The Nature of Graph Neural Network Workloads
Graphs & applications

• Large: up to billions of nodes
• Many graphs are Heterogeneous
• Rich node/edge attributes

Social networks

Amazon graph

Knowledge graphs
Graphs & applications

• Many small graphs

One graph has 100-1000 nodes, but there are many graphs.
A family of (deep) neural networks that learn node, edge, and graph embeddings.

How do GNNs work?

An ego-network around each node is used to learn an embedding that captures task-specific information.

The embeddings use both the structure of the graph and the features of the nodes and edges.

The embeddings are learned in an end-to-end fashion; thus, the predictions are a function of the target node’s ego-network.
A general graph neural network formalism

Graph neural networks are based on message-passing.

Message passing can be expressed with two sparse operators: SpMM and SDDMM.
SpMM

- **Edge-wise:** $m_e^{(t+1)} = x_u^{(t)}$
- **Node-wise:** $x_v^{(t+1)} = \sum_{(u,e,v) \in \mathcal{E}} m_e^{(t+1)}$

**Sparse-dense Matrix Multiplication**
SDDMM

• Edge-wise: $m_e^{(t+1)} = x_u^{(t)} x_v^{(t)}^T, (u, e, v) \in \mathcal{E}$
Generalized SpMM and SDDMM

• g-SpMM
  • Edge-wise: \( m_e^{(t+1)} = \phi \left( x_u^{(t)}, x_v^{(t)}, w_e^{(t)} \right) \)
  • Node-wise: \( x_v^{(t+1)} = \rho(\{m_e^{(t+1)}: (u, e, v) \in \mathcal{E}\}) \)

• g-SDDMM
  • Edge-wise: \( m_e^{(t+1)} = \phi \left( x_u^{(t)}, x_v^{(t)}, w_e^{(t)} \right), (u, e, v) \in \mathcal{E} \)
Model 1: GraphSage

Handles graphs with one node type and one edge type.

\[ M_{vw}^{(l)} = \frac{h_{w}^{(l-1)}}{d_v + 1} \]

\[ m_v^{(l)} = \sum_{w \in N(v) \cup \{v\}} M_{vw}^{(l)} \]

\[ h_v^{(l)} = \phi(m_v^{(l)}W1^{(l)} + h_v^{(l-1)}W2^{(l)}) \]

\[ M = \text{SpMM}(A, H)/\text{deg}(A) \]

\[ H = \text{ReLU}(\text{matmul}(M, W1) + b1 + \text{matmul}(H, W2) + b2) \]

\[ H = \text{Dropout}(H) \]
Model 2: Graph attention networks (GAT)

Handles graphs with one node type and one edge type.

\[ \alpha_{vw} = \frac{\exp(\text{LeakyReLU}(\tilde{a}^T [W\tilde{h}_v||W\tilde{h}_w]))}{\sum_{k \in N_v} \exp(\text{LeakyReLU}(\tilde{a}^T [W\tilde{h}_v||W\tilde{h}_k]))} \]

\[ M_{vw}^{(l)} = \alpha_{vw} h_w^{(l-1)} \]

\[ m_v^{(l)} = \sum_{w \in N(v) \cup \{v\}} M_{vw}^{(l)} \]

\[ h_v^{(l)} = \phi(m_v^{(l)}) \]

\[ H = \text{matmul}(W, H) \]
\[ E_l = \text{matmul}(W_{l_a}, H) \]
\[ E_r = \text{matmul}(W_{r_a}, H) \]
\[ E = \text{LeakyReLU}(\text{SDDMM}(E_l, E_r, A)) \]
\[ E = \text{edge_softmax}(E) \]
\[ M = \text{SpMM}(E, H) \]
\[ H = \text{ReLU}(M, W) \]
Model 3: Relational graph convolution networks (RGCN)

Handles graphs whose nodes are connected with different relations.

\[
M_{vw}^{(l)} = \frac{1}{c_{v,r}} W_r^{(l)} h_w^{(l-1)}, \text{ } r \text{ is the relation of } e_{vw}.
\]

\[
m_v^{(l)} = \sum_{w \in N(v) \cup \{v\}} M_{vw}^{(l)}
\]

\[
h_v^{(l)} = \sigma(m_v^{(l)})
\]

```python
M = []
# Sort the edges based on the edge type
src_id, dst_id, etype = g.edges()
etype, idx = sort(etype)
src_id, dst_id = src_id[idx], dst_id[idx]
H = H[src_id]
# Perform per-etype matrix multiplication
H_list = split(H, etypes)
for r in range(num_relations):
    M.append(matmul(H_list[r], W[r])/deg)
M = cat(M)
# Perform the final aggregation
A' = coo_matrix((range(num_edges), dst_id))
M = SpMM(A', M)
H = ReLU(M)
```
GNN benchmark

- **Graph types:**
  - Homogeneous
  - Heterogeneous

- **Graph size:**
  - Small
  - Large

- **Training methods:**
  - Full batch training
  - Mini-batch training

<table>
<thead>
<tr>
<th></th>
<th>Graph type</th>
<th>Graph Size</th>
<th>Training method</th>
</tr>
</thead>
<tbody>
<tr>
<td>OGBN-arxiv</td>
<td>Homogeneous</td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>OGBN-products</td>
<td>Homogeneous</td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td>MUTAG</td>
<td>Heterogeneous</td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$</td>
<td>ETYPE</td>
</tr>
<tr>
<td>OGBN-MAG</td>
<td>Heterogeneous</td>
<td>$</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$</td>
<td>ETYPE</td>
</tr>
</tbody>
</table>

Benchmark framework: Deep Graph Library (DGL)
Benchmark framework: Deep Graph Library (DGL)

• DGL provides sparse operators:
  • SpMM, SDDMM
  • Neighbor sampling for mini-batch training.

• Deep learning framework provides dense operators:
  • Matrix multiplication, element-wise operations, reduction, etc

• The benchmark uses DGL + Pytorch.
Full-graph training on small graphs

GraphSage full-batch training

<table>
<thead>
<tr>
<th>Setup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>OGBN-arxiv</td>
</tr>
<tr>
<td>#layers</td>
<td>3</td>
</tr>
<tr>
<td>Hidden dimensions</td>
<td>256</td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia T4</td>
</tr>
<tr>
<td>Model size</td>
<td>217K</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
M &= \text{SpMM}(A, H)/\text{deg}(A) \\
H &= \text{ReLU}(\text{matmul}(M, W_1) + b_1 + \text{matmul}(H, W_2) + b_2) \\
H &= \text{Dropout}(H)
\end{align*}
\]

Both sparse operations and dense operations have roughly the same amount of overhead.
GAT full-batch training

<table>
<thead>
<tr>
<th>Setup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>OGBN-arxiv</td>
</tr>
<tr>
<td>#layers</td>
<td>3</td>
</tr>
<tr>
<td>Hidden dimensions</td>
<td>256</td>
</tr>
<tr>
<td>#attention heads</td>
<td>3</td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia T4</td>
</tr>
<tr>
<td>Model size</td>
<td>1.4M</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
    H &= \text{matmul}(W, H) \\
    El &= \text{matmul}(Wl_a, H) \\
    Er &= \text{matmul}(Wr_a, H) \\
    E &= \text{LeakyReLU}(\text{SDDMM}(El, Er, A)) \\
    E &= \text{edge_softmax}(E) \\
    M &= \text{SpMM}(E, H) \\
    H &= \text{ReLU}(M, W)
\end{align*}
\]

Both sparse operations and dense operations have roughly the same amount of overhead.
RGCN full-batch training

<table>
<thead>
<tr>
<th>Setup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>mutag</td>
</tr>
<tr>
<td>#layers</td>
<td>2</td>
</tr>
<tr>
<td>Hidden dimensions</td>
<td>256</td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia T4</td>
</tr>
<tr>
<td>Model size</td>
<td>10M</td>
</tr>
</tbody>
</table>

```python
M = []
src_id, dst_id, etype = g.edges()
etype, idx = sort(etype)
src_id, dst_id = src_id[idx], dst_id[idx]
H = H[src_id]
H_list = split(H, etypes)
for r in range(num_relations):
    M.append(matmul(H_list[r], W[r])/deg)
M = cat(M)
A' = coo_matrix((range(num_edges), dst_id))
M = SpMM(A', M)
H = ReLU(M)
```
Mini-batch training on large graphs

- Another view of computing node embeddings.
- A mini batch represents the computation graph for target nodes.
- Small-world graphs lead to a huge computation graph.
Neighbor sampling

- Prune the computation graph:
  - Sample neighbors from a neighbor list of a vertex.
Mini-batch training on GPU

• Two ways of performing mini-batch training.
  • Pure GPU training: all data in GPU.
  • Mixed CPU-GPU training: the whole graph data in CPU and mini-batch computation in GPU.

• The benchmark covers pure GPU mini-batch training.
GraphSage mini-batch training

<table>
<thead>
<tr>
<th>Setup</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>OGBN-products</td>
</tr>
<tr>
<td>#layers</td>
<td>2</td>
</tr>
<tr>
<td>Hidden dimensions</td>
<td>256</td>
</tr>
<tr>
<td>fanout</td>
<td>25,10</td>
</tr>
<tr>
<td>Batch size</td>
<td>1000</td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia T4</td>
</tr>
<tr>
<td>Model size</td>
<td>217K</td>
</tr>
</tbody>
</table>

\[ M = \text{SpMM}(A, H)/\text{deg}(A) \]
\[ H = \text{ReLU}(\text{matmul}(M, W1) + b1 + \text{matmul}(H, W2) + b2) \]
\[ H = \text{Dropout}(H) \]

- Mini-batch construction is very expensive;
- Dense operations are much more expensive than sparse operations in mini-batch training.
GAT mini-batch training

<table>
<thead>
<tr>
<th>Setup</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataset</td>
<td>OGBN-products</td>
</tr>
<tr>
<td>#layers</td>
<td>3</td>
</tr>
<tr>
<td>Hidden dimensions</td>
<td>256</td>
</tr>
<tr>
<td>fanout</td>
<td>5,10,15</td>
</tr>
<tr>
<td>#attention heads</td>
<td>3</td>
</tr>
<tr>
<td>Batch size</td>
<td>1000</td>
</tr>
<tr>
<td>Hardware</td>
<td>Nvidia T4</td>
</tr>
<tr>
<td>Model size</td>
<td>1.4M</td>
</tr>
</tbody>
</table>

H = matmul(W, H)
El = matmul(Wl_a, H)
Er = matmul(Wr_a, H)
E = LeakyReLU(SDDMM(El, Er, A))
E = edge_softmax(E)
M = SpMM(E, H)
H = ReLU(M, W)

- Mini-batch computation of GAT is much more expensive; mini-batch construction is relatively cheap.
- Dense operations are much more expensive than sparse operations in mini-batch training.
RGCN mini-batch training

### Setup

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Dataset</strong></td>
<td>OGBN-MAG</td>
</tr>
<tr>
<td><strong>#layers</strong></td>
<td>2</td>
</tr>
<tr>
<td><strong>Hidden dimensions</strong></td>
<td>64</td>
</tr>
<tr>
<td><strong>Batch size</strong></td>
<td>512</td>
</tr>
<tr>
<td><strong>Fanout</strong></td>
<td>25,30</td>
</tr>
<tr>
<td><strong>Hardware</strong></td>
<td>Nvidia T4</td>
</tr>
<tr>
<td><strong>Model size</strong></td>
<td>309M</td>
</tr>
</tbody>
</table>

---

M = []
src_id, dst_id, etype = g.edges()
etype, idx = sort(etype)
src_id, dst_id = src_id[idx], dst_id[idx]
H = H[src_id]
H_list = split(H, etypes)
for r in range(num_relations):
    M.append(matmul(H_list[r], W[r])/deg)
M = cat(M)
A’ = coo_matrix((range(num_edges), dst_id))
M = SpMM(A’, M)
H = ReLU(M)
Summary

• For GNN workloads, both sparse and dense operations are important.
• Training methods have large impact on GNN workloads.
  • For full-graph training, both sparse and dense operations account for half of runtime.
  • For mini-batch training, runtime are more dominated by dense operations.
  • Mini-batch sampling may cause significant effort during training.