pyGSL: A Graph Structure Learning Toolkit

Max Wasserman and Gonzalo Mateos

University of Rochester

A Central Place for Graph Structure Learning

Brings together SOTA models with synthetic and real datasets. Models are built in a unified way using PyTorch allowing faster development, ease of extension, and seamless scaling to GPU settings.

Notation

- Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ denote an undirected and weighted graph, where $\mathcal{V} = \{1, \dots, N\}$ is the set of nodes, and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ collects the edges.
- A graph signal $\boldsymbol{x} = [x_1, \dots, x_N] \in R^N$ 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 $\boldsymbol{X} = [\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(P)}]$.
- 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.

The Graph Structure Learning Problem: Inferring Graphs from Data

Graph Structure Learning (GSL) is posed here as an inverse problem. Given graph $S(\boldsymbol{X})$, where $\boldsymbol{X} \sim \mathcal{F}(\boldsymbol{A}_L)$, recover the latent graph $\boldsymbol{A}_L$. The generative model $\mathcal{F}$ ties the nodal data $\boldsymbol{X}$ to the latent graph $\boldsymbol{A}_L$, e.g. statistical or diffusion processes. Here we focus on learning undirected graphs in the supervised setting; we do not assume direct access to nodal features $\boldsymbol{X}$.

A Unifying View of GSL Methods

Ad Hoc methods rely on intuitive approaches such as thresholding or kNN. **Model-Based (MB)** methods use data model $\mathcal{F}$ to pose a (sometimes convex) optimization problem with corresponding iterative solution procedures. When datasets are available **Unrolling-Based (UB)** approaches take these iterative procedures and use them to motivate a deep network architecture that uses backpropogation on a dataset to learn its parameter values; see Figure 1. **Deep Learning (DL)** approaches tend to parameterize a GNN with dense $S(\boldsymbol{X})$, and decode latent node features into edge predictions.

Model-Based GSL. Pose optimization problem

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

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. sparsity), and $\mathcal{C}$ encodes a convex constraint on the optimization variable $\boldsymbol{A}$. We introduce an iterative solution procedure (2) to solve (1) which takes generic form

$$\boldsymbol{A}[i+1] = h_\theta(\boldsymbol{A}[i], S(\boldsymbol{X})) \tag{2}$$

where $\boldsymbol{A}[i]$ is output on the i-th iteration, h_θ is the contractive function, and θ are the regularization parameters.

Unrolling-Based GSL. Use iterative solution procedure as inductive bias in the design of deep network architecture by mapping truncated iterations into layers, transforming regularization parameters into learnable parameters, and optimizing parameter values via backproprogation with a differentiable loss function on a given dataset.

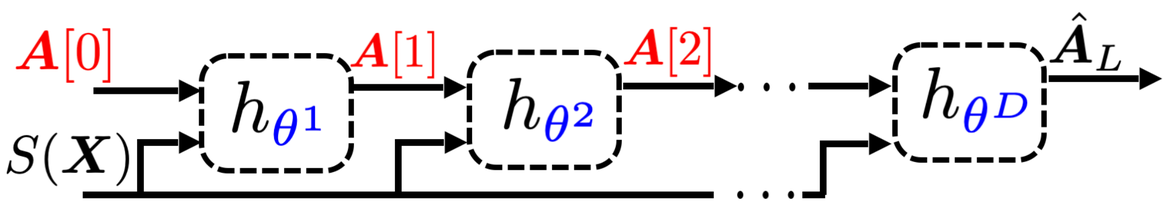

Figure 1. Schematic of a UB method obtained by unrolling iterative procedure (2).

Data Model	MB Iterative Procedure	UB Model
Gaussian	Alternating Minimization	GLAD [2]
Smoothness	Primal-Dual Splitting	L2G [1]
Diffusion	Proximal Gradient Descent	GDN [3]

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

UB: A Compromise Between MB and DL

UB methods gain interpretability, parameter efficiency, and graph-size-inductiveness from the inductive bias inherited from their respective MB methods. As in DL methods, UB methods gain expressivity by forgoing the convexity constraint and directly optimizing a (differentiable) metric of interest, explicit controls on complexity by truncating layers, and a tuneable trade off on training time and inference speed via network depth and width of layers.

The `Layer` and `Unrolling` classes

UB methods differ in the types of layers to stack and thus pyGSL provides the `Layer` and `Unrolling` abstract classes encapsulating their shared functionality. By implementing the required abstract methods in each, users automatically gain the ability to scale to many GPUs and leverage pre-configured logging and visualization tools.

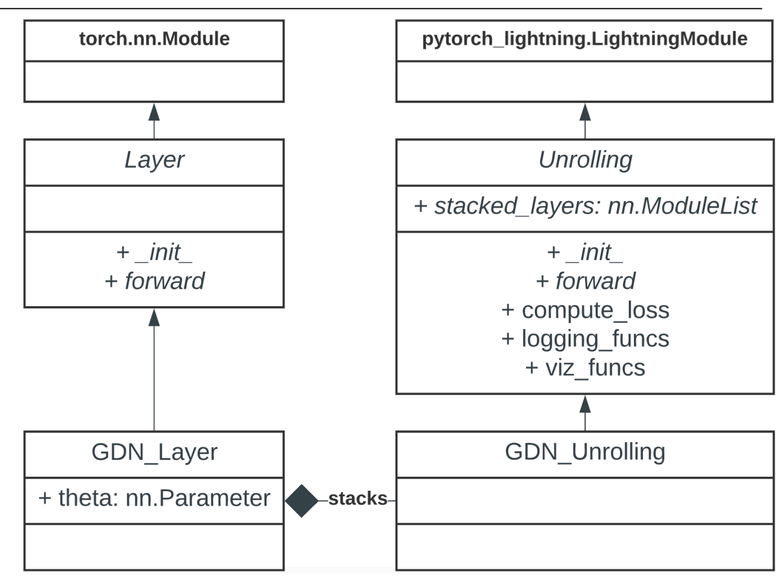

Figure 2. Class diagram for UB methods in pyGSL.

Workflow for UB Model Development

1. **Define Layer** h_θ: Subclass `Layer` specifying learned parameters θ in __init__() and the differentiable function h_θ in forward()
2. **Create Unrolling**: Subclass `Unrolling` and implement forward() to specify how a layers outputs should be fed as inputs to the following layer, applying e.g. normalization between layers. Optionally implement __init__() for custom functionality.
3. **Perform Learning**: Specify loss function in the unrolling and choose a dataset, e.g. 'squared_error' with $\boldsymbol{A}_L$'s being ErdősRényi graphs, $\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{0}, (\boldsymbol{A}_L + \epsilon \boldsymbol{I})^{-1})$, and $S(\boldsymbol{X})$ being sample covariance matrices.
4. **Evaluate**: Compare performance with other provided models.

Scaling to Larger Network Tasks

GPUs make the GSL problem feasible on significantly larger graphs: Figure 3 shows the dramatic speed-up GPUs provide in the learning process. An advantage of UB methods is the ability to explicitly choose inductive bias, e.g. sparsity. Future work will explore the use of sparse data representations to scale to graphs with orders of magnitude larger size.

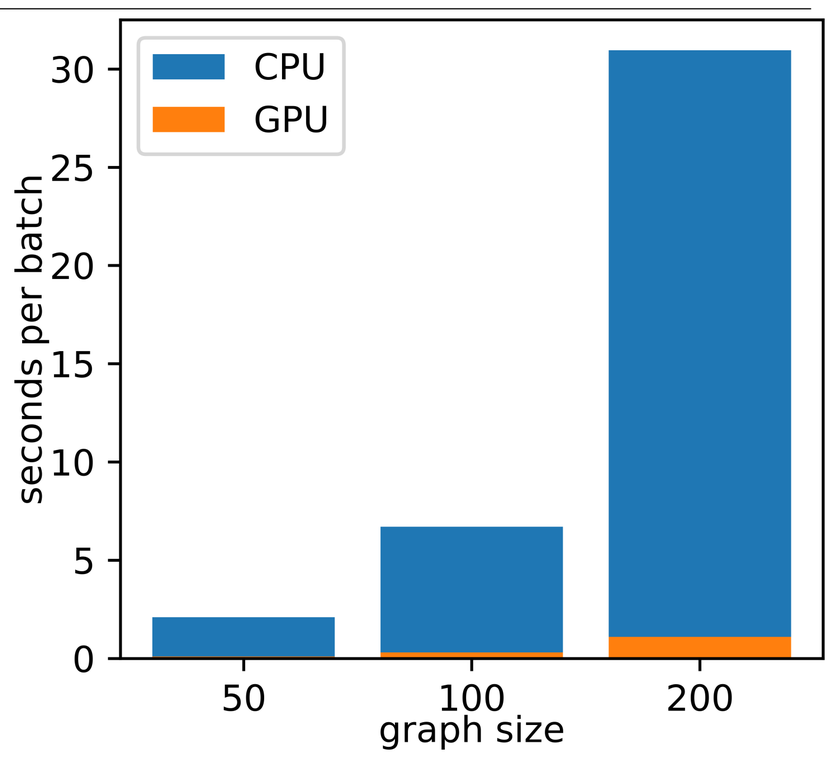

Figure 3. Time for the forward and backward pass in the GDN model on CPUs/GPU of g4dn.xlarge AWS instance.

References

[1] Xingyue Pu, Tianyue Cao, Xiaoyun Zhang, Xiaowen Dong, and Siheng Chen. Learning to learn graph topologies, 2021.

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

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