

UNIVERSITÄT
LEIPZIG

Introduction to Graph Neural Networks

Seminar Current Topics in Graph Machine Learning

Dr. Martin Ritzert

October 13, 2026

martin.ritzert@uni-leipzig.de

Today

1. Presentation of the seminar topics
2. Choosing a topic
3. Introduction to graph neural networks
4. Good presentations (separate slide set)

Presentation Schedule

Tuesday	Topics
November 17, 2026	1–2
November 24	3–4
December 1	5
December 8	6–7
December 15	8–9
December 22	Winter break
December 29	Winter break
January 5, 2027	10–11
January 12	12–13
January 19	14–15
January 26	16–17

Evaluation

The evaluation will take place on December 1, 2026.

Seminar paper

7 pages, formatted using the provided LaTeX styleguide file. Due two weeks after your presentation.

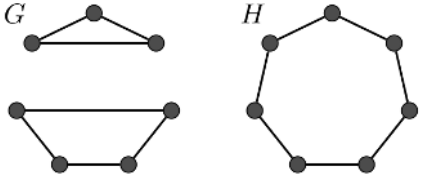
Final session

Tuesday, January 26, 2027.

Seminar Topics

Expressivity

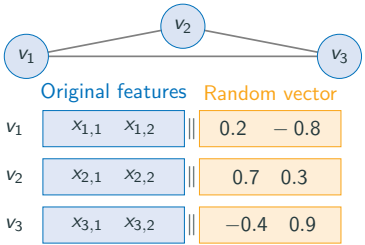
- Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks (AAAI 2019)
- How Powerful are Graph Neural Networks? (ICLR 2019)



Abboud et al., Fig. 1

Random Node Initialization

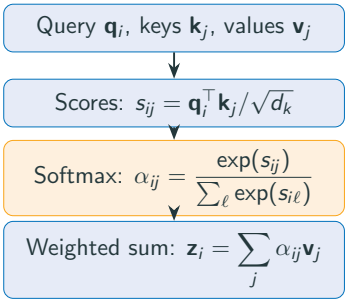
- The Surprising Power of Graph Neural Networks with Random Node Initialization (ICLR 2021)



Independent random vectors appended to node features

Attention

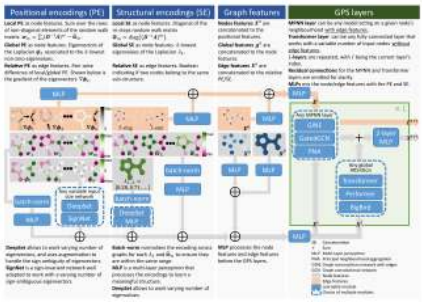
- Attention Is All You Need (NIPS 2017)
- Graph Attention Networks (ICLR 2018)
- How Attentive are Graph Attention Networks? (ICLR 2022)



Scaled dot-product softmax attention

Graph Transformers

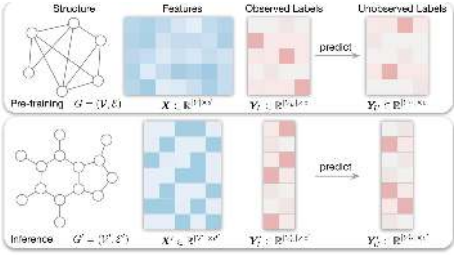
- Recipe for a General, Powerful, Scalable Graph Transformer (NeurIPS 2022)
- Do Transformers Really Perform Bad for Graph Representation? (NeurIPS 2021)



Rampášek et al., Fig. 1

Graph Foundation Models

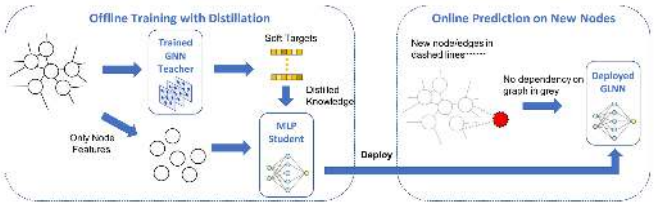
- Fully-inductive Node Classification on Arbitrary Graphs (ICLR 2025)



Zhao et al., Fig. 2

The full power of GNNs is not always needed

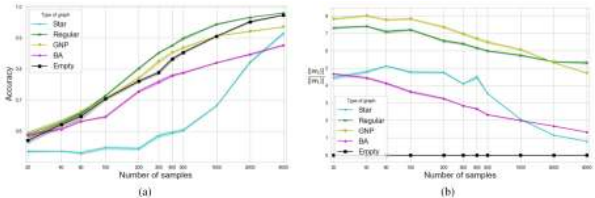
- Simplifying Graph Convolutional Networks (ICML 2019)
- Graph-less Neural Networks: Teaching Old MLPs New Tricks via Distillation (ICLR 2022)



Zhang et al., Fig. 2

Importance of Graph Structure

- Graph Neural Networks Use Graphs When They Shouldn't (ICML 2024)
- Revisiting the Necessity of Graph Learning and Common Graph Benchmarks (NeurIPS 2024)



Bechler-Speicher et al., Fig. 1

Benchmarking

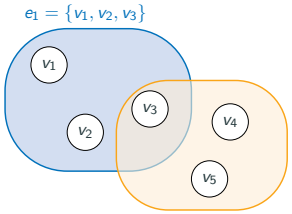
- Open Graph Benchmark: Datasets for Machine Learning on Graphs (NeurIPS 2020)
- Position: Graph Learning Will Lose Relevance Due To Poor Benchmarks (ICML 2025)
- GraphBench: Next-generation graph learning benchmarking



Hu et al., Fig. 1

Hypergraph Neural Networks

- Hypergraph Neural Networks (AAAI 2019)
- DHG-Bench: A Comprehensive Benchmark for Deep Hypergraph Learning (ICLR 2026; focus on Table 1)
- Improved Expressivity of Hypergraph Neural Networks through High-Dimensional Generalized Weisfeiler-Leman Algorithms (ICML 2025)

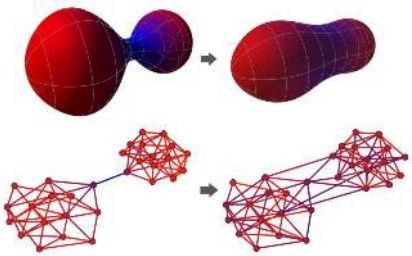


One hyperedge can connect more than two vertices.

Illustration: overlapping hyperedges

Oversquashing/Oversmoothing/Graph Rewiring

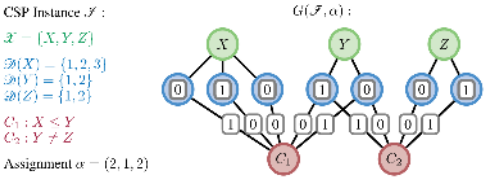
- Understanding over-squashing and bottlenecks on graphs via curvature (ICLR 2022)



Topping et al., Fig. 1

Constraint Satisfaction Problems

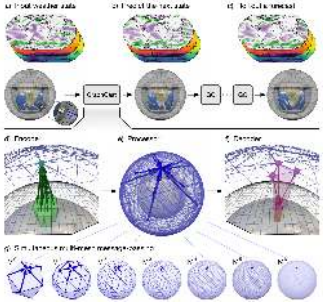
- Graph Neural Networks for Maximum Constraint Satisfaction (FRAI 2021)
- One Model, Any CSP: Graph Neural Networks as Fast Global Search Heuristics for Constraint Satisfaction (IJCAI 2023)



Tönshoff et al., Fig. 1

Weather Forecasting

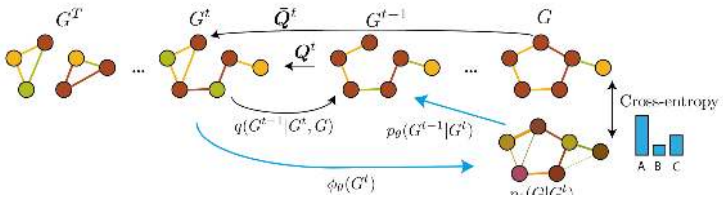
- GraphCast: Learning skillful medium-range global weather forecasting (Science 2023)
- Probabilistic Weather Forecasting with Hierarchical Graph Neural Networks (NeurIPS 2025)



Lam et al., Fig. 1

Graph Generation

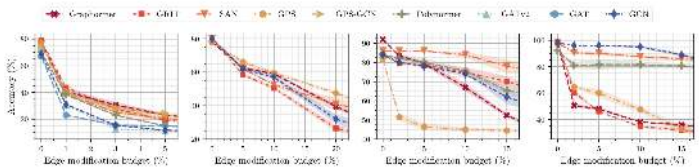
- DiGress: Discrete Denoising diffusion for graph generation (ICLR 2023)



Vignac et al., Fig. 1

Adversarial Attacks

- Adversarial Robustness of Graph Transformers (TMLR 2025)



(a) CLUSTER. (b) Reddit Threads. (c) UPFD politifact. (d) UPFD gossipcop.

Foth et al., Fig. 1

Temporal Graphs

- Towards Better Dynamic Graph Learning: New Architecture and Unified Library (NeurIPS 2023)

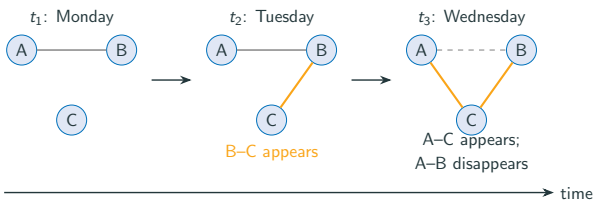


Illustration: graph structure changes over time

Relational Databases

- RelBench: A Benchmark for Deep Learning on Relational Databases (NeurIPS 2024)
- RelBench v2: A Large-Scale Benchmark and Repository for Relational Data (ICLR 2026)

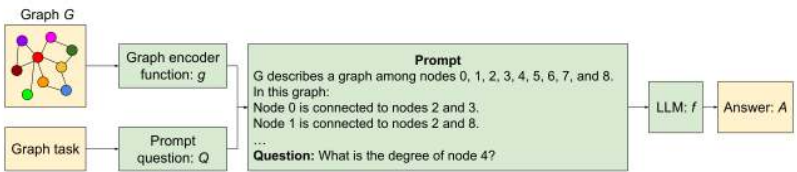


Rows become nodes; foreign keys define graph edges.

Illustrative relational database schema

LLMs on Graphs

- Talk like a Graph: Encoding Graphs for Large Language Models (ICLR 2024)
- Actions Speak Louder than Prompts: A Large-Scale Study of LLMs for Graph Inference (ICLR 2026)



Fatemi et al., Fig. 1

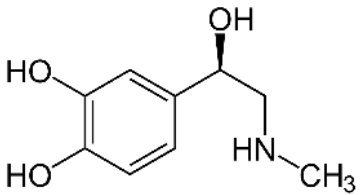
Choosing a Topic

- Choose from the 17 topics just presented.
- Submit your topic preferences in order of preference.
- We have 15 presentation slots; two topics will remain unassigned.

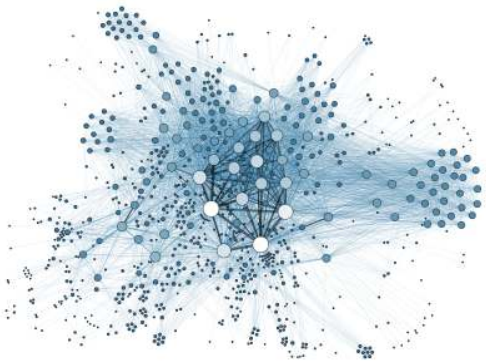
Why Learn on Graphs?

Introduction

Graphs are all around us:



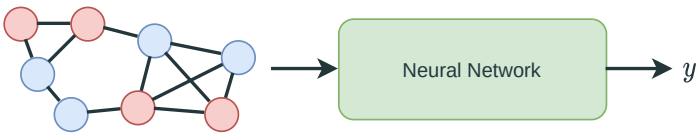
Molecules

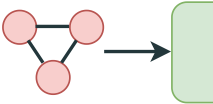


Social Networks

Graph Learning

Main task of Graph Learning: Learn functions on graphs from examples





Fundamental Challenges of Graph Learning:

- Process graphs of any size and structure
- Permutation Invariance

Challenges of Graph Learning

Graphs of any Size and Structure

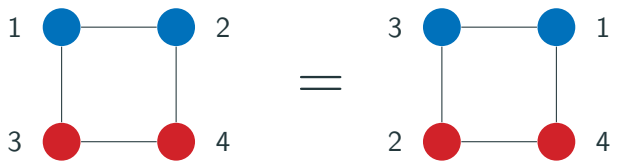
- Molecules ($\leq$ 20 nodes) and social networks ($\geq$ 1 million nodes)
- Extremely varying number of neighbors (e.g. celebrities vs normal people)
- Both dense and sparse graphs (although sparsity is more common)

Permutation Invariance

- Reordering/shuffling the nodes does not change the graph structure
- Adjacency matrix may look completely different
- Graph-level predictions should be invariant to node ordering
- Node-level predictions should reorder with the nodes (equivariance)

Permutation Invariance

The following two graphs are the same:



The numbering (ordering in the adjacency matrix) does not matter.

From Neural Networks to Message Passing

Multi-Layer Perceptrons

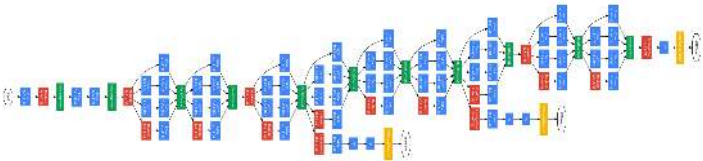
- Oldest neural networks
- consists of dense or linear layers
- Perceptron: a simple linear classifier

Let $W \in \mathbb{R}^{d \times e}$ be a weight matrix. A single dense layer $f_W : \mathbb{R}^d \to \mathbb{R}^e$ is given by

$$f_W(x) = \sigma\left(W^T x\right)$$

where σ is a nonlinearity, e.g. ReLU.

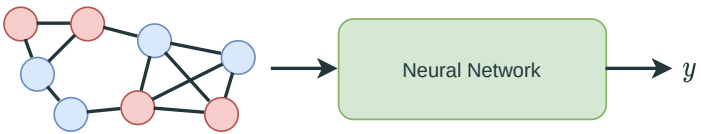
MLP combines multiple layers: $f = f_{W_k} \circ f_{W_{k-1}} \circ \cdots \circ f_{W_1}$



Basic Idea

- Perform a complex operation by combining simple matrix operations
- Train with Stochastic Gradient Descent on large amounts of data.
  - All operations must be differentiable to compute gradients
- Usually end-to-end learning with little feature engineering.

Can we design a neural network that works on graphs instead of vectors?



Fundamental Challenges

1. Process graphs of any size and structure
   - Not just vectors or grids: Highly irregular structure
   - Structure is often important
2. Permutation Invariance

Solution: Message Passing

Message Passing

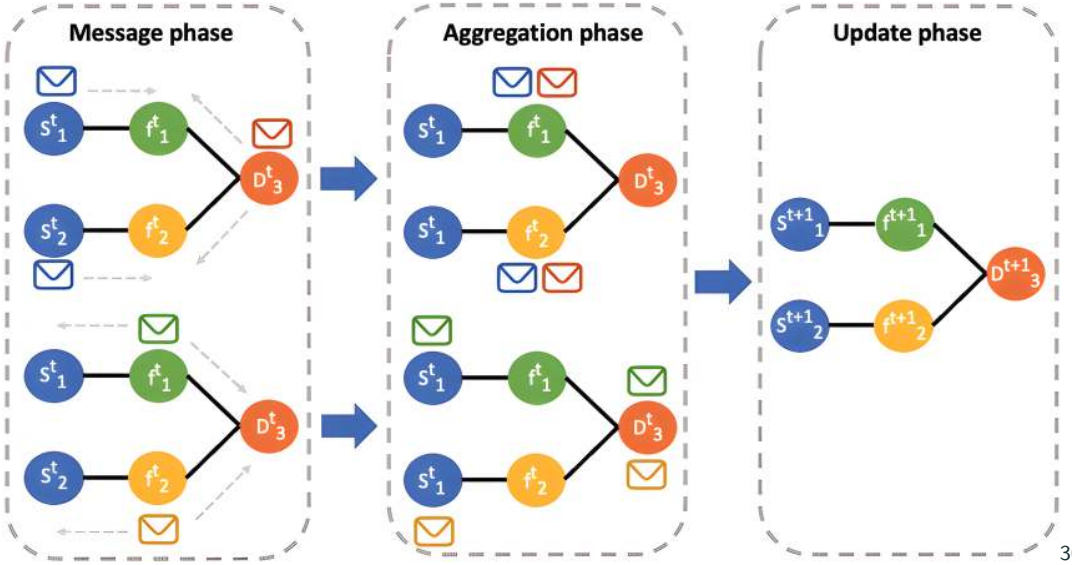
Input

- Graph $G = (V, E)$
- Node Features $X_V : V \to \mathbb{R}^d$
- Edge Features $X_E : E \to \mathbb{R}^{d'}$ (optional)

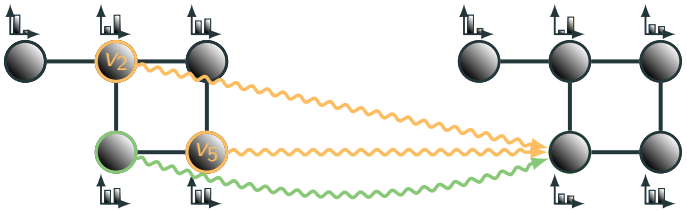
Idea of Message Passing

- Node features as initial embedding $h^0(v) = X_V(v)$
- Update latent node embedding $h^\ell(v) \in \mathbb{R}^{d^\ell}$ in each layer ℓ
- The update depends on the latent embedding of the neighbors
- All nodes share the same aggregation function (including weights)

Illustration of Message Passing



Message-passing graph neural networks



$$f^{(t)}(v) = \sigma(W_1 f^{(t-1)}(v) + W_2 \sum_{w \in N(v)} f^{(t-1)}(w))$$

$$f^{(t)}(v) = f_{\text{merge}}^{W_1}\left(f^{(t-1)}(v), f_{\text{aggr}}^{W_2}\left(\{\!\{f^{(t-1)}(w) \mid w \in N(v)\}\!\}\right)\right)$$

Elements of a GNN Layer

- Trainable functions $\mathbf{M}^\ell$, $\mathbf{U}^\ell$ (message generation and update)
- Aggregation function $\bigoplus$ (Sum, Mean, Max,...)

Function computed by the layer

$$h^\ell(v) = \mathbf{U}^\ell \left(h^{\ell-1}(v), \bigoplus_{u \in \mathcal{N}(v)} \mathbf{M}^\ell \left(h^{\ell-1}(u), X_E(vu) \right) \right)$$

Both $\mathbf{U}^\ell$ and $\mathbf{M}^\ell$ can range from simple linear layers to highly complex functions

How does it solve the challenges?

- ✓ Message passing is independent of the graph's size or structure
- ✓ Message passing (somehow) evaluates/depends on the graph's structure
- ✓ Shared functions and order-independent aggregation make node updates permutation equivariant
- ✓ Invariant pooling gives permutation-invariant graph predictions

Downstream Tasks

Node-Level Tasks

Apply predictive MLP to final vertex embedding:

$$y(v) = \text{MLP}(h^L(v))$$

Graph-Level Tasks

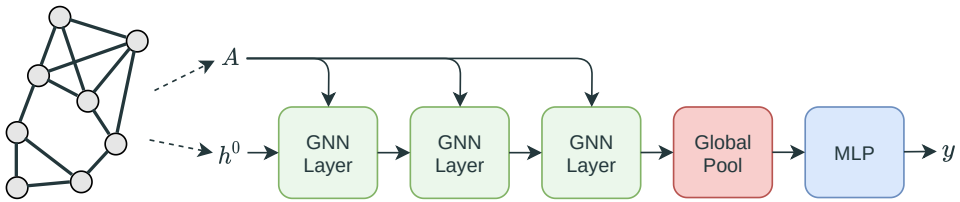
Pool vertex embedding into a graph embedding with readout function (Sum, Mean, ...):

$$h(G) = \sum_{v \in V} h^L(v)$$

Apply predictive MLP to pooled vector:

$$y(G) = \text{MLP}(h(G))$$

Graph Neural Networks



Typical GNN architecture consists of the following:

1. (Optional) An MLP processing the initial features
2. A number of message-passing layers (2-5 works well in many cases)
3. A pooling block (just for graph-level predictions)
4. An MLP to come up with final predictions ("head")

A Concrete Example: GCN

Graph Convolutional Neural Network (GCN)

- One of the first GNNs that actually worked.
- Simple and effective.
- Still often used as the default GNN for many tasks.

Limitations: Does not incorporate edge features, no distinction between self and neighbors during aggregation

The GCN Layer

Components

- Trainable weight matrix $W^{(\ell)} \in \mathbb{R}^{d^{(\ell-1)} \times d^{(\ell)}}$
- Activation Function σ (usually ReLU)

Update rule

$$h^{\ell}(v) = \sigma\Big(\sum_{u \in N(v) \cup \{v\}} \frac{1}{\sqrt{d_v d_u}} \cdot h^{\ell-1}(u) \cdot W^{(\ell)} \Big)$$

with $d_i = \deg(v_i) + 1$.

Backup: Implementing Message Passing

Implementing Message Passing

How do you actually implement Message Passing in PyTorch?

Simple Solution: Dense Matrix Multiplication

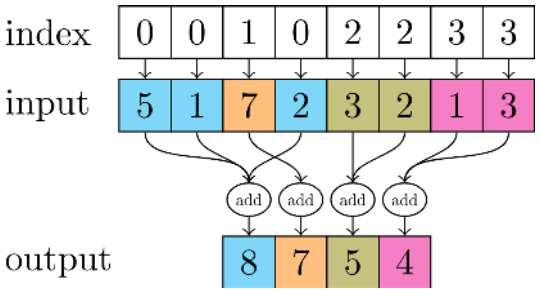
- ✓ Easy to implement
- ✓ Highly optimized for small graphs
- ✓ Simple batch creation
- ✗ Does not scale to large graphs
- ✗ Requires padding
- ✗ Only linear aggregations
- ✗ No edge features

Better Solution: **Scatter Operations**

PyTorch Scatter

Open Source Torch Extension[1], Compatible with CUDA and Backpropagation.

Enables aggregation of input with index list:



[1] https://github.com/rusty1s/pytorch_scatter

PyTorch Scatter Example

$$x = \begin{pmatrix} 0 & 2 \\ 1 & 0 \\ 1 & 1 \\ 3 & 5 \\ 0 & 1 \end{pmatrix}$$

$$\text{idx} = [0, 1, 0, 2, 2]$$

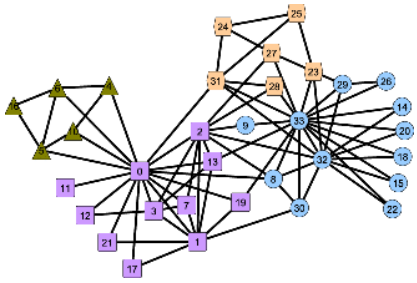
$$\texttt{scatter_sum}(x, idx, \texttt{dim=0}) = \begin{pmatrix} 1 & 3 \\ 1 & 0 \\ 3 & 6 \end{pmatrix}$$

$$\texttt{scatter_max}(x, idx, \texttt{dim=0})[0] = \begin{pmatrix} 1 & 2 \\ 1 & 0 \\ 3 & 5 \end{pmatrix}$$

Sparse Graph Representation

Sparse representation of $G = (V, E)$:

1. Directed edge list $\text{idx}_E \in \{0, \ldots, |V| - 1\}^{2 \times 2|E|}$
2. Node feature matrix $X_V \in \mathbb{R}^{|V| \times d}$
3. Optional edge feature matrix $X_E \in \mathbb{R}^{2|E| \times d'}$
4. Graph label $y \in \mathbb{R}^c$



Graphs in PyG

Creating a graph in PyG:

```
import torch
from torch_geometric.data import Data

edge_index = torch.tensor([[0, 1, 2], [1, 0, 2]], dtype=torch.long)
x = torch.tensor([[-1], [0], [1]], dtype=torch.float)

data = Data(x=x, edge_index=edge_index)
# Output: Data(x=[3, 1], edge_index=[2, 3])
```

Message Passing with Scatter

Given: $H^{(\ell-1)} \in \mathbb{R}^{|V| \times d_h}, \quad X_E \in \mathbb{R}^{2|E| \times d'}, \quad \text{idx}_E \in \{0, \dots, |V|-1\}^{2 \times 2|E|}$

Compute:

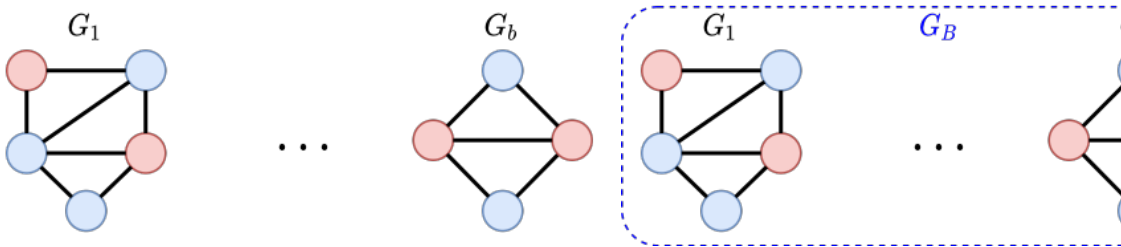
Messages: $Y^{(\ell)} = \mathbf{M}\big(H^{(\ell-1)}\big[\text{idx}_E[0]\big], X_E\big)$

Aggregation: $Z^{(\ell)} = \texttt{scatter_sum}(Y^{(\ell)}, \text{idx}_E[1], \text{dim=0})$

Update: $H^{(\ell)} = \mathbf{U}\big(H^{(\ell-1)}, Z^{(\ell)}\big)$

Batching Sparse Graphs

Given: Batch of training graphs $(\text{idx}_{E_1}, X_{V_1}, X_{E_1}, y_1), \ldots, (\text{idx}_{E_b}, X_{V_b}, X_{E_b}, y_b)$



How do we combine these into one input for parallel processing?

Compute disjoint union $G_B = (V_B, E_B)$: $V_B = \bigcup_{i=1}^{b} V_i$, $E_B = \bigcup_{i=1}^{b} E_i$

Batching Sparse Graphs

Representation of the batched graph G_B:

1. $\text{idx}_E \in \{0, \dots, |V_B| - 1\}^{2 \times 2|E_B|}$
2. $X_{V_B} \in \mathbb{R}^{|V_B| \times d}$
3. $X_{E_B} \in \mathbb{R}^{|E_B| \times d'}$
4. $y_B \in \mathbb{R}^{B \times c}$
5. $\text{batch_idx} \in \{0, \dots, b - 1\}^{|V_B|}$ (remember original graph for each node)

The indices in idx_{E_B} are re-enumerated.

Pooling becomes a scatter operation with batch_idx.

Example: A Simple GNN Model

```python
import torch
import torch.nn.functional as F
import torch_geometric as pyg

class GCN(torch.nn.Module):
  def __init__(self):
    super(GCN, self).__init__()
    self.conv1 = pyg.nn.GCNConv(dataset.num_node_features, 16)
    self.conv2 = pyg.nn.GCNConv(16, dataset.num_classes)

  def forward(self, data):
    x, edge_index = data.x, data.edge_index
    x = self.conv1(x, edge_index)
    x = F.relu(x)
    x = F.dropout(x, training=self.training)
    x = self.conv2(x, edge_index)
    return F.log_softmax(x, dim=1)
```

Training Loop for the GNN Model

```
optimizer = torch.optim.AdamW(model.parameters(), lr=0.01)

model.train()
for epoch in range(200):
  optimizer.zero_grad()
  out = model(data)
  loss = F.nll_loss(out[data.train_mask], data.y[data.train_mask])
  loss.backward()
  optimizer.step()

  if epoch % 10 == 0:
    print(f'Epoch {epoch}: Loss {loss.item()}')
```

Backup: Architecture Choices

GNN Components

Update function U

- Usually MLP with one or two layers
- If **U** has two layers, the second layer is usually linear

Message function M

- If no edge features are given, simply use identity (like the GCN)
- If edge features are given, **M** should be non-linear.

Special GNNs

Recurrent GNNs

- Choose **U** as GRU-Cell and reuse it in each message pass
- Number of message passes not fixed
- Application: Learning Heuristics for NP-hard problems

Heterogeneous GNNs

- Heterogeneous graphs contain nodes of different types
- Train different functions **U**, **M** for each type of node

Hypergraphs

- Turn hyperedges into nodes
- Perform message-passing only between nodes and hyperedges (alternatingly)

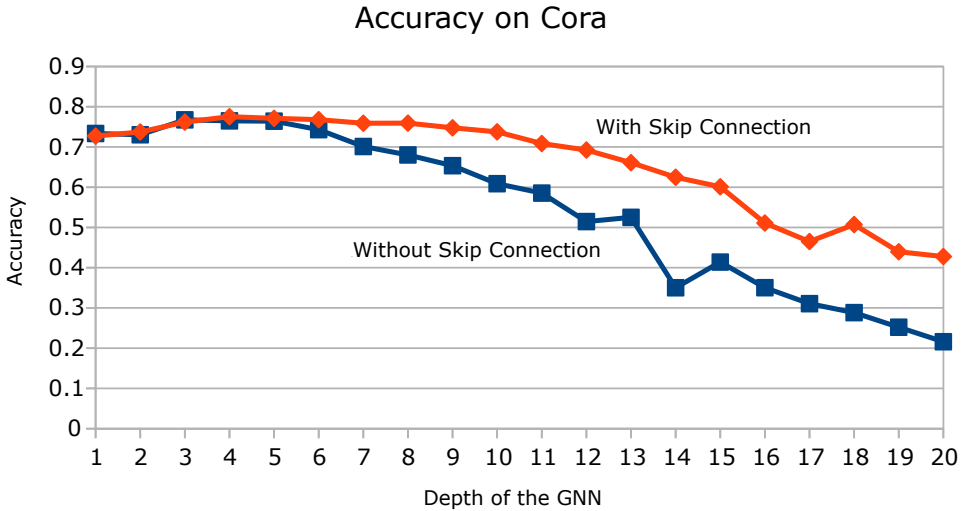
Standard Tricks

Most of the standard tricks from Deep Learning also work for GNNs:

- Batch/Layer Normalization
- Dropout
- Use Residual Connections

$$h^{\ell}(v) = h^{\ell-1}(v) + \mathbf{U}^{\ell}\left(h^{\ell-1}(v), \bigoplus_{u \in \mathcal{N}(v)} \mathbf{M}^{\ell}\left(h^{\ell-1}(u), X_E(vu)\right)\right)$$

Skip-Connections



Virtual Nodes

Perform global pooling after each GNN layer and pass result back to nodes:

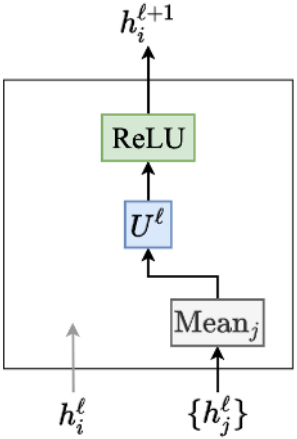
$$h^\ell(G) = \mathbf{V}^\ell\Big(\sum_{v \in V} h^\ell(v)\Big)$$

$$\tilde{h}^\ell(v) = h^\ell(v) + h^\ell(G)$$

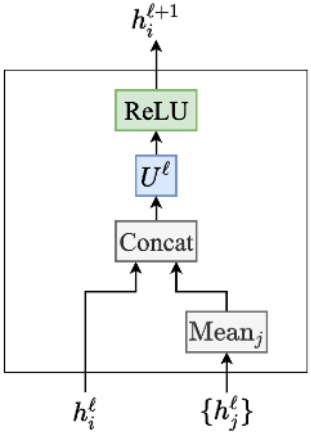
Here, $\mathbf{V}^\ell$ is a trainable MLP.

Virtual Nodes enable global information exchange after each layer.

GCN and GraphSAGE (Kipf and Welling, 2017 and Hamilton et al., 2017)



GCN



GraphSAGE