Signed Directed: A Software Package on Graph Neural Networks for Signed and Directed Graphs

Yixuan He
University of Oxford
yixuan.he@balliol.ox.ac.uk

Xitong Zhang
Michigan State University
zhangxit@msu.edu

Junjie Huang
Southwest University
junjiehuang@swu.edu.cn

Benedek Rozemberczki
Isomorphic Labs
benedek.rozemberczki@gmail.com

Mihai Cucuringu & Gesine Reinert
Univerisity of Oxford & The Alan Turing Institute
{mihai.cucuringu, reinert}@stats.ox.ac.uk

Abstract

Networks are ubiquitous in many real-world applications (e.g., social networks encoding trust/distrust relationships, correlation networks arising from time series data). While many networks are signed or directed, or both, there is a lack of unified software packages on graph neural networks (GNNs) specially designed for signed and directed networks. In this paper, we present PyTorch Geometric Signed Directed (PyGSD), a software package which fills this gap. Along the way, we evaluate the implemented methods with experiments with a view to providing insights into which method to choose for a given task. The deep learning framework consists of easy-to-use GNN models, synthetic and real-world data, as well as task-specific evaluation metrics and loss functions for signed and directed networks. As an extension library for PyG, our proposed software is maintained with open-source releases, detailed documentation, continuous integration, unit tests and code coverage checks. The *GitHub* repository of the library is https://github.com/SherylHYX/pytorch_geometric_signed_directed.

1 Introduction

Graph data arise in many areas, such as social networks, citation networks, and biochemical graphs. Such data are related to many learning problems [85, 92]. To tackle network inference tasks, graph neural networks (GNNs) are a useful tool. Based on deep learning techniques such as network architecture, optimization approaches and parallel computation, GNNs can be trained just like standard neural networks. By leveraging the network structure, GNNs are able to retain information from the nodes' neighborhood and incorporate long-range dependencies [92]. GNNs have a wide range of applications, such as social network analysis [36], traffic predictions [51], biological applications [48], recommender systems, computer vision, and natural language processing [85].

Although traditional network analysis usually focuses on a single fixed simple network, which could often be represented by a symmetric adjacency matrix with nonnegative entries, intricate network types are often more realistic. Signed networks, which have positive or negative edge weights, have been of interest in social network analysis [41, 69], with signs indicating positive or negative sentiments. The development of signed network algorithms has been motivated for example by the task of clustering time series with regards to correlations [3], including economic time series that capture macroeconomic variables [22], and financial applications [63, 94]. The empirical correlation matrix can be constructed as a weighted signed network, with signs indicating correlations, potentially sparsified after thresholding. Thresholds can be obtained for example as p-values of statistical tests of significance for individual correlations. Many signed networks are also directed, in that they have asymmetric sending and receiving patterns. Indeed, edge directionality could play an integral role in processing and learning from network data [55]; directed networks, even unsigned, have many applications such

Y. He et al., PyTorch Geometric Signed Directed: A Software Package on Graph Neural Networks for Signed and Directed Graphs. *Proceedings of the Second Learning on Graphs Conference (LoG 2023)*, PMLR 231, Virtual Event, November 27–30, 2023.

as clustering time-series data with lead-lag relationships [4], detecting influential groups in social networks [7, 35, 88], ranking [33], angular synchronization [37], and biological applications [55].

We note that there have been several GNNs [71, 80, 89] introduced for multi-relational graphs, i.e., graphs with different types of edges. There is typically a linear relationship between the number of parameters that can be learned in such networks and the number of edge types. In general, weighted signed graphs with real-valued edge weights are not just special cases of multi-relational graphs. If the graph is unweighted or the weighting function takes a finite number of values, however, we could view signed graphs as special multi-relational graphs. Nevertheless, there is an implicit relationship between the different edge types in signed graphs (possibly weighted), namely that edges with large weights are considered more important than edges with small weights and that negative edges are interpreted as the opposite of positive edges. As a result, if we consider an arbitrary multi-relational graph, we would have much more trainable parameters than using these relationships. Instead, GNN methods which are tailored for signed and directed networks, as presented in this library, are a useful addition to the tool set of GNNs for network analysis.

Moreover, deep learning frameworks [1, 8, 11, 62] have facilitated the emergence of open-source software on deep learning on graphs [17, 21, 26, 39, 68, 90]. Such open-source libraries are essential to the practical success and large-scale deployment of graph machine learning systems [68]. However, despite the popularity of signed and directed networks, there does not exist an open-source library to include GNNs as well as related data processing methods on them. This motivates the design of our open-source library, called *PyTorch Geometric Signed Directed* (PyGSD), which is an extension library of the popular *PyTorch Geometric* (PyG) [21], but is specially designed for signed and directed networks.

Main contributions. •(1) We present *PyTorch Geometric Signed Directed* (PyGSD), the first deep learning library for graph neural networks in signed and directed networks. This is an open-source library equipped with public releases, documentation, code examples, continuous integration, unit tests and code coverage checks. •(2) We provide easy-to-use GNN models, synthetic data generators and real-world data loaders, as well as task-specific evaluation metrics and loss functions for signed and directed networks in PyGSD. •(3) We evaluate task performance of GNNs available in PyGSD and provide insights into which methods to choose for which tasks.

2 Background

In this paper, let $\mathcal{G} = (\mathcal{V}, \mathcal{E}, w, \mathbf{X}_{\mathcal{V}})$ denote a (possibly signed, directed and weighted) network with node attributes, where $\mathcal{V}$ is the set of nodes, $\mathcal{E}$ is the set of (directed) edges or links, and $w \in (-\infty, \infty)^{|\mathcal{E}|}$ is the set of weights of the edges. The network $\mathcal{G}$ has adjacency matrix $\mathbf{A} = (A_{ij})_{i,j \in \mathcal{V}}$, where $\mathbf{A}_{ij} = w_{ij}$, the edge weight, if there is an edge from node v_i to node v_j; otherwise $\mathbf{A}_{ij} = 0$. Here, $\mathcal{G}$ could have self-loops, but no multiple edges. The total number of nodes is $n = |\mathcal{V}|$, and $\mathbf{X}_{\mathcal{V}} \in \mathbb{R}^{n \times d_{\text{in}}}$ is an attribute matrix whose rows are node attributes (which could be generated from $\mathbf{A}$), where d_{in} denotes the input attribute dimension. For a signed network, we split the edge set into positive and negative subsets $\mathcal{E} = \mathcal{E}^+ \cup \mathcal{E}^-$, where the weights for the edges in $\mathcal{E}^+$ and $\mathcal{E}^-$ are positive and negative, respectively. Here we use *network* and *graph* interchangeably.

2.1 Insights from Social Network Analysis: Social Balance Theory and Status Theory

Signed and possibly directed social networks are addressed by structural balance theory [38] and status theory [47] from sociology. A network is called *balanced* if all cycles have an even number of negative edges; otherwise it is unbalanced. Structural balance theory, proposed by Heider in 1946 [38], hypothesizes how humans are motivated to change their attitudes when their social networks are unbalanced. Its most famous example is the principle that *the friend of my friend is my friend and the enemy of my enemy is my friend*. Unlike balance theory, which is usually applied to undirected signed networks, status theory [27] considers the direction in signed directed networks. It supposes that a positive directed link "+" from A to B indicates that *target node B has higher status than source node A*; a negative directed link "-" from A to B indicates that *source node A has higher status than target node B*.

Incorporating such insights from social network analysis into GNNs requires specific tailoring and hence the development of particular tools. In particular, both sociological theories can be associated with signed triangles [12, 41]. Related works on graph representation learning which model the sociological theory described above include for example SGCN [19], SiGAT [40], SNEA [49], and SDGNN [41]. In PyGSD, these models are implemented, together with their loss functions (e.g., Structural Balance Loss in SGCN [19] and Signed Direction Loss in SDGNN [41]).

2.2 Spatial Graph Neural Networks

Spatial GNNs utilize information propagation among nodes to define convolutions on graphs [85]. The Message Passing Neural Network (MPNN) by [24] outlines a general framework of spatial GNNs, treating graph convolutions as a message-passing procedure among nodes and edges. The general form of message passing is defined as $\mathbf{h}_i^{(l)} = \text{UPDATE}^{(l-1)}\left(\mathbf{h}_i^{(l-1)}, \text{AGGREGATE}^{(l-1)}(\{\mathbf{h}_j : \mathbf{A}_{i,j} \neq 0\})\right)$ where UPDATE and AGGREGATE are arbitrary differentiable functions. However, these spatial methods naturally are not tailored to the analysis of signed and directed networks. In particular they do not take the specific insights from social network analysis into account, but can be adapted to do so. Some examples of such spatial GNNs for signed/directed graphs that could be viewed as instantiations of MPNNs are [35] and [19]. For signed networks, the aggregation function AGGREGATE provides the option to consider positive and negative neighbors separately, such as in [19] and [36]. The attention mechanism, which assigns an attention weight or importance to each neighbor, is also considered in signed directed GNNs under the MPNN framework [40, 41, 49]. Indeed, spectral GNNs could be implemented in a spatial way, as a special form of MPNN, as validated by [88] and [34].

2.3 Some Tools from Spectral Analysis: Graph Convolution and Laplacian Matrices

For a graph signal $\mathbf{x} \in \mathbb{R}^{|\mathcal{V}|}$ on an undirected graph $\mathcal{G}$, the conventional graph convolution is defined as $g_\theta \star \mathbf{x} = \mathbf{U} g_\theta \mathbf{U}^\top \mathbf{x}$, where $g_\theta = \text{diag}(\theta), \theta \in \mathbb{R}^{|\mathcal{V}|}$, and $\mathbf{U}$ is obtained from the eigendecomposition of the normalized Laplacian matrix: $\mathbf{L} = \mathbf{I}_{|\mathcal{V}|} - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} = \mathbf{U}\Lambda\mathbf{U}^\top$, where $\mathbf{D}_{ii} = \sum_j \mathbf{A}_{ij}$ and $\mathbf{U}^\top\mathbf{x}$ is the graph Fourier transform. [18] and [44] proposed related convolutional networks based on $\mathbf{L}$, but both of them theoretically require a symmetric adjacency matrix to have a complete set of eigenvectors, which is not usually satisfied for directed graphs, whose edges often have asymmetric sending and receiving patterns. [77] and [76] proposed symmetric matrices to incorporate the direction. [88] uses a complex-valued Laplacian matrix, encoding direction in the phase matrix and weights in the magnitude matrix.

3 Tasks and Evaluation Metrics

In this section, we review the typical problem setup, discuss typical tasks and loss functions as well as evaluation metrics, for signed and directed graphs. Typical machine learning tasks in signed/directed graphs include: 1) link prediction, 2) node classification, 3) node clustering. These tasks can be carried out in a supervised (a large proportion of the data are training data with label supervision), semi-supervised (label supervision on a small proportion of data), or unsupervised (no label supervision) setting. Here we do not include tasks relating to groups of graphs or to synthetic graph generation.

3.1 Link Prediction

For *signed* networks, a typical task is to predict the sign of an existing edge, i.e., link sign prediction (SP). One typical GNN pipeline first learns node embeddings based on some loss function dependent on, for example, node embeddings and edge signs [19, 41], and then applies a binary classifier such as the logistic regression classifier to output the final predictions. Another GNN pipeline conducts end-to-end training using a differentiable loss function such as the binary cross entropy loss [34].

For *directed* networks, the three typical link prediction tasks are: 1) Direction prediction (DP): predict the edge direction of pairs of vertices v_i, v_j for which either $(v_i, v_j) \in \mathcal{E}$ or $(v_j, v_i) \in \mathcal{E}$. 2) Existence prediction: predict if $(v_i, v_j) \in \mathcal{E}$ by considering ordered pairs of vertices (v_i, v_j). 3) Three-class classification (3C): classify an edge $(v_i, v_j) \in \mathcal{E}$, $(v_j, v_i) \in \mathcal{E}$, or $(v_i, v_j), (v_j, v_i) \notin \mathcal{E}$.

The link prediction tasks mentioned above could be conducted on signed and directed networks, but the task focus is on either link sign or directionality but not both. In this library, we also provide more general tasks with edge splitting tools related to both link signs and directionality: 4) Four-class classification (4C): classify an edge $(v_i, v_j) \in \mathcal{E}^+$, $(v_j, v_i) \in \mathcal{E}^+$, $(v_i, v_j) \in \mathcal{E}^-$, or $(v_j, v_i) \in \mathcal{E}^-$; 5) Five-class classification (5C): in addition to the classes in 4), add a class $(v_i, v_j), (v_j, v_i) \notin \mathcal{E}$.

For splitting edges into training/test sets, for training and evaluation we discard edges of the input network that fall into more than one classification category, but we keep these edges for the GNN during training. When treated as a classification problem, the cross-entropy loss function can be employed [88]. Results are typically evaluated with accuracy. The Area Under the Receiver Operating Characteristic Curve (AUC) [9] and the F1 score [72] are also employed in binary classification scenarios.

3.2 Node Classification

Node classification classifies each node in a graph into one of K classes. The cross-entropy loss function is usually applied for training, and performance is often evaluated via accuracy. The training setup is usually either fully-supervised, with a high proportion of training nodes, or semi-supervised, where only a small proportion of nodes in each class are provided with label information during training.

3.3 Node Clustering

A clustering into K clusters is a partition of the node set into disjoint sets $\mathcal{V} = \mathcal{C}_0 \cup \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_{K-1}$. Intuitively, nodes within a cluster should be similar to each other, while nodes across clusters should be dissimilar. In a semi-supervised setting, for each of the K clusters, a fraction of training nodes are selected as seed nodes, for which the cluster membership labels are known before training. The set of seed nodes is denoted as $\mathcal{V}^{\text{seed}} \subseteq \mathcal{V}^{\text{train}} \subset \mathcal{V}$, where $\mathcal{V}^{\text{train}}$ is the set of all training nodes. For this task, the goal is to use the embedding for assigning each node $v_i \in \mathcal{V}$ to a cluster containing known seed nodes. When no seeds are given, we are in a self-supervised setting, where only the number of clusters, K, is given. The quality of a clustering partition is often assessed through a modularity objective function [78] comparing the partition to that expected under a null model for the network, with the assumption that nodes within a cluster are relatively more densely connected than nodes across clusters. However, depending on the task at hand, *similarity* could have different meanings [32]. In a signed network with positive and negative edges, similarity may relate to the neighorhood of a node such as the proportion of shared friends or enemies. In a directed network, nodes could also be clustered with regard to their position within a directed flow on the network, see [35].

A GNN's objective plays an essential role in guiding the GNN to learn. In a node clustering task, when seed nodes with known cluster labels are available, the cross entropy loss function is usually applied. In [36], a contrastive loss function based on triplets of the nodes is used to push embeddings of nodes within clusters to be closer to each other than to nodes in other clusters. For unsupervised tasks, however, objectives which are independent of known labels are required. Differentiable objectives can then be defined based on the tasks at hand. A probabilistic version of the balanced normalized cut [13] loss is proposed in [36], while a probabilistic version of flow imbalance loss is introduced in [35].

For a node clustering problem, as cluster indices could be permuted, accuracy would not be an appropriate evaluation measure. Instead, the Adjusted Rand Index (ARI) [42] is often chosen, which is invariant to label permutation. Depending on the downstream task, other measures could be used, such as the *unhappy ratio* by [36], and the *imbalance scores* by [35].

4 Data Sets

Synthetic Data. PyGSD provides synthetic data generators for Signed Stochastic Block Models(SSBMs) and Polarized SSBMs from [36], Directed Stochastic Block Models (DSBMs) from [35], as well as Signed Directed Stochastic Block Models (SDSBMs) from [34], which are typical synthetic signed and directed networks used in related research papers. Details of these synthetic models can be found in Appendix C.

Table 1: Summary statistics for the real-world networks that can be loaded by *PyTorch Geometric Signed Directed* (PyGSD). Here n is the number of nodes, $|\mathcal{E}^+|$ and $|\mathcal{E}^-|$ denote the number of positive and negative edges, respectively. For an unsigned network, $|\mathcal{E}^-| = 0$. For an unweighted graph $|\mathcal{E}^-| = 0$ means that the set of edge weights only contains one value if we disregard signs. "Labeled" denotes whether node labels are provided. Note that for labeled data sets, all nodes are labeled. We can conduct link prediction and node clustering tasks on all data sets, while node classification is only possible on labeled data sets. In the last two columns we report whether the network is directed or weighted. Note that statistics for Fin-YNet, FiLL-pvCLCL, FiLL-OPCL, and Lead-Lag are averaged over 21, 21, 21 and 19 financial networks, respectively.

Data set	n	$\lvert\mathcal{E}^+\rvert$	$\lvert\mathcal{E}^-\rvert$	Labeled	Directed	Weighted
Sampson	25	148	182	✔	✘	✔
Cornell	183	298	0	✔	✔	✘
Texas	183	325	0	✔	✔	✘
Telegram	245	8,912	0	✔	✔	✔
Wisconsin	251	515	0	✔	✔	✘
Lead-Lag	269	29,159	0	✘	✔	✔
Rainfall	306	64,408	29,228	✘	✘	✔
FiLL-OPCL	430	84,467	100,013	✘	✔	✔
FiLL-pvCLCL	444	84,677	112,015	✘	✔	✔
Fin-YNet	451	148,527	54,313	✘	✘	✔
S&P 1500	1,193	1,069,319	353,930	✔	✘	✔
Blog	1,222	19,024	0	✘	✔	✔
Chameleon	2,277	36,101	0	✔	✔	✘
Cora-ML	2,995	8,416	0	✔	✔	✘
PPI	3,058	7,996	3,864	✘	✘	✔
Migration	3,075	721,432	0	✘	✔	✔
CiteSeer	3,312	4,715	0	✔	✔	✘
BitCoin-Alpha	3,783	22,650	1,536	✘	✔	✔
Squirrel	5,201	222,134	0	✔	✔	✔
BitCoin-OTC	5,881	32,029	3,563	✘	✔	✔
Wiki-Rfa	7,634	135,753	37,579	✘	✔	✔
WikiCS	11,701	297,110	0	✔	✔	✘
Slashdot	82,140	380,933	119,548	✘	✔	✘
Epinions	131,580	589,888	121,322	✘	✔	✘
WikiTalk	2,388,953	5,018,445	0	✘	✔	✘

Real-world Data Sets. Table 1 summarizes real-world data sets for which PyGSD provides data loaders. These are real-world data sets used in related research papers, with network size ranging from 25 to 2,388,953 nodes. Details of these data sets are given in Appendix D.

5 Methods Currently in PyGSD

In this section, we review GNNs which are currently implemented in PyGSD and which will be used in our experiments in Sec. 8. GNNs can be broadly classified into spatial or spectral, depending on whether or not they utilize the spectral theory outlined in Sec. 2.3. Table 2 lists whether the implemented GNNs can deal with signed edges or directed edges, together with the tasks they are concerned with, and whether the method is spatial or spectral. Note that non-GNN baselines are not implemented in our library, but we refer readers to [36] and [35] for their implementations.

Directed Unsigned Networks. Research on directed network analysis has been mainly spectral in nature, and centered around symmetrization-based techniques [52, 61, 70], but edge directionality itself can contain important information [16, 55]. Imbalanced flows in directed networks have been uncovered via Hermitian clustering [16] and motif-based techniques [79]. Recently, researchers have started applying GNNs to extract essential information from directed edges.

DGCN [77] uses first and second order proximity, and constructs three Laplacians, but can be space and speed-inefficient. DiGCN [76] simplifies DGCN, builds a directed Laplacian based on PageRank, and aggregates information dependent on higher-order proximity. We denote the variant with the so-called “inception blocks" as DiGCNIB, and the variant without “inception blocks" as DiGCN. MagNet [88] constructs a Hermitian matrix that encodes undirected geometric structure in the magnitude of its entries, and directional information in their phase. The Laplacian matrices proposed by [76, 77] and [88] are based on the spectral theory sketched in Sec. 2.3. DiGCL [75] introduces a directed network data augmentation method called *Laplacian perturbation* and conducts directed network contrastive learning. DIGRAC [35] conducts directed network clustering based on flow imbalance measures, with novel imbalance objectives and evaluation metrics.

Signed Undirected and Signed Directed Networks. Within the last decade, the landscape of signed network analysis has been dominated by non-GNN methods (in particular spectral methods), such as those based on the (potentially normalized) Signed Laplacian matrix [46] and its variations [57, 91], Balanced Normalized Cut and Balanced Ratio Cut [13], and many others [15, 83, 87].

Table 2: Signed/directed GNNs implemented in PyGSD

Model	Signed Edges	Directed Edges	Node-Level Tasks	Edge-Level Tasks	Spatial or Spectral
DGCN [77]	✘	✔	✔	✔	Spectral
DiGCN [76]	✘	✔	✔	✔	Spectral
DiGCNIB [76]	✘	✔	✔	✔	Spectral
MagNet [88]	✘	✔	✔	✔	Spectral
DiGCL [75]	✘	✔	✔	✘	Spectral
DIGRAC [35]	✘	✔	✔	✘	Spatial
SGCN [19]	✔	✘	✘	✔	Spatial
SiGAT [40]	✔	✔	✘	✔	Spatial
SNEA [49]	✔	✔	✘	✔	Spatial
SDGNN [41]	✔	✔	✘	✔	Spatial
SSSNET [36]	✔	✔	✔	✘	Spatial
MSGNN [34]	✔	✔	✔	✔	Spectral

More recently, GNNs for signed networks emerged. SGCN [19] proposes an information aggregation and propagation mechanism with the mean-pooling strategy for signed undirected networks based on social balance theory [31]. SiGAT [40] utilizes graph attention network GAT [81] in embedding learning for signed directed networks and devises a motif-based GNN architecture based on balance theory and status theory. SNEA [49] proposes another graph attentional layer, which uses a masked self-attention mechanism, and designs an objective function for both framework optimization and node representation learning. SDGNN [41], though also using graph attention, is more efficient than SiGAT, and its objective functions can model not only the edge sign, but in addition other vital features such as edge directions and triangles. SSSNET [36] conducts the semi-supervised node clustering task in signed networks, with a novel aggregation scheme that is not based on the popular social balance theory [31] that many previous signed GNNs rely on. The very recent MSGNN [34] introduces a novel magnetic signed Laplacian as a natural generalization of both the signed Laplacian on signed graphs and the magnetic Laplacian on directed graphs, and proposes a spectral GNN via this magnetic Laplacian matrix, extending the ideas from Sec. 2.3 to signed directed networks.

6 Framework Design of PyGSD

6.1 Neural Network Layers and Methods

PyGSD is built on the existing high-level neural network layer classes from the PyTorch and PyG ecosystems. We define neural network layers for signed and directed networks based on more

than 10 architectures. The constructors of these layers use type hinting to enable the setup of the hyperparameters. Users can build upon these layers and construct their own model, depending on the task at hand. Besides, our various auxiliary layers could be helpful in building new layers.

Building upon the layers, we also implement the full methods from various research papers, so that end-users can directly utilize the full models. For example, users can either employ the *MagNetConv* layer and construct their own full architecture, or they can directly call *MagNet_node_classification* or *MagNet_link_prediction* if they are interested in applying MagNet [88] to the node classification or the link prediction task, respectively. A framework diagram is provided in Fig. 3 with a logo in Fig. 2.

6.2 Data Structures

6.2.1 Network Generators and Data Classes

PyGSD includes synthetic network generators for models described in Sec. 4, i.e., SSBMs and Pol-SSBMs from [36], DSBMs from [35], as well as SDSBMs from [34]. To accommodate the distinct features of signed and directed networks, based on the *Data* class from PyG [21], our library introduces two data classes for signed and directed networks, respectively. These classes (called *SignedData* and *DirectedData*, respectively) provide properties for checking whether a network is signed (or directed, respectively), and can be initialized to obtain custom data objects with either *edge_index* only, *edge_index* together with *edge_weight*, or the sparse adjacency matrix. Such a data object can also inherit attributes from another data object, where edge attributes are stored in COO format.

We also provide class functions to construct node features based on edge information. For instance, we can set the node feature matrix as the stacked leading eigenvectors of the regularized adjacency matrix for a signed network as in [36], or as the stacked real and imaginary parts of top eigenvectors for a Hermitian matrix [16] constructed from the directed network adjacency matrix as in [35].

6.2.2 Data Loaders and Splitters

Our library provides easy-to-use data loaders for real-world data sets for signed and directed networks. The loaded data set is then transformed into a custom data object designed in this library. To facilitate downstream tasks, we also provide splitters on the data objects, with only the training set required to be nonempty. Our node splitter generates masks for training, validation, test and seed nodes, where seed nodes are a portion of the training set; such masks can be useful in e.g., the semi-supervised setting of [36]. Our link split utility function divides edges into training, validation and test groups, depending on the specific downstream task. The link splitter could be applied to any task mentioned in Sec. 3.1, and includes an option to keep all edges in the minimum spanning tree in the training set to maintain connectivity for message passing. It also produces more general tools for tasks related to both signs and directionality. The splitters can either be called separately or as a data class function.

6.3 Task-Specific Evaluations and Utilities

Tasks on signed and directed networks may have custom loss functions and evaluation metrics, such as the probabilistic normalized cut loss and the unhappy ratio introduced in [36], as well as the probabilistic imbalance objectives introduced in [35]. Therefore, PyGSD implements these custom loss functions and evaluation metrics to enable researchers to utilize these functionalities easily. Our library also provides some utility functions tailored to our implemented research papers, so that end-users are able to reproduce the whole process of a paper including data preprocessing, training, and testing.

6.4 Maintaining the Library

We maintain our library via open-source code, public releases, automatically updated documentation, code examples, continuous integration, and unit tests with almost *100%* test coverage. Since its first release in February 2022, the *GitHub* repository has to date attracted 100+ stars.

Open-Source library and Public Releases. The source code of our library is publicly available on *GitHub* under the MIT license. The code repository provides contributing guidelines, issue templates and test instructions, which enables the public to contribute to the library and report any problems. The public releases of the library are also made available on the *Python Package Index (PyPI)*. Installation is thus possible via the *pip* command using the terminal.

Documentation. We release our software with publicly available documentation at `https://pytorch-geometric-signed-directed.readthedocs.io/`, which is automatically built and updated as we modify the main branch of the *GitHub* repository. Our documentation covers signed and

directed GNN layers and the full methods in various research papers, data classes specifically designed for signed and directed networks, synthetic data generators, data loaders for real-world data sets, node-level and edge-level splitters, as well as task-specific evaluation metrics, loss and utility functions. The documentation also includes an in-depth installation guide and a tour of external resources.

Code Examples. In our *GitHub* repository, we present code examples for all papers whose methods have been implemented in our library. Our documentation also introduces examples on data structures, data loaders, and splitters. Some case study examples are provided in Appendix B.

Continuous Integration. We use the freely available *GitHub Actions* to provide continuous integration for our library. When the code is updated on the main branch of the *GitHub* repository or there is a pull request to the main branch, the build process is triggered and the library is deployed on *Linux*, *Windows*, and *macOS* virtual machines, for Python versions 3.7, 3.8 and 3.9.

Unit Tests and Code Coverage. We provide unit tests for all signed and directed graph neural network layers, task-specific loss functions, evaluation metrics, utility functions, data classes and data loaders. These unit tests can either be executed locally using the source code, or executed through *GitHub Actions* when the continuous integration process is triggered. When unit tests are running a coverage report by *CodeCov* is generated. We maintain a high test coverage rate of almost 100%.

7 Comparison with Existing Software

Similar to existing deep learning software on graphs, which are usually based on machine learning frameworks such as TensorFlow (TF) [1], PyTorch (PT) [62], MxNet (MX) [11] and JAX [8], PyGSD is built on the PyTorch ecosystem. We summarize the characteristics of related deep learning libraries on graphs in Table 3, comparing frameworks based on the backend, the purpose, whether to contain synthetic data generators, whether to contain dedicated data splitters like our edge or node splitters, and whether to contain dedicated objective functions and evaluation metrics. All libraries listed except OpenNE [30] provide GPU support. Compared to existing libraries, PyGSD contains various signed GNNs, while PyG [21] contains only one signed GNN implementation for SGCN [19], and the other comparison libraries do not contain signed GNNs. To the best of our knowledge, the proposed software is the *first* deep learning library which consists of GNN methods specifically designed for general signed and directed networks with GPU acceleration. Our directed network GNNs are not simple extension to existing GNNs that could be applied to directed networks. In contrast, the directed network method [74] contained in CogDL [10] requires the directed network to be acyclic, which is not generally satisfied. StellarGraph [17] has a GraphSAGE [29] version for directed networks, which is an extension to the original GraphSAGE where in-node and out-node neighborhoods are separately sampled and have different weights.

While based on PyG, PyGSD contains many new ideas. In addition to signed and directed GNNs PyGSD introduces novel data classes, data loaders, as well as data splitters which are specific to tasks in signed and directed networks. PyGSD also includes objective functions and evaluation metrics that are designed for signed and directed graphs, which are not included in PyG. Also our novel synthetic data generators for signed and directed networks are not available in PyG.

Table 3: Comparison of related open-source libraries for deep learning on graphs.

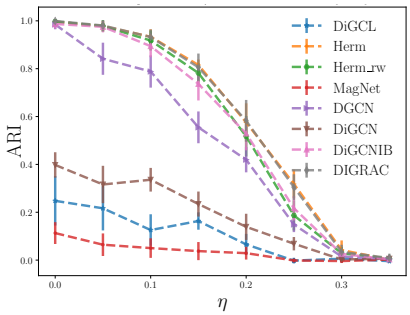
Library	Backend	Purpose	Synthetic Data Generators	Dedicated Data Splitters	Dedicated Objectives/Evaluation Metrics
OpenNE [30]	Custom	General Network Embedding	✘	✘	✘
CogDL [10]	PT	General GNN Tools	✘	✘	✘
Spektral [26]	TF	General GNN Tools	✘	✘	✘
TF Geometric [39]	TF	General GNN Tools & Methods	✘	✘	✘
StellarGraph [17]	TF	General Graph Algorithms	✘	✔	✘
DGL [90]	TF/PT/MX	General GNN Tools & Methods	✘	✘	✘
DIG [50]	PT	General GNN Tools & Methods	✘	✘	✘
Jraph [25]	JAX	General GNN Tools & Methods	✘	✘	✘
Graph-Learn [93]	TF/PT	Large-Scale GNN Tools & Methods	✘	✘	✔
PyG Temporal [68]	PT	Temporal GNN Tools & Methods	✘	✔	✘
PyG [21]	PT	General GNN Tools & Methods	✘	✘	✘
Our Work	PT	Signed/ Directed GNN Tools & Methods	✔	✔	✔

There are related open-source Python libraries for signed/directed networks that do not contain GNN methods. Python-igraph [14] contains network analysis tools; NetworkX [28] is a package for complex networks; NetworkKit [73] is a open-source toolkit for large-scale network analysis; SigNet[1] [15] contains spectral analysis methods on signed networks; CDLIB [66] allows to extract, compare and evaluate communities from complex networks; EvalNE [54] is designed for assessing and comparing the performance of network embedding methods on various downstream tasks.

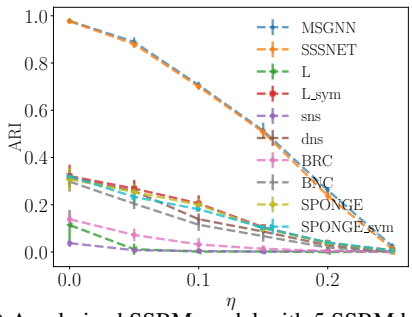
[1] https://github.com/alan-turing-institute/signet

8 Experimental Evaluation

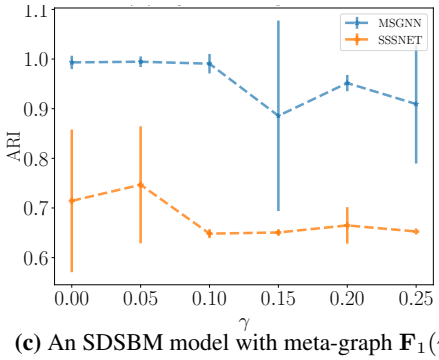
To demonstrate that the implemented methods can reproduce the performance in the original papers we evaluate the task performance of all GNNs implemented in the library. We report the mean and one standard deviation for each result, where ± 0.0 indicates that the standard deviation is less than 0.05%. Implementation details and average runtime are provided in Appendix E. Our implementations generally reproduce the results in the original papers, but slight differences (negligible considering standard deviations) may occur due to data splits or random seeds. The runtime reports also validate our implementation efficiency, showing that our implementations consume comparable or shorter runtime compared to the original implementations. In addition, we provide insights into which methods are preferred for various tasks.



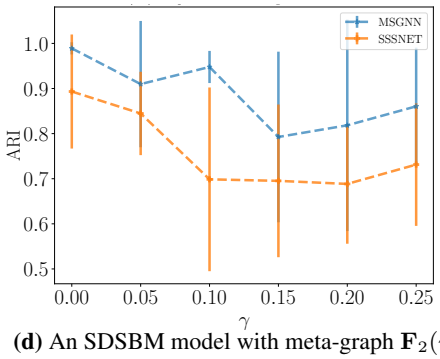
(a) A DSBM model with 3 clusters and edge density 0.02.



(b) A polarized SSBM model with 5 SSBM blocks.



(c) An SDSBM model with meta-graph $\mathbf{F}_1(\gamma)$.



(d) An SDSBM model with meta-graph $\mathbf{F}_2(\gamma)$.

Figure 1: Test ARI results averaged over 10 runs. Error bars indicate one standard deviation.

Node Clustering. Figure 1a reproduces test ARI for the directed unsigned GNNs together with two Hermitian variants mentioned in Sec. 5 on a DSBM("cyclic", F, $n = 1000, K = 3, p = 0.02, \rho = 1.5$) model for three loss variants. Figure 1b illustrates test ARI for SSSNET [36], MSGNN [34] and some spectral methods mentioned in Sec. 5 on a polarized SSBM model POL-SSBM($n = 5000, r = 5, p = 0.1, \rho = 1.5$), which is similar to the performance in the original paper. Figure 1c and Figure 1d reproduce results from [34] on SSSNET and MSGNN for SDSBM ($\mathbf{F}_1(\gamma), n = 1000, p = 0.1, \rho = 1.5, \eta = 0.1$) and SDSBM ($\mathbf{F}_2(\gamma), n = 1000, p = 0.1, \rho = 1.5, \eta = 0.1$) models, respectively, with varying γ where $\mathbf{F}_1$ and $\mathbf{F}_2$ are given in Appendix E. These methods are all efficient in runtime, and are recommended for relative network clustering tasks.

Table 4: Link sign prediction results (AUC, Macro-F1) for signed networks (%).

Method	Bitcoin-Alpha	Bitcoin-OTC	Wikirfa	Slashdot	Epinions
SGCN	(87.5±0.9, 68.9±0.9)	(88.8±0.7, 74.1±1.4)	(84.7±0.4, 71.7±0.6)	(79.8±0.2, 68.2±0.4)	(87.5±0.4, 79.6±0.2)
SiGAT	(86.5±2.4, 68.6±2.2)	(88.9±0.5, 73.8±0.8)	(87.6±0.3, 75.3±0.6)	(87.4±0.3, 75.5±0.4)	(92.4±0.6, 83.6±0.3)
SNEA	(**91.0±1.1**, 73.3±1.6)	(**91.6±0.4**, 78.6±0.6)	(87.8±1.1, 74.8±1.6)	(**90.5±0.2**, **79.1±0.2**)	(**94.8±0.1**, 86.3±0.2)
SDGNN	(88.3±1.4, **73.7±1.0**)	(90.8±0.8, **79.6±0.4**)	(**89.5±0.1**, **78.3±0.2**)	(89.4±0.3, 78.8±0.4)	(94.3±0.2, **86.6±0.2**)

Link Sign Prediction on Signed Graphs. We use the five real-world data sets to do link sign prediction on SGCN, SiGAT, SNEA, and SDGNN. Table 4 reproduces link sign prediction performance for these four methods on the five real-world data sets. Considering also the runtime comparison given in Table 7, as well as the data set statistics provided in Table 1, we recommend employing SGCN on smaller networks and SiGAT on networks with larger scales, if speed is the main concern. If performance is the main concern, then we would choose SNEA concerning AUC, while we would pick SDGNN regarding Macro-F1.

Table 5: Evaluation results for directed unsigned graphs (%).

Task	Method	Cornell	Texas	Wisconsin	CoraML	CiteSeer	Telegram
Node classification	DGCN	62.2±7.8	72.2±5.7	66.7±5.4	**80.1±2.1**	**65.5±1.4**	89.4±3.2
	DiGCN	52.4±7.3	64.9±5.1	61.6±3.3	79.7±1.1	63.0±2.7	70.2±2.9
	DiGCNIB	46.2±6.0	56.2±4.6	56.3±5.0	81.2±1.8	63.8±2.3	65.4±5.1
	MagNet	**70.3±5.1**	**82.4±4.9**	**82.4±4.4**	70.0±1.5	60.4±1.4	**91.3±4.1**
	DiGCL	33.5±7.6	55.1±9.7	44.3±5.1	74.7±2.0	55.2±3.2	77.9±3.7
Direction prediction	DGCN	82.6±6.1	79.8±7.3	84.8±4.7	85.0±0.8	84.4±0.9	96.4±0.6
	DiGCN	75.2±8.9	78.3±12.5	81.1±6.5	79.0±1.8	77.6±2.6	79.1±1.9
	DiGCNIB	**86.1±5.4**	**80.6±8.2**	**85.0±5.2**	86.0±0.5	**85.6±0.7**	96.5±0.5
	MagNet	79.8±9.4	78.9±8.8	84.1±4.7	**86.8±0.7**	84.6±0.7	**97.4±0.5**
Existence link prediction	DGCN	61.6±4.8	59.3±5.0	66.3±5.4	76.8±2.0	65.6±2.3	86.3±0.9
	DiGCN	59.9±5.4	60.8±7.3	60.1±4.7	74.5±1.0	68.5±1.3	73.9±2.0
	DiGCNIB	66.7±3.5	**68.9±3.9**	**71.0±4.1**	**77.8±0.6**	**73.0±1.2**	84.2±1.1
	MagNet	**66.9±4.2**	59.9±7.4	68.4±4.2	77.0±0.9	65.9±2.3	**86.9±0.6**
Three classes link prediction	DGCN	**60.6±6.8**	61.5±8.6	**69.2±4.6**	69.3±1.5	62.2±3.9	81.6±0.7
	DiGCN	51.8±6.7	56.5±7.2	49.6±4.2	65.0±1.4	54.0±1.4	63.6±5.4
	DiGCNIB	**60.6±5.2**	**67.2±5.9**	66.8±5.8	70.3±0.8	**68.5±1.1**	79.9±0.6
	MagNet	56.0±10.4	63.2±6.6	67.5±4.1	**70.5±1.1**	58.9±3.3	**83.3±0.6**

Node Classification and Edge-Level Tasks on Directed Unsigned Graphs. Table 5 present results on node classification and three link prediction tasks for directed unsigned graphs, for five directed graph methods (DGCN, DiGCN, DiGCNIB, MagNet, and DiGCL), where DiGCNIB is the "inception block" version of DiGCN introduced in [76]. Given also the runtime comparison in Table 8, and the data set statistics detailed in Table 1, we suggest using DiGCN, when speed is the main concern. If task performance is more important, then we would recommend DiGCNIB as the method of choice for data sets where directionality is largely emphasized (such as the data sets here except CoraML and CiteSeer). On CoraML and CiteSeer, the best-performing method varies case by case.

Table 6: Link prediction test accuracy (%) comparison for directions (and signs). The link prediction tasks are introduced in Sec. 3.1.

Data Set	Link Task	GCN	SGCN	SDGNN	SiGAT	SNEA	SSSNET	MSGNN
BitCoin-Alpha	SP	58.0±2.0	64.7±0.9	64.5±1.1	62.9±0.9	64.1±1.3	67.4±1.1	**71.3±1.2**
	DP	59.6±1.3	60.4±1.7	61.5±1.0	61.9±1.9	60.9±1.7	68.1±2.3	**72.5±1.5**
	3C	82.0±0.4	81.4±0.5	79.2±0.9	77.1±0.7	83.2±0.5	78.3±4.7	**84.4±0.6**
	4C	42.4±2.2	51.1±0.8	52.5±1.1	49.3±0.7	52.4±1.8	54.3±2.9	**58.5±0.7**
	5C	80.0±0.4	79.5±0.3	78.2±0.5	76.5±0.3	81.1±0.3	77.9±0.3	**81.9±0.9**
BitCoin-OTC	SP	55.3±1.2	65.6±0.9	65.3±1.2	62.8±1.3	67.7±0.5	70.1±1.2	**73.0±1.4**
	DP	55.3±2.3	63.8±1.2	63.2±1.5	64.0±2.0	65.3±1.2	69.6±1.0	**71.8±1.1**
	3C	78.8±0.8	79.0±0.7	77.3±0.7	73.6±0.7	82.2±0.4	76.9±1.1	**83.3±0.7**
	4C	42.2±1.2	51.5±0.4	55.3±0.8	51.2±1.8	56.9±0.7	57.0±2.0	**59.8±0.7**
	5C	75.8±0.5	77.4±0.7	77.3±0.8	74.1±0.5	80.5±0.5	74.0±1.6	**80.9±0.9**
Slashdot	SP	78.3±1.1	74.7±0.5	74.1±0.7	64.0±1.3	70.6±1.0	86.6±2.2	**92.4±0.2**
	DP	79.1±0.2	74.8±0.9	74.2±1.4	62.8±0.9	71.1±1.1	87.8±1.0	**93.1±0.1**
	3C	74.7±1.0	69.7±0.3	66.3±1.8	49.1±1.2	72.5±0.7	79.3±1.2	**86.1±0.3**
	4C	60.0±1.1	63.2±0.3	64.0±0.7	53.4±0.2	60.5±0.6	72.7±0.6	**78.2±0.3**
	5C	65.3±0.2	64.4±0.3	62.6±2.0	44.4±1.4	66.4±0.5	70.4±0.7	**76.8±0.6**
Epinions	SP	68.6±1.2	62.9±0.5	67.7±0.8	63.6±0.5	66.5±1.0	78.5±2.1	**85.4±0.5**
	DP	67.6±1.0	61.7±0.5	67.9±0.6	63.6±0.8	66.4±1.2	73.9±6.2	**86.3±0.3**
	3C	73.6±1.1	70.3±0.8	73.2±0.8	52.3±1.3	72.8±0.2	72.7±2.0	**83.1±0.5**
	4C	61.3±1.2	66.7±1.2	71.0±0.6	62.3±0.5	69.5±0.7	70.2±5.2	**78.7±0.9**
	5C	70.0±0.7	73.5±0.8	76.6±0.7	52.9±0.7	74.2±0.1	70.3±4.6	**80.5±0.5**
FiLL (avg.)	SP	87.7±0.0	88.4±0.0	82.0±0.3	76.9±0.1	90.0±0.0	88.7±0.3	**90.8±0.0**
	DP	87.7±0.0	88.5±0.1	82.0±0.2	76.9±0.1	90.0±0.0	88.8±0.3	**90.9±0.0**
	3C	61.7±0.1	63.0±0.1	59.3±0.0	55.3±0.1	64.3±0.1	62.2±0.3	**66.1±0.1**
	4C	71.6±0.1	81.7±0.0	78.8±0.1	70.5±0.1	83.2±0.1	80.0±0.3	**83.3±0.0**
	5C	54.6±0.1	63.8±0.0	61.1±0.1	55.5±0.1	**64.8±0.1**	60.4±0.4	**64.8±0.1**

Link Prediction on Signed Directed Graphs. Table 6 reproduces the results on five link prediction tasks introduced in Sec. 3.1, on the six signed methods implemented in the library (SGCN, SDGNN, SiGAT, SNEA, SSSNET, and MSGNN) and an unsigned baseline GCN, on five real-world signed directed data sets. Taking in addition the runtime comparison in Table 9 into account, as well as the data set statistics given by Table 1, we recommend MSGNN as the method of choice. Note that SSSNET is also quite competitive in terms of performance while being fast for large networks. The GCN employed here replaces MSConv in MSGNN with GCNConv from PyG, and additionally inputs absolute values of edge weights for message passing to avoid getting NAN values. We conclude that signed information is indeed helpful when comparing GCN and MSGNN. GCN is generally faster but not by a large margin, while MSGNN is considerably more accurate.

9 Conclusion and Outlook

We present *PyTorch Geometric Signed Directed* (PyGSD), an easy-to-use and easy-to-adapt end-to-end software package which uses GNNs to address three main tasks in the analysis of signed and directed networks: link prediction, node classification, and node clustering. Our evaluation and discussion of the implemented methods provide insights for users on which methods to pick for various tasks. As an extension library to PyG, our proposed PyGSD is open source and comes with an invitation to contribute new methods, data sets, utilities, and case studies. In the future, we envisage contributions in the area of scalability and metrics for assessing specific tasks. Our library is related to PyTorch Geometric Temporal, another extension library to PyG developed for the analysis of temporal networks. In future works, these two software packages are likely to grow closer together.

Author Contributions

Y.H.: Conceptualization, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Software, Visualization, Writing - original draft; X.Z.: Data curation, Software, Writing - review & editing; J.H: Data curation, Software, Writing - review & editing; B.R.: Writing - review & editing; M.C.: Conceptualization, Data curation, Formal analysis, Funding acquisition, Methodology, Project administration, Resources, Supervision, Visualization, Writing - review & editing; G.R.: Conceptualization, Formal analysis, Funding acquisition, Methodology, Project administration, Supervision, Validation, Writing - review & editing.

Acknowledgements

Y.H. is supported by a Clarendon scholarship. G.R. is supported in part by EPSRC grants EP/T018445/1, EP/W037211/1 and EP/R018472/1. M.C. acknowledges support from the EPSRC grants EP/N510129/1 and EP/W037211/1 at The Alan Turing Institute.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016. 2, 7

[2] Lada A Adamic and Natalie Glance. The political blogosphere and the 2004 US election: divided they blog. In *Proceedings of the 3rd International Workshop on Link Discovery*, pages 36–43, 2005. 22

[3] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi, and Teh Ying Wah. Time-series clustering–a decade review. *Information Systems*, 53:16–38, 2015. 1

[4] Stefanos Bennett, Mihai Cucuringu, and Gesine Reinert. Detection and clustering of lead-lag networks for multivariate time series with an application to financial markets. *7th SIGKDD Workshop on Mining and Learning from Time Series (MiLeTS)*, 2021. 2, 22

[5] Michael Bertolacci, Edward Cripps, Ori Rosen, John W Lau, Sally Cripps, et al. Climate inference on daily rainfall across the australian continent, 1876–2015. *Annals of Applied Statistics*, 13(2):683–712, 2019. 23

[6] Aleksandar Bojchevski and Stephan Günnemann. Deep Gaussian embedding of graphs: Unsupervised inductive learning via ranking. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2017. 22

[7] Alexandre Bovet and Peter Grindrod. The Activity of the Far Right on Telegram. https://www.researchgate.net/publication/346968575_The_Activity_of_the_Far_Right_on_Telegram_v21, 2020. 2, 22

[8] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, et al. Jax: composable transformations of python+ numpy programs. 2018. 2, 7

[9] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern Recognition*, 30(7):1145–1159, 1997. 3

[10] Yukuo Cen, Zhenyu Hou, Yan Wang, Qibin Chen, Yizhen Luo, Zhongming Yu, Hengrui Zhang, Xingcheng Yao, Aohan Zeng, Shiguang Guo, Yuxiao Dong, Yang Yang, Peng Zhang, Guohao Dai, Yu Wang, Chang Zhou, Hongxia Yang, and Jie Tang. CogDL: A comprehensive library for graph deep learning. In *Proceedings of the ACM Web Conference 2023 (WWW'23)*, 2023. 7

[11] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015. 2, 7

[12] Yiqi Chen, Tieyun Qian, Huan Liu, and Ke Sun. " bridge" enhanced signed directed network embedding. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 773–782, 2018. 2

[13] Kai-Yang Chiang, Joyce Whang, and Inderjit Dhillon. Scalable clustering of signed networks using balance normalized cut. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, pages 615–624, New York, United States, 2012. Association for Computing Machinery. 4, 5, 25

[14] Gabor Csardi, Tamas Nepusz, et al. The igraph software package for complex network research. *InterJournal, Complex Systems*, 1695(5):1–9, 2006. 7

[15] Mihai Cucuringu, Peter Davies, Aldo Glielmo, and Hemant Tyagi. Sponge: a generalized eigenproblem for clustering signed networks. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1088–1098. PMLR, 2019. 5, 7, 25

[16] Mihai Cucuringu, Huan Li, He Sun, and Luca Zanetti. Hermitian matrices for clustering directed graphs: insights and applications. In *International Conference on Artificial Intelligence and Statistics*, pages 983–992. PMLR, 2020. 5, 6, 22, 25

[17] CSIRO's Data61. StellarGraph machine learning library. `https://github.com/stellargraph/stellargraph`, 2018. 2, 7

[18] M. Defferrard, X. Bresson, and P. Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in Neural Information Processing Systems*, pages 3844–3852. Neural Information Processing Systems Foundation, 2016. 3

[19] Tyler Derr, Yao Ma, and Jiliang Tang. Signed graph convolutional networks. In *2018 IEEE International Conference on Data Mining (ICDM)*, pages 929–934, Singapore, 2018. IEEE, IEEE. 2, 3, 5, 7, 26

[20] Andrew Elliott, Paul Reidy Milton Martinez Luaces, Mihai Cucuringu, and Gesine Reinert. Anomaly detection in networks using spectral methods and network comparison approaches. *arXiv preprint arXiv:1901.00402*, 2019. 21

[21] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 2, 6, 7

[22] Sergio M Focardi. Clustering economic and financial time series: Exploring the existence of stable correlation conditions. *The Intertek Group*, 2005. 1

[23] C Lee Giles, Kurt D Bollacker, and Steve Lawrence. Citeseer: An automatic citation indexing system. In *Proceedings of the third ACM conference on Digital libraries*, pages 89–98, 1998. 22

[24] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International Conference on Machine Learning*, pages 1263–1272. PMLR, 2017. 3

[25] Jonathan Godwin, Thomas Keck, Peter Battaglia, Victor Bapst, Thomas Kipf, Yujia Li, Kimberly Stachenfeld, Petar Velickovic, and Alvaro Sanchez-Gonzalez. Jraph: A library for graph neural networks in jax., 2020. *URL http://github. com/deepmind/jraph*, 5. 7

[26] Daniele Grattarola and Cesare Alippi. Graph neural networks in tensorflow and keras with spektral [application notes]. *IEEE Computational Intelligence Magazine*, 16(1):99–106, 2021. 2, 7

[27] Ramanthan Guha, Ravi Kumar, Prabhakar Raghavan, and Andrew Tomkins. Propagation of trust and distrust. In *Proceedings of the 13th International Conference on World Wide Web*, pages 403–412, 2004. 2

[28] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using NetworkX. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008. 7

[29] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 30, 2017. 7

[30] X Han, S Cao, X Lv, Y Lin, Z Liu, and M Sun. OpenNE: An open source toolkit for network embedding. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (System Demonstrations), Brussels, Belgium*, pages 139–144, 2018. 7

[31] Frank Harary. On the notion of balance of a signed graph. *Michigan Mathematical Journal*, 2(2):143–146, 1953. 5

[32] Yixuan He. GNNs for node clustering in signed and directed networks. In *Proceedings of the Fifteenth ACM International Conference on Web Search and Data Mining*, pages 1547–1548, 2022. 4

[33] Yixuan He, Quan Gan, David Wipf, Gesine D Reinert, Junchi Yan, and Mihai Cucuringu. GNNRank: Learning global rankings from pairwise comparisons via directed graph neural networks. In *International Conference on Machine Learning*, pages 8581–8612. PMLR, 2022. 2

[34] Yixuan He, Michael Perlmutter, Gesine Reinert, and Mihai Cucuringu. MSGNN: a spectral graph neural network based on a novel magnetic signed Laplacian. In *Learning on Graphs Conference*, pages 40–1. PMLR, 2022. 3, 4, 5, 6, 8, 21, 24, 26

[35] Yixuan He, Gesine Reinert, and Mihai Cucuringu. DIGRAC: digraph clustering based on flow imbalance. In *Learning on Graphs Conference*, pages 21–1. PMLR, 2022. 2, 3, 4, 5, 6, 24, 25

[36] Yixuan He, Gesine Reinert, Songchao Wang, and Mihai Cucuringu. SSSNET: Semi-supervised signed network clustering. In *Proceedings of the 2022 SIAM International Conference on Data Mining (SDM)*, pages 244–252. SIAM, 2022. 1, 3, 4, 5, 6, 8, 20, 21, 23, 24, 26

[37] Yixuan He, Gesine Reinert, David Wipf, and Mihai Cucuringu. Robust angular synchronization via directed graph neural networks. *arXiv preprint arXiv:2310.05842*, 2023. 2

[38] Fritz Heider. Attitudes and cognitive organization. *The Journal of Psychology*, 21(1):107–112, 1946. 2

[39] Jun Hu, Shengsheng Qian, Quan Fang, Youze Wang, Quan Zhao, Huaiwen Zhang, and Changsheng Xu. Efficient graph deep learning in Tensorflow with TF_Geometric. In *Proceedings of the 29th ACM International Conference on Multimedia*, pages 3775–3778, 2021. 2, 7

[40] Junjie Huang, Huawei Shen, Liang Hou, and Xueqi Cheng. Signed graph attention networks. In *International Conference on Artificial Neural Networks*, pages 566–577. Springer, 2019. 2, 3, 5, 26

[41] Junjie Huang, Huawei Shen, Liang Hou, and Xueqi Cheng. SDGNN: Learning node representation for signed directed networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 196–203, 2021. 1, 2, 3, 5, 26

[42] Lawrence Hubert and Phipps Arabie. Comparing partitions. *Journal of Classification*, 2(1):193–218, 1985. 4

[43] Diederik Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015. 19, 24

[44] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. 3

[45] Srijan Kumar, Francesca Spezzano, VS Subrahmanian, and Christos Faloutsos. Edge weight prediction in weighted signed networks. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 221–230, Barcelona, Spain, 2016. IEEE, IEEE. 23

[46] Jérôme Kunegis, Stephan Schmidt, Andreas Lommatzsch, Jürgen Lerner, Ernesto W De Luca, and Sahin Albayrak. Spectral analysis of signed graphs for clustering, prediction and visualization. In *Proceedings of the 2010 SIAM International Conference on Data Mining*, pages 559–570, Sydney, Australia, 2010. SIAM, IEEE. 5, 25

[47] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1361–1370. Association for Computing Machinery, 2010. 2, 22

[48] Rui Li, Xin Yuan, Mohsen Radfar, Peter Marendy, Wei Ni, Terence J O'Brien, and Pablo M Casillas-Espinosa. Graph signal processing, graph neural network and graph learning on biological data: a systematic review. *IEEE Reviews in Biomedical Engineering*, 2021. 1

[49] Yu Li, Yuan Tian, Jiawei Zhang, and Yi Chang. Learning signed network embedding via graph attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 4772–4779, 2020. 2, 3, 5, 26

[50] Meng Liu, Youzhi Luo, Limei Wang, Yaochen Xie, Hao Yuan, Shurui Gui, Haiyang Yu, Zhao Xu, Jingtun Zhang, Yi Liu, et al. DIG: a turnkey library for diving into graph deep learning research. *Journal of Machine Learning Research*, 22(240):1–9, 2021. 7

[51] Zhanghui Liu and Huachun Tan. Traffic prediction with graph neural network: A survey. In *CICTP 2021*, pages 467–474. 2021. 1

[52] Yi Ma, Jianye Hao, Yaodong Yang, Han Li, Junqi Jin, and Guangyong Chen. Spectral-based graph convolutional network for directed graphs. *arXiv preprint arXiv:1907.08990*, 2019. 5

[53] Fragkiskos D Malliaros and Michalis Vazirgiannis. Clustering and community detection in directed networks: A survey. *Physics Reports*, 533(4):95–142, 2013. 21

[54] Alexandru Mara, Jefrey Lijffijt, and Tijl De Bie. EvalNE: A framework for network embedding evaluation. *SoftwareX*, 17:100997, 2022. 7

[55] Antonio G Marques, Santiago Segarra, and Gonzalo Mateos. Signal processing on directed graphs: The role of edge directionality when processing and learning from network data. *IEEE Signal Processing Magazine*, 37(6):99–116, 2020. 1, 2, 5

[56] Paolo Massa and Paolo Avesani. Controversial users demand local trust metrics: An experimental study on epinions.com community. In *AAAI*, pages 121–126, 2005. 23

[57] Pedro Mercado, Francesco Tudisco, and Matthias Hein. Clustering signed networks with the geometric mean of Laplacians. *Advances in Neural Information Processing System*, 29, 2016. 5, 25

[58] Péter Mernyei and Cătălina Cangea. Wiki-cs: A wikipedia-based benchmark for graph neural networks. *arXiv preprint arXiv:2007.02901*, 2020. 22

[59] Tyler Moore and Nicolas Christin. Beware the middleman: Empirical analysis of bitcoin-exchange risk. In *Financial Cryptography and Data Security: 17th International Conference, FC 2013, Okinawa, Japan, April 1-5, 2013, Revised Selected Papers 17*, pages 25–33. Springer, 2013. 23

[60] Bruno Ordozgoiti, Antonis Matakos, and Aristides Gionis. Finding large balanced subgraphs in signed networks. In *Proceedings of The Web Conference 2020*, WWW '20, page 1378–1388, New York, NY, USA, 2020. Association for Computing Machinery. 23

[61] William R. Palmer and Tian Zheng. Spectral clustering for directed networks. In Rosa M. Benito, Chantal Cherifi, Hocine Cherifi, Esteban Moro, Luis Mateus Rocha, and Marta Sales-Pardo, editors, *Complex Networks & Their Applications IX*, pages 87–99, Cham, 2021. Springer International Publishing. 5

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019. 2, 7

[63] Nicos G Pavlidis, Vassilis P Plagianakos, Dimitris K Tasoulis, and Michael N Vrahatis. Financial forecasting through unsupervised clustering and neural networks. *Operational Research*, 6(2):103–127, 2006. 1

[64] Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. Geom-GCN: geometric graph convolutional networks. In *International Conference on Learning Representations*, 2020. 22

[65] Marc J Perry. *State-to-state migration Flows, 1995 to 2000*. US Department of Commerce, Economics and Statistics Administration, US . . . , 2003. 22

[66] Giulio Rossetti, Letizia Milli, and Rémy Cazabet. CDLIB: a python library to extract, compare and evaluate communities from complex networks. *Applied Network Science*, 4(1):1–26, 2019. 7

[67] Benedek Rozemberczki, Carl Allen, and Rik Sarkar. Multi-scale attributed node embedding. *Journal of Complex Networks*, 9(2):cnab014, 2021. 22

[68] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzmán López, Nicolas Collignon, and Rik Sarkar. PyTorch Geometric Temporal: spatiotemporal signal processing with neural machine learning models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, CIKM '21, page 4564–4573, New York, NY, USA, 2021. Association for Computing Machinery. 2, 7

[69] Samuel Franklin Sampson. *A novitiate in a period of change: An experimental and case study of social relationships*. Cornell University, Ithaca, NY 14850, USA, 1968. 1, 23

[70] Venu Satuluri and Srinivasan Parthasarathy. Symmetrizations for clustering directed graphs. In *Proceedings of the 14th International Conference on Extending Database Technology*, pages 343–354, 2011. 5

[71] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pages 593–607. Springer, 2018. 2

[72] Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Information Processing & Management*, 45(4):427–437, 2009. 3

[73] Christian L Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. NetworKit: A tool suite for large-scale complex network analysis. *Network Science*, 4(4):508–530, 2016. 7

[74] Veronika Thost and Jie Chen. Directed acyclic graph neural networks. In *International Conference on Learning Representations*, 2021. 7

[75] Zekun Tong, Yuxuan Liang, Henghui Ding, Yongxing Dai, Xinke Li, and Changhu Wang. Directed graph contrastive learning. *Advances in Neural Information Processing Systems*, 34, 2021. 5, 25

[76] Zekun Tong, Yuxuan Liang, Changsheng Sun, Xinke Li, David Rosenblum, and Andrew Lim. Digraph inception convolutional networks. *Advances in Neural Information Processing Systems*, 33:17907–17918, 2020. 3, 5, 9

[77] Zekun Tong, Yuxuan Liang, Changsheng Sun, David S Rosenblum, and Andrew Lim. Directed graph convolutional network. *arXiv preprint arXiv:2004.13970*, 2020. 3, 5

[78] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. From Louvain to Leiden: guaranteeing well-connected communities. *Scientific Reports*, 9(1):1–12, 2019. 4

[79] William George Underwood, Andrew Elliott, and Mihai Cucuringu. Motif-based spectral clustering of weighted directed networks. *Applied Network Science*, 5(62), September 2020. 5

[80] Shikhar Vashishth, Soumya Sanyal, Vikram Nitin, and Partha Talukdar. Composition-based multi-relational graph convolutional networks. In *International Conference on Learning Representations*, 2020. 2

[81] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018. 5

[82] Arunachalam Vinayagam, Jonathan Zirin, Charles Roesel, Yanhui Hu, Bahar Yilmazel, Anastasia A Samsonova, Ralph A Neumüller, Stephanie E Mohr, and Norbert Perrimon. Integrating protein-protein interaction networks with phenotypes reveals signs of interactions. *Nature methods*, 11(1):94–99, 2014. 23

[83] Suhang Wang, Charu Aggarwal, Jiliang Tang, and Huan Liu. Attributed signed network embedding. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, pages 137–146, 2017. 5

[84] Robert West, Hristo S Paskov, Jure Leskovec, and Christopher Potts. Exploiting social network structure for person-to-person sentiment analysis. *Transactions of the Association for Computational Linguistics*, 2:297–310, 2014. 23

[85] 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. 1, 3

[86] yahoo! finance. S&p 1500 data set link. `https://finance.yahoo.com/`, 2021. [Online; accessed 19-January-2021]. 23

[87] S. Yuan, X. Wu, and Y. Xiang. SNE: signed network embedding. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10235, pages 183–195. Springer Verlag, 2017. 5

[88] Xitong Zhang, Yixuan He, Nathan Brugnone, Michael Perlmutter, and Matthew Hirn. MagNet: a neural network for directed graphs. *Advances in Neural Information Processing Systems*, 34:27003–27015, 2021. 2, 3, 5, 6, 25, 26

[89] Zhao Zhang, Fuzhen Zhuang, Hengshu Zhu, Zhiping Shi, Hui Xiong, and Qing He. Relational graph neural network with hierarchical attention for knowledge graph completion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 9612–9619, 2020. 2

[90] Da Zheng, Minjie Wang, Quan Gan, Zheng Zhang, and George Karypis. Learning graph neural networks with Deep Graph Library. In *Companion Proceedings of the Web Conference 2020*, WWW '20, page 305–306, 2020. 2, 7

[91] Quan Zheng and David B Skillicorn. Spectral embedding of signed networks. In *Proceedings of the 2015 SIAM International Conference on Data Mining*, pages 55–63. SIAM, 2015. 5

[92] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. 1

[93] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. Aligraph: a comprehensive graph neural network platform. *Proceedings of the VLDB Endowment*, 12(12):2094–2105, 2019. 7

[94] Hartmut Ziegler, Marco Jenny, Tino Gruse, and Daniel A Keim. Visual market sector analysis for financial time series data. In *Visual Analytics Science and Technology (VAST), 2010 IEEE Symposium on*, pages 83–90. IEEE, 2010. 1

A Logo and Diagram

Figure 2: PyGSD Logo.

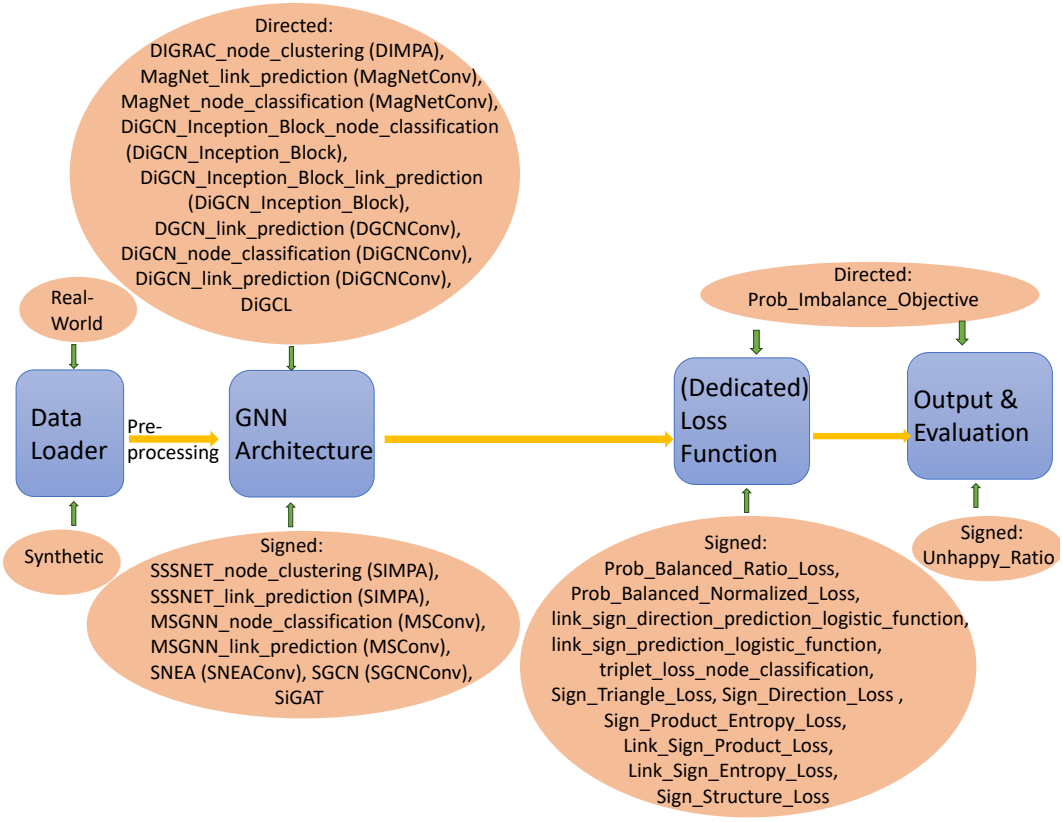


Figure 3: PyGSD framework overview: starting from our data loaders, input data is loaded into the GNN architecture (where the layers inside brackets could be treated individually to construct new architectures), then (dedicated) loss functions are employed, and finally we test the performance.

B Case Study Examples

Here we first provide a case study for clustering in signed networks, and then a case study for link prediction in directed networks.

B.1 Case Study on Signed Networks

In this subsection, we overview a simple end-to-end machine learning pipeline for signed networks designed with *PyTorch Geometric Signed Directed* (PyGSD). These three code snippets – Listings

1, 2 and 3 – solve the signed clustering problem on a Signed Stochastic Block Model of clustering the nodes in the signed network into five groups. The pipeline consists of data preparation (Listing 1), model definition, training, and evaluation phases (Listing 2). Listing 3 illustrates how to run experiments and output results. We explain the three listings in turn.

```
1 from sklearn.metrics import adjusted_rand_score
2 import scipy.sparse as sp
3 import torch
4 from torch_geometric_signed_directed.nn import \
5     SSSNET_node_clustering as SSSNET
6 from torch_geometric_signed_directed.data import \
7     SignedData, SSBM
8 from torch_geometric_signed_directed.utils import \
9     (Prob_Balanced_Normalized_Loss,
10 extract_network, triplet_loss_node_classification)
11
12 device = torch.device('cuda' if \
13 torch.cuda.is_available() else 'cpu')
14
15 num_classes = 5
16 num_nodes = 1000
17 (A_p_scipy, A_n_scipy), labels = SSBM(num_nodes, \
18 num_classes, 0.1, 0.1)
19 A = A_p_scipy - A_n_scipy
20 A, labels = extract_network(A=A, labels=labels)
21 data = SignedData(A=A, y=torch.LongTensor(labels))
22 data.set_spectral_adjacency_reg_features(num_classes)
23 data.node_split(train_size_per_class=0.8, \
24 val_size_per_class=0.1, \
25 test_size_per_class=0.1, seed_size_per_class=0.1)
26 data.separate_positive_negative()
27 data = data.to(device)
28 loss_func_ce = torch.nn.NLLLoss()
29 model = SSSNET(nfeat=data.x.shape[1], dropout=0.5,
30 hop=2, fill_value=0.5, hidden=32, nclass=num_classes).to(device)
```

Listing 1: Preparation, data loading, node splitting, as well as loss and model initialization.

B.1.1 Preparation, Data Loading and Splitting, Loss and Model Initialization

In Listing 1, as a first step, we import the essentials (lines 1-10). We then define the device to be used (lines 12-13). After that, we define default values to be used in the network generation process, generate a synthetic network and extract its largest connected component (lines 15-20); we use the SignedData class (line 21). As no node features are available initially, we use the *data.set_spectral_adjacency_reg_features()* class method to set up the node feature matrix (line 22).

We then create a train-validation-test-seed split of the node set by using the node splitting function and calculate separate positive and negative parts of the signed network to be stored inside the data object (lines 23-26). We then move the data object to the device (line 27). Finally, we initialize the cross-entropy loss function (line 28), construct the GNN model and map it to the device (lines 29-30).

```
1 def train(features, edge_index_p, edge_weight_p,
2                 edge_index_n, edge_weight_n, mask, seed_mask,
3                 loss_func_pbnc, y):
4     model.train()
5     Z, log_prob, _, prob = model(edge_index_p, edge_weight_p,
6                 edge_index_n, edge_weight_n, features)
7     loss_pbnc = loss_func_pbnc(prob[mask])
8     loss_triplet = triplet_loss_node_classification(
```

```
    y=y[seed_mask], Z=Z[seed_mask], n_sample=500, thre=0.1)
    loss_ce = loss_func_ce(log_prob[seed_mask], y[seed_mask])
    loss = 50*(loss_ce + 0.1*loss_triplet) + loss_pbnc
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    train_ari = adjusted_rand_score(y[mask].cpu(),
    (torch.argmax(prob, dim=1)).cpu()[mask])
    return loss.detach().item(), train_ari

def test(features, edge_index_p, edge_weight_p,
                edge_index_n, edge_weight_n, mask, y):
    model.eval()
    with torch.no_grad():
        _, _, _, prob = model(edge_index_p, edge_weight_p,
                edge_index_n, edge_weight_n, features)
    test_ari = adjusted_rand_score(y[mask].cpu(),
    (torch.argmax(prob, dim=1)).cpu()[mask])
    return test_ari
```

Listing 2: Defining the training and evaluation functions.

B.1.2 Defining Functions for Training and Evaluation

In Listing 2, we define the training and evaluation functions. Setting the model to be trainable (line 4), we obtain the node embedding matrix Z and cluster assignment probabilities *prob* and its logarithm *log_prob* with a forward pass of the model instance (lines 5-6). We then obtain the probabilistic balanced normalized cut loss (line 7), triplet loss (lines 8-9), and cross entropy loss (line 10) values. The weighted sum of the three losses serves as the training loss value (line 11).

We then backpropagate and update the model parameters (lines 12-14). After that, we calculate the ARI of the training nodes (lines 15-16). Finally, we return the loss value and training ARI (line 17).

For evaluation (function *test()*), we do not set the model to be trainable (lines 21-22). With a forward pass (lines 23-24), we obtain the probability assignment matrix. Taking argmax for the probabilities, we obtain test ARI (lines 25-26) and return the result (line 27).

```
for split in range(data.train_mask.shape[1]):
    optimizer = torch.optim.Adam(model.parameters(),
    lr=0.01, weight_decay=0.0005)
    train_index = data.train_mask[:, split].cpu().numpy()
    val_index = data.val_mask[:, split]
    test_index = data.test_mask[:, split]
    seed_index = data.seed_mask[:, split]
    loss_func_pbnc = Prob_Balanced_Normalized_Loss(
    A_p=sp.csr_matrix(data.A_p)[train_index][:, train_index],
    A_n=sp.csr_matrix(data.A_n)[train_index][:, train_index])
    for epoch in range(300):
        train_loss, train_ari = train(data.x,
        data.edge_index_p,
        data.edge_weight_p, data.edge_index_n,
        data.edge_weight_n, train_index,
        seed_index, loss_func_pbnc, data.y)
        Val_ari = test(data.x, data.edge_index_p,
        data.edge_weight_p, data.edge_index_n,
        data.edge_weight_n, val_index, data.y)
        print(f'Split: {split:02d}, Epoch: {epoch:03d},
        Train_Loss: {train_loss:.4f},
        Train_ARI: {train_ari:.4f},
        Val_ARI: {Val_ari:.4f}')

```

```
25     test_ari = test(data.x, data.edge_index_p,
26     data.edge_weight_p, data.edge_index_n,
27     data.edge_weight_n, test_index, data.y)
28     print(f'Split: {split:02d}, Test_ARI: {test_ari:.4f}')
29     model._reset_parameters_undirected()
```

Listing 3: Running the experiment and printing the results.

B.1.3 Running the Experiment

We run the actual experiments in Listing 3. For each of the data splits (line 1), we first initialize the Adam optimizer [43] (lines 2-3). We then obtain the data split indices (lines 4-7), initialize the self-supervised loss function (lines 8-10), and start the training process (line 11).

For each epoch, we apply the training function to obtain training loss and ARI score (lines 12-16), then evaluate with the *test()* function on validation nodes (lines 17-19). We then print the training and validation results (lines 20-23). After training, we obtain the test performance (lines 25-27) and print some logs (line 28). Finally, we reset the model parameters (line 29) and iterate to the next data split loop.

B.2 Case Study on Directed Networks

In this subsection, we overview a simple end-to-end machine learning pipeline for directed networks designed with PyGSD. These code snippets solve a link direction prediction problem on a real-world data set. We skip the actual training loop here; full examples can be found in the code repository.

```
1 from sklearn.metrics import accuracy_score
2 import torch
3
4 from torch_geometric_signed_directed.utils import \
5 link_class_split, in_out_degree
6 from torch_geometric_signed_directed.nn.directed import \
7 MagNet_link_prediction
8 from torch_geometric_signed_directed.data import \
9 load_directed_real_data
10
11 device = torch.device('cuda' if \
12 torch.cuda.is_available() else 'cpu')
13
14 data = load_directed_real_data(dataset='webkb',
15 root=path, name='cornell').to(device)
16 link_data = link_class_split(data, prob_val=0.15,
17 prob_test=0.05, task = 'direction', device=device)
18 model = MagNet_link_prediction(q=0.25, K=1, num_features=2,
19 hidden=16, label_dim=2).to(device)
20 criterion = torch.nn.NLLLoss()
```

Listing 4: Preparation, data loading, link splitting, as well as loss and model initialization.

B.2.1 Preparation, Data Loading and Splitting, Loss and Model Initialization

In Listing 4, after importing and defining the device (lines 1-12), we load the *DirectedData* object for the selected data set and map it to the device (lines 14-15). We then create a train-validation-test split of the edge set by using the directed link splitting function (lines 16-17). Finally, we construct the model instance (lines 18-19), and initialize the cross-entropy loss function (line 20).

```
1 def train(X_real, X_img, y, edge_index,
2 edge_weight, query_edges):
3     model.train()
4     out = model(X_real, X_img, edge_index=edge_index,
```

```
5                     query_edges=query_edges,
6                     edge_weight=edge_weight)
7     loss = criterion(out, y)
8     optimizer.zero_grad()
9     loss.backward()
10    optimizer.step()
11    train_acc = accuracy_score(y.cpu(),
12    out.max(dim=1)[1].cpu())
13    return loss.detach().item(), train_acc
14
15 def test(X_real, X_img, y, edge_index, edge_weight,
16 query_edges):
17    model.eval()
18    with torch.no_grad():
19        out = model(X_real, X_img, edge_index=edge_index,
20                    query_edges=query_edges,
21                    edge_weight=edge_weight)
22    test_acc = accuracy_score(y.cpu(),
23    out.max(dim=1)[1].cpu())
24    return test_acc
```

Listing 5: Defining the training and evaluation functions.

B.2.2 Defining Functions for Training and Evaluation

In Listing 5, we define the training and evaluation functions. Setting the model to be trainable (line 3), we obtain edge class assignment probablities with a forward pass of the model instance (lines 4-6). We then obtain the training loss value (line 7). After that, we backpropagate and update the model parameters (lines 8-10). Then, we calculate the accuracy of the training samples (lines 11-12). Finally, we return the loss value as well as the training accuracy (line 13).

For the evaluation function (function *test()*), we do not set the model to be trainable (lines 17-18). With a forward pass (lines 19-21), we obtain the probability assignment matrix. We then obtain test accuracy (line 22) and return the result (line 23).

C Synthetic Data Description

C.1 Signed Stochastic Block Model (SSBM)

A Signed Stochastic Block Model (SSBM) [36] for a network on n nodes with K blocks (clusters), is constructed as follows:

1) Assign block sizes $n_0 \leq n_1 \leq \cdots \leq n_{K-1}$ with size ratio $\rho \geq 1$, as follows. If $\rho = 1$, then the first $K-1$ blocks have the same size $\lfloor n/K \rfloor$, and the last block has size $n - (K-1)\lfloor n/K \rfloor$. If $\rho > 1$, we set $\rho_0 = \rho^{\frac{1}{K-1}}$. Solving $\sum_{i=0}^{K-1} \rho_0^i n_0 = n$ and taking integer value gives $n_0 = \left\lfloor n(1-\rho_0)/(1-\rho_0^K) \right\rfloor$. Further, set $n_i = \lfloor \rho_0 n_{i-1} \rfloor$, for $i = 1, \cdots, K-2$ if $K \geq 3$, and $n_{K-1} = n - \sum_{i=0}^{K-2} n_i$. Then, the ratio of the size of the largest to the smallest block is roughly $\rho_0^{K-1} = \rho$.

2) Assign each node to one of K blocks, so that each block has the allocated size.

3) For each pair of nodes in the same block, with probability p_{in}, create an edge with $+1$ as weight between them, independently of the other potential edges.

4) For each pair of nodes in different blocks, with probability p_{out}, create an edge with -1 as weight between them, independently of the other potential edges.

5) Flip the sign of the across-cluster edges from the previous stage with sign flip probability η_{in}, and η_{out} for edges within and across clusters, respectively.

C.2 Polarized SSBMs (POL-SSBM)

In a polarized SSBM model [36], SSBMs are planted in an ambient network; each block of each SSBM is a cluster, and the nodes not assigned to any SSBM form an ambient cluster. The polarized SSBM model that creates communities of SSBMs, is generated as follows:

1) Generate an Erdős-Rényi graph with n nodes and edge probability p, whose sign is set to ± 1 with equal probability 0.5.

2) Fix n_c as the number of SSBM communities, and calculate community sizes $N_1 \leq N_2 \leq \cdots \leq N_r$, for each of the r communities as in Sec. C.1, such that the ratio of the largest block size to the smallest block size is approximately ρ, and the total number of nodes in these SSBMs is $N \times n_c$.

3) Generate r SSBM models, each with $K_i = 2, i = 1, \ldots, r$ blocks, number of nodes according to its community size, with the same edge probability p, size ratio ρ, and flip probability η.

4) Place the SSBM models on disjoint subsets of the whole network; the remaining nodes not part of any SSBM are dubbed as *ambient* nodes. The resulting polarized SSBM model is denoted as POL-SSBM (n, r, p, ρ, η, N).

C.3 Directed Stochastic Block Model (DSBM)

A standard directed stochastic blockmodel (DSBM) is often used to represent a network cluster structure, see for example [53]. A DSBM model relies on a meta-graph adjacency matrix $\mathbf{F} = (\mathbf{F}_{k,l})_{k,l=0,\ldots,K-1}$ and a *filled* version of it, $\tilde{\mathbf{F}} = (\tilde{\mathbf{F}}_{k,l})_{k,l=0,\ldots,K-1}$, and on a noise level parameter $\eta \leq 0.5$. The meta-graph adjacency matrix $\mathbf{F}$ is generated from the given meta-graph structure, called $\mathcal{M}$. To include an ambient background, the filled meta-graph adjacency matrix $\tilde{\mathbf{F}}$ replaces every zero in $\mathbf{F}$ that is not part of the imbalance structure by 0.5. The filled meta-graph thus creates a number of *ambient nodes* which correspond to entries which are not part of $\mathcal{M}$ and thus are not part of a meaningful cluster; this set of *ambient nodes* is also called the *ambient cluster*. First, we provide examples of structures of $\mathbf{F}$ without any ambient nodes, where $\mathbb{1}$ denotes the indicator function:

1) "*cycle*": $\mathbf{F}_{k,l} = (1-\eta)\mathbb{1}(l = ((k+1) \mod K)) + \eta\mathbb{1}(l = ((k-1) \mod K)) + \frac{1}{2}\mathbb{1}(l = k)$.

2) "*path*": $\mathbf{F}_{k,l} = (1-\eta)\mathbb{1}(l = k+1) + \eta\mathbb{1}(l = k-1) + \frac{1}{2}\mathbb{1}(l = k)$.

3) "*complete*": assign diagonal entries $\frac{1}{2}$. For each pair (k, l) with $k < l$, let $\mathbf{F}_{k,l}$ be η and $1-\eta$ with equal probability, then assign $\mathbf{F}_{l,k} = 1 - \mathbf{F}_{k,l}$.

4) "*star*", following [20]: select the center node as $\omega = \lfloor \frac{K-1}{2} \rfloor$ and set $\mathbf{F}_{k,l} = (1-\eta)\mathbb{1}(k = \omega, l \text{ odd}) + \eta\mathbb{1}(k = \omega, l \text{ even}) + (1-\eta)\mathbb{1}(l = \omega, k \text{ odd}) + \eta\mathbb{1}(l = \omega, l \text{ even})$.

When ambient nodes are present, the construction involves two steps, with the first step the same as the above, but for "*cycle*" meta-graph structure, $\mathbf{F}_{k,l} = (1-\eta)\mathbb{1}(l = ((k+1) \mod (K-1))) + \eta\mathbb{1}(l = ((k-1) \mod (K-1))) + 0.5\,\mathbb{1}(l = k)$. The second step is to assign 0 (0.5, resp.) to the last row and the last column of $\mathbf{F}$ ($\tilde{\mathbf{F}}$, resp.).

A DSBM model, denoted by DSBM $(\mathcal{M}, \mathbb{1}(\text{ambient}), n, K, p, \rho, \eta)$, is built as follows:

1)-2) Same as 1)-2) in Sec. C.1.

3) For nodes $v_i, v_j \in \mathcal{C}_k$, independently sample an edge from v_i to v_j with probability $p \cdot \tilde{\mathbf{F}}_{k,k}$.

4) For each pair of different clusters $\mathcal{C}_k, \mathcal{C}_l$ with $k \neq l$, for each node $v_i \in \mathcal{C}_k$, and each node $v_j \in \mathcal{C}_l$, independently sample an edge from v_i to v_j with probability $p \cdot \tilde{\mathbf{F}}_{k,l}$.

C.3.1 Signed Directed Stochastic Block Model (SDSBM)

A signed directed stochastic block model (SDSBM) [34] relies on a meta-graph adjacency matrix $\mathbf{F} = (\mathbf{F}_{k,l})_{k,l=0,\ldots,C-1}$, edge sparsity level p, the number of nodes n, and on a sign flip noise level parameter $0 \leq \eta \leq 0.5$. An SDSBM model, denoted by SDSBM $(\mathbf{F}, n, p, \rho, \eta)$, is built as follows:

1)-2) same as 1)-2) in Sec. C.1.

3) For nodes $v_i \in \mathcal{C}_k$, and $v_j \in \mathcal{C}_l$, independently sample an edge from v_i to v_j with probability $p \cdot |\mathbf{F}_{k,l}|$. Give this edge weight 1 if $F_{k,l} \geq 0$ and weight -1 if $F_{k,l} < 0$.

4) Flip the sign of all the edges in the generated graph with sign flip probability η.

D Real-World Data Set Description

D.1 Directed Unsigned Networks

As real-world directed, unsigned networks we include the following data sets.

- *Blog* [2] records $|\mathcal{E}| = 19,024$ directed edges between $n = 1,212$ political blogs from the 2004 US presidential election.
- *Migration* [65] reports the number of people that migrated between pairs of counties in the US during 1995-2000. It involves $n = 3,075$ countries and $|\mathcal{E}| = 721,432$ directed edges after obtaining the largest weakly connected component. Since the original directed network has a few extremely large entries, to cope with these outliers we preprocess the input network by $\mathbf{A}_{i,j} = \frac{\mathbf{A}_{i,j}}{\mathbf{A}_{i,j}+\mathbf{A}_{j,i}} \mathbb{1}(\mathbf{A}_{i,j} > 0), \forall i, j \in \{1, \cdots, n\}$, which follows the preprocessing of [16].
- *WikiTalk* [47] contains all users and discussions from the inception of Wikipedia until Jan. 2008. The $n = 2,388,953$ nodes in the network represent Wikipedia users and a directed edge from node v_i to node v_j denotes that user i edited at least once a talk page of user j. We extract the largest weakly connected component.
- *Telegram* [7] is a pairwise influence network between 245 Telegram channels with $8,912$ edges. Labels are generated from the method discussed in [7], with a total of four classes.
- *Cora-ML* [6] is a popular citation network with node labels based on paper topics with seven classes. In this citation network, nodes represent papers, edges denote citations of one paper by another, and node features are the bag-of-words representation of papers. The resulting network has 2,995 nodes and 8,416 edges.
- *CiteSeer* [23] is a citation data set from an automatic citation indexing system. In this citation network, nodes represent papers, and edges denote citations of one paper by another. Node features are the bag-of-words representation of papers, and node labels are determined by the academic topic of a paper. The resulting network has 3,312 nodes and 4,715 edges.
- *Texas*, *Wisconsin*, and *Cornell* are WebKB data sets extracted from the CMU World Wide Knowledge Base (Web->KB) project[2]. They record hyperlinks between websites at different universities. WebKB is a webpage data set collected from computer science departments of various universities by Carnegie Mellon University. In these networks, nodes represent web pages, and edges are hyperlinks between them. Node features are the bag-of-words representation of web pages. The web pages are manually classified into the five categories, student, project, course, staff, and faculty [64]. The resulting networks have 183, 251, and 183 nodes respectively, along with 325, 515, and 298 edges, respectively.
- *Chameleon* and *Squirrel* [67] represent links between Wikipedia pages related to chameleons and squirrels. In these networks, nodes denote web pages and edges represent mutual links between them. Node features correspond to several informative nouns in the Wikipedia pages. The nodes are classified by [64] into five categories in terms of the number of the average monthly traffic of the web page. The resulting networks have 2,277 and 5,201 nodes respectively, along with 36,101 and 222,134 edges, respectively.
- *WikiCS* [58] is a directed network whose nodes correspond to Computer Science articles, and edges are based on hyperlinks. This network has 10 classes resenting different branches of the field. The resulting network has 11,701 nodes and 297,110 edges.
- *Lead-Lag* [4] contains yearly lead-lag matrices from 269 stocks from 2001 to 2019. Each lead-lag matrix is built from a time series of daily price log returns, as detailed in [4]. The lead-lag metric for entry (i, j) in the network encodes a measure of the extent to which stock i leads stock j, and is obtained by applying a functional that computes the signed normalized area under the curve (auc) of the standard cross-correlation function (ccf). The resulting matrix is skew-symmetric, and entry (i, j) quantifies the extent to which stock i leads or lags stocks j, thus leading to a directed network interpretation. Starting from the skew-symmetric matrix, we further convert negative entries to zero, so that the resulting directed network can be directly fed

[2] `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/theo-11/www/wwkb/`

into other methods; note that this step does not throw away any information, and is pursued only to render the representation of the directed network consistent with the format expected by all methods compared. The average number of edges is 29,159.

D.2 Signed Undirected and Signed Directed Networks

For signed networks, we provide data loaders for the following data sets.

- The Sampson monastery data [69] covers 4 social relationships, each of which could be positive or negative. We combine these relationships into a network of 25 nodes. For this data set, we use as node attribute whether or not they attended the minor seminary of "Cloisterville". As ground truth we take Sampson's division of the novices into four groups: Young Turks, Loyal Opposition, Outcasts, and an interstitial group. The number of positive and negative edges are 148 and 182, respectively.
- *Rainfall* [5] contains Australian rainfalls pairwise correlations. This data set is based on the analysis over 294 million daily rainfall measurements since 1876, spanning 17,606 sites across continental Australia. The data set is further processed in [36]. The resulting network has 306 nodes, and the number of positive and negative edges are 64,408 and 29,228, respectively.
- *Fin-YNet* [36, 86] consists of yearly correlation matrices for $n = 451$ stocks for 2000-2020 (21 distinct networks), using so-called *market excess returns*; that is, we compute each correlation matrix from overnight (previous close to open) and intraday (open-to-close) price daily returns, from which we subtract the market return of the S&P500 index. The resulting networks have on average 148,527 positive edges and 54,313 negative edges.
- *S&P1500* [86] considers daily prices for $n = 1,193$ stocks, in the S&P 1500 Index, between 2003 and 2015, and builds correlation matrices also from market excess returns. The result is a fully-connected weighted network, with stocks as nodes and correlations as edge weights. The resulting network has 1,069,319 positive edges and 353,930 negative edges.
- *PPI* [82] is a signed protein-protein interaction (PPI) network. The edge signs represent activation-inhibition relationships. This is a Drosophila melanogaster signed PPI network consisting of 6,125 signed PPIs connecting 3,352 proteins that can be used to identify positive and negative regulators of signaling pathways and protein complexes. The data set is further processed to keep the largest connected component. The resulting network has 3,058 nodes, 7,996 positive edges, and 3,864 negative edges.
- *Wiki-Rfa* [84] is a signed network describing voting information for electing Wikipedia managers. Positive edges represent supporting votes, while negative edges represent opposing votes. The data set is further processed in [36] to keep only the largest weakly connected component and to remove nodes with very low degrees. The resulting network has 7,634 nodes, 135,753 positive edges, and 37,579 negative edges.
- *BitCoin-Alpha* and *BitCoin-OTC* [45] describe bitcoin trading. As a cryptocurrency, Bitcoin is used to trade anonymously over the web, whose counterparty risk [59] has led to the emergence of several exchanges where Bitcoin users rate the level of trust they have in other users. Two such exchanges are OTC (for short) and Alpha (for short). Both exchanges enable users to rate others on a scale of -10 to 10 (excluding zero), where a rating of -10 should be given to fraudsters while 10 means to trust the person as trusting oneself. The rating values in between have intermediate meanings. The resulting networks have 3,783 and 5,881 nodes respectively. *BitCoin-Alpha* has 22,650 positive edges and 1,536 negative edges, while *BitCoin-OTC* has 32,029 positive edges and 3,563 negative edges.
- *Slashdot* [60] relates to a technology-related news website. This network contains friend/foe links between the users of Slashdot. The resulting network has 82,140 nodes, 380,933 positive edges, and 119,548 negative edges.
- *Epinions* [56] describes trust-distrust consumer reviews on `epinions.com`. `epinions.com` is a website in which users can write reviews about products and assign them a rating. This website also allows the users to express their Web of Trust, i.e. "reviewers whose reviews and ratings they have consistently found to be valuable" and their Block list, i.e. a list of authors whose reviews they find consistently offensive, inaccurate, or in general not valuable. Inserting a user in the Web of Trust is the same as issuing a trust statement which is recorded as a positive edge between the user and the reviewer, while inserting them in the Block List means issuing

a distrust statement which is recorded as a negative edge. The resulting network has 131,580 nodes, 589,888 positive edges, and 121,322 negative edges.

- **Fi**nancial **L**ead-**L**ag (FiLL) data sets. For each year in the data set, [34] builds a signed directed graph (*FiLL-pvCLCL*) based on the price return of 444 stocks at market close times on consecutive days. [34] also builds another graph (*FiLL-OPCL*), based on the price return of 430 stocks from market open to close. The lead-lag metric that is captured by the entry $\mathbf{A}_{i,j}$ in each network encodes a measure that quantifies the extent to which stock v_i leads stock v_j, and is obtained by computing the linear regression coefficient when regressing the time series (of length 245) of daily returns of stock v_i against the lag-one version of the time series (of length 245) of the daily returns of stock v_j. Specifically, [34] uses the beta coefficient of the corresponding simple linear regression as the one-day lead-lag metric. The resulting matrix is asymmetric and signed, rendering it amenable to a signed directed network interpretation. The data matrices stored in PyGSD are dense but there is a sparsification parameter provided for the data loader; [34] uses a parameter value of 0.2. The resulting annual graphs *FiLL-OPCL* have on average 84,467 positive edges and 100,013 negative edges, while the resulting annual graphs *FiLL-pvCLCL* have on average 84,677 positive edges and 112,015 negative edges.

E Implementation Details

The original signed GNN implementations for SGCN[3], SNEA[4], SiGAT, and SDGNN[5] are based on the code of GraphSAGE[6]. In our implementations, we inherit the class of MessagePassing in PyG and add new GraphConv operators (e.g., SNEAConv) for signed networks, because this combination is more efficient and unified than the original implementations, which is conducive to the generalization of the code. Tables 7, 8 and 9 report average runtime of our implemented methods on various tasks, showing that we have efficient implementations, with at least as short a runtime as the original implementations. For the other models, the original implementations are based on PyG, so there is not much an efficiency change compared to them. Note that as SSSNET, DIGRAC, and MagNet are not originally written with the MessagePassing class in PyG, we have transformed the model classes accordingly. We observe comparable (SGCN, SSSNET, DIGRAC, and MSGNN) or even better (SiGAT, SDGNN, SNEA, and MagNet) runtime efficiency for these GNN implementations. For the other GNN implementations (DGCN, DiGCN, DiGCNIB, and DiGCL), we clean up the original codes, add documentation for them, unify some functions, split up some auxiliary layers, add unit tests and provide examples.

Node Clustering. We use the same settings as in [36], [35], and [34] respectively for the node clustering task on signed undirected, unsigned directed, and signed directed graphs, respectively. For all the experiments, we use Adam [43] as our optimizer with learning rate 0.01 and employ ℓ_2 regularization with weight decay parameter $5 \cdot 10^{-4}$ to avoid overfitting. For SSSNET [36] and DIGRAC [35], we use hidden size 16, 2 hops, and $\tau = 0.5$. For MSGNN [34], we take a hidden size of 16 and use two layers of convolution. For all synthetic models, we train the GNNs with a maximum of 1000 epochs, and stop training when no gain in validation performance is achieved for 200 epochs (early-stopping). Note that we adapt SSSNET and DIGRAC from matrix multiplication implementations to MessagePassing (from PyG) implementations for their models, and we observe comparable runtime efficiency for our implementations and the original implementations. Specially, for the node clustering results reported in Figures 1a and 1b, both implementations of DIGRAC consume roughly 0.05 seconds for each epoch, while both implementations of SSSNET consume roughly 0.4 seconds per epoch. Experiments were conducted on a compute node with 8 Nvidia Tesla T4, 96 Intel Xeon Platinum 8259CL CPUs @ 2.50GHz and 378GB RAM.

For all node clustering tasks, we set 10% of all nodes from each cluster as test nodes, 10% as validation nodes to select the model, and the remaining 80% as training nodes. For signed networks, we are in a semi-supervised setting, where 10% of the training nodes are set to be seed nodes. Note that label supervision is not provided for all training nodes but only for seed nodes in SSSNET, DIGRAC, and MSGNN, while the subgraph induced by the set of training nodes is the graph to

[3]https://github.com/benedekrozemberczki/SGCN

[4]https://github.com/liyu1990/snea

[5]https://github.com/huangjunjie-cs/SiGAT

[6]https://github.com/williamleif/graphsage-simple

which the self-supervised loss function is applied. The other GNNs compared here (DiGCL, DiGCN, DiGCNIB, MagNet and DGCN) are trained in a fully-supervised manner, meaning that labels for all training nodes are used during training, and also the self-supervised loss function is not applied to them, see [35] for more details. For the DSBM experiments, we generate 5 DSBM networks under each parameter setting and use 10 different data splits for each network, then average over the 50 runs. For the remaining synthetic models, we first generate five different networks, each with two different data splits, then conduct experiments on them and report average performance over these 10 runs. The meta-graph adjacency matrices for the SDSBM models are defined as

$$\mathbf{F}_1(\gamma) = \begin{bmatrix} 0.5 & \gamma & -\gamma \\ 1-\gamma & 0.5 & -0.5 \\ -1+\gamma & -0.5 & 0.5 \end{bmatrix}, \mathbf{F}_2(\gamma) = \begin{bmatrix} 0.5 & \gamma & -\gamma & -\gamma \\ 1-\gamma & 0.5 & -0.5 & -\gamma \\ -1+\gamma & -0.5 & 0.5 & -\gamma \\ -1+\gamma & -1+\gamma & -1+\gamma & 0.5 \end{bmatrix}.$$

In Sec. 5 also some non-GNN methods are mentioned, and they are included in the experiments in Sec. 8. In detail,

Herm and Herm_sym are two variants from [16], L and L_sym are two Signed Laplacian variants from [46], sns and dns are two Laplacian variations from [57], BNC and BRC are abbreviations for Balanced Normalized Cut and Balanced Ratio Cut from [13], and finally SPONGE and SPONGE_sym are two variants from [15].

Link Sign Prediction on Signed Graphs. Link Sign Prediction on Signed Graphs is the task to predict the sign of given edges in a network, which is a binary classification task. Considering the label imbalance, F1 and AUC are used to evaluate the performance. Past signed GNNs on this task adopt a two-stage process, first using signed network embedding methods to learn representations from the train set, and then using a downstream classifier to predict the signs of edges in the test set. However, the setting is not conducive to searching hyper-parameters. In this paper, we split 80% edges for training, 10% edges for validation, and the remaining edges constitute the test set. We run such train/validation/test splits five times to report the mean and standard deviation over the five runs. The downstream classifier is a logistic regression classifier with 80% train edges. The dimension number of node representations $\mathbf{Z}$ is set to 20 for all the methods, which is widely used in related works. We use Adam optimizer to optimize all the models (The learning rate is set to 0.01 and the weight decay is set to 1e-3). We run 500 epochs (200 epochs as warming up) with 10 epoch evaluations on the AUC metric on the test set and adopt early stopping with the patience of 10 epochs to prevent overfitting. For other parameters, we follow the authors' suggestions and the released codes. Experiments were conducted on a compute node with an NVIDIA Tesla V100S GPU (32GB) and Intel(R) Xeon(R) Silver 4310 CPUs (250GB). Table 7 reports the average runtime per epoch for both our implementations and the original ones, where we observe a huge improvement in runtime efficiency over the original implementations for SiGAT, SNEA and SDGNN. For SGCN, the original and our implementations consume comparable runtime, and hence we omit the runtime report for the original implementation here.

Table 7: Link sign prediction average runtime per epoch (in seconds) for signed networks, where "ours" denotes our implementation, and "original" denotes the original implementation. The fastest is marked in **bold**.

Method	Bitcoin-Alpha	Bitcoin-OTC	Wikirfa	Slashdot	Epinions
SGCN(ours)	**0.08**	**0.08**	0.26	0.80	1.27
SiGAT(ours)	0.10	0.11	**0.11**	**0.14**	**0.19**
SiGAT(original)	2.93	4.71	51.83	121.56	236.53
SDGNN(ours)	**0.08**	0.07	0.12	0.25	0.41
SDGNN(original)	7.30	12.06	93.21	359.57	out of memory
SNEA(ours)	0.09	0.16	0.38	0.91	1.41
SNEA(original)	0.92	2.16	10.11	220.15	431.42

Node Classification and Edge-Level Tasks on Directed Unsigned Graphs. We adopt the settings in [88] for MagNet, DGCN, DiGCN, and DiGCNIB, for directed network node classification and edge-level tasks. For DiGCL results, we use the settings in [75] for Cora_ML and CiteSeer for node

classification and search parameters for the other data sets. For link prediction tasks, we follow the settings from [88], training all GNNs for each link prediction task for 3000 epochs and set the early stopping as 500. We tune the number of filters in $[16, 32, 64]$ for MagNet, DGCN, DiGCN, and search the number of filters of DiGCNIB in $[6, 11, 21]$. The coefficient α of DiGCN and DiGCNIB is searched in $[0.05, 0.1, 0.15, 0.2]$. The q parameter in MagNet is tuned in $[0.05, 0.1, 0.15, 0.2, 0.25]$. Table 8 reports the total runtime for the node classification and link prediction tasks on directed unsigned networks with 2 layers of different directed graph convolution, training with 10 folds $\times$ 500 epochs per fold. 32 filters are used in DGCN, DiGCN, DiGCNIB and MagNet. For DiGCL, 128 filters are used for GCNs and 64 filters are used for the projection. As shown in the table, our implementations are efficient. The running time of directed unsigned graphs is evaluated on a compute node with 16 threads of i9-11900KF CPU, RTX3090 GPU, and 32GB memory. We adopted the original implementation for directed graph convolution layers except for MagNet. We developed MagNet with MessagePassing from PyG in our package. The comparison of the MagNet runtimes validates our efficiency improvement.

Table 8: Runtime (seconds) comparison on link tasks. The fastest is marked in **bold**. MagNet (original) denotes the original implementation and MagNet (ours) denotes the implementation of our package.

Task	Method	Cornell	Texas	Wisconsin	CoraML	CiteSeer	Telegram
Node classification	DGCN	16	17	17	43	43	17
	DiGCN	**9**	**9**	**9**	**32**	**37**	**10**
	DiGCNIB	19	19	19	42	46	20
	DiGCL	28	29	30	63	57	37
	MagNet (original)	73	74	81	215	173	54
	MagNet (ours)	21	21	21	96	94	20
Direction prediction	DGCN	30	30	31	56	55	34
	DiGCN	**10**	**10**	**10**	**31**	**34**	**11**
	DiGCNIB	19	19	19	41	42	21
	Magnet(original)	53	52	52	61	57	56
	MagNet(ours)	36	36	36	40	39	43
Existence link prediction	DGCN	29	29	30	58	56	35
	DiGCN	**10**	**9**	**10**	**31**	**34**	**13**
	DiGCNIB	19	20	19	40	43	22
	Magnet(original)	51	52	53	63	59	60
	MagNet(ours)	36	37	37	42	40	43
Three classes link prediction	DGCN	30	30	31	59	55	36
	DiGCN	**10**	**10**	**10**	**33**	**34**	**13**
	DiGCNIB	19	19	19	42	44	22
	Magnet(original)	52	52	52	61	57	59
	MagNet(ours)	36	37	37	41	39	41

Link Prediction on Signed Directed Graphs. We follow the settings from [34], training all GNNs for each link prediction task for 300 epochs. We use the proposed loss functions from their original papers for SGCN [19], SNEA [49], SiGAT [40], and SDGNN [41], and we use the cross-entropy loss for SSSNET [36] and MSGNN [34]. For all link prediction experiments in Table 6, we sample 20% edges as test edges, and use the rest of the edges for training. Five splits were generated randomly for each input graph. We calculate the in- and out-degrees based on both signs from the observed input graph (removing test edges) to obtain a four-dimensional feature vector for each node for training SSSNET [36], and MSGNN [34], and we use the default settings from relative papers for SGCN [19], SNEA [49], SiGAT [40], and SDGNN [41]. Note that the "avg." results for *FiLL* first average the accuracy values across all individual networks (a total of 42 networks), then report the mean and standard deviation over the five runs. Table 9 reports average runtimes for the experiments. Experiments were conducted on a compute node with 8 Nvidia Tesla T4, 96 Intel Xeon Platinum 8259CL CPUs @ 2.50GHz and 378GB RAM.

Table 9: Runtime (seconds) comparison on link tasks. The fastest is marked in **bold**.

Data Set	Task	GCN	SGCN	SDGNN	SiGAT	SNEA	SSSNET	MSGNN
BitCoin-Alpha	SP	**23**	352	124	277	438	59	29
	DP	**24**	328	196	432	498	78	37
	3C	**32**	403	150	288	446	77	37
	4C	**30**	385	133	293	471	57	36
	5C	**31**	350	373	468	570	82	37
BitCoin-OTC	SP	**27**	340	140	397	584	68	30
	DP	**26**	471	243	426	941	80	38
	3C	**37**	292	252	502	551	92	**37**
	4C	**31**	347	143	487	607	68	37
	5C	**37**	460	507	500	959	86	38
Slashdot	SP	**167**	4218	3282	1159	5792	342	227
	DP	**171**	4231	3129	1200	5773	311	222
	3C	**208**	3686	6517	1117	6628	263	322
	4C	**133**	4038	5296	948	7349	202	232
	5C	**212**	4269	7394	904	8246	424	327
Epinions	SP	**272**	6436	4725	2527	8734	300	370
	DP	**273**	6437	4605	2381	8662	404	369
	3C	**322**	6555	8746	2779	10536	471	510
	4C	**214**	6466	6923	2483	10380	272	384
	5C	**329**	7974	9310	2719	11780	460	517
FiLL (avg.)	SP	**29**	591	320	367	617	61	32
	DP	**30**	387	316	363	386	53	36
	3C	**36**	542	471	298	657	79	43
	4C	**27**	608	384	343	642	56	35
	5C	**38**	318	534	266	521	63	44