A General Perspective on Graph Neural Networks
Modern ML Toolbox

Modern deep learning toolbox is designed for simple sequences & grids.
Data is stored in **Relational Tables**

- **Commerce**
- **Finance**
- **Health care**
Relational data is a graph!
But networks are far more complex!

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)
- No fixed node ordering or reference point
- Often dynamic and have multimodal features
Graph Neural Networks
A Naïve Approach

- Join adjacency matrix and features
- Feed them into a deep neural net:

Issues with this idea:
- $O(|V|)$ parameters
- Not applicable to graphs of different sizes
- Sensitive to node ordering
CNN on an image:

Goal is to generalize convolutions to graphs

Credit: Stanford CS224W
Single Convolutional neural network (CNN) layer with 3x3 filter:

**Idea:** transform information at the neighbors and combine it:
- Transform “messages” $h_i$ from neighbors: $W_i h_i$
- Add them up: $\sum_i W_i h_i$

Credit: Stanford CS224W
Idea: Node’s neighborhood defines a computation graph

Determine node computation graph
Propagate and transform information

Learn how to propagate information across the graph to compute node features

Credit: Stanford CS224W
Key idea: Generate node embeddings based on local network neighborhoods.
**Intuition**: Nodes aggregate information from their neighbors using neural networks.
**Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!

Credit: Stanford CS224W
Model can be of arbitrary depth:

- Nodes have embeddings at each layer
- Layer-0 embedding of node $u$ is its input feature, $x_u$
- Layer-$k$ embedding gets information from nodes that are $K$ hops away
Neighborhood aggregation: Key distinctions are in how different approaches aggregate information across the layers.

What is in the box?

TARGET NODE

INPUT GRAPH

Credit: Stanford CS224W
**Basic approach:** Average information from neighbors and apply a neural network

1. Average messages from neighbors
2. Apply neural network

Credit: Stanford CS224W
Assume we have a graph $G$:

- $V$ is the vertex set
- $A$ is the adjacency matrix (assume binary)
- $X \in \mathbb{R}^{m \times |V|}$ is a matrix of node features
- $v$: a node in $V$; $N(v)$: the set of neighbors of $v$.

**Node features:**

- Relational data: User/item descriptions, categories
- Social networks: User profile, User image
- Biological networks: Gene expression profiles, gene functional information
- What if there is no node feature in the graph dataset?
Basic approach: Average neighbor messages and apply a neural network

- Initial 0-th layer embeddings are equal to node features:
  \[ h_v^0 = x_v \]

- Embedding of \( v \) at layer \( l \):
  \[ h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, ..., L - 1\} \]

- Average of neighbor’s previous layer embeddings:
  \[ z_v = h_v^{(L)} \]

- Non-linearity (e.g., ReLU):
  \[ h_v^{(l+1)} = \sigma(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, ..., L - 1\} \]

- Total number of layers:
  \[ z_v = h_v^{(L)} \]

Credit: Stanford CS224W
How do we train the model to generate embeddings?

Need to define a loss function on the embeddings

Credit: Stanford CS224W
Node embedding $z_v$ is a function of input graph

**Supervised setting**: we want to minimize the loss $\mathcal{L}$:

$$\min_{\Theta} \mathcal{L}(y, f(z_v))$$

- $y$: node label
- $\mathcal{L}$ could be L2 if $y$ is real number, or cross entropy if $y$ is categorical
Model Parameters

$$h_v^{(0)} = x_v$$

$$h_v^{(l+1)} = \sigma \left( W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)} \right), \forall l \in \{0, ..., L - 1\}$$

$$z_v = h_v^{(L)}$$

- We can feed these embeddings into any loss function and run SGD to train the weight parameters

- **Trainable weight matrices** (i.e., what we learn)

**Final node embedding**

$$h_v^{l} : \text{the hidden representation of node } v \text{ at layer } l$$

- **$W_l$**: weight matrix for neighborhood aggregation
- **$B_l$**: weight matrix for transforming hidden vector of self

---

Credit: Stanford CS224W
Directly train the model for a supervised task (e.g., node classification)

Is user going to churn in the next week?

E.g., a relational graph of users, sales, products
Directly train the model for a supervised task (e.g., node classification)

- Use cross entropy loss

$$L = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))$$

Credit: Stanford CS224W
Designing a GNN
GNN Layer = Message + Aggregation

- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

GNN Layer 1

(1) Message

(2) Aggregation
Connect GNN layers into a GNN
- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity
Idea: Raw input graph ≠ computational graph

- Graph feature augmentation
- Graph structure augmentation

(4) Graph augmentation
How do we train a GNN

- Supervised/Unsupervised objectives
- Node/Edge/Graph level objectives
A General GNN Framework (5)

(5) Learning objective

(2) Aggregation

(1) Message

(3) Layer connectivity

GNN Layer 1

GNN Layer 2

(4) Graph augmentation

J. You, R. Ying, J. Leskovec. Design Space of Graph Neural Networks, NeurIPS 2020
A Single Layer of a GNN
GNN Layer = Message + Aggregation

- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

J. You, R. Ying, J. Leskovec. *Design Space of Graph Neural Networks*, NeurIPS 2020
A Single GNN Layer

- **Idea of a GNN Layer:**
  - Compress a set of vectors into a single vector
  - Two step process:
    - (1) Message
    - (2) Aggregation

Input node embedding $h_v^{(l-1)}$, $h_{u \in N(v)}^{(l-1)}$

Output node embedding $h_v^{(l)}$

Node $v$
(1) Message computation

- **Message function:** \( m_{u}^{(l)} = MSG^{(l)} \left( h_{u}^{(l-1)} \right) \)
  - **Intuition:** Each node will create a message, which will be sent to other nodes later
  - **Example:** A Linear layer \( m_{u}^{(l)} = W^{(l)} h_{u}^{(l-1)} \)
    - Multiply node features with weight matrix \( W^{(l)} \)

(2) Aggregation
(2) Aggregation

**Intuition:** Each node will aggregate the messages from node $v$'s neighbors

$$h_v^{(l)} = \text{AGG}^{(l)} \left( \{ m_u^{(l)}, u \in N(v) \} \right)$$

**Example:** Sum($\cdot$), Mean($\cdot$) or Max($\cdot$) aggregator

$$h_v^{(l)} = \text{Sum}(\{ m_u^{(l)}, u \in N(v) \})$$
Message Aggregation: Issue

- **Issue**: Information from node \( v \) itself **could get lost**
  - Computation of \( h_v^{(l)} \) does not directly depend on \( h_v^{(l-1)} \)

- **Solution**: Include \( h_v^{(l-1)} \) when computing \( h_v^{(l)} \)
  - **(1) Message**: compute message from node \( v \) itself
    - Usually, a **different message computation** will be performed
      \[
      m_u^{(l)} = W^{(l)} h_u^{(l-1)} \quad \quad m_v^{(l)} = B^{(l)} h_v^{(l-1)}
      \]

  - **(2) Aggregation**: After aggregating from neighbors, we can aggregate the message from node \( v \) itself
    - Via **concatenation** or **summation**

\[
 h_v^{(l)} = \text{CONCAT} \left( \text{AGG} \left( \left\{ m_u^{(l)} , u \in N(v) \right\} \right), m_v^{(l)} \right)
\]

First aggregate from neighbors

Then aggregate from node itself
Putting things together:

- **(1) Message**: each node computes a message
  \[ m_u^{(l)} = MSG^{(l)} \left( h_u^{(l-1)} \right), u \in \{N(v) \cup v\} \]

- **(2) Aggregation**: aggregate messages from neighbors
  \[ h_v^{(l)} = AGG^{(l)} \left( \{m_u^{(l)}, u \in N(v)\}, m_v^{(l)} \right) \]

- **Nonlinearity (activation)**: Adds expressiveness
  - Often written as \( \sigma(\cdot): \text{ReLU}(\cdot), \text{Sigmoid}(\cdot), \ldots \)
  - Can be added to message or aggregation
Classical GNN Layers: GCN (1)

- (1) Graph Convolutional Networks (GCN)

\[ h_v^{(l)} = \sigma \left( W^{(l)} \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \right) \]

- How to write this as Message + Aggregation?

\[ h_v^{(l)} = \sigma \left( \sum_{u \in N(v)} W^{(l)} \frac{h_u^{(l-1)}}{|N(v)|} \right) \]
(1) Graph Convolutional Networks (GCN)

\[
h^{(l)}_v = \sigma \left( \sum_{u \in N(v)} W^{(l)} \frac{h^{(l-1)}_u}{|N(v)|} \right)
\]

- **Message:**
  - Each Neighbor: \( m^{(l)}_u = \frac{1}{|N(v)|} W^{(l)} h^{(l-1)}_u \)

- **Aggregation:**
  - Sum over messages from neighbors, then apply activation
  - \( h^{(l)}_v = \sigma \left( \text{Sum} \left( \{ m^{(l)}_u, u \in N(v) \} \right) \right) \)

Normalized by node degree
(In the GCN paper they use a slightly different normalization)
Classical GNN Layers: GraphSAGE

(2) GraphSAGE

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

How to write this as Message + Aggregation?

- **Message** is computed within the \( \text{AGG}(\cdot) \)
- **Two-stage aggregation**
  - **Stage 1:** Aggregate from node neighbors
    \[ h_{N(v)}^{(l)} \leftarrow \text{AGG} \left( \{ h_u^{(l-1)}, \forall u \in N(v) \} \right) \]
  - **Stage 2:** Further aggregate over the node itself
    \[ h_v^{(l)} \leftarrow \sigma \left( W^{(l)} \cdot \text{CONCAT}(h_v^{(l-1)}, h_{N(v)}^{(l)}) \right) \]
(3) Graph Attention Networks

\[
    h_v^{(l)} = \sigma \left( \sum_{u \in N(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)} \right)
\]

- **In GCN / GraphSAGE**
  - \( \alpha_{vu} = \frac{1}{|N(v)|} \) is the **weighting factor (importance)** of node \( u \)'s message to node \( v \)
  - \( \Rightarrow \) \( \alpha_{vu} \) is defined **explicitly** based on the structural properties of the graph (node degree)
  - \( \Rightarrow \) All neighbors \( u \in N(v) \) are equally important to node \( v \)
Can we do better than simple neighborhood aggregation?

Can we let weighting factors $\alpha_{vu}$ to be learned?

- **Goal:** Specify **arbitrary importance** to different neighbors of each node in the graph
- **Idea:** Compute embedding $h^{(l)}_v$ of each node in the graph following an **attention strategy**:
  - Implicitly specify different weights to different nodes in a neighborhood
Let $\alpha_{vu}$ be computed as a byproduct of an attention mechanism $a$:

1. Let $a$ compute attention coefficients $e_{vu}$ across pairs of nodes $u, v$ based on their messages:

$$e_{vu} = a(W(l)h_u^{(l-1)}, W(l)h_v^{(l-1)})$$

- $e_{vu}$ indicates the importance of $u$'s message to node $v$

$$e_{AB} = a(W(l)h_A^{(l-1)}, W(l)h_B^{(l-1)})$$
Attention Mechanism (2)

- **Normalize** $e_{vu}$ into the final attention weight $\alpha_{vu}$
  - Use the **softmax** function, so that $\sum_{u \in N(v)} \alpha_{vu} = 1$:
    $$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

- **Weighted sum** based on the final attention weight $\alpha_{vu}$
  $$h_v^{(l)} = \sigma(\sum_{u \in N(v)} \alpha_{vu} W^{(l)} h_u^{(l-1)})$$

Weighted sum using $\alpha_{AB}, \alpha_{AC}, \alpha_{AD}$:
$$h_A^{(l)} = \sigma(\alpha_{AB} W^{(l)} h_B^{(l-1)} + \alpha_{AC} W^{(l)} h_C^{(l-1)} + \alpha_{AD} W^{(l)} h_D^{(l-1)})$$
What is the form of attention mechanism $a$?

- E.g., use a simple single-layer neural network
  - $a$ have trainable parameters (weights in the Linear layer)

$$e_{AB} = a \left( W^{(l)} h_A^{(l-1)}, W^{(l)} h_B^{(l-1)} \right)$$
$$= \text{Linear} \left( \text{Concat} \left( W^{(l)} h_A^{(l-1)}, W^{(l)} h_B^{(l-1)} \right) \right)$$

- Parameters of $a$ are trained jointly:
  - Learn the parameters together with weight matrices (i.e., other parameter of the neural net $W^{(l)}$) in an end-to-end fashion
Graph Manipulation in GNNs
Idea: Raw input graph ≠ computational graph

- Graph feature augmentation
- Graph structure manipulation

(4) Graph manipulation
Why Manipulate Graphs

Our assumption so far has been

- Raw input graph = computational graph

Reasons for breaking this assumption

- Feature level:
  - The input graph lacks features → feature augmentation

- Structure level:
  - The graph is too sparse → inefficient message passing
  - The graph is too dense → message passing is too costly
  - The graph is too large → cannot fit the computational graph into a GPU

- It’s just unlikely that the input graph happens to be the optimal computation graph for embeddings
Graph Manipulation Approaches

- **Graph Feature manipulation**
  - The input graph lacks features → feature augmentation

- **Graph Structure manipulation**
  - The graph is too sparse → Add virtual nodes / edges
  - The graph is too dense → Sample neighbors when doing message passing
  - The graph is too large → Sample subgraphs to compute embeddings
    - Will cover later in lecture: Scaling up GNNs
Why do we need feature augmentation?

- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix
- **Standard approaches:**
- **a) Assign constant values to nodes**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Why do we need feature augmentation?

- **(1) Input graph does not have node features**
  - This is common when we only have the adj. matrix

- **Standard approaches:**
  - b) Assign unique IDs to nodes
    - These IDs are converted into **one-hot vectors**

```
[0, 0, 0, 0, 1, 0]
```

**Total number of IDs = 6**
## Feature Augmentation on Graphs

- **Feature augmentation:** constant vs. one-hot

<table>
<thead>
<tr>
<th></th>
<th>Constant node feature</th>
<th>One-hot node feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant node feature</strong></td>
<td><img src="image1.png" alt="Constant" /></td>
<td><img src="image2.png" alt="One-hot" /></td>
</tr>
<tr>
<td><strong>Expressive power</strong></td>
<td><strong>Medium.</strong> All the nodes are identical, but GNN can still learn from the graph structure</td>
<td><strong>High.</strong> Each node has a unique ID, so node-specific information can be stored</td>
</tr>
<tr>
<td><strong>Inductive learning (Generalize to unseen nodes)</strong></td>
<td><strong>High.</strong> Simple to generalize to new nodes: we assign constant feature to them, then apply our GNN</td>
<td><strong>Low.</strong> Cannot generalize to new nodes: new nodes introduce new IDs, GNN doesn’t know how to embed unseen IDs</td>
</tr>
<tr>
<td><strong>Computational cost</strong></td>
<td><strong>Low.</strong> Only 1 dimensional feature</td>
<td><strong>High.</strong> (O(</td>
</tr>
<tr>
<td><strong>Use cases</strong></td>
<td>Any graph, inductive settings (generalize to new nodes)</td>
<td>Small graph, transductive settings (no new nodes)</td>
</tr>
</tbody>
</table>
Why do we need feature augmentation?

- (2) Certain features can help GNN learning
- Other commonly used augmented features:
  - Node degree
  - PageRank
  - Clustering coefficient
  - …

- Any useful graph statistics can be used!
Motivation: Augment sparse graphs

(1) Add virtual edges

- Common approach: Connect 2-hop neighbors via virtual edges
- Intuition: Instead of using adj. matrix $A$ for GNN computation, use $A + A^2$

Use cases: Bipartite graphs

- Author-to-papers (they authored)
- 2-hop virtual edges make an author-author collaboration graph
Motivation: Augment sparse graphs

(2) Add virtual nodes

- The virtual node will connect to all the nodes in the graph
  - Suppose in a sparse graph, two nodes have shortest path distance of 10
  - After adding the virtual node, all the nodes will have a distance of 2
    - Node A – Virtual node – Node B

Benefits: Greatly improves message passing in sparse graphs
Previously:

- All the nodes are used for message passing

New idea: (Randomly) sample a node’s neighborhood for message passing
For example, we can randomly choose 2 neighbors to pass messages

- Only nodes $B$ and $D$ will pass message to $A$
Next time when we compute the embeddings, we can sample different neighbors

- Only nodes $C$ and $D$ will pass message to $A$
In expectation, we can get embeddings similar to the case where all the neighbors are used

- **Benefits**: greatly reduce computational cost
- And in practice it works great!