AT82.10: Recent Trends In Machine Learning / Deep Learning

Handout for the papers:

0. THE GRAPH NEURAL NETWORK MODEL (GNN)
1. SEMI-SUPERVISED CLASSIFICATION WITH GRAPH CONVOLUTIONAL NETWORKS (GCN)
2. GRAPH ATTENTION NETWORKS (GAT)

Scan for Code Implementations

Prepared by: **Arya Shah, st125462**
Under the guidance of: **Dr. Chaklam Silpasuwanchai, Pranissa Charnparttaravanit**

The purpose of this handout is to have it as reference during my presentation of the above mentioned papers. Since all of us come from diverse backgrounds and none of us have had prior exposure to the topic of Graph Neural Networks, I have also included the original paper which proposes Graph Neural Networks and we slowly build our way towards GCN, followed by GAT.

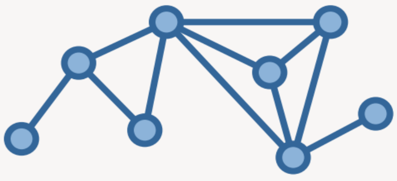
Before we begin, here is a quick overview of all three papers:

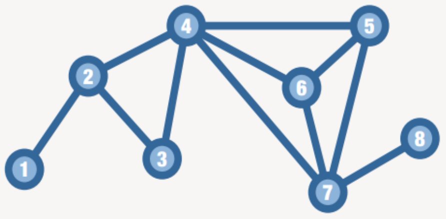
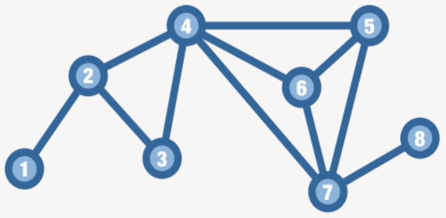
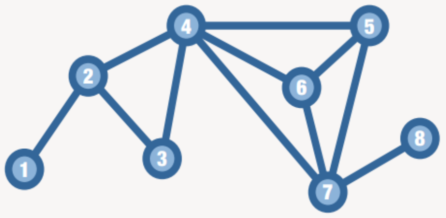
Aspect	**[2009] Graph Neural Network (GNN)**	**[2016] Graph Convolutional Networks (GCN)**	**[2017] Graph Attention Networks (GAT)**
Main Point	Extends neural networks to process graph-structured data through information diffusion and relaxation mechanisms	Enables efficient semi-supervised learning on graphs by approximating spectral convolutions	Introduces self-attention mechanism for graph neural networks, enabling different weights for neighboring nodes
Problem Addressed	Processing and learning from graph-structured data where relationships between objects are represented as graphs	Semi-supervised node classification in graphs where labels are only available for a small subset of nodes	Developing neural network architectures that can effectively process graph-structured data while addressing limitations of previous graph convolutional approaches
Importance	Many real-world problems involve relational data naturally represented as graphs, requiring models that can exploit both node features and graph topology	Graph-based data is prevalent in real-world applications, but traditional methods have limitations in effectively utilizing both graph structure and node features	Graph-structured data is prevalent in many domains but cannot be processed by traditional CNNs designed for grid-like structures

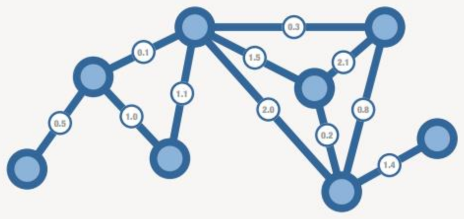
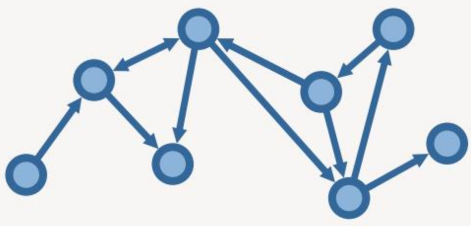
Previous Works	Recursive neural networks (limited to directed acyclic graphs) and random walk models; most traditional methods required preprocessing to convert graphs to vectors	Graph Laplacian regularization methods, skip-gram based graph embeddings, and early graph neural networks; often involved multi-step pipelines or restrictive assumptions	Spectral methods (GCN, Chebyshev filters) which depend on graph structure, and non-spectral methods (GraphSAGE) which sample fixed-size neighborhoods
Key Innovation	Framework where each node has a state that depends on its neighborhood, updated through diffusion until equilibrium; unifies recursive neural networks and random walk approaches	Simplified layer-wise propagation rule based on first-order approximation of spectral graph convolutions; encodes both local graph structure and node features	Masked self-attention layers that assign different importance to nodes in a neighborhood; computationally efficient and parallelizable
Results	State-of-the-art results on Mutagenesis dataset (96% accuracy on regression-unfriendly part); demonstrated on subgraph matching and web page ranking	Outperformed state-of-the-art on citation networks (e.g., 81.5% accuracy on Cora vs. 75.7% for previous best); faster training times	State-of-the-art results on four benchmarks: Cora (83.0%), Citeseer (72.5%), Pubmed (79.0%), and PPI dataset (0.973 micro-F1 score)
Key Advantage	Handles general graph types including directed, undirected, cyclic, or mixed	Scales linearly with graph edges	Directly applicable to inductive learning problems

---

Now that we have the overview of the three papers sorted, let's begin slowly, slowly, taking baby steps. Here's some basic graph terminology and high level overview of methods for you to understand some aspects of these papers:

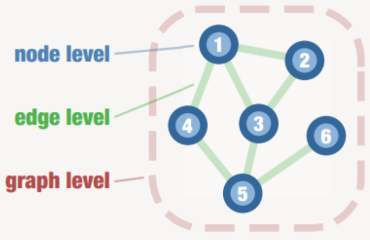
graph	a set of nodes and edges G = (V, E)	
node / vertex	what do the nodes represent? problem dependent (manual choice) properties: id, label (optional), feature (optional)	
edge	What do edges represent? problem dependent (manual choice) properties: id, weight (optional), direction (optional), label (optional), feature (optional)	

adjacency matrix	$A = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$	
list of edges	(1, 2) (2, 3) (2, 4) (3, 4) (4, 5) (4, 6) (4, 7) (5, 6) (5, 7) (6, 7) (7, 8)	
adjacency list	1: 2 2: 1, 3, 4 3: 2, 4 4: 2, 3, 5, 6, 7 5: 4, 6, 7 6: 4, 5, 7 7: 4, 5, 6, 8 8: 7	

edge weights	unweighted: all edges represented as “1” in adjacency matrix weighted: all real numbers possible	 0.1 0.3 1.5 2.1 1.1 1.0 0.5 2.0 0.8 0.2 1.4
edge directions (directed / undirected graph)	examples undirected: collaborations, friendships on facebook examples directed: phone calls, Instagram followers	

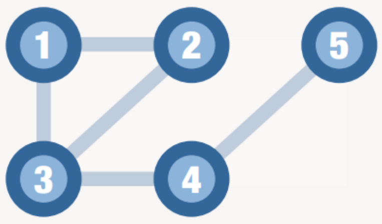
target levels for representation

- Node level representation
  Example: Social network bot / troll detection
- Edge level representation
  Example: Social network missing link prediction
- Graph level representation
  Example: Molecular property prediction (e.g., toxicity, solubility coeffcient)

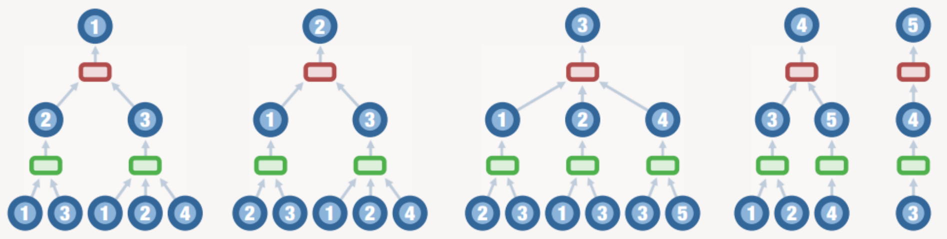


representational graph

- Refers to graph representation of the problem at hand

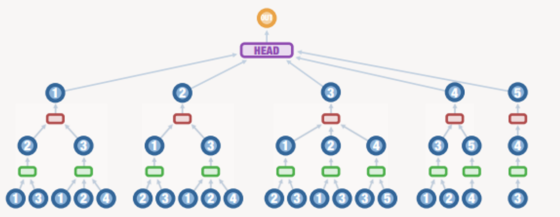


GNN compute graph



- Derived (unrolled) from representational graph above
- Shown here: 2 hops compute graph (i.e., compute graph for a 2-layer GNN)
- Green: Layer 1 (message compute & aggregation)
- Red: Layer 2 (message compute & aggregation)
- Same color (red / green) = shared weights
- Representational graph can be derived from compute graph and vice versa

getting to the target level



- In most cases, core GNNs produce node-level representations
- Edge-level representation: aggregating representations of adjacent nodes
- Graph-level representation: aggregating representations of all nodes in the graph
- Example above: graph level task

Head for edge representations:

$$y_{uv} = Head_{edge}\left\{h_u^{(l)}, h_v^{(l)}\right\}$$

Examples:

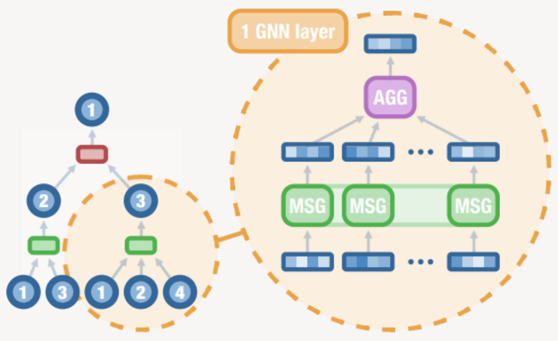
$$y_{uv} = Linear(Concat(h_u^{(l)}, h_v^{(l)}))$$
$$y_{uv} = h_u^{(l)} + h_v(l)$$
$$y_{uv} = dot(h_u^{(l)}, h_v^{(l)})$$

Head for graph representations:

$$y_G = Head_{graph}\left\{(h_v^{(l)}) \in R^d, \forall v \in G)\right\}$$

message computation

- Node features get transformed by a function MSG(), typically a neural network (MLP)
- Input: fixed-length feature vector (concatenation or sum of multiple embeddings)
- Output: Message (fixed-length vector)
- Function is applied to each input vector individually



One GNN layer:

$$h_v^{(l)} = AGG^{(l)}\left(\left\{MSG^{(l)}\left(h_u^{(l-1)}\right), u \in N(v)\right\}\right)$$

Alternatives for $MSG(\cdot)$:

$$MSG^{(l)}\left(h_u^{(l-1)}\right) = m_u^{(l)} = W^{(l)} h_u^{(l-1)}$$
$$MSG^{(l)}\left(h_u^{(l-1)}\right) = m_u^{(l)} = MLP(h_u^{(l-1)})$$

message aggregation

- Multiple messages get aggregated into a single vector by a function AGG()
- Input: N messages
- Output: Fixed-length vector (typically of same size)
- Requirement: Permutation invariance (no concatenation!)
- Examples: Sum(), Mean(), Max()
- Optimal: MLP(Sum(message)) with message = MLP(feature)

Alternatives for $AGG(\cdot)$:

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

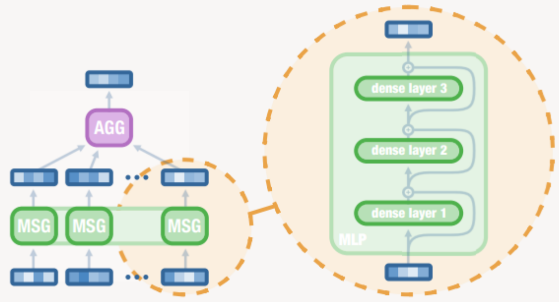
$$AGG^{(l)}(\cdot) = Mean(\cdot)$$
$$AGG^{(l)}(\cdot) = Max(\cdot)$$
$$AGG^{(l)}(\cdot) = MLP(Sum(\cdot))$$

Optional extensions:

$$h_v^{(l)} \leftarrow AGG^{(l)}\left(\left\{MSG^{(l)}\left(h_u^{(l-1)}\right), u \in N(v)\right\}\right)$$
$$h_v^{(l)} \leftarrow CONCAT\left(h_v^{(l)}, MSG_t^{(l)}\left(h_v^{(l-1)}\right)\right)$$
$$h_v^{(l)} \leftarrow \sigma\left(W^l \cdot h_v^{(l)}\right)$$
$$h_v^{(l)} \leftarrow \frac{h_v^{(l)}}{\|h_v^{(l)}\|_2}$$

layer connectivity

- Each GNN-layer increases "receptive field" by one hop
- GNNs typically have 2-3 layers (more is often not helpful)
- Instead:
  - Increase the size / number of layers of the networks within the GNN-layers
  - Pre-process raw features with deep networks
  - Add skip-connections



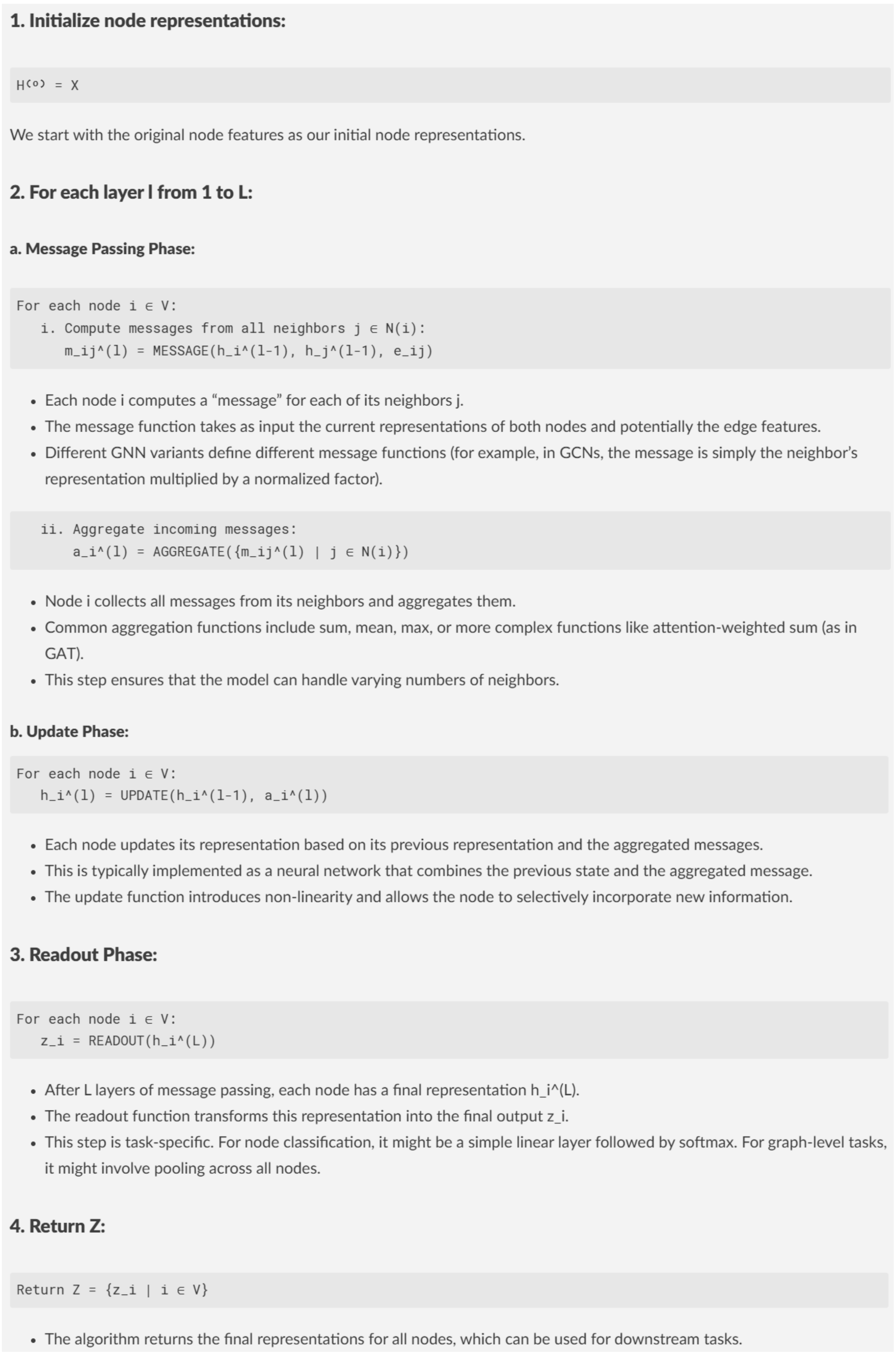
Pseudocode/Algorithm Explanations:

Graph Neural Network (GNN)

```
Algorithm: Message Passing Graph Neural Network (MPNN)
Input: Graph G = (V, E) with node features $X \in \mathbb{R}^{n \times d}$, edge features eij (optional)
Output: Node representations $Z \in \mathbb{R}^{n \times k}$ for downstream tasks

1. Initialize node representations: $H^{(0)} = X$

2. For each layer l from 1 to L:
   a. Message Passing Phase:
      For each node i ∈ V:
         i. Compute messages from all neighbors j ∈ N(i):
            m_ij^(l) = MESSAGE(h_i^(l-1), h_j^(l-1), e_ij)

         ii. Aggregate incoming messages:
             a_i^(l) = AGGREGATE({m_ij^(l) | j ∈ N(i)})

   b. Update Phase:
      For each node i ∈ V:
         h_i^(l) = UPDATE(h_i^(l-1), a_i^(l))

3. For each node i ∈ V:
   z_i = READOUT(h_i^(L))

4. Return Z = {z_i | i ∈ V}
```

Where:

- **MESSAGE** function computes a message based on the source node, target node, and edge features
- **AGGREGATE** function combines messages from all neighbors (e.g., sum, mean, max)
- **UPDATE** function updates the node representation based on its previous state and aggregated messages
- **READOUT** function transforms the final node representation for the target task



1. Initialize node representations:

$$H^{(0)} = X$$

We start with the original node features as our initial node representations.

2. For each layer l from 1 to L:

a. Message Passing Phase:

```
For each node i ∈ V:
   i. Compute messages from all neighbors j ∈ N(i):
      m_ij^(l) = MESSAGE(h_i^(l-1), h_j^(l-1), e_ij)
```

- Each node i computes a “message” for each of its neighbors j.
- The message function takes as input the current representations of both nodes and potentially the edge features.
- Different GNN variants define different message functions (for example, in GCNs, the message is simply the neighbor’s representation multiplied by a normalized factor).

```
   ii. Aggregate incoming messages:
       a_i^(l) = AGGREGATE({m_ij^(l) | j ∈ N(i)})
```

- Node i collects all messages from its neighbors and aggregates them.
- Common aggregation functions include sum, mean, max, or more complex functions like attention-weighted sum (as in GAT).
- This step ensures that the model can handle varying numbers of neighbors.

b. Update Phase:

```
For each node i ∈ V:
   h_i^(l) = UPDATE(h_i^(l-1), a_i^(l))
```

- Each node updates its representation based on its previous representation and the aggregated messages.
- This is typically implemented as a neural network that combines the previous state and the aggregated message.
- The update function introduces non-linearity and allows the node to selectively incorporate new information.

3. Readout Phase:

```
For each node i ∈ V:
   z_i = READOUT(h_i^(L))
```

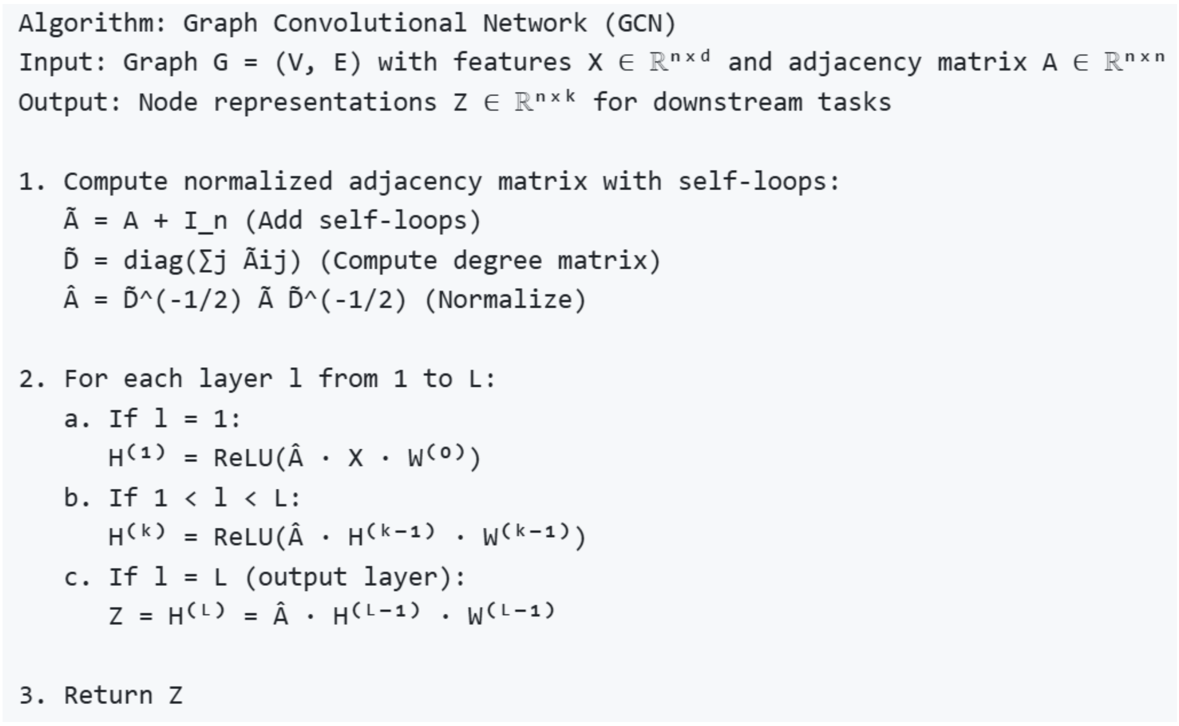
- After L layers of message passing, each node has a final representation h_i^(L).
- The readout function transforms this representation into the final output z_i.
- This step is task-specific. For node classification, it might be a simple linear layer followed by softmax. For graph-level tasks, it might involve pooling across all nodes.

4. Return Z:

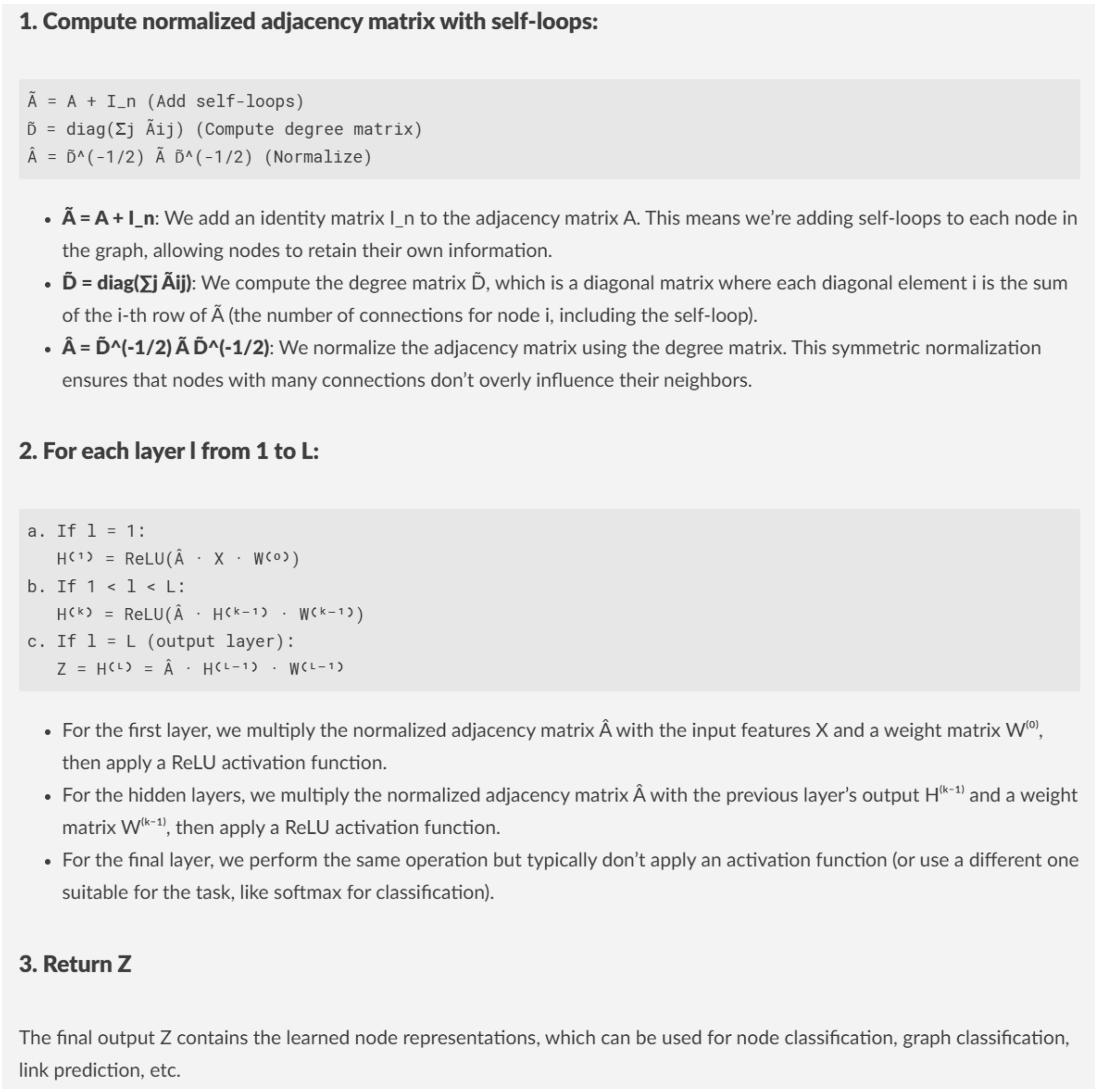
```
Return Z = {z_i | i ∈ V}
```

- The algorithm returns the final representations for all nodes, which can be used for downstream tasks.

Graph Convolution Network (GCN)



```
Algorithm: Graph Convolutional Network (GCN)
Input: Graph G = (V, E) with features $X \in \mathbb{R}^{n \times d}$ and adjacency matrix $A \in \mathbb{R}^{n \times n}$
Output: Node representations $Z \in \mathbb{R}^{n \times k}$ for downstream tasks

1. Compute normalized adjacency matrix with self-loops:
   $\tilde{A} = A + I_n$ (Add self-loops)
   $\tilde{D} = diag(\sum_j \tilde{A}_{ij})$ (Compute degree matrix)
   $\hat{A} = \tilde{D}^{(-1/2)} \tilde{A} \tilde{D}^{(-1/2)}$ (Normalize)

2. For each layer l from 1 to L:
   a. If l = 1:
      $H^{(1)} = ReLU(\hat{A} \cdot X \cdot W^{(0)})$
   b. If 1 < l < L:
      $H^{(k)} = ReLU(\hat{A} \cdot H^{(k-1)} \cdot W^{(k-1)})$
   c. If l = L (output layer):
      $Z = H^{(L)} = \hat{A} \cdot H^{(L-1)} \cdot W^{(L-1)}$

3. Return Z
```



1. Compute normalized adjacency matrix with self-loops:

```
Ã = A + I_n (Add self-loops)
D̃ = diag(Σj Ãij) (Compute degree matrix)
Â = D̃^(-1/2) Ã D̃^(-1/2) (Normalize)
```

- **Ã = A + I_n**: We add an identity matrix I_n to the adjacency matrix A. This means we're adding self-loops to each node in the graph, allowing nodes to retain their own information.
- **D̃ = diag(∑j Ãij)**: We compute the degree matrix D̃, which is a diagonal matrix where each diagonal element i is the sum of the i-th row of Ã (the number of connections for node i, including the self-loop).
- **Â = D̃^(-1/2) Ã D̃^(-1/2)**: We normalize the adjacency matrix using the degree matrix. This symmetric normalization ensures that nodes with many connections don't overly influence their neighbors.

2. For each layer l from 1 to L:

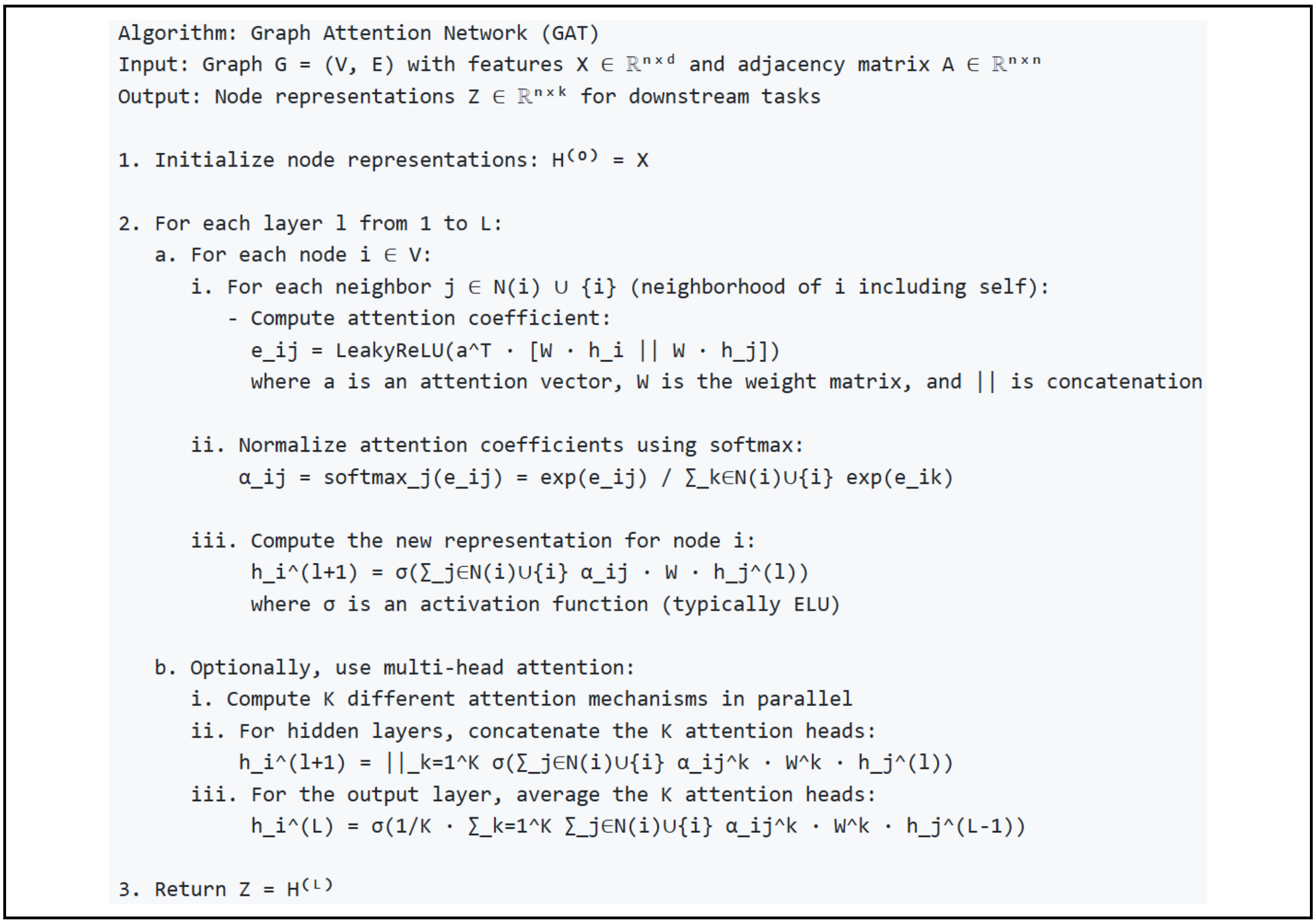
```
a. If l = 1:
   $H^{(1)} = ReLU(\hat{A} \cdot X \cdot W^{(0)})$
b. If 1 < l < L:
   $H^{(k)} = ReLU(\hat{A} \cdot H^{(k-1)} \cdot W^{(k-1)})$
c. If l = L (output layer):
   $Z = H^{(L)} = \hat{A} \cdot H^{(L-1)} \cdot W^{(L-1)}$
```

- For the first layer, we multiply the normalized adjacency matrix Â with the input features X and a weight matrix $W^{(0)}$, then apply a ReLU activation function.
- For the hidden layers, we multiply the normalized adjacency matrix Â with the previous layer's output $H^{(k-1)}$ and a weight matrix $W^{(k-1)}$, then apply a ReLU activation function.
- For the final layer, we perform the same operation but typically don't apply an activation function (or use a different one suitable for the task, like softmax for classification).

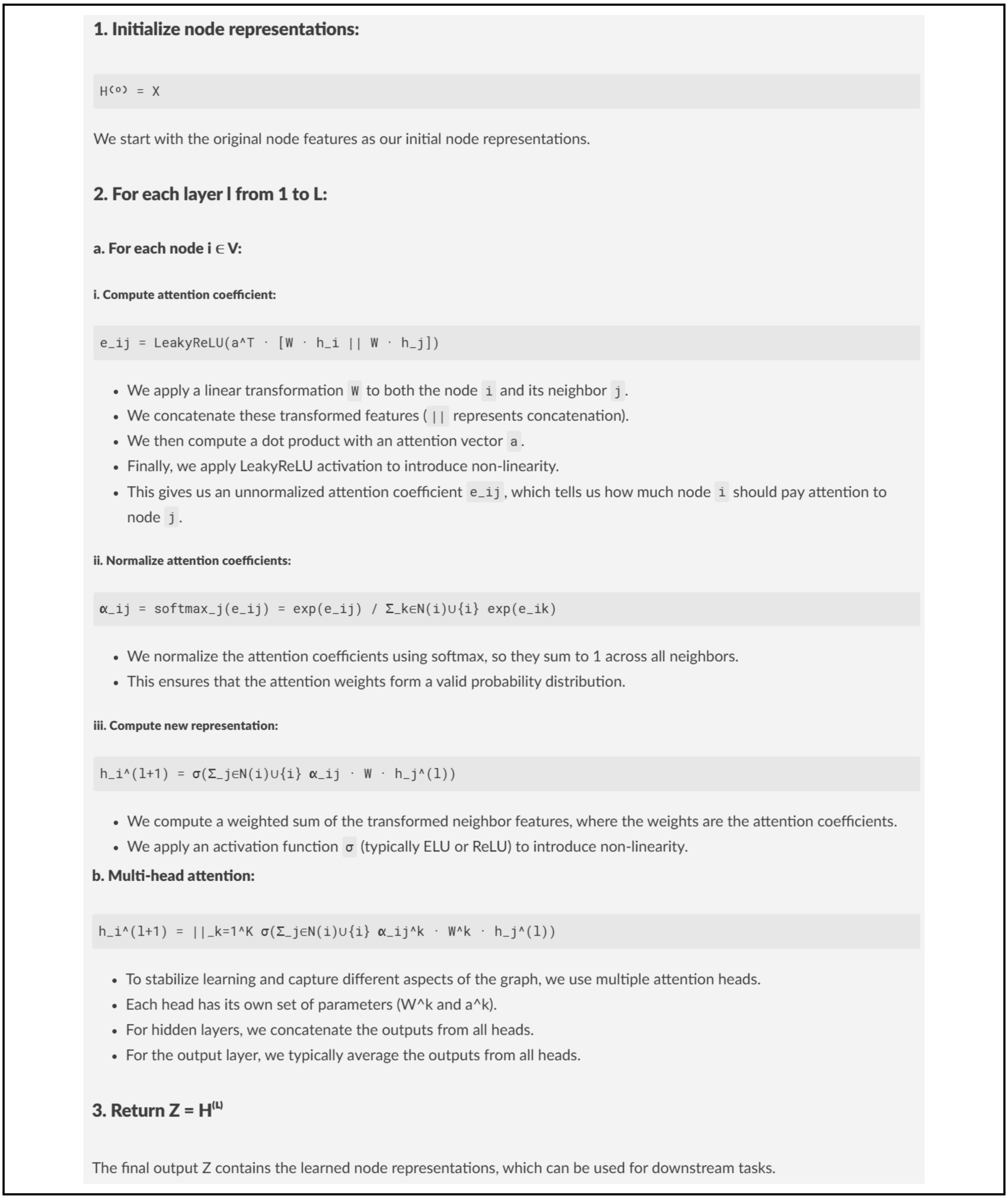
3. Return Z

The final output Z contains the learned node representations, which can be used for node classification, graph classification, link prediction, etc.

Graph Attention Network (GAT)



```
Algorithm: Graph Attention Network (GAT)
Input: Graph $G = (V, E)$ with features $X \in \mathbb{R}^{n \times d}$ and adjacency matrix $A \in \mathbb{R}^{n \times n}$
Output: Node representations $Z \in \mathbb{R}^{n \times k}$ for downstream tasks

1. Initialize node representations: $H^{(0)} = X$

2. For each layer l from 1 to L:
   a. For each node $i \in V$:
      i. For each neighbor $j \in N(i) \cup \{i\}$ (neighborhood of i including self):
         - Compute attention coefficient:
           $e_{ij} = \mathrm{LeakyReLU}(a^T \cdot [W \cdot h_i \,||\, W \cdot h_j])$
           where a is an attention vector, W is the weight matrix, and || is concatenation

      ii. Normalize attention coefficients using softmax:
          $\alpha_{ij} = \mathrm{softmax}_j(e_{ij}) = \exp(e_{ij}) / \sum_{k \in N(i) \cup \{i\}} \exp(e_{ik})$

      iii. Compute the new representation for node i:
           $h_i^{(l+1)} = \sigma(\sum_{j \in N(i) \cup \{i\}} \alpha_{ij} \cdot W \cdot h_j^{(l)})$
           where σ is an activation function (typically ELU)

   b. Optionally, use multi-head attention:
      i. Compute K different attention mechanisms in parallel
      ii. For hidden layers, concatenate the K attention heads:
          $h_i^{(l+1)} = ||_{k=1}^{K} \sigma(\sum_{j \in N(i) \cup \{i\}} \alpha_{ij}^k \cdot W^k \cdot h_j^{(l)})$
      iii. For the output layer, average the K attention heads:
           $h_i^{(L)} = \sigma(1/K \cdot \sum_{k=1}^{K} \sum_{j \in N(i) \cup \{i\}} \alpha_{ij}^k \cdot W^k \cdot h_j^{(L-1)})$

3. Return $Z = H^{(L)}$
```



1. Initialize node representations:

$$H^{(0)} = X$$

We start with the original node features as our initial node representations.

2. For each layer l from 1 to L:

a. For each node i ∈ V:

i. Compute attention coefficient:

```
e_ij = LeakyReLU(a^T · [W · h_i || W · h_j])
```

- We apply a linear transformation `W` to both the node `i` and its neighbor `j`.
- We concatenate these transformed features (`||` represents concatenation).
- We then compute a dot product with an attention vector `a`.
- Finally, we apply LeakyReLU activation to introduce non-linearity.
- This gives us an unnormalized attention coefficient `e_ij`, which tells us how much node `i` should pay attention to node `j`.

ii. Normalize attention coefficients:

```
α_ij = softmax_j(e_ij) = exp(e_ij) / Σ_k∈N(i)∪{i} exp(e_ik)
```

- We normalize the attention coefficients using softmax, so they sum to 1 across all neighbors.
- This ensures that the attention weights form a valid probability distribution.

iii. Compute new representation:

```
h_i^(l+1) = σ(Σ_j∈N(i)∪{i} α_ij · W · h_j^(l))
```

- We compute a weighted sum of the transformed neighbor features, where the weights are the attention coefficients.
- We apply an activation function `σ` (typically ELU or ReLU) to introduce non-linearity.

b. Multi-head attention:

```
h_i^(l+1) = ||_k=1^K σ(Σ_j∈N(i)∪{i} α_ij^k · W^k · h_j^(l))
```

- To stabilize learning and capture different aspects of the graph, we use multiple attention heads.
- Each head has its own set of parameters (W^k and a^k).
- For hidden layers, we concatenate the outputs from all heads.
- For the output layer, we typically average the outputs from all heads.

3. Return $Z = H^{(L)}$

The final output Z contains the learned node representations, which can be used for downstream tasks.

For References and Attribution, please visit the GitHub Repository:
https://github.com/aryashah2k/GNN_GCN_GAT