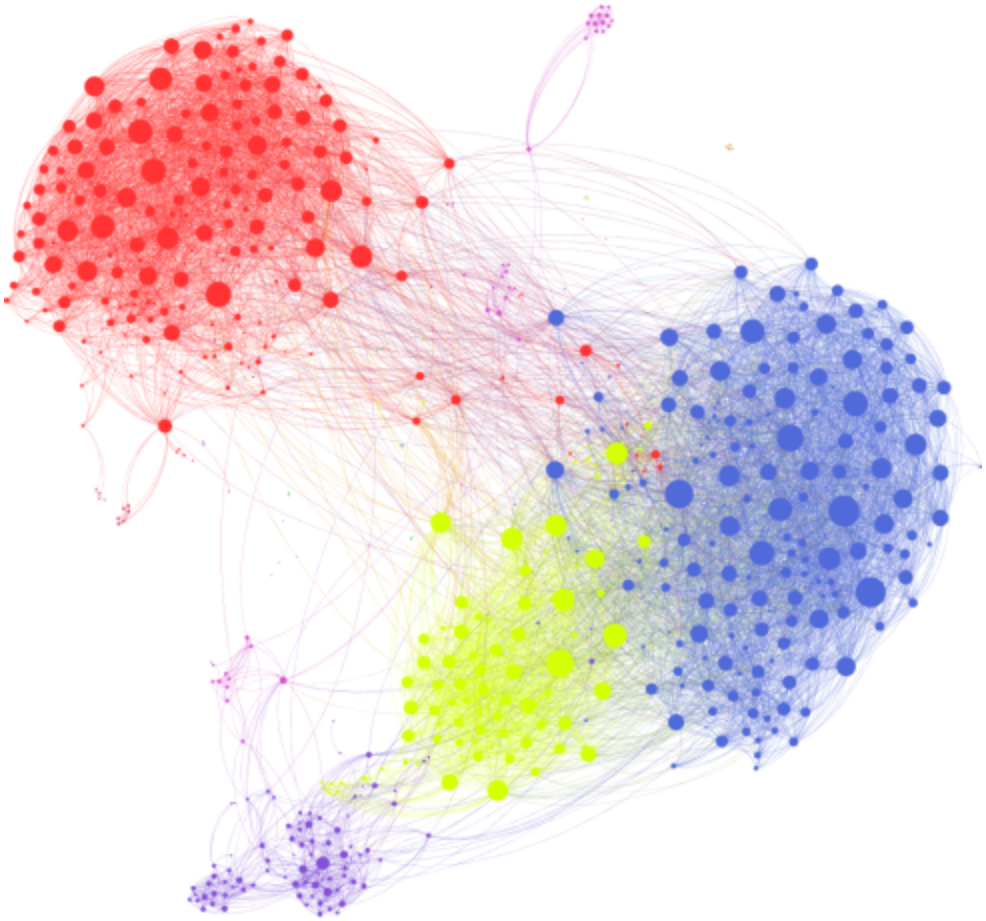


Graph Neural Networks

Author: Aishwarya Naresh Reganti

Index

1. Introduction to Graph Theory

A graph is a mathematical structure that consists of a set of nodes (also known as vertices) and a set of edges that connect those nodes. Graphs can be used to represent a wide variety of systems, including social networks, transportation networks, and molecular structures.

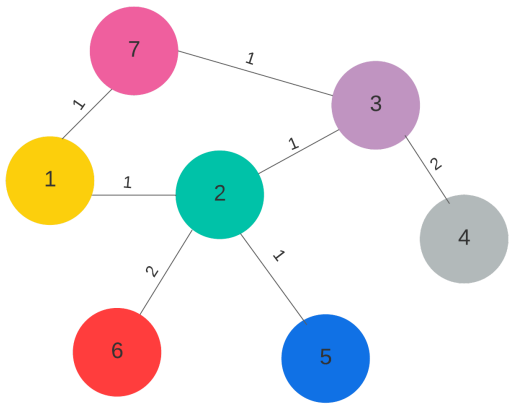
A set of edges and vertices form the fundamental components of a graph **G = (V, E)**

Some basic graph notations you should know:

1. **Graph**: A graph G is a pair (V, E) where V is a set of vertices (or nodes) and E is a set of edges that connect pairs of vertices.
2. **Vertex** (also called a node or point): A fundamental unit of a graph, typically represented as a point or a circle. Vertices are used to represent entities or objects in the system being modeled by the graph.
3. **Edge** (also called an arc or link): A connection between two vertices in a graph, typically represented as a line or an arrow. Edges are used to represent relationships or interactions between the entities or objects in the system being modeled by the graph.
4. **Weight**: A numerical value assigned to an edge, typically used to represent the strength or intensity of the relationship between the vertices it connects. Weighted graphs are graphs where edges have associated weights.
5. **Weighted graph**: A weighted graph is a graph in which each edge has a weight (or cost) associated with it.
6. **Undirected graph**: A graph in which each edge is bidirectional. In other words, the edge connects two vertices, and there is no concept of a source or destination.
7. **Directed graph (also called a digraph)**: A graph in which each edge has a direction associated with it. In other words, an edge goes from one vertex (the source) to another vertex (the destination).
8. **Degree**: The degree of a vertex in a graph is the number of edges incident to that vertex. In a directed graph, the degree is split into the in-degree (the number of edges pointing to the vertex) and the out-degree (the number of edges pointing away from the vertex).
9. **Path**: A path in a graph is a sequence of vertices connected by edges. The length of a path is the number of edges that it contains.
10. **Cycle**: A cycle in a graph is a path that starts and ends at the same vertex.
11. **Node Neighborhood**: The neighborhood of a node refers to the set of nodes that are connected to that node by an existing path. The neighborhood of a node can be defined in several different ways, depending on the specific context of the graph.

For example, the first hop neighborhood of a given node ***x***, generally refers to all the nodes in the graph that are directly connected to ***x*** via an edge.

Let's explore the meaning of each of the terms mentioned above in the context of the given graph.



- **Graph** G = (V, E) where V = {1, 2, 3, 4, 5, 6, 7} and E = {(1,2), (1,7), (2,1), (2,6), (2,5), (2,3), (3,2).... and so on. Note that we have both edges (1,2) and (2,1) since this is an **undirected graph**, which means that, if (u,v) exists in the graph, (v,u) will also be present.
- The above graph is a **weighted graph** since edges have weights/costs associated with them. For instance, the cost of traversing from node 3 to node 4 is **2 units** and the cost for traversing from node 3 to node 2 is **1 unit**
- The degree of node 2 is **4** since 4 edges are being formed with node 2.
- Let's consider nodes 1 and 3, there are 2 possible **paths**- {(1, 7), (7,3)} and {(1, 2), (2, 3)}
- A **cycle** is formed by the nodes {1, 7, 3, 2} because starting from any of these nodes and traversing through the other nodes will always lead back to the starting node.
- Node 1's **first-hop neighborhood** comprises of nodes {2, 7}, as these nodes can be directly reached from node 1 via an edge. Node 1's **second-degree neighborhood** includes nodes {3, 5, 6}.

Types of Graphs

In the previous section, we covered several types of graphs, including weighted, undirected, and directed graphs. From a GNN perspective, it's also important to distinguish between homogeneous and heterogeneous graphs.

Homogeneous Graphs: A homogeneous graph is a graph where all nodes and edges are of the same type. In other words, all nodes represent the same type of entity, and all edges represent the same type of relationship. For example, a social network where all nodes represent people, and all edges represent friendship relationships, would be a homogeneous graph.

Heterogeneous Graphs: A heterogeneous graph is a graph where nodes and edges can represent different types of entities and relationships. For example, a bibliographic network where nodes represent papers, authors, and conferences, and edges represent relationships like authorship or citation, would be a heterogeneous graph. In this case, the graph contains different types of nodes (papers, authors, and conferences) and different types of edges (authorship and citation).

Graph Representations

Graphs can be represented in several ways, depending on the specific use case and the type of graph being modeled. The most common method of representing graphs is through adjacency lists and matrices.

Adjacency matrix: The adjacency matrix of a graph is a square matrix where the entry in row ***i*** and column ***j*** is 1 if there is an edge between vertices ***i*** and ***j***, and 0 otherwise. For weighted graphs, the entries can be replaced with the weights of the corresponding edges.

Adjacency list: The adjacency list of a graph is a list of lists where the ***i***th list contains the neighbors of vertex ***i***.

Please find examples here

In many graph processing libraries however, graphs are often stored in more space-efficient formats such as CSR, CSC, and COO. These formats are built using adjacency matrices, but are designed to be more space-efficient than traditional adjacency matrices, which can be quite large for large graphs with many nodes and edges.

1. **CSR (Compressed Sparse Row)**: In the CSR format, the matrix is stored as three arrays: ***row_ptr***, ***col_idx***, and ***values***. The ***row_ptr*** array contains the starting index of each row in the ***col_idx*** and values arrays. The ***col_idx*** array contains the column index of each non-zero element, and the values array contains the value of each non-zero element.

2. **CSC (Compressed Sparse Column)**: The CSC format is similar to CSR, but the roles of the rows and columns are switched. In the CSC format, the ***col_ptr*** array contains the starting index of each column in the ***row_idx*** and values arrays. The ***row_idx*** array contains the row index of each non-zero element, and the values array contains the value of each non-zero element.
3. **COO (Coordinate Format)**: In the COO format, each non-zero element is stored as a tuple (***row_idx, col_idx, value***). This format is simple and flexible, but it can be less space-efficient than the CSR and CSC formats for large sparse matrices.

2. Graph Representation Learning

In the previous section, we learned about representing graphs as matrices. However, when we want to predict properties of entities within the graph, we need to use these representations in a different way. Graph representation learning is the task of learning a low-dimensional vector representation (or embedding) for each node or edge in a graph. The goal is to capture the structural and relational information of the graph in a compact and meaningful way.

This task of learning low-dimensional vector representations for graph entities such as nodes, edges, or subgraphs is similar to word embeddings in natural language processing. In word embeddings, a one-hot representation of words in a vocabulary is used to represent a sentence (just like we use adjacency matrices for graphs), while a self-supervised model like word2vec or BERT is used to create embeddings.

In graph representation learning, we aim to create embeddings for graph entities that can capture the essential properties of the entities while reducing their dimensionality. The following diagram illustrates this goal of generating graph representations or embeddings.

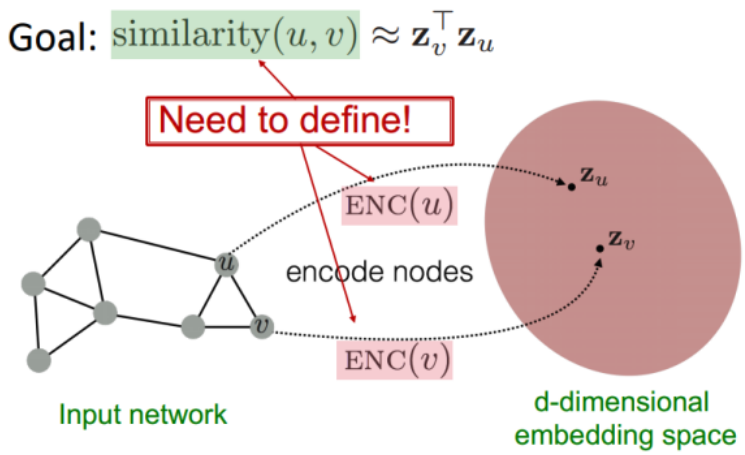


Image Source: CS224w Stanford course by Jure Leskovec

What are the objectives of generating embeddings?

1. To transform a graph into a vector space while retaining its structure and inherent properties.
2. To convert the graph's network structure into a numerical representation that can be used with traditional machine learning algorithms.
3. Typically, the embedding space has lower dimensions than the number of nodes in the original graph. The algorithm aims to preserve the initial structure of the original graph within the embedding space.

Which graph entities can be used for generating embeddings?

We can choose various entities such as nodes, edges, subgraphs, or even entire graphs to encode into embeddings, depending on the task at hand. For instance, consider a social network graph with users as nodes and edges between users who are friends. We can generate embeddings for the following purposes:

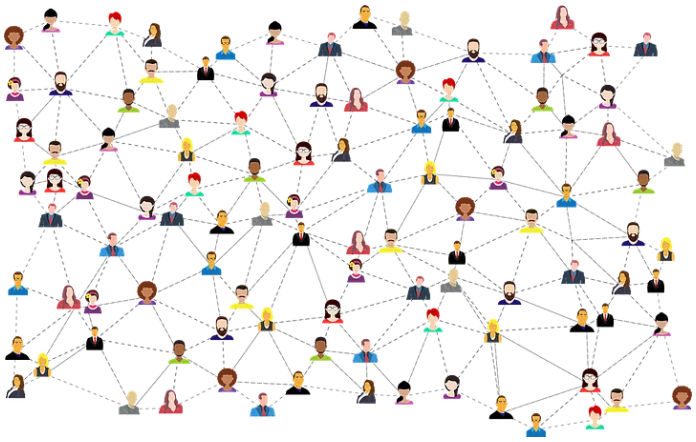


Image Source: https://towardsdatascience.com/an-introduction-to-graph-neural-network-gnn-for-analysing-structured-data-afce79f4cfdc

1. Embeddings of users to classify them as bots or real users (**Node Classification**)
2. Embeddings of groups of users (subgraphs) to detect anomalous groups that might pose potential threats or vulnerabilities (**Graph Classification**)
3. Embeddings of users to recommend new connections between users who are not already friends (**Link Prediction**)

These are just a few examples of how embeddings can be used, and there are many other options depending on the specific use case. The below figure gives a quick overview of different tasks that can be performed using graph representations.

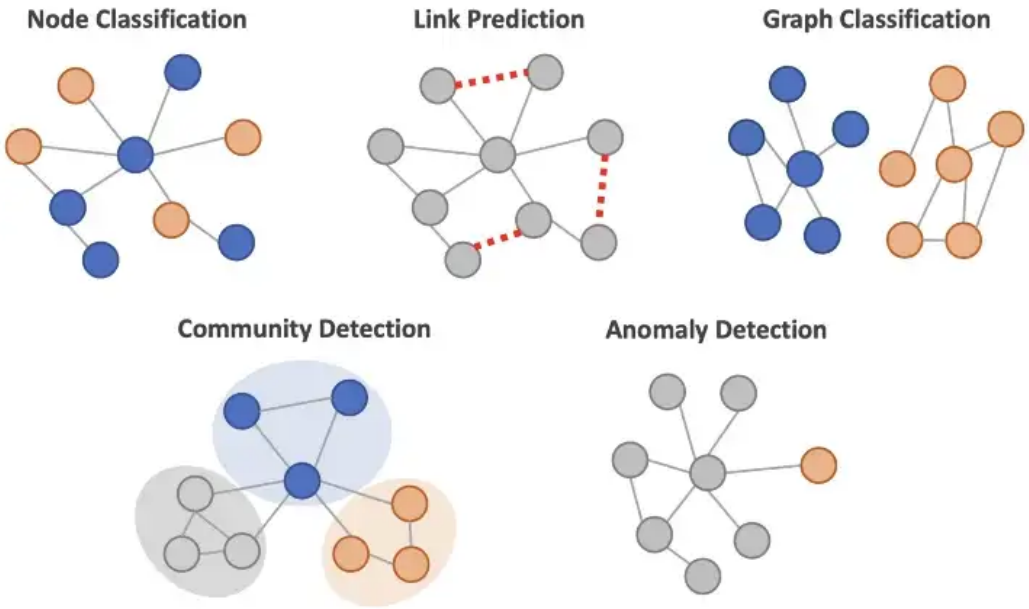


Image Source: https://towardsdatascience.com/an-introduction-to-graph-neural-network-gnn-for-analysing-structured-data-afce79f4cfdc

3. Graph Neural Networks (GNNs)

Considering the task at hand, how can we obtain representations of nodes (or users) to classify them as fraudulent or benign?

One simple method is to use standard deep neural networks that consume user features and construct a binary classification system. However, this approach would only consider the user feature information and would not incorporate information about the other users with whom the user is interacting or the network dynamics.

In contrast, graph models can leverage the graph topology, along with the node and edge features, to learn the underlying patterns and relationships that may indicate fraudulent behavior. By modeling the graph structure and the interactions between entities, graph models can better detect fraud in a more holistic and comprehensive manner. Additionally, graph models can be updated easily to adapt to new types of fraud that may emerge over time. This is one of the main reasons why graph models are so popular in fraud detection. In short, graph neural networks model both the node feature information and the node neighborhood information to come up with meaningful representations that can be used for downstream analysis.

How do we deal with graph data? Can you think of a deep learning algorithm that can ingest both neighborhood information and semantic information?

If you have a background in natural language processing, you might be familiar with LSTMs, which can capture both forward and backward contextual information.

LSTMs for graphs

- LSTMs (Long Short-Term Memory) cannot be directly applied to graphs because they are designed to operate on sequential data, where the temporal order of the data is important. In contrast, graphs are non-sequential data structures that lack a natural ordering of the nodes.
- In a graph, each node can have a different number of neighbors, and the order in which nodes are processed can significantly impact the resulting output. This makes it challenging to apply standard sequential models like LSTMs to graph data.
- Additionally, LSTMs require fixed-size input sequences, while graph structures can vary in size and shape. This means that it can be difficult to represent a graph as a fixed-size input sequence that can be processed by an LSTM.

If you have a background in computer vision, you may be familiar with the use of filters in CNNs to capture information from neighboring pixels and generate image embeddings.

CNNs for Graphs

- CNNs cannot be directly applied to graphs because they are designed to operate on grid-like structures such as images, where the spatial locality of the features is important. In contrast, graphs are non-Euclidean data structures that lack the regular grid-like structure of images.
- In a graph, each node can have a different number of neighbors and can be connected to any other node in the graph, which makes it difficult to apply standard convolutional operations.

Unlike text, where the data ordering is fixed, and images, which have a fixed ordering of pixels, graph structures lack a natural ordering. This distinguishes graph data from text and images and requires specialized models to effectively build representations.

GNNs are designed specifically for processing graph data and can learn to capture the graph structure and propagate information across the nodes. They can effectively handle non-Euclidean data structures and learn to handle variable-sized input data. GNNs can be used to generate node embeddings that capture both the features of the node and its local graph structure.

Having discussed the requirements of a model that can encode graphs, let's delve into the model that meets these requirements, namely GNN.

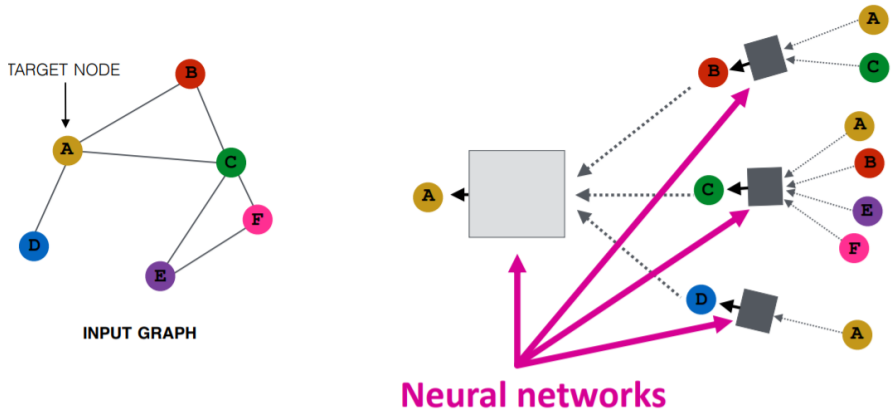


Image Source: CS224w Stanford course by Jure Leskovec

The diagram above provides a high-level overview of how a graph neural network operates. At each node, messages are received from its neighboring nodes and aggregated to compute the node's representation. The messages are generated using a neural network that takes the feature information as input.

A GNN typically consists of two components:

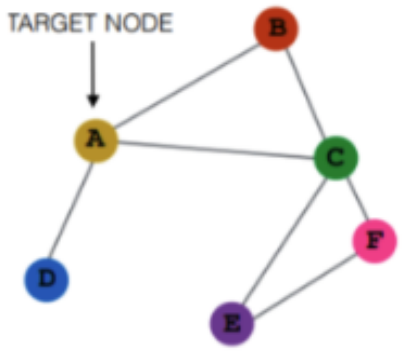
Message Function: The message function in a GNN defines how information is passed between nodes in the graph. It takes as input the features of the nodes and the edges and produces a message vector for each edge, which contains information about the features of the adjacent nodes.

Aggregation Function: The aggregation function, on the other hand, defines how the messages from the neighboring nodes are combined to compute the new representation of the node. It takes as input the messages from the neighboring nodes and produces a new representation vector for the node.

Let's run through an example for the above graph and learn how this works.

1. Consider the graph below, Let's build a simple **1 layer Graph Neural Network** with the following setup:
    a. The **message-passing function** essentially involves applying a feed-forward neural network to each node's feature set, which is then passed on to the target node.
    b. The **aggregation function** computes the mean of the messages received from all neighboring nodes.

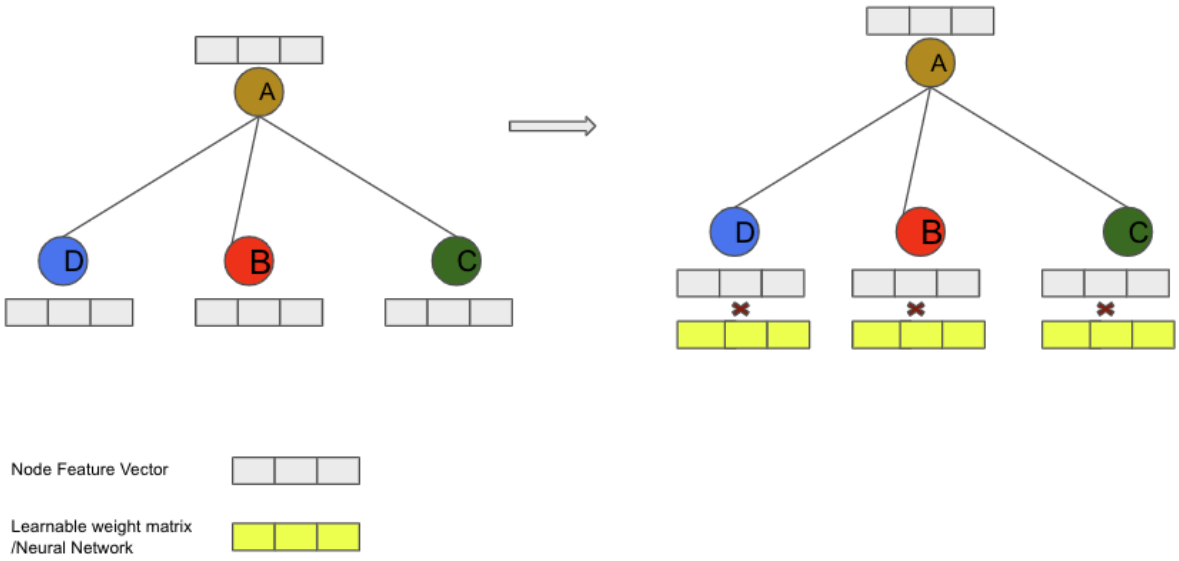
    Let's go through the process of computing the representation for a single node, **A**, in the graph step by step to gain a better understanding.



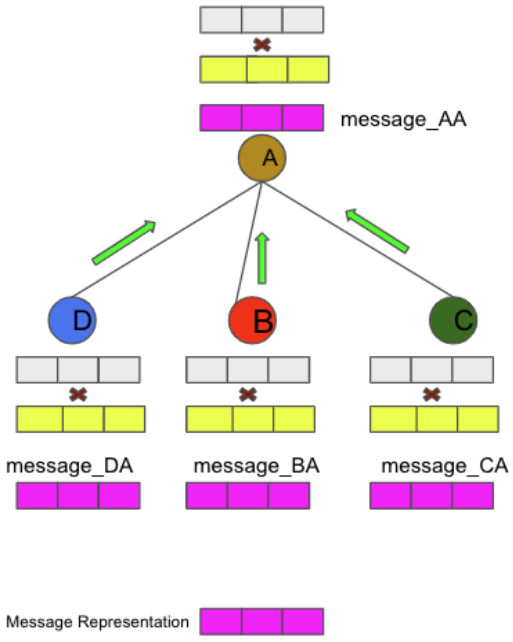
INPUT GRAPH

2. Node A has its own feature representation. In the next step, it requests messages from its neighboring nodes, which are B, C, and D. Keep in mind that this is a single-layer network in the context of GNNs. Unlike traditional deep learning models, a single-layer network in GNNs implies that we consider a single-hop neighborhood for a given node. So, how are messages formed? We simply use a learnable matrix or a feed-forward model to process messages from neighboring nodes. The sub-graph structure depicted in the image on the right is referred to as

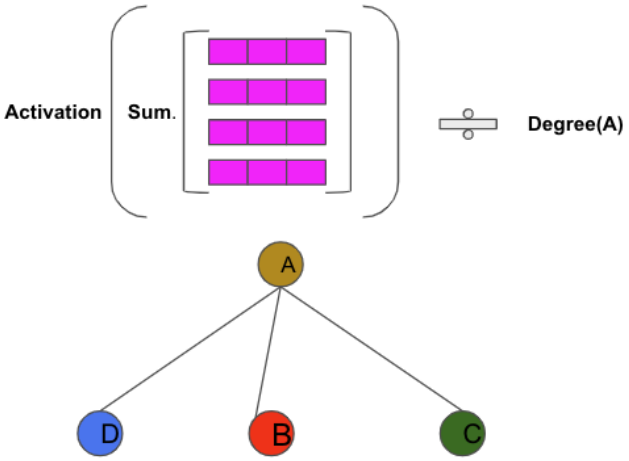
the **Message Flow Graph** for A. This structure determines the flow of message reception for node A.



3. After calculating the message representation at each neighboring node, it is passed to node A for aggregation. Node A obtains its final representation by combining the representations of its neighbors and its own message (message_AA) using an aggregation function.



4. In the end, A's representation is updated using the aggregation function, which computes the mean of the messages according to our setup. Next, a non-linear activation function is applied. After this step, we use a task-dependent layer to calculate the loss function and obtain gradients for each data point. The remaining process is similar to the execution of any standard deep learning network, we repeat the process for other nodes and compute the loss function to learn weights of the neural network.



And that's it! We've successfully constructed what is known as a Graph Convolutional Neural Network!

To expand the above method for a 2-layer GCN, our **Message Flow Graph** for node A would appear as shown below. The second-hop neighborhood sends messages to A's first-hop neighborhood recursively, using the same process.

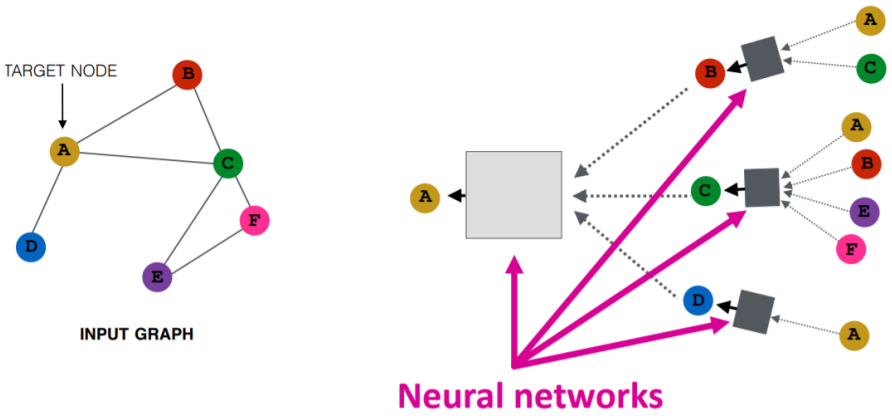


Image Source: http://snap.stanford.edu/class/cs224w-2020/slides/06-GNN1.pdf

Neighbor Sampling and Message Flow Graph

We used the term Message Flow Graph in the illustration above, If you're still a bit confused on the meaning:

A **Message Flow Graph** is a visualization of the flow of messages between nodes in a GNN during the message-passing process. It illustrates how nodes receive messages from their neighbors and how these messages are aggregated to update the node representations. The Message Flow Graph helps in understanding the local and global structure of the graph and how information propagates through the network. In multilayer GNNs, the Message Flow Graph shows how the neighborhood expands with each layer, depicting the flow of messages across multiple hops.

In the illustration above, each target node receives messages from all its neighbors. However, in real-world graphs, some nodes may have an excessive number of neighbors, such as a celebrity node in a social network. This makes the message-passing process computationally expensive. By sampling a subset of neighbors, GNNs can be trained more efficiently without sacrificing too much accuracy. This method enables GNNs to scale to large graphs, as the computational complexity now depends on the fixed number of sampled neighbors rather than the total number of neighbors. GNNs generally have a parameter called fan-out, which essentially refers to the number of neighbor nodes that should be sampled to compute messages for each node. Various methods can be used to identify the nodes that must be sampled. In the assignment, we'll use a DGL-based sampler, which you can learn more about as you proceed.

4. Basic GNN Architectures

In the previous section, we learned that all Graph Neural Networks (GNNs) have message-passing and aggregation functions at their core. These functions enable GNNs to learn and propagate information throughout the graph, allowing them to make predictions or decisions based on both local and global graph properties.

The **message-passing** function computes messages (or transformed features) for each node in the graph, considering its neighbors' features and potentially edge features. This step is essential for capturing the **local structure** of the graph and understanding the relationships between nodes.

The aggregation function combines the messages from neighboring nodes for each node in the graph. This process is crucial for consolidating local information and learning higher-level representations that encompass the **global properties** of the graph.

The example we examined earlier featured a Graph Convolutional Network (GCN) that employs a normalized MLP-based message function and a sum aggregation function. Indeed, you've just grasped the workings of a GCN!

While these two functions are fundamental components of all GNNs, different GNN architectures employ various message-passing and aggregation functions, tailored to specific applications or problem domains. Now that we have explored the basic structure of a GNN, let's delve into how GNNs differ in terms of their message and aggregation functions. The following table will provide details on various GNN architectures and how they utilize distinct message and aggregation functions to achieve their specific goals.

No.	GNN Type	Message Passing Function	Aggregation Function
1	Graph Convolutional Networks (GCNs)	The message from a node to its neighbors is the node's own feature vector.	Sum or mean of the messages from neighboring nodes
2	GraphSAGE (Graph Sample and Aggregation)	Concatenation of the node's feature vector with its neighbors' feature vectors	Element-wise mean, max, or LSTM-based pooling of the concatenated messages
3	Graph Attention Networks (GATs)	LeakyReLU-based dot product attention mechanism that weighs the importance of neighboring nodes	Weighted sum of the messages from neighboring nodes, with attention scores as weights
4	Graph Isomorphism Networks (GINs)	Linear transformation of the neighboring nodes' feature vectors, followed by an element-wise sum with the node's own feature vector	Sum of the messages from neighboring nodes