Algorithms and Data Structures with Applications in Machine Learning

Graph Representation Learning

January 1, 2025

Outline

Graph Terminology and Representation

Graph Representation Learning: DeepWalk and Node2Vec

Graph Neural Networks

Outline

Definition

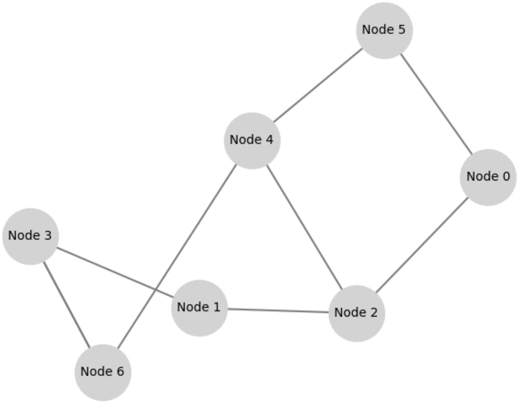
A graph is defined as:

$$G = (V, E, u)$$

- **Nodes (Vertices):** The set V represents the nodes in the graph.
- **Edges:** The set $E \subseteq V \times V$ represents the connections (relationships) between the nodes.
- **Features:** Each node can have a feature vector $u(v)$ representing its attributes.
- **Labels:** Nodes (or edges) can also have labels, which are used for tasks like classification.

Example Graph

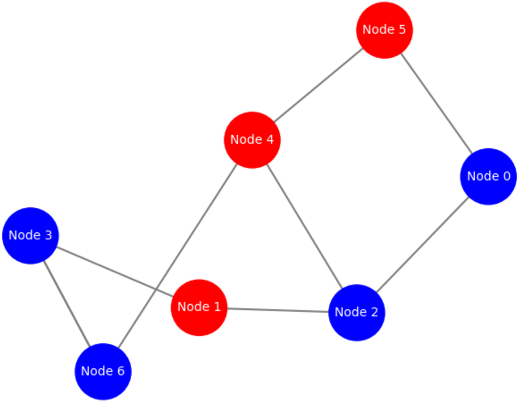
Example: The graph below has 7 connected nodes ($V = \{0, 1, 2, 3, 4, 5, 6\}$) and their edges ($E$).



Example Graph: Node Labels

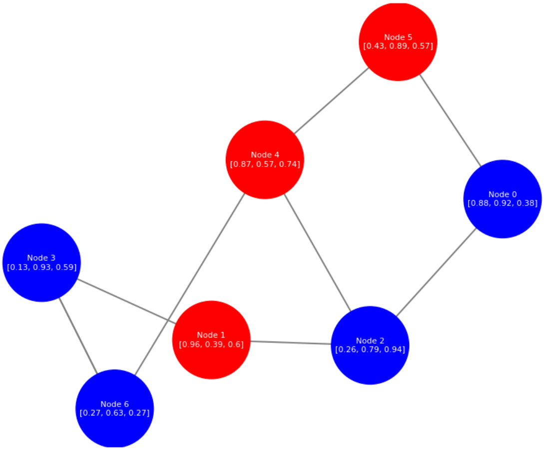
Example: Nodes in a graph can be associated with labels.

Blue nodes: Label 0 **Red nodes**: Label 1



Example Graph: Node Features

Example: Each node in the graph can have associated features. In this case: Each node has a feature vector of dimension 3.

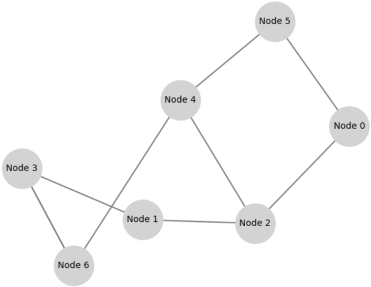


Definition

The adjacency matrix A of a graph $G = (V, E)$ is a matrix of size $|V| \times |V|$, where:

- $A[i][j] = 1$ if there is an edge between node i and node j.
- $A[i][j] = 0$ if there is no edge between node i and node j.

Example: A graph and its corresponding adjacency matrix:



Adjacency Matrix:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

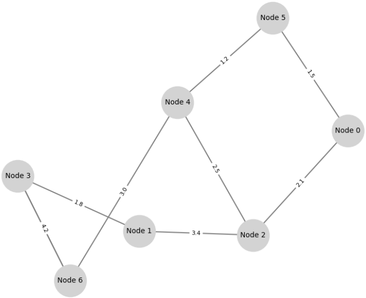
Weighted Adjacency Matrix

Definition

The adjacency matrix A can be extended to a weighted matrix W, where:

- $W[i][j]$ represents the weight of the edge between node i and node j.

Example: A graph and its a weighted adjacency matrix:



Weighted Matrix:

$$W = \begin{bmatrix} 0 & 0 & 2.1 & 0 & 0 & 1.5 & 0 \\ 0 & 0 & 3.4 & 1.8 & 0 & 0 & 0 \\ 2.1 & 3.4 & 0 & 0 & 2.5 & 0 & 0 \\ 0 & 1.8 & 0 & 0 & 0 & 0 & 4.2 \\ 0 & 0 & 2.5 & 0 & 0 & 1.2 & 3.0 \\ 1.5 & 0 & 0 & 0 & 1.2 & 0 & 0 \\ 0 & 0 & 0 & 4.2 & 3.0 & 0 & 0 \end{bmatrix}$$

Applications of Machine Learning on Graphs

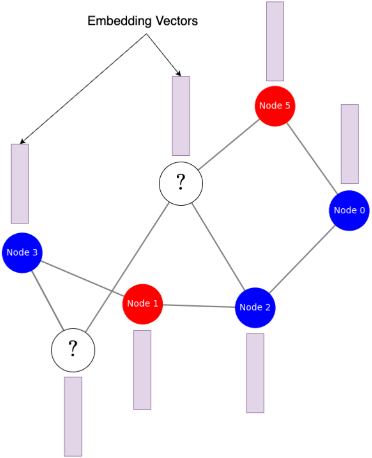
Applications: Machine Learning on graphs enables a variety of tasks, including:

- **Node Prediction:** Predict properties or labels of nodes in a graph (e.g., user classification in social networks).
- **Link Prediction:** Predict the existence or strength of a connection between two nodes (e.g., recommendation systems).
- **Graph Classification:** Assign labels to entire graphs (e.g., chemical compound classification).
- **Clustering:** Group nodes into communities or clusters based on their properties or structure.

Objective: Node Classification

Objective: The objective of this course is two-fold:

1. **Learning a D-dimensional representation:** Create embedding vectors for nodes that capture the structure of the graph.
2. **Node Classification:** Use the learned embeddings to predict the labels of the nodes.



Outline

Graph Structure-Based Embeddings: Introduction

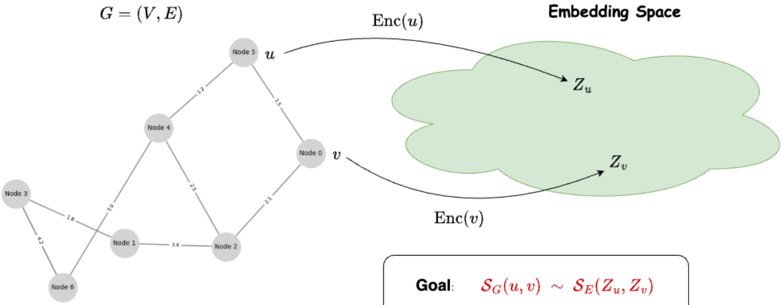
Objective: We aim to learn a mapping:

$$f : V \to \mathbb{R}^D$$

where each node $u \in V$ is mapped to a D-dimensional vector $\mathbf{Z}_u \in \mathbb{R}^D$.

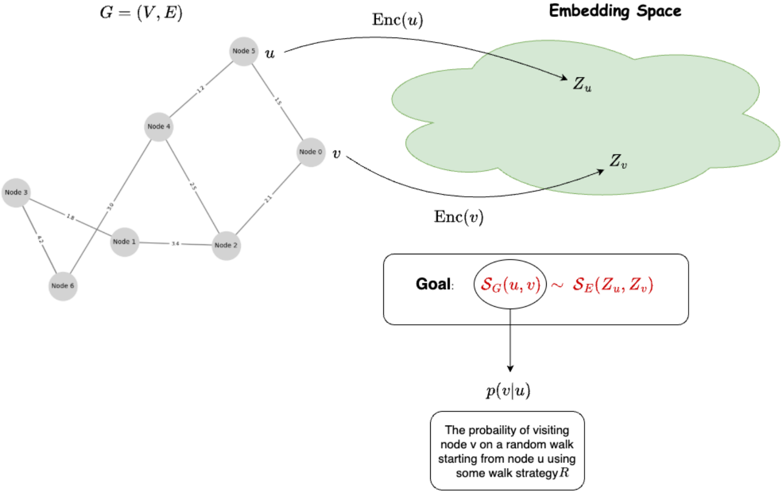
- In this section, we focus on leveraging the graph's **structure** to generate embedding vectors for nodes.
- The embeddings can be used for downstream tasks, such as node classification or link prediction.
- **No use of feature vectors:** We only use the graph topology (connections between nodes) to derive the embeddings.

Graph Structure-Based Embeddings: Objective

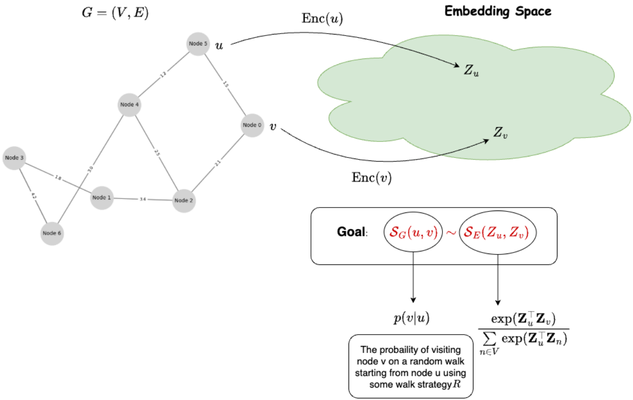


Goal: $\mathcal{S}_G(u, v) \sim \mathcal{S}_E(Z_u, Z_v)$

Graph Structure-Based Embeddings: Objective

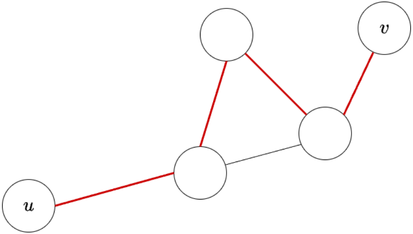


Graph Structure-Based Embeddings: Objective



Random Walks:

- A random walk is a sequence of steps through the graph, starting from a given node u, where each step randomly selects a neighboring node.
- The nodes visited during these walks represent the local neighborhood structure around u, denoted $\mathcal{N}_R(u)$
- Here is an example of a random walk from node u to node v.



Determining Neighbors Using Random Walks

Algorithm Fixed-Length Random Walks

Require: Graph $G = (V, E)$, starting node u, walk length L, number of walks N

Ensure: $\mathcal{N}_R(u)$ Multiset of nodes visited during random walks starting from u

Initialize an empty multiset of neighbors: neighbors $\leftarrow$ []
for $n = 1$ to N **do** ▷ Perform N random walks
    Initialize current_node $\leftarrow u$
    for $l = 1$ to L **do** ▷ Walk for L steps
        Sample a random neighbor $v \in$ Neighbors(current_node)
        neighbors.append(v)
        current_node $\leftarrow v$
    end for
end for
return neighbors

Introducing Node2Vec: Biased Random Walks

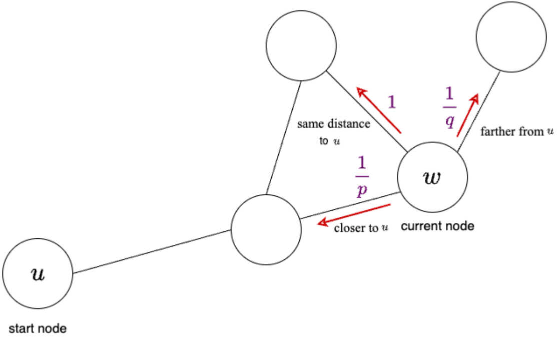
- The Node2Vec algorithm modifies traditional random walks by introducing **biases** that control how the walk explores the graph.
- This bias allows us to interpolate between two extremes:
  1. **Local Behavior:** Tendency to return to previously visited nodes, capturing local neighborhood structures. This is controlled by the **return hyperparameter** p.
  2. **Global Behavior:** Tendency to explore new, distant nodes, capturing the global structure of the graph. This is controlled by the **in-out hyperparameter** q.
- By adjusting p and q, Node2Vec generates embeddings that can reflect different graph traversal strategies.
- This flexibility makes Node2Vec suitable for capturing diverse graph structures. (See Programming Session 6).

Introducing Node2Vec: Biased Random Walks

- When the walk moves from node u to w, the neighbors of w are categorized based on their distance to u.
- We define the following **unnormalized probabilities**:
  1. Nodes closer to u than w receive an unnormalized probability of $\frac{1}{p}$.
  2. Nodes farther from u than w receive an unnormalized probability of $\frac{1}{q}$.
  3. Nodes at the same distance as w from u receive an unnormalized probability of 1.
- These unnormalized probabilities are normalized to form a valid probability distribution, which guides the biased random walk.

Here is an example of assigning the unnormalized probabilities:

- Starting at node u, the walk reaches node w.
- The probabilities assigned to w's neighbors depend on their distance to u, as described in the previous slide.



Determining Neighbors Using Biased Random Walks

Algorithm Biased Random Walks

Require: Graph $G = (V, E)$, starting node u, walk length L, number of walks N, return parameter p, in-out parameter q

Ensure: $\mathcal{N}_R(u)$: Multiset of nodes visited during biased random walks starting from u

Initialize an empty multiset of neighbors: neighbors $\leftarrow$ []
for $n = 1$ to N **do** ▷ Perform N biased random walks
    Initialize current_node $\leftarrow u$ and prev_node $\leftarrow$ None
    for $l = 1$ to L **do** ▷ Walk for L steps
        Compute probabilities using prev_node and current_node
        Sample the next node v based on the these probabilities
        neighbors.append(v)
        Update prev_node and current_node
    end for
end for
return neighbors

Defining the Loss Function:

- Now that we know how to define $\mathcal{N}_R(u)$, we can derive the loss function to train the embeddings.
- The objective is to minimize the following loss function:

$$\mathcal{L}(\theta) = -\sum_{u \in V} \sum_{v \in \mathcal{N}_R(u)} \log \left(\frac{\exp(\mathbf{Z}_u^\top \mathbf{Z}_v)}{\sum_{n \in V} \exp(\mathbf{Z}_u^\top \mathbf{Z}_n)} \right)$$

Where:

- $\mathbf{Z}_i \in \mathbb{R}^D$ is the embedding vectors for nodes $i \in V$.
- $\theta = \{\mathbf{Z}_i \mid i \in V\}$ represents all the embedding parameters to be learned.

Outline

From Node2Vec to Feature-Aware Embeddings

Node2vec recap:

- Node2Vec generates embeddings by combining graph topology and biased random walks.
- Focuses solely on the graph structure, without leveraging node-specific feature vectors.

Paradigm Shift:

- Our new objective is to incorporate both **graph structure** and **node features** into the embeddings.
- Instead of manually defining the impact of neighbors (e.g., via p and q), we aim for the model to **learn** the importance of different neighbors.

Notations:

- $\mathbf{h}_v^{(k)}$: Learned embedding of node v at iteration k.
- $\mathcal{N}(v)$: Set of neighbors of node v.

At each iteration, embeddings are refined by aggregating information from the local neighborhood and updating the node's representation.

Steps for One Iteration (k):

1. **Aggregation:** Gather information from neighbors of node v:

$$\mathbf{a}_v^{(k)} = f_{\text{aggregate}}\left(\{\mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}(v)\}\right)$$

2. **Update:** Combine aggregated information and the previous embedding to compute the new embedding:

$$\mathbf{h}_v^{(k)} = f_{\text{update}}(\mathbf{a}_v^{(k)}, \mathbf{h}_v^{(k-1)})$$

Algorithm Message Passing Framework

Require: Graph $G = (V, E)$, node features $\{\mathbf{x}_v \mid v \in V\}$, number of iterations K, $f_{\text{aggregate}}$, f_{update}

Ensure: Final node embeddings $\{\mathbf{h}_v^{(K)} \mid v \in V\}$

1: Initialize embeddings: $\mathbf{h}_v^{(0)} \leftarrow \mathbf{x}_v$ for all $v \in V$

2: **for** $k = 1$ to K **do**

3: **for** each node $v \in V$ **do**

$$\mathbf{a}_v^{(k)} \leftarrow f_{\text{aggregate}}\left(\{\mathbf{h}_u^{(k-1)} \mid u \in \mathcal{N}(v)\}\right)$$

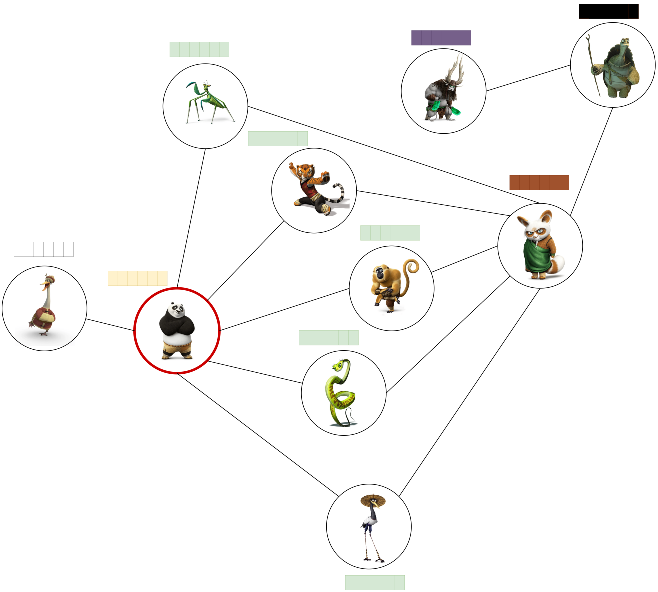
$$\mathbf{h}_v^{(k)} \leftarrow f_{\text{update}}(\mathbf{a}_v^{(k)}, \mathbf{h}_v^{(k-1)})$$

4: **end for**

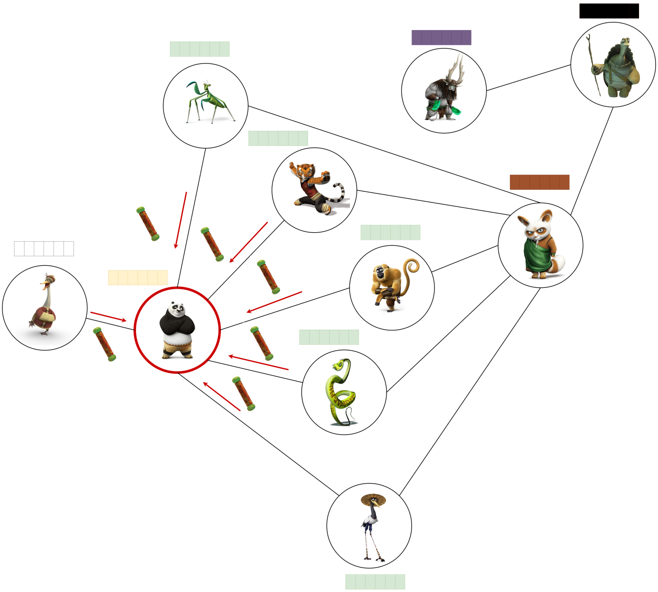
5: **end for**

6: **return** $\{\mathbf{h}_v^{(K)} \mid v \in V\}$

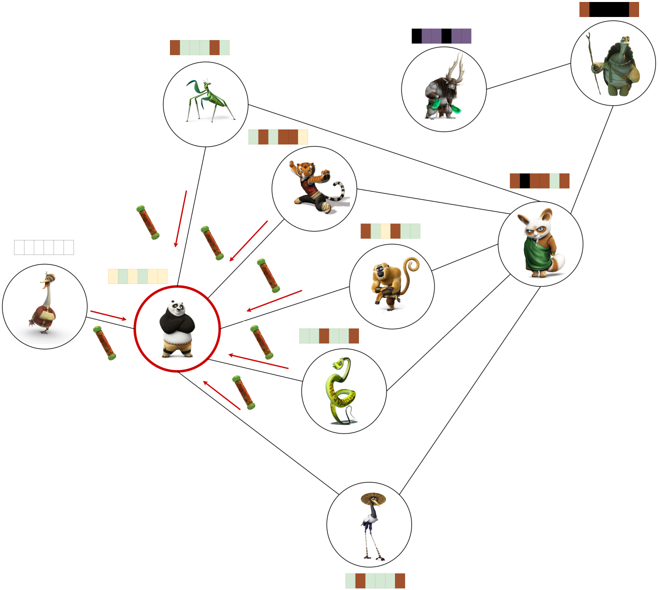
An Example: Graph Initialization with Feature Vectors



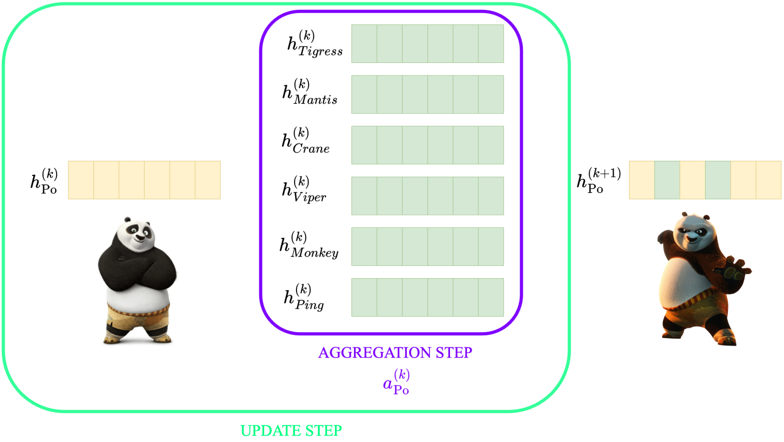
An Example: Aggregation Step



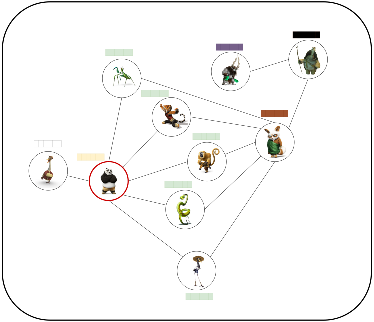
An Example: Update Step



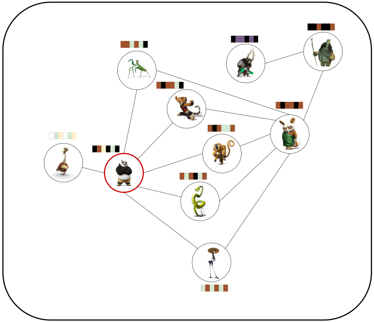
An Example: Recap of Both Steps



An Example: Final Embedding Vectors



...



Aggregation Function ($f_{\text{aggregate}}$):

$$\mathbf{a}_v^{(k)} = \frac{1}{\deg(v)} \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(k-1)}$$

Update Function (f_{update}):

$$\mathbf{h}_v^{(k)} = \sigma\left(W^{(k)} \cdot \left[\mathbf{h}_v^{(k-1)} \,\|\, \mathbf{a}_v^{(k)}\right]\right)$$

Description:

- Aggregates the mean of the neighbors' embeddings.
- Updates the embedding with a learned linear transformation using weights $W^{(k)}$ and a non-linear activation σ (e.g., ReLU).

Aggregation Function ($f_{\text{aggregate}}$):

$$\mathbf{a}_v^{(k)} = \sum_{u \in \mathcal{N}(v) \cup \{v\}} \frac{\mathbf{h}_u^{(k-1)}}{\sqrt{\deg(v) \cdot \deg(u)}}$$

Update Function (f_{update}):

$$\mathbf{h}_v^{(k)} = \sigma\left(W^{(k)} \cdot \mathbf{a}_v^{(k)}\right)$$

Description:

- **Aggregation:** Aggregates information from neighbors and the node itself, normalized by the degree of both nodes.
- **Update:** Applies a linear transformation using $W^{(k)}$, followed by a non-linear activation σ (e.g., ReLU).

Aggregation Function ($f_{\text{aggregate}}$):

$$\mathbf{a}_v^{(k)} = \sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{vu} \mathbf{h}_u^{(k-1)}$$

Where:

$$\alpha_{vu} = \frac{\exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top \left[\mathbf{h}_v^{(k-1)} \| \mathbf{h}_u^{(k-1)}\right]\right)\right)}{\sum_{w \in \mathcal{N}(v) \cup \{v\}} \exp\left(\text{LeakyReLU}\left(\mathbf{a}^\top \left[\mathbf{h}_v^{(k-1)} \| \mathbf{h}_w^{(k-1)}\right]\right)\right)}$$

Description:

- **Aggregation:** Computes a weighted sum of neighbor embeddings using attention coefficients α_{vu}.
- **Attention Coefficients** α_{vu}: Learn to assign importance to each neighbor dynamically.

Update Function (f_{update}):

$$\mathbf{h}_v^{(k)} = \Big\Vert_{k=1}^{K} \sigma\left(W_k^{(k)} \mathbf{a}_v^{(k)}\right)$$

Description:

- **Multi-Head Attention:** Combines results from K independent attention heads by concatenation ($\Vert$).
- **Non-Linearity:** Applies a learned linear transformation $W_k^{(k)}$ followed by a non-linear activation σ (e.g., ReLU).
- GATs allow each node to focus on the most relevant neighbors dynamically, enabling better representation learning for tasks such as node classification or graph-level predictions.

Unsupervised Training

Objective: Train node embeddings $\mathbf{h}_v^{(K)}$ by leveraging the graph structure, without requiring labels.

The Loss Function:

$$\mathcal{L}(\theta) = -\sum_{u \in V} \sum_{v \in \mathcal{N}_R(u)} \log \left(\frac{\exp(\mathbf{h}_u^{(K)^\top} \mathbf{h}_v^{(K)})}{\sum_{n \in V} \exp(\mathbf{h}_u^{(K)^\top} \mathbf{h}_n^{(K)})} \right)$$

Where:

- $\mathbf{h}_u^{(K)}$: Final embedding of node u after K message-passing layers.
- $\mathcal{N}_R(u)$: Neighborhood of u defined using some random walk strategy.
- We usually approximate the denominator using **negative sampling**.

Supervised Training: Node Classification

Objective: Predict the label of each node $v \in \mathcal{V}_{\text{train}}$ using the GNN-generated embeddings $\mathbf{h}_v^{(K)}$.

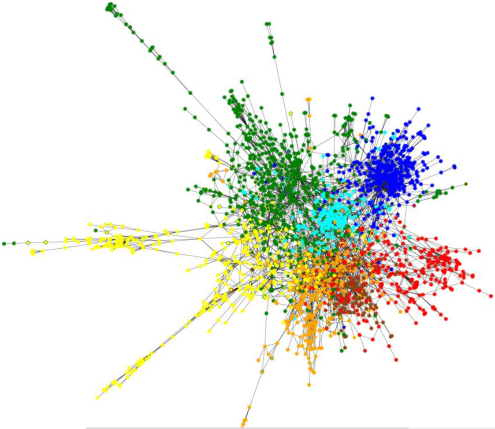
The Loss Function (Cross-Entropy):

$$\mathcal{L} = -\sum_{v \in \mathcal{V}_{\text{train}}} \sum_{c=1}^{C} y_v^c \log \hat{y}_v^c$$

Where:

- y_v^c: Ground-truth label (one-hot encoded) for node v.
- $\hat{y}_v^c = \text{softmax}\left(W_{\text{out}} \mathbf{h}_v^{(K)}\right)$: Predicted probability of class c, computed from the node embedding.

- During the programming session, we will work on the **Cora dataset**.
- The objective will be to build and train a **Graph Neural Network (GNN)** for node classification.



Thank you for your attention