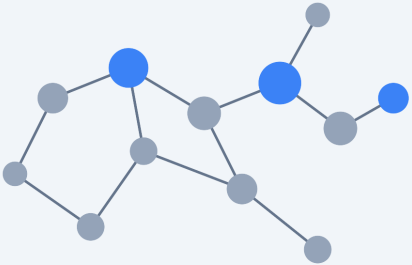
UNIVERSITY OF LUXEMBOURG

GraphSAGE

Inductive Representation Learning on Large Graphs

Thomas Gantz
Alberto Finardi
Tommaso Crippa
Jan Marxen

December 8, 2025



UNIVERSITY OF LUXEMBOURG

Introduction

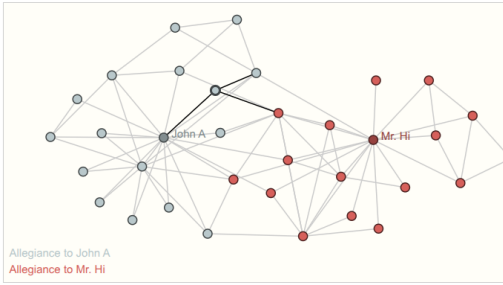
UNIVERSITY OF LUXEMBOURG

Graph Prediction Tasks

Introduction

Primary focus: Node-level prediction — predict properties of individual nodes (classification, regression).

- **Node-level (focus):** Predict node attributes or labels using features and neighbor information.
- **Edge-level:** Predict relationships between node pairs (link prediction, edge classification).
- **Graph-level:** Predict properties of whole graphs (e.g., molecule properties).

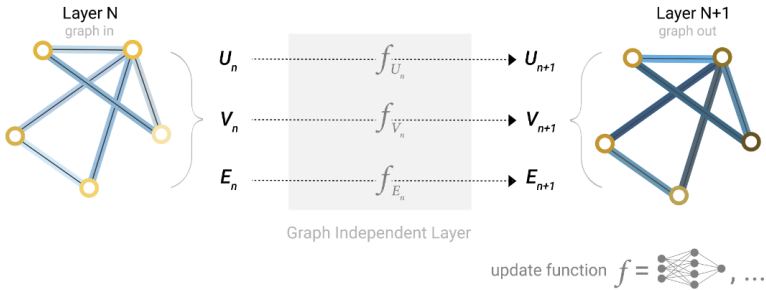


What is a Graph Neural Network?

Introduction

A learnable transformation on graph attributes that:

- Updates node/edge/graph features using **neural networks**
- **Respects graph structure** by aggregating information from neighbors
- Is **permutation invariant** (order of nodes doesn't matter)

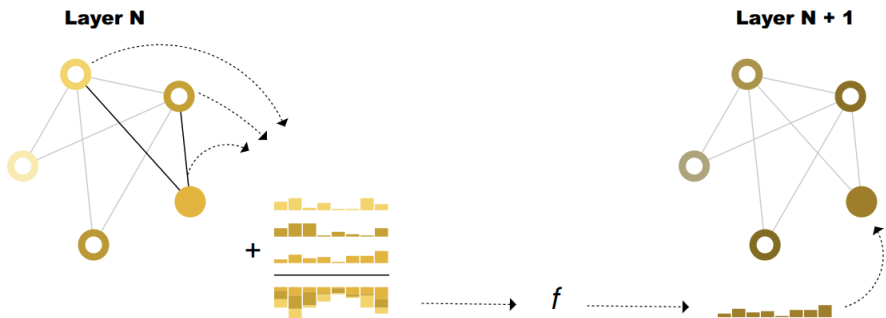


Message Passing: The Core Idea

Introduction

Three steps repeated at each layer:

1. **Gather:** Collect embeddings from neighboring nodes
2. **Aggregate:** Combine neighbors' info (sum / mean / max)
3. **Update:** Apply learned transform using the aggregated vector



Message Passing: Notation

Introduction

Message Passing Equation

$$h_v^{(k)} = \sigma\left(W \cdot \left[h_v^{(k-1)}, \text{AGG}\left(\{h_u : u \in N(v)\}\right)\right]\right)$$

- $h_v^{(k)}$ — Node v repr. at layer k
- h_u — Neighbor node u repr.
- $N(v)$ — Neighbors of v
- k — Layer index (hops)
- $\text{AGG}(\cdot)$ — Aggregator (mean, sum, max)
- W — Learnable weight matrix
- σ — Activation (ReLU, tanh)

Before GraphSAGE: The Problem

Introduction

Let's audit the limitations of existing approaches...

- **DeepWalk / node2vec:** Transductive: embed every node; new nodes need full retraining.
  - ▷ Not GNNs — random-walk based embeddings, no message passing
- **GCNs:** Often require access to the full graph during training/inference; can be costly to scale.
  - ▷ GNNs — but transductive: fixed node set at training
- **Result:** No compact parametric function to generate embeddings for unseen nodes.

This motivates GraphSAGE's key insight...

The Key Insight

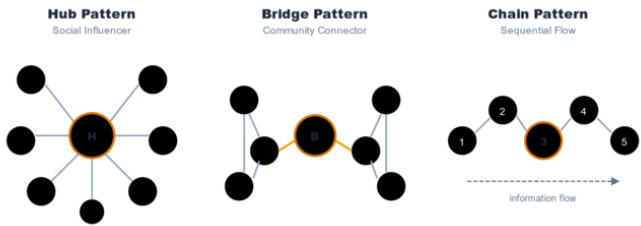
The Key Insight

The Key Insight

Don't learn embeddings for each node...

Learn a FUNCTION that generates embeddings

By sampling & aggregating neighborhood features



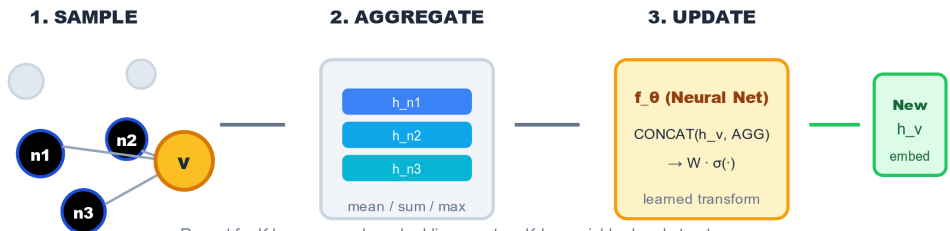
GraphSAGE Framework

GraphSAGE: Inductive Framework

GraphSAGE Framework

Core Principle: Sample + Aggregate

- Learn **aggregator functions** (not node embeddings)
- For any node v: **sample neighbors, aggregate their features**
- Pass through learned neural networks
- **Inference:** Apply same function to unseen nodes



Implementation

UNIVERSITY OF LUXEMBOURG

System Architecture Overview

Implementation

Dataset: ogbn-products (Amazon co-purchase network)

- 2.4M nodes (products), 61M edges, 47 classes
- 8% train / 2% val / 90% test split

Training Environment:

- **HPC Cluster:** MeluXina (Luxembourg National Supercomputer)
- **GPUs:** Up to 4x NVIDIA A100 (40GB) per node
- **Framework:** PyTorch 2.1.2 + PyTorch Geometric
- **Containerization:** Apptainer/Singularity for reproducibility

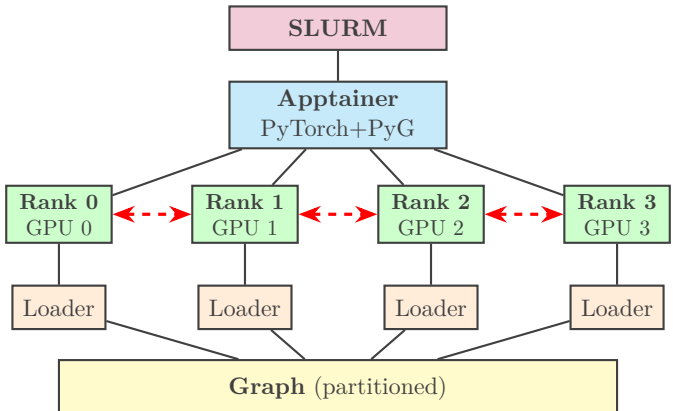
Model Architecture:

- 5-layer GraphSAGE (SAGEConv + LayerNorm + ReLU)
- Hidden dimension: 256, Dropout: 0.5

UNIVERSITY OF LUXEMBOURG

System Architecture Diagram

Implementation



Multi-GPU DDP: data partitioned across 4 GPUs, gradients synced via NCCL

UNIVERSITY OF LUXEMBOURG

Execution Model: Distributed Data Parallel

Implementation

Process Initialization:

- SLURM launches 4 processes (1 per GPU) on single node
- Each process: independent Python interpreter + CUDA context
- NCCL backend for GPU-to-GPU communication (InfiniBand)

Data Distribution:

- Training set (250k nodes) partitioned: 62.5k per rank
- No overlap between ranks → each processes unique subset
- Each rank has independent NeighborLoader for k-hop sampling

Gradient Synchronization:

- DDP automatically wraps model: synchronizes gradients after backward()
- All-reduce operation: averages gradients across all GPUs
- Learning rate scaled linearly: $\text{lr}_{\text{eff}} = \text{lr}_{\text{base}} \times N_{\text{GPUs}}$

UNIVERSITY OF LUXEMBOURG

Execution Model: Training Workflow

Implementation

1. **Data Loading (Parallel)**:
   - Each rank: 4-8 worker processes prefetch batches
   - NeighborLoader samples k-hop subgraphs
   - Pinned memory → GPU transfer (non-blocking)
2. **Forward + Backward (Parallel)**:
   - Each GPU: processes batch independently
   - 5-layer message passing: aggregate → transform → activate
   - Compute loss (cross-entropy), backpropagate gradients
3. **Gradient Synchronization**:
   - DDP all-reduce: average gradients across 4 GPUs (NCCL)
   - Optimizer step with synchronized gradients
4. **Evaluation (Rank 0 Only)**:
   - Every 5 epochs: validation accuracy on full validation set
   - Checkpoint best model, early stopping (patience = 10)

UNIVERSITY OF LUXEMBOURG

Code Structure & Containerization

Implementation

Main Python Scripts:

- `train_graphsage.py` — Single-GPU training baseline
- `train_graphsage_ddp.py` — Multi-GPU DDP training (enhanced)
- `plot_batch.py` — Batch size benchmarking analysis
- `plot_neighbor.py` — Neighbor sampling strategy analysis

Apptainer Container:

- **Base:** `pytorch/pytorch:2.1.2-cuda12.1-cudnn8-runtime`
- **Dependencies:** PyTorch Geometric, pyg-lib, OGB, torch-scatter/sparse
- **Why containerize?**
  - — Complex dependency graph (CUDA-compiled extensions)
  - — Reproducibility across HPC environments
  - — Avoid version conflicts on shared cluster

Why HPC and Parallelism?

Implementation

1. **Scale of the Problem:**
   - Dataset cannot fit in single GPU
   - Neighbor sampling: Each batch node samples 5-hop neighborhoods
   - Fanout [15,10,10,10,10] → exponential growth: ∼150k neighbors per seed node
   - Batch 128 nodes × 150k expansion = 19.2M nodes per batch

2. **Memory Bottleneck:**
   - Model parameters: only ∼8 MB
   - Sampled subgraphs + intermediate activations: **30-40 GB** per batch
   - Single GPU (40 GB) cannot handle large batches → slow training

3. **Solution: Multi-GPU Parallelism**
   - Split data across 4 GPUs → 4x memory capacity
   - Effective batch size: 2048 nodes (faster convergence)
   - Training time: **hours instead of days**

UNIVERSITY OF LUXEMBOURG

Hyperparameters

Hyperparameters

Hyperparameters

Hyperparameter tuning details coming soon...

UNIVERSITY OF LUXEMBOURG

Results

Experimental Setup

Results

Hardware: MeluXina Supercomputer (Luxembourg National Supercomputer)

- 4× NVIDIA A100 (40GB) per node
- HDR200 InfiniBand (400 Gb/s aggregate bandwidth)

Benchmarking Focus:

1. **Mini-batch sampling:** Batch size impact on time & memory
2. **Neighbor sampling:** Fanout strategies (accuracy vs. efficiency)
3. **GPU scaling:** 1/2/4/8 GPU distributed training

Section 6.1

Mini-batch Sampling

UNIVERSITY OF LUXEMBOURG

Mini-batch Benchmarking: Configurations

Results

What's being tested?

How does **batch size** and **gradient accumulation** affect:

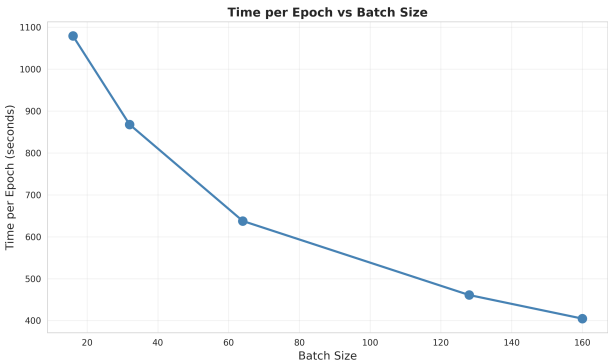
- Training time per epoch
- Peak GPU memory usage
- Overall training efficiency

Batch Size	Accum. Steps
16	5
32	5
64	5
64	1, 2, 3, 4
128	**5**
128	1, 2, 3, 4
160	5
192	5
256	5
512	5

UNIVERSITY OF LUXEMBOURG

Batch Size: Training Time

Results

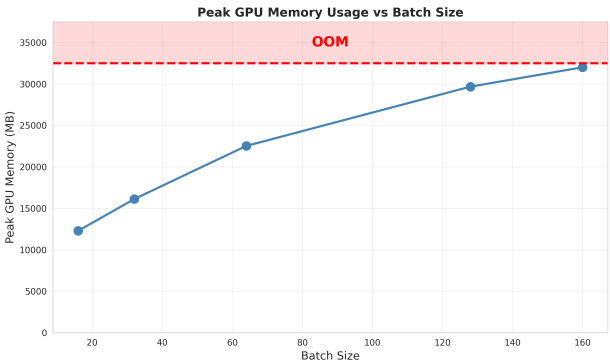


Key Findings:

- Inverse relationship: larger batch = faster training
- Better GPU utilization with larger batches
- Reduced forward pass / backward pass overhead

Batch Size: Memory Consumption

Results



Key Findings:

- **Max batch size:** 160
- Uses ∼32GB (7.5GB system overhead)
- Bigger batches go OOM

Section 6.2

Neighbor Sampling

UNIVERSITY OF LUXEMBOURG

Neighbor Sampling: Configurations

Results

What's being tested?

How do different **fanout strategies** affect:

- Test accuracy
- Training time per epoch
- GPU memory usage

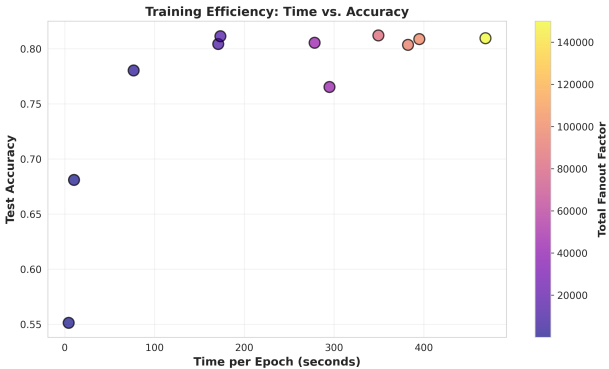
Fanout = neighbors sampled per layer
Total Fanout = product across all layers

$$F_{\text{total}} = \prod_{i=1}^{5} L_i$$

L1	L2	L3	L4	L5
50	0	0	0	0
15	0	0	0	0
15	10	0	0	0
15	10	10	0	0
15	10	10	10	0
15	**10**	**10**	**10**	**10**
10	10	10	10	10
12	12	12	12	12
15	15	15	15	15
20	15	10	5	3
15	15	10	5	5
15	15	15	5	5
12	11	10	9	8
5	5	5	5	5
3	5	10	15	20
30	10	5	3	3

Neighbor Sampling: Accuracy vs. Time

Results

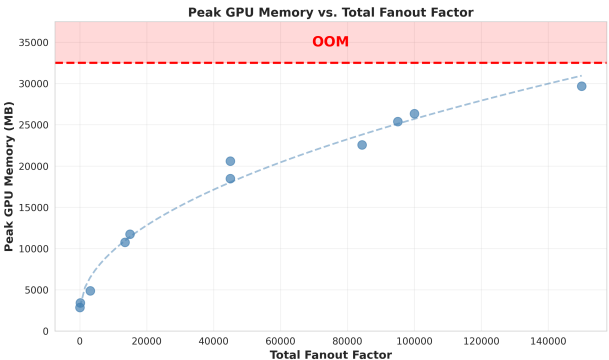


Key Findings:

- Upper-left = optimal
- Moderate fanouts best balance
- Aggressive sampling hurts accuracy or stability
- Very high fanouts: no proportional gains

Neighbor Sampling: Memory vs. Fanout

Results



Key Findings:

- Rapid memory growth with fanout
- Must balance receptive field vs. memory
- High fanouts hit memory limits and go OOM

UNIVERSITY OF LUXEMBOURG

Top 5 Neighbor Configurations

Results

Rank	Neighbor Sampling	Test Acc	Time (h)
1	[15, 15, 15, 5, 5]	0.8121	14.56
2	[15, 10, 10, 10, 0]	0.8114	**7.22**
3	[15, 10, 10, 10, 10]	0.8096	19.53
4	[10, 10, 10, 10, 10]	0.8088	16.46
5	[20, 15, 10, 5, 3]	0.8055	11.60

Key insight: [15,10,10,10,0] nearly matches accuracy, and **2.7× faster**

Section 6.3

GPU Scaling

UNIVERSITY OF LUXEMBOURG

GPU Scaling: Configurations

Results

What's being tested?

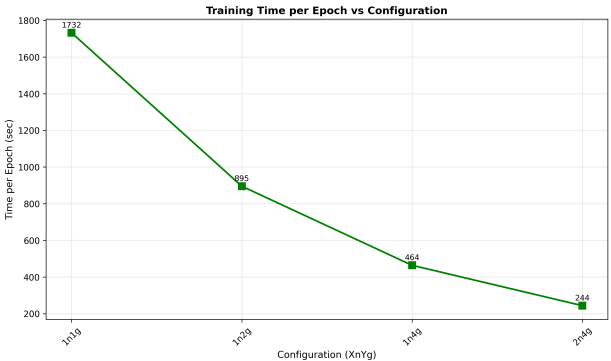
How does **distributed training** scale across:

- 1/2/4 GPUs per node
- 4 GPUs across 2 nodes (8 GPUs total)

Configuration	**Setup**
1 GPU	1 node
2 GPUs	1 node
4 GPUs	1 node
8 GPUs	2 nodes
Fixed Hyperparameters	
Batch size	128
Neighbors	[15,10,10,10,10]
Epochs	10

GPU Scaling: Training Time

Results



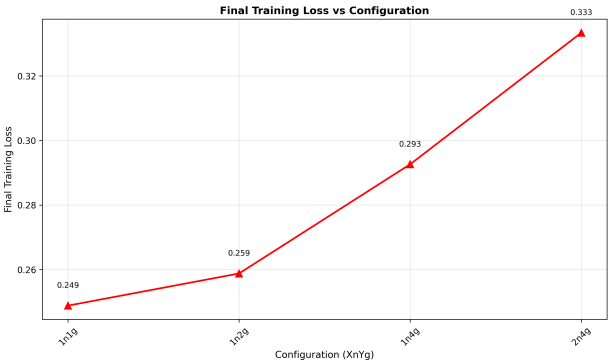
Key Findings:

- **1→2 GPUs:** 1.93× (96.5% efficiency)
- **1→4 GPUs:** 3.73× (93% efficiency)
- **1→8 GPUs:** 7.1× (89% efficiency)

UNIVERSITY OF LUXEMBOURG

GPU Scaling: Training Loss

Results



Key Findings:

- Loss increases with more GPUs
- Cause: larger effective batch size
- Wider minima, reduced gradient noise

Discussion

UNIVERSITY OF LUXEMBOURG

Bottlenecks Discovered

Discussion

- **Memory wall:** Neighbor sampling expansion dominates memory usage
  - Fanout [15,10,10,10,10] = 150k neighbors per seed node
  - 128 batch size × 150k expansion = 19.2M nodes per batch
  - Sampled subgraph + activations » model parameters (8MB)
- **Inter-node communication:** InfiniBand NCCL all-reduce overhead
  - 4 GPUs (1 node): 93% efficiency
  - 8 GPUs (2 nodes): 89% efficiency (4% degradation)
- **Loss convergence trade-off:** Larger effective batch sizes hurt convergence
  - Training loss increases with GPU count
  - Wider minima, reduced gradient noise

UNIVERSITY OF LUXEMBOURG

What Worked Well

Discussion

- **DDP implementation:** Seamless gradient synchronization across GPUs
  - — PyTorch DDP handles all-reduce automatically
  - — Minimal code changes from single-GPU baseline
- **Strong intra-node scaling:** Excellent up to 4 GPUs
  - — Linear speedups: 1→4 GPUs = 3.73× (93% efficiency)
  - — PCIe/NVLink bandwidth sufficient for this dataset size
- **Containerization:** Reproducibility & portability
  - — Apptainer/Singularity ensures consistent environment
  - — Complex PyG dependencies properly managed

Lessons Learned

Discussion

Implementing High-Performance Systems:

1. **Profile first, optimize second:** Memory was the true bottleneck, not computation
2. **Trade-offs are everywhere:** No free lunch -¿ accuracy, speed, memory, communication always compete
3. **Reproducibility matters:** Containerization and fixed seeds essential for scientific HPC work

UNIVERSITY OF LUXEMBOURG

Any Question?

UNIVERSITY OF LUXEMBOURG

References

References