pyGSL: A Graph Structure Learning Toolkit

Anonymous Author(s)
Affiliation
Address
email

Abstract

We introduce pyGSL, a Python library that provides efficient implementations of state-of-the-art graph structure learning models along with diverse datasets to evaluate them on. The implementations are written in GPU-friendly ways, allowing one to scale to much larger network tasks. A common interface is introduced for algorithm unrolling methods, unifying implementations of recent state-of-the-art techniques and allowing new methods to be quickly developed by avoiding the need to rebuild the underlying unrolling infrastructure. Implementations of differentiable graph structure learning models are written in PyTorch, allowing us to leverage the rich software ecosystem that exists e.g., around logging, hyperparameter search, and GPU-communication. This also makes it easy to incorporate these models as components in larger gradient based learning systems where differentiable estimates of graph structure may be useful, e.g. in latent graph learning. Diverse datasets and performance metrics allow consistent comparisons across models in this fast growing field. The full code repository on GitHub will be included upon acceptance.

1 Introduction

Graph Learning (GL), also referred to as Network Topology Inference or Graph Structure Learning, refers to the task of inferring graphs from data. Here we focus on learning undirected graphs in the supervised setting. GL has deep roots in statistics [Dempster, 1972] with significant contributions for probabilistic graphical model selection; see e.g. [Kolaczyk, 2009, Friedman et al., 2008, Drton and Maathuis, 2017]. Approaches dubbed 'latent graph learning' have been used to learn interactions among coupled dynamical systems [Kipf et al., 2018], or to obtain better task-driven representations of relational data for machine learning (ML) applications [Wang et al., 2019, Kazi et al., 2020, Veličković et al., 2020]; for the related task of link-prediction see [Hamilton, 2020] . Graph signal processing (GSP) has been the source of recent advances using cardinal properties of network data such as smoothness [Dong et al., 2016, Kalofolias, 2016] and graph stationarity [Segarra et al., 2017, Pasdeloup et al., 2018], exploiting models of network diffusion [Wasserman et al., 2022, Daneshmand et al., 2014], or taking a signal representation approach [Dong et al., 2019, Mateos et al., 2019]. These works, referred to as Model-Based (MB) graph learning, use such data models to formulate the topology inference task as a (convex) optimization problem to be solved for each problem instance. When datasets are available, one can build on these MB approaches by unrolling an iterative solution procedure to produce a learned architecture optimized on the given data [Monga et al., 2021]. Such Unrolling-Based (UB) methods offer several advantages over MB methods: they tend to be more expressive (MB methods are restricted to a small set of constraints expressible in a convex manner), require fewer layers than the corresponding iterative algorithm for a given performance requirement, and allow one to directly optimize for a (differentiable) metric of interest [Dong et al., 2019, Wasserman et al., 2022, Shrivastava et al., 2019]. Thus by learning a distribution over such graphs by investing time upfront in training a model we can obtain better task-specific performance while avoiding expensively resolving each new problem instance.

Submitted to 36th Conference on Neural Information Processing Systems (NeurIPS 2022). Do not distribute.

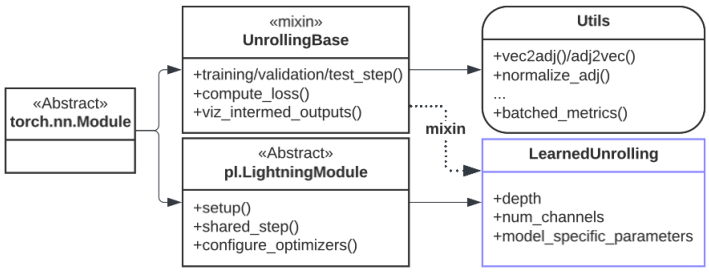

Figure 1: Class diagram of pyGSL's learned unrollings with a subset of variables and methods shown. For a specific learned unrolling, say GDN, we would replace the LearnedUnrolling class with a GDN class which simply overwrites the setup(), shared_step(), and configure_optimizers() methods from its parent class pl.LightningModule. Since the GDN class also mixes in `UnrollingBase`, it automatically inherits functionality common to all UB methods.

While the interdisciplinary history of GL has been a source of progress, it has also been a hindrance: these different fields use different tools (MATLAB/Octave, R, Python, etc) in implementing their algorithms. This has slowed the spread of new algorithms and made comparing relative performance of different algorithms difficult. Importantly, there has been little effort to ensure implementations can leverage the increasing computation power of the GPU. Due to the reliance of GL algorithms on matrix/vector operations, as well as the interest in handling ever increasing network sizes, GPUs offer an ideal environment for such computation to occur. Thus the first aim of pyGSL is to provide standardized GPU-compatible implementations of fundamental GL methods, along with datasets to evaluate them on. The second aim is lowering the barrier to use and develop such GL methods. As such, we provide a framework which makes it easy for researchers to extend - or build completely new - graph learning methods without needing to rebuild all the underlying software machinery. By consolidating wide-ranging GL methods into in a single GPU-compatible framework, pyGSL can be seen as the most comprehensive Python library for GL to date.

2 Architecture

Both MB and UB methods rely heavily on matrix/vector operations, often require hyperparameter search capabilities, and are typically applied to problems with large graph sizes and/or large sets of graphs; see Section 3 for further details on said GL approaches. For these reasons, pyGSL implements all methods in PyTorch which has a rich ecosystem of software to interface with GPUs, run and visualize hyperparameter searches via W&B [Biewald, 2020], and abstract away the bug-prone training, validation and optimization logic required in gradient based training via PyTorch-Lightning [Falcon and The PyTorch Lightning team, 2019].

Implementations of UB methods share significant structure, motivating our novel `UnrollingBase` class, which encapsulates this shared functionality; see Fig. 1. This minimizes repeated code, and makes implementing new UB methods as simple as defining the core layer-wise logic for that specific unrolling. By doing so, users can (i) cut their development time down significantly by relying on pretested methods; and (ii) automatically gain functionality such as intermediate output visualization and metric logging. See Appendix A.4 for a code snippet showing how pyGSL can be used.

3 A Unifying View of Graph Learning Methods

The GL problem is to infer a graph from data to uncover a latent complex information structure. Here, we focus on undirected and weighted graphs $\mathcal{G}(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \ldots, N\}$ is the set of nodes (henceforth common to all graphs), and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ collects the edges. A graph signal $\boldsymbol{x} = [x_1, \ldots, x_N] \in \mathbb{R}^N$ is a map $\boldsymbol{x} : \mathcal{V} \to \mathbb{R}$ which assigns a real value (say, a feature) to each vertex. We collect the P graph signal observations together into data matrix $\boldsymbol{X} = [\boldsymbol{x}^{(1)}, \cdots, \boldsymbol{x}^{(P)}]$. A similarity function $S(\boldsymbol{X}) : \mathbb{R}^{N \times P} \mapsto \mathbb{R}^{N \times N}$ is chosen to compute the observed direct similarity between nodes. Common choices for S include sample covariance/correlation or Euclidean distance. We can conceptualize the output of $S(\boldsymbol{X})$ as the symmetric adjacency matrix of an observed graph

Table 1: Model-Based methods with the associated Unrolling-Based method it inspires.

Data Model	**MB Iterative Solution Procedure**	**UB Model**
Gaussian	Alternating Minimization	GLAD
Smoothness	Primal-Dual Splitting	L2G
Diffusion	Proximal Gradient Descent	GDN

from which we would like to recover a latent graph with symmetric adjacency matrix denoted $\boldsymbol{A}_L \in \mathbb{R}_+^{N \times N}$. Below we discuss three popular approaches to tackle the GL task which exemplify the link between $S(\boldsymbol{X})$ and $\boldsymbol{A}_L$; for discussions on Deep Learning (DL) methods for GL and tradeoffs between the three methods refer to A.2 and A.3, respectively.

Model-Based Graph Learning. These GL approaches postulate some data model relating the observed data $\boldsymbol{X}$ to the latent graph $\mathcal{G}_L$ via $\boldsymbol{X} \sim \mathcal{F}(\boldsymbol{A}_L)$. This model can be the result of a network process, e.g. linear network diffusion, or statistical where $\boldsymbol{X}$ follows a distribution determined by $\boldsymbol{A}_L$ e.g. in probabilistic graphical models. Thus the GL task reduces to an attempt to invert this relation $\mathcal{F}^{-1}(\boldsymbol{X})$. To do so, these techniques formulate a (convex) optimization problem which can be solved via iterative optimization methods. These MB GL problems have the general form

$$\boldsymbol{A}^* \in \underset{\boldsymbol{A} \in \mathcal{C}}{\operatorname{argmin}} \{\mathcal{L}_{\text{data}}(\boldsymbol{A}, \boldsymbol{X}) + \mathcal{L}_{\text{reg}}(\boldsymbol{A})\}, \tag{1}$$

where $\mathcal{L}_{\text{data}}(\boldsymbol{A}, \boldsymbol{X})$ is the data fidelity term, $\mathcal{L}_{\text{reg}}(\boldsymbol{A})$ is the regularization term incorporating the structural priors (e.g., $\|\boldsymbol{A}\|_1$ for sparsity), and $\mathcal{C}$ encodes a convex constraint on the optimization variable $\boldsymbol{A}$, e.g., symmetry, non-negativity, or hollow diagonal. We inject our assumptions about the generative model into the objective of the problem. Note that we parameterize the canonical problem form with an adjacency $\boldsymbol{A}$ for notational convenience, but many methods use other graph shift operators such as the Laplacian $\boldsymbol{L} := \text{diag}(\boldsymbol{A}\mathbf{1}) - \boldsymbol{A}$ or its normalized counterparts.

To make this clear, consider the following data models and resulting optimization formulations for graph recovery, summarized in Table 1. When the data $\boldsymbol{X}$ are assumed to be Gaussian, the GL task is to estimate the precision matrix; the sparsity-regularized log-likelihood function for such a precision matrix is convex $\arg\min_{\mathbf{A} \succeq \mathbf{0}} \{\text{Tr}(S(\boldsymbol{X})\mathbf{A}) + \alpha\|\mathbf{A}\|_1 - \beta \log\det \mathbf{A}\}$, where $S(\boldsymbol{X})$ is the sample covariance matrix [Friedman et al., 2008]. When the data is assumed to be smooth on $\mathcal{G}_L$, i.e., the total variation $\text{Tr}(\boldsymbol{X}^\top \boldsymbol{L} \boldsymbol{X})$ of the signals on the graph is small, a standard GL formulation is $\arg\min_{\mathbf{A} \in \mathcal{C}} \left\{\|\boldsymbol{A} \odot S(\boldsymbol{X})\|_1 - \alpha \mathbf{1}^\top \log(\boldsymbol{A}\mathbf{1}) + \beta\|\boldsymbol{A}\|_F^2\right\}$, where $S(\boldsymbol{X})$ is the Euclidean distance matrix [Kalofolias, 2016]. When the observed graph $S(\boldsymbol{X})$ satisfies a graph convolutional relationship with $\boldsymbol{A}_L$, i.e. $S(\boldsymbol{X}) = \sum_{i=0}^K \alpha_i \boldsymbol{A}_L^i$, as is the case in linear network diffusion where again $S(\boldsymbol{X})$ is the sample covariance matrix, we can pose the GL task as the non-convex problem $\arg\min_{\mathbf{A} \in \mathcal{C}} \left\{\|S(\boldsymbol{X}) - \sum_{i=0}^K \alpha_i \boldsymbol{A}_L^i\|_F^2 + \beta\|\boldsymbol{A}\|_1\right\}$ [Wasserman et al., 2022].

The MB methods that admit iterative solutions take the generic form: $\boldsymbol{A}[i+1] = h_\theta(\boldsymbol{A}[i], S(\boldsymbol{X}))$ where $\boldsymbol{A}[i]$ is output on the i-th iteration, h_θ is the contractive function, and θ are the regularization parameters. We implement the function h_θ using GPU compatible operations in PyTorch and wrap it in a loop until uniform convergence is achieved.

Unrolling-Based Graph Learning. Algorithm unrolling uses iterative algorithms, like those often used in signal processing, as an inductive bias in the design of the neural network architectures. The algorithm is unrolled into a deep network by associating each layer to a single iteration in the truncated algorithm, producing a finite number of stacked layers as shown in Figure 2. We transform the regularization parameters of the iterative algorithms into learnable parameters of the neural network. Using a dataset, such parameters can be optimized for a given task by choosing an appropriate loss function and backproprogating gradients; see [Monga et al., 2021] for more details.

All UB models share a significant amount of code in their construction, training, and evaluation. We consolidate this code into the novel `UnrollingBase` class. To instantiate a new UB model, one simply inherits from `UnrollingBase` and Pytorch-Lightning's `LightningModule`, declares a layers' learnable parameters θ^i, and implements the $h_{\theta^i}(\boldsymbol{A}[i], S(\boldsymbol{X}))$ function - denoted as `shared_step` in the users class. Recall that each layer of the resulting unrolled network performs the same operation represented by $h_\theta(\boldsymbol{A}[i], S(\boldsymbol{X}))$ here; thus specifiying the entire unrolled network reduces

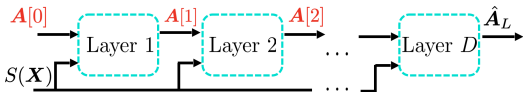

Figure 2: Schematic of a generic UB method.

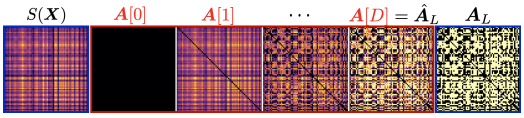

Figure 3: Intermediate outputs plotted via pyGSL.

to providing a desired depth and specifying h for a single layer. The UnrollingBase mixin takes care of stacking the layers together, feeding outputs of one layer as inputs of the next, and implements the required interface to allow the `LightningModule` to automate the training and optimization.

4 Data

Synthetics As discussed in Section 3, MB methods motivate the optimization problem through an assumed generative model on the observed data. Since UB methods are inspired by their MB counterparts, synthetic datasets which match these assumed generative models are important for model validation across GL methods. We thus provide (i) an efficient graph sampling interface using NetworkX [Hagberg et al., 2008] for a broad range of random graph ensembles; and (ii) an ability to generate signals conforming to smoothness, linear network diffusion ($\boldsymbol{x} = \sum_{i=0}^{p} \alpha_i A_L^i \boldsymbol{w}$, where $\boldsymbol{w}$ is typically white), or Gaussianity ($\boldsymbol{x} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{A}^{-1})$). These datasets can be accessed via the `smooth`, `diffuse`, and `gaussian` classes, respectively, in the `gl.data` subdirectory.

Real Data We provide real datasets to evaluate the models on, including over 1000 Structural Connectivity (SC) and Functional Connectivity (FC) pairs extracted from the HCP-YA neuroimaging dataset [Glasser et al., 2016], as well as co-location and social network data from the Thiers13 dataset [Génois and Barrat, 2018]. We also provide the ability to create 'Pseudo-synthetic' data by using the graphs from the real datasets, and sampling synthetic signals on top of them that are e.g., smooth, diffused, or Gaussian. This provides a gentler transition to real data, which we found useful when developing and validating GL methods.

See Appendix A.1 for a further discussion on the data and its associated class layout in pyGSL.

5 Related Work

While there has been an explosion of software development in adjacent fields such as geometric DL [Fey and Lenssen, 2019] and network-based modeling of complex systems [Traxl et al., 2016], there has been little in the way of comprehensive GL packages. GSPBox is (unmaintained) Matlab toolkit which focuses on traditional GSP tasks and offers a single function to perform GL on the smooth signal case [Perraudin et al., 2016]. There are many algorithms for Gaussian graphical model selection from a likelihood based [Yuan and Lin, 2007, Friedman et al., 2008], regression based [Meinshausen and Bühlmann, 2006, Peng et al., 2009], and constrained $\ell 1$-minimization [Cai et al., 2011] approaches; almost all implementations are in R or Matlab. Recently Choi et al. [2021] released an R-based GPU implementation of CONCORD-PCD, a regression based approach which uses parallelized coordinate descent for fast inference. Software in [Lassance et al., 2020] offers benchmarks for MB approaches. To the best of the authors knowledge, no encompassing framework exists for UB methods, only some standalone implementations released by the respective methods' authors.

6 Concluding remarks

We introduced the pyGSL framework for GPU-aware fast and scalable GL. We provide several synthetic and real datasets for model evaluation, as well as a novel class for UB architectures which dramatically lowers the time and expertise required to use and build such methods. It also ensures their compatibility as sub-modules in larger gradient-based learning systems. This is an active project, and we plan to add more benchmark datasets and the latest GL methods as they are published. We welcome more researchers and engineers to join, develop, maintain, and improve this toolkit to push forward the research and deployment of network topology inference algorithms.

References

Arthur P Dempster. Covariance selection. *Biometrics*, pages 157–175, 1972.

E. D. Kolaczyk. *Statistical Analysis of Network Data: Methods and Models*. Springer, New York, NY, 2009.

J. Friedman, T. Hastie, and R. Tibshirani. Sparse inverse covariance estimation with the graphical lasso. *Biostatistics*, 9(3):432–441, 2008.

Mathias Drton and Marloes H Maathuis. Structure learning in graphical modeling. *Annu. Rev. Stat. Appl*, 4:365–393, 2017.

Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. Neural relational inference for interacting systems. In *International Conference on Machine Learning*, volume 80, pages 2688–2697, 10–15 Jul 2018.

Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph CNN for learning on point clouds. *ACM Trans. Graph.*, 38(5), October 2019.

Anees Kazi, Luca Cosmo, Nassir Navab, and Michael M. Bronstein. Differentiable graph module (DGM) for graph convolutional networks. *CoRR*, abs/2002.04999, 2020. URL `https://arxiv.org/abs/2002.04999`.

Petar Veličković, Lars Buesing, Matthew C. Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. In *Advances in Neural Information Processing Systems*, 2020.

William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159, 2020.

Xiaowen Dong, Dorina Thanou, Pascal Frossard, and Pierre Vandergheynst. Learning laplacian matrix in smooth graph signal representations. *IEEE Trans. Signal Process.*, 64(23):6160–6173, 2016.

Vassilis Kalofolias. How to learn a graph from smooth signals. In *Artificial Intelligence and Statistics*, pages 920–929, 2016.

Santiago Segarra, Antonio G. Marques, Gonzalo Mateos, and Alejandro Ribeiro. Network topology inference from spectral templates. *IEEE Trans. Signal Inf. Process. Netw.*, 3(3):467–483, August 2017.

Bastien Pasdeloup, Vincent Gripon, Grégoire Mercier, Dominique Pastor, and Michael G. Rabbat. Characterization and inference of graph diffusion processes from observations of stationary signals. *IEEE Trans. Signal Inf. Process. Netw.*, 4(3):481–496, 2018.

Max Wasserman, Saurabh Sihag, Gonzalo Mateos, and Alejandro Ribeiro. Learning graph structure from convolutional mixtures, 2022. URL `https://arxiv.org/abs/2205.09575`.

Hadi Daneshmand, Manuel Gomez-Rodriguez, Le Song, and Bernhard Schoelkopf. Estimating diffusion network structures: Recovery conditions, sample complexity & soft-thresholding algorithm. In *International Conference on Machine Learning*, pages 793–801. PMLR, 2014.

Xiaowen Dong, Dorina Thanou, Michael Rabbat, and Pascal Frossard. Learning graphs from data: A signal representation perspective. *IEEE Signal Process. Mag.*, 36(3):44–63, 2019.

Gonzalo Mateos, Santiago Segarra, Antonio G. Marques, and Alejandro Ribeiro. Connecting the dots: Identifying network structure via graph signal processing. *IEEE Signal Process. Mag.*, 36(3): 16–43, May 2019.

Vishal Monga, Yuelong Li, and Yonina C. Eldar. Algorithm unrolling: Interpretable, efficient deep learning for signal and image processing. *IEEE Signal Process. Mag.*, 38(2):18–44, 2021.

Harsh Shrivastava, Xinshi Chen, Binghong Chen, Guanghui Lan, Srinvas Aluru, Han Liu, and Le Song. Glad: Learning sparse graph recovery, 2019.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

William Falcon and The PyTorch Lightning team. PyTorch Lightning, 3 2019. URL https://github.com/PyTorchLightning/pytorch-lightning.

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.

Matthew F Glasser, Stephen M Smith, Daniel S Marcus, Jesper LR Andersson, Edward J Auerbach, Timothy EJ Behrens, Timothy S Coalson, Michael P Harms, Mark Jenkinson, Steen Moeller, et al. The human connectome project's neuroimaging approach. *Nature neuroscience*, 19(9):1175–1187, 2016.

Mathieu Génois and Alain Barrat. Can co-location be used as a proxy for face-to-face contacts? *EPJ Data Science*, 7(1):11, May 2018.

Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric, 2019.

Dominik Traxl, Niklas Boers, and Jürgen Kurths. Deep graphs - a general framework to represent and analyze heterogeneous complex systems across scales. *Chaos*, 26(6):065303, 2016. doi: http://dx.doi.org/10.1063/1.4952963. URL http://arxiv.org/abs/1604.00971v1.

Nathanaël Perraudin, Johan Paratte, David Shuman, Lionel Martin, Vassilis Kalofolias, Pierre Vandergheynst, and David K. Hammond. Gspbox: A toolbox for signal processing on graphs, 2016.

Ming Yuan and Yi Lin. Model selection and estimation in the Gaussian graphical model. *Biometrika*, 94(1):19–35, 2007.

Nicolai Meinshausen and Peter Bühlmann. High-dimensional graphs and variable selection with the lasso. *The annals of statistics*, 34(3):1436–1462, 2006.

Jie Peng, Pei Wang, Nengfeng Zhou, and Ji Zhu. Partial correlation estimation by joint sparse regression models. *Journal of the American Statistical Association*, 104(486):735–746, 2009.

Tony Cai, Weidong Liu, and Xi Luo. A constrained 1 minimization approach to sparse precision matrix estimation. *Journal of the American Statistical Association*, 106(494):594–607, 2011.

Young-Geun Choi, Seunghwan Lee, and Donghyeon Yu. An efficient parallel block coordinate descent algorithm for large-scale precision matrix estimation using graphics processing units, 2021.

Carlos Lassance, Vincent Gripon, and Gonzalo Mateos. Graph topology inference benchmarks for machine learning. In *Intl. Workshop on Machine Learning for Signal Processing (MLSP)*, pages 1–6, 2020.

David C Van Essen, Stephen M Smith, Deanna M Barch, Timothy EJ Behrens, Essa Yacoub, Kamil Ugurbil, Wu-Minn HCP Consortium, et al. The WU-Minn Human Connectome Project: An overview. *Neuroimage*, 80:62–79, 2013.

Zhengwu Zhang, Maxime Descoteaux, Jingwen Zhang, Gabriel Girard, Maxime Chamberland, David Dunson, Anuj Srivastava, and Hongtu Zhu. Mapping population-based structural connectomes. *Neuroimage*, 172:130–145, May 2018.

Zhengwu Zhang, Genevera I. Allen, Hongtu Zhu, and David Dunson. Tensor network factorizations: Relationships between brain structural connectomes and traits. *NeuroImage*, 197:330–343, 2019. ISSN 1053-8119. doi: https://doi.org/10.1016/j.neuroimage.2019.04.027. URL https://www.sciencedirect.com/science/article/pii/S1053811919303131.

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

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks?, 2018. URL https://arxiv.org/abs/1810.00826.

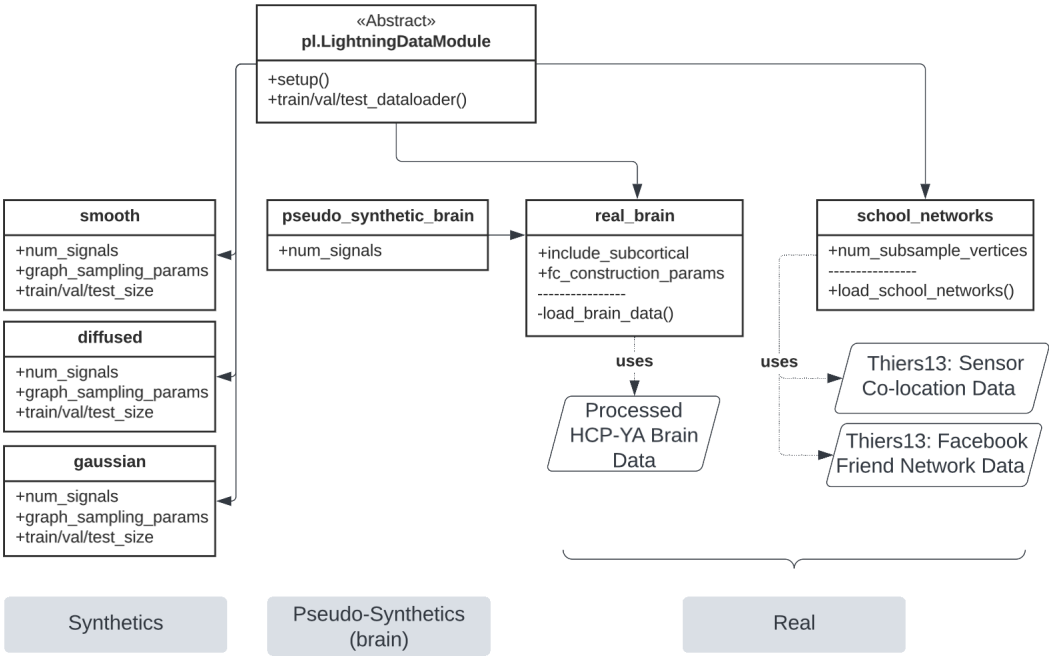

Figure 4: Class diagram of pyGSL's provided datasets

A Appendix

A.1 Data Class Diagram

There are three main categories of data provided by pyGSL as shown in A.1. The first being Synthetics, which are constructed as to be faithful to a generative model assumed by at least one of the provided MB methods, e.g. diffused and GDNs. The second are Real Data, particularly FC-SC pairs derived from HCP-YA neuroimaging data [Van Essen et al., 2013] and a large Sensor Co-location - Facebook Friend Network graph pair derived from the Theirs13 dataset [Génois and Barrat, 2018]. The last category of data are the 'Pseudo-synthetic' data which is a compromise between real and synthetic data: we use real observed graphs but sample synthetic signals - either smooth, diffused, or Gaussian - on top of them. This provides a smoother transition from synthetic to the more difficult real data.

Raw HCP-YA data can be found here, and use of raw data - and derivative forms of such data - is subject to the HCP Data Use Agreement. If the processed SC's are used, please cite [Zhang et al., 2018, 2019].

A.2 Deep Learning Approach to GL

Classic Graph Neural Networks (GNN) can be manipulated to perform the GL task, often by decoding latent node feature vectors into predicted edges. Due to low inductive bias and poor scaling properties DL models struggle on many GL tasks, but we include two popular GNNs, Graph Convolutional Network (GCN) [Kipf and Welling, 2017] and Graph Isomorphism Networks (GIN) [Xu et al., 2018], with appropriate adjustments for tackling the GL task.

A.3 Tradeoffs in GL Approaches

Factors which influence the decision of which GL method to use include training and inference complexity, scaling properties, expressive power, parameter efficiency, and empirical performance.

Inference in MB methods involves solving an optimization problem which often has cubic run-time and memory complexity w.r.t. the number of nodes $\mathcal{O}(N^3)$. Training in MB methods involves performing a hyperparameter search, which involves solving many such optimization problems, posing the same problematic run-time and memory issues. MB methods are highly parameter

(here regularization coefficients) efficient - a characteristic of convex optimization problems, but lack expressivity due to the limited constraints expressible in a convex form. These methods are empirically observed to perform well when the graphs are small enough such that comprehensive searches over the regularization coefficients can be performed.

Inference and training in DL methods involves forward passes (and backward passes for training) through the deep network which is typically quadratic in run-time and memory w.r.t the number of nodes $\mathcal{O}(N^2)$. These methods tend to be highly expressive, and while more parameter efficient than MLPs due to parameter sharing, they still typically require on the order of $10^3 - 10^6$ parameters, and often even more, depending on the particular GNN. For reasonable performance they tend to require a significant amount of data, more than is typically available in most use cases.

While one could categorize UB methods as DL methods - because they do in fact generate a deep NN architecture - they are unique enough for their own category. Training and inference is identical to other DL methods, involving forward passes of the data through the network (and the corresponding backward pass to propagate gradients for training) with the associated quadratic run-time and memory complexities. These methods have significant inductive bias inherited from the associated MB iterative algorithm, making them highly parameter efficient but also more expressive because the (now learnt) parameters can be tuned on a dataset via gradient based learning. Empirically, these methods tend to perform best in most cases.

An additional note is that MB and UB methods are naturally inductive, meaning you can train on graphs of size N_{train} and deploy that same model on graphs of size $N_{test} >> N_{train}$, while - in general - DL methods do not have this property. This is because DL methods can often use features of the graph which functionally depend on the graph size - tieing them to the particular graph size in inference. With some adjustment, it is possible to break this dependence and make these methods inductive.

A.4 Python Example

```python
import graph_learning as gl
import pytorch-lightning as pl

graph_sampling = {'graphs': 'ER', 'n': 100, 'p': 0.5}
data_size = {'train': 500, 'val': 100, 'test': 100}
dm = gl.data.smooth(graph_sampling, **data_size)

model = gl.models.unroll.glad(depth=20)

trainer = pl.Trainer(max_epochs=200, 'gpus': 1)
trainer.fit(model, datamodule=dm)
trainer.test(model, datamodule=dm, ckpt_path='best')
```

Figure 5: Example of how to use the package. All training loops, optimization, and GPU communication are abstracted away, simplifying the interface.

Figure 5 shows how pyGSL would be used in a simple example of training a GLAD network - an UB method motived from a MB method which postulaes a Gaussian generative data model - of depth 20 on 'smooth' data, see Section 3 for details.