We start at 10:05

Scaling GNNs

Note: much of the material for this lecture was taken from *Stanford's CS224W (Winter 2022)* by Jure Leskovec

Slides by Jay Siri (jsiri@caltech.edu)

Overview

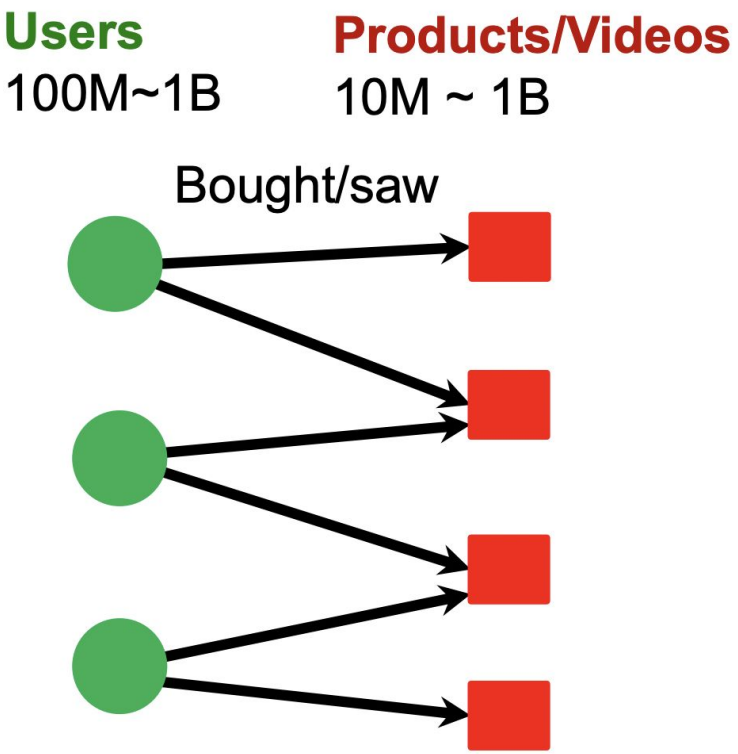
1. Applications of Large Graphs
2. Scalability Issues
3. Sampling Techniques
4. Simplifying GNN Architecture

Applications of Large Graphs

Why do we care about scaling GNNs?

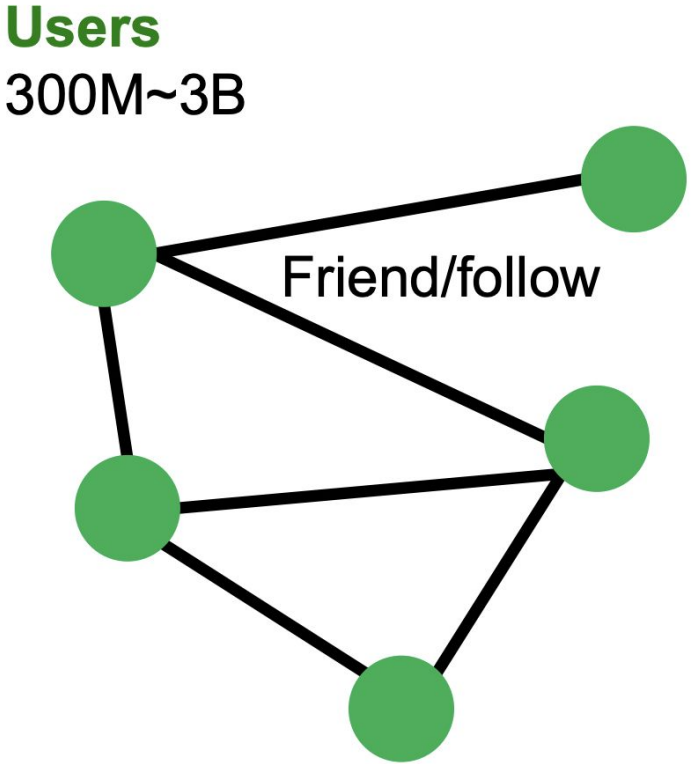
Graphs at Scale

- Recommender systems:
  - Amazon
  - YouTube
  - Pinterest
  - Etc.
- ML tasks:
  - Recommend items (link prediction)
  - Classify users/items (node classification)



Graphs at Scale

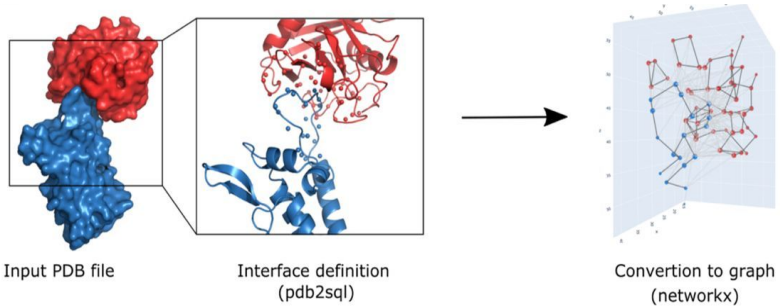
- Social networks:
  - Facebook
  - Twitter
  - Instagram
  - Etc.
- ML tasks:
  - Friend recommendation (link-level)
  - User property prediction (node-level)



Graphs at Scale

- Biology:
  - Protein force-field modeling
  - Other graph examples: disease pathways, gene interaction networks
- ML tasks:
  - Implicit solvation free energy prediction (graph-level)
  - Pairwise energy prediction (link-level)

Atoms
>10k

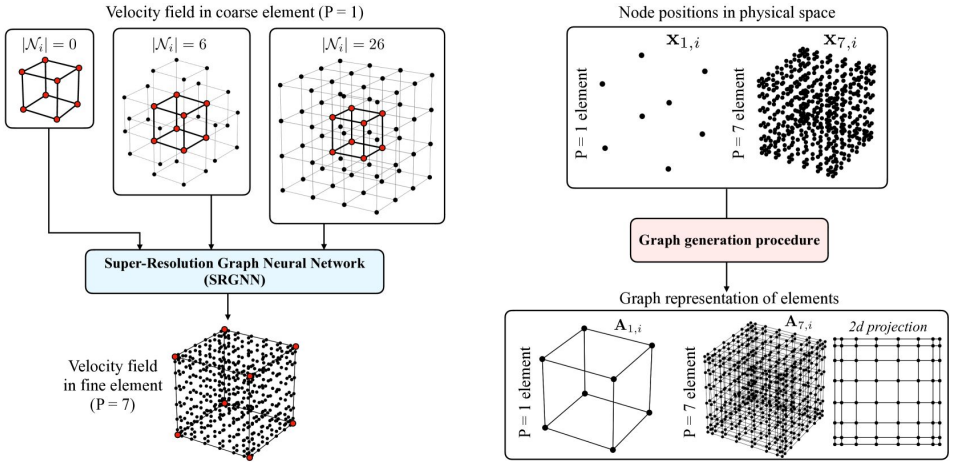


Source: DeepRank-GNN: A Graph Neural Network Framework to Learn Patterns in Protein-Protein Interfaces (2021)

Graphs at Scale

- Physics:
  - Super-resolution of fluid flows
- ML tasks:
  - Full-field modeling (graph-level)
  - Local velocity field modeling (node-level)

Mesh
~36^3 points



Source: Mesh-based Super-Resolution of Fluid Flows with Multiscale Graph Neural Networks (2024)

What is in Common?

- Large-scale:
  - #nodes ranges from 10k to 10B
  - #edges ranges from 1M to 100B
- Tasks:
  - Node-level: user/atom/point prediction
  - Edge-level: interaction prediction
  - Graph-level: structure prediction (e.g., proteins, fluid fields)

Scalability Issues

Challenges in training and inference on large graphs

Question 1

What are some possible issues when training large GNNs? Why might it be bad to use mini-batch SGD like typical model training?

Why is it Hard?

- **Recall**: How we usually train an ML model on large data (*N* = #data is large)
- **Objective**: Minimize the averaged loss

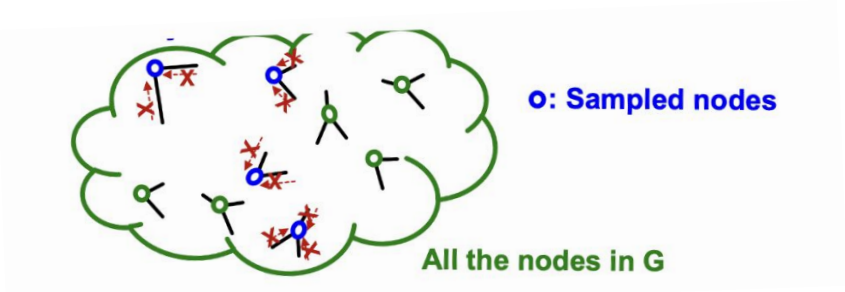
$$\ell(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=0}^{N-1} \ell_i(\boldsymbol{\theta})$$

$\boldsymbol{\theta}$: model parameters, $\ell_i(\boldsymbol{\theta})$: loss for i-th data point.

- We perform Stochastic Gradient Descent (SGD).
  - Sample *M* ($\ll$ *N*) data points (mini-batches).
  - Compute the $\ell_{sub}(\boldsymbol{\theta})$ over the *M* data points.
  - Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \ell_{sub}(\boldsymbol{\theta})$

Why is it Hard?

- In mini-batch, we sample M ($\ll$ N) nodes independently:



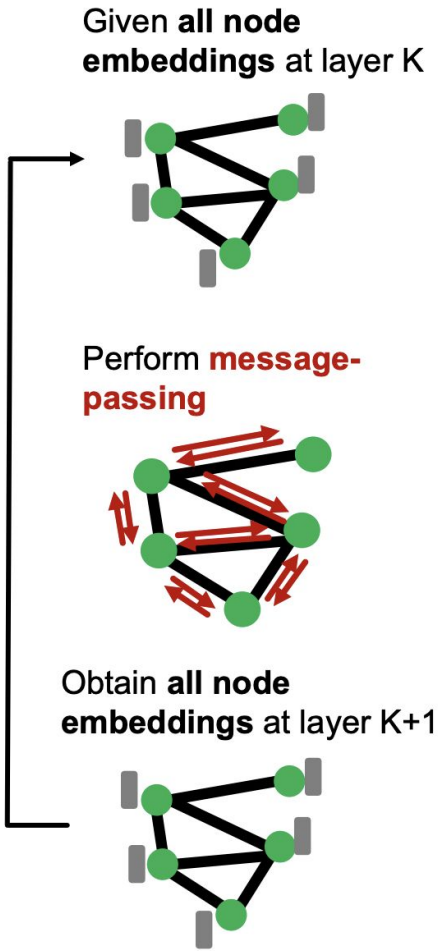
- Sampled nodes will be be isolated from each other!
- GNN generates node embeddings by aggregating neighboring node features.
  - GNN does not access to neighboring nodes within the mini-batch!
  - Standard SGD cannot effectively train GNNs.

Why is it Hard?

- Naïve full-batch implementation: Generate embeddings of all the nodes at the same time:

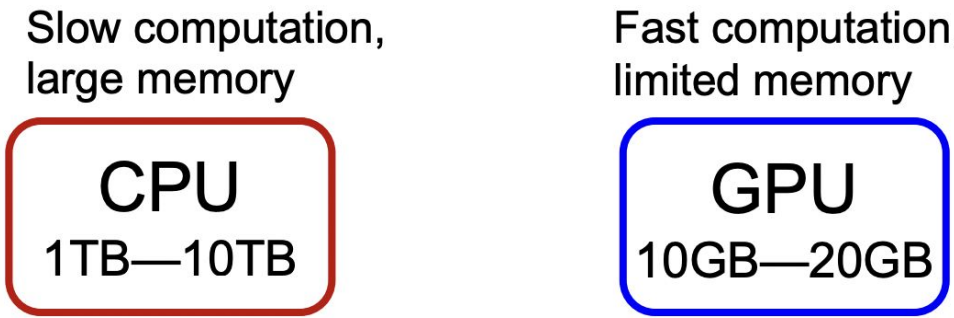
$$H^{(k+1)} = \sigma(\tilde{A}H^{(k)}W_k^{\mathrm{T}} + H^{(k)}B_k^{\mathrm{T}})$$

  - Load the entire graph A and features X. Set $H^{(0)} = X$.
  - At each GNN layer: Compute embeddings of all nodes using all the node embeddings from the previous layer.
  - Compute the loss
  - Perform gradient descent



Why is it Hard?

- However, **Full-batch** implementation is **not feasible** for large graphs.
- Why?
  - Because we want to use GPU for fast training, but GPU memory is extremely limited (10GB-80GB).
  - The entire graph and the features cannot be loaded on GPU.



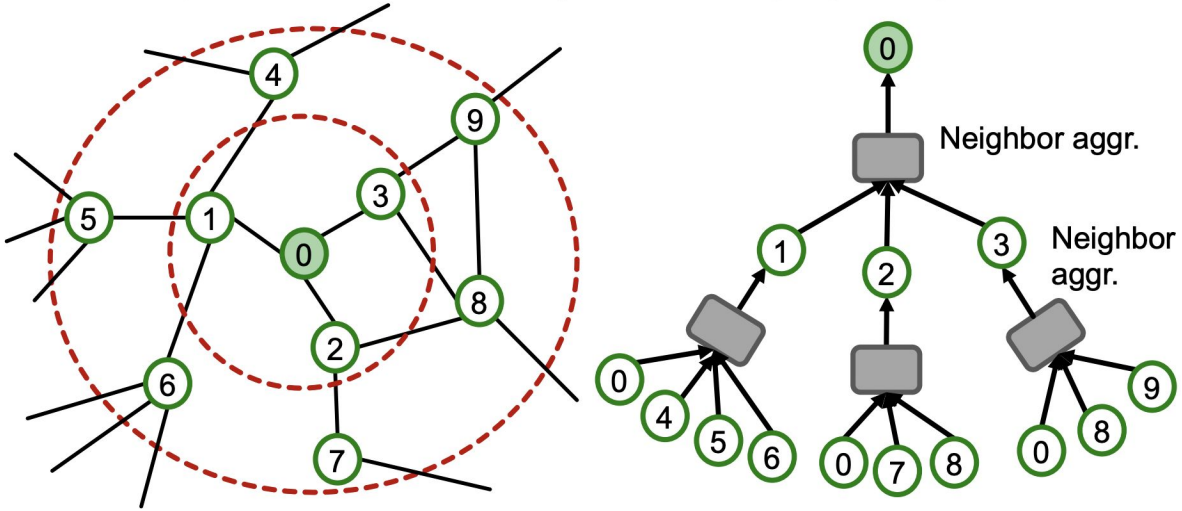
Sampling Techniques

Reduce the size of data needed to train GNNs

Neighborhood Sampling

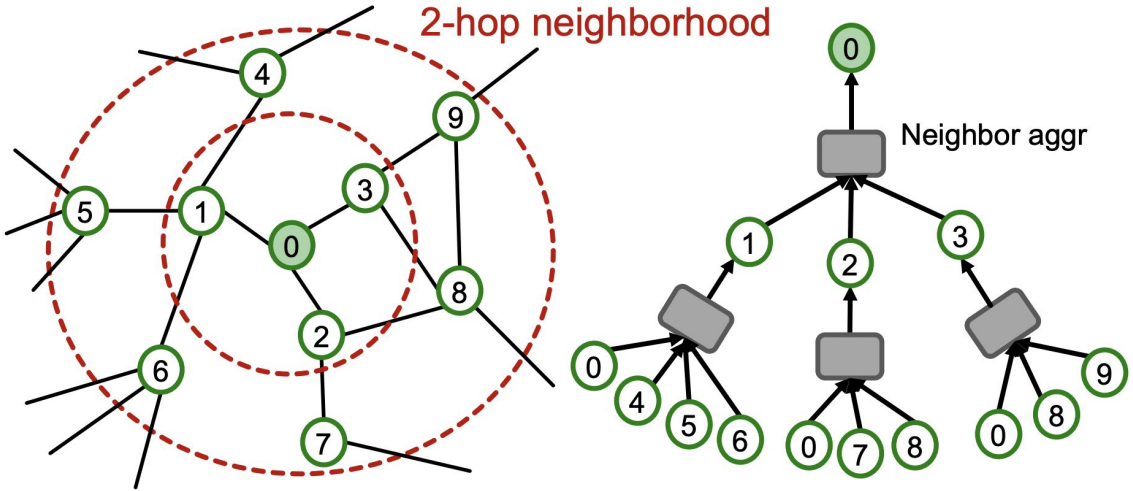
Recall: Computational Graph

- Recall: GNNs generate node embeddings via neighbor aggregation.
  - Represented as a computational graph (right).



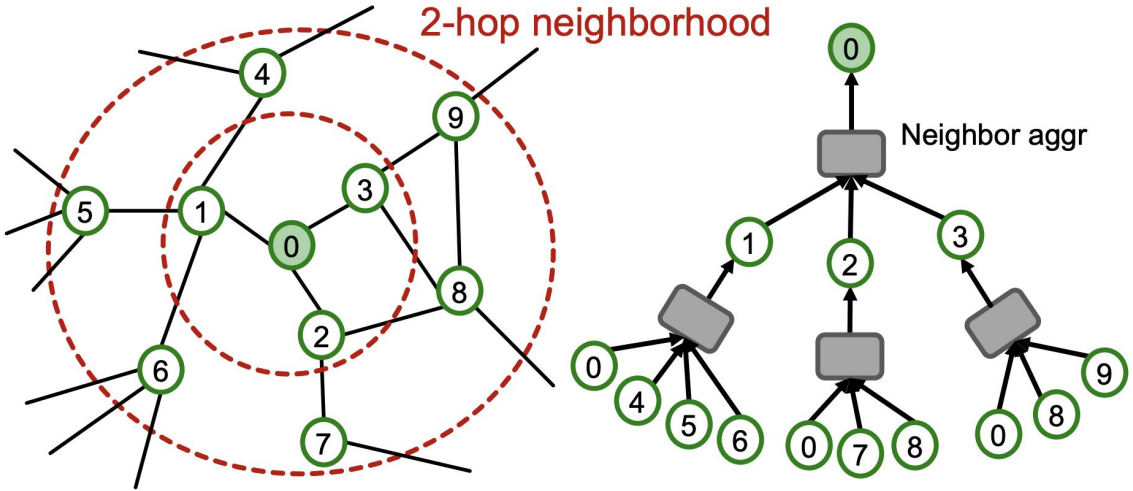
Recall: Computational Graph

- Observation: A 2-layer GNN generates embedding of node “0” using 2-hop neighborhood structure and features



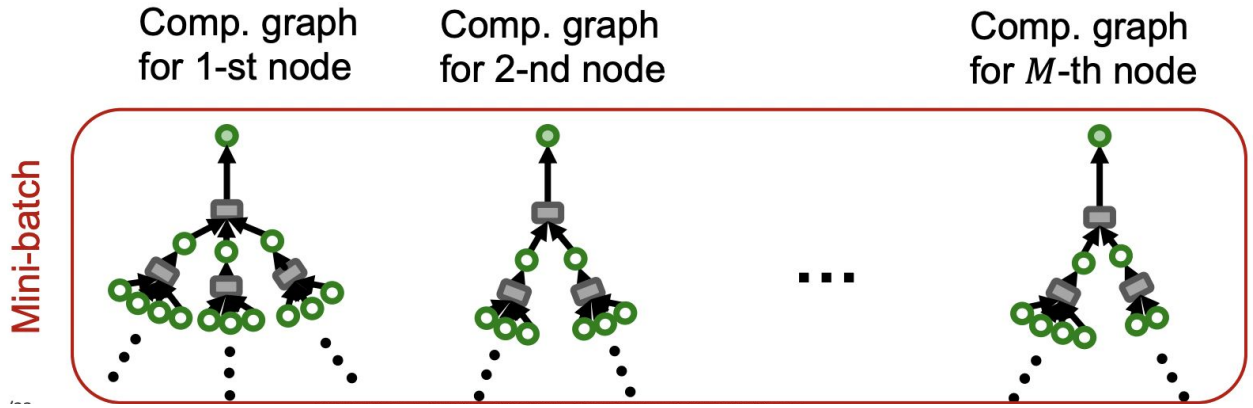
Recall: Computational Graph

- Observation: More generally, *K*-layer GNNs generate embedding of a node using *K*-hop neighborhood structure and features.



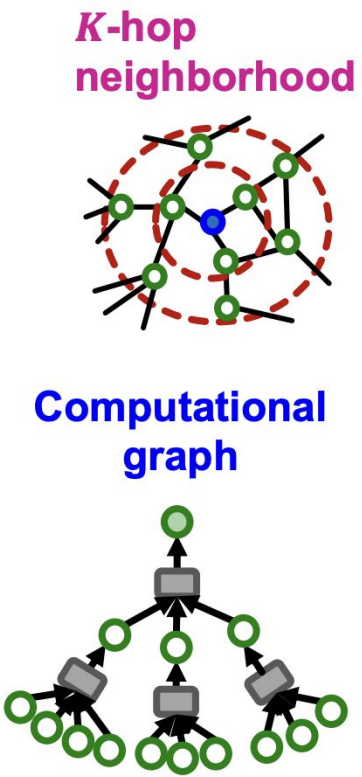
Computing Node Embeddings

- **Key insight**: To compute embedding of a single node, all we need is **the *K*-hop neighborhood** (which defines the computation graph).
- Given a set of *M* different nodes in a mini-batch, we can generate their embeddings using *M* computational graphs. **Can be computed on GPU!**



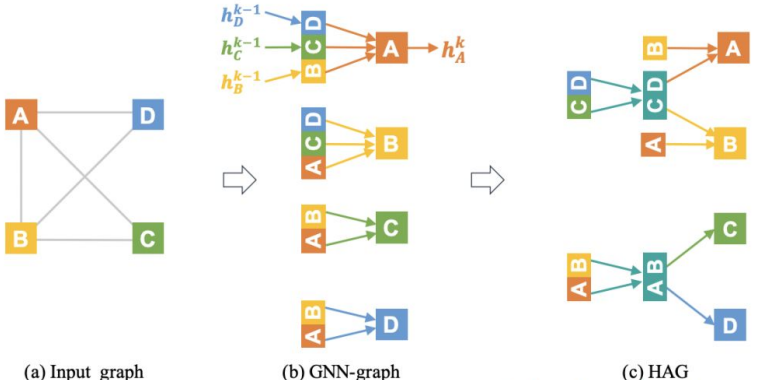
Stochastic Training of GNNs

- We can now consider the following SGD strategy for training K-layer GNNs:
  - Randomly sample M ($\ll N$) root nodes.
  - For each sampled root node v:
    - Get K-hop neighborhood and construct the computation graph.
    - Use the above to generate v's embedding.
  - Compute the loss $\ell_{sub}(\boldsymbol{\theta})$ over the *M* nodes.
  - Perform SGD: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \ell_{sub}(\boldsymbol{\theta})$



Issue with Stochastic Training (1/2)

- For each node, we need to get the entire K-hop neighborhood and pass it through the computation graph.
- We need to aggregate lot of information just to compute one node embedding.
- Some computational redundancy:

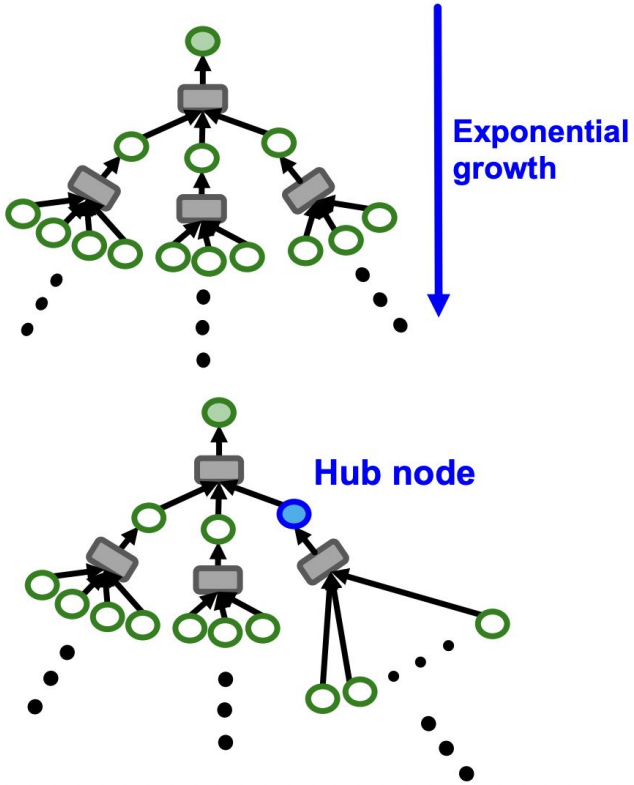


(a) Input graph (b) GNN-graph (c) HAG

Redundancy-Free Computation for Graph Neural Networks, KDD 202

Issue with Stochastic Training (2/2)

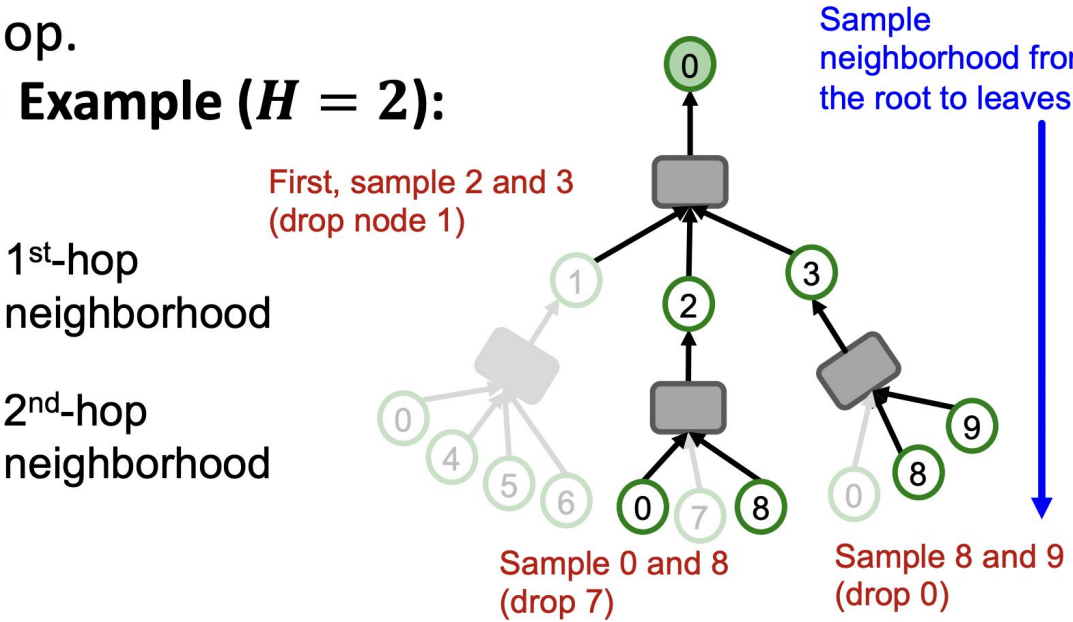
- Computation graph becomes **exponentially large** with respect to the layer size K.
- Computation graph explodes when it hits a **hub node** (high-degree node).
- **Solution**: make the computational graph more compact!



Neighborhood Sampling

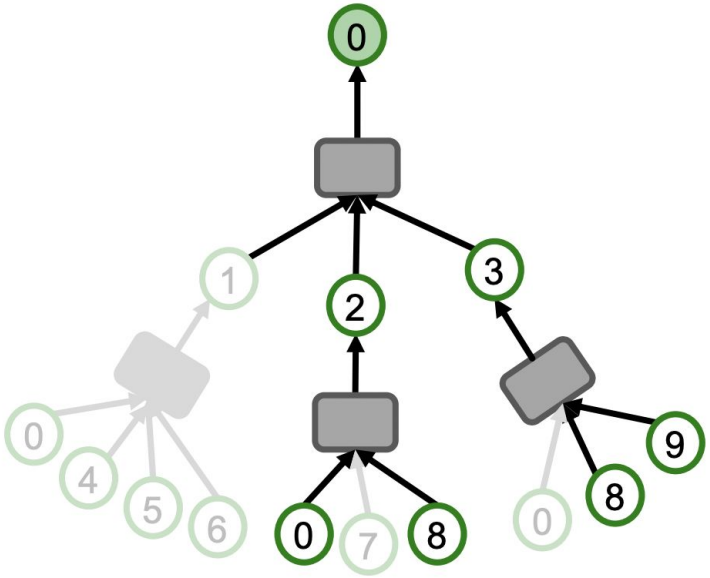
Key idea: **Construct the computational graph** by (randomly) sampling at most H neighbors at each hop.

- **Example ($H = 2$):**



Neighborhood Sampling

We can use the pruned computational graph to more efficiently compute node embeddings.



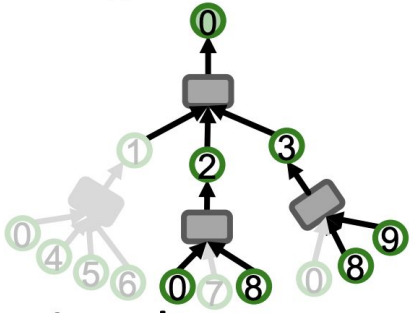
Neighborhood Sampling Algorithm

Neighbor sampling for K-layer GNN

- For $k = 1, 2, \ldots, K$:
  - For each node in k-hop neighborhood:
  - (Randomly) sample at most H_k neighbors:

1st-hop neighborhood Sample $H_1 = 2$ neighbors

2nd-hop neighborhood Sample $H_2 = 2$ neighbors



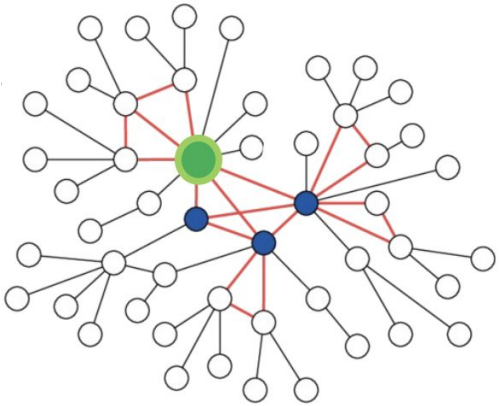
- K-layer GNN will at most involve $\prod_{k=1}^{K} H_k$ leaf nodes in comp. graph.

Remarks on Neighborhood Sampling (1/2)

- **Trade-off in sampling number H**
  - Smaller H leads to more efficient neighbor aggregation, but results are less stable training due to the **larger variance** in neighbor aggregation.
- **Computational time**
  - Even with neighbor sampling, the size of the computational graph is still **exponential** with respect to number of GNN layers K.
  - Adding one GNN layer would make computation H times more expensive.

Remarks on Neighborhood Sampling (1/2)

- **How to sample the nodes**
  - **Random Sampling**: fast but many times not optimal (may sample many “unimportant” nodes)
  - **Random Walk with Restarts**:
    - Natural graphs are “scale free”, sampling random neighbors, samples many low degree “leaf” nodes.
    - Strategy to sample important nodes:
      - Compute Random Walk with Restarts score R_i starting at the **green** node
      - At each level sample H neighbors i with the highest R_i
    - This strategy works much better in practice.



Summary: Neighborhood Sampling

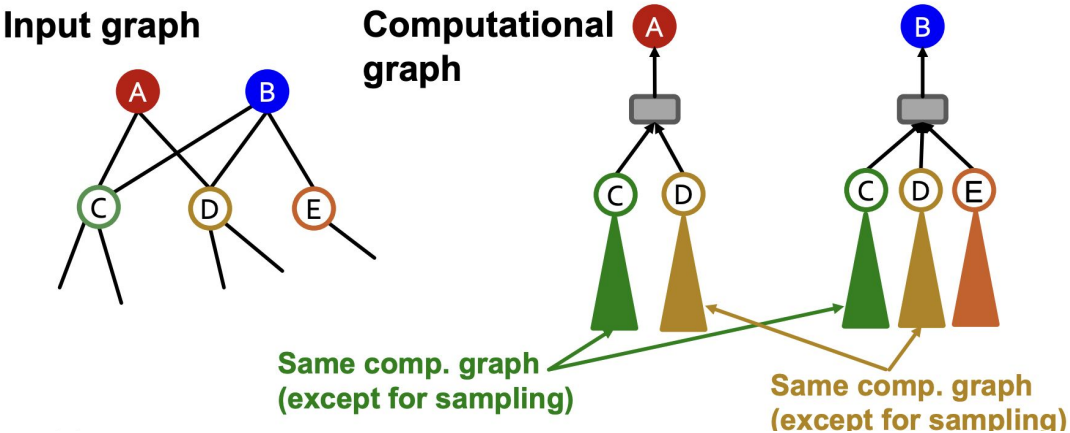
- A computational graph is constructed for each node in a mini-batch.
- In neighbor sampling, the comp. graph is pruned/sub-sampled to increase computational efficiency.
- The pruned comp. graph is used to generate a node embedding.

- However, computational graphs can still become large, especially for GNNs with many message-passing layers.

Cluster-GCN

Issue with Neighborhood Sampling

- The size of computational graph becomes exponentially large w.r.t. the #GNN layers.
- Computation is **redundant**, especially when nodes in a mini-batch share many neighbors.



Recall: Full-batch GNN

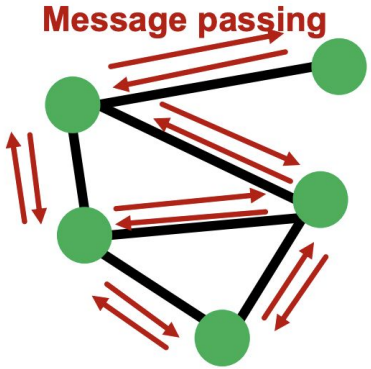
- In full-batch GNN implementation, all the node embeddings are updated together using embeddings of the previous layer.

Update for all $v \in V$

Message

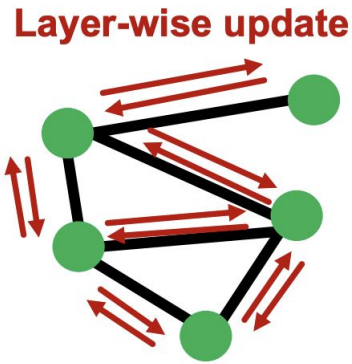
$$h_v^{(\ell)} = COMBINE\left(h_v^{(\ell-1)}, AGGR\left(\left\{h_u^{(\ell-1)}\right\}_{u \in N(v)}\right)\right)$$

- In each layer, only 2*#(edges) messages need to be computed.
- For K-layer GNN, only $2K$*#(edges) messages need to be computed.
- GNN's entire computation is only **linear** in #(edges) and #(GNN layers). **Fast!**



Insight from Full-batch GNN

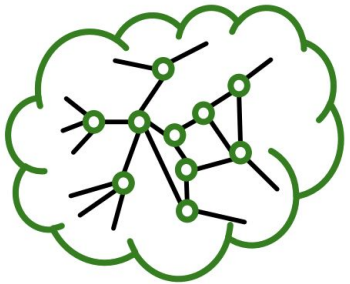
- The layer-wise node embedding update allows the re-use of embeddings from the previous layer.
- This significantly reduces the computational redundancy of neighbor sampling.
  - Of course, the layer-wise update is not feasible for a large graph due to limited GPU memory.
    - Requires putting the entire graph and features on GPU.



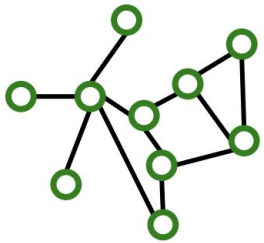
Subgraph Sampling

- **Key idea**: We can **sample a small subgraph** of the large graph and then perform the efficient **layer-wise** node embeddings update over the subgraph.

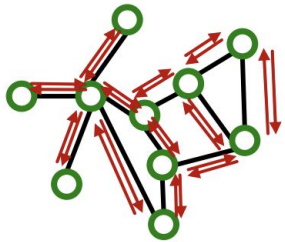
Large graph



Sampled subgraph
(small enough to
be put on a GPU)



Layer-wise
node embeddings
update on the GPU



Subgraph Sampling

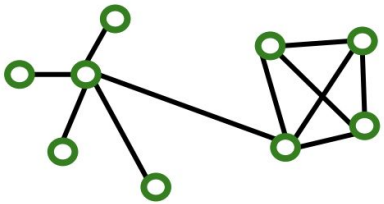
- **Key question: What subgraphs are good for training GNNs?**
- Recall: GNN performs node embedding by passing messages **via the edges**.
  - Subgraphs **should retain edge connectivity structure** of the original graph as much as possible.
  - This way, the GNN over the subgraph generates embeddings closer to the GNN over the original graph.

Question 2

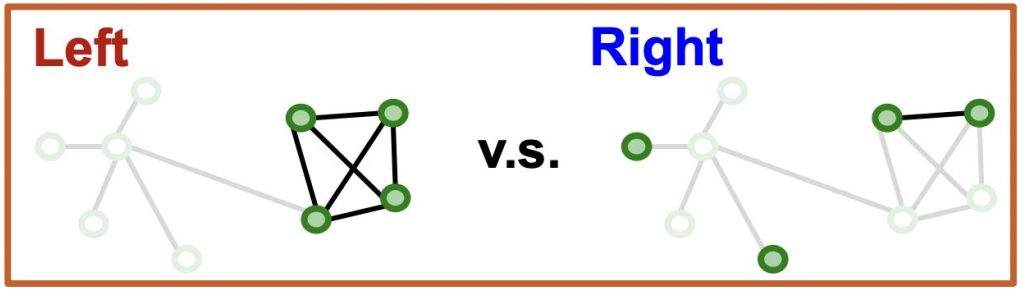
Subgraph Sampling: Case Study

- **Which subgraph is good for training GNN?**

Original graph

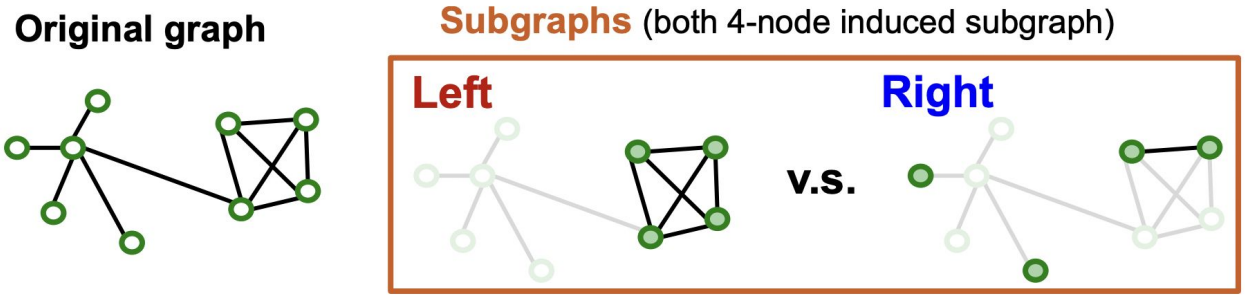


Subgraphs (both 4-node induced subgraph)



Subgraph Sampling: Case Study

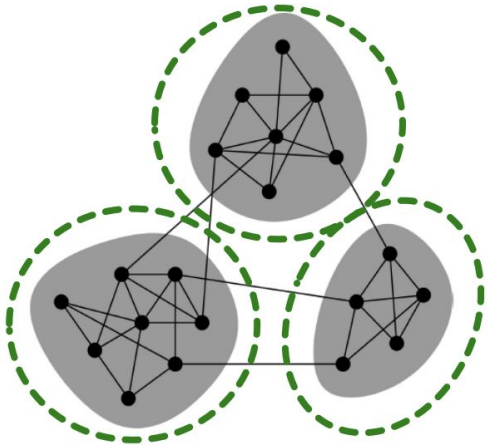
- **Which subgraph is good for training GNN?**



- **Left subgraph** retains the essential community structure among the 4 nodes → **Good**
- **Right subgraph** drops many connectivity patterns, even leading to isolated nodes → **Bad**

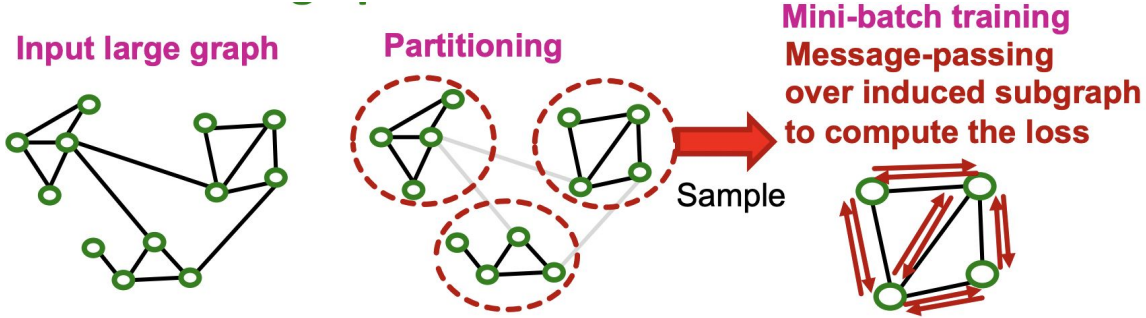
Exploiting Community Structure

- **Real-world graph exhibits community structure**
  - A large graph can be decomposed into many small communities
- **Key insight [Chiang et al. KDD 2019]:**
  - Sample a community as a subgraph. Each subgraph retains essential local connectivity pattern of the original graph.



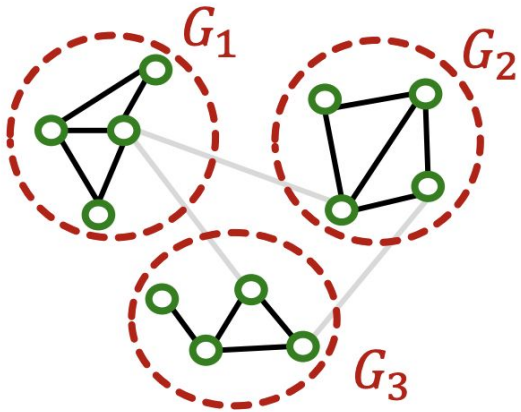
Cluster-GCN: Overview

- We first introduce “vanilla” Cluster-GCN.
- **Cluster-GCN** consists of two steps:
  1. **Pre-processing:** Given a large graph, partition it into groups of nodes (i.e., subgraphs).
  2. **Mini-batch training:** Sample one node group at a time. Apply GNN’s message passing over the **induced subgraph**.



Cluster-GCN: Pre-Processing

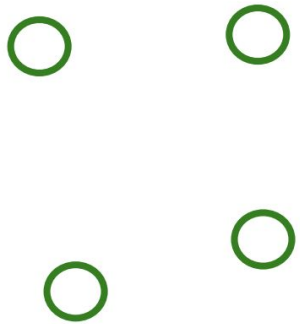
- Given a large graph $G = (V, E)$, **partition its nodes V into C groups: $V_1, ..., V_C$.**
  - We can use any scalable community detection methods, e.g., Louvain, METIS [Karypis et al. SIAM 1998].
- $V_1, ..., V_C$ **induces** C subgraphs, $G_1, ..., G_c$,
  - Recall: $G_C \equiv (V_C, E_C)$, where $E_C = \{(u, v) \mid u, v \in V_c\}$
  - Notice: Between-group edges are not included in $G_1, ..., G_c$



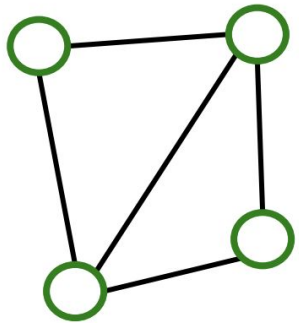
Cluster-GCN: Mini-Batch Training

- For each mini-batch, randomly sample a node group V_C.
- Construct induced subgraph $G_C = (V_C, E_C)$

Sampled node group V_c

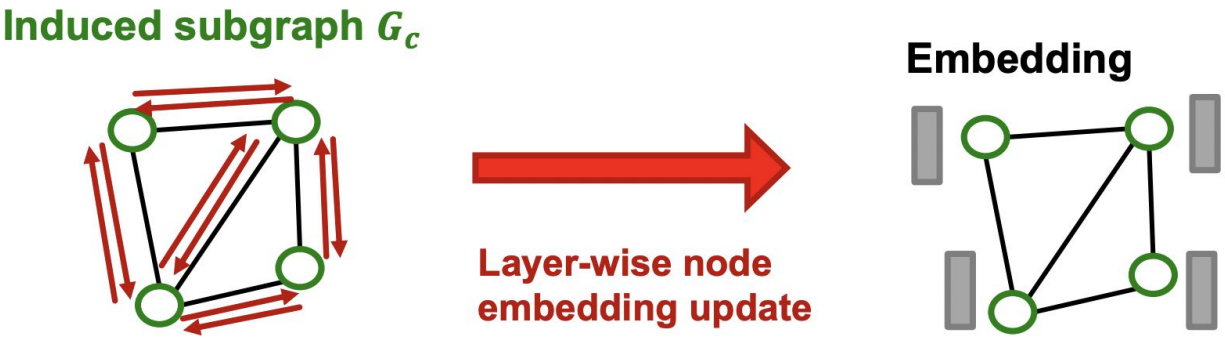


Induced subgraph G_c



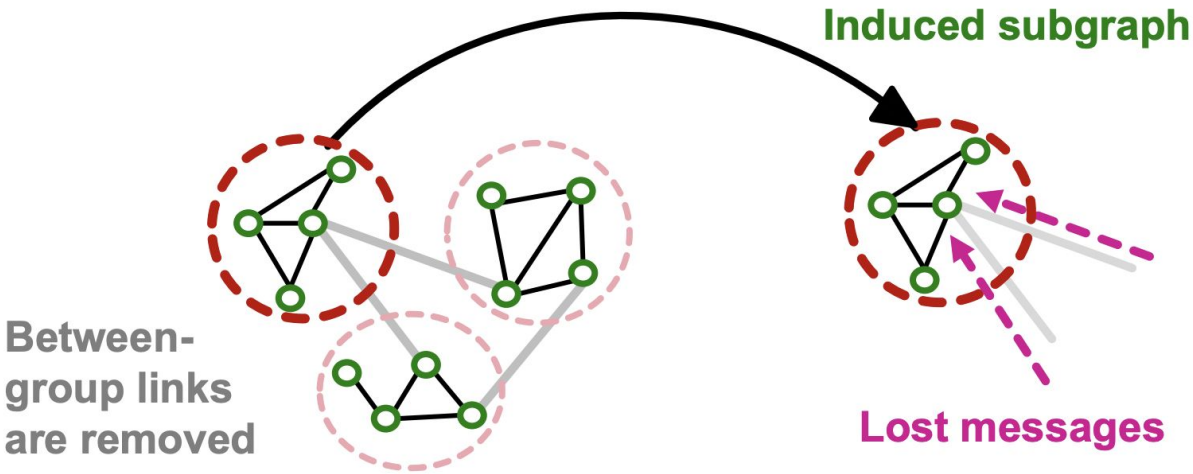
Cluster-GCN: Mini-Batch Training

- Apply GNN's **layer-wise node update** over G_c to obtain embedding h_v for each node $v \in V_c$.
- Compute the loss for each node $v \in V_c$ and take average: $\ell_{sub}(\boldsymbol{\theta}) = (1/|V_c|) \cdot \sum_{v \in V_c} \ell_v(\boldsymbol{\theta})$
- Update params: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \ell_{sub}(\boldsymbol{\theta})$



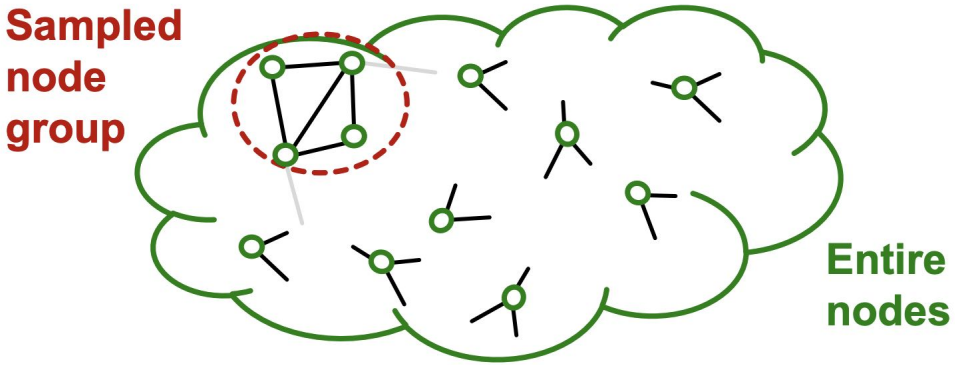
Issues with Cluster-GCN (1/3)

- The induced subgraph **removes** between group links.
- As a result, **messages from other groups will be *lost during message passing*, which could hurt the GNN's performance**.



Issues with Cluster-GCN (2/3)

- Graph community detection algorithm puts **similar nodes together in the same group**.
- **Sampled node group** tends to only cover the small-concentrated portion of the entire data.



Issues with Cluster-GCN (3/3)

- Sampled nodes are not diverse enough to be represent the entire graph structure:
  - As a result, the gradient averaged over the sampled nodes,

$$\frac{1}{|V_c|}\sum_{v\in V_c}\ell_v(\boldsymbol{\theta})$$

    , becomes unreliable.
    - Fluctuates a lot from a node group to another.
    - In other words, the gradient has high variance.
  - Leads to slow convergence of SGD

Cluster-GCN Summary

- Cluster-GCN first **partitions the entire nodes into a set of small node groups**.
- At each mini-batch, multiple node groups are sampled, and their nodes are aggregated.
- **GNN performs layer-wise node embeddings update over the induced subgraph.**
- Cluster-GCN is more computationally efficient than neighbor sampling, especially when #(GNN layers) is large.
- But Cluster-GCN leads to *systematically biased* gradient estimates (due to missing cross-community edges)

Simplifying GNN Architecture

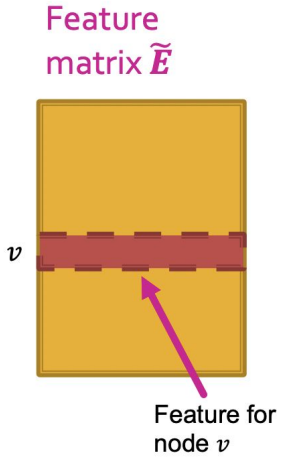
Reduce the number of computations performed in training

Roadmap of Simplifying GCN

- **We start from Graph Convolutional Network** (GCN) [Kipf & Welling ICLR 2017].
- We simplify GCN (“SimplGCN”) by **removing the non-linear activation** from the GCN [Wu et al. ICML 2019].
  - SimplGCN demonstrated that the performance on benchmark is not much lower by the simplification.
  - Simplified GCN turns out to be extremely scalable by the model design.
  - The simplification strategy is very similar to the one used by LightGCN for recommender systems.

Simplified GCN: “SimplGCN”

- Let $\widetilde{\boldsymbol{E}} = \widetilde{\boldsymbol{A}}^K \boldsymbol{E}$ be pre-processed feature matrix.
  - Each row stores the pre-processed feature for each node.
  - $\widetilde{\boldsymbol{E}}$ can be used as input to any scalable ML models (e.g., linear model, MLP).
- SimplGCN empirically shows learning a linear model over $\widetilde{\boldsymbol{E}}$ often gives performance comparable to GCN!



Comparison to Other Methods

- Compared to neighbor sampling and cluster-GCN, **SimplGCN is much more efficient**.
  - **SimplGCN computes $\widetilde{E}$ only once at the beginning.**
    - **The pre-processing (sparse matrix vector product, ($E \leftarrow \widetilde{A}\,E$) can be performed efficiently on CPU.**
  - Once $\widetilde{E}$ is obtained, getting an embedding for node v only takes **constant time!**
    - Just look up a row for node v in $\widetilde{E}$.
    - No need to build a computational graph or sample a subgraph.
- But the model is **less** expressive (next).

Issues with SimplGCN

- Compared to the original GNN models, **SimplGCN's expressive power is limited due to the lack of non-linearity in generating node embeddings**.

Performance of SimplGCN

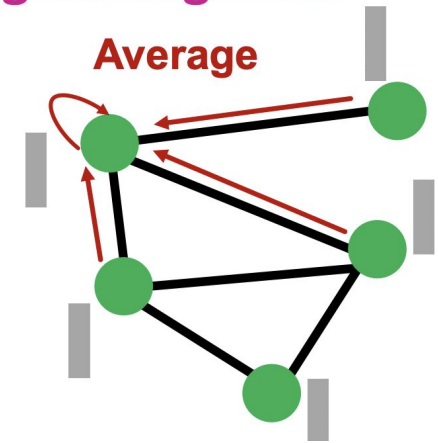
- Surprisingly, in semi-supervised node classification benchmark, SimplGCN works comparably to the original GNNs despite being less expressive.
- Why?
  - Graph homophilly!

Graph Homophilly

- Many node classification tasks exhibit homophily structure, i.e., **nodes connected by edges tend to share the same target labels**.
- **Examples**:
  - Paper category classification in paper-citation network
    - Two papers tend to share the same category if one cites another.
  - Movie recommendation for users in social networks
    - Two users tend to like the same movie if they are friends in a social network.

When Does Simplified GCN Work?

- Recall the preprocessing step of the simplified GCN: **Do $E \leftarrow \tilde{A}\,E$ for K times.**
  - **E is node feature matrix $E = X$**
- Pre-processed features are obtained **by iteratively averaging their neighboring node features.**



- **As a result, nodes connected by edges tend to have similar pre-processed features.**

When Does Simplified GCN Work?

- **Premise**: Model uses the pre-processed node features to make prediction.
- Nodes connected by edges tend to get similar pre-processed features.
- **Nodes connected by edges tend to be predicted the same labels by the model**
- Simplified SGC's prediction aligns well with the graph homophily in many node classification benchmark datasets.

SimplGCN: Summary

- **Simplified GCN removes non-linearity in GCN and reduces to the simple pre-processing of node features.**
- Once the pre-processed features are obtained, scalable mini-batch SGD can be directly applied to optimize the parameters.
- Simplified GCN works surprisingly well in node classification benchmark.
  - The feature pre-processing aligns well with graph homophily in real-world prediction tasks.

Recap of GNN Scaling

1. **Applications**: some use-cases (e.g., social networks, fluid dynamics) need large-scale GNNs
2. **Scalability Challenges**: Standard mini-batch SGD doesn't work well for GNN training
3. **Sampling**: neighborhood sampling, Cluster-GCN can be used to pick a *part* of the graph to train on
4. **Architecture**: changing model architecture, like simplGCN, can reduce computation and maintain good results

References

- **Lectures:**
  - Stanford CS224W (2022) - Lectures and Colab 5 on Scaling GNNs
- **Papers:**
  - Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks (2019)
  - GraphSAINT: Graph Sampling Based Inductive Learning Method (2020)
  - Distributed Graph Neural Network Training: A Survey (2022)
- **Articles**:
  - https://diplodoc.medium.com/graph-neural-networks-gnn-enable-the-study-of-drug-interactions-and-the-discovery-of-new-6cb94ab82b53