A General Perspective on Graph Neural Networks

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
Mina Ghashami, Amazon
http://cs246.stanford.edu
Modern deep learning toolbox is designed for simple sequences & grids.
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
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
Real-World Graphs

But our graphs look like this:

- There is no fixed notion of locality or sliding window on the graph
- Graph is permutation invariant

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
**Idea: Aggregate Neighbors**

- **Intuition:** Nodes aggregate information from their neighbors using neural networks.

![Input Graph](image)

![Neural Networks Diagram](image)

Credit: Stanford CS224W
**Intuition:** Network neighborhood defines a computation graph

Every node defines a computation graph based on its neighborhood!
Deep Model: Many Layers

- 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?
Basic approach: Average information from neighbors and apply a neural network

(1) average messages from neighbors

(2) apply neural network
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
- $\nu$: a node in $V$; $N(\nu)$: the set of neighbors of $\nu$.

Node features:

- Social networks: User profile, User image
- Biological networks: Gene expression profiles, gene functional information

When there is no node feature in the graph dataset:

- Indicator vectors (one-hot encoding of a node)
- Vector of constant 1: $[1, 1, \ldots, 1]$
Basic approach: Average neighbor messages and apply a neural network

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

Initial 0-th layer embeddings are equal to node features

Embedding after L layers of neighborhood aggregation

Total number of layers

Average of neighbor's previous layer embeddings

Non-linearity (e.g., ReLU)

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

Need to define a loss function on the embeddings

Credit: Stanford CS224W
We can feed these embeddings into any loss function and run SGD to train the weight parameters:

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

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

Final node embedding

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

\( h_v^l \): the hidden representation of node \( v \) at layer \( l \)
- \( W_k \): weight matrix for neighborhood aggregation
- \( B_k \): weight matrix for transforming hidden vector of self

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

**Supervised setting**: we want to minimize the loss $\mathcal{L}$ (see also slide 15):

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

- $y$: node label
- $\mathcal{L}$ could be L2 if $y$ is real number, or cross entropy if $y$ is categorical
Directly train the model for a supervised task (e.g., node classification)

Safe or toxic drug?

Safe or toxic drug?

E.g., a drug-drug interaction network
Directly train the model for a supervised task (e.g., node classification)

- Use cross entropy loss (slide 16)

\[
\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(z_v^T \theta)) + (1 - y_v) \log(1 - \sigma(z_v^T \theta))
\]
Designing a GNN

CS246: Mining Massive Datasets
Jure Leskovec, Stanford University
http://cs246.stanford.edu
A General GNN Framework (1)

GNN Layer = Message + Aggregation
• Different instantiations under this perspective
• GCN, GraphSAGE, GAT, ...

GNN Layer 1
(1) Message
(2) Aggregation
A General GNN Framework (2)

Connect GNN layers into a GNN
- Stack layers sequentially
- Ways of adding skip connections

(3) Layer connectivity

INPUT GRAPH

TARGET NODE

GNN Layer 1

GNN Layer 2
A General GNN Framework (3)

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
A Single Layer of a GNN
A GNN Layer

GNN Layer = Message + Aggregation
- Different instantiations under this perspective
- GCN, GraphSAGE, GAT, ...

(1) Message
(2) Aggregation

Jure Leskovec & Mina Ghashami. *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

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

Input node embedding $h_v^{(l-1)}$, $h_{u \in N(v)}^{(l-1)}$
(from node itself + neighboring nodes)
(1) Message computation

- **Message function:** \( m_u^{(l)} = \text{MSG}^{(l)}(h_u^{(l-1)}) \)
  - **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

---

Jure Leskovec & Mina Ghashami, Stanford University
(2) Aggregation

**Intuition:** Each node will aggregate the messages from node \( v \)'s neighbors

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

**Example:** Sum(·), Mean(·) or Max(·) 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**

Then aggregate from node itself

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

First aggregate from neighbors
A Single GNN Layer

- **Putting things together:**
  - (1) **Message:** each node computes a message
    \[
    m_u^{(l)} = MSG^{(l)} \left( h_u^{(l-1)} \right), \quad u \in \{N(v) \cup v\}
    \]
  - (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 \(\sigma(\cdot):\) ReLU(\(\cdot\)), Sigmoid(\(\cdot\)), ...
    - Can be added to message or aggregation
Classical GNN Layers: GCN (1)

(1) Graph Convolutional Networks (GCN)

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

How to write this as Message + Aggregation?

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

Message

Aggregation

Jure Leskovec & Mina Ghashami, Stanford University
Classical GNN Layers: GCN (2)

(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
    \[ h^{(l)}_v = \sigma \left( \text{Sum} \left( \left\{ m^{(l)}_u, u \in N(v) \right\} \right) \right) \]

Normalized by node degree
(In the GCN paper they use a slightly different normalization)
(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
      \[ h_v^{(l)} \leftarrow \sigma \left( W^{(l)} \cdot \text{CONCAT}( h_v^{(l-1)}, h_{N(v)}^{(l)} ) \right) \]
GraphSAGE Neighbor Aggregation

- **Mean**: Take a weighted average of neighbors

\[
AGG = \frac{\sum_{u \in N(v)} h_u^{(l-1)}}{|N(v)|}
\]

- **Pool**: Transform neighbor vectors and apply symmetric vector function Mean(·) or Max(·)

\[
AGG = \text{Mean}\left(\{\text{MLP}(h_u^{(l-1)}), \forall u \in N(v)\}\right)
\]
GraphSAGE: L2 Normalization

\( \ell_2 \) Normalization:

- **Optional:** Apply \( \ell_2 \) normalization to \( h_v^{(l)} \) at every layer

\[
h_v^{(l)} \leftarrow \frac{h_v^{(l)}}{\|h_v^{(l)}\|_2} \quad \forall v \in V \text{ where } \|u\|_2 = \sqrt{\sum_i u_i^2} \text{ (}\ell_2\text{-norm)}
\]

- Without \( \ell_2 \) normalization, the embedding vectors have different scales (\( \ell_2 \)-norm) for vectors
- In some cases (not always), normalization of embedding results in performance improvement
- After \( \ell_2 \) normalization, all vectors will have the same \( \ell_2 \)-norm
(3) Graph Attention Networks

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

Attention weights

In GCN / GraphSAGE

- \( \alpha_{vu} = \frac{1}{|N(v)|} \) is the \textbf{weighting factor (importance)} of node \( u \)'s message to node \( v \)
- \( \Rightarrow \alpha_{vu} \) is defined \textbf{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 $h_v^{(l)}$ of each node in the graph following an attention strategy:
- Nodes attend over their neighborhoods’ message
- Implicitly specifying different weights to different nodes in a neighborhood
Attention Mechanism (1)

- 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 $v$

$$
\begin{align*}
    e_{AB} &= a(W^{(l)}h_A^{(l-1)}, W^{(l)}h_B^{(l-1)})
\end{align*}
$$

Jure Leskovec & Mina Ghashami, Stanford University
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 $\alpha_{AB}$, $\alpha_{AC}$, $\alpha_{AD}$:
$$
  h_A^{(l)} = \sigma(\alpha_{AB} W^{(l)} h_B^{(l-1)} + \alpha_{AC} W^{(l)} h_C^{(l-1)} + \alpha_{AD} W^{(l)} h_D^{(l-1)})
  $$

Jure Leskovec & Mina Ghashami, Stanford University
What is the form of attention mechanism \( a \)?

- The approach is agnostic to the choice of \( a \)
  - 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
Multi-head attention: Stabilizes the learning process of attention mechanism

Create multiple attention scores (each replica with a different set of parameters):

\[
\begin{align*}
  h_v^{(l)}[1] &= \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^1 W^{(l)} h_u^{(l-1)} \right) \\
  h_v^{(l)}[2] &= \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^2 W^{(l)} h_u^{(l-1)} \right) \\
  h_v^{(l)}[3] &= \sigma \left( \sum_{u \in N(v)} \alpha_{vu}^3 W^{(l)} h_u^{(l-1)} \right)
\end{align*}
\]

Outputs are aggregated:

- By concatenation or summation

\[
  h_v^{(l)} = \text{AGG}(h_v^{(l)}[1], h_v^{(l)}[2], h_v^{(l)}[3])
\]
Benefits of Attention Mechanism

- **Key benefit:** Allows for (implicitly) specifying different importance values ($\alpha_{vu}$) to different neighbors

- **Computationally efficient:**
  - Computation of attentional coefficients can be parallelized across all edges of the graph
  - Aggregation may be parallelized across all nodes

- **Storage efficient:**
  - Sparse matrix operations do not require more than $O(V + E)$ entries to be stored
  - **Fixed** number of parameters, irrespective of graph size

- **Localized:**
  - Only attends over local network neighborhoods

- **Inductive capability:**
  - It is a shared edge-wise mechanism
  - It does not depend on the global graph structure
Apply activation to $i$-th dimension of embedding $x$

- **Rectified linear unit (ReLU)**
  \[
  \text{ReLU}(x_i) = \max(x_i, 0)
  \]
  - Most commonly used

- **Sigmoid**
  \[
  \sigma(x_i) = \frac{1}{1 + e^{-x_i}}
  \]
  - Used only when you want to restrict the range of your embeddings

- **Parametric ReLU**
  \[
  \text{PReLU}(x_i) = \max(x_i, 0) + a_i \min(x_i, 0)
  \]
  - $a_i$ is a trainable parameter
  - Empirically performs better than ReLU
Graph Manipulation in GNNs
General GNN Framework

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 $\rightarrow$ feature augmentation

- **Graph Structure manipulation**
  - The graph is too sparse $\rightarrow$ Add virtual nodes / edges
  - The graph is too dense $\rightarrow$ Sample neighbors when doing message passing
  - The graph is too large $\rightarrow$ 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**
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**

---

**Diagram:**

- **INPUT GRAPH**
  - Nodes labeled A, B, C, D, E, F
  - Edges connecting the nodes

- **Example:**
  - Node with ID = 5
  - One-hot vector: 
    
    \[
    [0, 0, 0, 0, 1, 0] \]

- **Total number of IDs = 6**

---

Jure Leskovec & Mina Ghashami, Stanford University
## Feature Augmentation on Graphs

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

<table>
<thead>
<tr>
<th></th>
<th><strong>Constant node feature</strong></th>
<th><strong>One-hot node feature</strong></th>
</tr>
</thead>
<tbody>
<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</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
Add Virtual Nodes / Edges

- **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!
Recap: A general perspective for GNNs

- GNN Layer:
  - Transformation + Aggregation
  - Classic GNN layers: GCN, GraphSAGE, GAT

- Layer connectivity:
  - Deciding number of layers
  - Skip connections

- Graph Manipulation:
  - Feature augmentation
  - Structure manipulation