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 \( G(V, E) \) denote an undirected and weighted graph, where \( V = \{1, \ldots, N\} \) is the set of nodes, and \( E \subseteq V \times V \) collects the edges.
- A graph signal \( x = [x_1, \ldots, x_N] \in \mathbb{R}^N \) is a map \( x : V \rightarrow \mathbb{R} \) which assigns a real value (say, a feature) to each vertex. We collect the \( P \) graph signal observations together into data matrix \( X = [x_1, \ldots, x_P] \).
- A similarity function \( S(X) : \mathbb{R}^{N \times P} \rightarrow \mathbb{R}^{N \times N} \) 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 \( S(X) \), where \( X \sim F(A_L) \), recover the latent graph \( A_L \). The generative model \( F \) ties the nodal data \( X \) to the latent graph \( A_L \), 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 \( X \).

### 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 \( F \) 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 backpropagation on a dataset to learn its parameters. Here we focus on learning undirected graphs in the supervised setting; we do not assume direct access to nodal features \( X \).

#### Model-Based GSL

Pose optimization problem

\[
A^* = \arg\min_{A \in \mathcal{C}} \mathcal{L}_{data}(A, X) + \mathcal{L}_{reg}(A),
\]

where \( \mathcal{L}_{data}(A, X) \) is the data fidelity term, \( \mathcal{L}_{reg}(A) \) is the regularization term incorporating the structural priors (e.g., sparsity), and \( \mathcal{C} \) encodes a convex constraint on the optimization variable \( A \). We introduce an iterative solution procedure (2) to solve (1) which takes generic form

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

where \( A[i] \) is the output at the \( i \)-th iteration, \( h_{\theta} \) is the constructive function, and \( \theta \) is 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 backpropagation with a differentiable loss function on a given dataset.

\[
\begin{align*}
A[0] & \quad \mathbb{h}_{\theta_1} \quad A[1] \\
S(X) & \quad \mathbb{h}_{\theta_2} \quad A[2] \\
& \quad \mathbb{h}_{\theta_D} \quad A_L
\end{align*}
\]

### 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.

### Workflow for UB Model Development

1. Define Layer \( h_\theta \): Subclass \( \text{Layer} \) specifying learned parameters \( \theta \) in \( \text{__init__()} \) and the differentiable function \( h_\theta \) in \( \text{forward()} \).
2. Create Unrolling: Subclass \( \text{Unrolling} \) and implement \( \text{forward()} \) to specify how a layers outputs should be fed as inputs to the following layer; applying e.g. normalization between layers. Optionally implement \( \text{__init__()} \) for custom functionality.
3. Perform Learning: Specify loss function in the unrolling and choose a dataset, e.g. `squared_error` with \( A_L \)'s being Erdős-Rényi graphs, \( X \sim \mathcal{N}(0, (A_L + I)^{-1}) \), and \( S(X) \) being sample covariance matrices. Optionally tune 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.

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

**Figure 2.** Class diagram for UB methods in pyGSL.

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

### References


---

**A Central Place for Graph Structure Learning**

Max Wasserman and Gonzalo Mateos

University of Rochester

---

**pyGSL: A Graph Structure Learning Toolkit**

---

**Workshop for New Frontiers in Graph Learning**

---

**References**

