COMP 642 — Machine Learning Feb 28, 2023

Lecture 14: Graph Neural Networks

Lecturer: Anshumali Shrivastava Scribe By: Tianqi Xia, Jiashuo Ge, Dongwei Li, Danyu Wang

1 Graph

1.1 Definition

A graph is a data structure that consists of the following two components:

1. A finite set of vertices also called as nodes.
2. A finite set of ordered pair of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.

Graphs are used to represent many real-life applications: Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node)[4]. Each node is a structure and contains information like person id, name, gender, and locale. Following is an example of an undirected graph with 5 vertices.

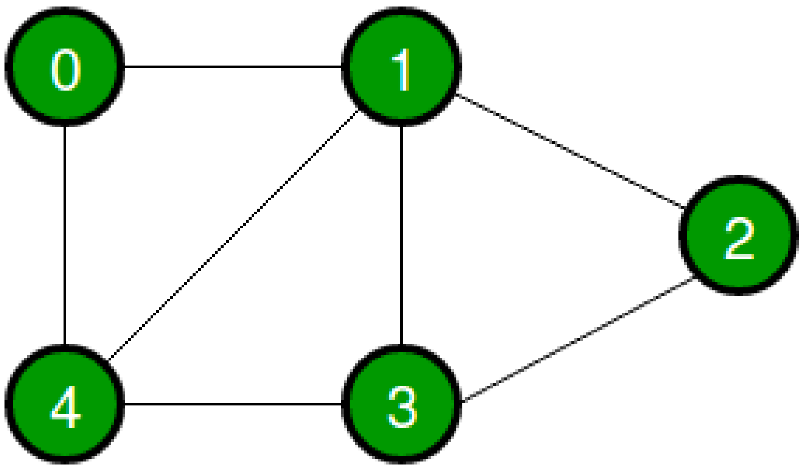


Figure 1: Graph

1.2 Representation

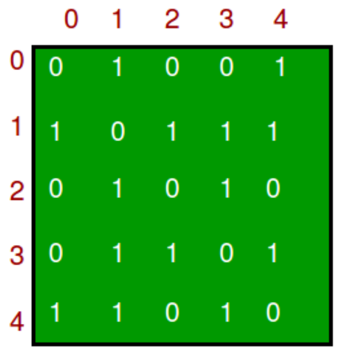
The following two are the most commonly used representations of a graph.

1. Adjacency Matrix
2. Adjacency List

There are other representations also like, Incidence Matrix and Incidence List. The choice of graph representation is situation-specific. It totally depends on the type of operations to be performed and ease of use.

1.2.1 Adjacency Matrix

Adjacency Matrix is a 2D array of size $V * V$ where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w. In case of an undirected graph, we need to show that there is an edge from vertex i to vertex j and vice versa. In code, we assign adj[i][j] = 1 and adj[j][i] = 1 In case of a directed graph, if there is an edge from vertex i to vertex j then we just assign adj[i][j]=1.
The adjacency matrix for the above example graph is:



	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Figure 2: Adjacency Matrix

Pros: Representation is easier to implement and follow. Removing an edge takes $O(1)$ time. Queries like whether there is an edge from vertex u to vertex v are efficient and can be done $O(1)$.

Cons: Consumes more space $O(V^2)$. Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is $O(V^2)$ time. Computing all neighbors of a vertex takes $O(V)$ time (Not efficient).

1.2.2 Adjacency List

An array of lists is used. The size of the array is equal to the number of vertices. Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the i^{th} vertex. This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs. Following is the adjacency list representation of the above graph.

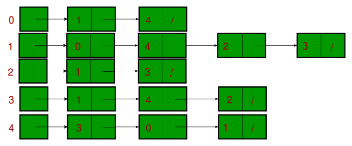


Figure 3: List Adjacency

Pros: Saves space $O(|V|+|E|)$. In the worst case, there can be C(V, 2) number of edges in a graph thus consuming $O(V^2)$ space. Adding a vertex is easier. Computing all neighbors of a vertex takes optimal time.

Cons: Queries like whether there is an edge from vertex u to vertex v are not efficient and can be done $O(V)$.

In Real-life problems, graphs are sparse($|E| << |V|^2$). That's why adjacency lists Data structure is commonly used for storing graphs. Adjacency matrix will enforce ($|V|^2$) bound on time complexity for such algorithms.

2 PageRank Algorithm

2.1 What is PageRank

The Internet can be regarded as a graph, where nodes and edges possess attributes. PageRank is an algorithm created by Google to rank web pages. It is a link analysis algorithm of graphs. It calculates the importance of the page and assigns different weights to each inbound. The more important the page provided by the link, the higher the inbound. The importance of the current page is determined by the importance of other pages. After combining all other factors such as the title and the keywords, Google uses PageRank to adjust the results, so that those pages with more importance can have higher ranking in the search results.

2.2 How PageRank works

For a certain Internet web page A, the calculation of PageRank of A is based on the following two basic assumptions:

1. Quantity assumption: In the Web graph model, if a page node receives more incoming links from other web pages, the page is more important.

2. Quality assumption: The quality of inbound links pointing to page A is different. High-quality pages will pass more weight to other pages through links. So the more high-quality pages point to page A, the more important page A is.

2.2.1 Simplified PageRank Algorithm

Let us take an example to illustrate how simplified PageRank calculates[5]. Suppose there are 4 web pages A, B, C, D in total. The link information between them is shown in the Figure 4.

The influence of a web page can be represented as the sum of the weighted influence of all inbound pages, expressed as

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

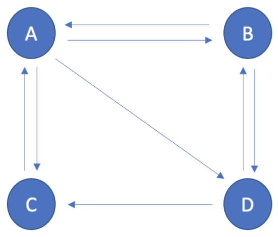


Figure 4: Example of Simplified PageRank

Here u is the page to be evaluated, and B_u is the set of incoming links of page u. For any page v in the inbound link set, the influence it can bring to u is its own influence $PR(v)$ divided by the number of outlinks of v. This means that the page v distributes the influence $PR(v)$ evenly. Then count all pages v that can bring links to u, and the sum is the influence of page u, which is $PR(u)$. In this example, A has three outgoing links to B, C, and D respectively. Then when the user visits A, the possibility of jumping to B, C or D is 1/3. B has two outgoing links, which are linked to A and D, and the jump probability is 1/2. In this way, we can get the transition matrix M of the four pages A, B, C, and D:

$$M = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix}$$

We assume that the initial influences of the four pages A, B, C, and D are the same.

$$w_0 = \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

After the first transfer, the influence of each page becomes w_1.

$$w_1 = Mw_0 = \begin{bmatrix} 0 & 1/2 & 1 & 0 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 0 & 0 & 1/2 \\ 1/3 & 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix} = \begin{bmatrix} 9/24 \\ 5/24 \\ 5/24 \\ 5/24 \end{bmatrix}$$

Then we calculate $w_2 = Mw_1$, until the influence of w_n does not change after the n^{th} iteration. Finally the weight can converge to $(0.3333, 0.2222, 0.2222, 0.2222)$. It corresponds to the influence of the four pages A, B, C, and D in the final balanced state. We can see that page A has more weights than other pages, which means the PR value is higher. Pages B, C, and D have equal PR values.

2.2.2 General PageRank Algorithm

In the calculation of simplified PageRank, there are two possible problems.

1. Rank Leak: As shown in Figure 5, if a page does not have a outbound link, it is like a black hole, absorbing the influence of other pages without releasing it. This will eventually cause the PR value of other pages to be 0.

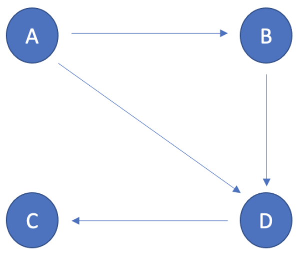


Figure 5: Example of Rank Leak

2. Rank Sink: As shown in Figure 13, if a page has only outbound links and no inbound links, the PR value of the page will become 0 as there is no V in the formula.

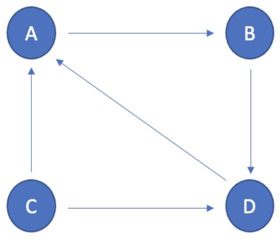


Figure 6: Example of Rank Sink

In order to solve problems of rank leak and rank sink in the simplified model, Larry Page proposed the random walking model of PageRank[1]. He assumed that users do not always visit pages by jumping links. There is also a possibility that no matter which page they are currently on, they may visit any other page. For example, the user directly enters the URL to visit other pages. So he defined a damping factor d, which represents the probability of users accessing the Internet by jumping the link. Usually, it can take a fixed value of 0.85, and $1-d$ means that the user does not access the web pages through the jump link, such as entering the URL directly. Then the formula of calculating PageRank becomes:

$$PR(u) = \frac{1-d}{N} + d \sum_{v \in B_u} \frac{PR(v)}{L(v)},$$

where N is the total number of web pages. Then we can iterate the weight calculation of web pages with the damping factor d. Through mathematical theorems, it can be proved that the final PageRank random walking model can converge. This means that we can obtain a stable and normal PR value.

3 Graph Neural Network

3.1 What is GNN?

A graph neural network (GNN) is a class of artificial neural networks for processing data that can be represented as graphs.

In the more general subject of ”geometric deep learning”, certain existing neural network architectures can be interpreted as GNNs operating on suitably defined graphs. Convolutional neural networks, in the context of computer vision, can be seen as a GNN applied to graphs structured as grids of pixels. Transformers, in the context of natural language processing, can be seen as GNNs applied to complete graphs whose nodes are words in a sentence[2].

The key design element of GNNs is the use of pairwise message passing, such that graph nodes iteratively update their representations by exchanging information with their neighbors. Since their inception, several different GNN architectures have been proposed, which implement different flavors of message passing, started by recursive or convolutional constructive approaches. As of 2022, whether it is possible to define GNN architectures ”going beyond” message passing, or if every GNN can be built on message passing over suitably defined graphs, is an open research question.

Relevant application domains for GNNs include social networks, citation networks, molecular biology, chemistry, physics and NP-hard combinatorial optimization problems.

3.2 The architecture of GNN

In graph theory, we implement the concept of Node Embedding. It means mapping nodes to a d- dimensional embedding space (low dimensional space rather than the actual dimension of the graph), so that similar nodes in the graph are embedded close to each other.

Our goal is to map nodes so that similarity in the embedding space approximates similarity in the network.

Let’s define u and v as two nodes in a graph.

x_u and xv are two feature vectors.

Now we’ll define the encoder function Enc(u) and Enc(v), which convert the feature vectors to z_u and z_v.

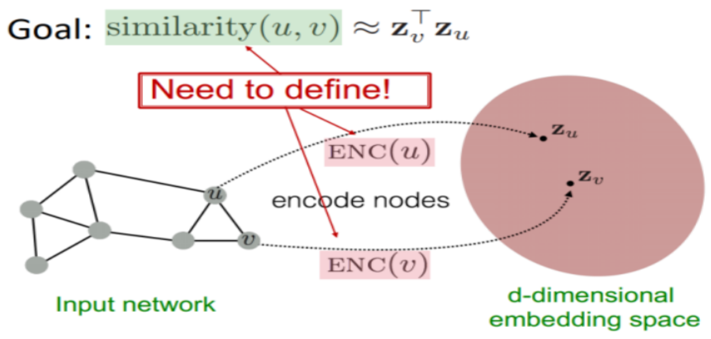


Figure 7: A simple exple

The encoder function should be able to perform :

Locality (local network neighborhoods) Aggregate information Stacking multiple layers (computation) Locality information can be achieved by using a computational graph. As shown in the graph below, i is the red node where we see how this node is connected to its neighbors and those neighbors’ neighbors. We’ll see all the possible connections, and form a computation graph.

By doing this, we’re capturing the structure, and also borrowing feature information at the same time.

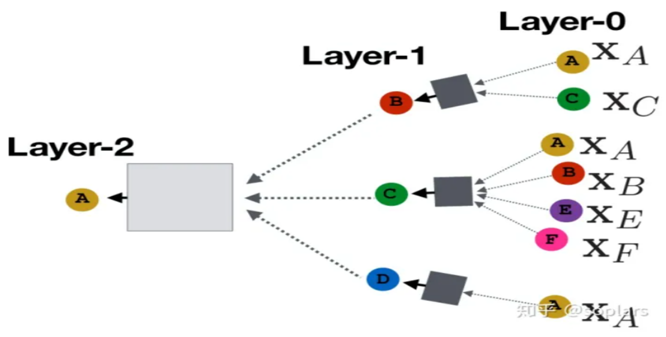


Figure 8: A simple exple

Once the locality information preserves the computational graph, we start aggregating. This is basically done using neural networks.

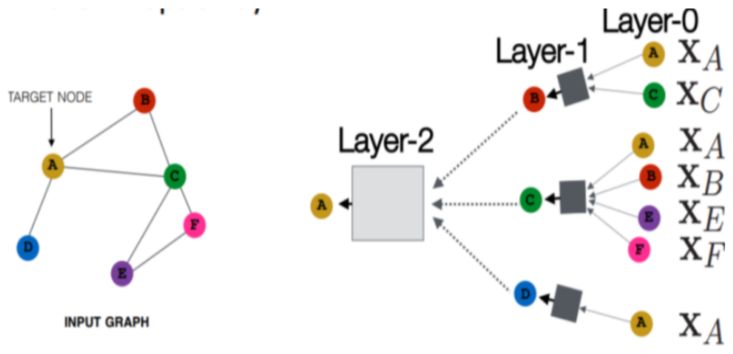


Figure 9: A simple exple

Neural Networks are presented in grey boxes. They require aggregations to be order-invariant, like sum, average, maximum, because they are permutation-invariant functions. This property enables the aggregations to be performed.

Training can be unsupervised or supervised[3]:

Unsupervised training: Use only the graph structure: similar nodes have similar embeddings. Unsupervised loss function can be a loss based on node proximity in the graph, or random walks. Supervised training: Train model for a supervised task like node classification, normal or anomalous node.

3.3 How GNN works

The basic idea behind GNNs is to recursively update the node features by aggregating information from their neighboring nodes. The process is typically performed through a series of message passing steps, where each node sends a message to its neighbors, which are then aggregated to produce an updated feature representation for each node.

The message passed from node i to node j is typically a function of the features of node i and the edge between i and j, and can be defined as:

$$m_{ij} = M(h_i, h_j, e_{ij}),$$

where h_i and h_j are the feature representations of node i and node j, respectively, and e_{ij} is the edge between them. M is a learnable function that computes the message to be sent.

Once the messages have been passed, they are aggregated by each node to update its own feature representation. The aggregation function can be defined as:

$$a_i = \text{AGGREGATE}(m_{ij}, \forall j \in \mathcal{N}(i)),$$

where $\mathcal{N}(i)$ is the set of neighbors of node i and AGGREGATE is a learnable function that aggregates the messages.

Finally, the updated feature representation of node i is obtained by combining the original feature representation with the aggregated messages:

$$h_i^{(t-1)} = \text{COMBINE}(h_i^{(t)}, a_i),$$

where COMBINE is a learnable function that combines the current feature representation $h_i^{(t)}$ with the aggregated messages a_i.

This process is typically repeated for several iterations, allowing each node to gather information from increasingly distant nodes in the graph. The final output of the GNN can be obtained by feeding the updated feature representations of all nodes through a fully connected layer or other appropriate output layer.

In the following section, we present a classic taxonomy of graph neural networks (GNNs). It categorize graph neural networks (GNNs) into recurrent graph neural networks (RecGNNs), convolutional graph neural networks (ConvGNNs), graph autoencoders (GAEs), and spatial-temporal graph neural networks (STGNNs). In the following, we give a brief introduction of how each category GNN works differently.

For the introduction later, we need to make some basic definitions in advance.

- **Definition 1 (Graph)**: A graph is represented as $G = (V, E)$ where V is the set of vertices or nodes (we will use nodes throughout the paper), and E is the set of edges. Let $v_i \in V$ to denote a node and $e_{ij} = (v_i, v_j) \in E$ to denote an edge pointing from v_j to v_i. The neighborhood of a node v is defined as $N(v) = \{u \in V | (v, u) \in E\}$.

- **Definition 2 (Spatial-Temporal Graph)**: A spatial-temporal graph is an attributed graph where the node attributes change dynamically over time. The spatial-temporal graph is defined as $G^{(t)} = (\mathbf{V}, \mathbf{E}, \mathbf{X}^{(t)})$ with $\mathbf{X}^{(t)} \in \mathbf{R}^{n \times d}$.

Recurrent graph neural networks (RecGNNs): RecGNNs are recognized as a groundbreaking contribution to the field of graph neural networks. These models employ recurrent neural architectures to acquire node representations by assuming that a node in a graph consistently communicates with its neighbors until a stable equilibrium is attained.

Convolutional graph neural networks (ConvGNNs): Different from RecGNNs, ConvGNNs stack multiple graph convolutional layers to extract high-level node representations. The main idea of Convolutional graph neural wetworks is to generate a node v's representation by aggregating its own features $\mathbf{X}_v$ and neighbors' features $\mathbf{X}_u$, where $u \in N(v)$.

Figure 10 shows a ConvGNN for node classification. The figure highlights that each node's hidden representation is encapsulated by a graph convolutional layer that aggregates feature information from its neighbors. Following the feature aggregation, a non-linear transformation is applied to the resulting outputs. By stacking multiple layers, the final hidden representation of each node receives messages from a broader neighborhood.

Besides, figure 11 demonstrates a ConvGNN with pooling and readout layers for graph classification. The ConvGNN begins with a graph convolutional layer followed by a pooling layer that coarsens the graph into sub-graphs. This helps in representing higher graph-level

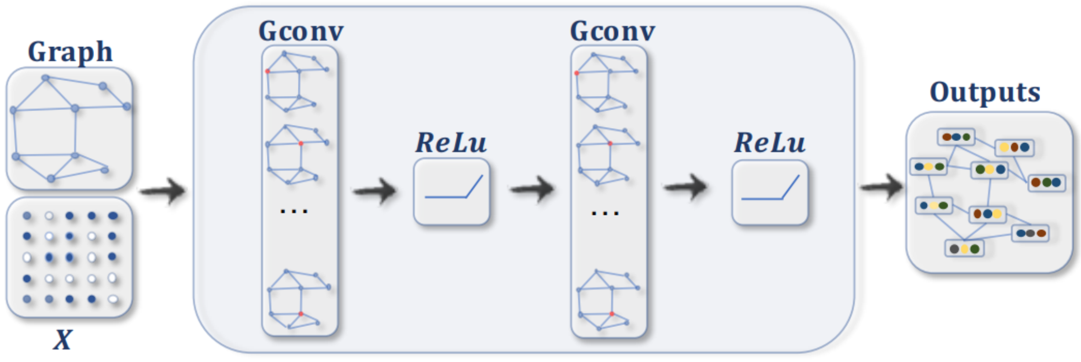


Figure 10: ConvGNN for node classification

representations through node representations on coarsened graphs. Finally, a readout layer summarizes the final graph representation by taking the sum/mean of hidden representations of sub-graphs.

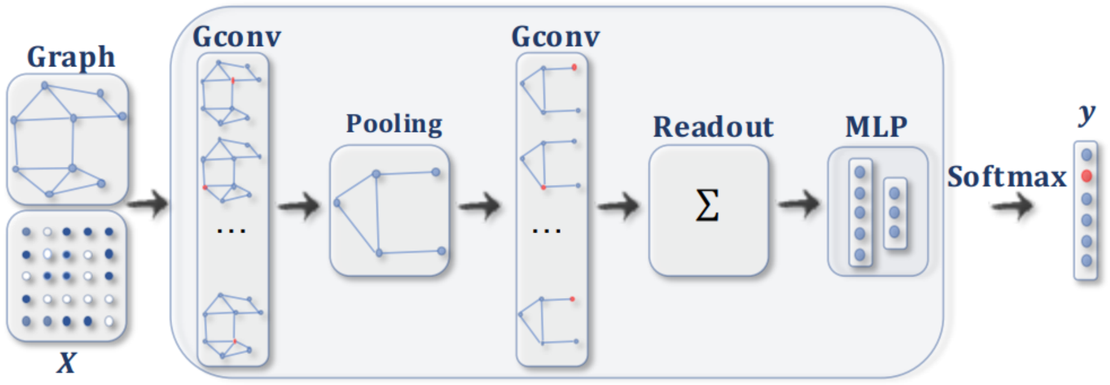


Figure 11: ConvGNN for graph classification

Graph autoencoders (GAEs): GAEs are machine learning frameworks that operate without the need for labeled data. They utilize a process where nodes and graphs are transformed into a latent vector space and then reconstructed based on the encoded information. In the context of network embedding, GAEs can be used to learn latent node representations by reconstructing graph structural information such as the graph adjacency matrix.

Figure 12 presents a GAE for network embedding. In this architecture, the encoder employs graph convolutional layers to generate a network embedding for each node. The decoder then uses these embeddings to compute pair-wise distances. After applying a non-linear activation function, the decoder reconstructs the graph adjacency matrix. The network is trained by minimizing the difference between the actual adjacency matrix and the reconstructed adjacency matrix.

Spatial-temporal graph neural networks (STGNNs): STGNNs are designed to extract latent patterns from spatial-temporal graphs. The core concept behind STGNNs is to consider both spatial and temporal dependencies concurrently. Current techniques typically incorporate graph convolutions to capture spatial dependencies and RNNs or CNNs to model temporal dependencies. In Figure 13, a STGNN architecture is presented for spatial-temporal

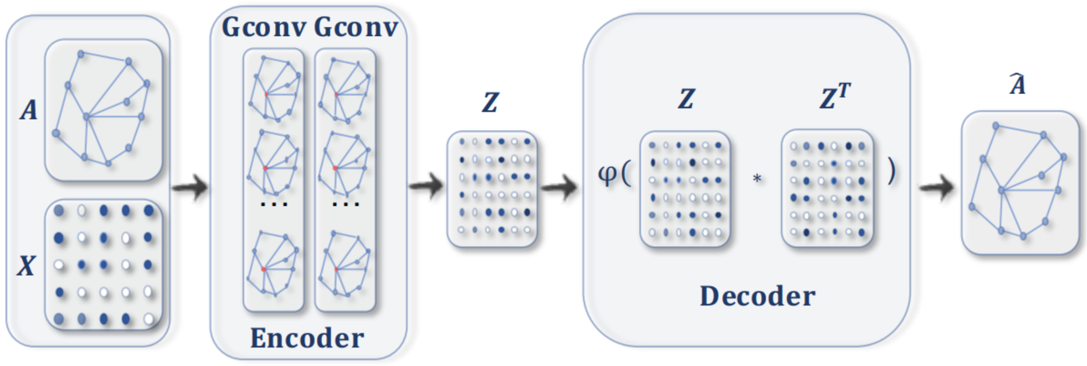


Figure 12: GAE for network embedding

graph prediction. The graph convolutional layer takes A and $X^{(t)}$ as input to capture spatial dependencies. The subsequent 1D-CNN layer operates on X along the time axis to capture temporal dependencies. The output layer generates a prediction for each node, such as its future value at the next time step, using a linear transformation.

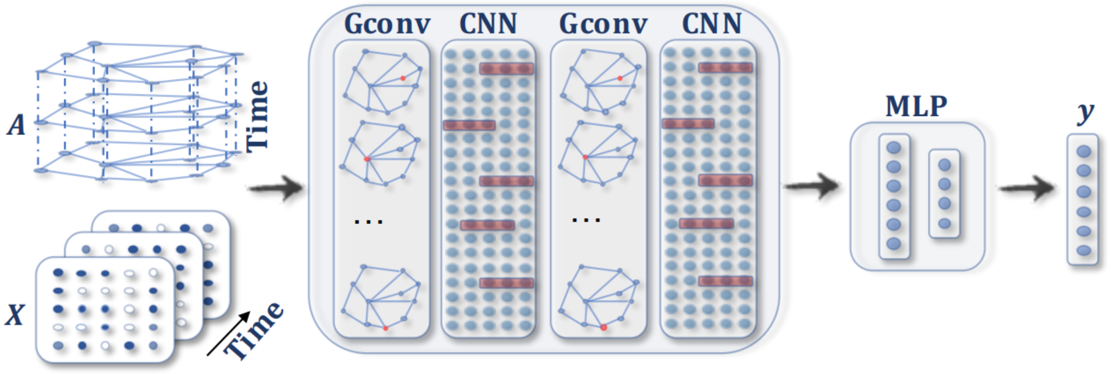


Figure 13: STGNN for spatial-temporal graph forecasting

Training Frameworks: Many GNNs (e.g., ConvGNNs) can be trained in a (semi-) supervised or purely unsupervised way within an end-to-end learning framework, depending on the learning tasks and label information available at hand.

- **Semi-supervised learning** for node-level classification. ConvGNNs can effectively identify class labels for unlabeled nodes in a network that contains both labeled and unlabeled nodes. To accomplish this, a robust model is learned using a single network. The process involves building an end-to-end framework by stacking multiple graph convolutional layers, followed by a softmax layer for multi-class classification.

- **Supervised learning** for graph-level classification. The goal of graph-level classification is to predict the label or labels that correspond to an entire graph. Achieving this objective through end-to-end learning requires a combination of graph convolutional layers, graph pooling layers, and readout layers. Graph convolutional layers extract high-level representations of individual nodes, whereas graph pooling layers downsample each graph by creating sub-structures. The role of readout layers is to combine node representations for each graph into a single graph representation. To create an end-to-end framework for graph classification, we can employ a multi-layer perceptron and a softmax layer on the graph representations.

- **Unsupervised learning** for graph embedding. If there are no class labels available for graphs, we can still learn their embeddings in an unsupervised manner using an end-to-end framework. These algorithms leverage edge-level information in two ways. One approach is to use an autoencoder framework where the encoder uses graph convolutional layers to embed the graph into a latent representation, which is then reconstructed by a decoder to reproduce the graph structure. Another common approach is to employ negative sampling, where a subset of node pairs are randomly selected as negative pairs, while the node pairs with existing links in the graph serve as positive pairs. A logistic regression layer is then applied to distinguish between positive and negative pairs.

3.4 Applications of GNN

The problems that GNN solve can be broadly classified into three categories:

- Node Classification
- Link Prediction
- Graph Classification

In node classification, the task is to predict the node embedding for every node in a graph. This type of problem is usually trained in a semi-supervised way, where only part of the graph is labeled. Typical applications for node classification include citation networks, Reddit posts, Youtube videos, and Facebook friends relationships.

In link prediction, the task is to understand the relationship between entities in graphs and predict if two entities have a connection in between. For example, a recommender system can be treated as link prediction problem where the model is given a set of users' reviews of different products, the task is to predict the users' preferences and tune the recommender system to push more relevant products according to users' interest.

In graph classification, the task is to classify the whole graph into different categories. It is similar to image classification but the target changes into graph domain. There is a wide range of industrial problems where graph classification can be applied, for example, in chemistry, biomedical, physics, where the model is given a molecular structure and asked to classify the target into meaningful categories. It accelerates the analysis of atom, molecule or any other structured data types.

References

[1] Lawrence Page et al. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[2] Benjamin Sanchez-Lengeling et al. "A gentle introduction to graph neural networks". In: *Distill* 6.9 (2021), e33.

[3] Franco Scarselli et al. "The graph neural network model". In: *IEEE transactions on neural networks* 20.1 (2008), pp. 61–80.

[4] Douglas Brent West et al. *Introduction to graph theory.* Vol. 2. Prentice hall Upper Saddle River, 2001.

[5] Wikipedia. *PageRank — Wikipedia, The Free Encyclopedia.* `http://en.wikipedia.org/w/index.php?title=PageRank&oldid=1141191830`. [Online; accessed 04-March-2023]. 2023.