The Henry and Marilyn Taub
Faculty of Computer Science

Module 5: Message-Passing Graph Neural Networks

StructML Course

Benny Kimelfeld
Technion – Israel Institute of Technology

Spring 2026

One Slider

- We have seen how to *engineer features* from graphs for capturing structural patterns
- We have also seen how to learn *node embeddings* from graphs
- Graph Neural Networks (GNNs) maintain a vector representation for each node and repeatedly update it using the embeddings of neighboring nodes
- Multiple rounds of message passing allow GNNs to capture graph patterns
- Since all computations are differentiable, the entire model can be trained end-to-end using gradient-based learning
- However, graphs introduce unique challenges: scaling to large graphs, constructing mini-batches, over-smoothing, and over-squashing

Module Goals

- Learn the general anatomy of message-passing GNNs
- Get a taste of GNN *expressiveness analysis* via the *Weisfeiler-Lehman* test
- Study some *famous GNN architectures*
- Learn how GNNs incorporate *heterogeneity*
- Familiarize with common learning challenges of GNNs
- Learn graph *mini-batching* for GNN learning
- Familiarize with the deep-learning approach to large-scale *link prediction*

Outline

Outline

Setup

- In this and the next part, we assume an undirected featured graph $(V, E, \mathbf{X})$ where $\mathbf{X} : V \to \mathbb{R}^d$
  - We denote the vector $\mathbf{X}(v)$ as $\mathbf{x}_v$
  - We generalize to hetero-graphs in the next section
- With a slight abuse of notation, we view $\mathbf{X}$ as a matrix in $\mathbb{R}^{|V| \times d}$
- We denote by $\mathbf{A}$ the (symmetric) adjacency matrix of G
  - Hence, $\mathbf{A} \in \{0,1\}^{|V| \times |V|}$
  - We assume that $\mathbf{A}$ corresponds to an ordering $v_1, \ldots, v_n$ of V

Abstraction of a Graph Architecture

- A *graph architecture* can be conceptualized as a function $F_{d,\boldsymbol{\theta}}$ that:
  - Takes as input:
    - A featured graph $G = (V, E, \mathbf{X})$ with the feature dimension d
    - An assignment of values to its sequence $\boldsymbol{\theta}$ of parameters
  - Produces a matrix $\mathbf{Y} \in \mathbb{R}^{|V| \times k}$ that associates a result $\mathbf{y}_v$ for every node v
- Hence, we focus on *multivariate node regression* tasks where the goal is to map every node to a vector
  - As the special case of $k = 1$, map each node to a number: *node regression task*
- We sometimes mention *graph-level* tasks, where the goal is to produce a single vector for the entire input
- By setting the parameters to learned values, we get a specific *graph model*
- As notation, we omit the specification of d and write $F(G; \boldsymbol{\theta})$

An Algebraic Notation

- To leverage DNN learning machinery, we typically use the matrix representation of G, that is, $(\mathbf{A}, \mathbf{X})$
- Hence, we can write $F(\mathbf{A}, \mathbf{X}; \boldsymbol{\theta})$ instead of $F(G; \boldsymbol{\theta})$, adopting a full algebraic notation

Starting with a Bad Suggestion

- Suppose that one suggests a graph model in the form of

$$F(\mathbf{A}, \mathbf{X}; \theta) = \text{MLP}(\mathbf{A}_1, \ldots, \mathbf{A}_n, \mathbf{x}_1, \ldots, \mathbf{x}_n; \theta)$$

  where $\mathbf{A}_i$ is the ith row of $\mathbf{A}$

- *What are the problems with this suggestion?*
  1. The MLP cannot have a varying number of inputs; how do we generalize across graphs?
  2. The MLP is sensitive to the ordering of the nodes — if we reorder the nodes and reapply the model, we will get a different result for v!
  3. Also, it is not at all clear how the MLP is utilizing the core structure of the graph, namely, the edges and neighborhoods
     - In particular, how is the MLP expected to know that $\mathbf{x}_j$ is the feature vector of a neighbor v_j of v_i? From the content of $\mathbf{A}_i$?

Node Reordering

- Let us phrase sensitivity to node ordering more formally
- A *permutation* over V is a bijection $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$, representing a reordering of $(v_1, \ldots, v_n)$, namely $(v_{\pi(1)}, \ldots, v_{\pi(n)})$
- The adjacency matrix $\mathbf{A}_\pi$ that corresponds to reordering π is obtained from the original $\mathbf{A}$ permuting the rows and columns according to π, that is:

$$\mathbf{A}_\pi := \mathbf{P}\mathbf{A}\mathbf{P}^\top$$

for the *permutation matrix* $\mathbf{P}$ defined by $\mathbf{P}_{i,j} := \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise} \end{cases}$

- Similarly, the permuted feature matrix $\mathbf{X}_\pi$ is $\mathbf{P}\mathbf{X}$

Permutation and Equivariance

Returning to our desired function $F(\mathbf{A}, \mathbf{X}; \boldsymbol{\theta})$, we can think of two relevant properties

- Permutation invariance: $F(\mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{P}\mathbf{X}; \boldsymbol{\theta}) = F(\mathbf{A}, \mathbf{X}; \boldsymbol{\theta})$
  - We typically want this for graph-level tasks, not node-level tasks!
- Permutation equivariance: $F(\mathbf{P}\mathbf{A}\mathbf{P}^\top, \mathbf{P}\mathbf{X}; \boldsymbol{\theta}) = \mathbf{P}F(\mathbf{A}, \mathbf{X}; \boldsymbol{\theta})$
  - *This is what we want!*
- We will discuss models that are all equivariant
  - It will usually be clear from the definition (but sometimes may require more thinking)

Intuition on MPNNs

- The idea of a message-passing is to compute a value $\mathbf{h}_u$ for each u as follows
  1. We start with the value $\mathbf{h}_u = \mathbf{x}_u$
  2. Then, each u collects the values $\mathbf{h}_v$ of its neighbors v and uses them to update $\mathbf{h}_u$
     - Hence, the value of a node accounts for the value of its neighbors
  3. Next, each node u collects the updated values $\mathbf{h}_v$ of its neighbors v, and again uses these values to update $\mathbf{h}_u$
     - Hence, the value of a node accounts for its entire 2-hop neighborhood
  4. And so on, for a predefined number ℓ of steps
     - Hence, *the value of a node accounts for its entire ℓ-hop neighborhood*
- To assure that the resulting function is equivariant, we make sure to collect data from the neighbors in a way that is insensitive to the **ordering** of the neighbors

Outline

Illustration of an MPNN Layer

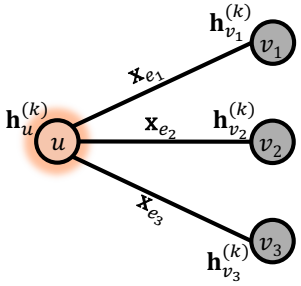


Illustration of an MPNN Layer

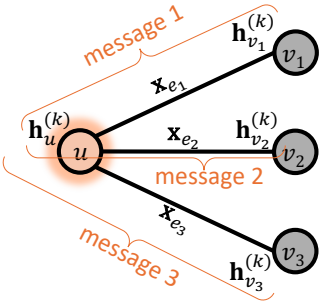
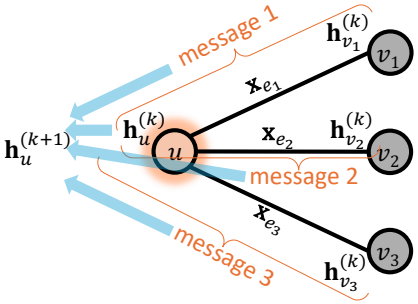


Illustration of an MPNN Layer



Message Functions

- Let $\{u, v\}$ be an edge
- A *message function* is a function $M : \mathbb{R}^{2d} \to \mathbb{R}^{d_M}$ that transforms the values $\mathbf{x}_u$ and $\mathbf{x}_v$ of u and v, respectively, into a *message* $M(\mathbf{x}_u, \mathbf{x}_v)$
- If G has *edge features* $\mathbf{x}_e \in \mathbb{R}^{d_E}$, then these are taken as well into account to create the message $M(\mathbf{x}_u, \mathbf{x}_v, \mathbf{x}_{\{u,v\}})$
  - Hence, $M : \mathbb{R}^{2d+d_E} \to \mathbb{R}^{d_M}$
  - As usual, we will assume that there are no edge features, unless stated explicitly

Message-Passing Layer

A *message-passing layer* is a function $\Lambda(G, u)$ that transforms the feature vector $\mathbf{x}_u$ of the node u into a new vector by combining messages from its neighbors:

$$\Lambda(G, u) := U\Big(\mathbf{x}_u, \alpha\big(\{\!\{M(\mathbf{x}_u, \mathbf{x}_v) \mid v \in N(u)\}\!\}\big)\Big)$$

3 Functions

$$\Lambda(G, u) := U\Big(\mathbf{x}_u, \alpha\big(\{\!\{M(\mathbf{x}_u, \mathbf{x}_v) \mid v \in N(u)\}\!\}\big)\Big)$$

- $M : \mathbb{R}^{2d} \to \mathbb{R}^{d_M}$ is a message function
- α is an *aggregation function* that maps the **bag** of all neighbor messages in $\mathbb{R}^{d_M}$ to a single *aggregated message* in $\mathbb{R}^{d'}$
  - Using *bags* assures that aggregation functions are insensitive to the input ordering
- $U : \mathbb{R}^{d+d'} \to \mathbb{R}^{d''}$ is an *update function* that computes an updated value for u, based on the previous value and the aggregated message
  - Typically, $d = d'$, so $U : \mathbb{R}^{2d} \to \mathbb{R}^{d''}$
- We will say that this message-passing layer is a $d \to d''$ *layer*
  - That is, we start with node embeddings of dimension d and end up with node embeddings of dimension d''

Parameters

- Each of the three functions has *parameters* (that we omit in the presentation)
- As usual, we assume that no parameters are shared among the three
  - Or among different layers, as we define next
- Note: the three functions, along with their parameters, are shared among all nodes

Message Passing Neural Network

- A *message-passing GNN* (MPNN) is a chain of $d_{k-1} \to d_k$ message-passing layers, for $k = 1, \ldots, \ell$, where $d_0 = d$ is the dimension of the source node features

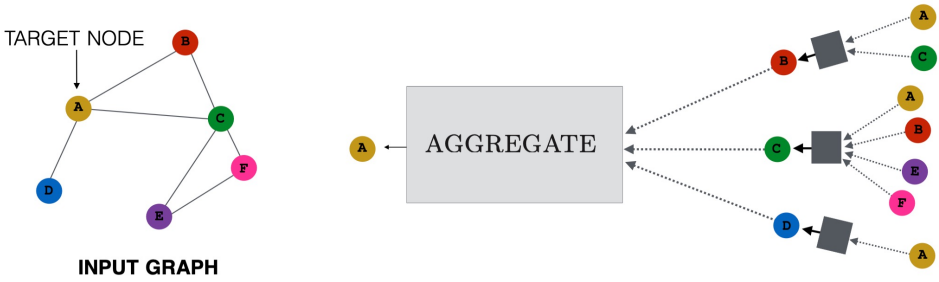
$$\mathbf{h}_u^{(0)} := \mathbf{x}_u \quad \in \mathbb{R}^{d_0}$$
$$\mathbf{h}_u^{(k)} := U^{(k)}\left(\mathbf{h}_u^{(k-1)}, \alpha^{(k)}\left(\{\!\{ M^{(k)}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N(u) \}\!\}\right)\right) \quad \in \mathbb{R}^{d_k}$$

- We sometimes use $\mathbf{m}$ to denote the aggregated message:

$$\mathbf{m}_u^{(k)} := \alpha^{(k)}\left(\{\!\{ M^{(k)}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N(u) \}\!\}\right)$$

- Hence, $\mathbf{h}_u^{(k)} := U^{(k)}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_u^{(k)})$

Illustration of a 2-Layer Simple MPNN



[Hamilton, 2020]

The Output Layer

- In our definition, the *result* is the output of the last layer, namely $\mathbf{h}_u^{(\ell)}$ for all $u \in V$
- For node-level tasks, it is common to have an additional *decision layer* (a.k.a. *output layer*), such as

$$\text{Softmax}(\text{MLP}(\mathbf{h}_u^{(\ell)}))$$

  for multi-class classification
- For graph-level tasks, the decision layer typically involves a *readout layer* that aggregates all node embeddings into a graph-level prediction, e.g.,

$$\text{Softmax}(\text{MLP}(\alpha(\mathbf{H}^{(\ell)}))$$

- Later, we also discuss link-level tasks

Additional Notes about the Definition

- Since a GNN is a *learned* DNN, we assume that all functions are differentiable
- We assume that no parameters are shared between layers, unless explicitly stated otherwise
- When there are edge features, the definition is as follows:

$$\mathbf{h}_u^{(k)} := U^{(k)}\left(\mathbf{h}_u^{(k-1)}, \alpha^{(k)}\left(\{\!\{ M^{(k)}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}, \mathbf{x}_{\{u,v\}}) \mid v \in N(u) \}\!\}\right)\right)$$

  - (The GNN does not update edge values)

Numerical Representation

In DNN formalism, an MPNN is a special case of a *Graph Neural Network* (GNN) of the form:

$$\mathbf{H}^{(0)} := \mathbf{X}$$
$$\mathbf{H}^{(k)} := f^{(k)}(\mathbf{A}, \mathbf{H}^{(k-1)}; \boldsymbol{\theta}^{(k)})$$

where each $f^{(k)}$ is an equivariant function

- That is, for every permutation matrix $\mathbf{P}$ it holds that

$$f^{(k)}(\mathbf{P}\mathbf{A}\mathbf{P}^{\top}, \mathbf{P}\mathbf{H}; \boldsymbol{\theta}^{(k)}) = \mathbf{P} \cdot f^{(k)}(\mathbf{A}, \mathbf{H}; \boldsymbol{\theta}^{(k)})$$

Special Case: Simple MPNNs

- The literature often focuses on a simple class of MPNNs
  - And often defines MPNNs as this class, to begin with
- In this class, message function simply copies the value of the neighbor:

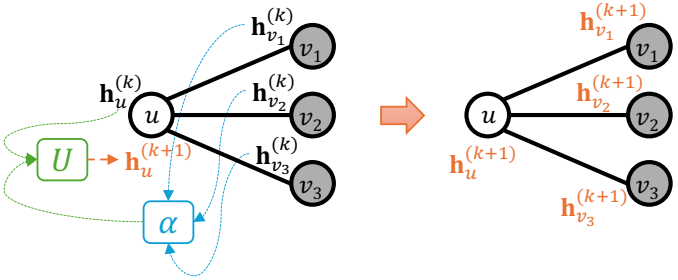
$$M(\mathbf{x}_u, \mathbf{x}_v) = \mathbf{x}_v$$

  - In particular, without accounting for the value of u
  - For example, the aggregated message is the average of the neighbor messages

Definition of a Simple MPNN

$$\mathbf{h}_u^{(0)} := \mathbf{x}_u \quad \in \mathbb{R}^{d_0}$$

$$\mathbf{h}_u^{(k)} := U^{(k)}\left(\mathbf{h}_u^{(k-1)}, \alpha^{(k)}\left(\{\!\{\mathbf{h}_v^{(k-1)} \mid v \in N(u)\}\!\}\right)\right) \quad \in \mathbb{R}^{d_k}, \text{ for } k = 1, \cdots \ell$$



Outline

WL on Featured Graphs

- Recall the WL (more precisely, 1-WL) on non-featured graphs:

  1. Initialize $\kappa_0(u) = 0$ for all $u \in V$
  2. For all $i = 1, \ldots, k$, for all $u \in V$ do:
     $$\kappa_i(u) = \textsc{Hash}\big(\kappa_{i-1}(u), \{\!\{\kappa_{i-1}(v) \mid v \in N(u)\}\!\}\big)$$

- We also adopt 1-WL to featured graphs
- To this aim, we assume that all features are integers, that is, $\mathbf{X} \in \mathbb{Z}^{|V| \times d}$

  1. Initialize $\kappa_0(u) = \textsc{Hash}'(\mathbf{x}_u)$ for all $u \in V$
  2. For all $i = 1, \ldots, k$, for all $u \in V$ do:
     $$\kappa_i(u) = \textsc{Hash}\big(\kappa_{i-1}(u), \{\!\{\kappa_{i-1}(v) \mid v \in N(u)\}\!\}\big)$$

- Here, $\textsc{Hash}' : \mathbb{Z}^d \to \mathbb{N}$ is a perfect hash function

MPNNs Cannot Distinguish what WP Cannot Distinguish

Theorem ([Morris et al., 2019])

Let Γ be a simple MPNN. Let v and v' be two nodes of a graph G. If 1-WL assigns to v and v' the same color after ℓ steps, then Γ assigns to them the same value over ℓ layers.

In other words, if $\kappa_\ell(v) = \kappa_\ell(v')$, then $\mathbf{h}_v^{(\ell)} = \mathbf{h}_{v'}^{(\ell)}$.

In particular, if WL thinks that two graphs G_1 and G_2 are isomorphic, then the MPNN constructs the same matrix $\mathbf{h}^{(\ell)}$ up to a permutation [Xu et al., 2019]

Simple MPNNs Can Simulate 1-WP

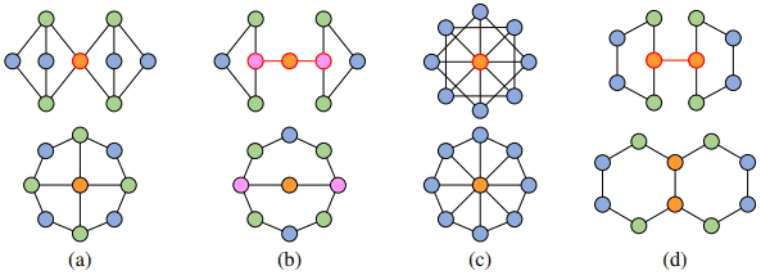
Theorem ([Morris et al., 2019])

For every graph G, there is a simple MPNN Γ with message-passing layers of the form

$$\mathbf{h}_u^{(k)} := \mathrm{ReLU}\left(\mathbf{W}_1^{(k)}\mathbf{h}_u^{(k-1)} + \mathbf{W}_2^{(k)} \sum_{v \in N(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)}\right)$$

such that the following holds for all $\ell \in \mathbb{N}$. For every two nodes v and v', WL assigns to v and v' the same color after ℓ steps if and only if Γ assigns to them the same value over ℓ layers, that is, $\kappa_\ell(v) = \kappa_\ell(v')$ if and only if $\mathbf{h}_v^{(\ell)} = \mathbf{h}_{v'}^{(\ell)}$.

Examples of Failures



https://ai-scholar.tech/en/articles/gnn/GD-WL

Nodes with the same color will get the same value in every MPNN

Outline

Notation

In the following slides, we use the following notation for a node u:

- $\hat{N}(u) := N(u) \cup \{u\}$, that is, the neighborhood of u, *including u itself*
- $\hat{\deg}(u) := |\hat{N}(u)| = \deg(u) + 1$
- $\mathbf{D}$ is the diagonal matrix with $\mathbf{D}_{i,i} = \deg(v_i)$
- $\hat{\mathbf{D}}$ is the diagonal matrix with $\hat{\mathbf{D}}_{i,i} = \hat{\deg}(v_i)$, that is, $\hat{\mathbf{D}} = \mathbf{D} + \mathbf{I}$

Graph Convolutional Networks

- Idea: Update via a *linear transformation*, *normalized* by node degrees
  - Used for avoiding exploding gradients
  - [Kipf and Welling, 2017, Schlichtkrull et al., 2018b]
- Node centric:

$$\mathbf{h}_u^{(k)} = \sigma\left(\sum_{v \in \hat{N}(u)} \frac{1}{\sqrt{(\hat{\deg}(u)) \cdot (\hat{\deg}(v))}} \mathbf{h}_v^{(k-1)} \cdot \mathbf{W}^{(k)}\right)$$

  where σ is a (pointwise) activation function
- Matrix-multiplication form:

$$\mathbf{H}^{(k)} = \sigma\left(\hat{\mathbf{D}}^{-\frac{1}{2}} \cdot \hat{\mathbf{A}} \cdot \hat{\mathbf{D}}^{-\frac{1}{2}} \cdot \mathbf{H}^{(k-1)} \cdot \mathbf{W}^{(k)}\right)$$

  where: $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$

Graph Attention Networks (GAT)

- Idea: each neighbor gets a different coefficient (attention) of a weighted average, based on its value [Velickovic et al., 2018]
- Formally: $\mathbf{h}_u^{(k)} = \sigma\left(\sum_{v \in \hat{N}(u)} \alpha_{u,v}^{(k)} \cdot \mathbf{W}^{(k)} \cdot \mathbf{h}_v^{(k-1)}\right)$
  - $\mathbf{W}^{(k)}$ is a *weight matrix* (of learned parameters)
  - $\mathbf{W}^{(k)} \cdot \mathbf{h}_v^{(k-1)}$ serves as the *value* of the node v
  - $\alpha_{u,v}^{(k)}$ is the attention that u pays to v, defined next
- As in the usual attention mechanism, the $\alpha_{u,v}^{(k)}$ are defined as a Softmax
- Let $\hat{N}(u) = \{v_1, \ldots, v_m\}$; then:

$$\left[\alpha_{u,v_1}^{(k)}, \ldots, \alpha_{u,v_m}^{(k)}\right] := \text{Softmax}\left(\mathbf{e}_u^{(k)}\right)$$

where $\mathbf{e}_u^{(k)}$ is defined next:

Additive Attention Coefficient

- Unlike the common *multiplicative* attention coefficient that we learned, $\mathbf{e}_{u,v'}^{(k)}$ is defined in an *additive manner*
- Specifically:

$$[\mathbf{e}_{u,v_1}^{(k)}, \ldots, \mathbf{e}_{u,v_m}^{(k)}] := \text{LeakyReLU}\left(\mathbf{a}^{(k)}[\mathbf{W}^{(k)}\mathbf{h}_u^{(k-1)} \oplus \mathbf{W}^{(k)}\mathbf{h}_{v_1}^{(k-1)}], \ldots, \mathbf{a}^{(k)}[\mathbf{W}^{(k)}\mathbf{h}_u^{(k-1)} \oplus \mathbf{W}^{(k)}\mathbf{h}_{v_m}^{(k-1)}]\right)$$

  - Here, $\hat{N}(u) = \{v_1, \ldots, v_m\}$
  - Recall: $\oplus$ is the concatenation operator
  - $\mathbf{a}^{(k)}$ is a *weight vector* (of learned parameters)
  - Rather degenerated version of attention:
    - $\mathbf{W}^{(k)}\mathbf{h}_{v'}$ serves as both the *query* and the *key* of v'
    - Same weight vector $\mathbf{W}^{(k)}$ for the queries, keys, and values

Multi-head GAT

- A *multi-head* version is defined similarly to the ordinary multi-head attention
- In this case, $\mathbf{h}_u^{(k)}$ is the concatenation of h functions:

$$\mathbf{h}_u^{(k)} := \mathbf{h}_{1,u}^{(k)} \oplus \cdots \oplus \mathbf{h}_{h,u}^{(k)}$$

where each $\mathbf{h}_{i,u}^{(k)}$ is defined using its own $\mathbf{W}_i^{(k)}$ and $\mathbf{a}_i^{(k)}$

Experiments from the GAT Paper (Publication Datasets)

Method	Cora	Citeseer	Pubmed
MLP	55.1%	46.5%	71.4%
ManiReg (Belkin et al., 2006)	59.5%	60.1%	70.7%
SemiEmb (Weston et al., 2012)	59.0%	59.6%	71.7%
LP (Zhu et al., 2003)	68.0%	45.3%	63.0%
DeepWalk (Perozzi et al., 2014)	67.2%	43.2%	65.3%
ICA (Lu & Getoor, 2003)	75.1%	69.1%	73.9%
Planetoid (Yang et al., 2016)	75.7%	64.7%	77.2%
Chebyshev (Defferrard et al., 2016)	81.2%	69.8%	74.4%
GCN (Kipf & Welling, 2017)	81.5%	70.3%	**79.0%**
MoNet (Monti et al., 2016)	$81.7 \pm 0.5\%$	—	$78.8 \pm 0.3\%$
GCN-64*	$81.4 \pm 0.5\%$	$70.9 \pm 0.5\%$	$\mathbf{79.0} \pm 0.3\%$
GAT (ours)	$\mathbf{83.0} \pm 0.7\%$	$\mathbf{72.5} \pm 0.7\%$	$\mathbf{79.0} \pm 0.3\%$

- Same #ftrs
- Same depth (2)

2 layers, 8 attention heads, single attention head for the second, 8 features

Graph Isomorphism Networks

- GIN [Xu et al., 2019] are inspired by the WL test
  - Replacing the input of HASH with a sum
  - Replacing HASH with an MLP

$$\mathbf{h}_u^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1 + \varepsilon^{(k)}\right) \mathbf{h}_u^{(k-1)} + \sum_{v \in N(u)} \mathbf{h}_v^{(k-1)} \right)$$

- Theoretically, such a function (with some parameters) can have the separation power of 1-WL [Xu et al., 2019]
  - Using fact that MLPs can approximate any permutation-invariant function due to [Hornik et al., 1989]
- They show that GIN has good performance on classification, already for a one-layer MLP (i.e., a perceptron)

Experiments from [Xu et al., 2019]

	Datasets	IMDB-B	IMDB-M	RDT-B	RDT-M5K	COLLAB	MUTAG	PROTEINS	PTC	NCI1
Datasets	# graphs	1000	1500	2000	5000	5000	188	1113	344	4110
	# classes	2	3	2	5	3	2	2	2	2
	Avg # nodes	19.8	13.0	429.6	508.5	74.5	17.9	39.1	25.5	29.8
Baselines	WL subtree	73.8 ± 3.9	50.9 ± 3.8	81.0 ± 3.1	52.5 ± 2.1	78.9 ± 1.9	90.4 ± 5.7	75.0 ± 3.1	59.9 ± 4.3	$\mathbf{86.0 \pm 1.8}$ *
	DCNN	49.1	33.5	–	–	52.1	67.0	61.3	56.6	62.6
	PATCHYSAN	71.0 ± 2.2	45.2 ± 2.8	86.3 ± 1.6	49.1 ± 0.7	72.6 ± 2.2	$\mathbf{92.6 \pm 4.2}$ *	75.9 ± 2.8	60.0 ± 4.8	78.6 ± 1.9
	DGCNN	70.0	47.8	–	–	73.7	85.8	75.5	58.6	74.4
	AWL	74.5 ± 5.9	51.5 ± 3.6	87.9 ± 2.5	54.7 ± 2.9	73.9 ± 1.9	87.9 ± 9.8	–	–	–
GNN variants	SUM–MLP (**GIN-0**)	$\mathbf{75.1 \pm 5.1}$	$\mathbf{52.3 \pm 2.8}$	$\mathbf{92.4 \pm 2.5}$	$\mathbf{57.5 \pm 1.5}$	$\mathbf{80.2 \pm 1.9}$	$\mathbf{89.4 \pm 5.6}$	$\mathbf{76.2 \pm 2.8}$	$\mathbf{64.6 \pm 7.0}$	$\mathbf{82.7 \pm 1.7}$
	SUM–MLP (**GIN-ϵ**)	$\mathbf{74.3 \pm 5.1}$	$\mathbf{52.1 \pm 3.6}$	$\mathbf{92.2 \pm 2.3}$	$\mathbf{57.0 \pm 1.7}$	$\mathbf{80.1 \pm 1.9}$	$\mathbf{89.0 \pm 6.0}$	$\mathbf{75.9 \pm 3.8}$	63.7 ± 8.2	$\mathbf{82.7 \pm 1.6}$
	SUM–1–LAYER	74.1 ± 5.0	$\mathbf{52.2 \pm 2.4}$	90.0 ± 2.7	55.1 ± 1.6	$\mathbf{80.6 \pm 1.9}$	$\mathbf{90.0 \pm 8.8}$	$\mathbf{76.2 \pm 2.6}$	63.1 ± 5.7	82.0 ± 1.5
	MEAN–MLP	73.7 ± 3.7	$\mathbf{52.3 \pm 3.1}$	50.0 ± 0.0	20.0 ± 0.0	79.2 ± 2.3	83.5 ± 6.3	75.5 ± 3.4	$\mathbf{66.6 \pm 6.9}$	80.9 ± 1.8
	MEAN–1–LAYER (GCN)	74.0 ± 3.4	51.9 ± 3.8	50.0 ± 0.0	20.0 ± 0.0	79.0 ± 1.8	85.6 ± 5.8	76.0 ± 3.2	64.2 ± 4.3	80.2 ± 2.0
	MAX–MLP	73.2 ± 5.8	51.1 ± 3.6	–	–	–	84.0 ± 6.1	76.0 ± 3.2	64.6 ± 10.2	77.8 ± 1.3
	MAX–1–LAYER (GraphSAGE)	72.3 ± 5.3	50.9 ± 2.2	–	–	–	85.1 ± 7.6	75.9 ± 3.2	63.9 ± 7.7	77.7 ± 1.5

$\epsilon = 0$ vs. learned ϵ

Classification tasks over *bioinformatics* and *social network* datasets [Xu et al., 2019]

Outline

Outline

Options for MPNNs on Directed Graphs

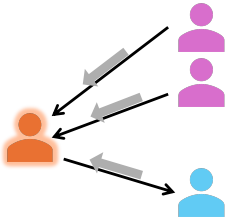
- Recall the definition of the message-passing layer for undirected graphs:

$$\mathbf{h}_u^{(k)} := U^{(k)}\left(\mathbf{h}_u^{(k-1)}, \mathbf{m}_u^{(k)}\right) \quad \text{where} \quad \mathbf{m}_u^{(k)} := \alpha^{(k)}\left(\{\!\{\mathbf{h}_v^{(k-1)} \mid v \in N(u)\}\!\}\right)$$

- What do we do when G is directed?
- Several options [Rossi et al., 2023]:
  - MPNN-U: Ignore the directions
  - MPNN-D: Let messages flow only in the edge direction
  - Dir-GNN: Use two $\mathbf{m}_u^{(k)}$s—one for each direction (described next)

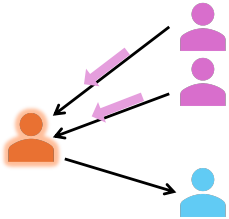
Illustrating the Difference on a Social Network (Follows)

MPNN-U:



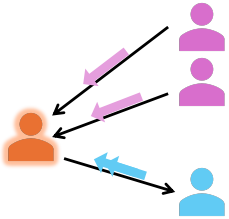
MP does not distinguish between "follows" and "followed-by"

MPNN-D:



No messages are sent from neighbors who are followed

Dir-GNN:



Messages come from both types while distinguishing between the two

Dir-GNN is the Way to Go

- Recall:
  - The GNN collects information along connections to come up with a good prediction
  - Edge directions express some intrinsic semantics of the connection in our data
- Unlike MPNN-D, we should account for edges regardless of their direction
  - *No reason to think that an MPNN can benefit from only one direction*
- Unlike MPNN-U, we should account for the direction itself, as its intrinsic semantics may be quite informative for our prediction
- **Dir-GNN** seem like the way to do

Formal Definition of a Dir-GNN

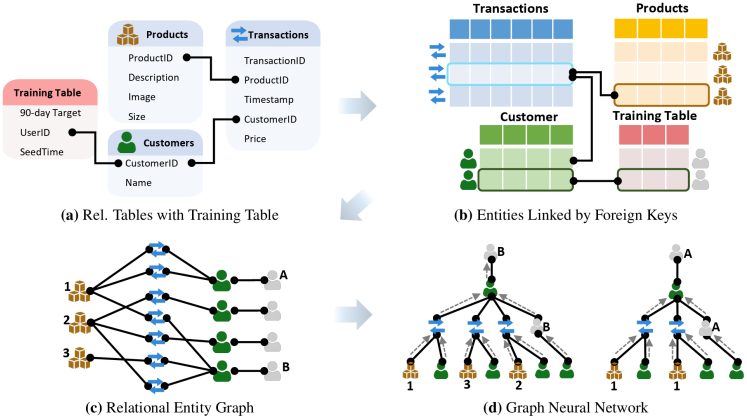
- Use a proprietary message function for each direction:

$$\mathcal{B}_{\rightarrow}(u) := \{\!\{ M_{\rightarrow}^{(k)}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N^{\text{out}}(u) \}\!\}$$
$$\mathcal{B}_{\leftarrow}(u) := \{\!\{ M_{\leftarrow}^{(k)}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N^{\text{in}}(u) \}\!\}$$

- Then: $\mathbf{h}_u^{(k)} := U^{(k)}\left(\mathbf{h}_u^{(k-1)}, \alpha^{(k)}\left(\mathcal{B}_{\rightarrow}(u), \mathcal{B}_{\leftarrow}(u)\right)\right)$
- Note that $\alpha^{(k)}(\cdot,\cdot)$ aggregates the two input bags into a single vector
- In the next part, we will extend this approach to incorporate both *directions* and *labels*

Outline

Context: Relational Learning



(a) Rel. Tables with Training Table

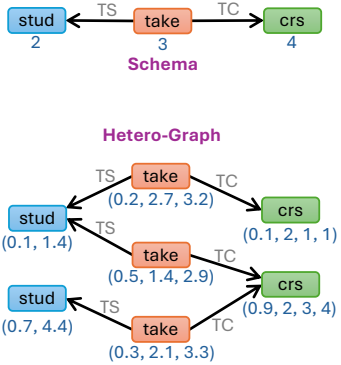
(b) Entities Linked by Foreign Keys

(c) Relational Entity Graph

(d) Graph Neural Network

[Fey et al., 2024]

Recalling Hetero-Graphs and Schemas



- A *hetero-graph schema* is a tuple (L_v, L_e, M, δ) where:
  - L_v is a set of node labels
  - L_e is a set of edge labels
  - $M : L_e \to L_v \times L_v$ maps every edge label into a pair of node labels
  - $\delta : L_v \to \mathbb{N}$ is a dimension function
- A hetero-graph $G = (L_V, L_E, V, E, \lambda, d, \mathbf{X})$ over the schema (L_v, L_e, δ, M) satisfies:
  - $L_V = L_v$ and $L_E = L_e$
  - Every edge (u, τ, v) is such that $M(\tau) = (\lambda(u), \lambda(v))$
  - $d = \delta$

Heterogeneous MPNN

- Importantly, a *hetero-MPNN* is designed for graphs of a specific hetero-graph schema (L_v, L_e, M, δ)
- In particular, labels and their dimensions are consistent among graph instances (including those used for training/validation/testing)
- Main uses cases: relational databases, knowledge graphs

Enriching Edge Labels

- Let $\tau \in L_e$ be an edge label with $M(\tau) = (\rho_1, \rho_2)$
- We say that τ is an *outgoing edge label* of ρ_1 and an *incoming edge label* of ρ_2
- If $\lambda(u) = \rho_1$, then we denote by $N_{\rightarrow\tau}(u)$ the set of outgoing neighbors of u via τ-labeled edges
- If $\lambda(u) = \rho_2$, then we denote by $N_{\leftarrow\tau}(u)$ the set of incoming neighbors of u via τ-labeled edges
- We denote by $L_e^{\leftrightarrow}$ the *enriched label set* $\{\rightarrow\tau \mid \tau \in L_e\} \cup \{\leftarrow\tau \mid \tau \in L_e\}$
- If $\tau' \in L_e^{\leftrightarrow}$ is such that $N_{\tau'}(u)$ is not defined above (i.e., the label of u does not match τ'), then we use the convention of $N_{\tau'}(u) = \emptyset$
- We will define hetero-MPNNs using the enriched label set $L_e^{\leftrightarrow}$ rather than the original L_e

Hetero-MPNN Message Function

- A *hetero-MPNN* uses a unique message function for each incoming and outgoing edge label of u

$$\mathcal{B}_{\to\tau}(u) := \{\!\{ M^{(k)}_{\to\tau}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N_{\to\tau}(u) \}\!\}$$
$$\mathcal{B}_{\leftarrow\tau}(u) := \{\!\{ M^{(k)}_{\leftarrow\tau}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N_{\leftarrow\tau}(u) \}\!\}$$

- In other words, for each $\tau' \in L_{\mathrm{e}}^{\leftrightarrow}$ we define:

$$\mathcal{B}_{\tau'}(u) := \{\!\{ M^{(k)}_{\tau'}(\mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}) \mid v \in N_{\tau'}(u) \}\!\}$$

- Note: this is equivalent to adding to G the *reverse* edge labels and edges, and considering only incoming (or outgoing) edges for message passing (MPNN-D)

Hetero-MPNN Aggregation Function

- We aggregate and update with parameters specialized for $\lambda(u)$:

$$\mathbf{h}_u^{(k)} := U_{\lambda(u)}^{(k)}\left(\mathbf{h}_u^{(k-1)}, \alpha_{\lambda(u)}^{(k)}\left(\mathcal{B}_{\tau_1'}(u), \ldots, \mathcal{B}_{\tau_q'}(u)\right)\right)$$

  for a predefined ordering $\tau_1', \ldots, \tau_q'$ of $L_{\mathrm{e}}^{\leftrightarrow}$

- Note 1: $\alpha_{\lambda(u)}^{(k)}$ aggregates its input sequence of bags into a single vector
- Note 2: if u_1 and u_2 are such that $\lambda(u_1) = \lambda(u_2)$, then $\mathbf{h}_{u_1}^{(k)}$ and $\mathbf{h}_{u_2}^{(k)}$ are defined using the exact same functions

Outline

Relational GCN (R-GCN)

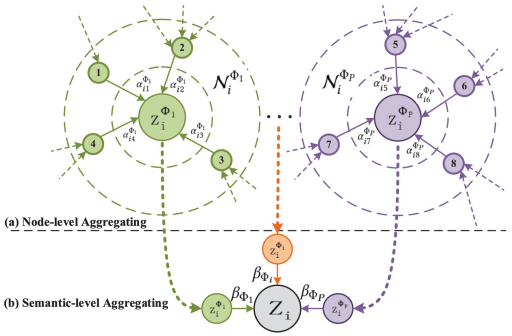
An adaptation of GCNs to heterogeneous ("relational") graphs [Schlichtkrull et al., 2018a]:

$$\mathbf{h}_u^{(k)} := \sigma\left(\mathbf{W}^{(k)}\mathbf{h}_u^{(k-1)} + \sum_{\tau' \in L_e^{\leftrightarrow}} \sum_{v \in N_{\tau'}(u)} \frac{1}{|N_{\tau'}(u)|} \cdot \mathbf{W}_{\tau'}^{(k)} \cdot \mathbf{h}_v^{(k-1)}\right)$$

Regarding the normalization factor $N_{\tau'}(u)$:

- We do not encounter division by zero when $N_{\tau'}(u)$ is empty, since we do not reach the division in the summation in that case
- The authors discuss the option of *learning* a normalization factor $c_{\tau'}(u)$ instead of $|N_{\tau'}(u)|$

Heterogeneous Attention Network (HAN)



[Wang et al., 2019]

- Two levels of *attention mechanisms* to update $\mathbf{h}_u^{(k)}$ from the $\mathbf{h}_v^{(k-1)}$:
  - Node-level: For each label $\tau' \in L_e^{\leftrightarrow}$, attention layer (weighted average) $\mathbf{z}_{\tau',u}$ over all neighbors $v \in N'_\tau(u)$
    - Layer specialized to τ'
  - Semantic level: Build an attention over the $\mathbf{z}'_{\tau,u}$ for all $\tau' \in L_e^{\leftrightarrow}$
- Strengthening:
  - Multi-head attention
  - Looks beyond edges into paths using *meta-paths* (next)

Recalling Meta-Paths

- A *meta-path* Φ over $(L_{\mathrm{v}}, L_{\mathrm{e}}, M, d)$ is a sequence of the form

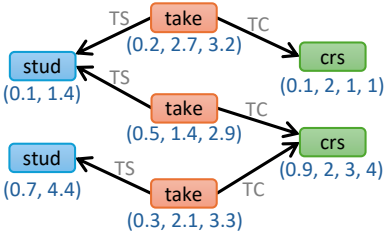
$$\sigma_0 \xrightarrow{\tau_1} \sigma_1 \xrightarrow{\tau_2} \sigma_2 \ldots \sigma_{k-1} \xrightarrow{\tau_k} \sigma_k$$

where:

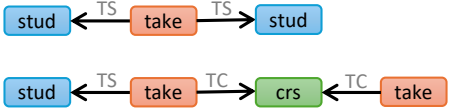
- Each σ_i is in L_{v}
- Each τ_i is in L_{e}
- $M(\tau_i) = (\sigma_{i-1}, \sigma_i)$ or $M(\tau_i) = (\sigma_i, \sigma_{i-1})$ for $i = 1, \ldots, k$

Illustration

Hetero-graph



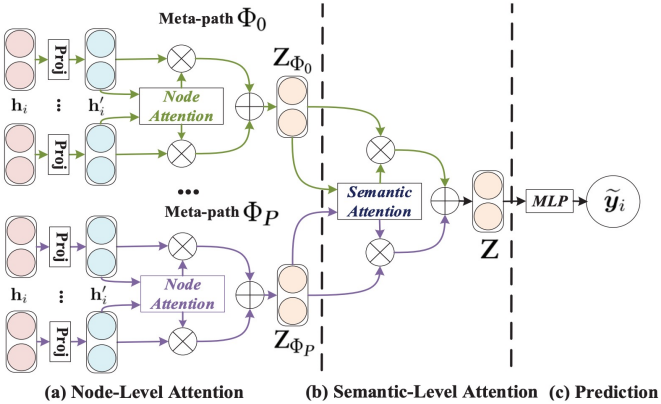
Meta-paths



Enriching a Hetro-Graph with Meta-Paths

- The Φ-*neighborhood* of a node u, denoted $N_\Phi(u)$, is the set of all nodes v that can be reached through a path $u = v_1, v_2, \ldots, v_k = v$ that conforms to Φ
- Architectures that utilize meta-paths (such as HAN):
  - Have a predefined set $\mathbf{\Phi}$ of meta paths
  - Traverse all relevant Φ-neighborhoods, where $\Phi \in \mathbf{\Phi}$ rather than only the immediate neighborhoods connected by edges

The Bigger HAN Architecture

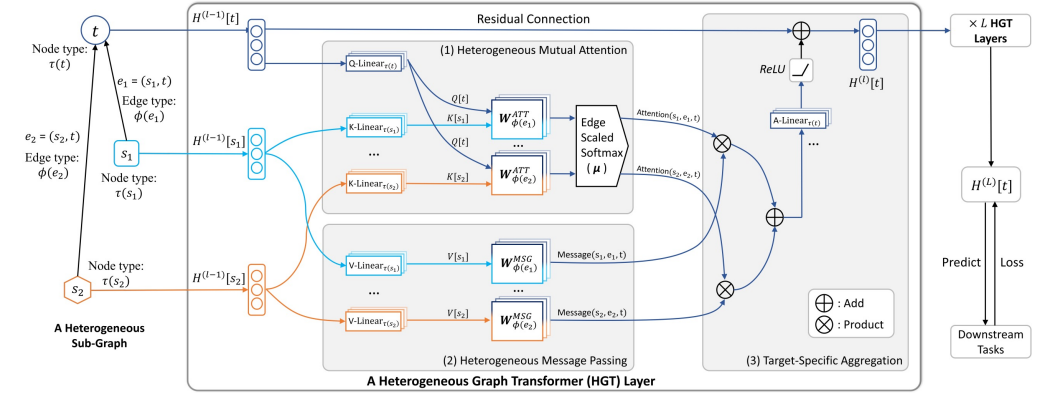


[Wang et al., 2019]

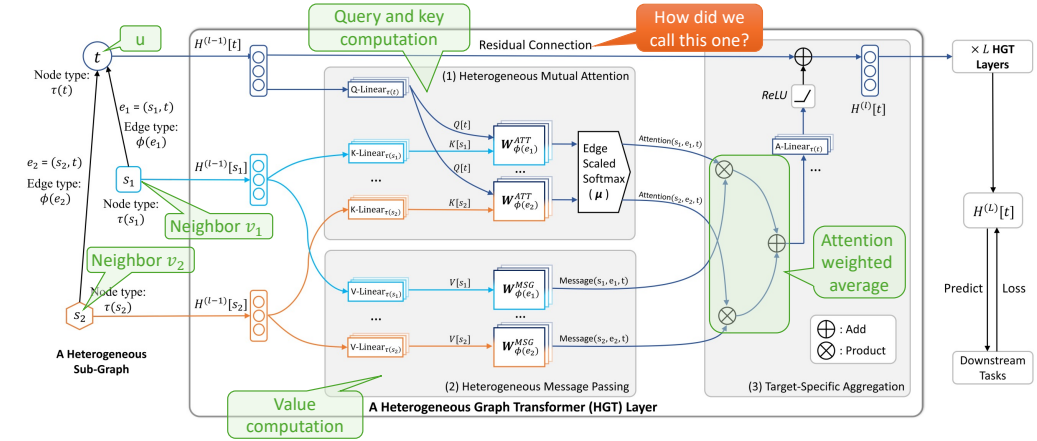
Heterogeneous Graph Transformer (HGT)

- HGT [Hu et al., 2020] is also a hetro-architecture based on (multi-head) attention
- All neighbors are attended (labels mixed)
- The computation of the keys, queries, and values uses *label-specific parameters*
- Utilizes meta-paths similarly to HAN
  - But no second layer of attention over the meta-paths
- Implementation includes a utilization of the attention mechanism to learn useful meta-paths

HGT Architecture [Hu et al., 2020]



HGT Architecture [Hu et al., 2020]



Outline

Outline

Learning Challenges in GNNs

- The usual deep-learning challenges of gradient-based learning still hold, specifically *vanishing/exploding gradients*
- There are also graph-specific challenges
- Over-smoothing: Node embeddings become too similar as layers are stacked
  - Common remedies: skip connections, normalization
- Over-squashing: Information from distant nodes is compressed into fixed-size embeddings, so long-distance impacts are diminished
  - Long-distance impacts are "squashed" through the layers
  - Common remedies: *graph rewiring* (shortcuts), attention mechanisms

Illustration of Over-Smoothing [Keriven, 2022]

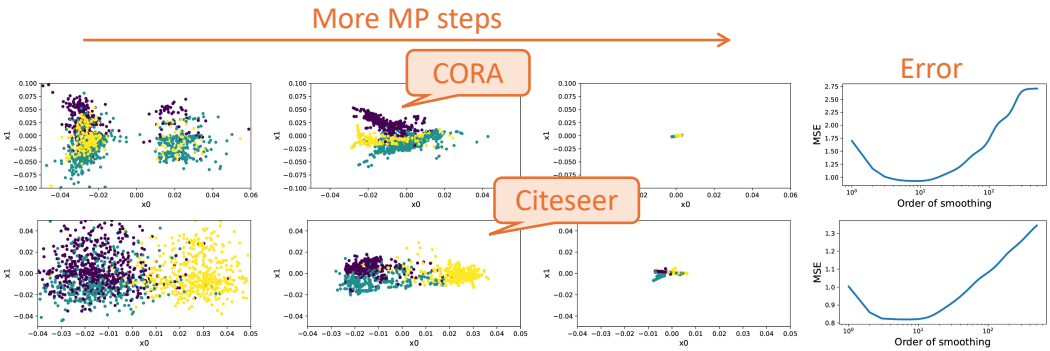
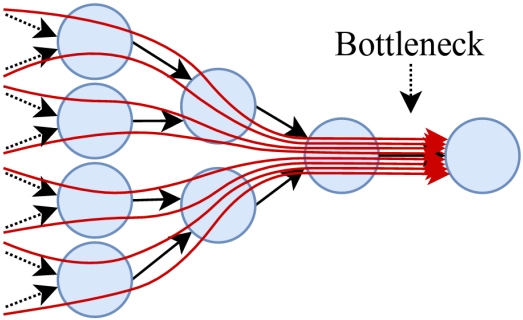


Illustration of Over-Squashing [Alon and Yahav, 2021]



Comparing the Anomalies

- Both over-smoothing and over-squashing arise from the practice of passing messages through multiple layers
- Yet, in a sense, they tackle opposite issues
- Consider two nodes v_1 and v_2 with opposite (real) classes
  - The ℓ-hop neighborhoods may, in general, look very similar, even through their immediate (e.g., 1-hop) neighborhoods are different; these differences might be diminished by incorporating the farther layers
    - *Which issue is that?*
  - On the other hand, they may look similar in their immediate neighborhood, but have good signals farther away; alas, these distinctions are hardly noticed among many many other items in the bigger neighborhood
    - *Which issue is that?*

Outline

Database Sizes in Learning Benchmarks

Name	Domain	#Tasks	Tables		
			#Tables	#Rows	#Cols
rel-amazon	E-commerce	7	3	15,000,713	15
rel-avito	E-commerce	4	8	20,679,117	42
rel-event	Social	3	5	41,328,337	128
rel-f1	Sports	3	9	74,063	67
rel-hm	E-commerce	3	3	16,664,809	37
rel-stack	Social	5	7	4,247,264	52
rel-trial	Medical	5	15	5,434,924	140
Total		30	51	103,466,370	489

RelBench Benchmark [Robinson et al., 2024]

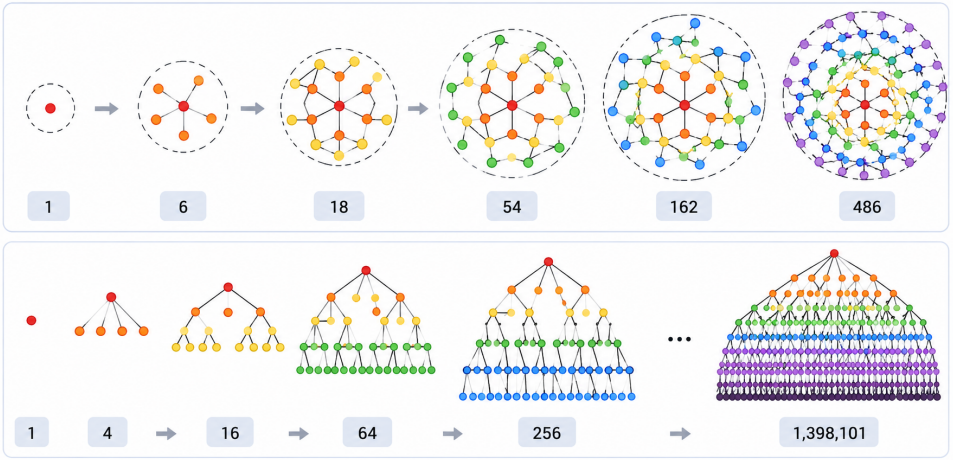
Dataset	# Tables	# Columns	# Rows
AVS	3	24	349,967,371
Outbrain (OB)	8	31	2,170,441,217
Diginetica (DN)	5	28	3,672,396
RetailRocket (RR)	3	11	23,033,676
Amazon (AB)	3	15	24,291,489
StackExchange (SE)	7	49	5,399,818
MAG	5	13	21,847,396
Seznam (SZ)	4	14	2,681,983

4DBInfer Benchmark [Wang et al., 2024]

Batch Learning of GNNs?

- Common situation of node classification: we wish to train a GNN based on a single large graph G and a set $T = \{(v_1, y_1), \ldots, (v_m, y_m)\}$ of examples
  - Each v_i is a node of G ; each y_i is a label
  - Equivalently, we have one big database, and labels over one of its tables
- In principle, we can learn the weights of a GNN by running vanilla GD on (T, G) by processing the entire loss function $L(T, G)$ in every step
- However, the processed formula may be too big (to fit the GPU), and we wish to adopt SGD or mini-batch SGD, as is conventional in DNN training
- But, unlike tabular learning, even a batch computation may reach a large portion of the graph!
  - Or even the entire graph!

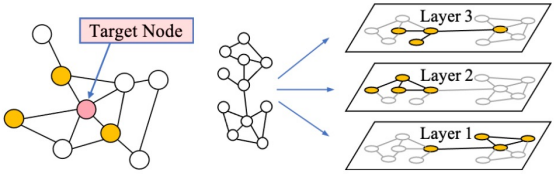
Illustration



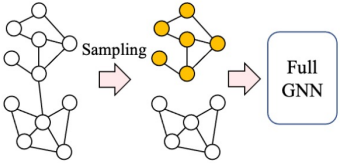
Mini-Batch How?

- **How can we control the size of a batch?**
- $T = \{(v_1, y_1), \ldots, (v_m, y_m)\}$
- For each batch $B \subseteq T$ (or even a single node v_i), the formula involves each node's ℓ-hop neighborhood, which may again be too large
- As an alternative, we can sample a random subgraph of G; but then **we will obtain useless batches**
  - We will likely get disconnected fragments that do not allow the GNN to activate
- Instead, we employ *GNN-aware* sampling techniques in order to produce batches B

3 Sampling Strategies: Node, Layer, Graph



(a) Node-wise. (b) Layer-wise.



(c) Graph-wise.

[Wu et al., 2022]

Node-Wise Sampling

- General idea:
  - Select a batch $B \subseteq T$ of target nodes for the upper layer
  - For each node v_i, sample independently a small set of neighbors for the lower layer
  - Continue doing so for the new sampled nodes, going down the layers
- Prominent example: **SAGE** [Hamilton et al., 2017]
  - Simple sampling of a fixed number of neighbors per layer:
    1. Starting with a batch B, define $B_\ell := B$
    2. Define $B_{\ell-1} := B_\ell$; add to $B_{\ell-1}$ a fixed number of random neighbors for each $v_i \in B_\ell$
    3. Continue that for $k = \ell - 2, \ldots, 0$
    4. Each layer $\mathbf{h}_k$ is trained on B_k, with neighbors restricted to those selected for B_{k-1}
  - Follow-up work developed more sophisticated sampling to maximize the size-utility tradeoff (e.g., **VR-GCN** [Chen et al., 2018b] applies *variation reduction*)

Full Sage Algorithm (Forward Phase)

$\mathcal{B}^K \leftarrow \mathcal{B}$;
for $k = K...1$ **do**
  $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$;
  for $u \in \mathcal{B}^k$ **do**
    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$;
  end
end
$\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$;
for $k = 1...K$ **do**
  for $u \in \mathcal{B}^k$ **do**
    $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\})$;
    $\mathbf{h}_u^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k)\right)$;
    $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$;
  end
end
$\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$

[Hamilton et al., 2017]

Full Sage Algorithm (Forward Phase)

$\mathcal{B}^K \leftarrow \mathcal{B}$;
for $k = K...1$ **do**
  $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$;
  for $u \in \mathcal{B}^k$ **do**
    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$;
  end
end
$\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$;
for $k = 1...K$ **do**
  for $u \in \mathcal{B}^k$ **do**
    $\mathbf{h}^k_{\mathcal{N}(u)} \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}^{k-1}_{u'}, \forall u' \in \mathcal{N}_k(u)\})$;
    $\mathbf{h}^k_u \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}^{k-1}_u, \mathbf{h}^k_{\mathcal{N}(u)})\right)$;
    $\mathbf{h}^k_u \leftarrow \mathbf{h}^k_u / \|\mathbf{h}^k_u\|_2$;
  end
end
$\mathbf{z}_u \leftarrow \mathbf{h}^K_u, \forall u \in \mathcal{B}$

Graph sampling for the batch

Considering only the layer's neighbors

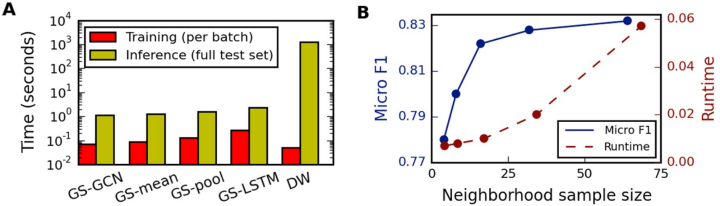
Layer normalization

[Hamilton et al., 2017]

- In the experiments, used sampled graphs of 500 or fewer nodes
- Studied MPNNs with 4 aggregation functions: *mean*, *max*, *LSTM* (which is **not** permutation equivariant), and *convolution* (GCNConv)

Experiments from the SAGE Paper

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600
% gain over feat.	39%	46%	55%	63%	19%	45%



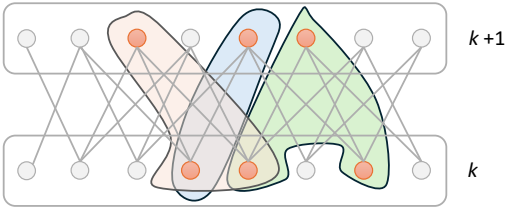
[Hamilton et al., 2017]

Layer-Wise Sampling

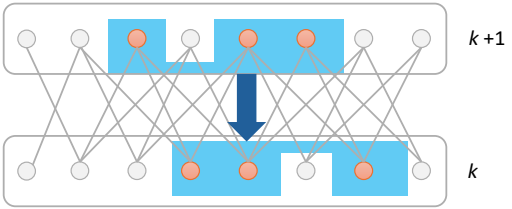
- General idea:
  - Like node-wise, we start at the uppermost layer and sample the lower layer iteratively
  - However, in layer-wise sampling, we sample the full set of nodes of the next layer from a distribution that accounts for all nodes of the upper layer
    - In other words, instead of sampling neighbors from individual nodes, we sample a full layer from a full layer
- Prominent examples (on GCNs):
  - **FastGCN** [Chen et al., 2018a]: samples each layer independently, based on a precomputed node distribution
    - Applicable to *dense* graphs
  - **AS-GCN** (adaptive sampling) [Huang et al., 2018]: samples from the layer k by estimating the joint probability of each node based on the full previous layer

Illustration of Node vs. Layer Sampling

Node-wise



Layer-wise

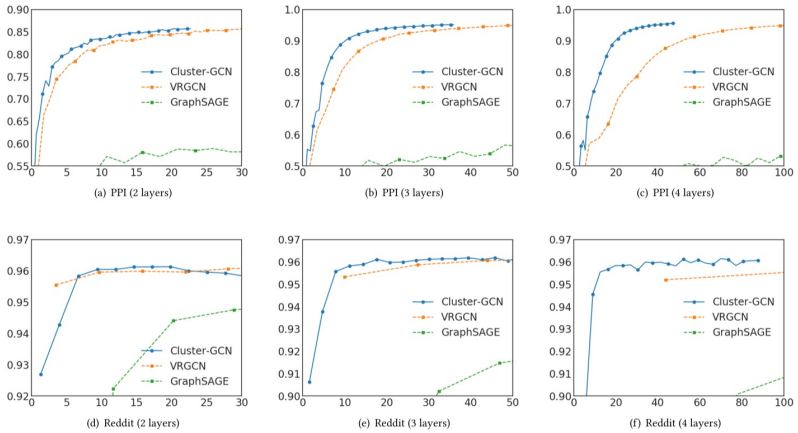


Graph-Wise Sampling

- General idea:
  - A batch is a *random subgraph* G_i of G
  - The GNN (forward/backward) is applied too the full G_i
- Prominent examples:
  - **Cluster-GCN**: [Chiang et al., 2019]
    - Motivation: we want the training to miss as few connections as possible
    - Uses a *graph clustering* algorithm[1] to construct a collection $\{G_1, \ldots, G_p\}$ of clusters
    - 2 versions: *vanilla version*: every sample is a random G_i ; *stochastic multiple partitions*: every sample is a union of multiple G_is (to account for cross-sample edges)
  - **GraphSAINT** [Zeng et al., 2020]
    - Presents and analyzes 3 subgraph-sampling techniques: (1) *random nodes*, (2) *random edges*, (3) *random walks* ; use normalization to account for subgraph bias
    - Each node/edge/walk is sampled independently from a pre-computed distribution

[1] METIS clustering in their implementation

Experiments from the Cluster-GCN Paper



[Chiang et al., 2019] F1-score (y-axis) vs. training time (x-axis). **PPI** is a *protein-protein interaction* dataset ; **Reddit** is a *social-network* (news) dataset

Graph Sampling Training from the GraphSAINT Paper

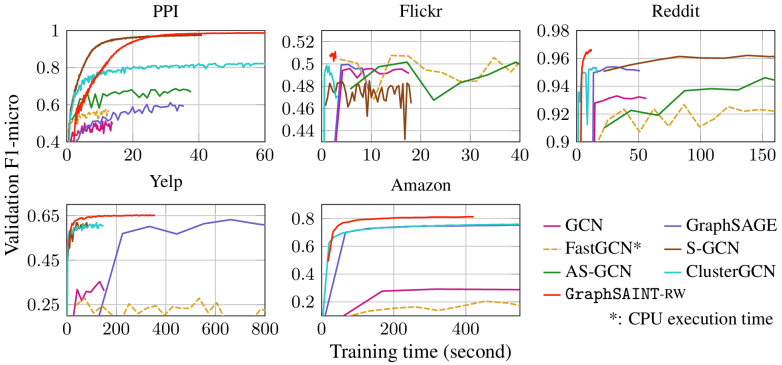
Algorithm 1 `GraphSAINT` training algorithm

Input: Training graph $\mathcal{G}(\mathcal{V}, \mathcal{E}, \boldsymbol{X})$; Labels $\overline{\boldsymbol{Y}}$; Sampler `SAMPLE`;
Output: GCN model with trained weights

1: Pre-processing: Setup `SAMPLE` parameters; Compute normalization coefficients α, λ.
2: **for** each minibatch **do**
3: $\quad \mathcal{G}_s(\mathcal{V}_s, \mathcal{E}_s) \leftarrow$ Sampled sub-graph of $\mathcal{G}$ according to `SAMPLE`
4: $\quad$ GCN construction on $\mathcal{G}_s$.
5: $\quad \{\boldsymbol{y}_v \mid v \in \mathcal{V}_s\} \leftarrow$ Forward propagation of $\{\boldsymbol{x}_v \mid v \in \mathcal{V}_s\}$, normalized by α
6: $\quad$ Backward propagation from λ-normalized loss $L(\boldsymbol{y}_v, \overline{\boldsymbol{y}}_v)$. Update weights.
7: **end for**

[Zeng et al., 2020]

Experiments from the GraphSAINT Paper



[Zeng et al., 2020]

Outline

Scalability Problem with ML-Based Link Prediction

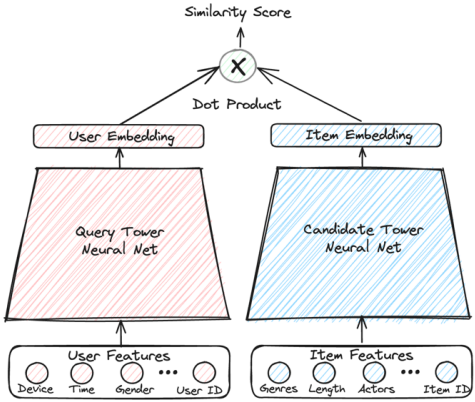
- Consider the task of link prediction:
  - We have sets U and P of nodes (e.g., Users and Products)
  - We wish to classify which $u \in U$ is going to be associated with which $p \in P$
    - For example, user u will purchase product p, user u will watch movie p, user u will follow user p, etc.
- We can treat this task as an ordinary *classification* task, where we train a GNN to map triples (G, u, p) into a probability
  - E.g., $\text{Softmax}(\text{MLP}(\text{GNN}(G, u, v)))$
- At evaluation time (e.g., recommendation), we apply the learned model to $U \times P$
- However, a common scenario is that only few (best) links are expected
- *How many pairs do we need to check?*
  - **AmazonBooks**: 52,643 users $\times$ 91,599 books $=$ 4,822,046,157 pairs
  - **MovieLens 25M** Dataset: 62,000 movies $\times$ 162,000 users $=$ 10,044,000,000 pairs

So? What do we do?

Two common techniques for overcoming the scale:

- Blocking: Use a heuristic to cluster U and P into small *blocks* so that cross-cluster links are unlikely
  - Then, apply the model to every pair in each cluster: $\sum_i(|U_i| \cdot |P_i|)$ instead of $|U| \cdot |P|$
- Two-tower recommendation: uses a model that consists of two phases ("towers")
  1. First, encode every u and p in a shared space $\mathbb{R}^d$ (e.g., using a GNN)
  2. Second, compute a *similarity score* between $\text{Enc}(u)$ and $\text{Enc}(p)$, which is typically the dot product of the two, and use that as the link probability
  - *How does it help?*
    - Now, at evaluation time, we can compute the encoding of each node, and apply efficient vector clustering/search to find nearby $u \in U$ for each $p \in P$
    - E.g., *FAISS* (Facebook AI Similarity Search)

Illustration of a Two-Tower Recommendation



Source: Medium

https://medium.com/tech-p7s1/video-recommendations-at-joyn-two-tower-or-not-to-tower-that-was-never-a-question-6c6f182ade7c

Key Takeaways from Module 5

- MPNNs compute node embeddings by repeatedly aggregating messages from neighbors
- 1-WL provides an intuition on what can be expressed by MPNNs
- Heterogeneous MPNNs extend the homogeneous ones by using label-dependent functions (coefficients) and deploying meta-paths
- Graphs present unique challenges to deep learning: over-smoothing, over-squashing, mini-batching
- Mini-batches are commonly based on graph traversals and graph clustering
- Link-prediction is effectively done by applying a two-tower approach: applying a pairwise similarity layer on top of individual embeddings

The End

Many thanks for helping with the slides:

- **Shunit Agmon** (Technion)
- **Boaz Berger** (Technion)
- **Ilias Fountalis** (RelationalAI)

References I


Alon, U. and Yahav, E. (2021).
On the bottleneck of graph neural networks and its practical implications.
In *ICLR*. OpenReview.net.


Chen, J., Ma, T., and Xiao, C. (2018a).
FastGCN: Fast learning with graph convolutional networks via importance sampling.
In *ICLR (Poster)*. OpenReview.net.


Chen, J., Zhu, J., and Song, L. (2018b).
Stochastic training of graph convolutional networks with variance reduction.
In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pages 941–949. PMLR.


Chiang, W., Liu, X., Si, S., Li, Y., Bengio, S., and Hsieh, C. (2019).
Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks.
In *KDD*, pages 257–266. ACM.

References II


Fey, M., Hu, W., Huang, K., Lenssen, J. E., Ranjan, R., Robinson, J., Ying, R., You, J., and Leskovec, J. (2024).
Position: Relational deep learning - graph representation learning on relational databases.
In *ICML*. OpenReview.net.


Hamilton, W. L. (2020).
Graph Representation Learning.
Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.


Hamilton, W. L., Ying, Z., and Leskovec, J. (2017).
Inductive representation learning on large graphs.
In *NIPS*, pages 1024–1034.


Hornik, K., Stinchcombe, M. B., and White, H. (1989).
Multilayer feedforward networks are universal approximators.
Neural Networks, 2(5):359–366.

References III

Hu, Z., Dong, Y., Wang, K., and Sun, Y. (2020).
Heterogeneous graph transformer.
In *Proceedings of the web conference 2020*, pages 2704–2710.

Huang, W., Zhang, T., Rong, Y., and Huang, J. (2018).
Adaptive sampling towards fast graph representation learning.
In *NeurIPS*, pages 4563–4572.

Keriven, N. (2022).
Not too little, not too much: a theoretical analysis of graph (over)smoothing.
In *NeurIPS*.

Kipf, T. N. and Welling, M. (2017).
Semi-supervised classification with graph convolutional networks.
In *ICLR (Poster)*. OpenReview.net.

References IV

Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., and Grohe, M. (2019).

Weisfeiler and leman go neural: Higher-order graph neural networks.

In *AAAI*, pages 4602–4609. AAAI Press.

Robinson, J., Ranjan, R., Hu, W., Huang, K., Han, J., Dobles, A., Fey, M., Lenssen, J. E., Yuan, Y., Zhang, Z., He, X., and Leskovec, J. (2024).

Relbench: A benchmark for deep learning on relational databases.

In *NeurIPS*.

Rossi, E., Charpentier, B., Giovanni, F. D., Frasca, F., Günnemann, S., and Bronstein, M. M. (2023).

Edge directionality improves learning on heterophilic graphs.

In *LoG*, volume 231 of *Proceedings of Machine Learning Research*, page 25. PMLR.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. (2018a).

Modeling relational data with graph convolutional networks.

In *European semantic web conference*, pages 593–607. Springer.

References V


Schlichtkrull, M. S., Kipf, T. N., Bloem, P., van den Berg, R., Titov, I., and Welling, M. (2018b).
Modeling relational data with graph convolutional networks.
In *ESWC*, Lecture Notes in Computer Science, pages 593–607. Springer.


Velickovic, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. (2018).
Graph attention networks.
In *ICLR (Poster)*. OpenReview.net.


Wang, M., Gan, Q., Wipf, D., Zhang, Z., Faloutsos, C., Zhang, W., Zhang, M., Cai, Z., Li, J., Mao, Z., Song, Y., Tang, J., Zhang, Y., Yang, G., Lei, C., Qin, X., Li, N., Zhang, H., Wang, Y., and Zhang, Z. (2024).
4dbinfer: A 4d benchmarking toolbox for graph-centric predictive modeling on rdbs.
In *NeurIPS*.


Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., and Yu, P. S. (2019).
Heterogeneous graph attention network.
In *The world wide web conference*, pages 2022–2032.

References VI


Wu, L., Cui, P., Pei, J., and Zhao, L., editors (2022).
Graph Neural Networks: Foundations, Frontiers, and Applications.
Springer Nature Singapore, Singapore.


Xu, K., Hu, W., Leskovec, J., and Jegelka, S. (2019).
How powerful are graph neural networks?
In *International Conference on Learning Representations (ICLR)*.


Zeng, H., Zhou, H., Srivastava, A., Kannan, R., and Prasanna, V. K. (2020).
Graphsaint: Graph sampling based inductive learning method.
In *ICLR*. OpenReview.net.