IT5429E-1-24 (24.1A01)(Fall 2024): Graph Analytics for Big Data

Week 7: Recommender Systems & Scaling GNNs

Instructor: Thanh H. Nguyen

Many slides are adapted from https://web.stanford.edu/class/cs224w/

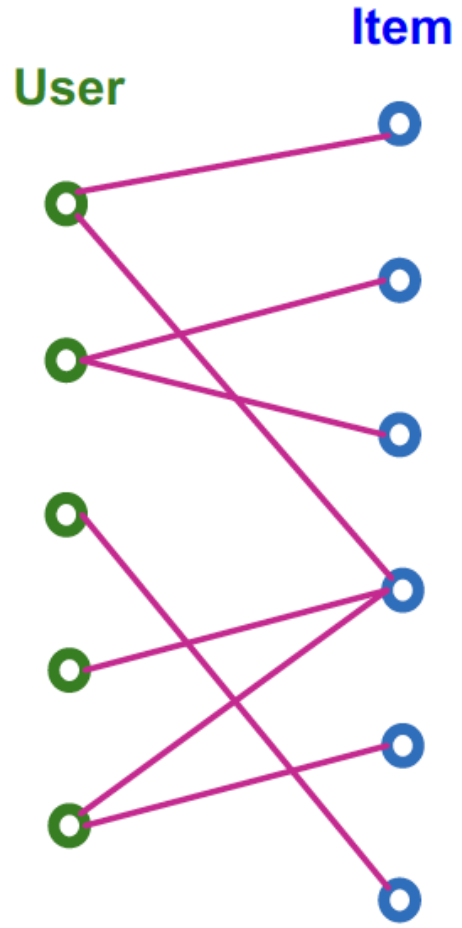
Preliminary of Recommendation

- Information Explosion in the era of Internet
  - 10K+ movies in Netflix
  - 12M products in Amazon (350m on Marketplace)
  - 70M+ music tracks in Spotify
  - 10B+ videos on YouTube
  - 200B+ pins (images) in Pinterest

- Personalized recommendation (i.e., suggesting a small number of interesting items for each user) is critical for users to effectively explore the content of their interest.

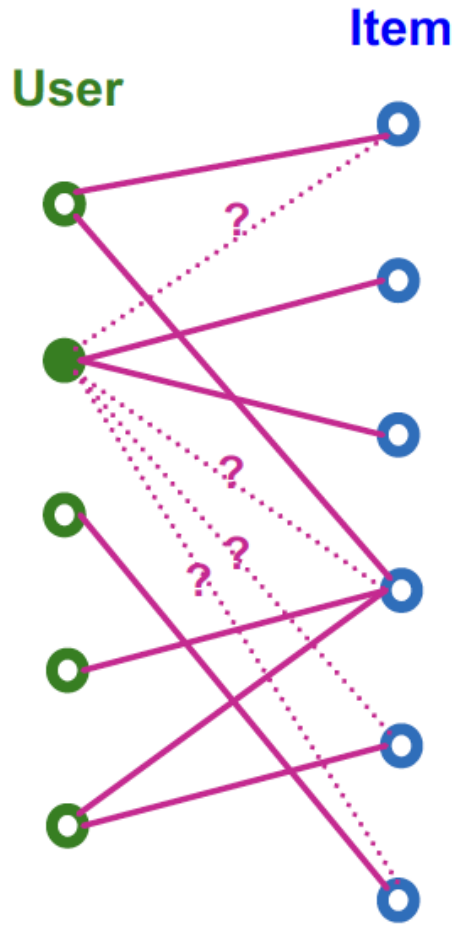
Recommender System as a Graph

- Recommender system can be naturally modeled as a bipartite graph
  - A graph with two node types: users and items.
  - Edges connect users and items
    - Indicates user-item interaction (e.g., click, purchase, review etc.)
    - Often associated with timestamp (timing of the interaction).



Recommendation Task

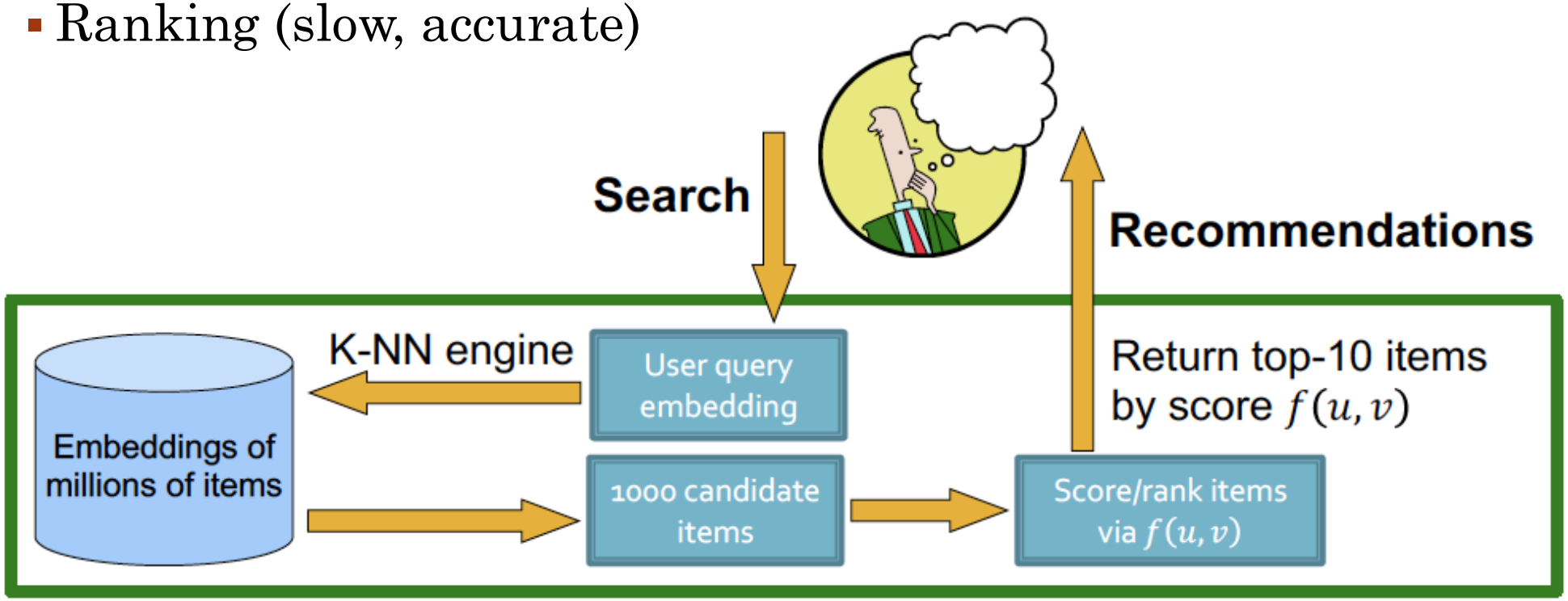
- Given
  - Past user-item interactions
- Task
  - Predict new items each user will interact in the future.
  - Can be cast as link prediction problem.
    - Predict new user-item interaction edges given the past edges.
  - For $u \in \boldsymbol{U}$, $v \in \boldsymbol{V}$, we need to get a real-valued score $f(u, v)$.



Modern Recommender System

- Problem: Cannot evaluate $f(u, v)$ for every user u – item v pair.
- Solution: 2-stage process:
  - Candidate generation (cheap, fast)
  - Ranking (slow, accurate)

Example: $f(u, v) = z_u \cdot z_v$



Top-K Recommendation

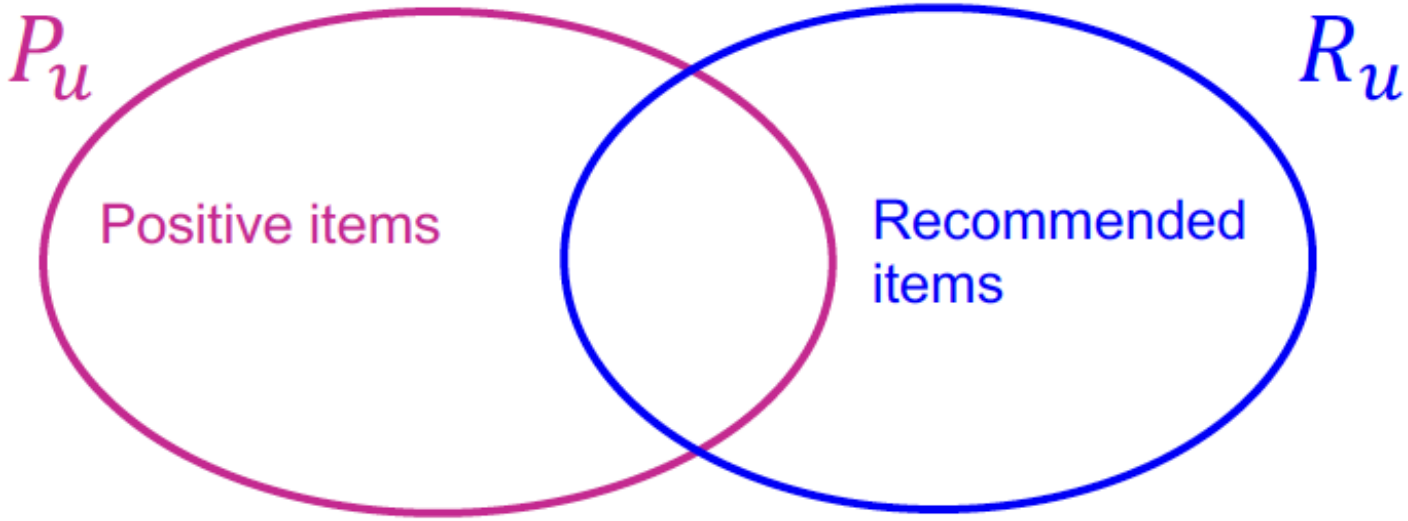
- For each user, we recommend K items.
  - For recommendation to be effective, ***K*** needs to be much smaller than the total number of items (up to billions)
  - K is typically in the order of 10—100.

- The goal is to include as many positive items as possible in the top-K recommended items.
  - Positive items = Items that the user will interact with in the future.

- Evaluation metric: Recall@K (defined next)

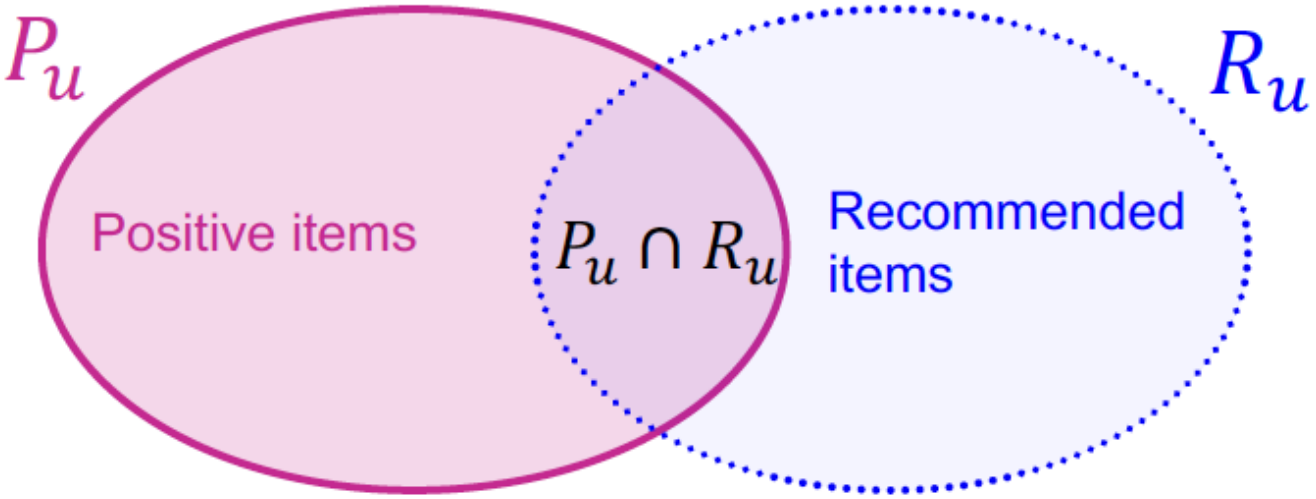
Evaluation Metric: Recall@K (1)

- For each user $\boldsymbol{u}$,
  - Let P_u be a set of positive items the user will interact in the future.
  - Let R_u be a set of items recommended by the model.
    - In top-K recommendation, $|R_u| = K$.
    - Items that the user has already interacted are excluded.



Evaluation Metric: Recall@K (2)

- Recall@K for user u is $\frac{|P_u \cap R_u|}{|P_u|}$
  - Higher value indicates more positive items are recommended in top-K for user u.



- The final Recall@K is computed by averaging the recall values across all users.

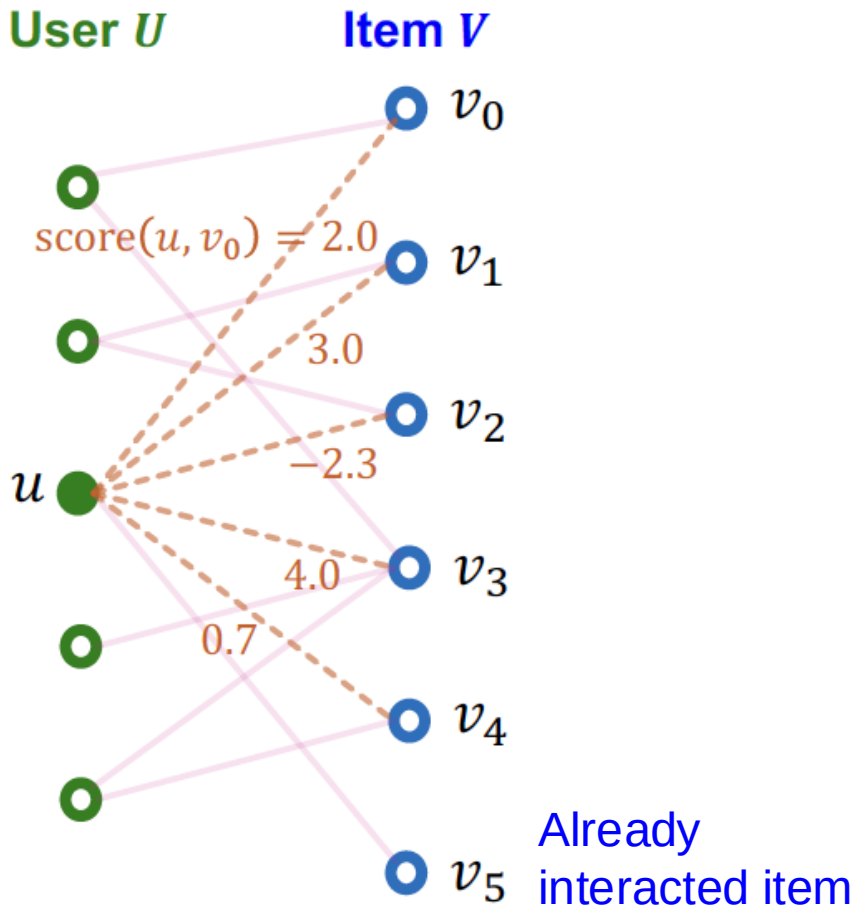
Recommender Systems: Embedding-based Models

Notation:

- Notation:
  - $\boldsymbol{U}$: A set of all users
  - $\boldsymbol{V}$: A set of all items
  - $\boldsymbol{E}$: A set of observed user-item interactions
  - $\boldsymbol{E} = \{(u, v) \mid u \in \boldsymbol{U}, v \in \boldsymbol{V}, u \text{ interacted with } v\}$

Score Function

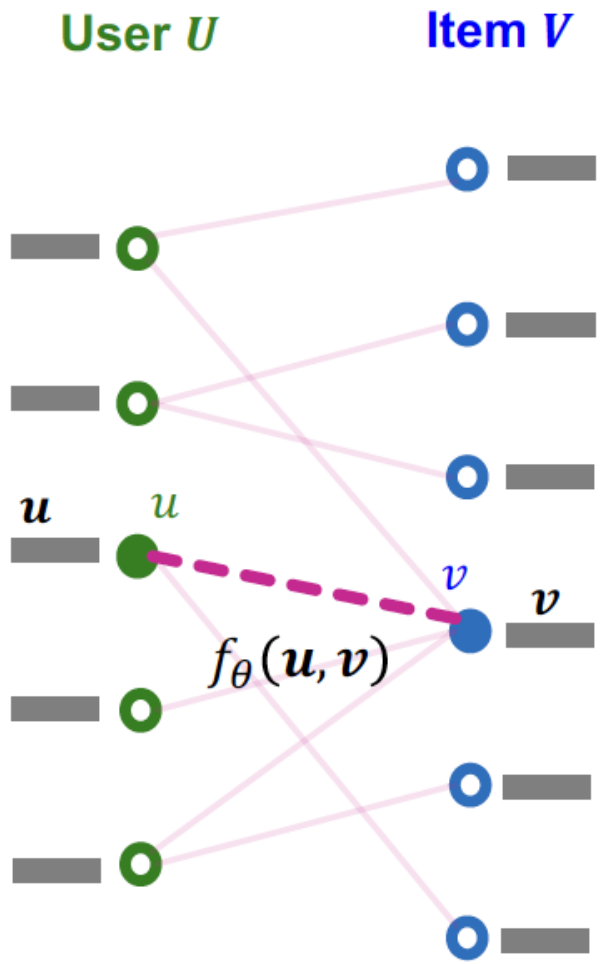
- To get the top-K items, we need a score function for user-item interaction:
  - For $u \in \boldsymbol{U}$, $v \in \boldsymbol{V}$, we need to get a real-valued scalar score(u, v).
  - $\boldsymbol{K}$ items with the largest scores for a given user u (excluding already interacted items) are then recommended).



For K = 2, recommended items for user u would be $\{v_1, v_3\}$.

Embedding-based Models

- We consider embedding-based models for scoring user-item interactions.
  - For each user $u \in \boldsymbol{U}$, let $\boldsymbol{u} \in \mathbb{R}^{\mathrm{D}}$ be its D-dimensional embedding.
  - For each item $v \in \boldsymbol{V}$, let $\boldsymbol{v} \in \mathbb{R}^{\mathrm{D}}$ be its D-dimensional embedding.
  - Let $f_\theta(\cdot,\cdot): \mathbb{R}^{\mathrm{D}} \times \mathbb{R}^{\mathrm{D}} \rightarrow \mathbb{R}$ be a parametrized function.
  - Then, score $score(u, v) = f_\theta(u, v)$



Training Objective

- Embedding-based models have three kinds of parameters:
  - An encoder to generate user embeddings $u \in U$
  - An encoder to generate item embeddings $v \in V$
  - Score function $f_\theta(\cdot,\cdot)$

- Training objective: Optimize the model parameters to achieve high recall@K on seen (i.e., training) user-item interactions
  - We hope this objective would lead to high recall@K on unseen (i.e., test) interactions.

Surrogate Loss Functions

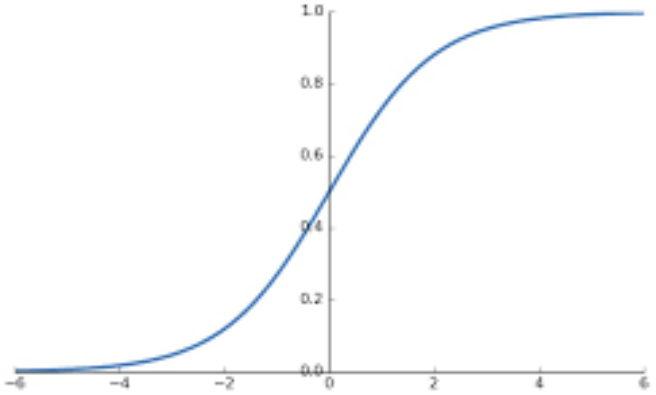
- The original training objective (recall@K) is not differentiable.
  - Cannot apply efficient gradient-based optimization.

- Two surrogate loss functions are widely-used to enable efficient gradient-based optimization.
  - Binary loss
  - Bayesian Personalized Ranking (BPR) loss

- Surrogate losses are differentiable and should align well with the original training objective.

Binary Loss (1)

- Define positive/negative edges
  - A set of positive edges ***E*** (i.e., observed/training user-item interactions)
- A set of negative edges $E_{neg} = \{(u, v) | (u, v) \notin E, u \in U, v \in V\}$
- Define sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)}$
- Maps real-valued scores into binary likelihood scores, i.e., in the range of [0,1].



Binary Loss (2)

- Binary loss: Binary classification of positive/negative edges using $\sigma(f_\theta, \boldsymbol{u}, \boldsymbol{v})$:

$$-\frac{1}{|E|}\sum_{(u,v)\in E}\log\Big(\sigma\big(f_\theta(u,v)\big)\Big) - \frac{1}{|E_{neg}|}\sum_{(u,v)\in E_{neg}}\log\Big(1-\sigma\big(f_\theta(u,v)\big)\Big)$$

During training, these terms can be approximated using mini-batch of positive/negative edges

- Binary loss pushes the scores of positive edges higher than those of negative edges.
  - This aligns with the training recall metric since positive edges need to be recalled.

Issue with Binary Loss (1)

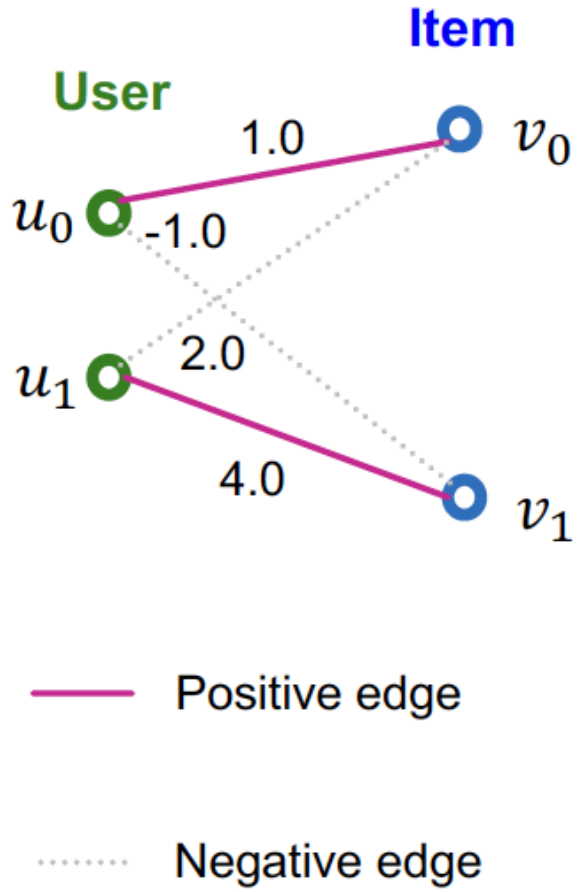
- Issue: In the binary loss, the scores of ALL positive edges are pushed higher than those of ALL negative edges.

- This would unnecessarily penalize model predictions even if the training recall metric is perfect.

- Why? (example in the next slide)

Issue with Binary Loss (2)

- Let's consider the simplest case:
  - Two users, two items
  - Metric: Recall@1.
  - A model assigns the score for every user-item pair (as shown in the right).



- Training Recall@1 is 1.0 (perfect score), because v_0 (resp. v_1) is correctly recommended to u_0 (resp. v_1)

- However, the binary loss would still penalize the model prediction because the negative (u_1, v_0) edge gets the higher score than the positive edge (u_0, v_0)

Issue with Binary Loss (3)

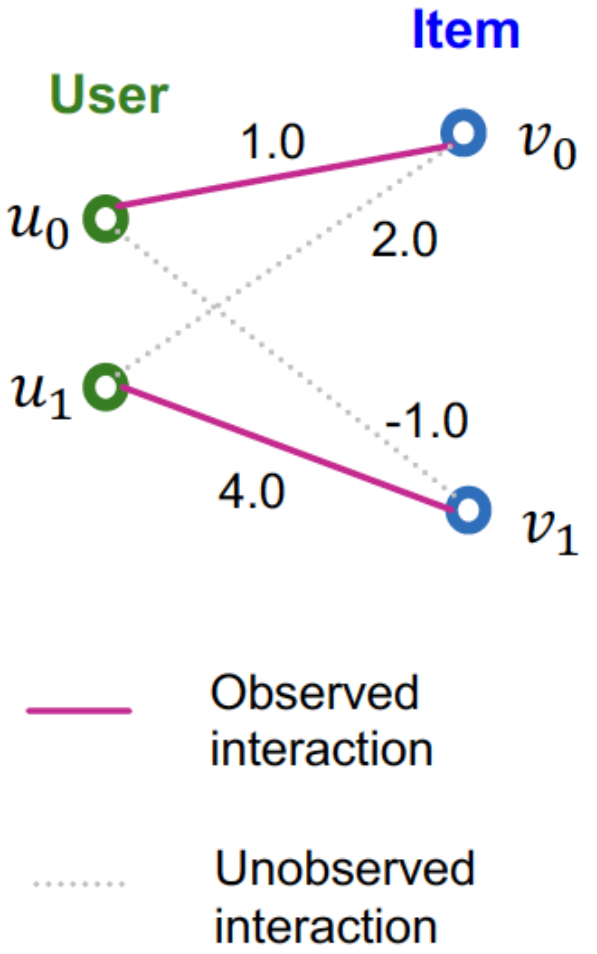
- Key insight: The binary loss is non-personalized in the sense that the positive/negative edges are considered across ALL users at once.

- However, the recall metric is inherently personalized (defined for each user).
  - The non-personalized binary loss is overly-stringent for the personalized recall metric.

Desirable Surrogate Loss

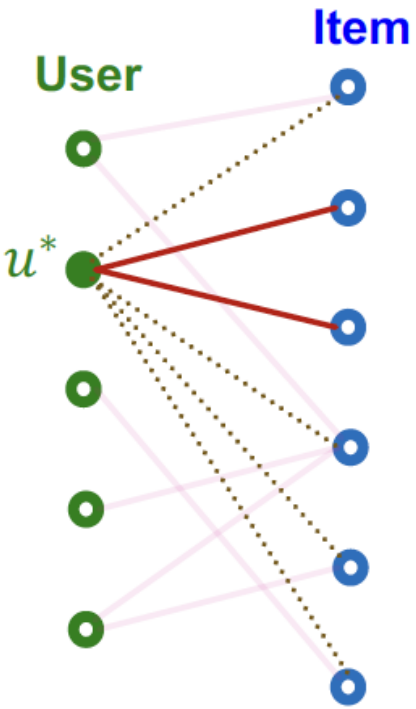
- Lesson learned: Surrogate loss function should be defined in a personalized manner.
  - For each user, we want the scores of positive items to be higher than those of the negative items
  - We do not care about the score ordering across users.

- Bayesian Personalized Ranking (BPR) loss achieves this!



Loss Function: BPR Loss (1)

- Bayesian Personalized Ranking (BPR) loss is a personalized surrogate loss that aligns better with the recall@K metric.



- For each user $u^* \in \boldsymbol{U}$, define the rooted positive/negative edges as
  - Positive edges rooted at u^*
    - $\boldsymbol{E}(u^*) \equiv \{(u^*, v) \mid (u^*, v) \in \boldsymbol{E}\}$
  - Negative edges rooted at u^*
    - $\boldsymbol{E}_{\text{neg}}(u^*) \equiv \{(u^*, v) \mid (u^*, v) \in \boldsymbol{E}_{\text{neg}}\}$

Loss Function: BPR Loss (2)

- Training objective: For each user u^*, we want the scores of rooted positive edges $\boldsymbol{E}(u^*)$ to be higher than those of rooted negative edges $\boldsymbol{E_{neg}}(u^*)$.
  - Aligns with the personalized nature of the recall metric.
- BPR Loss for user $\boldsymbol{u}^*$:

Encouraged to be positive for each user
=positive edge score is higher than negative edge score

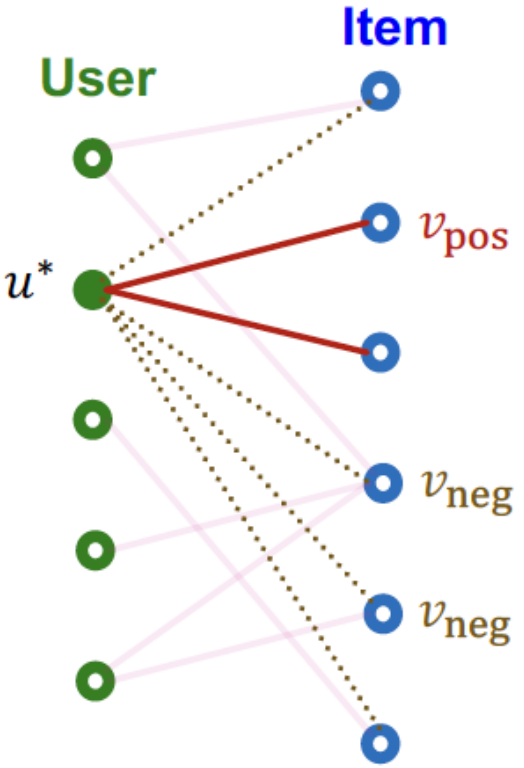
$$\text{Loss}(u^*) = \frac{1}{|E(u^*)| \cdot |E_{\text{neg}}(u^*)|} \underbrace{\sum_{(u^*, v_{\text{pos}}) \in E(u^*)} \sum_{(u^*, v_{\text{neg}}) \in E_{\text{neg}}(u^*)} -\log\left(\sigma\left(f_\theta(u^*, v_{\text{pos}}) - f_\theta(u^*, v_{\text{neg}})\right)\right)}$$

Can be approximated using a mini-batch

- Final BPR loss: $\frac{1}{|U|} \sum_{u^* \in U} \text{Loss}(u^*)$

Loss Function: BPR Loss (3)

- Mini-batch training for the BPR loss:
  - In each mini-batch, we sample a subset of users $U_{mini} \in U$.
  - For each user $u^* \in U_{mini}$, we sample one positive item v_{pos} and a set of sampled negative items $V_{neg} = \{v_{neg}\}$.
  - The mini-batch loss is computed as:



$$\frac{1}{|U_{\text{mini}}|} \sum_{u^* \in U_{\text{mini}}} \frac{1}{|V_{\text{neg}}|} \sum_{v_{\text{neg}} \in V_{\text{neg}}} -\log\left(\sigma\left(f_\theta(u^*, v_{\text{pos}}) - f_\theta(u^*, v_{\text{neg}})\right)\right)$$

Average over users in the mini-batch

Summary So Far

We have introduced

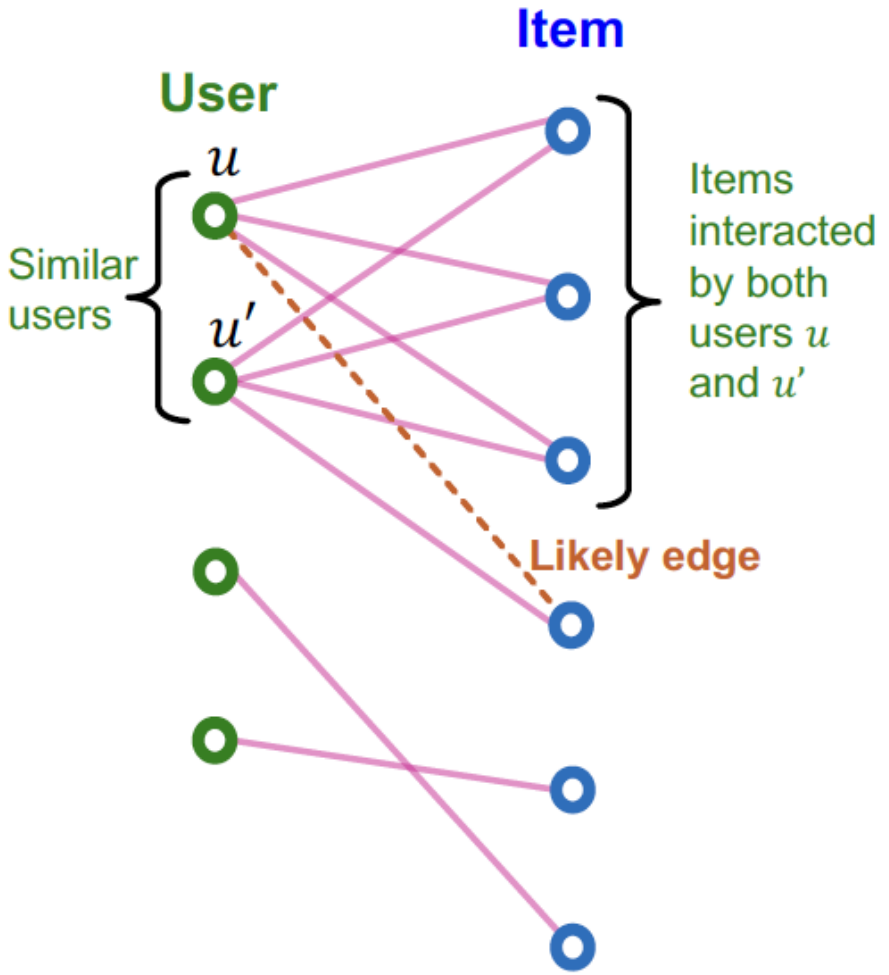
- Recall@K as a metric for personalized recommendation
- Embedding-based models
  - Three kinds of parameters to learn
    - user encoder to generate user embeddings
    - item encoder to generate item embeddings
    - score function to predict the user-item interaction likelihood.
  - Surrogate loss functions to achieve the high recall metric.
- Embedding-based models have achieved SoTA in recommender systems.
  - Why do they work so well?

Why Embedding Models Work?

Underlying idea:

- Collaborative filtering
  - Recommend items for a user by collecting preferences of many other similar users.
  - Similar users tend to prefer similar items.

- Key question: How to capture similarity between users/items?



Why Embedding Models Work?

- Embedding-based models can capture similarity of users/items!
- Low-dimensional embeddings cannot simply memorize all user-item interaction data.
- Embeddings are forced to capture similarity between users/items to fit the data.
- This allows the models to make effective prediction on unseen user-item interactions.

This Lecture: GNNs for RecSys

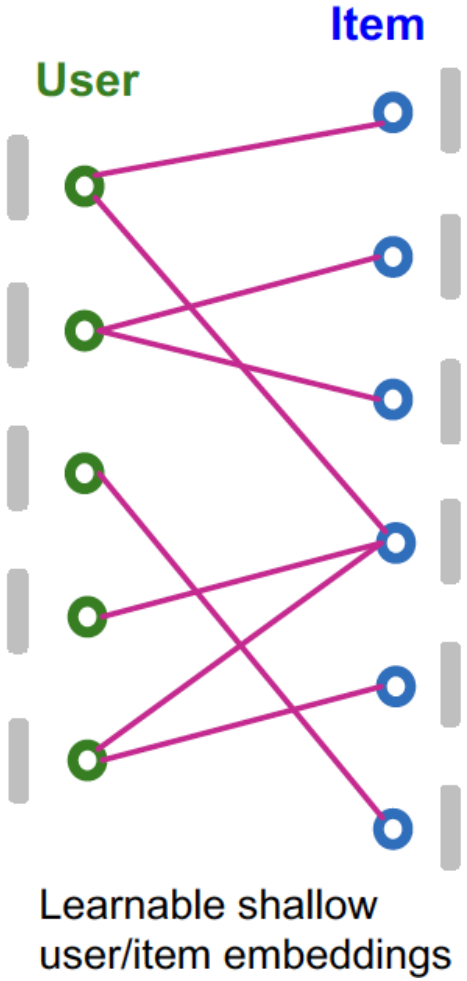
In this lecture, we teach two representative GNN approaches for recommender systems.

- (1) Neural Graph Collab. Filtering (NGCF) [Wang et al. 2019]
- (2) LightGCN [He et al. 2020]
  - Improve the conventional collaborative filtering models (i.e., shallow encoders) by explicitly modeling graph structure using GNNs.
  - Assumes no user/item features.
- PinSAGE [Ying et al. 2018]
  - Use GNNs to generate high-quality embeddings by simultaneously capturing rich node attributes (e.g., images) and the graph structure.

Neural Graph Collaborative Filtering

Conventional Collaborative Filtering

- Conventional collaborative filtering model is based on shallow encoders:
  - No user/item features.
  - Use shallow encoders for users and items:
    - For every $u \in \boldsymbol{U}$ and $v \in \boldsymbol{V}$, we prepare shallow learnable embeddings $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^{\mathrm{D}}$.
  - Score function for user u and item v is $f_\theta(u, v) = z_u^T z_v$



Limitations of Shallow Encoders

- The model itself does not explicitly capture graph structure
- The graph structure is only implicitly captured in the training objective.
- Only the first-order graph structure (i.e., edges) is captured in the training objective.
- High-order graph structure (e.g., K-hop paths between two nodes) is not explicitly captured.

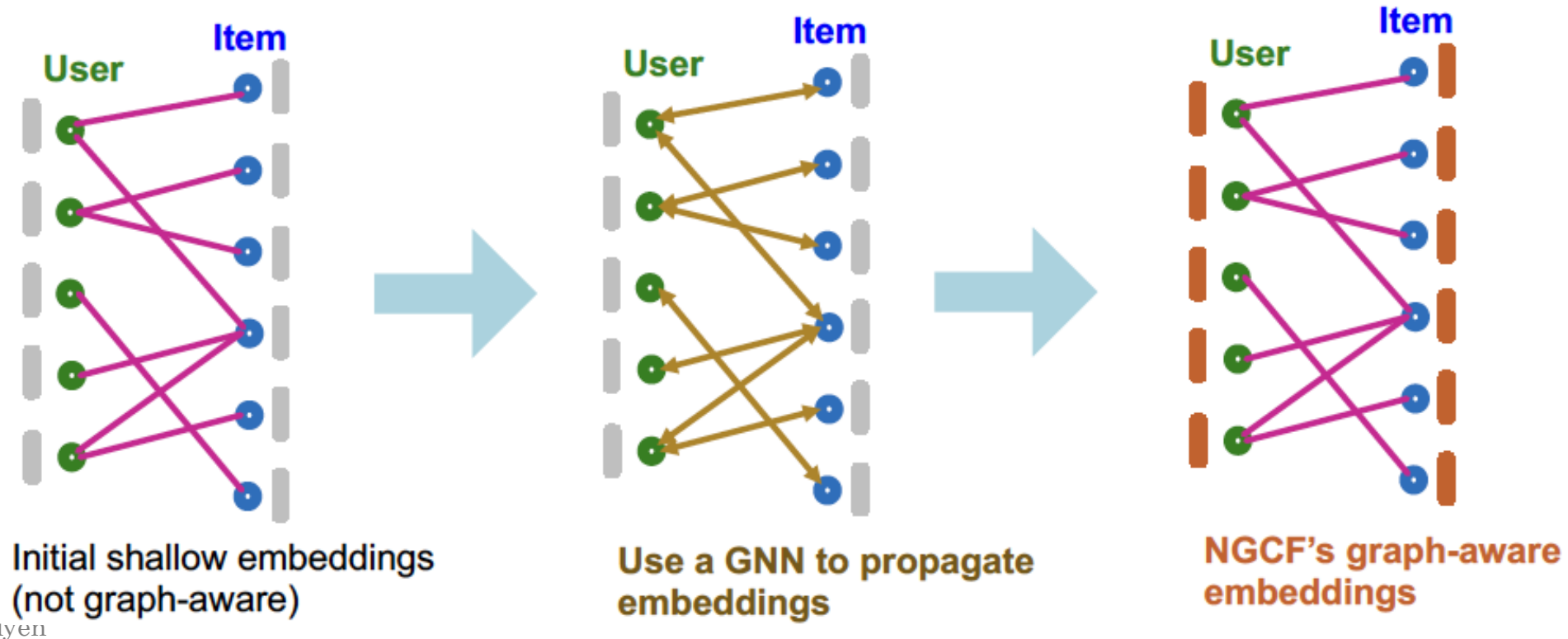
Motivation

- We want a model that…
  - explicitly captures graph structure (beyond implicitly through the training objective)
  - captures high-order graph structure (beyond the first-order edge connectivity structure)

- GNNs are a natural approach to achieve both!
  - Neural Graph Collaborative Filtering (NGCF) [Wang et al. 2019]
  - LightGCN [He et al. 2020]
    - A simplified and improved version of NGCF

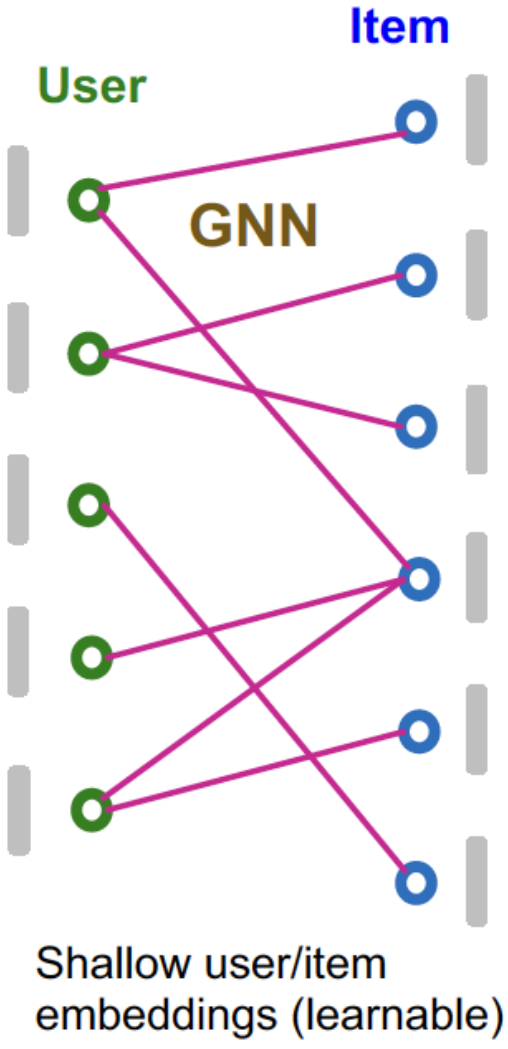
NGCF: Overview

- Neural Graph Collaborative Filtering (NGCF) explicitly incorporates high-order graph structure when generating user/item embeddings.
- Key idea: Use a GNN to generate graph-aware user/item embeddings.



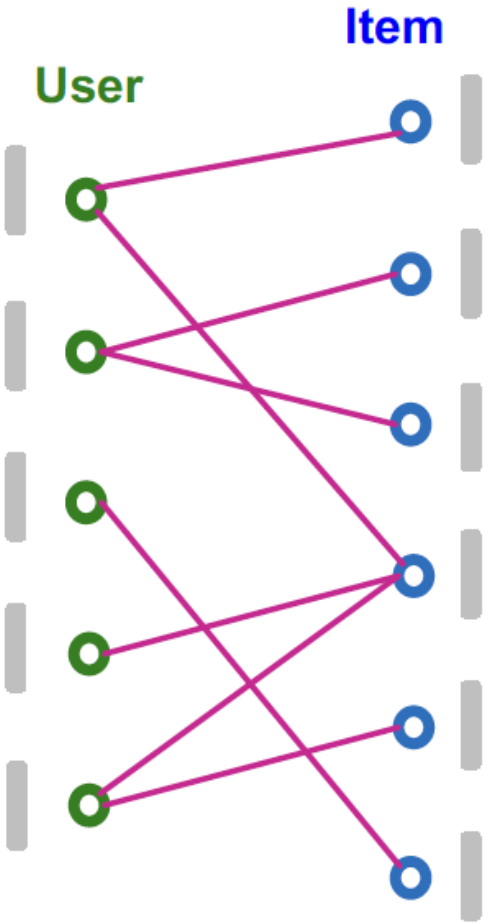
NGCF Framework

- Given: User-item bipartite graph.
- NGCF framework:
  - Prepare shallow learnable embedding for each node.
  - Use multi-layer GNNs to propagate embeddings along the bipartite graph.
    - High-order graph structure is captured.
  - Final embeddings are explicitly graph aware!
- Two kinds of learnable params are jointly learned:
  - Shallow user/item embeddings
  - GNN's parameters



Initial Node Embeddings

- Set the shallow learnable embeddings as the initial node features:
  - For every user $u \in \boldsymbol{U}$, set $h_u^{(0)}$ as the user's shallow embedding.
  - For every item $v \in \boldsymbol{V}$, set $h_v^{(0)}$ as the item's shallow embedding.



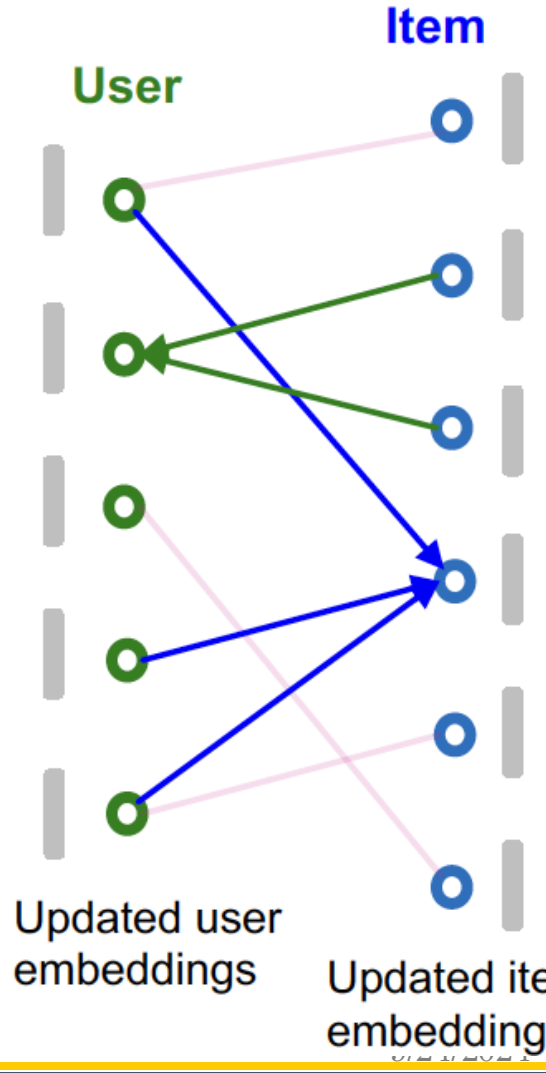
Neighbor Aggregation

- Iteratively update node embeddings using neighboring embeddings.

$$\boldsymbol{h}_v^{(k+1)} = \text{COMBINE}\left(\boldsymbol{h}_v^{(k)}, \text{AGGR}\left(\left\{\boldsymbol{h}_u^{(k)}\right\}_{u \in N(v)}\right)\right)$$

$$\boldsymbol{h}_u^{(k+1)} = \text{COMBINE}\left(\boldsymbol{h}_u^{(k)}, \text{AGGR}\left(\left\{\boldsymbol{h}_v^{(k)}\right\}_{v \in N(u)}\right)\right)$$

- High order graph structure is captured through iterative neighbor aggregation
- Different architecture choices are possible for AGGR and COMBINE.
  - AGGR(·) can be MEAN()
  - COMBINE($\boldsymbol{x}$, $\boldsymbol{y}$) can be ReLU Linear(Concat($\boldsymbol{x}$, $\boldsymbol{y}$))



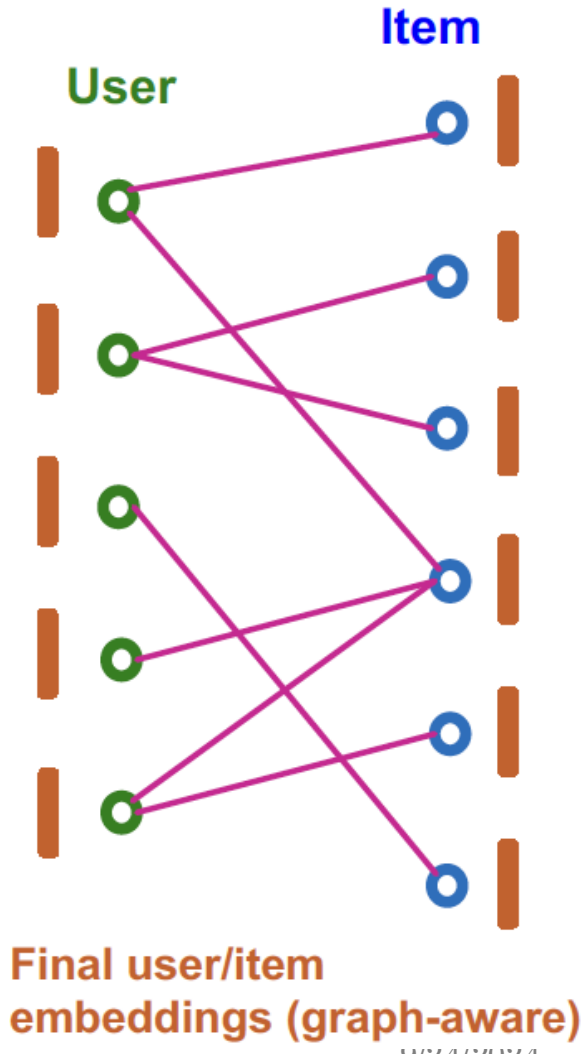
Final Embeddings and Score Function

- After K rounds of neighbor aggregation, we get the final user/item embeddings $h_u^{(K)}$ and $h_v^{(K)}$
- For all $u \in \boldsymbol{U}$, $v \in \boldsymbol{V}$, we set:

$$u = h_u^{(K)}, v = h_v^{(K)}$$

- Score function is the inner product

$$score(u, v) = u^T v$$



NGCF Summary

- Conventional collaborative filtering uses shallow user/item embeddings.
  - The embeddings do not explicitly model graph structure.
  - The training objective does not model high-order graph structure.

- NGCF uses a GNN to propagate the shallow embeddings.
  - The embeddings are explicitly aware of high-order graph structure.

LightGCN

LightGCN: Motivation (1)

- Recall: NGCF jointly learns two kinds of parameters:
  - Shallow user/item embeddings
  - GNN's parameters

- Observation: Shallow learnable embeddings are already quite expressive.
  - They are learned for every (user/item) node.
  - Most of the parameter counts are in shallow embeddings when N (#nodes) $\gg D$ (embedding dimensionality)
    - Shallow embeddings: $O(ND)$.
    - GNN: $O(D^2)$.
  - The GNN parameters may not be so essential for performance.

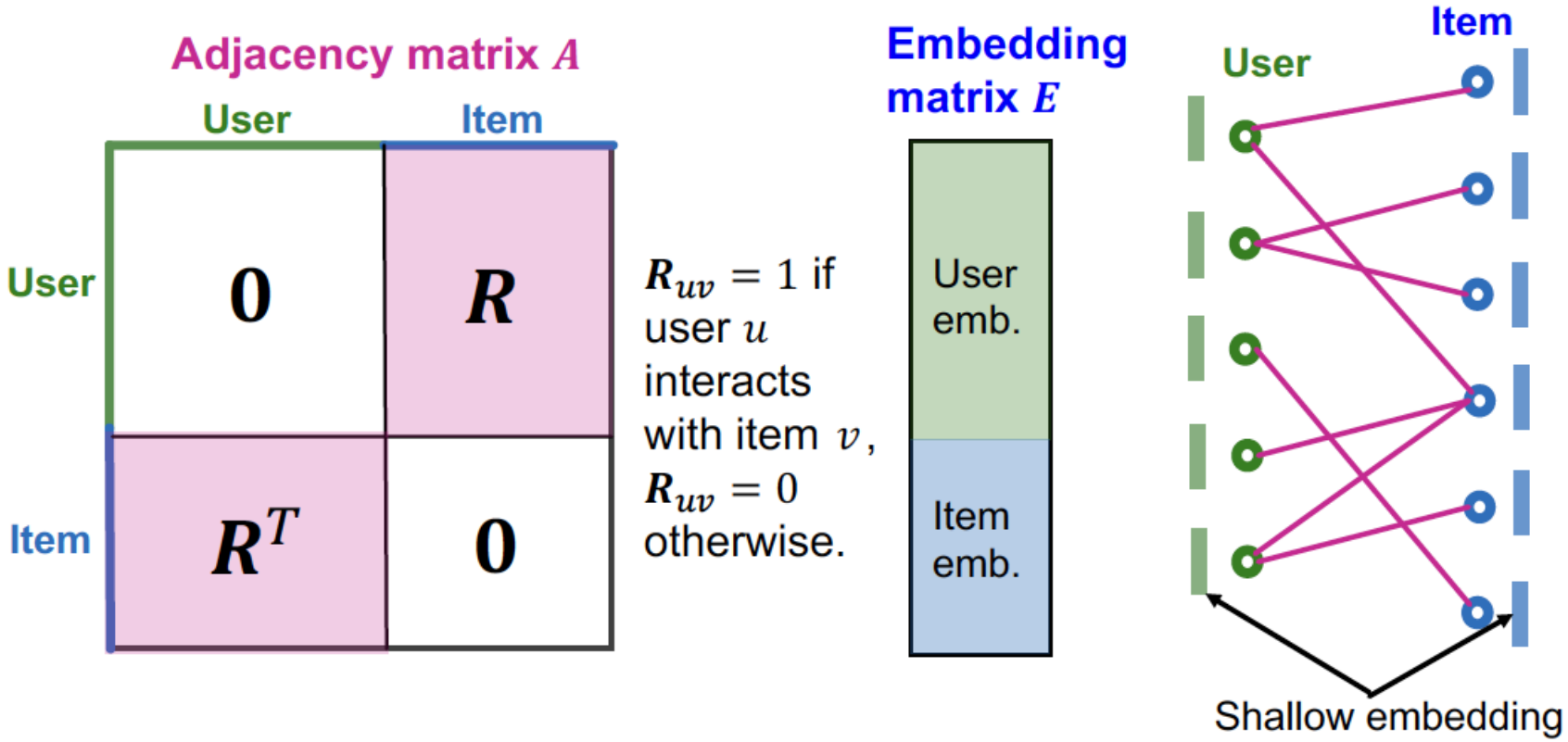
LightGCN: Motivation (2)

- Can we simplify the GNN used in NGCF (e.g., remove its learnable parameters)?
  - Answer: Yes!
  - Bonus: Simplification improves the recommendation performance!

- Overview of the idea:
  - Adjacency matrix for a bipartite graph
  - Matrix formulation of GCN
  - Simplification of GCN by removing non-linearity
    - Related: SGC for scalable GNN [Wu et al. 2019]

Adjacency and Embedding Matrices

- Adjacency matrix of a (undirected) bipartite graph.
- Shallow embedding matrix.



Matrix Formulation of GCN

- Define: The diffusion matrix
- Let $\boldsymbol{D}$ be the degree matrix of $\boldsymbol{A}$.
- Define the normalized adjacency matrix $\tilde{A}$ as

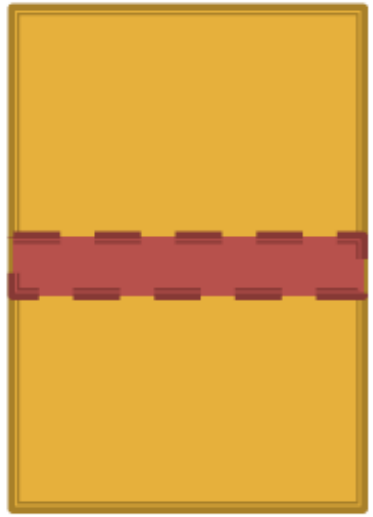
$$\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

- Let $\boldsymbol{E}^{(k)}$ be the embedding matrix at k-th layer.
- Each layer of GCN's aggregation can be written in a matrix form:

$$\boldsymbol{E}^{(k+1)} = \mathrm{ReLU}\left(\tilde{\boldsymbol{A}}\boldsymbol{E}^{(k)}\boldsymbol{W}^{(k)}\right)$$

Neighbor aggregation Learnable linear transformation

Matrix of node embeddings $\boldsymbol{E}^{(k)}$



Each row stores node embedding

Simplifying GCN (1)

- Simplify GCN by removing ReLU non-linearity:

$$\boldsymbol{E}^{(k+1)} = \tilde{\boldsymbol{A}}\boldsymbol{E}^{(k)}\boldsymbol{W}^{(k)}$$

Original idea from SGC [Wu et al. 2019]

- The final node embedding matrix is given as

$$\boldsymbol{E}^{(K)} = \tilde{\boldsymbol{A}}\,\boldsymbol{E}^{(K-1)}\boldsymbol{W}^{(K-1)}$$

$$= \tilde{\boldsymbol{A}}\left(\tilde{\boldsymbol{A}}\boldsymbol{E}^{(K-2)}\boldsymbol{W}^{(K-2)}\right)\boldsymbol{W}^{(K-1)}$$

$$= \tilde{\boldsymbol{A}}\left(\tilde{\boldsymbol{A}}\left(\cdots\left(\tilde{\boldsymbol{A}}\boldsymbol{E}^{(0)}\boldsymbol{W}^{(0)}\right)\cdots\right)\boldsymbol{W}^{(K-2)}\right)\boldsymbol{W}^{(K-1)}$$

Set $\boldsymbol{E}$ as input embedding $\boldsymbol{E}^{(0)}$

$$= \tilde{\boldsymbol{A}}^{K}\,\boldsymbol{E}\left(\boldsymbol{W}^{(0)}\cdots\boldsymbol{W}^{(K-1)}\right)$$

Simplifying GCN (2)

- Removing ReLU significantly simplifies GCN!

$$\boldsymbol{E}^{(K)} = \boxed{\widetilde{\boldsymbol{A}}^{K}\,\boldsymbol{E}}\,\boldsymbol{W}$$

Diffusing node embeddings along the graph

$$\boldsymbol{W} \equiv \boldsymbol{W}^{(0)} \ldots \boldsymbol{W}^{(K-1)}$$

- Algorithm: Apply $E \leftarrow \tilde{A}E$ for K times
  - Each matrix multiplication diffuses the current embeddings to their one-hop neighbors.
  - Note: $\tilde{A}^K$ is dense and never gets materialized. Instead, the above iterative matrix-vector product is used to compute $\tilde{A}^K E$

Multi-Scale Diffusion

- We can consider multi-scale diffusion

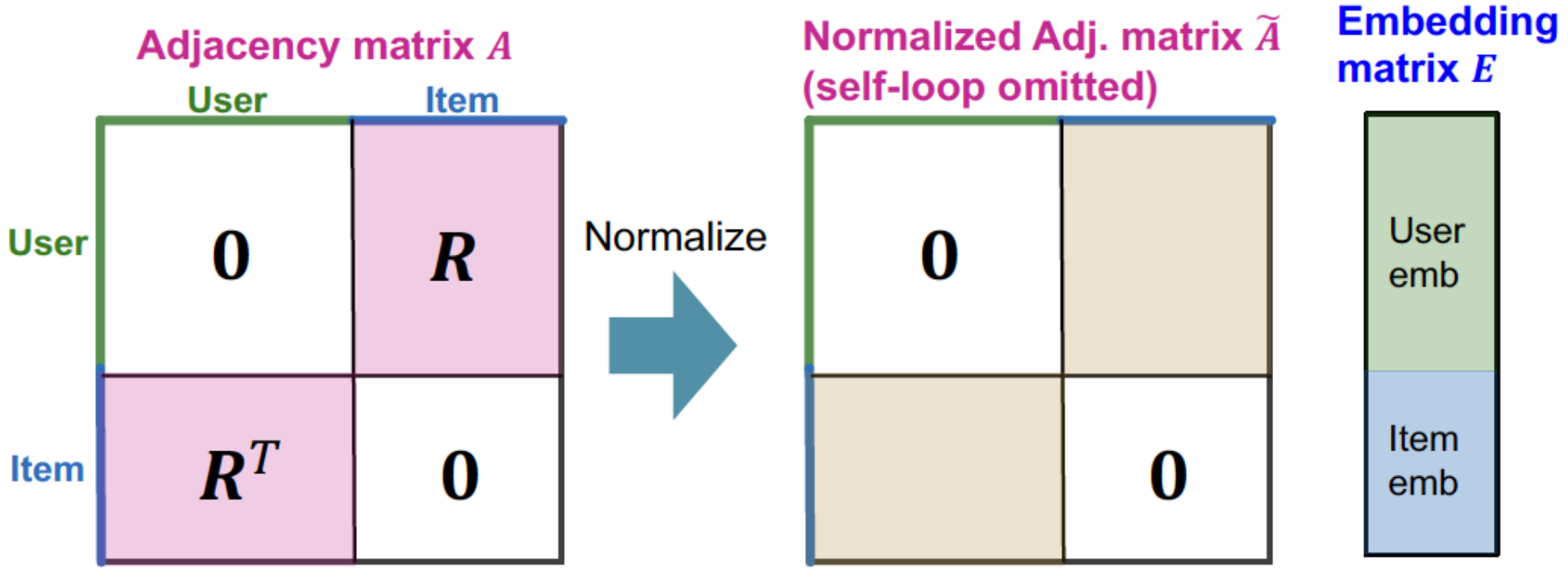
$$\alpha_0 E^{(0)} + \alpha_1 E^{(1)} + \cdots \alpha_K E^{(K)}$$

  - The above includes embeddings diffused at multiple hop scales.
  - $\alpha_0 E^{(0)} = \alpha_0 \tilde{A}^0 E^{(0)}$ acts as a self-connection (that is omitted in the definition $\tilde{A}$)
  - The coefficients $\alpha_0, \alpha_1, \cdots, \alpha_K$ are hyper-parameters

- For simplicity, LightGCN uses the uniform coefficient, i.e., $\alpha_k = \frac{1}{K+1}$ for all $k = 1, 2, \cdots, K$.

LightGCN: Model Overview (1)

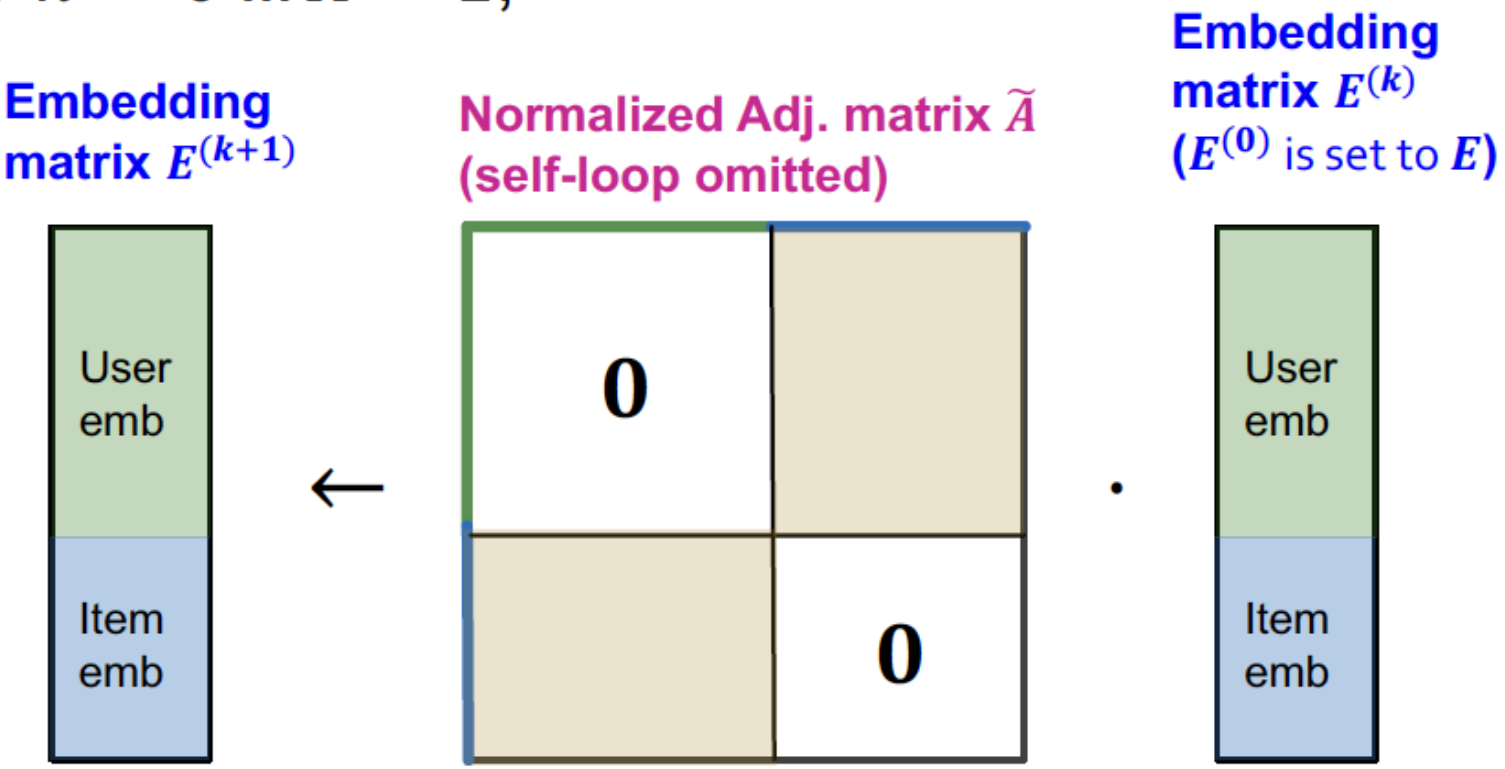
- Given:
  - Adjacency matrix A
  - Initial learnable embedding matrix $\boldsymbol{E}$



LightGCN: Model Overview (2)

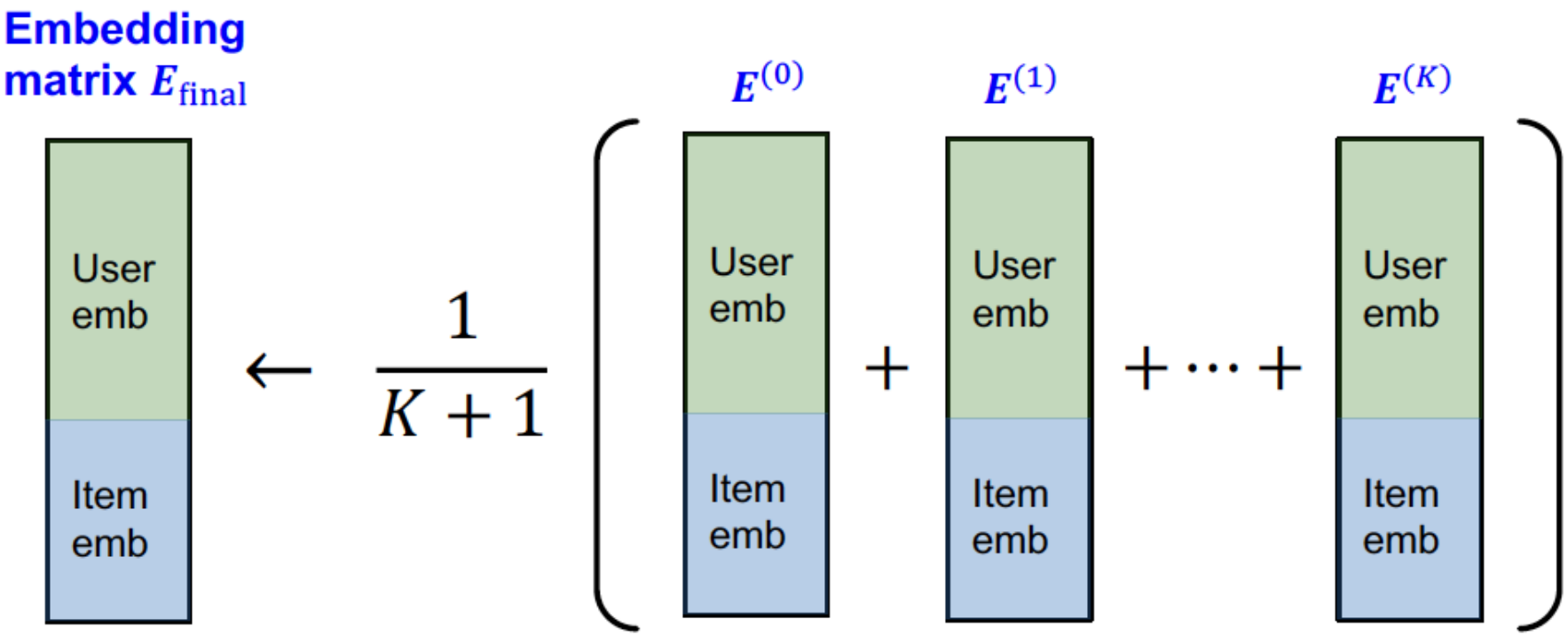
- Iteratively diffuse embedding matrix $\boldsymbol{E}$ using $\tilde{A}$

For $k = 0 \ldots K - 1$,



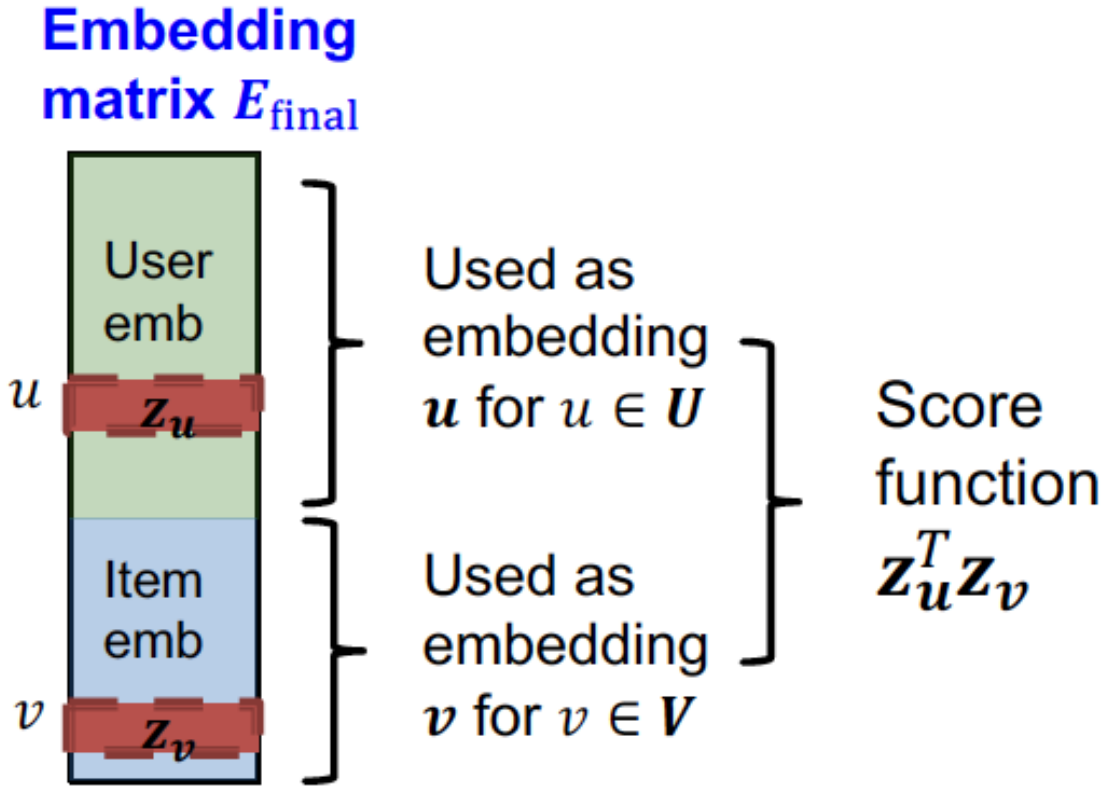
LightGCN: Model Overview (3)

- Average the embedding matrices at different scales.



LightGCN: Model Overview (4)

- Score function:
  - Use user/item vectors from E_{final} to score user-item interaction



LightGCN: Intuition

- Question: Why does the simple diffusion propagation work well?

- Answer: The diffusion directly encourages the embeddings of similar users/items to be similar.
  - Similar users share many common neighbors (items) and are expected to have similar future preferences (interact with similar items).

LightGCN and GCN

- The embedding propagation of LightGCN is closely related to GCN
- Recall: GCN (neighbor aggregation part)

$$\boldsymbol{h}_v^{(k+1)} = \sum_{u \in N(v)} \frac{1}{\sqrt{d_u}\sqrt{d_v}} \cdot \boldsymbol{h}_u^{(k)}$$

Node degree

  - Self-loop is added in the neighborhood definition.
- LightGCN uses the same equation except that
  - Self-loop is not added in the neighborhood definition.
  - Final embedding takes the average of embeddings from all the layers:

$$h_v = \frac{1}{K+1} \sum_{k=o}^{K} h_v^{(K)}$$

LightGCN and MF: Comparison

- Both LightGCN and shallow encoders learn a unique embedding for each user/item.
- The difference is that LightGCN uses the diffused user/item embeddings for scoring.
- LightGCN performs better than shallow encoders but are also more computationally expensive due to the additional diffusion step.
  - The final embedding of a user/item is obtained by aggregating embeddings of its multi-hop neighboring nodes.

LightGCN Summary

- LightGCN simplifies NGCF by removing the learnable parameters of GNNs.
- Learnable parameters are all in the shallow input node embeddings.
- Diffusion propagation only involves matrix-vector multiplication.
- The simplification leads to better empirical performance than NGCF.

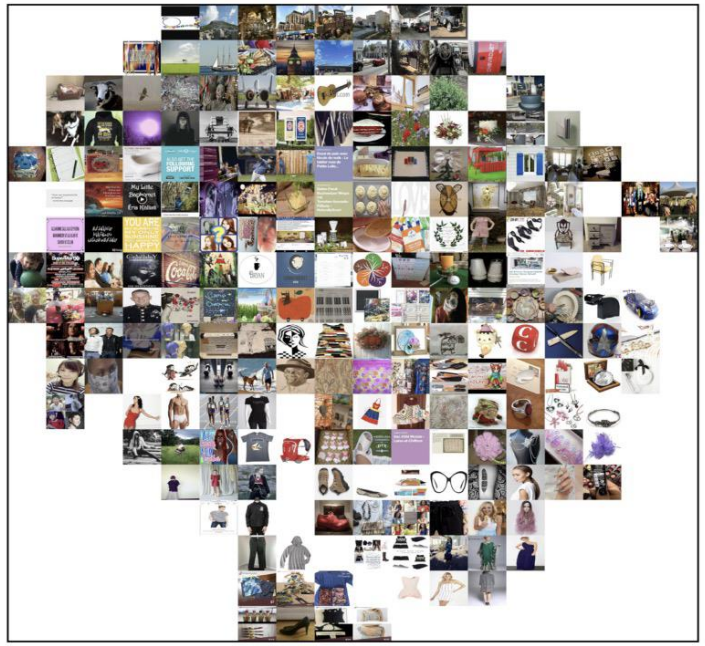
PinSAGE

Motivation

- P2P recommendation

PinSAGE: Pin Embedding

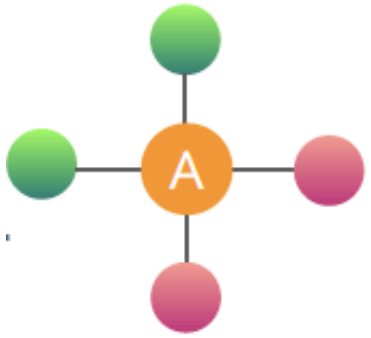
- Unifies visual, textual, and graph information.
- The largest industry deployment of a Graph Convolutional Networks.
- Huge Adoption across Pinterest
- Works for fresh content and is available in a few seconds after pin creation



Graph Convolutional Neural Networks for Web-Scale Recommender Systems, Ying et al., 2018

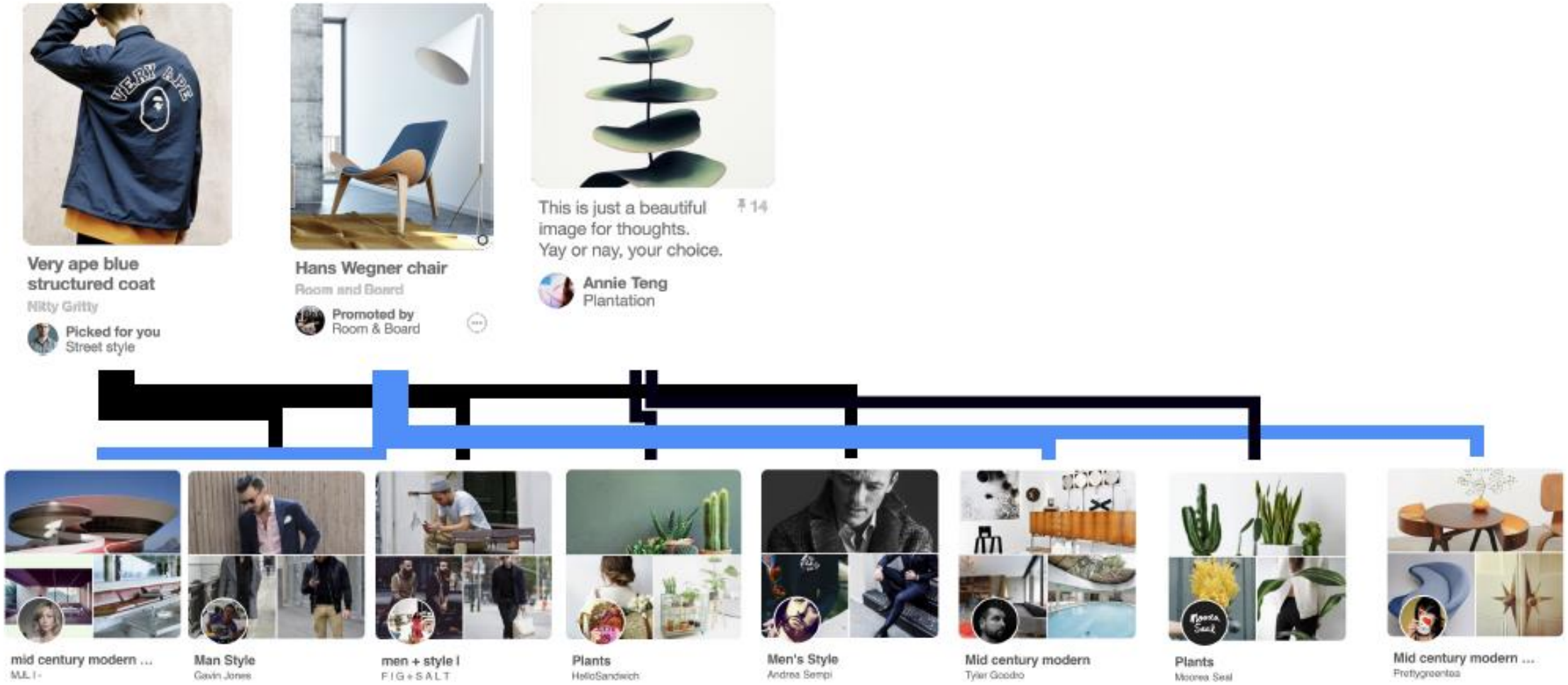
Application: Pinterest

- PinSage graph convolutional network:
  - Goal: Generate embeddings for nodes in a large-scale Pinterest graph containing billions of objects
  - Key Idea: Borrow information from nearby nodes
    - E.g., bed rail Pin might look like a garden fence, but gates and beds are rarely adjacent in the graph



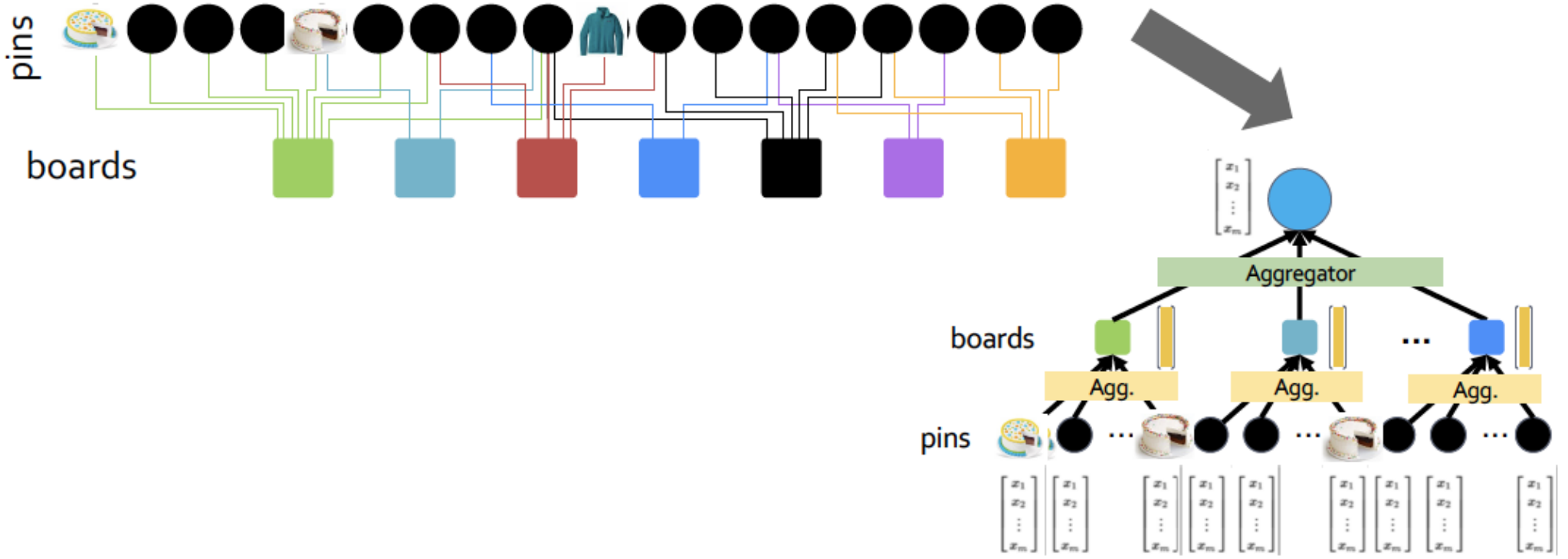
    - Pin embeddings are essential to various tasks like recommendation of Pins, classification, ranking
      - Services like “Related Pins”, “Search”, “Shopping”, “Ads”

Harnessing Pins and Boards



PinSAGE: Graph Neural Network

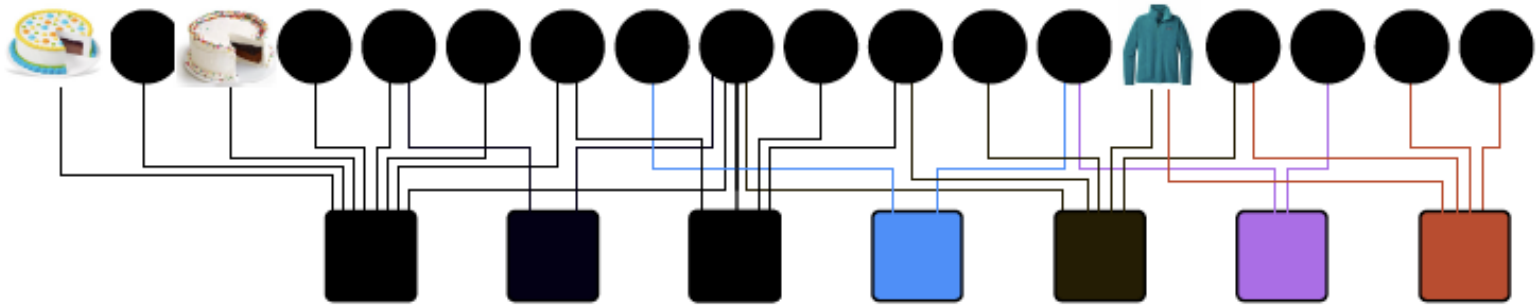
- Graph has tens of billions of nodes and edges
- Further resolves embeddings across the Pinterest graph



PinSAGE: Methods for Scaling Up

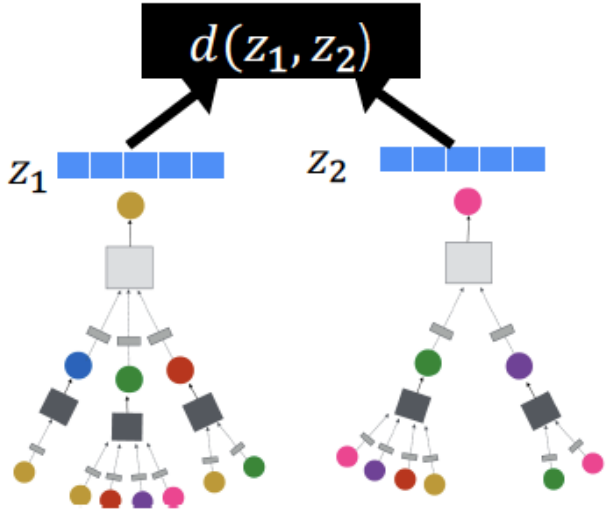
- In addition to the GNN model, the PinSAGE introduces several methods to scale the GNN to a billion-scale recommender system (e.g., Pinterest).
  - Shared negative samples across users in a mini-batch
  - Hard negative samples
  - Curriculum learning
  - Mini-batch training of GNNs on a large-graph (to be covered in the next part of the lecture)

PinSAGE Model



- Task: Recommend related pins to users
- Learn node embeddings z_i such that

$$d(z_{cake1}, z_{cake2}) < d(z_{cake1}, z_{sweater})$$

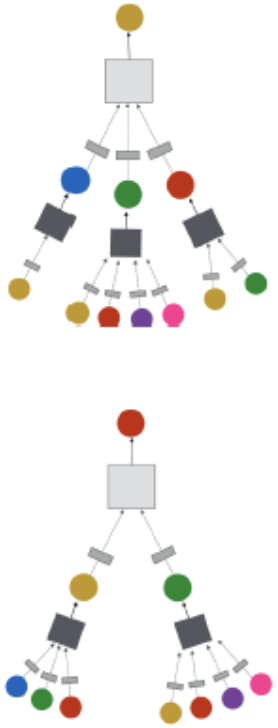


Training Data

- 1+B repin pairs:
  - From Related Pins surface
  - Capture semantic relatedness
  - Goal: Embed such pairs to be “neighbors”
- Example positive training pairs (Q,X):

Shared Negative Samples (1)

- Recall: In BPR loss, for each user $u^* \in U_{mini}$, we sample one positive item v_{pos} and a set of sampled negative items $V_{neg} = \{v_{neg}\}$
- Using more negative samples per user improves the recommendation performance, but is also expensive.
- We need to generate $|U_{mini}| \cdot |V_{neg}|$ embeddings for negative nodes.
- We need to apply $|U_{mini}| \cdot |V_{neg}|$ GNN computational graphs (see right), which is expensive.



Shared Negative Samples (2)

- Key idea: We can share the same set of negative samples $V_{neg} = \{v_{neg}\}$ across all users U_{mini} in the mini-batch.

- This way, we only need to generate $|V_{neg}|$ embeddings for negative nodes.
  - This saves the node embedding generation computation by a factor of $|U_{mini}|$!
  - Empirically, the performance stays similar to the non-shared negative sampling scheme.

Hard Negatives (1)

- Challenge: Industrial recsys needs to make extremely fine-grained predictions.
  - #Total items: Up to billions.
  - #Items to recommend for each user: 10 to 100.
- Issue: The shared negative items are randomly sampled from all items
  - Most of them are "easy negatives", i.e., a model does not need to be fine-grained to distinguish them from positive items.
- We need a way to sample "hard negatives" to force the model to be fine-grained!

PinSAGE: Curriculum Learning

- Idea: use harder and harder negative samples
- Include more and more hard negative samples for each epoch

Source pin

Positive


Easy negative

Hard negative

Curriculum Learning

- Key insight: It is effective to make the negative samples gradually harder in the process of training.

- At n-th epoch, we add $n - 1$ hard negative items.
  - #(Hard negatives) gradually increases in the process of training.

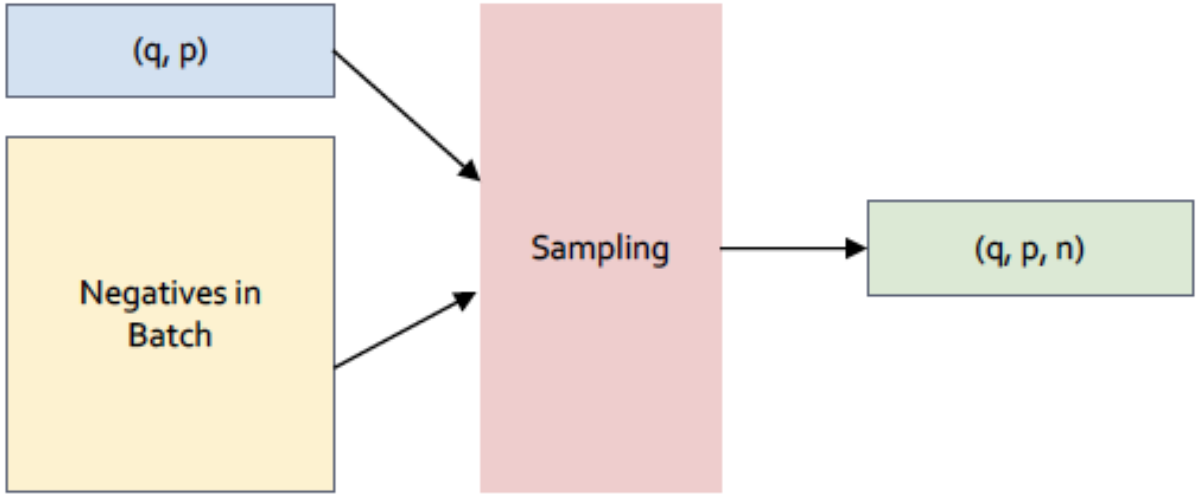
- The model will gradually learn to make finer grained predictions.

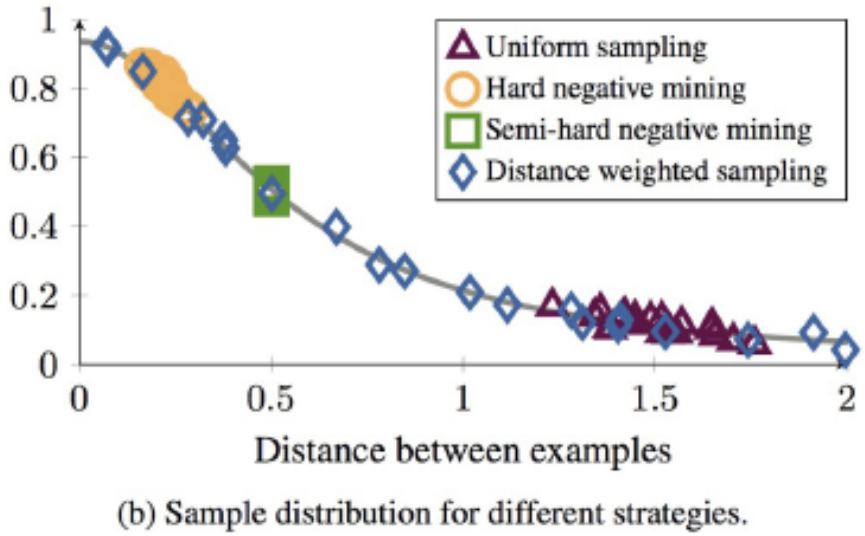
Hard Negatives (2)

- For each user node, the hard negatives are item nodes that are close (but not connected) to the user node in the graph.
- Hard negatives for user $u \in \boldsymbol{U}$ are obtained as follows:
  - Compute random walks from user u.
    - Run random walk with restart from u, obtain visit counts for other items/nodes.
  - Sort items in the descending order of their visit count.
  - Randomly sample items that are ranked high but not too high, e.g., 2000th —5000th .
    - Item nodes that are close but not too close (connected) to the user node.
- The hard negatives for each user are used in addition to the shared negatives.

PinSAGE: Negative Sampling

- (q, p) positive pairs are given but various methods to sample negatives to form (q, p, n)
- Distance Weighted Sampling (Wu et al., 2017)
  - Sample negatives so that query-negative distance distribution is approx U[0.5, 1.4]





(b) Sample distribution for different strategies.

Fine-Grained Object Similarity

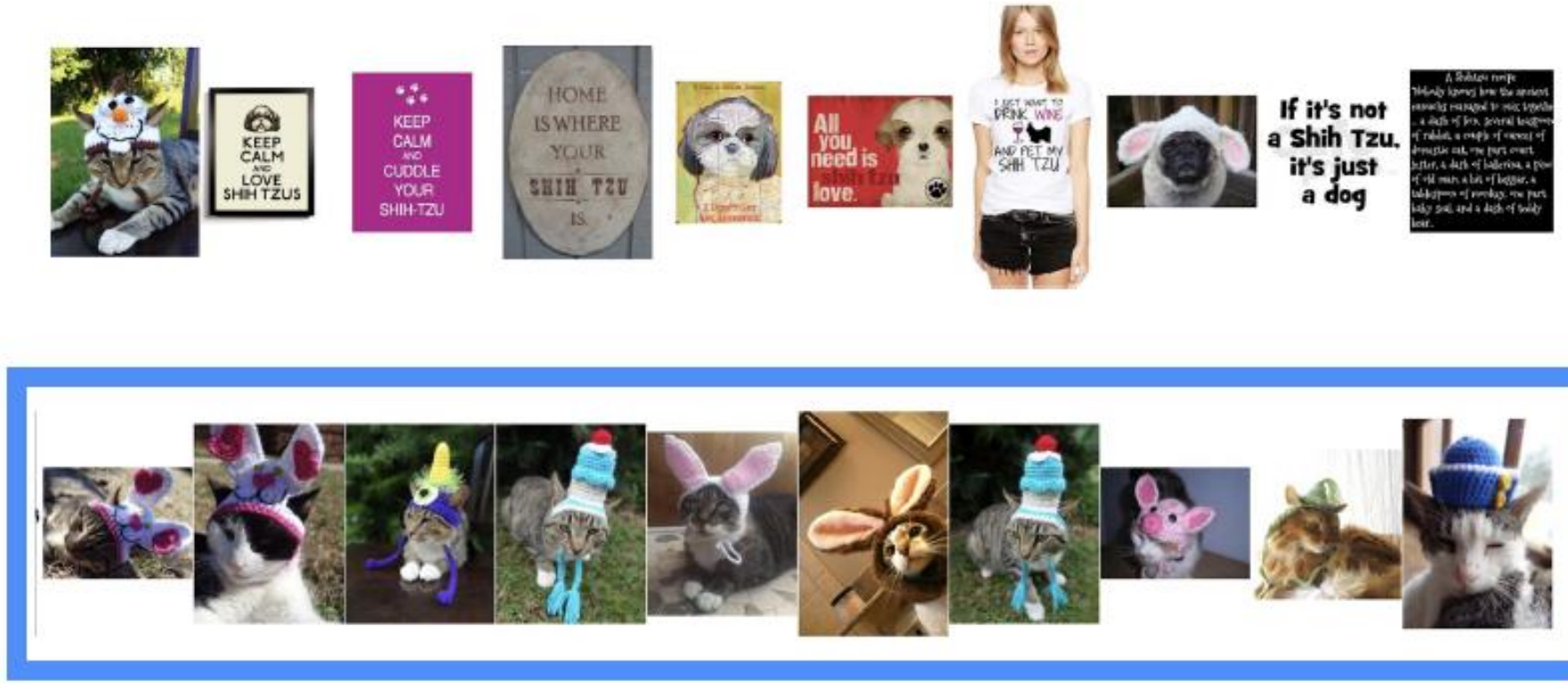
Query

Visual only

PinSAGE

Compare against Prod

Query



PinSAGE: Summary

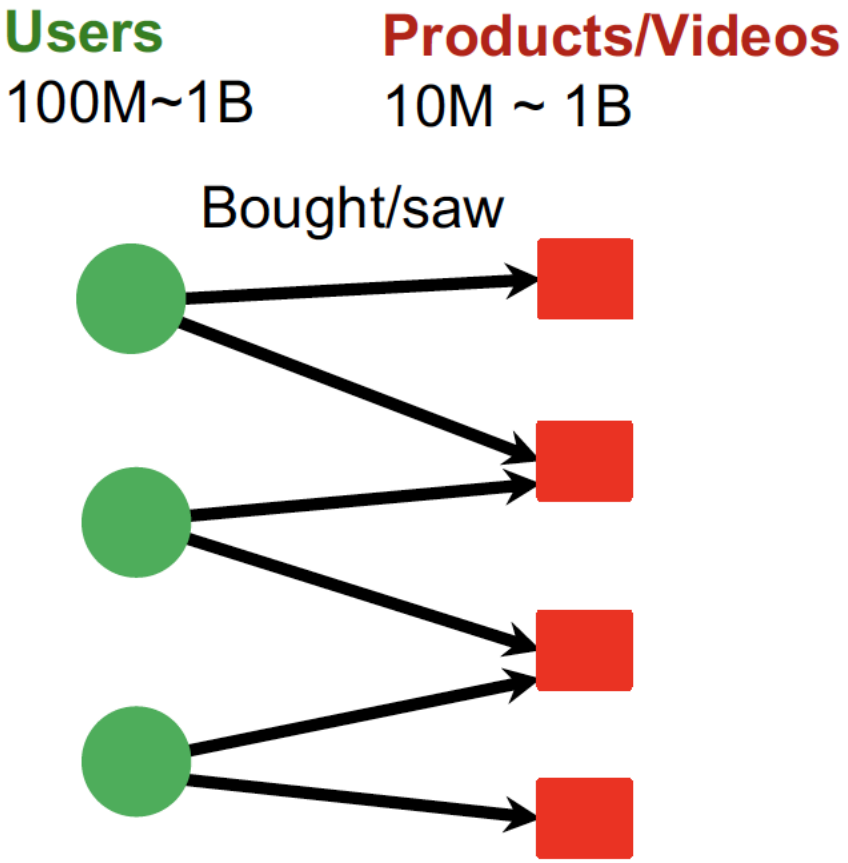
- PinSAGE uses GNNs to generate high-quality user/item embeddings that capture both the rich node attributes and graph structure.
- The PinSAGE model is effectively trained using sophisticated negative sampling strategies.
- PinSAGE is successfully deployed at Pinterest, a billion-scale image content recommendation service.
  - Uncovered in this lecture: How to scale up GNNs to large-scale graphs. Will be covered in a later lecture.

Scaling Up GNNs

Graphs in Modern Applications

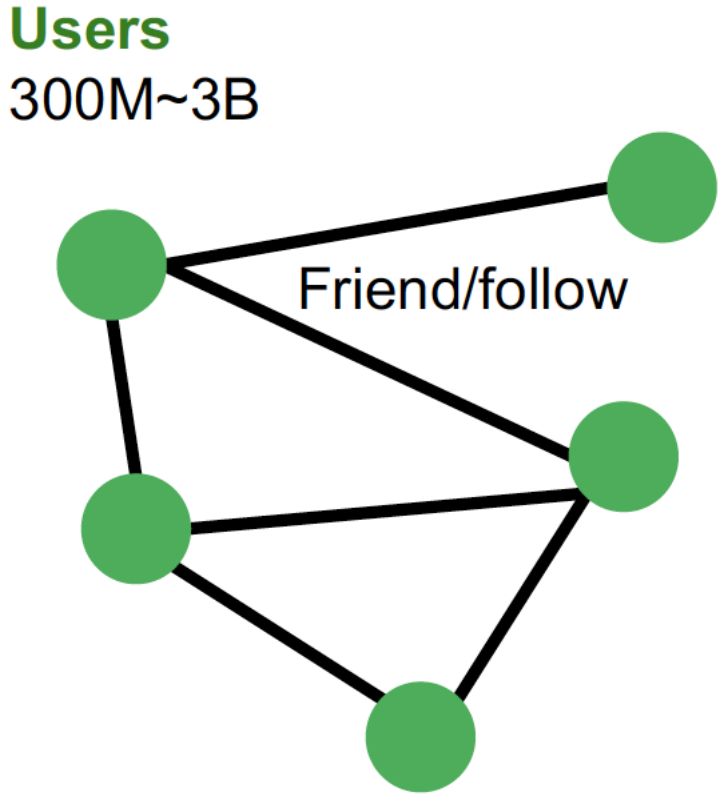
- Recommender systems:
  - Amazon
  - YouTube
  - Pinterest
  - Etc.

- ML tasks:
  - Recommend items (link prediction)
  - Classify users/items (node classification)



Graphs in Modern Applications

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

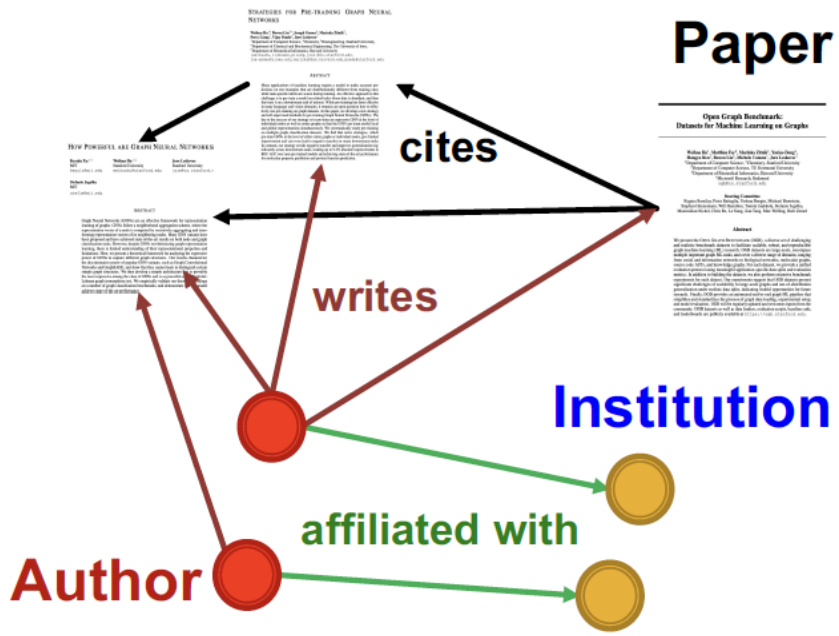


Graphs in Modern Applications

- Academic graph:
  - Microsoft Academic Graph

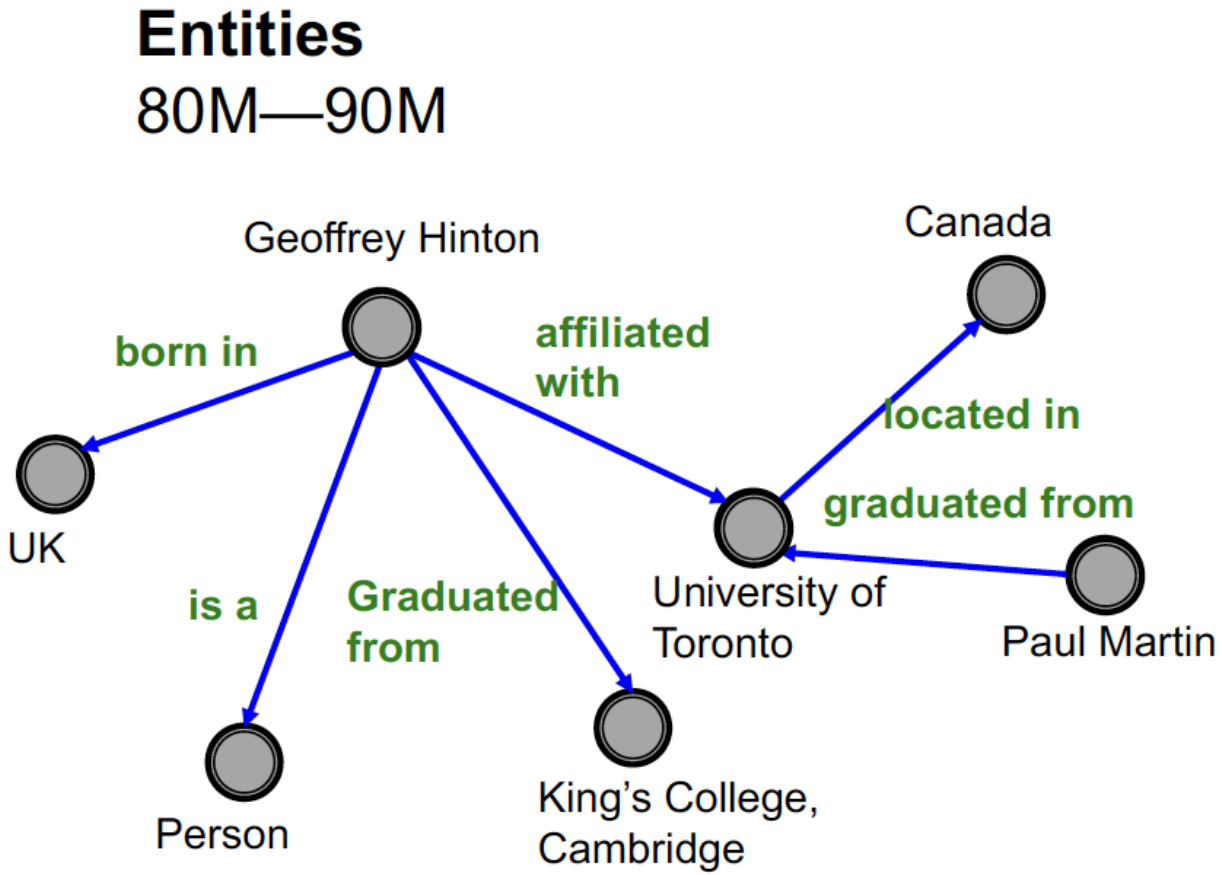
Papers 120M Authors 120M

- ML tasks:
  - Paper categorization (node classification)
  - Author collaboration recommendation
  - Paper citation recommendation (link prediction)



Graphs in Modern Applications

- Knowledge Graphs (KGs):
  - Wikidata
  - Freebase
- ML tasks:
  - KG completion
  - Reasoning



What is in Common?

- Large-scale:
  - #nodes ranges from 10M to 10B.
  - #edges ranges from 100M to 100B.
- Tasks
  - Node-level: User/item/paper classification.
  - Link-level: Recommendation, completion.
- Todays' lecture
  - Scale up GNNs to large graphs!

Why is it Hard?

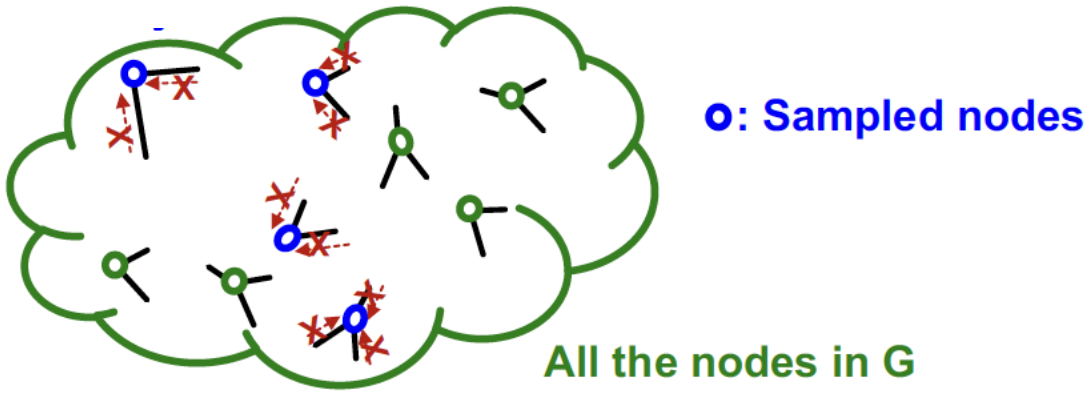
- Recall: How we usually train an ML model on large data ($\boldsymbol{N}$ = #data is large)?
- Objective: Minimize the averaged loss

$$l(\theta) = \frac{1}{N}\sum_{i=0}^{N-1} l_i(\theta)$$

  - θ: model parameters, $l_i(\theta)$: loss for i-th data point
- We perform Stochastic Gradient Descent
  - Sample M (<<N) data points (mini-batches)
  - Compute the $l_{sub}(\theta)$ over the M data points
  - Perform SGD: $\theta = \theta - \nabla l_{sub}(\theta)$

Why is it Hard?

- What if we were to use the standard SGD for GNN?
- 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!

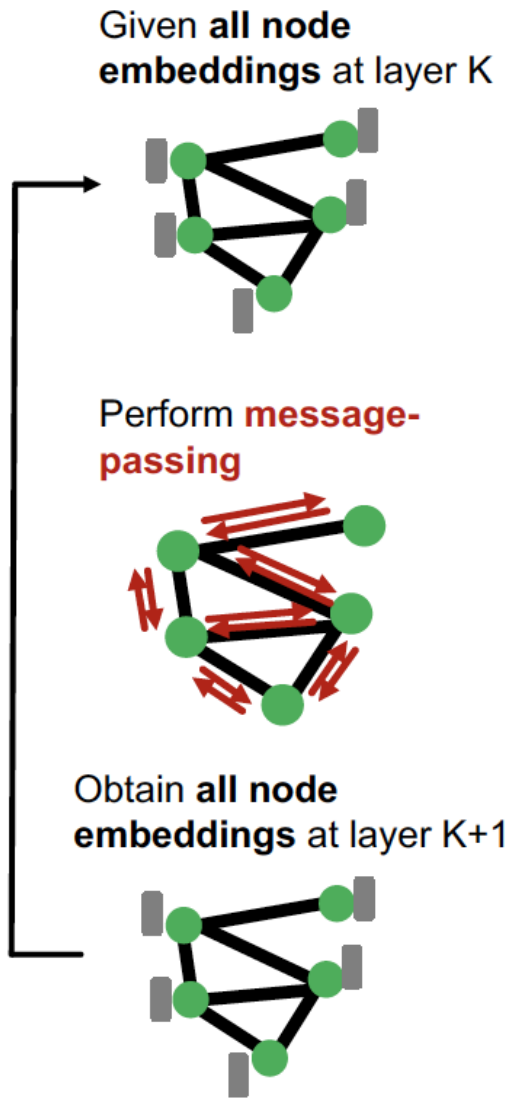
- Standard SGD cannot effectively train GNNs.

Why is it Hard?

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

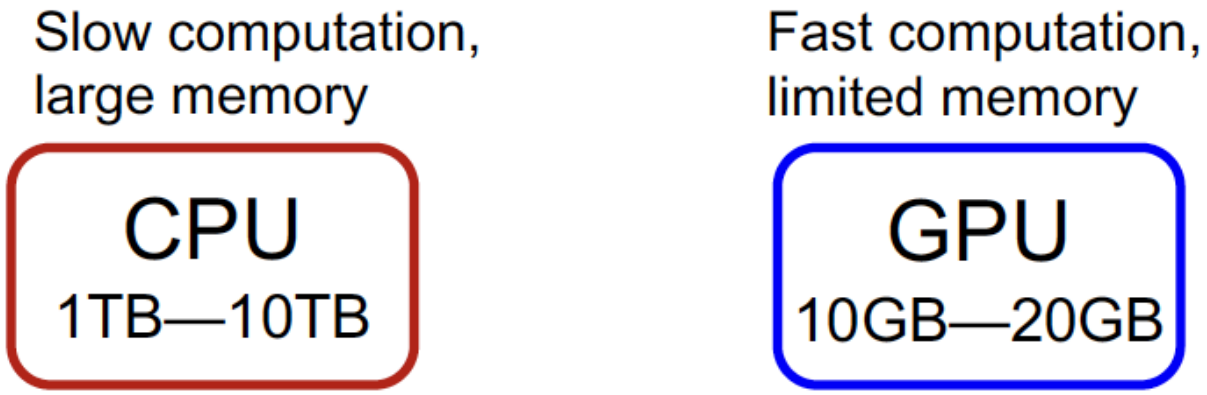
$$H^{(k+1)} = \sigma\left(\tilde{A}H^{(k)}W_k^T\right) + H^{(k)}B_k^T$$

  - 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 a 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.



Lecture Outline

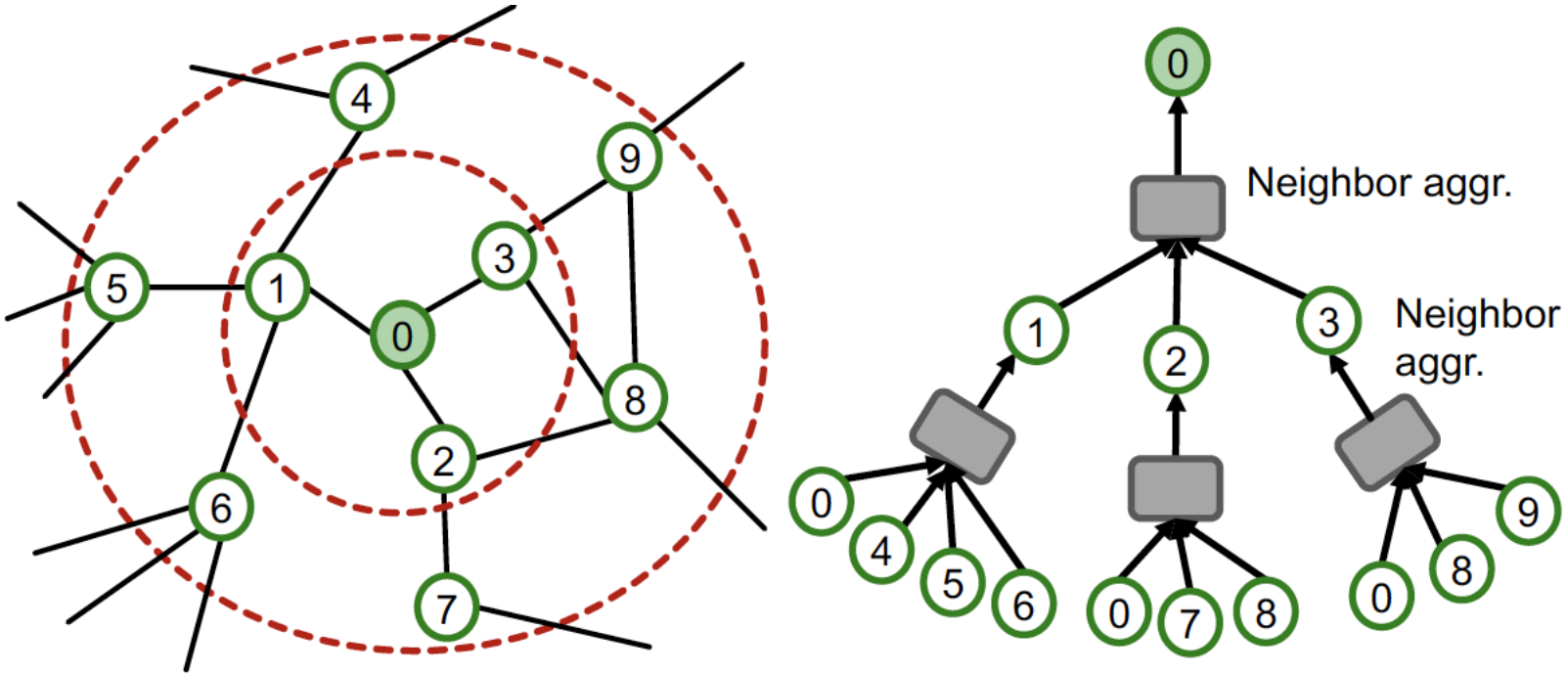
We introduce three methods for scaling up GNNs:

- Two methods perform message-passing over small subgraphs in each mini-batch; only the subgraphs need to be loaded on a GPU at a time.
  - Neighbor Sampling [Hamilton et al. NeurIPS 2017]
  - Cluster-GCN [Chiang et al. KDD 2019]
- One method simplifies a GNN into feature preprocessing operation (can be efficiently performed even on a CPU)
  - Simplified GCN [Wu et al. ICML 2019]

GraphSAGE Neighbor Sampling

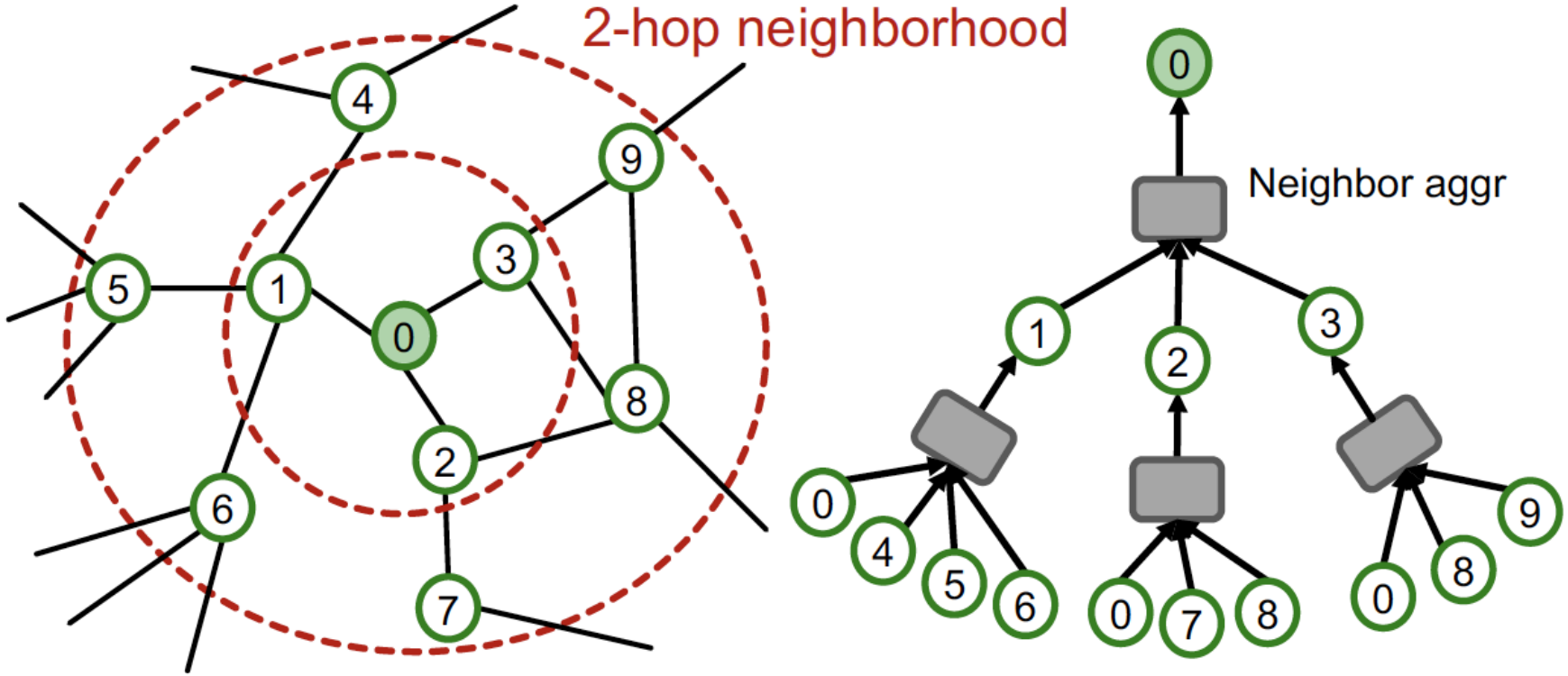
Recall: Computational Graph

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



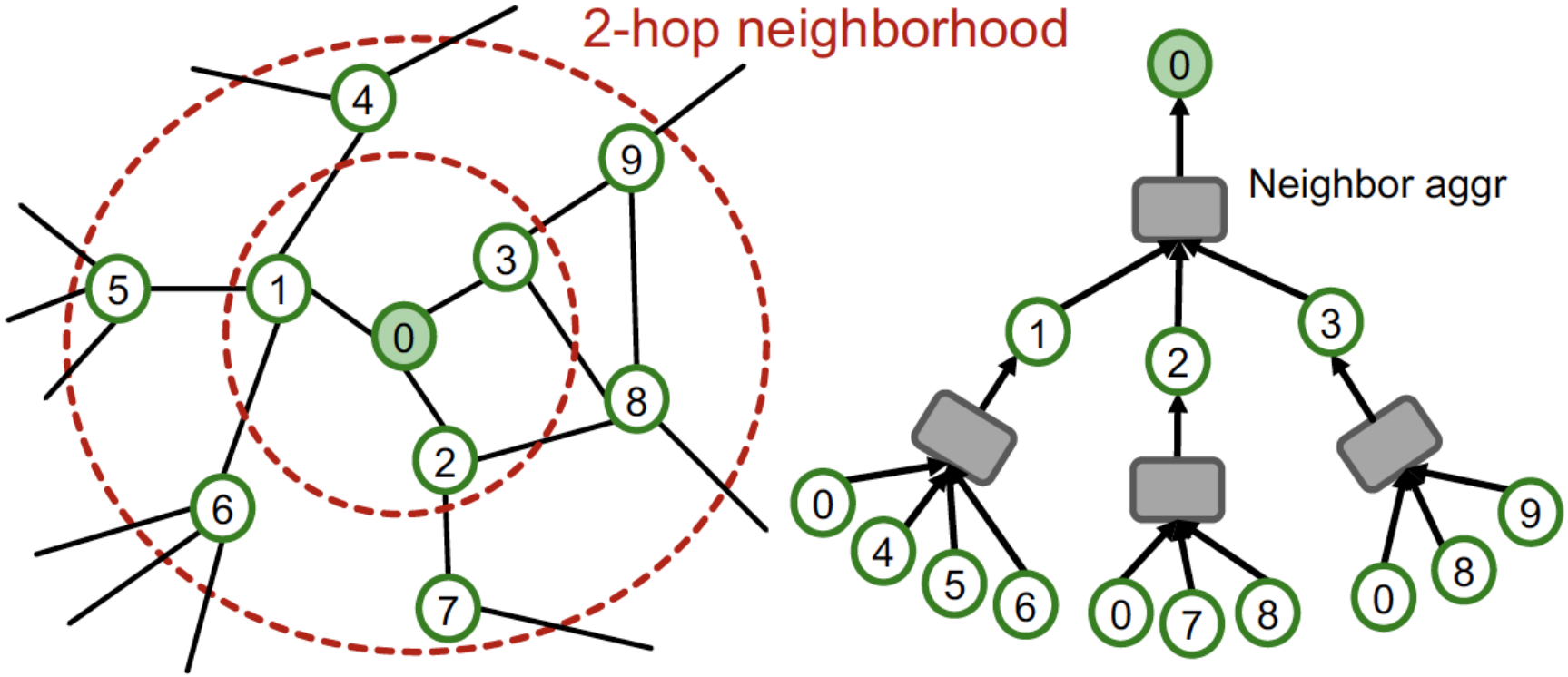
Recall: Computational Graph

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



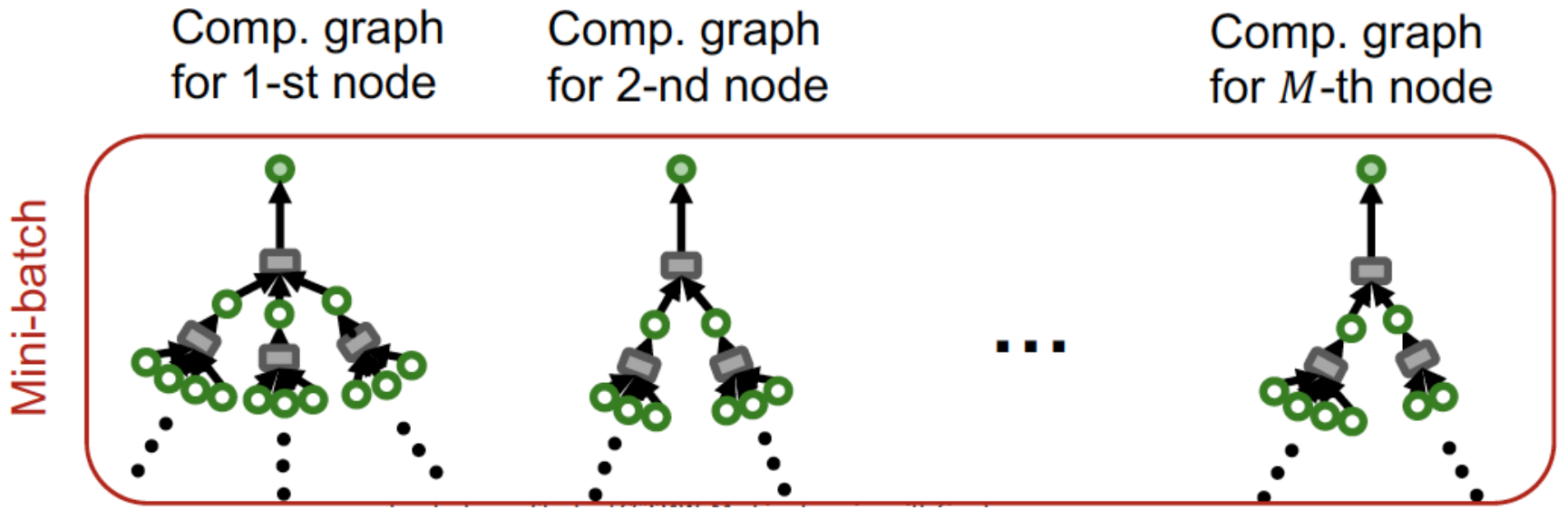
Recall: Computational Graph

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



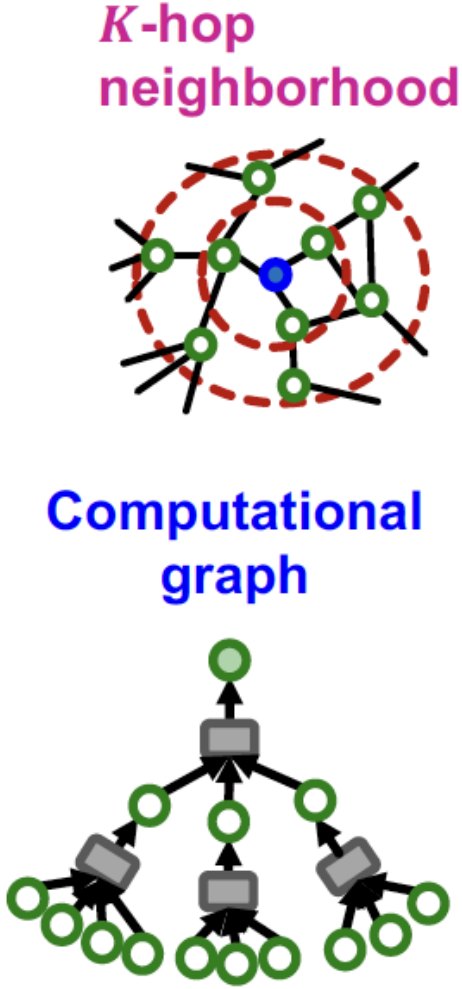
Computing Node Embeddings

- 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!



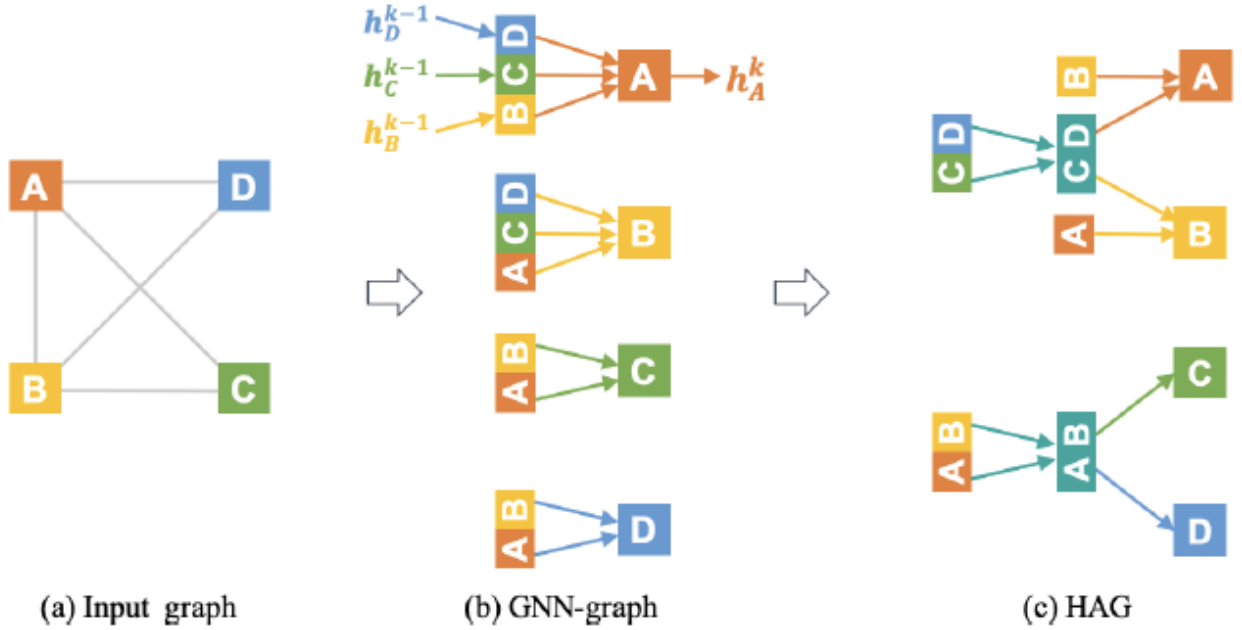
Stochastic Training of GNNs

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



Issue with Stochastic Training (1)

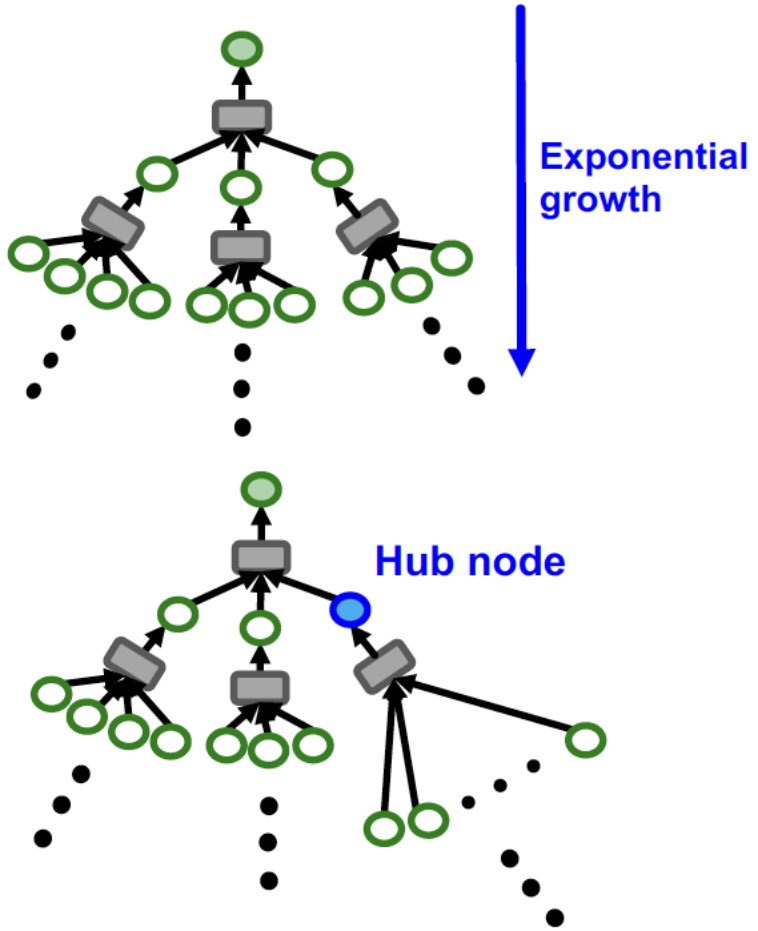
- For each node, we need to get the entire ***K***-hop neighborhood and pass it through the computation graph.
- We need to aggregate lot of information to compute one node embedding.
- Some computational redundancy:



Issue with Stochastic Training (2)

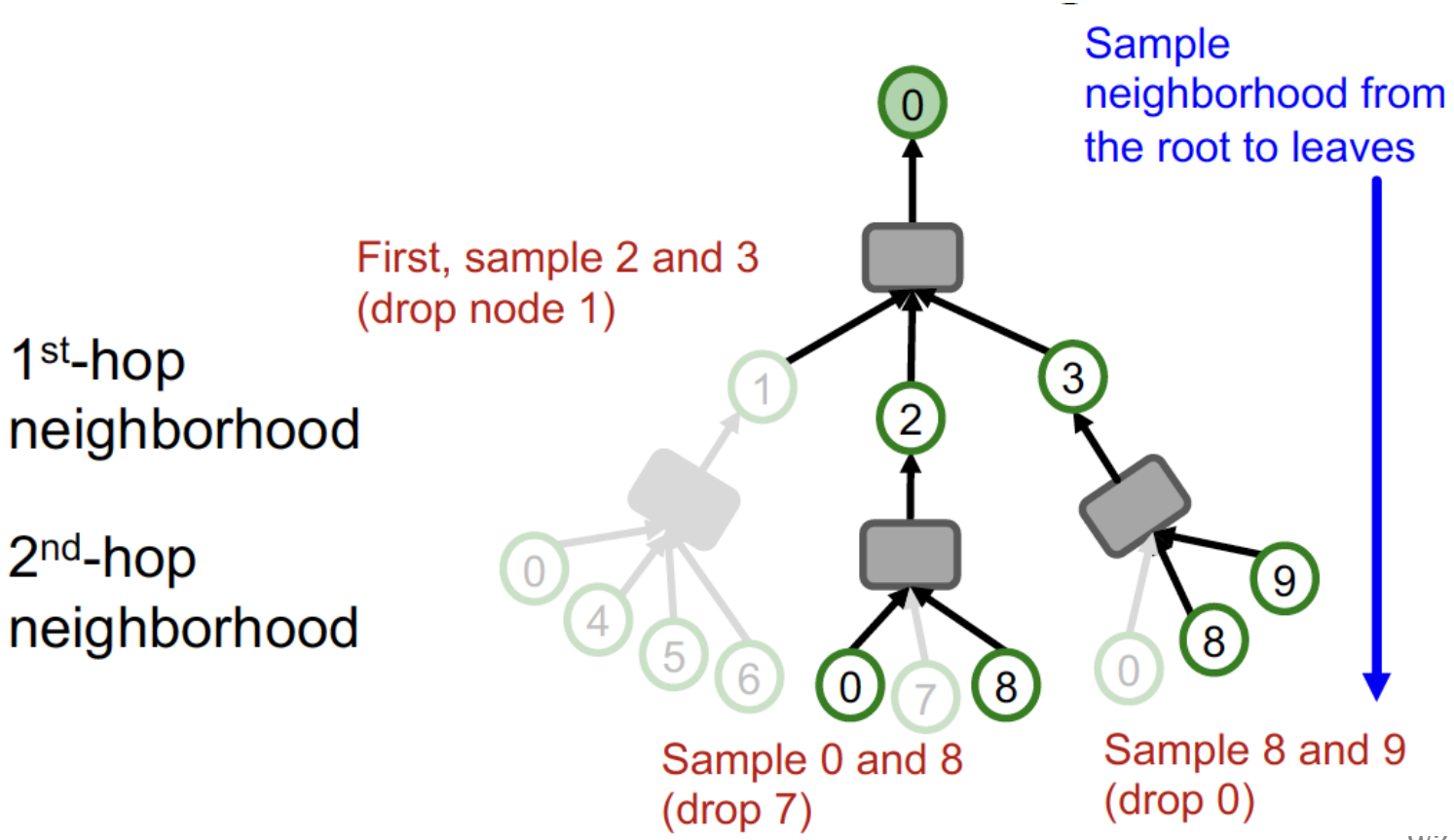
- 2nd issue:
  - Computation graph becomes exponentially large with respect to the layer size K.
  - Computation graph explodes when it hits a hub node (high-degree node).

- Next: Make the comp. graph more compact!



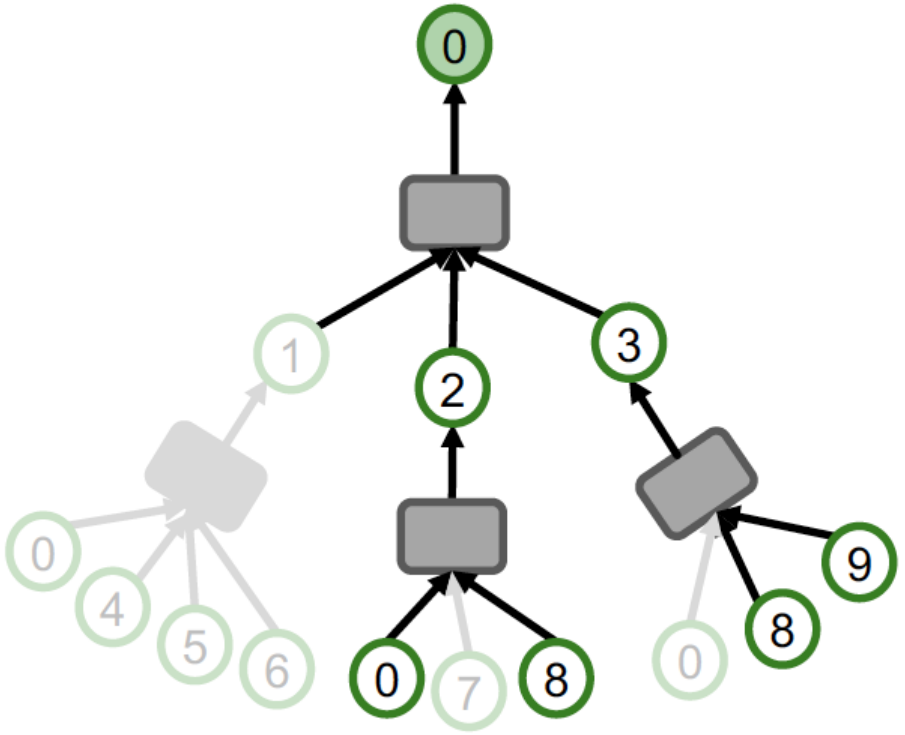
Neighborhood Sampling

- Key idea: Construct the computational graph by (randomly) sampling at most H neighbors at each hop.
- Example ($\boldsymbol{H = 2}$):



Neighborhood Sampling

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



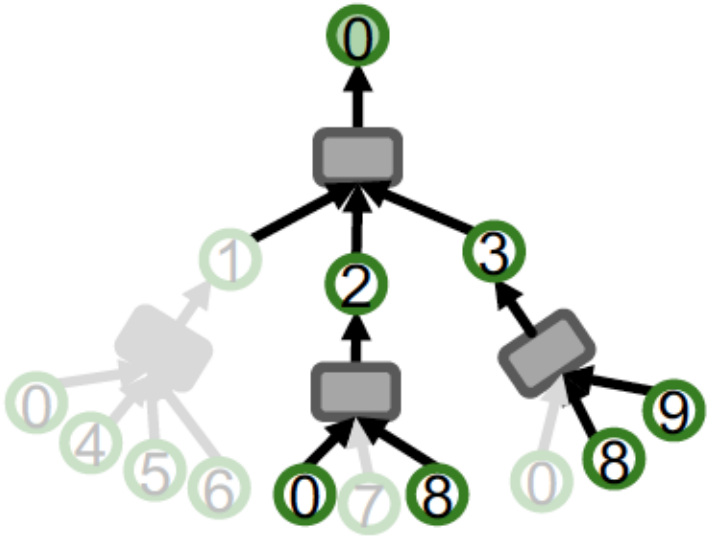
Neighborhood Sampling Algorithm

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.

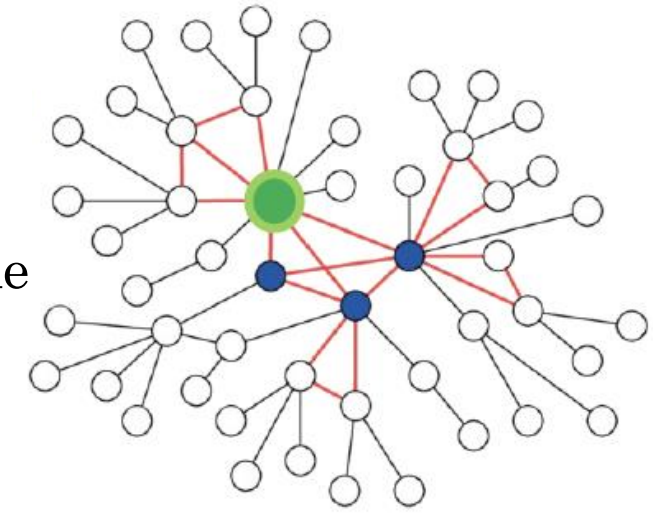
Remarks on Neighborhood Sampling (1)

- Remark 1: 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.

- Remark 2: 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 (2)

- Remark 3: 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: Neighbor Sampling

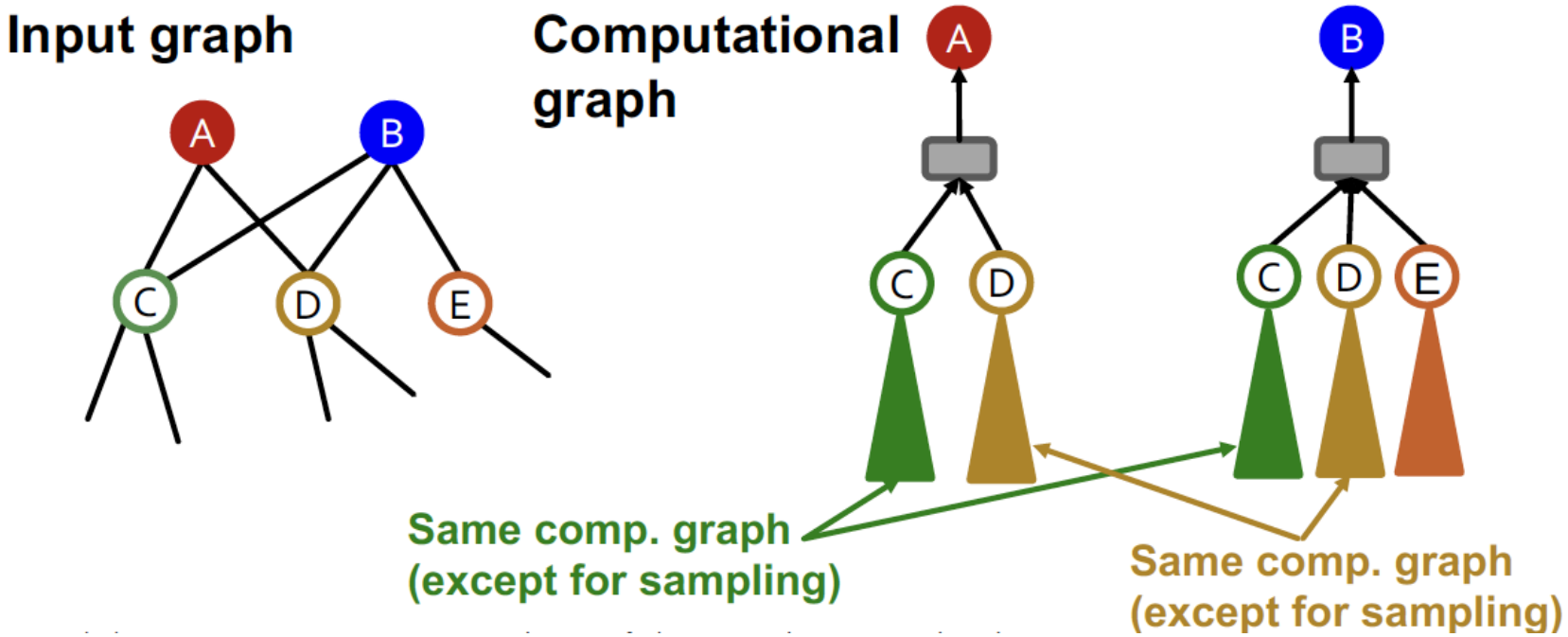
- A computational graph is constructed for each node in a mini-batch.
- In neighbor sampling, the comp. graph is pruned/sub-sampled to increase computational efficiency.
- The pruned comp. graph is used to generate a node embedding.

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

Cluster-GCN

Issues with Neighbor 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

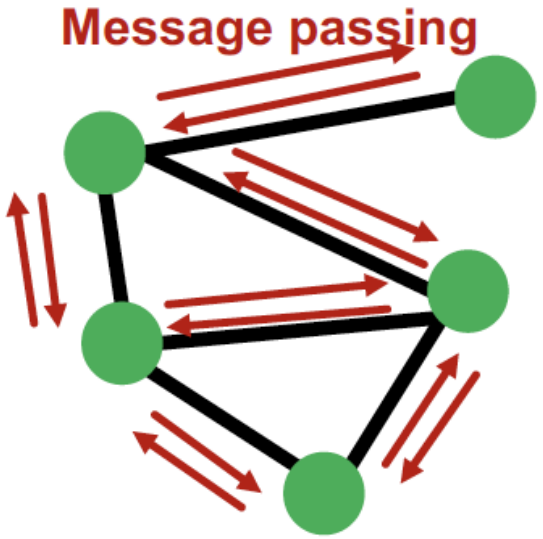
- In full-batch GNN implementation, all the node embeddings are updated together using embeddings of the previous layer.

Update for all $v \in V$

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

Message

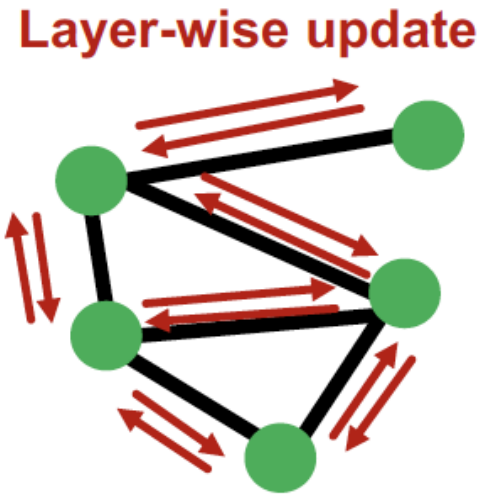
- In each layer, only 2*#(edges) messages need to be computed.
- For K-layer GNN, only $2K$*#(edges) messages need to be computed.
- GNN's entire computation is only linear in #(edges) and #(GNN layers). Fast!



Insight from Full-Batch GNN

- The layer-wise node embedding update allows the re-use of embeddings from the previous layer.

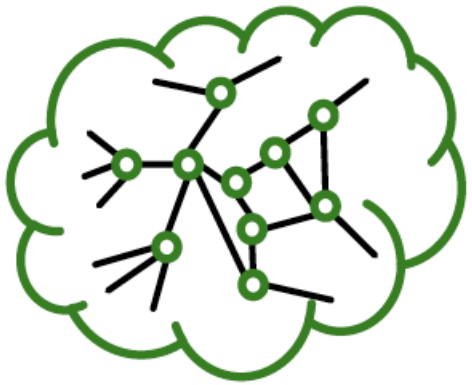
- This significantly reduces the computational redundancy of neighbor sampling.
  - Of course, the layer-wise update is not feasible for a large graph due to limited GPU memory.
  - Requires putting the entire graph and features on GPU.



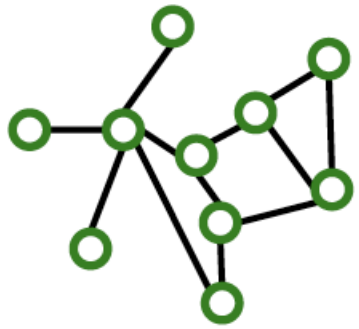
Subgraph Sampling

- 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.

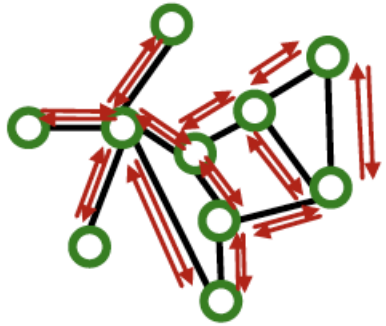
Large graph



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



**Layer-wise
node embeddings
update on the GPU**

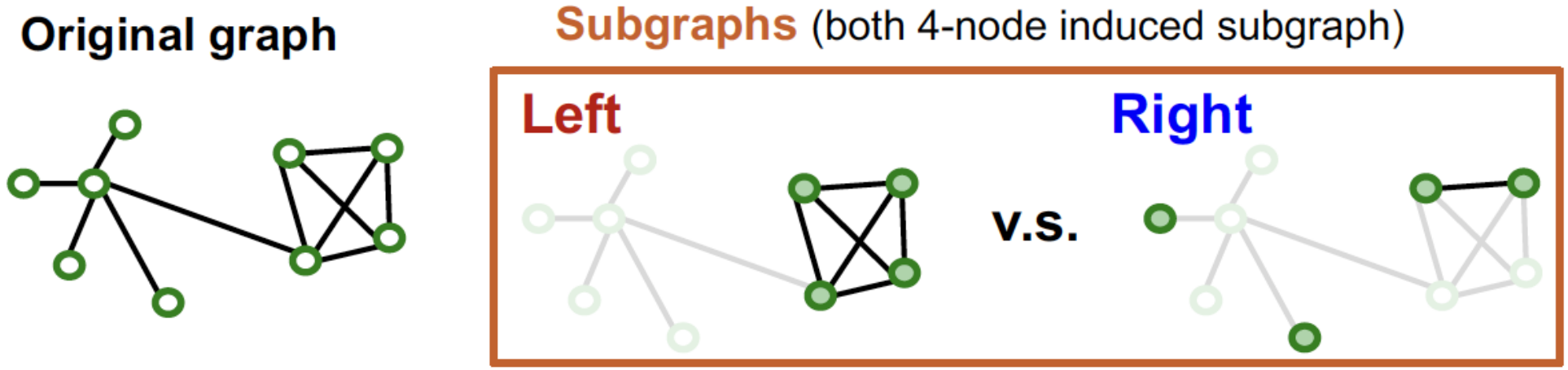


Subgraph Sampling

- 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.

Subgraph Sampling: Case Study

- Which subgraph is good for training GNN?

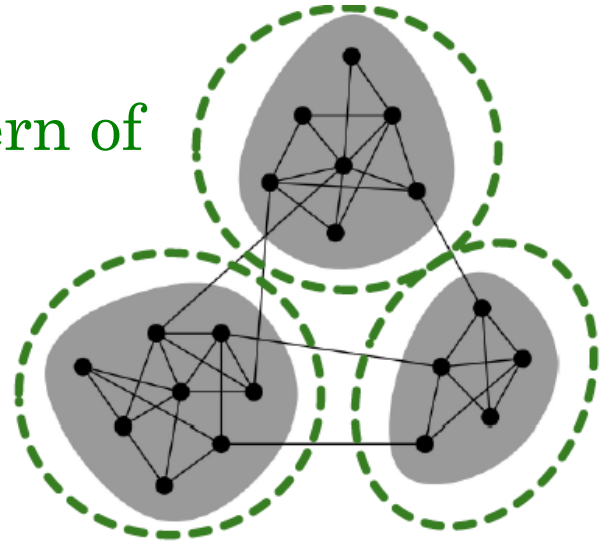


- Left subgraph retains the essential community structure among the 4 nodes (Good)
- Right subgraph drops many connectivity patterns, even leading to isolated nodes (Bad)

Exploiting Community Structure

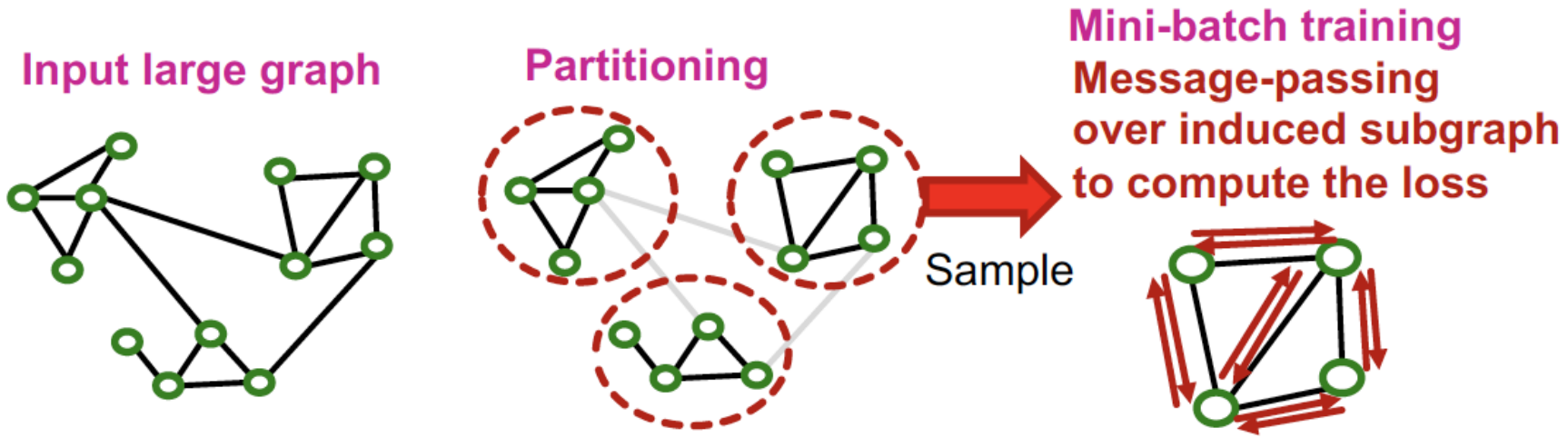
- Real-world graph exhibits community structure
  - A large graph can be decomposed into many small communities.

- 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

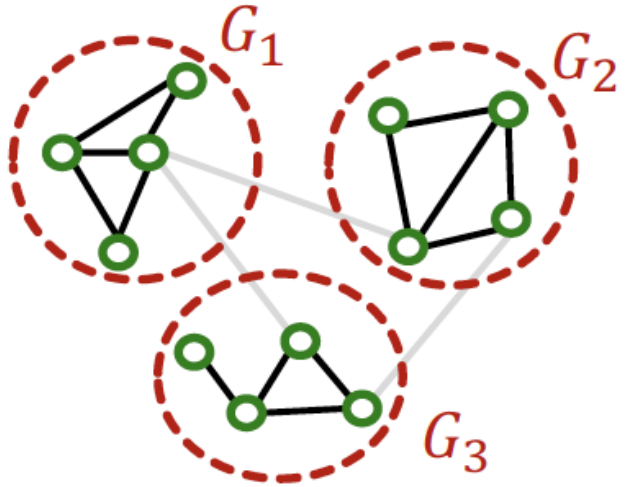
- We first introduce "vanilla" Cluster-GCN.
- Cluster-GCN consists of two steps:
  - Pre-processing: Given a large graph, partition it into groups of nodes (i.e., subgraphs).
  - Mini-batch training: Sample one node group at a time. Apply GNN's message passing over the induced subgraph.



Cluster-GCN: Pre-processing

- Given a large graph $G = (V, E)$, partition its nodes $\boldsymbol{V}$ into $\boldsymbol{C}$ groups: $\boldsymbol{V_1, \ldots, V_C}$.
  - We can use any scalable community detection methods, e.g., Louvain, METIS [Karypis et al. SIAM 1998].
- $V_1, \ldots, V_C$ induces C subgraphs, $G_1, \ldots, G_C$,
  - Recall: $G_c \equiv (V_c, E_c)$, where $E_c = \{(u, v) | u, v \in V_c\}$

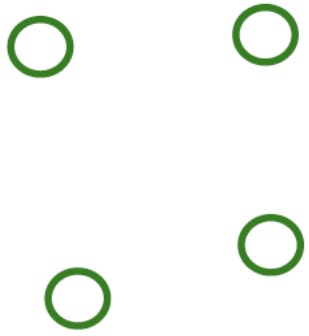
Notice: Between-group edges are not included in $G_1, \ldots, G_C$



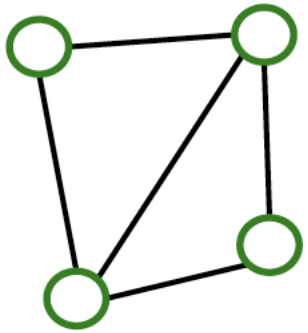
Cluster-GCN: Mini-Batch Training

- 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

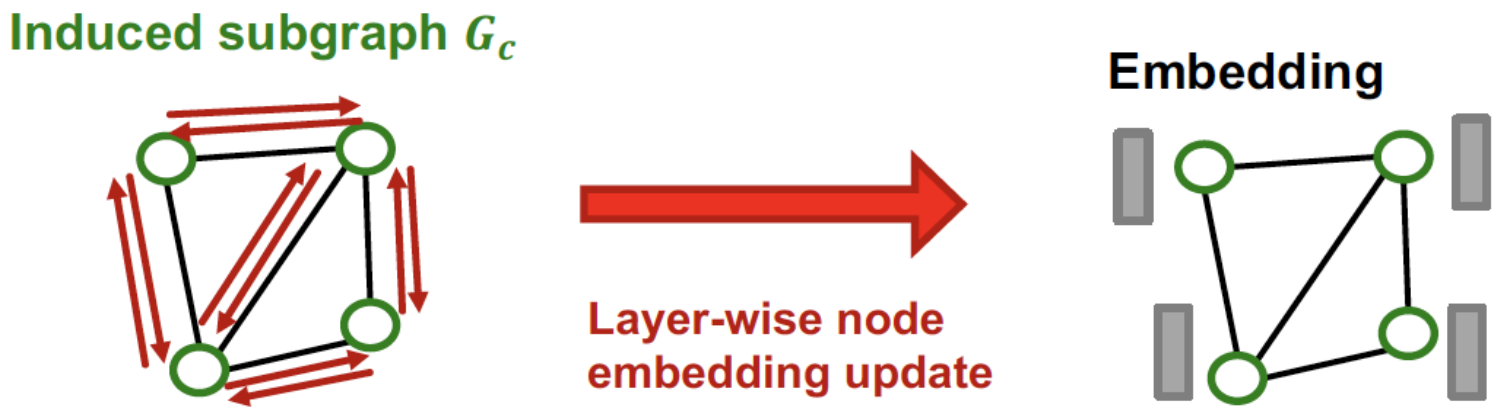


Induced subgraph G_c



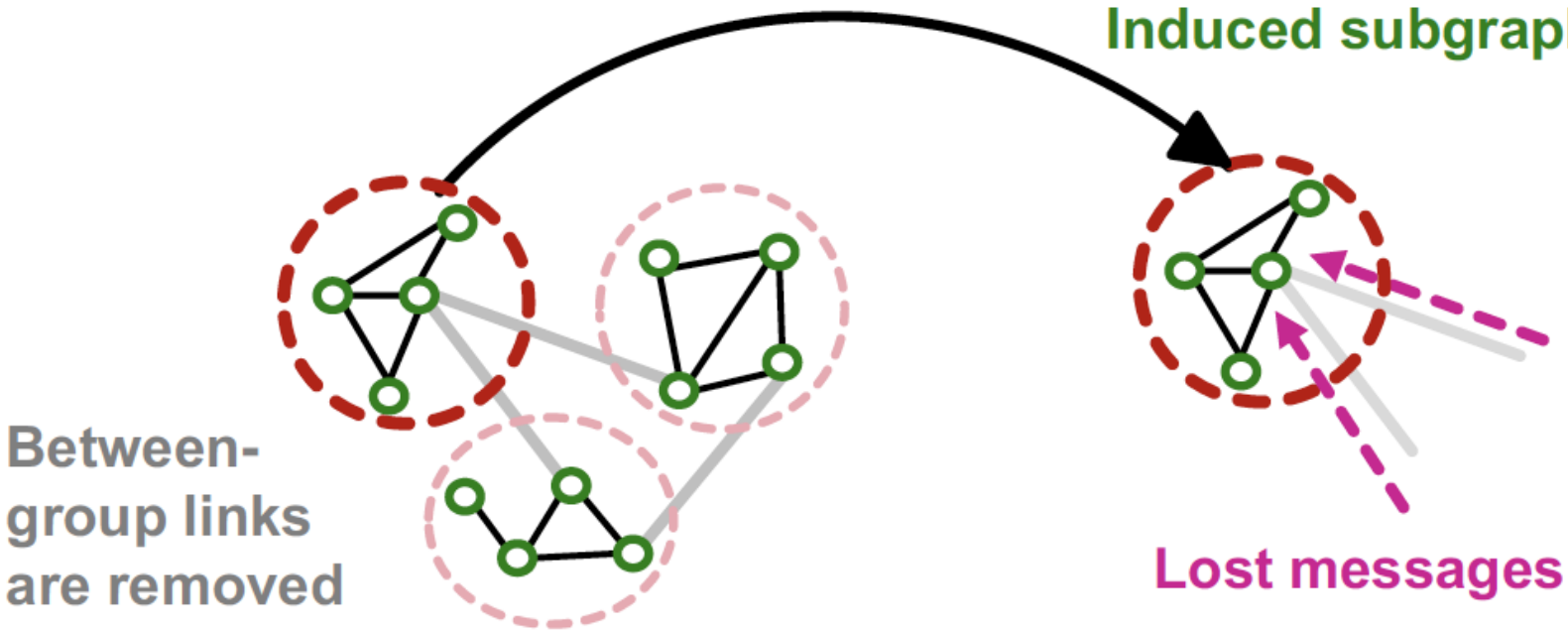
Cluster-GCN: Mini-Batch Training

- Apply GNN's layer-wise node update over $\boldsymbol{G_c}$ to obtain embedding $\boldsymbol{h}_v$ for each node $v \in V_c$.
- ¡ Compute the loss for each node $v \in V_c$ and take average: $l_{sub}(\theta) = \left(\frac{1}{|V_c|}\right) \sum_{v \in V_c} l_v(\theta)$
- ¡ Update params: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \nabla \ell_{sub}(\theta) * (\boldsymbol{\theta})$



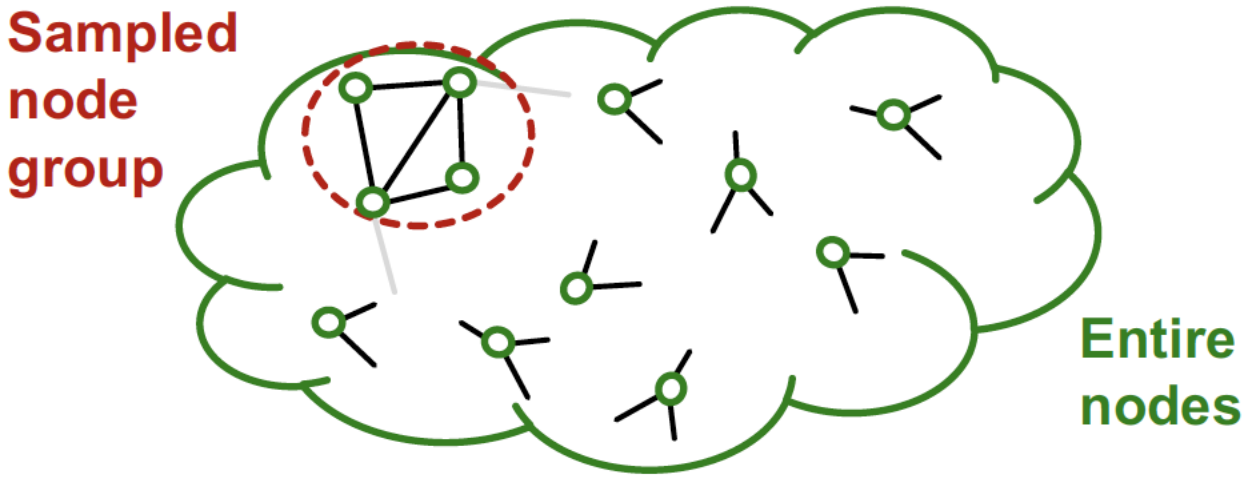
Issues with Cluster-GCN (1)

- 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)

- 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)

- Sampled nodes are not diverse enough to be represent the entire graph structure:

- As a result, the gradient averaged over the sampled nodes, $\left(\frac{1}{|V_c|}\right)\sum_{v \in V_c} l_v(\theta)$, becomes unreliable.
  - Fluctuates a lot from a node group to another.
  - In other words, the gradient has high variance.

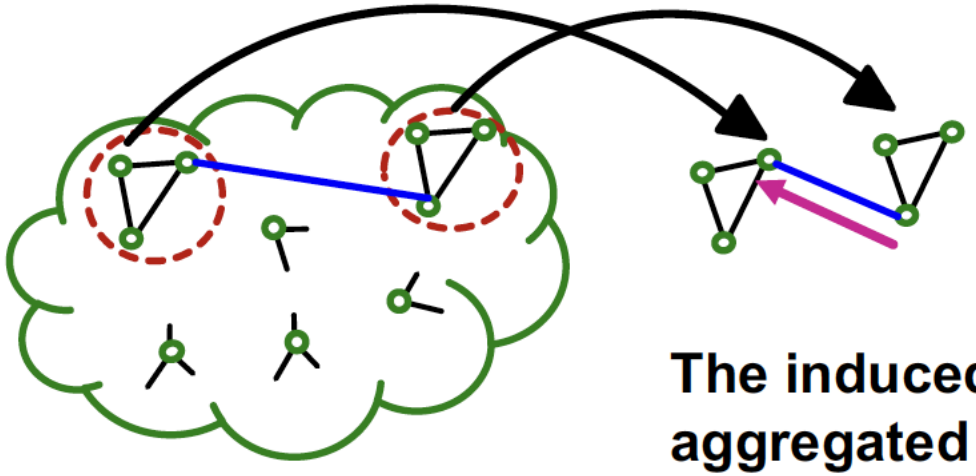
- Leads to slow convergence of SGD

Advanced Cluster-GCN: Overview

- Solution: Aggregate multiple node groups permini-batch.
- Partition the graph into relatively-small groups of nodes.
- For each mini-batch:
  - Sample and aggregate multiple node groups.
  - Construct the induced subgraph of the aggregated node group.
  - The rest is the same as vanilla Cluster-GCN (compute node embeddings and the loss, update parameters)

Advanced Cluster-GCN: Overview

- Why does the solution work?
- Sampling multiple node groups → Makes the sampled nodes more representative of the entire nodes. Leads to less variance in gradient estimation.



The induced subgraph over aggregated node groups
→ Includes edges between groups
→ Message can flow across groups.

Advanced Cluster-GCN

- Similar to vanilla Cluster-GCN, advanced Cluster-GCN also follows a 2-step approach.

1) Pre-processing step:

- Given a large graph $G = (V, E)$, partition its nodes V into C relatively-small groups: $V_1, \ldots, V_C$.
  - $V_1, \ldots, V_C$ needs to be small so that even if multiple of them are aggregated, the resulting group would not be too large.

Advanced Cluster-GCN

2) Mini-batch training:

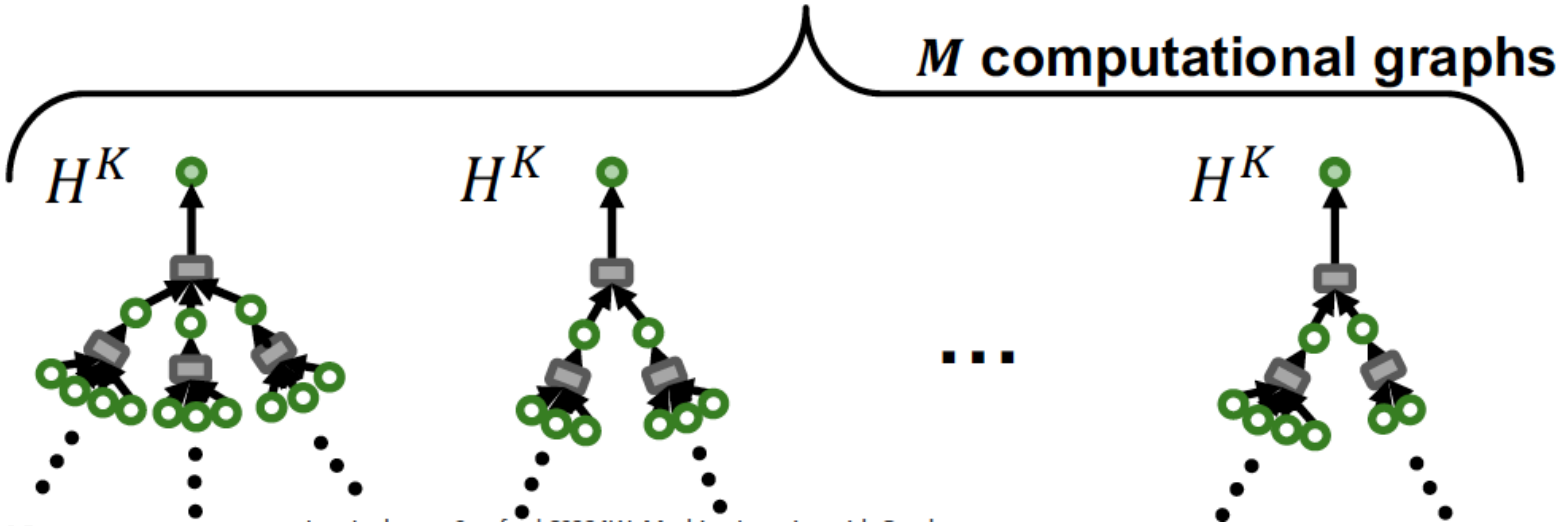
- For each mini-batch, randomly sample a set of $\boldsymbol{q}$ node groups: $\{V_{t_1}, \ldots, V_{t_q}\} \subset \{V_1, \ldots, V_C\}$.
- Aggregate all nodes across the sampled node groups:

$$V_{aggr} = V_{t_1} \cup \cdots \cup V_{t_q}$$

- Extract the induced subgraph $\boldsymbol{G_{aggr} = (V_{aggr}, E_{aggr})}$,
  - where $E_{aggr} = \{(u, v) \mid u, v \in V_{aggr}\}$
  - $\boldsymbol{Eaggr}$ also includes between-group edges!

Comparison of Time Complexity

- Generate $\boldsymbol{M}(\ll \boldsymbol{N})$ node embeddings using
- $\boldsymbol{K}$-layer GNN ($\boldsymbol{N}$: #all nodes).
- Neighbor-sampling (sampling H nodes per layer):
  - For each node, the size of K-layer computational graph is H^K.
- For M nodes, the cost is $\boldsymbol{M} \cdot \boldsymbol{H^K}$



Comparison of Time Complexity

- Generate $\boldsymbol{M}(\ll \boldsymbol{N})$ node embeddings using $\boldsymbol{K}$-layer GNN ($\boldsymbol{N}$: #all nodes).
- Cluster-GCN:
  - Perform message passing over a subgraph induced by the M nodes.
  - The subgraph contains $M \cdot D_{avg}$: edges, where D_{avg} : is the average node degree.
  - K-layer message passing over the subgraph costs at most $\boldsymbol{K} \cdot \boldsymbol{M} \cdot \boldsymbol{D_{avg}}$.

Comparison of Time Complexity

- In summary, the cost to generate embeddings for M nodes using K-layer GNN is:
  - Neighbor-sampling (sample $\boldsymbol{H}$ nodes per layer): $\boldsymbol{M \cdot H^K}$
  - Cluster-GCN: $\boldsymbol{K \cdot M \cdot D_{avg}}$

- Assume $\boldsymbol{H = D_{avg}/2}$. In other words, 50% of neighbors are sampled.
  - Then, Cluster-GCN (cost: $\boldsymbol{2MHK}$) is much more efficient than neighbor sampling (cost: $\boldsymbol{M \cdot H^K}$).
  - Linear (instead of exponential) dependency w.r.t. $\boldsymbol{K}$.

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

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.

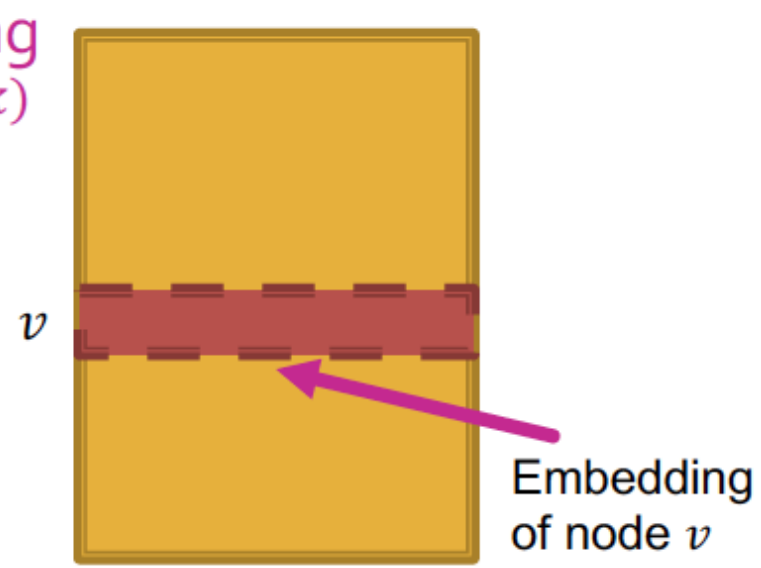
Quick Overview of LightGCN (1)

- Adjacency matrix: $\boldsymbol{A}$
- Degree matrix: $\boldsymbol{D}$
- Normalized adjacency matrix:

$$\tilde{A} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$$

- Let $\boldsymbol{E}^{(k)}$ be the embedding matrix at k-th layer.
- Let $\boldsymbol{E}$ be the input embedding matrix.
  - We backprop into $\boldsymbol{E}$.
- GCN's aggregation in the matrix form

$$E^{(k+1)} = ReLU\left(\tilde{A}\boldsymbol{E}^{(k)}W^{(k)}\right)$$



Quick Overview of LightGCN (2)

- Removing ReLU non-linearity gives us

$$\boldsymbol{E}^{(K)} = \boxed{\tilde{\boldsymbol{A}}^{K}\ \boldsymbol{E}}\ \boldsymbol{W}, \text{ where } \boldsymbol{W} \equiv \boldsymbol{W}^{(0)} \cdots \boldsymbol{W}^{(K-1)}$$

Diffusing node embeddings along the graph

- Efficient algorithm to obtain $\tilde{A}^K E$
  - Start from input embedding matrix $\boldsymbol{E}$.
  - Apply $\boldsymbol{E} \leftarrow \tilde{A}E$
- Weight matrix $\boldsymbol{W}$ can be ignored for now.
  - $\boldsymbol{W}$ acts as a linear classifier over the diffused node embeddings $\tilde{A}^K E$

Differences to LightGCN

- SimplGCN adds self-loops to adjacency matrix $\boldsymbol{A}$:

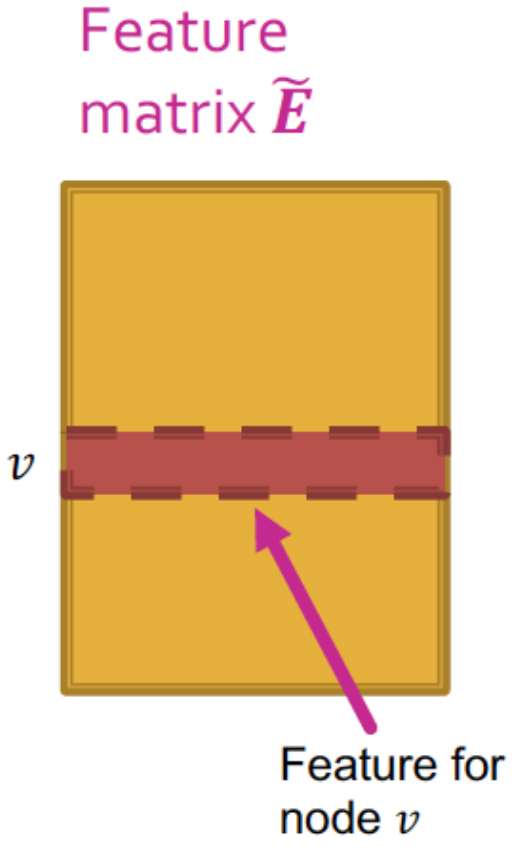
- $\boldsymbol{A} \leftarrow \boldsymbol{A} + \boldsymbol{I}$
  - Follows the original GCN by Kipf & Welling.

- SimplGCN assumes input node embeddings $\boldsymbol{E}$ to be given as features:
  - Input embedding matrix $\boldsymbol{E}$ is fixed rather than learned.
  - Important consequence: $\tilde{A}^K E$ needs to be calculated only once.
    - Can be treated as a pre-processing step.

Simplified GCN: "SimplGCN"

- Let $\tilde{E} = \tilde{A}^K E$ be pre-processed feature matrix.
  - Each row stores the pre-processed feature for each node.
  - $\tilde{E}$ can be used as input to any scalable ML models (e.g., linear model, MLP).

- SimplGCN empirically shows learning a linear model over $\tilde{E}$ often gives performance comparable to GCN!



Comparison with Other Methods

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

- But the model is less expressive (next).

Potential Issue of Simplified GCN

- 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 Simplified GCN

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

- Why?

Graph Homophily

- Many node classification tasks exhibit homophily structure, i.e., nodes connected by edges tend to share the same target labels.

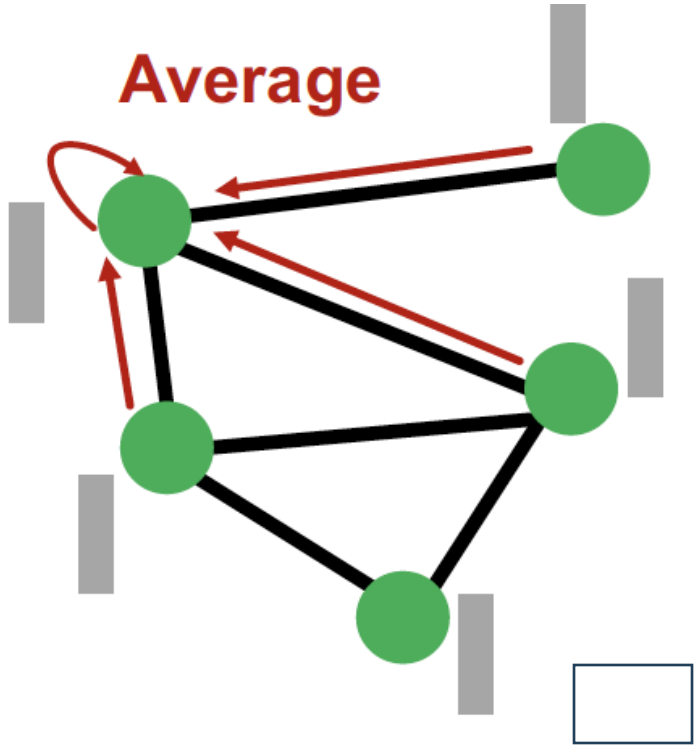
- Examples:
  - Paper category classification in paper-citation network
  - Two papers tend to share the same category if one cites another.
  - Movie recommendation for users in social networks
  - Two users tend to like the same movie if they are friends in a social network.

When does Simplified GCN Work?

- Recall the preprocessing step of the simplified GCN: Do E ← $\tilde{A}$E for K times.
  - $\boldsymbol{E}$ is node feature matrix $\boldsymbol{E} = \boldsymbol{X}$

- 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.

Simplified GCN: 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.