



Lecture 1

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

What is a Graph?



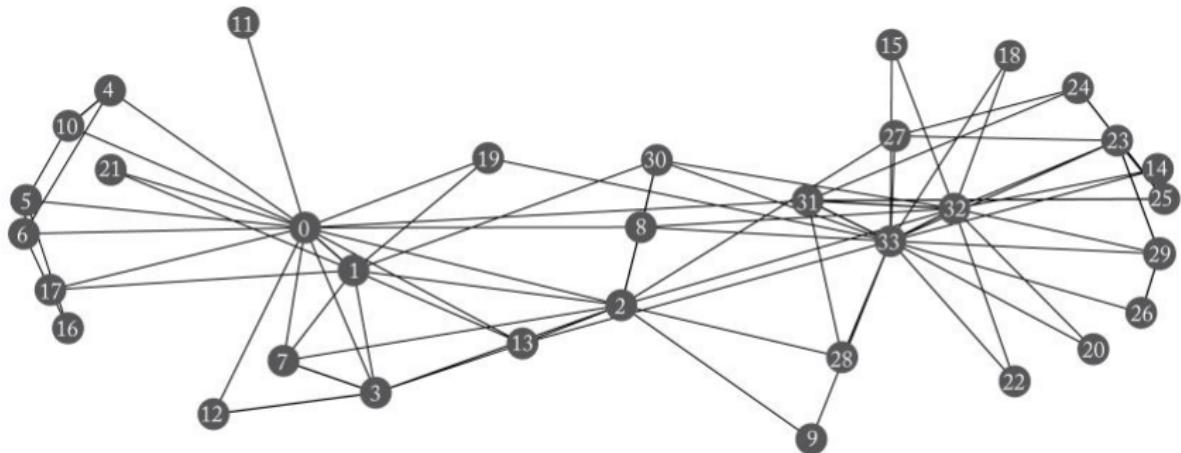
- ▶ What is a graph?
- ▶ A graph is simply a collection of nodes, together with a set of edges which signify relationships between those nodes.
- ▶ As a simple example a graph could represent a social network where the nodes are individuals and edges signify friendship between pairs of individuals.
- ▶ The key idea behind the graph formalism is that its focus is on the relationship between points, rather than just the properties of the individual points.

- ▶ Formally a graph $G = (V, E)$ is defined as a set of vertices V and a set of edges E such that an edge going from node $u \in V$ to $v \in V$ is denoted as $(u, v) \in E$
- ▶ A simple graph, which is the kind that we will be most interested in, is a graph where there is atmost one edge between any pair of nodes, no edges between a node and itself, and where the edges are all undirected, i.e $(u, v) \in E \implies (v, u) \in E$.

Example of a simple graph



Here is an example of a simple graph.



- ▶ A convenient way to represent a simple graph is through the use of adjacency matrices.
- ▶ An adjacency matrix is of dimension $|V| \times |V|$, ie $A \in R^{|V| \times |V|}$
- ▶ We order the nodes in the graph such that every node represents a particular row and column of the adjacency matrix.
- ▶ If an edge $(u, v) \in E$, we set $A[u, v] = 1$, otherwise we set $A[u, v] = 0$.
- ▶ For undirected graphs A is necessarily a symmetric matrix.
For directed graphs it may be asymmetric.

Multi-relational graph



- ▶ Beyond the categories of directed and undirected, weighted and unweighted graphs we can think of one more category - graphs which contain different types of edges.
- ▶ As an example, consider a situation where taking a pair of drugs together might result in multiple side-effects.
- ▶ We would like to represent each side-effect as a separate edge between the pair of nodes in question.
- ▶ We extend the edge notation $(u, v) \in E$ to include an edge or relation type τ so that a multi-relational edge takes the form $(u, v, \tau) \in E$.
- ▶ This boils down to having one adjacency matrix A_r for each edge type r .

- ▶ The preceding argument show that in the context of multi-relational graphs we are dealing with an adjacency tensor $A \in R^{|V| \times |R| \times |V|}$.
- ▶ In the context of graph neural networks, there are two important subsets of multi-relational graphs, i.e heterogeneous graphs and multiplex graphs.
- ▶ Heterogeneous graphs are those in which associate a type attribute to the nodes of the graph.
- ▶ Thus the nodes of the graph V can be partitioned into disjoint sets V_1, V_2, \dots, V_n .

- ▶ One use case of a heterogeneous graph is in a biomedical setting.
- ▶ We may have one type of node representing diseases, another node type representing drugs and yet another type representing proteins.
- ▶ Edges representing treatments will be placed between drug nodes and disease nodes. Other types of edges might be placed only between drug nodes.

Multi-relational graph - Multiplex graphs



- ▶ In multiplex graphs the graphs can be decomposed into a set of k layers.
- ▶ Each node is taken to belong to all the layers.
- ▶ Each layer corresponds to an intra-layer edge type.
- ▶ There could be an inter-edge layer type connecting a given node across layers.

inter-layer edge

- ▶ A multiplex transportation network can be modeled as a multiplex graph.
- ▶ Each node might represent a city and each layer might represent a different mode of transportation, for example air travel and train travel
- ▶ Intra-layer edges between a pair of cities would mean that the two cities are connected by a particular mode of transportation.
- ▶ Inter-layer edges would mean that there is a possibility of switching modes of transportation within a particular city.

- ▶ We are familiar with machine learning in a standard context covering supervised learning and unsupervised learning.
- ▶ These notions will need to be modified in the context of graph neural networks as they are not the most useful or informative for graph neural networks.
- ▶ We shall study the problem of machine learning in the context of node classification on graphs, relation prediction, clustering and community detection,
- ▶ We shall also study the problem of graph classification, regression and clustering.

- ▶ The basic idea here is to predict the label, category or attribute y_u of all the nodes in V given only the true labels of a subset V_{train} of V .
- ▶ A practical instance of node classification is the following: suppose we have a large social network consisting of many bots. The presence of bots is problematic for a company trying to advertise its products to only genuine users. The bots might also be indulging in behaviour violative of the code of conduct for users on the social network. Therefore it is important to evolve a mechanism to identify bots.
- ▶ We seek a model that classifies users as bots given only a small number of manually labeled examples.

- ▶ Node classification seems similar to standard supervised classification from machine learning but there are some important differences.
- ▶ Standard supervised classification requires that the instances for which training labels are available be independent and identically distributed.
- ▶ Given such labeled instances we are asked to predict values for unknown instances drawn from the same distribution.
- ▶ If we do not have these assumptions we cannot hope to learn from few examples and generalize to unknown data points.

- ▶ The problem with node classification is that the nodes of the graph are not independent, identically distributed. There are correlations between the labels of the given nodes.
- ▶ Therefore node classification breaks a key requirement of supervised classification.
- ▶ Many of the key node classification techniques actually take advantage of the relationship between nodes
- ▶ For example, a popular idea in node classification is that nodes tend to have similar labels as their neighbours.
- ▶ The concept of structural equivalence states that nodes that have the same local-neighbourhood structure will have similar labels.

- ▶ We could think of node-classification as a kind of semi-supervised technique, although even here the parallel is not exact.
- ▶ This is because during training we have access to all the test nodes. We just do not have labels for the test nodes.
- ▶ In standard supervised learning, we do not have access to the test nodes, but can rely on the iid assumption.
- ▶ In contrast, for node classification, we do not have the labels of test nodes but know the neighbourhood structure of the test nodes.

Relation prediction



- ▶ In node classification we make inferences about node labels based on knowing the relationships that the nodes have with other nodes that have labels.
- ▶ But how can we make inferences about node labels if some relationships that the node has with other nodes are missing?
- ▶ We need to use machine learning to infer information about the edges in the graph.
- ▶ As a practical example, in a given cell we know of some protein-protein interactions. Can we infer the other missing interactions?

Relation prediction



- ▶ There are many real-world applications of relation prediction - recommending content to social media users, predicting drug side-effects, inferring new relationships in a relational database.
- ▶ Formally we are given a set of nodes V and a set of edges $E_{train} \subset E$. Can we use this information to infer the missing edges $E - E_{train}$?

Relation prediction



- ▶ The complexity of the task depends on the graph data we are examining.
- ▶ For simple social networks with friendship relationships, two nodes will have an edge between them if their neighbour sets have a large intersection, i.e they have many common neighbours.
- ▶ This strategy might not work for complex multi-relational graphs.

- ▶ Community detection is the graph analog of unsupervised clustering.
- ▶ Why do we wish to study this problem?
- ▶ Many real-life graphs exhibit community structure, i.e it is not equally likely that there is an edge between any two pair of nodes.
- ▶ For example, consider a collaboration graph that is constructed so that any two researchers who have co-authored a paper together will be connected together by an edge in the graph. This graph is likely to have dense clusters, for example a Computer Science cluster, a Biology cluster and so on.

- ▶ These are classification, regression and clustering problems for entire graphs.
- ▶ For example, given a graph representing the structure of a molecule, can we build a regression model that predicts that molecule's toxicity or solubility?
- ▶ As an example of a classification problem, consider the problem of classifying a computer program as malicious or not. We could create a graph-based representation of its syntax and data-flow, and see if we can detect maliciousness analyzing such a graph.
- ▶ In graph clustering, the task is to learn an unsupervised measure of similarity between pairs of graphs.



Lecture 2

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- ▶ What kind of machine learning approaches were used for learning on graphs before the advent of deep learning approaches?
- ▶ We will begin with a discussion of basic graph statistics, kernel methods and their use for node and graph classification tasks.
- ▶ We will introduce and discuss various measures for measuring the overlap between node neighbourhoods which underpins techniques for relation prediction.
- ▶ We will look at spectral clustering using graph Laplacians.

- ▶ Traditional approaches to graph classification using graph data follow the standard machine learning paradigm that was popular before the advent of deep learning.
- ▶ Following a traditional machine learning approach means extracting some features from the graphs and using them as input to a standard machine learning classifier like logistic regression.
- ▶ But what were these features?
- ▶ We shall illustrate node-level statistics and features by way of an example using the following social network illustrating 15th century Florentine marriages.

Node-level features and statistics

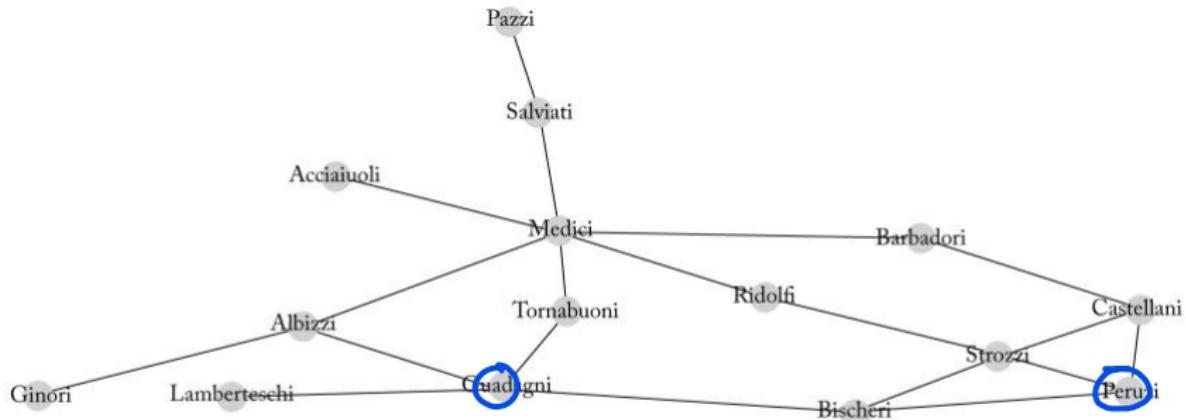


Figure 2.1: A visualization of the marriages between various different prominent families in 15th-century Florence [Padgett and Ansell, 1993].

Pazzi vs Guadagni

- ▶ The graph in the previous slide shows the Medici family at the centre with marital relationships with other prominent Florentine families.
- ▶ Political marriages were an important way of consolidating power in Florentine politics in the 15th century, and the network of marriage connections of the Medicis helped in their rise to political power.
- ▶ We would like to answer the following question from a machine learning perspective: what features or statistics could a machine learning model use to predict the rise of the Medicis to political power?
- ▶ What properties or statistics of the Medici node distinguish it from the rest of the graph?

- ▶ The properties and statistics that we use as features could be used as input to a node classification model.
- ▶ We would not be able to train a machine learning model on a graph as small as the Florentine marriages network.
- ▶ It is illustrative to consider the kinds of features that could be used to distinguish nodes in a real-world network.

- ▶ The most straightforward feature to observe is node degree.
- ▶ The degree of a node $u \in V$ is denoted by d_u and is simply the number of edges incident on the node.
- ▶
$$d_u = \sum_{v \in V} A[u, v].$$
- ▶ In case of directed and weighted graphs we have different notions of degree - corresponding to outgoing edges or incoming edges, by summing over rows or columns in the above equation.
- ▶ The highest node degree in the Florentine-marriage graph is indeed the Medici node, but it does not outrank the closest competitors by a lot.

- ▶ Node degree simply measures the number of nodes that are neighbours of a given node.
- ▶ This might not be enough to measure the importance of a given node.
- ▶ We could look for a measure called node centrality to assess the importance of a given node.
- ▶ There are different ways of measuring centrality - let us look into some of them.

Eigenvector centrality

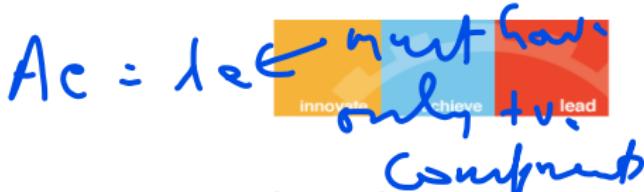


- ▶ Eigenvector centrality takes into account how important a node's neighbours are, whereas degree is simply a measure of how many neighbours a given node has.
- ▶ We can define eigenvector centrality e_u of a node $u \in V$, using a recurrence relation in terms of the eigenvector centralities of its neighbours. It is simply the weighted average of the centralities of all its neighbours:

with $\rightarrow e_u = \frac{1}{\lambda} \sum_{v \in V} A[u, v]e_v$, where λ is a constant.

Now Using vector notation where we denote \mathbf{e} as the vector of node centralities we see that the above equation is a restatement of the eigenvector equation for the adjacency matrix $\mathbf{A}\mathbf{e} = \lambda\mathbf{e}$.

Eigenvector centrality

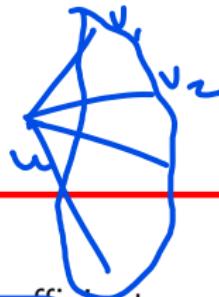


- ▶ Assuming that we require only positive centrality values, we can use the Perron-Frobenius theorem to see that the eigenvector we are looking for corresponds to the largest eigenvalue of A → adjacency matrix
- ▶ How can we interpret eigenvector centrality?
- ▶ One way to interpret it is that eigenvector centrality ranks the likelihood that a node is visited on a random walk of infinite length on the graph.
- ▶ To see this, note that since we are interested in the eigenvector corresponding to the largest eigenvalue we can use the power iteration method to find the eigenvector.
- ▶ We have the power iteration $\mathbf{e}^{t+1} = \mathbf{A}\mathbf{e}^t$. We start with $\mathbf{e}^0 = [1, 1, \dots, 1]^T$.

- ▶ We can see from the power-iteration method that e^1 gives the number of paths of length 1 that end at each node. For any node the degree of the node is the number of length 1 paths that end at that node.
- ▶ e^t gives the number of paths of length t that arrive at each node. The way to see this is as follows: since e^{t-1} gives the number of paths of length $t - 1$ arriving at each node v we can take each of those paths and affix the edge from v to u to get a path of length t arriving at node u . When we compute Ae^{t-1} we see that $A[u, v] = 1$ exactly when there is an edge between u and v . Thus Ae^{t-1} is the total number of paths of length t arriving at node u once we consider all paths of length $t - 1$ ending at neighbours of u .

- ▶ Degree and centrality are useful in distinguishing the Medici family from the other nodes in the network. How do we distinguish between the other nodes in the graph?
- ▶ The Peruzzi and Guadagni nodes in the Florentine marriage graph have very similar degrees (3 and 4 respectively) and very similar eigenvector centralities (0.28 vs 0.29)
- ▶ However the two nodes have very different neighbourhoods - the Peruzzi node forms a complete subgraph with respect to its neighbourhood whereas the Guadagni node ~~sits~~ sits at the center of a star-shaped network with respect to its neighbourhood.
- ▶ It turns out that we can distinguish between nodes like Peruzzi and Guadagni by using the clustering coefficient which is a measure of the number of closed triangles in a node's local neighbourhood.

Clustering coefficient



- ▶ One version of the clustering coefficient can be computed as follows: $c_u = \frac{|\{(v_1, v_2) \in E : (v_1, v_2) \in N(u)\}|}{\binom{d_u}{2}}$ $d_u = \text{degree } u$
- ▶ The numerator in the above formula counts the number of triangles formed between the node u and vertices in its neighbourhood. The denominator counts the total number of possible triangles between u and vertices in its neighbourhood.
- ▶ A clustering coefficient of 1 means that all of node u 's neighbours are connected to each other.

- ▶ This measure helps distinguish between the Peruzzi and Guadagni nodes - for the Guadagni node the clustering coefficient is 0 while for the Peruzzi node, it is 0.66.
- ▶ A study of real-world networks from the social and biological sciences shows that the clustering coefficients of nodes tends to be much higher than their values in random graphs, i.e where the edges are placed randomly. This suggests that there is a lot of structure in real-world graphs.

- ▶ How do we go about the task of graph-level classification?
- ▶ As an example of a graph-level classification problem, we would like to classify the solubility of molecules based on their graph structure.
- ▶ We will describe a bunch of methods that fall in the category of graph-kernel methods, which are approaches to designing features for graphs or implicit kernel functions which can then be used in machine-learning algorithms.

- ▶ Let us first describe the simplest approach to graph-level classification, known as the Bag of Nodes model.
- ▶ The idea here is to simply aggregate node-level statistics. We could compute histograms based on node-degrees and use the histogram as a representation for the given graph.
- ▶ The main drawback of this approach is that it essentially collect local-level information and can miss global properties in the graph which may play a crucial role in graph classification.

- ▶ It uses a strategy of iterative-neighbourhood aggregation as a first improvement over the bag of nodes approach.
- ▶ The basic idea behind the Weisfieler-Lehman kernel can be described in the following steps:
 - ▶ Assign an initial label $l^0(v)$ to each node which could just be the degree of the node.
 - ▶ Iteratively assign a new label to each node based on the hash value returned by hashing the multi-set of the current labels within the node's neighbourhood.
$$l^i(v) = \text{HASH}(l^{i-1}(u), \forall u \in N(v))$$
- ▶ After running K iterations, the label $l^K(v)$ for each node summarizes its K -hop neighbourhood. We can compute histograms over these labels as a feature-representation for the graph.

- ▶ The WL kernel is well-studied and is known to have important theoretical properties.
- ▶ One popular way to approximate graph isomorphism is to see if the two graphs in question have the same label sets after K rounds of labeling. This approach works for a broad set of graphs.

- ▶ One powerful strategy for defining features over graphs is to simply count the occurrence of different small subgraph structures.
- ▶ The subgraph structures are called graphlets.
- ▶ The graphlet kernel involves enumerating all possible graph structures of a particular size and counting each of them in the given graph.
- ▶ Counting these graphlets is a combinatorially difficult problem.

- ▶ In this approach we simply examine the different kinds of paths that occur in the graph.
- ▶ One example is the random walk kernel where we perform random walks in the graph, and then count the occurrence of different degree sequences.
- ▶ Another idea is to count the different degree sequences but only on the shortest paths between nodes rather than random walks.
- ▶ One advantage of path-based methods is that we do not have to deal with combinatorial explosion of graphlets.

Neighbourhood overlap detection



- ▶ The node and graph-level statistics are useful for many classification tasks but they cannot capture the relationships between nodes.
- ▶ We consider various statistical measures of neighbourhood overlap between pairs of nodes. A high amount of neighbourhood overlap between a pair of nodes can be used to infer that there should be an edge between the two nodes.
- ▶ The simplest neighbourhood overlap measure simply counts the number of neighbours that the two nodes share:
 $S[u, v] = |N(u) \cap N(v)|$, where $S[u, v]$ denotes the relationship between nodes u and v , and $S \in R^{|V| \times |V|}$ is the matrix of similarity measures.
- ▶ We can now take the likelihood of an edge
 $P(A[u, v] = 1) \propto S[u, v]$.

- ▶ Local overlap statistics are simply functions of the number of neighbours that vertices u and v share, i.e functions of $|N(u) \cap N(v)|$.
- ▶ The Sorenson index can be defined as follows:
$$S_{Sorenson}[u, v] = \frac{2|N(u) \cap N(v)|}{d_u + d_v}.$$
- ▶ The maximum value of the Sorenson index is 1. Normalization helps in the sense that nodes with high node degrees are not favoured by the measure.
- ▶ Other measures are $S_{Salton}[u, v] = \frac{2|N(u) \cap N(v)|}{\sqrt{d_u d_v}}$,
$$S_{Jaccard}(u, v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}.$$

- ▶ There are some measures that go beyond merely counting the number of common neighbours.
- ▶ These measures take into account the importance of common neighbours in some way.
- ▶ A measure of this kind is the Resource Allocation measure:
$$S_{RA}[v_1, v_2] = \sum_{u \in N(v_1) \cap N(v_2)} \frac{1}{d_u}.$$
- ▶ Here we capture the intuition that if a common neighbour of v_1 and v_2 has high degree, then it is not very important since it is not surprising that a high-degree node would have both v_1 and v_2 as its neighbours.

- ▶ Local overlap measures are often comparable to deep learning based approaches in terms of their effectiveness. However they are still limited in that they do not take advantage of any global information.
- ▶ It is possible in a social network for two nodes to not have any common neighbours but still belong to the same community.
- ▶ The most basic global overlap measure is the Katz index which is computed as follows: $S_{Katz}[u, v] = \sum_{i=1}^{i=\infty} \beta^i A^i[u, v]$.
- ▶ $\beta \in R^+$ dictates how important long paths are compared to short paths. A small value of β indicates that we do not wish to give much importance to long paths.

β is a positive constant

how is this computationally efficient?

The solution to basic geometric series of matrices is given by the following theorem:

Theorem: Let X be a real-valued square matrix and let λ_1 denote the largest eigenvalue of X . Then $(I - X)^{-1} = \sum_{i=0}^{\infty} X^i$ if and only if $\lambda_1 < 1$ and $(I - X)$ is non-singular.

Proof of the theorem



- ▶ Let $s_n = \sum_{i=0}^{i=n} \mathbf{X}^i$. We then note that $\mathbf{X}s_n = \mathbf{X} \sum_{i=0}^{i=n} \mathbf{X}^i = \sum_{i=1}^{i=n+1} \mathbf{X}^i$. This leads to $s_n - \mathbf{X}s_n = \sum_{i=0}^{i=n} \mathbf{X}^i - \sum_{i=1}^{i=n+1} \mathbf{X}^i = \mathbf{I} - \mathbf{X}^{n+1}$.
- ▶ Thus we have $(\mathbf{I} - \mathbf{X})s_n = \mathbf{I} - \mathbf{X}^{n+1}$. Thus $s_n = (\mathbf{I} - \mathbf{X})^{-1}(\mathbf{I} - \mathbf{X}^{n+1})$. Now if the largest eigenvalue $\lambda_1 < 1$, then we see that $\lim_{n \rightarrow \infty} \mathbf{X}^n = 0$. This gives us $\lim_{n \rightarrow \infty} s_n = (\mathbf{I} - \mathbf{X})^{-1}$.

- ▶ Using the result of our theorem, for the Katz measure we see that $S_{Katz} = (\mathbf{I} - \beta \mathbf{A})^{-1} - \mathbf{I}$ where $S_{Katz} \in R^{|V| \times |V|}$ is the full matrix of node-node similarity values.
- ▶ An issue with the Katz index is that it is heavily biased by node degree.
- ▶ It will give higher similarity scores for nodes with high degree since high-degree nodes will be involved in more paths.

- ▶ LHN stands for Leicht, Holme and Newman similarity
- ▶ It improves upon the Katz measure by considering the ratio between the actual number of observed paths and the expected number of observed paths between a pair of nodes.
- ▶ Thus the LHN-similarity can be calculated as follows:
$$\frac{\mathbf{A}^i[u,v]}{E[\mathbf{A}^i[u,v]]}.$$
- ▶ This goes some way in addressing the main shortcoming of the Katz index in that higher-degree nodes are given higher similarity scores.

- ▶ How do we compute the expected number of paths of a certain length between a pair of nodes?
- ▶ We can compute $E[A^1[u, v]] = \frac{d_u d_v}{2m}$, where m is the number of edges in the graph, and d_u and d_v are the node degrees of the nodes u, v .
- ▶ To see why this formula makes sense, an edge leaving the node u will end at the node v with probability $\frac{d_v}{\sum_{i=1}^{n_e} d_i} = \frac{d_v}{2m}$ if edges are placed randomly in the graph.
- ▶ Since there are d_u edges leaving u , the probability that $A[u, v] = 1$ or $P(A[u, v] = 1) = \frac{d_v}{2m} + \frac{d_v}{2m} + \dots + \frac{d_v}{2m} = \frac{d_u d_v}{2m}$. Thus
$$E[A[u, v]] = 1.P(A[u, v] = 1) + 0.P(A[u, v] = 0) = \frac{d_u d_v}{2m}.$$

- ▶ The expected number of paths of length 2, $E[A^2[u, v]]$ can be computed as follows: $E[A^2[v_1, v_2]] = \frac{d_{v_1} d_{v_2}}{(2m)^2} \sum_{u \in V} d_u(d_u - 1)$.
- ▶ To see how this formula arises, consider the following argument: a path of length 2 from v_1 to v_2 could pass through any intermediate vertex u . Such a path would have two sections v_1 to u and u to v_2 . The probability of an edge from v_1 to u is $P(v_1, u) = \frac{d_{v_1} d_u}{2m}$ and the probability of a path from u to v_2 is $P(u, v_2) = \frac{(d_u - 1)d_{v_2}}{2m}$. In the calculation of the second probability we take into account the fact that once one of the edges incident upon u is used by the link to v_1 , there are only $d_u - 1$ edges that can be used to link up to v_2 .

- ▶ Thus the probability of a path of length 2 from v_1 to v_2 through a particular node u is $P(v_1, u)P(u, v_2)$ since the choice of an edge from v_1 to u is independent of the choice of an edge from u to v_2 . Taking the sum over these probabilities over all the nodes u gives us the final expression.
- ▶ Computing the expected number of paths for paths of length 3 or more turns out to be quite hard. We rely on approximating the growth in the number of paths by using the largest eigenvalue.
- ▶ This method is based on the logic outlined in the next slide.

- ▶ Let the vector $\mathbf{v} = \mathbf{e}_i$, i.e the vector of all 0s except for a single 1 in the i th position.
- ▶ Define $\mathbf{p}_1 = \mathbf{A}\mathbf{v}$. We can see that the components of \mathbf{p}_1 represent the number of paths of length 1 from node 1 to node i , node 2 to node i , and so on.
- ▶ Define $\mathbf{p}_2 = \mathbf{A}\mathbf{p}_1$. We see that the components of \mathbf{p}_2 represent the number of paths of length 2 from node 1 to node i , node 2 to node i and so on.
- ▶ Since \mathbf{A} is a symmetric matrix, it has a full-set of eigenvectors which span the entire space. Therefore we can write $\mathbf{v} = c_1\mathbf{s}_1 + c_2\mathbf{s}_2 + \dots + c_n\mathbf{s}_n$ where \mathbf{s}_1 represents the eigenvector corresponding to the dominant eigenvalue

- ▶ We see that $\mathbf{A}^I \mathbf{v} = c_1 \mathbf{A}^I \mathbf{s}_1 + c_2 \mathbf{A}^I \mathbf{s}_2 + \dots + c_n \mathbf{A}^I \mathbf{s}_n = c_1 \lambda_1^I \mathbf{s}_1 + c_2 \lambda_2^I \mathbf{s}_2 + \dots + c_n \lambda_n^I \mathbf{s}_n$.
- ▶ As $I \rightarrow \infty$, $\mathbf{A}^I \mathbf{v} \approx c_1 \lambda_1^I \mathbf{s}_1$ since λ_1 is the dominant eigenvalue.
- ▶ Based on the solution for large i , as well as for $i = 1$ we arrive at $E[\mathbf{A}^i[u, v]] = \frac{d_u d_v \lambda^{i-1}}{2m}$.
- ▶ Thus $S_{LNH}[u, v] = \mathbf{I}[u, v] + \frac{2m}{d_u d_v} \sum_{i=0}^{i=\infty} \beta \lambda^{1-i} \mathbf{A}^i[u, v]$.
- ▶ Using the theorem for geometric series of matrices, we arrive at $S_{LNH} = 2\alpha m \lambda_1 \mathbf{D}^{-1} (\mathbf{I} - \frac{\beta}{\lambda_1} \mathbf{A}) \mathbf{D}^{-1}$, where \mathbf{D} is a diagonal matrix with node degrees on the diagonal.

- ▶ Define the stochastic matrix $\mathbf{P} = \mathbf{AD}^{-1}$. Compute $\mathbf{q}_u = c\mathbf{P}\mathbf{q}_u + (1 - c)\mathbf{e}_u$ where the vector \mathbf{e}_u is a one-hot indicator with a 1 in the u th row and 0s everywhere else.
- ▶ We are simulating a random walk that can be restarted at the node u at any time step with probability $(1 - c)$.
- ▶ The ability to restart the walk at node u allows us to obtain a measure of importance specific to node u . We solve for \mathbf{q}_u to get $\mathbf{q}_u = (1 - c)(\mathbf{I} - c\mathbf{P})^{-1}\mathbf{e}_u$.
- ▶ We can then take the node-to-node similarity measure $S_{RW}[u, v] = \mathbf{q}_u[v] + \mathbf{q}_v[u]$, which measures how likely we are likely to reach each node with a random walk starting from the other node.



Lecture 3

Math Foundations Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- ▶ We have discussed traditional approaches to classification of graph data as well as traditional approaches to relation prediction.
- ▶ We now turn to the problem of learning how to cluster nodes in a graph.
- ▶ First we need to define some important matrices that can be used to represent graphs (other than the adjacency matrix). We will define a new matrix called a Laplacian that will play a crucial role in spectral graph theory.

- ▶ The most basic Laplacian is the unnormalized Laplacian.
- ▶ It is defined in terms of the Laplacian matrix $L = D - A$, where A is the adjacency matrix as before, and D is the diagonal matrix representing node degrees. The nodes constituting the rows of both D and A are organized in the same order, so that the D_{ii} represents the node degree of the i th node and A_i is the row corresponding to the i th row.
- ▶ The first thing we observe about the Laplacian matrix is that it is symmetric, ie $L^T = L$.
- ▶ Another property is that it is positive semi-definite, i.e $x^T L x \geq 0$ for any vector x .

- ▶ We can see that $\mathbf{x}^T \mathbf{L} \mathbf{x} = \mathbf{x}^T (\mathbf{D} - \mathbf{A}) \mathbf{x} = \sum_{u=1}^{u=N} \mathbf{D}_{uu} x_u^2 - \sum_{u=1}^{u=N} \sum_{v=1}^{v=N} x_u \mathbf{A}_{uv} x_v.$
- ▶ This can be rewritten as $\frac{1}{2} \sum_{u \in V} \sum_{v \in V} \mathbf{A}_{uv} (\mathbf{x}_u - \mathbf{x}_v)^2.$
- ▶ To see this note that $\mathbf{D}_{uu} = \sum_{v \in V} \mathbf{A}_{uv}.$ Thereafter the above can be obtained by simple rearrangement.
- ▶ Finally we can write $\mathbf{x}^T \mathbf{L} \mathbf{x} = \sum_{(u,v) \in E} (\mathbf{x}_u - \mathbf{x}_v)^2$ which shows that the quantity is always greater than or equal to zero.

- ▶ The matrix L has $|V|$ non-negative eigenvalues.
- ▶ The smallest eigenvalue is 0. To see this note that $x = [1, 1, 1 \dots 1]^T$, the vector of all ones is in the nullspace of L .
- ▶ The eigenvalues can be ordered as follows: $\lambda_{|V|}, \lambda_{|V|-1}, \dots \lambda_1$.

The Laplacian matrix L summarizes many important properties of the graph. For example, we have the following theorem:

Theorem The geometric multiplicity of the 0 eigenvalue of the Laplacian L corresponds to the number of connected components in the graph.

The proof is given in the next slide.

Proof of the theorem



- ▶ We note first that $\mathbf{L}\mathbf{e} = 0\mathbf{e} = 0$ for any eigenvector corresponding to the eigenvalue 0.
- ▶ This gives $\mathbf{e}^T \mathbf{L}\mathbf{e} = 0$ which implies $\sum_{(u,v) \in E} (\mathbf{e}_u - \mathbf{e}_v)^2 = 0$.
- ▶ From the above we conclude that $\mathbf{e}_u = \mathbf{e}_v$ for all $(u, v) \in E$.
This implies \mathbf{e}_u has the same value for all nodes u which belong to the same connected component.

Proof of the theorem



- ▶ Thus if the graph is fully connected there is only one eigenvector corresponding to the eigenvalue 0.
- ▶ If there are K components, we have K possible eigenvectors for the eigenvalue 0, where for the first eigenvector we set $e_u = 1$ for all nodes $u \in V$ that belong to the first component, and 0 for all the other nodes. Similarly for the second eigenvector we set $e_u = 1$ for all nodes u that belong to the second component and 0s for all other nodes.

- ▶ So far we have seen how eigenvectors corresponding to the 0 eigenvalue can be used to assign nodes to clusters which are based on the connected components that contain these nodes.
- ▶ This is a trivial application.
- ▶ What is of greater interest is if we can use the Laplacian to get an optimal clustering of nodes within a fully connected graph.

- ▶ First we need to define what we mean by an optimal cluster.
- ▶ Let $A \subset V$ and \bar{A} denote the complement of this set.
- ▶ Given a partitioning of the graph into K disjoint subsets we define the cut-value of this partition, $\text{cut}(A_1, A_2, \dots, A_K) = \frac{1}{2} \sum_{k=1}^{k=K} |(u, v) \in E, \text{ such that } u \in A_k, v \in \bar{A}_k|$.
- ▶ The cut value is simply a count of the number of edges that cross the boundaries across the partitions of nodes.
- ▶ We could define an optimal clustering of nodes into K clusters as a partition that minimizes the cut value.

- ▶ One problem with the definition of an optimal cluster as given on the previous slide is that it tends to create clusters containing just one node.
- ▶ To avoid this problem, we could minimize the cut while ensuring that all the partitions are reasonably large.
- ▶ Thus we would like to minimize the

$$\text{RatioCut}(A_1, A_2, \dots, A_k) = \frac{1}{2} \sum_{k=1}^{k=K} \frac{|(u, v) \in E, \text{ such that } u \in A_k, v \in \bar{A}_k|}{|A_k|}$$

Minimizing *RatioCut* using the Laplacian



- ▶ We will minimize the *RatioCut* using the Laplacian spectrum.
- ▶ Let us consider the case $K = 2$. The problem becomes
 $\text{minimize}_{A \in V} \text{RatioCut}(A, \bar{A})$
- ▶ We can rewrite this problem in a more convenient way by defining the vector $\mathbf{a}_u = \sqrt{\frac{|\bar{A}|}{|A|}} \mathbf{u}$, $u \in A$ and $\mathbf{a}_u = -\sqrt{\frac{|A|}{|\bar{A}|}} \mathbf{u}$, $u \in \bar{A}$.
- ▶ We know that $\mathbf{a}^T \mathbf{L} \mathbf{a} = \sum_{(u,v) \in E} (\mathbf{a}_u - \mathbf{a}_v)^2$.
- ▶ We can further simplify the above into something very interesting relating directly to the *RatioCut*.

Minimizing *RatioCut* using the Laplacian



$$\begin{aligned}\mathbf{a}^T \mathbf{L} \mathbf{a} &= \sum_{(u,v) \in E} (\mathbf{a}_u - \mathbf{a}_v)^2 \\&= \sum_{(u,v) \in E: u \in A, v \in \bar{A}} (\mathbf{a}_u - \mathbf{a}_v)^2 \\&= \sum_{(u,v) \in E: u \in A, v \in \bar{A}} \left(\sqrt{\frac{|\bar{A}|}{|A|}} - \left(-\sqrt{\frac{|A|}{|\bar{A}|}} \right) \right)^2 \\&= \text{cut}(A, \bar{A}) \left(\frac{|\bar{A}|}{|A|} + \frac{|A|}{|\bar{A}|} + 2 \right) \\&= \text{cut}(A, \bar{A}) \left(\frac{|\bar{A}| + |A|}{A} + \frac{|A| + |\bar{A}|}{\bar{A}} \right) \\&= |V| \text{RatioCut}(A, \bar{A})\end{aligned}$$

$\frac{|A|}{|\bar{A}|} + \frac{|\bar{A}|}{|A|} = \frac{|\bar{A}| + |A|}{|A|}$



Minimizing *RatioCut* using the Laplacian



- ▶ We can see that the vector \mathbf{a} defined in the previous slide is orthogonal to the vector of all ones, ie $\mathbf{a} \perp [1, 1, 1, \dots, 1]^T$.
- ▶ Also $\|\mathbf{a}\|^2 = |V|$.
- ▶ The problem of minimizing the *RatioCut* boils down to finding an assignment to each node of the vertex set V to the category A or \bar{A} . This is a discrete optimization problem which we will formally set up in the next slide.

Minimizing *RatioCut* using the Laplacian



We can set up the formal optimization problem as follows over the vector \mathbf{a} defined previously:

$$\min \mathbf{a}^T \mathbf{L} \mathbf{a} \text{ subject to}$$

$$\mathbf{a} \perp \mathbf{1}$$

$$\|\mathbf{a}\|^2 = |V|$$

Minimizing *RatioCut* using the Laplacian



- ▶ The formal problem defined in the previous slide turns out to be NP-hard. It is essentially a discrete optimization problem where we need to decide to put each vertex into one of two buckets and is quite like the satisfiability problem for Boolean formulae.
- ▶ Instead of solving the discrete optimization problem we can choose to relax the problem by allowing a to be any real vector. That would turn the given problem into a more tractable continuous optimization problem.

Minimizing *RatioCut* using the Laplacian



- ▶ The Rayleigh-Ritz theorem states that the solution to the continuous optimization problem is obtained by setting \mathbf{a} to the eigenvector corresponding to the second-smallest eigenvalue of the Laplacian matrix \mathcal{L} .
- ▶ Let us delve into this result ...
- ▶ Since \mathcal{L} is a symmetric matrix, by the spectral theorem, it can be factorized as follows: $\mathcal{L} = \mathbf{Q}\Lambda\mathbf{Q}^T$ where \mathbf{Q} is an orthogonal matrix consisting of eigenvectors of \mathcal{L} .
- ▶ The first eigenvector in \mathbf{Q} corresponding to the eigenvalue 0 is $\mathbf{1}$.

Minimizing *RatioCut* using the Laplacian



- ▶ We can write $\mathbf{a} = c_1 \mathbf{q}_1 + c_2 \mathbf{q}_2 + \dots + c_N \mathbf{q}_N$ since the vectors $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_N$ constitutes a basis for the space R^N .
- ▶ The objective function $\mathbf{a}^T \mathbf{L} \mathbf{a}$ becomes $c_1^2 \lambda_1 + c_2^2 \lambda_2 + \dots + c_N^2 \lambda_N$ and the constraint boils down to requiring that $c_1^2 + c_2^2 + \dots + c_N^2 = |V|$. Note that $\lambda_1 \leq \lambda_2 \dots \leq \lambda_N$.
- ▶ Since \mathbf{a} is orthogonal to the first eigenvector we must have $c_1 = 0$.
- ▶ Then the objective function $\mathbf{a}^T \mathbf{L} \mathbf{a}$ is clearly minimized for $c_2 = \sqrt{|V|}, c_3 = c_4 = \dots = c_N = 0$. This gives the result that \mathbf{a} is the eigenvector corresponding to the second smallest eigenvalue.

Minimizing *RatioCut* using the Laplacian



- ▶ It now remains to convert the solution to the continuous optimization problem to a solution to the original discrete optimization problem.
- ▶ This can be done by assigning nodes to the clusters A or \bar{A} based on the sign of the corresponding entry in \mathbf{a} .
- ▶ In other words we set $u \in A$ if $\mathbf{a}[u] \geq 0$ and $u \in \bar{A}$ if $\mathbf{a}[u] < 0$.

- ▶ The general idea can be extended to K clusters by examining the K smallest eigenvectors of the Laplacian.
- ▶ The steps of this general approach can be stated as below:
 - ▶ Find the K smallest eigenvectors of \mathbf{L} . Label them: $\mathbf{e}_{|V|-1}, \mathbf{e}_{|V|-2}, \dots, \mathbf{e}_{|V|-K}$.
 - ▶ Form the matrix $\mathbf{U} \in R^{|V| \times (K-1)}$ with the eigenvectors from Step 1 as columns.
 - ▶ Represent each node by its corresponding row in the matrix \mathbf{U} . Let $z_u = \mathbf{U}[u], \forall u \in V$.
 - ▶ Run K -means clustering on the node embeddings $z_u, \forall u \in V$.



Lecture 4

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Neighbourhood reconstruction methods



- ▶ We will now be concerned with learning node embeddings.
- ▶ The idea is to encode nodes in terms of low-dimensional vectors that summarize their position in their graph and the structure of their local neighbourhood.
- ▶ The geometric relationship between nodes such as the Euclidean distance between them will give an idea about the strength of the relationship between them.

- ▶ We view the graph representation learning problem as involving two key operations.
- ▶ First we encode each node in the graph as a low-dimensional vector or embedding.
- ▶ Then the operation of decoding consists of taking the low-dimensional vector encodings of the nodes and reconstructing the local neighbourhood of the node in the graph, such as finding out its immediate neighbours.
- ▶ Let us look at these steps in greater detail.

- ▶ The encoder is a function that maps nodes $v \in V$ to a vector embedding $\mathbf{z}_v \in R^d$.
- ▶ The simplest way of looking at the encoder is in terms of the following: $ENC : V \rightarrow R^d$.
- ▶ This is known as a shallow embedding as the encoder function is simply an embedding lookup based on the node ID.
- ▶ We can view the encoder as a matrix $\mathbf{Z} \in R^{|V| \times d}$ where the v th row \mathbf{Z}_v is the d -dimensional vector corresponding to the embedding for node v .

- ▶ The role of the decoder is to reconstruct certain graph statistics from the node embeddings that are generated by the encoder.
- ▶ For example, given a node embedding z_u for node u , the decoder could reconstruct the row corresponding to the node u in the adjacency matrix \mathbf{A} , i.e \mathbf{A}_u .
- ▶ Conceptually this becomes possible if the encoder used similar-looking embeddings for neighbouring vertices.
- ▶ Let us capture the functioning of a decoder more formally.

The decoder



- ▶ The standard practice is to define pairwise decoders whose signature can be expressed as follows: $DEC : R^d \times R^d \rightarrow R^+$.
- ▶ In other words the pairwise decoder takes two d -dimensional vectors representing the embeddings of two nodes and outputs a positive number describing their affinity or similarity.
- ▶ Applying the pairwise decoder to a pair of embeddings (z_u, z_v) results in the reconstruction of the relationship between nodes u and v .

$DEC(ENC(u), ENC(v)) = DEC(z_u, z_v) \approx S[u, v]$

*↑
nodes with node IDs*

*Similarity matrix
provided during training*

- ▶ In the preceding slide, $\mathbf{S}[u, v]$ stands for a similarity measure.
- ▶ It could be the familiar adjacency matrix \mathbf{A}
- ▶ Nothing stops us from defining \mathbf{S} using any of the node overlap statistics that we have studied in previous lectures.
- ▶ With the definition of the pairwise decoder now given, it remains to find a way to optimize the encoder-decoder model.

- ▶ We need to find an encoder that achieves the aim of reconstruction as well as possible.
- ▶ To do so, we set up an empirical loss function that captures the loss L over a set of training pairs D .
- ▶ Formally the loss function can be expressed as follows:

$$L = \sum_{(u,v) \in D} I(DEC(z_u, z_v), S([u, v]))$$

- ▶ Here $I : R \times R \rightarrow R$ is a loss function that measures the discrepancy between the decoded similarity values $DEC(z_u, z_v)$ and the actual similarity value $S[u, v]$.

in f

R

- ▶ The loss function could be a mean-squared loss function familiar to practitioners of machine learning, or it could even be a cross-entropy based loss function.
- ▶ The overall objective is to train the encoder and decoder so that the pairwise node relationships can be effectively reconstructed on the training set D .
- ▶ Minimizing the loss function is usually done by stochastic gradient descent.
- ▶ In some special cases we can use techniques from linear algebra such as matrix factorization.

Loss function



Method	Decoder	Similarity Measure	Loss Function
Lap Eigenmaps	$\ \mathbf{z}_u - \mathbf{z}_v\ _2^2$	General	$\text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]$
Graph Factorization	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u, v]$	$\ \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]\ _2^2$
GraRep	$\mathbf{z}_u^\top \mathbf{z}_v$	$\mathbf{A}[u, v], \dots, \mathbf{A}^k[u, v]$	$\ \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]\ _2^2$
HOPE	$\mathbf{z}_u^\top \mathbf{z}_v$	General	$\ \text{DEC}(\mathbf{z}_u, \mathbf{z}_v) \cdot \mathbf{S}[u, v]\ _2^2$
DeepWalk	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p\mathcal{G}(v u)$	$-\mathbf{S}[u, v] \log(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v))$
node2vec	$\frac{e^{\mathbf{z}_u^\top \mathbf{z}_v}}{\sum_{k \in \mathcal{V}} e^{\mathbf{z}_u^\top \mathbf{z}_k}}$	$p\mathcal{G}(v u)$ (biased)	$-\mathbf{S}[u, v] \log(\text{DEC}(\mathbf{z}_u, \mathbf{z}_v))$

- ▶ The question of decoding a node's neighbourhood structure is essentially the same as reconstructing the adjacency matrix of the graph.
- ▶ We can view this task in terms of using matrix factorization to learn a low-dimensional approximation of a node-node similarity matrix \mathbf{S}
- ▶ \mathbf{S} could be the adjacency matrix itself or it could be represent a more generalized version of similarity.
- ▶ Let us look at the method of Laplacian eigenmaps which is one of the earliest and most influential matrix-factorization approach.

- ▶ We define the decoder based on the L_2 -distance between the node embeddings: $DEC(\mathbf{z}_u, \mathbf{z}_v) = \|\mathbf{z}_u - \mathbf{z}_v\|_2^2$.
- ▶ The loss function then weighs pairs of nodes according to their similarity in the graph $L = \sum_{(u,v) \in D} DEC(\mathbf{z}_u, \mathbf{z}_v) \mathbf{S}[u; v]$
- ▶ The idea behind constructing a loss function like the above is to penalize the model when it assigns very different embeddings to nodes that are very similar.
- ▶ If \mathbf{S} is constructed so that it satisfies the properties of a Laplacian matrix, then the node embeddings that minimize the loss are identical to the solution for spectral clustering - i.e the d smallest eigenvectors excluding the eigenvector of all ones give the d -dimensional optimal node embeddings.

- ▶ How does this result arise?
- ▶ We shall present an analysis from the paper by Belkin and Niyogi, 2002.
- ▶ Consider the problem of mapping the nodes of a weighted, connected graph G so that connected points stay as close as possible together.
- ▶ We choose $y_i \in R$ to minimize $\sum_{i,j} (y_i - y_j)^2 W_{ij}$ Let $\mathbf{y} = (y_1, y_2, y_3, \dots, y_n)^T$ be the map from the graph to the real line.

$$\sum_{i,j} (y_i - y_j)^2 w_{ij} = \sum_{(i,j) \in E} (y_i - y_j)^2 = \mathbf{y}^T L \mathbf{y}$$

- ▶ Recall from a previous lecture on Laplacians and spectral clustering that for any \mathbf{y} , we note that
$$\frac{1}{2} \sum_{i,j} (y_i - y_j)^2 W_{ij} = \mathbf{y}^T \mathbf{L} \mathbf{y}.$$
 Here $\mathbf{L} = \mathbf{D} - \mathbf{W}$
- ▶ The minimization problem reduces to $\operatorname{argmin}_{\mathbf{y}^T \mathbf{D} \mathbf{y} = 1} \mathbf{y}^T \mathbf{L} \mathbf{y}.$
- ▶ Why do we need the constraint $\mathbf{y}^T \mathbf{D} \mathbf{y} = 1$? This is needed to remove an arbitrary scaling factor in the embedding.
- ▶ From the above equation and noting that \mathbf{L} is a positive-semidefinite matrix, we see that the vector \mathbf{y} that minimizes the objective function is given by $\mathbf{L} \mathbf{y} = \lambda \mathbf{D} \mathbf{y}.$

- ▶ To see the previous result, note that the optimization problem is of the form

$$\begin{aligned} & \min \mathbf{y}^T \mathbf{L} \mathbf{y} \text{ subject to} \\ & \mathbf{y}^T \mathbf{D} \mathbf{y} = 1 \end{aligned}$$

- ▶ We can set up the Lagrangian
$$L(\mathbf{y}, \lambda) = f - \lambda g = \mathbf{y}^T \mathbf{L} \mathbf{y} - \lambda \mathbf{y}^T \mathbf{D} \mathbf{y}.$$
- ▶ The stationary point of the Lagrangian is obtained at
$$\nabla f = \lambda \nabla g$$
 which means $\mathbf{L} \mathbf{y} = \lambda \mathbf{D} \mathbf{y}.$



- ▶ Let the vector $\mathbf{1}$ represent the vector of all ones.
- ▶ We have already seen in the previous lecture that $\mathbf{1}$ is in the nullspace of \mathbf{L} and therefore it is only the eigenvector for the eigenvalue 0 when the graph is connected.
- ▶ Mapping all the nodes to the same value on the real line does not make any sense in our context.
- ▶ This is why we put in an additional orthogonality constraint to our optimization problem.

- ▶ Our optimization problem now becomes

$$\min \mathbf{y}^T \mathbf{L} \mathbf{y} \text{ subject to}$$

$$\mathbf{y}^T \mathbf{D} \mathbf{y} = 1$$

$$\mathbf{y}^T \mathbf{1} = 0$$

- ▶ Thus the solution \mathbf{y}_{opt} is given by the eigenvector with the smallest non-zero eigenvalue.
- ▶ In general the embedding of the nodes into R^m is given by the $n \times m$ matrix $\mathbf{Y} = [\mathbf{y}_1 \mathbf{y}_2 \dots \mathbf{y}_m]$ where the i th row, denoted by \mathbf{Y}_i^T provides the embedding coordinates of the i th vertex.

- ▶ More recent work has employed an inner-product based decoder, ie $DEC(\mathbf{z}_u, \mathbf{z}_v) = \mathbf{z}_u^T \mathbf{z}_v$.
- ▶ We assume that the similarity between nodes is given by the dot product of the embeddings of the two nodes.
- ▶ Methods of this kind such as Graph Factorization (GF), GraRep, and HOPE differ from each other primarily in how they define the similarity measure \mathbf{S} .
- ▶ The loss function used is

$$L = \sum_{(u,v) \in D} \|DEC(\mathbf{z}_u, \mathbf{z}_v) - \mathbf{S}[u, v]\|_2^2.$$

- ▶ Inner product methods are referred to as matrix factorization methods since their loss functions can be minimized using factorization algorithms such as SVD.
- ▶ To see that this is the case we create a matrix Z of node embeddings by letting every row of the matrix represent a node embedding vector. Thus we have a $|V| \times d$ matrix Z .
- ▶ The loss function can be represented as $L = \|ZZ^T - S\|_2^2$.
- ▶ This corresponds to a low-dimensional factorization of the node-node similarity matrix S and can be solved using SVD and taking the top d singular values of the SVD decomposition of the matrix S .

$$Z = \begin{bmatrix} & & \\ & \vdots & \\ & & \end{bmatrix} \quad |V| \times d$$
$$ZZ^T = \begin{bmatrix} & & \\ & \ddots & \\ & & \end{bmatrix}$$



Lecture 5

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- ▶ The previous methods based on computing inner-products between node embeddings were all deterministic.
- ▶ They often define the similarity matrix \mathbf{S} as a polynomial in terms of the adjacency matrix \mathbf{A} and optimization is then performed so as to ensure that node embeddings for the nodes u and v , i.e \mathbf{z}_u and \mathbf{z}_v become such that $\underline{\mathbf{z}_u^T \mathbf{z}_v \approx \mathbf{S}[u, v]}$.
- ▶ In this lecture we would like to investigate approaches that adapt the inner-product approach to use stochastic measures of node-overlap.
- ▶ The basic idea incorporated by random-walk embeddings is that two nodes have similar embeddings if they tend to co-occur on short random walks in the graph.

- Mathematically the goal is to learn embeddings so that the following relationship is approximately satisfied:

$$\begin{aligned} DEC(\mathbf{z}_u, \mathbf{z}_v) &= \frac{e^{\mathbf{z}_u^T \mathbf{z}_v}}{\sum_{\mathbf{w} \in V} e^{\mathbf{z}_u^T \mathbf{w}}} \\ &\approx p_{G,T}(v|u) \end{aligned}$$

- Here $p_{G,T}(v|u)$ is the probability of visiting v on a random walk starting from u with T such that $T \in 2, 3, \dots, 10$.
 - Compared to the inner-product measure we see that the similarity measure defined here is both stochastic and asymmetric.
- short random walk*

Training random walk embeddings



- ▶ The idea is to minimize a loss function that models cross-entropy loss as follows:

$$L = \sum_{(u,v) \in D} -\log(DEC(z_u, z_v))$$

DEC(z_u, z_v)
towards 1
To and 1

- ▶ Here D denotes the training set of random walks, which is generated by sampling random walks starting at each node.
- ▶ We can assume that N pairs of co-occurring nodes for each node u are sampled from the distribution $(u, v) \sim p_{G,T}(v|u)$.

Calculating training loss



- ▶ The cross-entropy loss defined in the previous slide is expensive to evaluate.
- ▶ This is because the denominator involves calculating inner products across all nodes for a single node $u \in V$. This has complexity $O(|V|)$.
- ▶ For a training set of size $|D|$, the total complexity would be $O(|D||V|)$.
- ▶ How do we address this computational challenge?
- ▶ We will look at different approaches, including those employed by *DeepWalk* and *node2vec*.

- ▶ Hierarchical softmax is a technique used to reduce the computational burden of the regular softmax computation.
- ▶ The regular softmax computation takes a time of $O(|V|)$. Hierarchical softmax reduces this to $O(\log(|V|))$ by organizing the computation in the form of a binary tree.
- ▶ We shall describe the hierarchical softmax computation following the treatment given in
**[https://yinwenpeng.wordpress.com/2013/09/26/
hierarchical-softmax-in-neural-network-language-model/](https://yinwenpeng.wordpress.com/2013/09/26/hierarchical-softmax-in-neural-network-language-model/)**
- ▶ Consider the NLP problem of predicting the next word given a context. This can be posed in terms of calculating probabilities over words in a vocabulary.

- We can state the central equation concerning the hierarchical softmax probability as follows:

$$p(w|w_I) = \prod_{j=1}^{L(w)-1} \sigma([n(w, j+1) = ch(n(w, j))], (v'_{n(w,j)})^T v_{w_I})$$

*length of the path in the tree from
root to the word*

input context $t(w) = \sigma(\log |V|)$

- The formula above represents a continuous multiplication.
- v_{w_I} represents the embedding of the history seen so far (set of words in a certain time window before the current word).
- $v'_{n(w,j)}$ is the embedding of the j th ancestor of the word in the tree. The inner product $(v'_{n(w,j)})^T v_{w_I}$ represents the similarity between the history and the current word.
- We can show that $\sum_{w=1}^{w=W} p(w|w_I) = 1$.

- ▶ The leaves and non-leaves of the tree will have initialized embeddings which will be updated gradually through learning.
- ▶ The intuition being reflected here is that if a given context is more similar to the ancestor of a given word, then the word in question has a higher probability of being the target word.
- ▶ Given a similarity score between the word and the context stored at a given node, Hierarchical Softmax converts it into two probabilities to search the left and right subtree. This is achieved by the equation given on the previous slide.
- ▶ Looking closely at the equation we observe that $[n(w, j + 1) = ch(n(w, j))]$ returns +1 or -1 depending on whether the $(j + 1)$ th ancestor of the target word is the right or left child of the j th ancestor.

- ▶ We note that for the sigmoid function $\sigma(x) + \sigma(-x) = 1$. This is because $\sigma(x) = \frac{1}{1+e^{-x}}$ and $\sigma(-x) = \frac{1}{1+e^x}$ and their sum can clearly be seen to be 1.
- ▶ Thus the left and right-child probabilities add up to 1.
- ▶ Note that since we are talking about a binary tree there is a unique path from the root to any leaf, so we know which branch to take at any node in order to reach the target word.
- ▶ Thus the whole algorithm consists of making a series of decisions at the interior nodes on the way to the leaf node corresponding to the target word.

A practical example ...

- Example: ["the", "dog", "and", "the", "cat"]

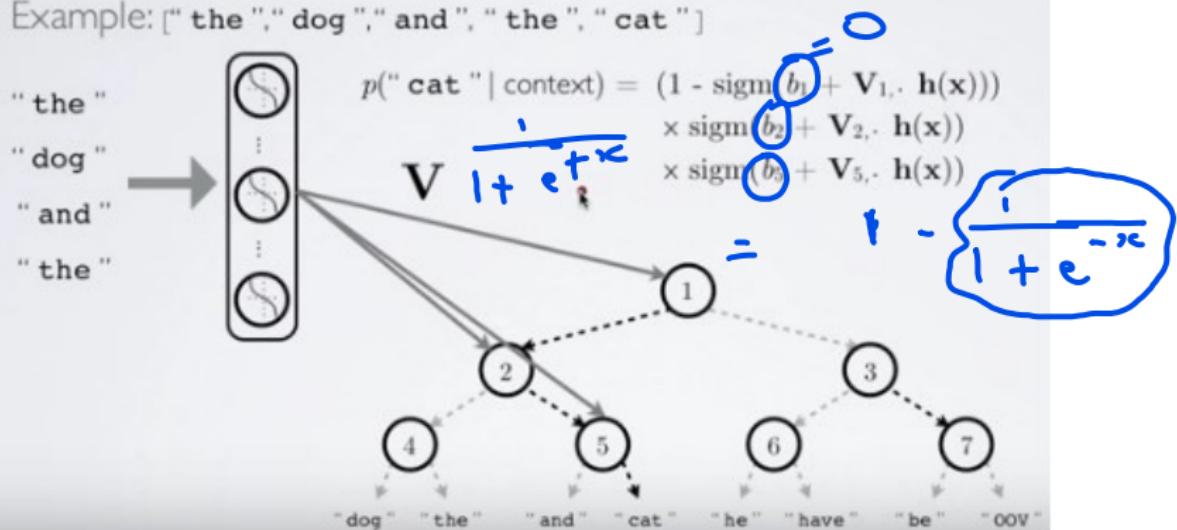


Figure 2: Hierarchical softmax computations ([Hugo Lachouelle's Youtube lectures](#))

- ▶ Sampling-based approaches aim to do away with the softmax computation entirely.
- ▶ We base our discussion of sampling-based approaches on material from the website:
<https://www.ruder.io/word-embeddings-softmax/#softmaxarchitecturemodifications>.
- ▶ These approaches approximate the normalization in the denominator of the softmax with some other loss that is easy to compute.
- ▶ Sampling-based approaches are only useful at training time, since during the test phase the full softmax function needs to be used.

- ▶ We would like to gain some intuitions regarding the softmax denominator's impact on the loss function.
- ▶ Let us compute the gradient of the the loss function J_θ with respect to the parameters θ .

$$\begin{aligned} J_\theta &= -\log(DEC(\mathbf{z}_u, \mathbf{z}_v)) \\ &= -\log \left(\frac{e^{\mathbf{h}^T \mathbf{v}'_w}}{\sum_{\mathbf{w}_i \in V} e^{\mathbf{h}^T \mathbf{v}'_{w_i}}} \right) \end{aligned}$$

- ▶ We can use the formula $\log \left(\frac{x}{y} \right) = \log x - \log y$ to decompose J_θ .

- ▶ We can simplify the expression for J_θ to obtain the following:
$$J_{\theta} = \mathbf{h}^T \mathbf{v}'_w + \log \left(\sum_{w_i \in V} e^{\mathbf{h}^T \mathbf{v}'_{w_i}} \right)$$
- ▶ We can define $J_\theta = \epsilon(w) + \log \left(\sum_{w_i \in V} e^{-\epsilon(w_i)} \right)$ where $\epsilon(w) = \mathbf{h}^T \mathbf{v}'_w$.
- ▶ For back-propagation we need to compute $\nabla_\theta J_\theta$, i.e the gradient of our loss function with respect to the parameters θ .
- ▶ The application of the chain rule results in the following expression for the gradient:

$$\nabla_\theta J_\theta = \nabla_\theta \epsilon(w) + \frac{1}{\sum_{w_i \in V} e^{-\epsilon(w_i)}} \nabla_\theta \sum_{w_i \in V} e^{-\epsilon(w_i)}.$$

Sampling-based approaches



- We can simplify further to obtain

$$\nabla_{\theta} J_{\theta} = \nabla_{\theta} \epsilon(w) + \frac{1}{\sum_{w_i \in V} e^{-\epsilon(w_i)}} \sum_{w_i \in V} e^{-\epsilon(w_i)} \nabla_{\theta} (-\epsilon(w_i)).$$

- We can rewrite this as

$$\nabla_{\theta} J_{\theta} = \nabla_{\theta} \epsilon(w) + \sum_{w_i \in V} \frac{e^{-\epsilon(w_i)}}{\sum_{w_j \in V} e^{-\epsilon(w_j)}} \nabla_{\theta} (-\epsilon(w_i)).$$

$\rightarrow P(w_i|c)$

- The quantity $\frac{e^{-\epsilon(w_i)}}{\sum_{w_j \in V} e^{-\epsilon(w_j)}}$ is simply the softmax probability $P(w_i|c)$. We shall omit the reference to the context c in this expression from this point on.

- Therefore we finally end up with

$$\nabla_{\theta} J_{\theta} = \nabla_{\theta} \epsilon(w) + \sum_{w_i \in V} P(w_i) \nabla_{\theta} (-\epsilon(w_i)).$$

- After some minor adjustments we have

$$\nabla_{\theta} J_{\theta} = \nabla_{\theta} \epsilon(w) - \sum_{w_i \in V} P(w_i) \nabla_{\theta} (\epsilon(w_i)).$$

$$\sum_{\omega} f(\omega) P(\omega) = E(f(\omega))$$

- ▶ Let us analyze the final expression given on the previous slide.
- ▶ We see that the gradient of the loss function has two parts - one, a positive reinforcement of the target word and two, a negative reinforcement for all other words w_i which is weighted by their probability.
- ▶ This negative reinforcement is just the expectation $E_{w_i \sim P}[\nabla_{\theta}\epsilon(w_i)]$. In other words,
$$\sum_{w_i \in V} P(w_i)\nabla_{\theta}\epsilon(w_i) = E_{w_i \sim P}[\nabla_{\theta}\epsilon(w_i)].$$
- ▶ The crux of sampling-based approaches is to approximate this negative reinforcement since we do not want to compute it over all the words in the vocabulary.

- ▶ One way of estimating the expectation in the previous slide is to use the Monte-Carlo method.
- ▶ We take the mean of random samples of the probability distribution.
- ▶ If we knew the probability distribution we could take $E_{w_i \sim P}[\nabla_{\theta}\epsilon(w_i)] = \frac{1}{m} \sum_{i=1}^{i=m} \nabla_{\theta}\epsilon(w_i)$ where w_1, w_2, \dots, w_m are samples from $P(w)$.
- ▶ However, in order to sample from P , we need to compute P which is what we want to avoid. Importance sampling is a technique of sampling from some other distribution Q , called the proposal distribution, from which it is cheap to sample. Preferably Q should be similar to P .

Noise contrastive estimation (NCE)

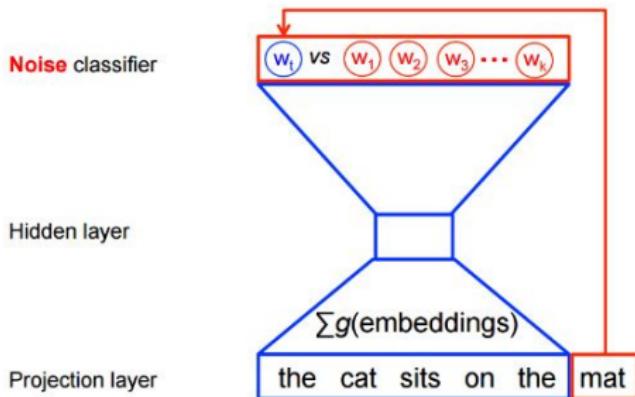


- ▶ Noise contrastive estimation is a more stable sampling method than importance sampling.
- ▶ Importance sampling runs the risk that the proposal distribution Q might diverge from the true distribution P .
- ▶ Noise contrastive estimation does not attempt to calculate the probability of a word directly.
- ▶ It uses an auxiliary loss that optimizes the goal of maximizing the probability of correct words.
- ▶ In NCE we train a model to distinguish the target word from noise.

Noise contrastive estimation (NCE)



The model tries to distinguish genuine data from noise as illustrated in the figure below (taken from Stephan Gouws' Ph.D dissertation):



Noise contrastive estimation (NCE)



- ▶ For every word, given its context consisting of n previous words $w_{i-1}, w_{i-2}, \dots, w_{i-n}$, we generate k samples from a noise distribution Q .
- ▶ We can take Q to be the distribution of unigrams on the training set.
- ▶ Since we are performing binary classification, we label all correct words w_i given the context c_i as true ($y = 1$) and all noise samples \tilde{w}_{ik} as false ($y = 0$).
- ▶ We can now use logistic regression to minimize the negative log-likelihood as follows:

$$J_\theta = - \sum_{w_i \in V} [\log P(y = 1 | w_i, c_i) + k \sum_{\tilde{w}_j \sim Q} [\log P(y = 0 | \tilde{w}_j, c_i)]]$$

$P(y = 1 | w_i, c_i) \rightarrow 1$
 $P(y = 0 | \tilde{w}_j, c_i) \rightarrow 0$

$\frac{1}{1+e^{-y}}$

Noise contrastive estimation (NCE)



- ▶ Instead of taking the expectation $E_{\tilde{w}_{ij} \sim Q}$ over all noise samples, we can choose to approximate it using a Monte-Carlo approximation to get the following:

$$J_\theta = - \sum_{w_i \in V} [\log P(y = 1 | w_i, c_i) + \sum_{j=1}^{j=k} [\log P(y = 0 | \tilde{w}_{ij}, c_i)]]$$

- ▶ We can treat $P(y = 1 | w_i, c_i)$ as a training distribution P_{train} and the distribution $P(y = 0 | \tilde{w}_{ij}, c_i)$ as a noise distribution Q .

Noise contrastive estimation (NCE)



- We can thus represent the probability of sampling either a positive sample or a noise sample as a mixture of those two distributions which are weighted based on the number of samples coming from each -

$$P(w|c) = \frac{1}{k+1} P_{train}(w|c) + \frac{k}{k+1} Q(w|c).$$

first sample a biased coin, then

- Then, given the above mixture, we can calculate the probability $P(y = 1|w, c) = \frac{\frac{1}{k+1} P_{train}(w|c)}{\frac{1}{k+1} P_{train}(w|c) + \frac{k}{k+1} Q(w|c)}$.
- We can finally simplify this to $P(y = 1|w, c) = \frac{P_{train}(w|c)}{P_{train}(w|c) + kQ(w|c)}$.
- Since we do not know $P_{train}(w)$ (this is what we would like to compute), we would like to use $P(w)$ in place of $P_{train}(w)$. This gives us $P(y = 1|w, c) = \frac{P(w|c)}{P(w|c) + kQ(w|c)}$.

Noise contrastive estimation (NCE)



- ▶ We recall that computing $P(w|c)$ can be done through the formula $P(w|c) = \frac{e^{h^T v'_w}}{\sum_{w_i \in V} e^{h^T v'_{w_i}}}.$ softmax
- ▶ We can designate the denominator as $Z(c)$, which requires us to compute probabilities over all words in the vocabulary.
- ▶ In NCE we treat $Z(c)$ as a parameter that the model could learn.
- ▶ In some models we actually set $Z(c) = 1$ which happens to be the case even when $Z(c)$ is learned.

$$P(w|c) = e^{h^T v'_w}$$

Negative sampling (NEG)



- ▶ Negative sampling can be seen as an approximation to Noise Contrastive Estimation (NCE)
- ▶ NCE can be shown to approximate the loss of softmax as the number of samples k grows. [theoretical guarantee]
- ▶ NEG does not have such theoretical guarantees as its objective is to learn high-quality word representations instead of achieving low perplexity on a test set.
- ▶ Recall that NCE calculated the probability that a word w comes from the empirical training distribution P_{train} given a context c as follows:

$$P(y = 1|w, c) = \frac{e^{\mathbf{h}^T \mathbf{v}'_w}}{e^{\mathbf{h}^T \mathbf{v}'_w} + kQ(w)}$$

instead of P_{train}

Negative sampling



Let $k = |V|$, $q(w)$ be a uniform distribution over V

- The key difference between NEG and NCE is that NEG makes a simple approximation to this probability to get

$$P(y=1|w, c) = \frac{e^{\mathbf{h}^T \mathbf{v}'_w}}{e^{\mathbf{h}^T \mathbf{v}'_w} + 1}$$

- We calculate the logistic regression loss as follows:

$$J_\theta = - \sum_{w_i \in V} [\log \left(\frac{1}{1 + e^{-\mathbf{h}^T \mathbf{v}'_w}} \right) + \sum_{j=1}^{j=k} \log \left(\frac{1}{1 + e^{\mathbf{h}^T \mathbf{v}'_w}} \right)]$$

$$\sigma(+s) \rightarrow P(y=1|w, c)$$

$$\sigma(-s) \rightarrow P(y=0|w, c)$$

Negative sampling



- ▶ Setting $\sigma(x) = \frac{1}{1+e^{-x}}$ we have

$$J_\theta = - \sum_{w_i \in V} [\log (\sigma(\mathbf{h}^T \mathbf{v}'_{w_i}))] + \sum_{j=1}^{j=k} \log (\sigma(-\mathbf{h}^T \mathbf{v}'_{w_{ij}}))]$$

- ▶ NEG only approximates NCE which means that it will not directly optimize the likelihood of correct words which is what is needed for language modeling.
- ▶ NEG is useful for learning word embeddings.

- ▶ This part of the lecture is drawn from the paper **Neural word embedding as implicit matrix factorization**
- ▶ Both negative sampling and NCE can be shown to be performing an implicit matrix factorization, even though the methods themselves seem to have little to do with matrix factorization.
- ▶ We shall demonstrate this for Negative Sampling.

- ▶ First we rewrite the objective function of negative sampling in a different notation, which appears to perform negative sampling over a random corpus for a given word rather than over random words for a given context as in our previous slides on this topic.
- ▶ Thus we shall rewrite the objective function in the following fashion:

$$I = \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) (\log(\sigma(w.c)) + k E_{c_N \sim P_D} [\log(\sigma(-w.c_N))])$$

if $\#(w, c) > 0$, the other bracket is relevant

- ▶ In the above equation $\#(w, c)$ counts the number of times we observe a word-context pair in the corpus D . Optimizing the above objective makes observed word-context pairs have similar embeddings while scattering unobserved word-context pairs.
- ▶ Both words and their contexts are embedded into a low-dimensional space R^d resulting in word and context matrices W and C respectively.
- ▶ Let us consider the product $W \cdot C^T = M$ where $W \in |V_W| \times d$ and $C \in |V_C| \times d$ are word and context-embedding matrices where the rows in each of these matrices carries the embedding of the corresponding word and context in d -dimensional space.

Word embedding and matrix factorization



$$\begin{aligned} I &= \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) (\log(\sigma(w.c)) + \\ &\quad \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) [k E_{c_N \sim P_D} [\log(\sigma(-w.c_N))]]) \\ &= \sum_{w \in V_W} \sum_{c \in V_C} \#(w, c) (\log(\sigma(w.c)) + \\ &\quad \sum_{w \in V_W} \#(w) (k E_{c_N \sim P_D} [\log(\sigma(-w.c_N))]) \text{ uniform dist}) \\ &\quad \sum_{c \in V_C} \#(w, c) = \#(c) \\ E_{c_N \sim P_D} [\log(\sigma(-w.c_N))] &= \sum_{c_N \in V_C} \frac{\#(c_N)}{|D|} \log(\sigma(-w.c_N)) \\ &= \frac{\#(c)}{|D|} \log(\sigma(-w.c)) + \sum_{c_N \in V_C \setminus c} \frac{\#(c_N)}{|D|} \log(\sigma(-w.c_N)) \end{aligned}$$

Combining the two equations in the previous slide we have for a specific (w, c) pair the following equation:

$$l(w, c) = \#(w, c) \log(\sigma(w.c)) + k.\#(w). \frac{\#c}{|D|} \log \sigma(-w.c)$$

To optimize the objective, we can set $x = w.c$ and take $\frac{\partial l}{\partial x}$ to get

$$\frac{\partial l}{\partial x} = \#(w, c).\sigma(-x) - k.\#(w).\frac{\#(c)}{|D|}.\sigma(x)$$

To get the optimal value of x we can set the derivative to zero which we do in the next slide.

$$\frac{dl}{dx} = 0 \rightarrow \text{why?}$$

Setting the derivative to zero and simplifying gives us the following equation which is quadratic in e^x :

$$y^2 - \left(\frac{\#(w, c)}{k \cdot \#(w) \cdot \frac{\#(c)}{|D|}} - 1 \right) e^x - \frac{\#(w, c)}{k \cdot \#(w) \cdot \frac{\#(c)}{|D|}} = 0$$

This quadratic equation (setting $y = e^x$) has two solutions including $y = -1$ which is invalid given the definition of y . The other solution is

$$y = \frac{\#(w, c)}{k \cdot \#(w) \cdot \frac{\#(c)}{|D|}} = \frac{\#(w, c) \cdot |D|}{\#(w) \cdot \#(c)} \cdot \frac{1}{k}$$

$$y = e^{w \cdot c}$$

Finally we can obtain the following:

$$\begin{aligned}\underline{w.c} &= \log \frac{\#(w, c).|D|}{\#(w).\#(c)} \cdot \frac{1}{k} \\ w.c &= \log \left[\frac{\#(w, c).|D|}{\#(w).\#(c)} \right] - \log k \\ &= \underline{\underline{PMI(w, c)}}\end{aligned}$$

The matrix M that Negative Sampling is factorizing has entries

$M_{ij} = W_i.C_j = w_i.c_j = PMI(w_i, c_j) - k$. Thus Negative Sampling is factorizing a PMI matrix.



A hand-drawn diagram illustrating the formula for PMI. It shows a large bracket over the term $\#(w, c)$ with the label "PMI". Below it, a horizontal line with arrows at both ends connects the terms $\#(w)$ and $\#(c)$, with the label "PMI" written between them. To the left of the line, there is a small circle with the letter "i".



Graph Neural Networks (S*-**_AIMLCZG514)

BITS Pilani
Pilani Campus

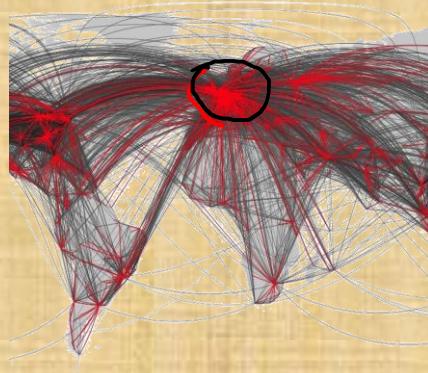


Module V

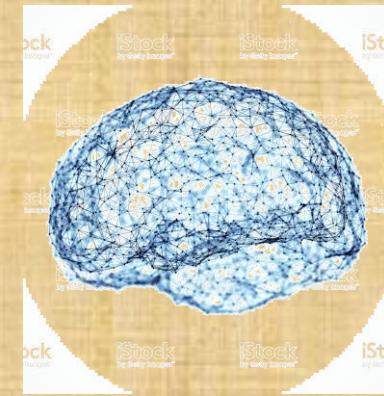
Data as Graphs - Explicit



Social Graphs



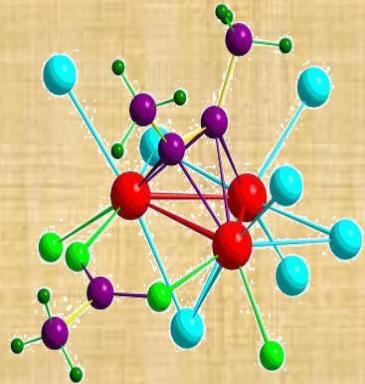
Transportation Graphs



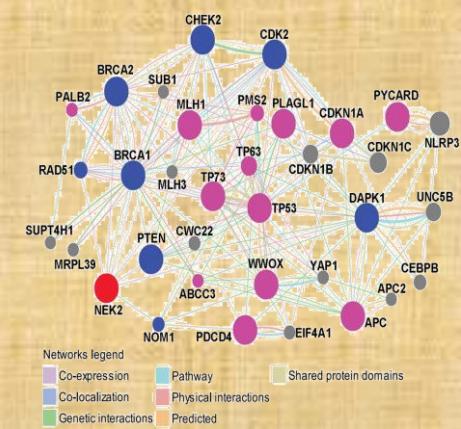
Brain Graphs



Web Graphs

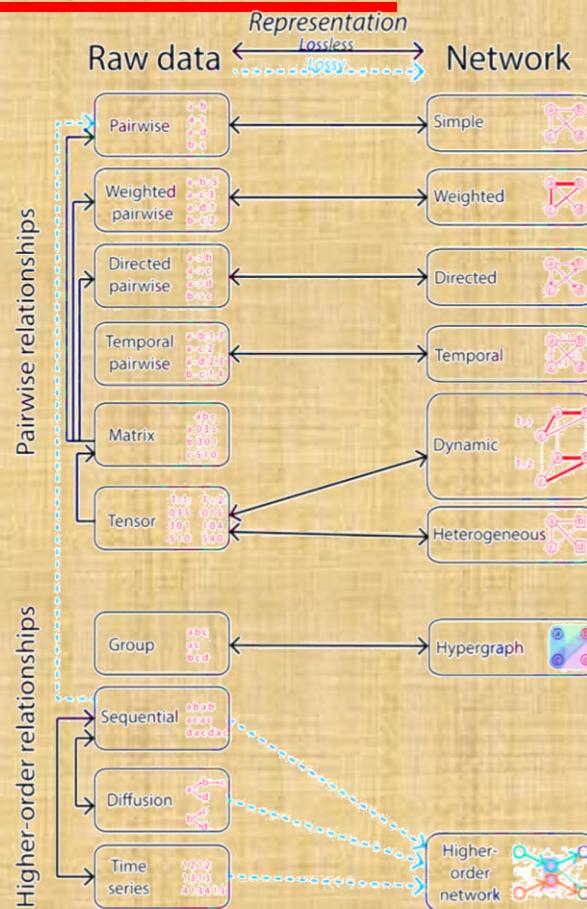
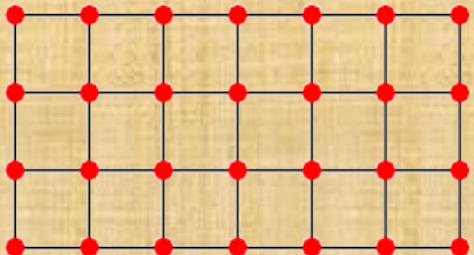


Molecular Graphs



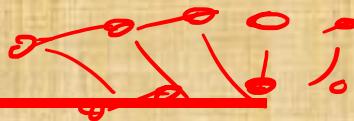
Gene Graphs

Data as Graphs - Implicit



Jian Xu. Representing Big Data as Networks. PhD Dissertation, University of Notre Dame

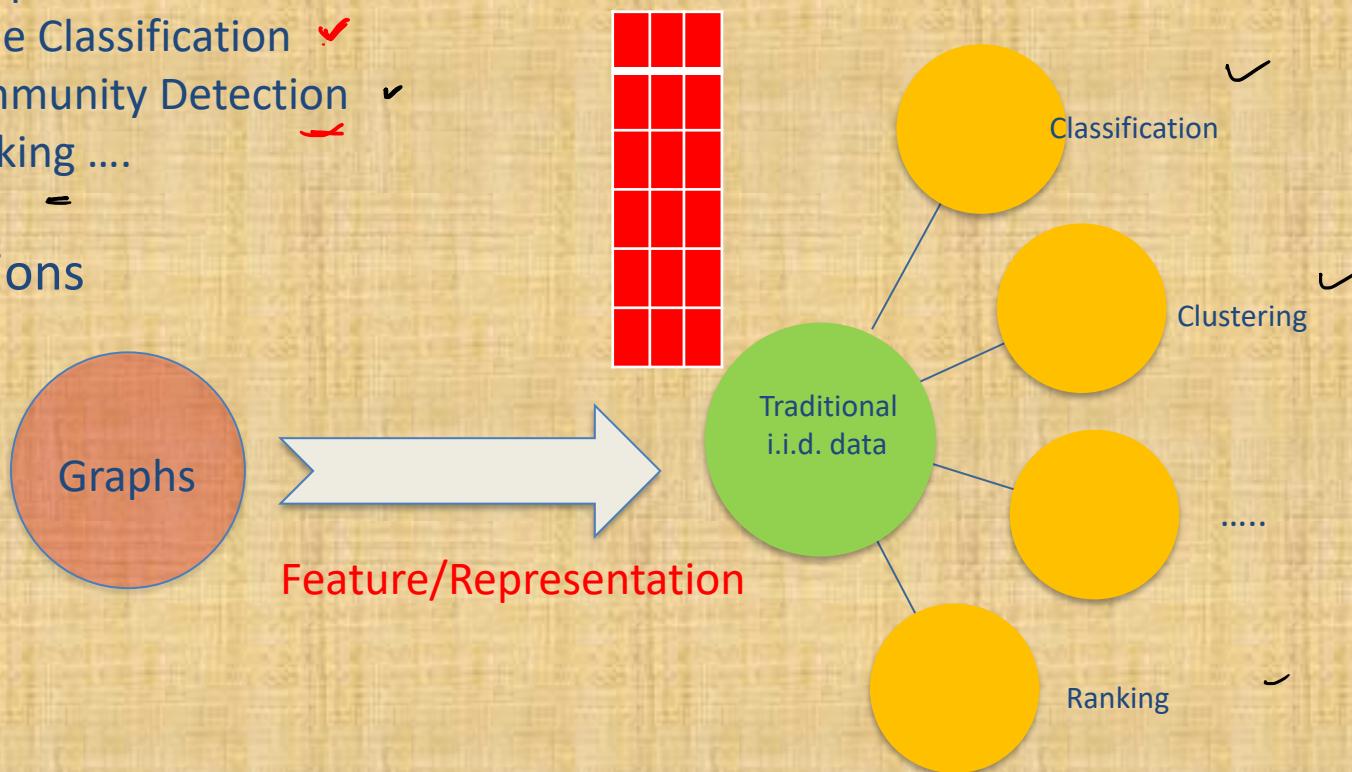
ML on Graphs



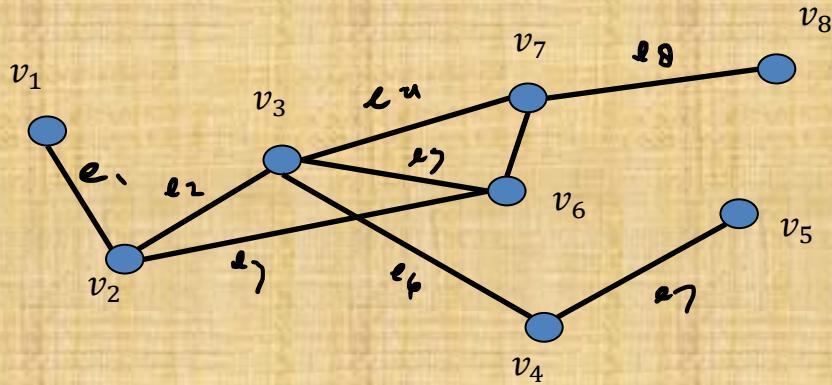
Numerous real-world problems can be summarized as a set of tasks on graphs

- Link prediction ✓
- Node Classification ✓
- Community Detection ↗
- Ranking ↗

ML solutions

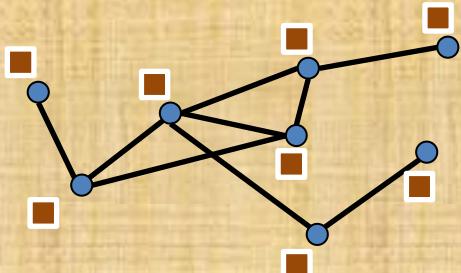


Graphs and Graph Signals



- ✓ $\mathcal{V} = \{v_1, \dots, v_N\}$
- ✓ $\mathcal{E} = \{e_1, \dots, e_M\}$
- ✓ $\mathcal{G} = \{\underline{\mathcal{V}}, \underline{\mathcal{E}}\}$

Graphs and Graph Signals



$$\mathcal{V} = \{v_1, \dots, v_N\}$$

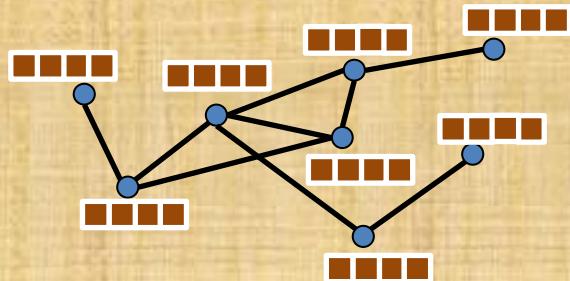
$$\mathcal{E} = \{e_1, \dots, e_M\}$$

$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

Graph Signal: $f : \mathcal{V} \rightarrow \mathbb{R}^N$

$$\mathcal{V} \longrightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Graphs and Graph Signals



$$\mathcal{V} = \{v_1, \dots, v_N\}$$

$$\mathcal{E} = \{e_1, \dots, e_M\}$$

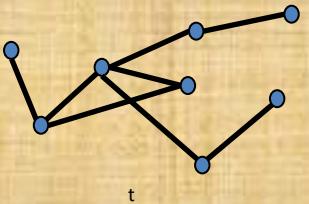
$$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$$

Graph Signal: $f : \mathcal{V} \rightarrow \underbrace{\mathbb{R}^{N \times d}}$

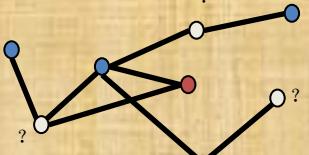
$$\mathcal{V} \rightarrow \begin{bmatrix} f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \\ f(8) \end{bmatrix}$$

Tasks on Graph-Structured Data

Node-level



Link Prediction



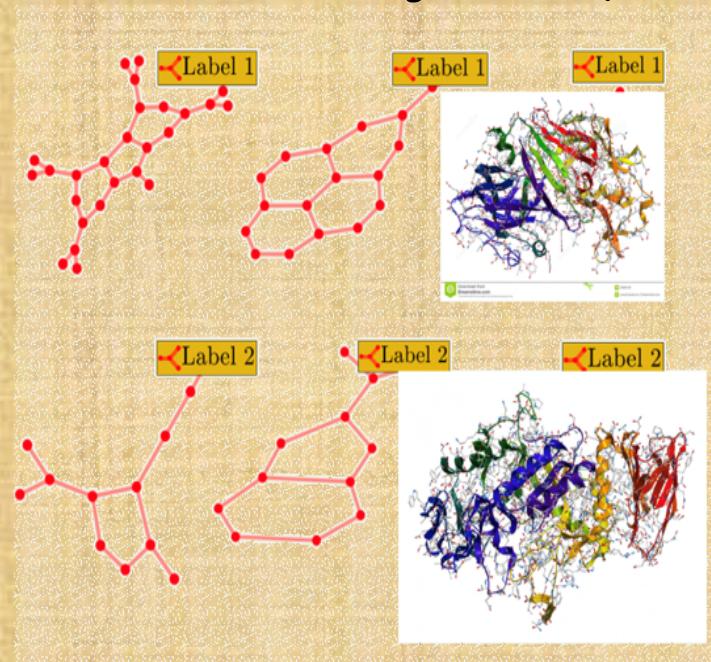
classifies

→ Node
entity
(single large
graph)

graph & level.

Graph-level

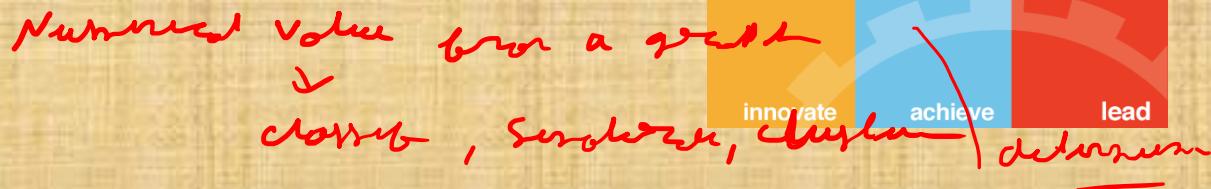
(a large Number of
small graphs)
Graph Classification
(predict the
of whole
sub graph)





Embedding

Embedding



- ❖ An **embedding** is a relatively low-dimensional space into which you can translate high-dimensional vectors.
- ❖ It captures some of the semantics of the input by placing semantically similar inputs close together in the embedding space and is a vector (list) of floating point numbers.
- ❖ The distance between two vectors measures their relatedness. Small distances suggest high relatedness and large distances suggest low relatedness.
- ❖ It is used as input to downstream machine learning tasks such as *node classification*, *link prediction* and *kNN similarity graph construction*.

Embedding

- ❖ Embeddings should capture the graph topology, relationships between nodes and further relevant information. How the embeddings should capture this inherent information of the graph is not fixed. It depends on the questions we ask about the network.
- ❖ Here is a chronological list of research papers where you can check them out:
 1. [Distributed large-scale natural graph factorization, Ahmed et al., 2013;](#)
 2. [DeepWalk: online learning of social representations, Perozzi et al., 2014;](#)
 3. [Deep Neural Networks for Learning Graph Representations, Cao, et al., 2015;](#)
 4. [LINE: Large-scale Information Network Embedding, Tang, et al., 2015;](#)
 5. [node2vec: Scalable Feature Learning for Networks, Grover and Leskovec, 2016;](#)
 6. [Asymmetric Transitivity Preserving Graph Embedding, Ou et al., 2016.](#)

Embedding

- ❖ Uptill now, we have discussed the diversity of methods for node embeddings largely stem from the use of different decoders (dec), similarity measures ($S(u, v)$), and loss functions (\mathcal{L}).
- ❖ The decoder gives a score between a pair of node embeddings; ↴
- ❖ The similarity function defines what kind of node-node similarity we are trying to decode;
- ❖ The loss function tells us how to evaluate the discrepancy between the output of the decoder and the ground truth similarity measure.

RANDOM WALK EMBEDDINGS (node2vec)



- ❖ Slides 2 – 5 from Lecture 5

$$\mathcal{L} = \sum_{(u,v) \in \mathcal{D}} -\log(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - \gamma \mathbb{E}_{v_n \sim P_n(V)} [\log(-\sigma(\mathbf{z}_u^\top \mathbf{z}_{v_n}))].$$

- ❖ Here, we use σ to denote the logistic function,
- ❖ $P_n(V)$ to denote a distribution over the set of nodes V , and
- ❖ we assume that $\gamma > 0$ is a hyperparameter.

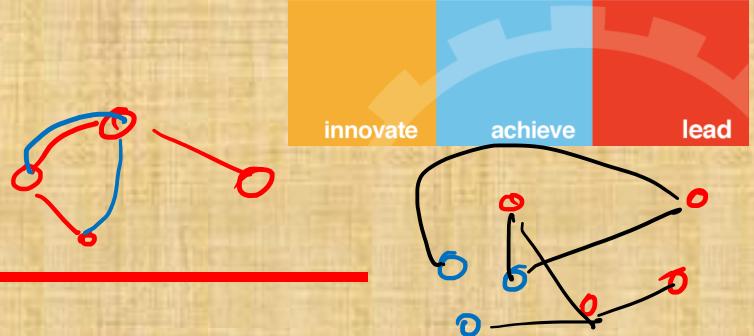
In practice, $P_n(V)$ is often defined to be a uniform distribution, and the expectation is approximated using Monte Carlo sampling.



Multi-Relational Data and Knowledge Graphs

$e \in (u, v)$

Knowledge Graph

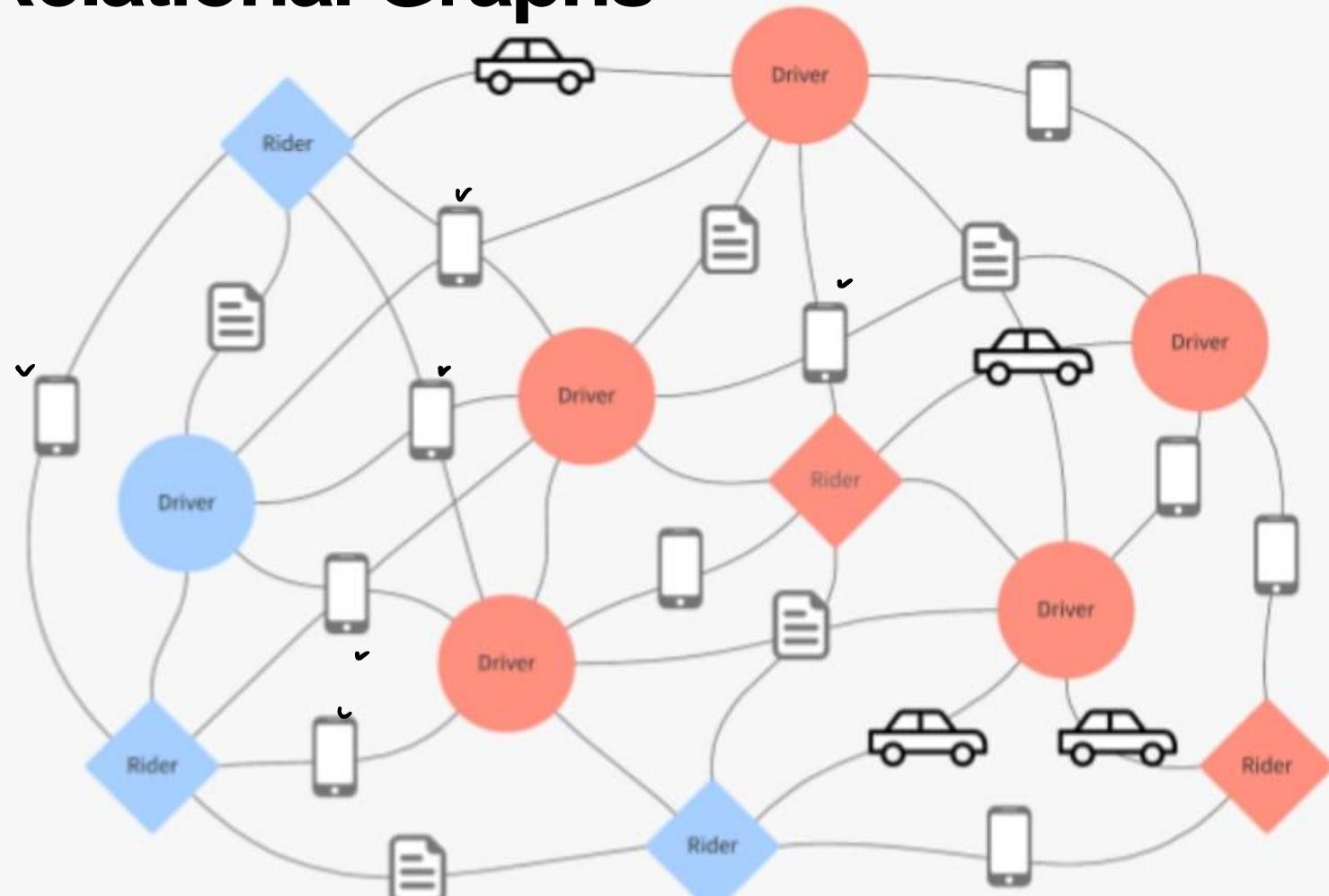


- ❖ we are given a multi-relational graph $G = (V, E)$, where the edges are defined as tuples $e = (\underline{u}, \tau \underline{v})$ indicating the presence of a particular relation $\tau \in T$ holding between two nodes.
- ❖ Such multirelational graphs are often referred to as *knowledge graphs*, since we can interpret the tuple $(\underline{u}, \tau \underline{v})$ as specifying that a particular “fact” holds between the two nodes \underline{u} and \underline{v} .

Multi-Relational Graphs

Lor

Football Plat



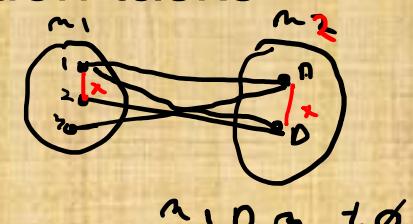
1. r-GAT: Relational Graph Attention Network for Multi-Relational Graphs Meiqi Chen at. el. 2021
2. Graph Heterogeneous Multi-Relational Recommendation Chong Chen at. el 2021
3. Fraud Detection: Using Relational Graph Learning to Detect Collusion: <https://www.uber.com/en-LK/blog/fraud-detection/>

Knowledge Graph



- ❖ e.g. in a biomedical knowledge graph, we might have an edge type $\xi = \text{TREATS}$ and the edge (u, TREATS, v) could indicate that the drug associated with node u treats the disease associated with node v .
- ❖ Generally, the goal in knowledge graph completion is to predict missing edges in the graph, i.e., relation prediction, but there are also examples of node classification tasks using multi-relational graphs¹
→ (❖ Bipartite and (multi) k-partite Graphs)

Not all knowledge graph completion methods rely on embeddings, and we will not cover every variation of embedding method²



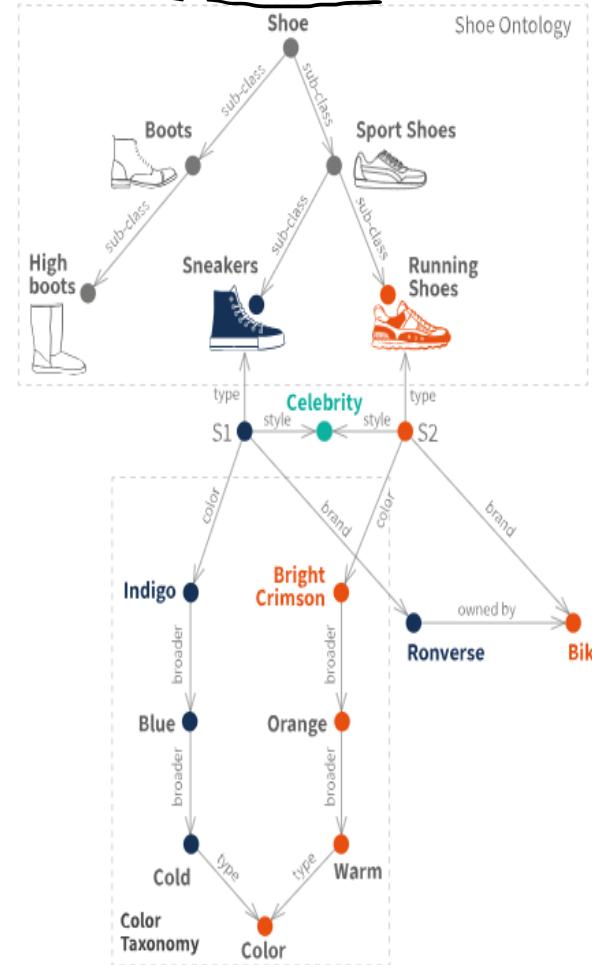
- 1. Modeling Relational Data with Graph Convolutional Networks_2017 (<https://arxiv.org/abs/1703.06103>)
- 2. A review of relational machine learning for knowledge graph_2016 (<https://arxiv.org/abs/1503.00759>)

Knowledge Graph

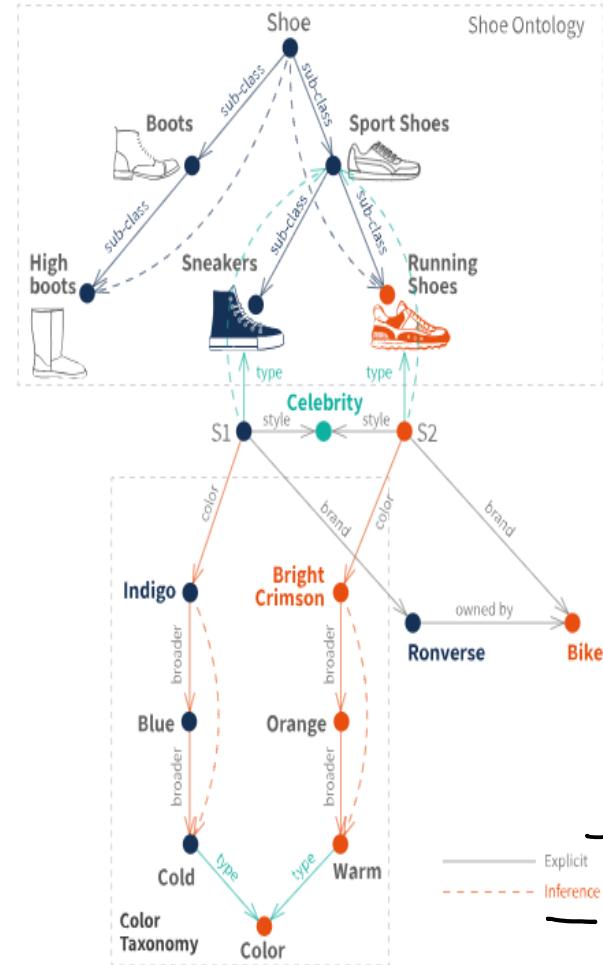
Plain Graph



Knowledge Graph



Knowledge Graph with Inference





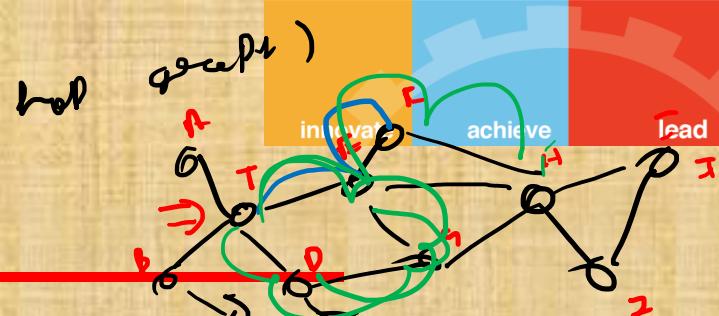
$$(u, \tau, v) \sim_{G,R}$$

Multi-Relational Data

$$1\text{-hop}(T) = \{A, B, D, F\}$$

$$2\text{-hop}(T) = \{F, C, G, K\}$$

$$3\text{-hop}(T) = \{C, D, E\}$$



❖ In the multi-relational setting, we will also see a diversity of decoders and loss functions.

❖ Nearly all multi-relational embedding methods simply define the similarity measure directly based on the adjacency tensor. i.e., all the methods here assume that we are trying to reconstruct immediate (multi-relational) neighbors from the low-dimensional embeddings.

This is due to the difficulty of defining higher-order neighborhood relationships in multi-relational graphs, as well as the fact that most multi-relational embedding methods were specifically designed for relation prediction.

RECONSTRUCTING MULTI-RELATIONAL DATA



- ❖ Given the embeddings $\underline{z_u}$ and $\underline{z_v}$ of two nodes,
 - ❖ Our goal is to reconstruct the relationship between these nodes in the presence of multiple different types of edges.
- ❖ we augment our decoder to make it multi-relational by accepting a pair of node embeddings as well as a relation type, i.e., $DEC: R^d \times \mathbb{R} \times R^d \rightarrow R^+$.
Dec: $P^\alpha \times P^\alpha \rightarrow P^\alpha$
- ❖ We can interpret the output of this decoder, i.e., $DEC(z_u \cdot \tau \cdot z_v)$, as the likelihood that the edge $(\underline{u}, \underline{\tau}, v)$ exists in the graph

RECONSTRUCTING MULTI-RELATIONAL DATA

$$\underline{z}_u^T \cdot \underline{z}_v$$

- ❖ The simplest and earliest approaches to learning multirelational embeddings “RESCAL¹” defined the decoder as: $\text{DEC}: (\underline{u}, \tau, \underline{v}) = \underline{z}_u^T \underline{R}_\tau \underline{z}_v$
 - ❖ here $R_\tau \in \mathbb{R}^{d \times d}$ is a learnable matrix specific to relation $\tau \in \mathbb{R}$
- ❖ With this decoder, we could train our embedding matrix \mathbf{Z} and our relation matrices $\mathbb{R}_\tau, \forall \tau \in \mathbb{R}$ using a basic reconstruction loss:

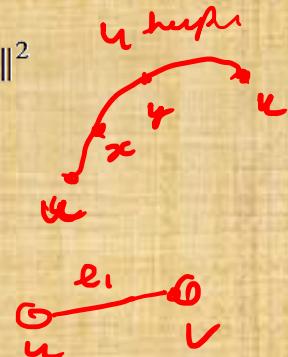
$$\begin{aligned}\underline{\mathcal{L}} &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\text{DEC}(u, \tau, v) - \mathcal{A}[u, \tau, v]\|^2 \\ &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\underline{z}_u^T \underline{R}_\tau \underline{z}_v - \mathcal{A}[u, \tau, v]\|^2,\end{aligned}$$

1. M. Nickel, V. Tresp, and H. Kriegel. A three-way model for collective learning on multirelational data.

RECONSTRUCTING MULTI-RELATIONAL DATA

$$\begin{aligned}\mathcal{L} &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\text{DEC}(u, \tau, v) - \mathcal{A}[u, \tau, v]\|^2 \\ &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v - \mathcal{A}[u, \tau, v]\|^2,\end{aligned}$$

- ❖ where $\mathcal{A} \in \mathbb{R}^{|V| \times |\mathcal{R}| \times |V|}$ is the adjacency tensor for the multi-relational graph.



- ❖ An adjacency tensor is a generalization of the concept of an adjacency matrix from graphs to hypergraphs, in which hyperedges may be of arbitrary arity.
- ❖ A hypergraph is a graph in which hyperedges (generalized edges) can connect to a subset of vertices/nodes rather than two vertices/nodes.
- ❖ The edges (also known as hyperedges) of a hypergraph are arbitrary nonempty sets of vertices! A k-hypergraph has all such hyperedges connecting exactly k vertices; a normal graph is thus a 2-hypergraph (as one edge connects 2 vertices).
- ❖ The rank of the adjacency tensor is equal to the arity of the hyperedges in the hypergraph.

RECONSTRUCTING MULTI-RELATIONAL DATA



$$\begin{aligned}\mathcal{L} &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\text{DEC}(u, \tau, v) - \mathcal{A}[u, \tau, v]\|^2 \\ &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v - \mathcal{A}[u, \tau, v]\|^2,\end{aligned}$$

- ❖ where $A \in \mathbb{R}^{|V| \times |\mathcal{R}| \times |V|}$ is the adjacency tensor for the multi-relational graph.
- ❖ If we were to optimize the loss function, we would in fact be performing a kind of tensor factorization.
- ❖ This idea of factorizing a tensor thus generalizes the matrix factorization approaches discussed earlier.

LOSS FUNCTIONS

$$\begin{aligned}\mathcal{L} &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\text{DEC}(u, \tau, v) - \mathcal{A}[u, \tau, v]\|^2 \\ &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v - \mathcal{A}[u, \tau, v]\|^2,\end{aligned}$$

- ❖ The two key ingredients for a multi-relational node embedding method are the **decoder** and the **loss function**.
- ❖ There are two major problems with this loss function:
 1. It is extremely expensive to compute. The nested sums require $O(|V|^2 |\mathbb{R}|)$ operations, and this computation time will be prohibitive in many large graphs. Moreover, since many multi-relational graphs are sparse i.e., $|\mathcal{E}| \ll |V|^2 |\mathbb{R}|$, we would ideally want a loss function that is $O(|\mathcal{E}|)$.

LOSS FUNCTIONS

$$\begin{aligned}\mathcal{L} &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\text{DEC}(u, \tau, v) - \mathcal{A}[u, \tau, v]\|^2 \\ &= \sum_{u \in \mathcal{V}} \sum_{v \in \mathcal{V}} \sum_{\tau \in \mathcal{R}} \|\mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v - \mathcal{A}[u, \tau, v]\|^2,\end{aligned}$$

- ❖ The two key ingredients for a multi-relational node embedding method are the ***decoder*** and the ***loss function***.
- ❖ There are two major problems with this loss function:
 2. (more subtle) the goal is to decode the adjacency tensor from the low-dimensional node embeddings, and (in most cases) this tensor will contain only binary values, but the mean-squared error in above Equation is not well suited to such a binary comparison. In fact, the mean-squared error is a natural loss for *regression* whereas our target is something closer to *classification on edges*.

Cross-Entropy with Negative Sampling



- ❖ One popular loss function that is both efficient and suited to our task is the cross-entropy loss with negative sampling.

$$\mathcal{L} = \sum_{\substack{(u, \tau, v) \in \mathcal{E}}} -\log(\sigma(\text{DEC}(z_u, \tau, z_v))) - \gamma \mathbb{E}_{v_n \sim P_{n,u}(V)} [\log (\sigma (-\text{DEC}(z_u, \tau, z_{v_n})))],$$

- ❖ Here, we use σ to denote the logistic function,
- ❖ $P_{n,u}(V)$ to denote a “**Negative Sampling**” distribution over the set of nodes V (which might depend on u).
- ❖ We assume that $\gamma > 0$ is a hyperparameter.

This is essentially the same loss as we saw for *node2vec*, but here we are considering general multi-relational decoders. ✓

Cross-Entropy with Negative Sampling

↓ correct middle yes

$$\mathcal{L} = \sum_{(u, \tau, v) \in \mathcal{E}} -\log(\sigma(\text{DEC}(z_u, \tau, z_v))) + \gamma \mathbb{E}_{v_n \sim P_{n,u}(\mathcal{V})} [\log (\sigma (-\text{DEC}(z_u, \tau, z_{v_n})))],$$

*No edge in the graph
and we add products
No*

- ❖ We call this a *cross-entropy loss* because it is derived from the standard binary cross-entropy loss.
- ❖ Since we are feeding the output of the decoder to a logistic function, we obtain normalized scores in [0,1] that can be interpreted as probabilities
 1. First part log-likelihood that we predict “true” for an edge that does actually exist in the graph.
 2. Second part equals the expected log-likelihood that we correctly predict “false” for an edge that does not exist in the graph

Max-Margin Loss

- ❖ The other popular loss function used for multi-relational node embedding is the margin loss:

$$\mathcal{L} = \sum_{(u, \tau, v) \in \mathcal{E}} \sum_{v_n \in \mathcal{P}_{n,u}} \max(0, -\text{DEC}(\underline{\mathbf{z}_u}, \tau, \underline{\mathbf{z}_v}) + \text{DEC}(\mathbf{z}_u, \tau, \mathbf{z}_{v_n}) + \Delta).$$

- ❖ In this loss we are again comparing the decoded score for a true pair compared to a negative sample a strategy often termed *contrastive estimation*.

- ❖ *Noise Contrastive Estimation (NCE) is a powerful parameter estimation method for log-linear models, which avoids calculation of the partition function or its derivatives at each training step, a computationally demanding step in many cases. It is closely related to negative sampling methods, now widely used in NLP.*

MULTI-RELATIONAL DECODERS

- ❖ We have learned two most popular loss functions for learning multi-learning node embeddings.
- ❖ These losses can be combined with various different decoder functions, and we turn our attention to the definition of these decoders now.
- ❖ There are few other decoder functions that can be used for multi-relational data (*almost same as it was in shallow embedding algorithms for simple graphs*)

Cross-Entropy with Negative Sampling



Name	Decoder	Relation Parameters
RESCAL	$\mathbf{z}_u^\top \mathbf{R}_\tau \mathbf{z}_v$	$\mathbf{R}_\tau \in \mathbb{R}^{d \times d}$
TransE	$- \ \mathbf{z}_u + \mathbf{r}_\tau - \mathbf{z}_v\ $	$\mathbf{r}_\tau \in \mathbb{R}^d$
TransX	$- \ g_{1,\tau}(\mathbf{z}_u) + \mathbf{r}_\tau - g_{2,\tau}(\mathbf{z}_v)\ $	$\mathbf{r}_\tau \in \mathbb{R}^d, g_{1,\tau}, g_{2,\tau} \in \mathbb{R}^d \rightarrow \mathbb{R}^d$
DistMult	$\langle \mathbf{z}_u, \mathbf{r}_\tau, \mathbf{z}_v \rangle$	$\mathbf{r}_\tau \in \mathbb{R}^d$
ComplEx	$\text{Re}(\langle \mathbf{z}_u, \mathbf{r}_\tau, \bar{\mathbf{z}}_v \rangle)$	$\mathbf{r}_\tau \in \mathbb{C}^d$
RotatE	$- \ \mathbf{z}_u \circ \mathbf{r}_\tau - \mathbf{z}_v\ $	$\mathbf{r}_\tau \in \mathbb{C}^d$

Summary of some popular decoders used for multi-relational data

- ❖ There are $O(d^2)$ parameters for each relation type in RESCAL, which means that relations require an order of magnitude more parameters to represent, compared to entities.
- ❖ More popular modern decoders aim to use only $O(d)$ parameters to represent each relation.



References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3



Thank You



Graph Neural Networks (S*-**_AIMLCZG514)

BITS Pilani
Pilani Campus



Module VI



BITS Pilani
Pilani Campus



Graph Neural Network

The Graph Neural Network Model



- ❖ Till now we have discussed various approaches for learning low-dimensional embeddings of the nodes in a graph using *shallow embedding*.
- ❖ Here we will turn our focus to more complex encoder models
- ❖ We will introduce the *graph neural network (GNN)* formalism, which is a general framework for defining deep neural networks on graph data.
 - ❖ The key idea is that we want to generate representations of nodes that actually depend on
 - ❖ the structure of the graph, as well as
 - ❖ any feature information we might have.

The Graph Neural Network Model (*challenges*)



- ❖ The primary **challenge** in developing complex encoders for graph-structured data is that our usual deep learning toolbox does not apply.
 - ❖ For example,
 - ❖ Convolutional neural networks (CNNs) are well defined only over grid-structured inputs (e.g., images),
 - ❖ Recurrent neural networks (RNNs) are well defined only over sequences (e.g., text).
- ❖ To define a deep neural network over general graphs, we need to define a new kind of deep learning architecture.

Some Key terms

❖ Multi-layer Perceptron

- ❖ It is fully connected dense layers, which transform any input dimension to the desired dimension. A multi-layer perception is a neural network that has multiple layers. To create a neural network we combine neurons together so that the outputs of some neurons are inputs of other neurons.

❖ *permutation invariant*

$$f(PAP^T) = f(A) \quad (\text{Permutation Invariance})$$

- ❖ the function does not depend on the arbitrary ordering of the rows/columns in the adjacency matrix,

❖ permutation equivariance

$$f(PAP^T) = Pf(A) \quad (\text{Permutation Equivariance}),$$

- ❖ the output of f is permuted in a consistent way when we permute the adjacency matrix. (*The shallow encoders we discussed earlier are an example of permutation equivariant functions.*)

❖ Ensuring invariance or equivariance is a key challenge when we are learning over graphs,

The Graph Neural Network Model



- ❖ To define a deep neural network over graphs would be to simply use the adjacency matrix as input to a deep neural network.
- ❖ To generate an embedding of an entire graph we could simply flatten the adjacency matrix and feed the result to a multi-layer perceptron (MLP):

$$\mathbf{z}_G = \text{MLP}(\mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \dots \oplus \mathbf{A}[|\mathcal{V}|]),$$

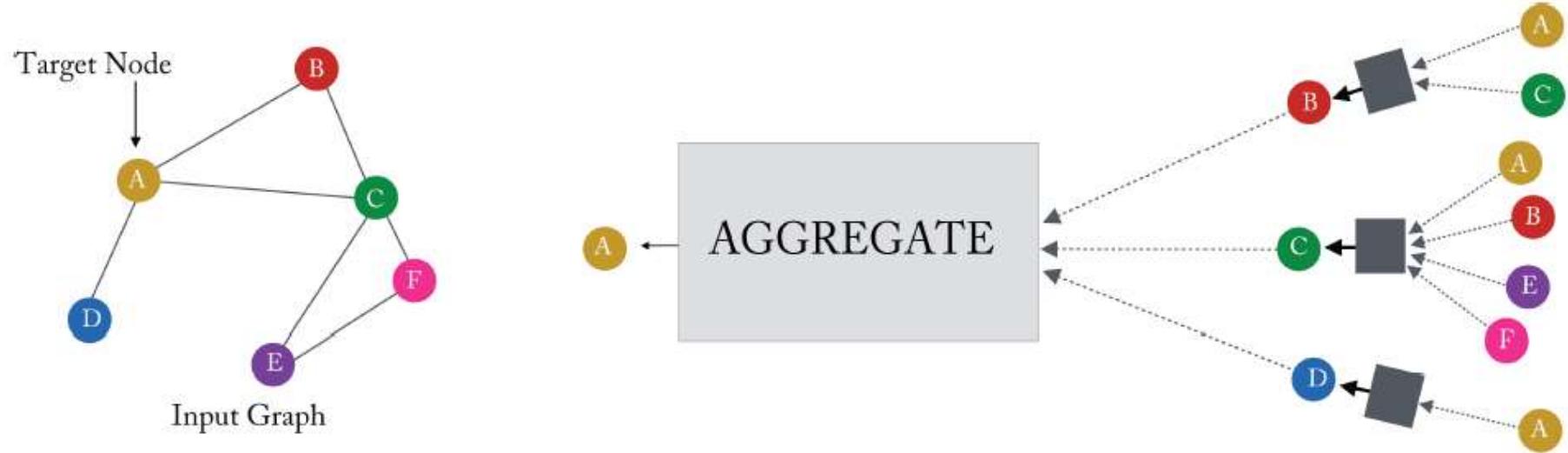
- ❖ Here $A[i] \in R^{|\mathcal{V}|}$, is row in adjacency matrix and \oplus is vector concatenation.
- ❖ The issue here is, it is not permutation invariant, as it depends on the arbitrary ordering of nodes of graph (*isomorphic problem*), which is the basic requirement for designing neural network over graph.

NEURAL MESSAGE PASSING

- ❖ The defining feature of a basic GNN model is that, it uses a form of *neural message passing*, in which vector messages, are exchanged between nodes and updated using neural networks.
- ❖ The same fundamental GNN model has been derived as
 1. Generalization of convolutions to non-Euclidean data
 2. Differentiable variant of belief propagation
 3. Analogy to classic graph isomorphism tests

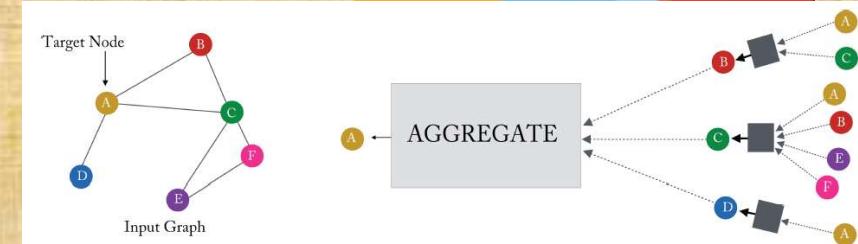
1. J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. In *ICLR*, 2014.
2. H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *ICML*, 2016
3. W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NeurIPS*, 2017b.

NEURAL MESSAGE PASSING



Overview of how a single node aggregates messages from its local neighborhood

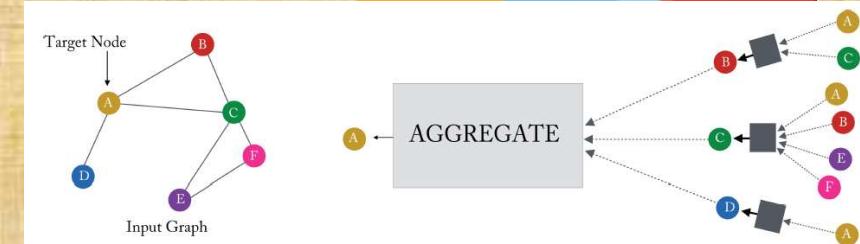
NEURAL MESSAGE PASSING



- ❖ The model aggregates messages from A's local graph neighbors (i.e., B, C, and D), and
- ❖ in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on.
- ❖ This visualization shows a **two-layer** version of a message-passing model.
- ❖ This is an iterative process.

Notice that the computation graph of the GNN forms a tree structure by unfolding the neighborhood around the target node.

NEURAL MESSAGE PASSING



- ❖ During each message-passing iteration in a GNN, a *hidden embedding* $h_u^{(k)}$ corresponding to each node $u \in V$ is updated according to information aggregated from u's graph neighborhood $N(u)$.

$$\begin{aligned} h_u^{(k+1)} &= \text{UPDATE}^{(k)} \left(h_u^{(k)}, \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in N(u)\}) \right) \\ &= \text{UPDATE}^{(k)} \left(h_u^{(k)}, m_{N(u)}^{(k)} \right), \end{aligned}$$

- ❖ **update** and **aggregate** are arbitrary differentiable functions (i.e., neural networks) and
- ❖ $m_{N(u)}$ is the “message” that is aggregated from u 's graph neighborhood $N(u)$.

References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3



Network Measures

SOCIAL MEDIA MINING



Dear instructors/users of these slides:

Please feel free to include these slides in your own material, or modify them as you see fit. If you decide to incorporate these slides into your presentations, please include the following note:

R. Zafarani, M. A. Abbasi, and H. Liu, *Social Media Mining: An Introduction*, Cambridge University Press, 2014.
Free book and slides at **<http://socialmediamining.info/>**

or include a link to the website:
<http://socialmediamining.info/>

Why Do We Need Measures?

- Who are the central figures (influential individuals) in the network?
 - **Centrality**
- What interaction patterns are common in friends?
 - **Reciprocity and Transitivity**
 - **Balance and Status**
- Who are the like-minded users and how can we find these similar individuals?
 - **Similarity**
- To answer these and similar questions, one first needs to define measures for quantifying **centrality**, **level of interactions**, and **similarity**, among others.

Centrality

Centrality defines how important a node is within a network

Centrality in terms of those who you are connected to

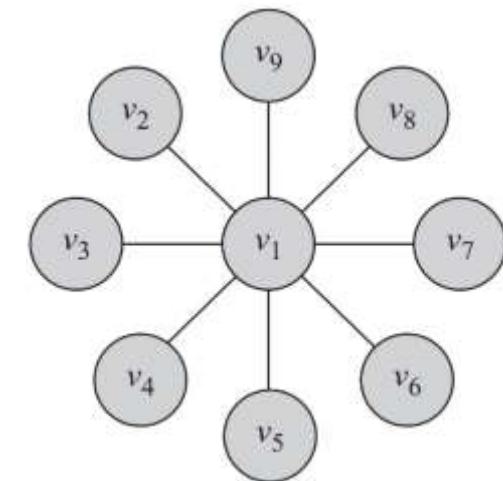
Degree Centrality

- **Degree centrality:** ranks nodes with more connections higher in terms of centrality

$$C_d(v_i) = d_i$$

- d_i is the degree (number of friends) for node v_i
 - i.e., the number of length-1 paths (can be generalized)

In this graph, degree centrality for node v_1 is $d_1=8$ and for all others is $d_j = 1, j \neq 1$



Degree Centrality in Directed Graphs

- In directed graphs, we can either use the in-degree, the out-degree, or the combination as the degree centrality value:
- In practice, mostly in-degree is used.

$$C_d(v_i) = d_i^{\text{in}} \quad (\textit{prestige})$$

$$C_d(v_i) = d_i^{\text{out}} \quad (\textit{gregariousness})$$

$$C_d(v_i) = d_i^{\text{in}} + d_i^{\text{out}}$$

d_i^{out} is the number of outgoing links for node v_i

Normalized Degree Centrality

- Normalized by the maximum possible degree

$$C_d^{\text{norm}}(v_i) = \frac{d_i}{n-1}$$

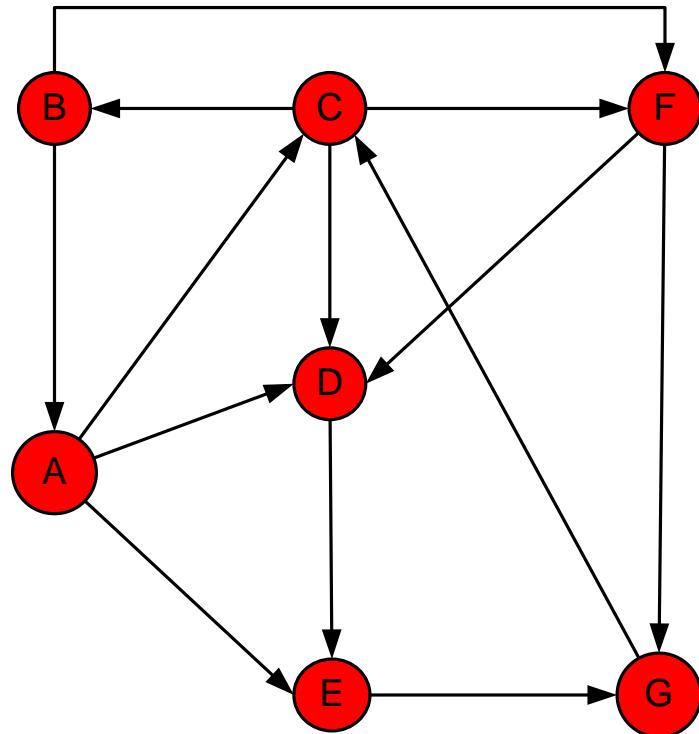
- Normalized by the maximum degree

$$C_d^{\text{max}}(v_i) = \frac{d_i}{\max_j d_j}$$

- Normalized by the degree sum

$$C_d^{\text{sum}}(v_i) = \frac{d_i}{\sum_j d_j} = \frac{d_i}{2|E|} = \frac{d_i}{2m}$$

Degree Centrality (Directed Graph) Example



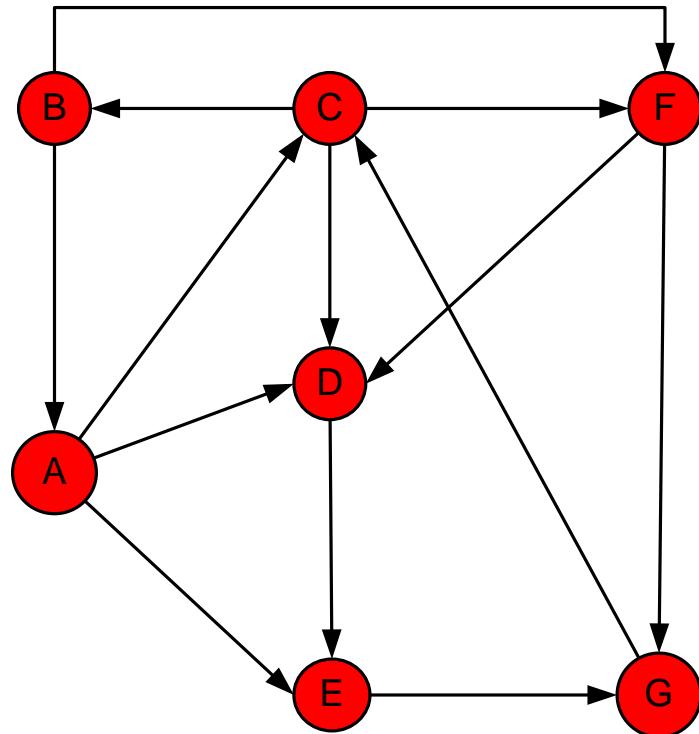
Wrt out degree

Node	In-Degree	Out-Degree	Centrality	Rank
A	1	3	1/2	1
B	1	2	1/3	3
C	2	3	1/2	1
D	3	1	1/6	5
E	2	1	1/6	5
F	2	2	1/3	3
G	2	1	1/6	5

Normalized by the maximum possible degree

$$C_d^{\text{norm}}(v_i) = \frac{d_i}{n-1}$$

Degree Centrality (Directed Graph) Example



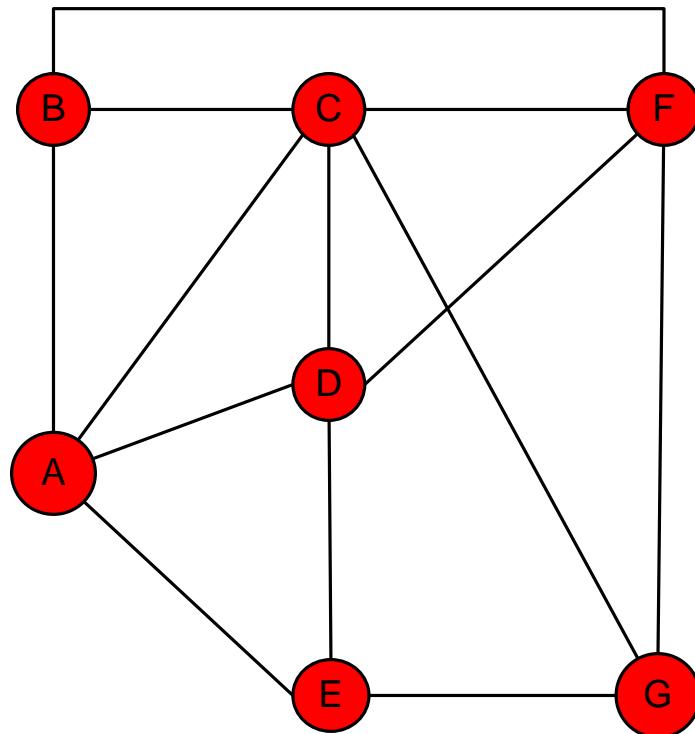
Wrt in degree

Node	In-Degree	Out-Degree	Centrality	Rank
A	1	3	1/6	6
B	1	2	1/6	6
C	2	3	1/3	2
D	3	1	1/2	1
E	2	1	1/3	2
F	2	2	1/3	2
G	2	1	1/3	2

Normalized by the maximum possible degree

$$C_d^{\text{norm}}(v_i) = \frac{d_i}{n-1}$$

Degree Centrality (undirected Graph) Example



Node	Degree	Centrality	Rank
A	4	2/3	2
B	3	1/2	5
C	5	5/6	1
D	4	2/3	2
E	3	1/2	5
F	4	2/3	2
G	3	1/2	5

Eigenvector Centrality

- Having more friends does not by itself guarantee that someone is more important
 - Having more **important friends** provides a stronger signal
- Eigenvector centrality generalizes degree centrality by incorporating the importance of the neighbors (undirected)
- For directed graphs, we can use incoming or outgoing edges



Phillip Bonacich

Formulation

- Let's assume the eigenvector centrality of a node is $c_e(v_i)$ (**unknown**)
- We would like $c_e(v_i)$ to be higher when **important** neighbors (node v_j with higher $c_e(v_j)$) point to us
 - Incoming or outgoing neighbors?
 - For incoming neighbors $A_{j,i} = 1$
- We can assume that v_i 's centrality is the summation of its neighbors' centralities

$$c_e(v_i) = \sum_{j=1}^n A_{j,i} c_e(v_j)$$

- Is this summation bounded?

- We have to normalize!

λ : some fixed constant

$$c_e(v_i) = \frac{1}{\lambda} \sum_{j=1}^n A_{j,i} c_e(v_j)$$

Eigenvector Centrality (Matrix Formulation)

$$c_e(v_i) = \frac{1}{\lambda} \sum_{j=1}^n A_{j,i} c_e(v_j)$$

- Let $\mathbf{C}_e = (C_e(v_1), C_e(v_2), \dots, C_e(v_n))^T$
- $\rightarrow \lambda \mathbf{C}_e = A^T \mathbf{C}_e$
- This means that C_e is an eigenvector of adjacency matrix A^T (or A when undirected) and λ is the corresponding eigenvalue
- We will choose the eigenvalue-eigenvector pair which is largest

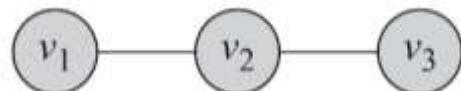
Eigenvector Centrality, cont.

Theorem 1 (Perron-Frobenius Theorem). *Let $A \in \mathbb{R}^{n \times n}$ represent the adjacency matrix for a [strongly] connected graph or $A : A_{i,j} > 0$ (i.e. a positive n by n matrix). There exists a positive real number (Perron-Frobenius eigenvalue) λ_{\max} , such that λ_{\max} is an eigenvalue of A and any other eigenvalue of A is strictly smaller than λ_{\max} . Furthermore, there exists a corresponding eigenvector $\mathbf{v} = (v_1, v_2, \dots, v_n)$ of A with eigenvalue λ_{\max} such that $\forall v_i > 0$.*

So, to compute eigenvector centrality of A ,

1. We compute the eigenvalues of A
2. Select the largest eigenvalue λ
3. The corresponding eigenvector of λ is \mathbf{C}_e .
4. Based on the Perron-Frobenius theorem, all the components of \mathbf{C}_e will be positive
5. The components of \mathbf{C}_e are the eigenvector centralities for the graph.

Eigenvector Centrality: Example 1



$$\lambda \mathbf{C}_e = A \mathbf{C}_e \quad (A - \lambda I) \mathbf{C}_e = 0 \quad \mathbf{C}_e = [u_1 \ u_2 \ u_3]^T$$

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 - \lambda & 1 & 0 \\ 1 & 0 - \lambda & 1 \\ 0 & 1 & 0 - \lambda \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$
$$\det(A - \lambda I) = \begin{vmatrix} 0 - \lambda & 1 & 0 \\ 1 & 0 - \lambda & 1 \\ 0 & 1 & 0 - \lambda \end{vmatrix} = 0$$

$$(-\lambda)(\lambda^2 - 1) - 1(-\lambda) = 2\lambda - \lambda^3 = \lambda(2 - \lambda^2) = 0$$

Eigenvalues are

$$(-\sqrt{2}, 0, +\sqrt{2})$$

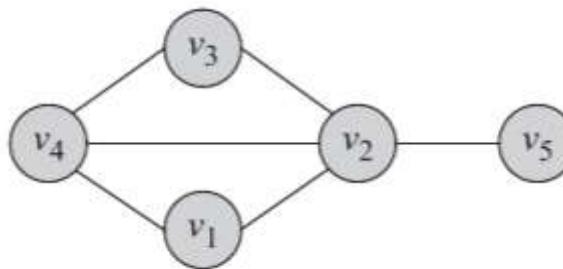
Largest Eigenvalue

Corresponding eigenvector (assuming \mathbf{C}_e has norm 1)

$$\begin{bmatrix} 0 - \sqrt{2} & 1 & 0 \\ 1 & 0 - \sqrt{2} & 1 \\ 0 & 1 & 0 - \sqrt{2} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{C}_e = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 1/2 \\ \sqrt{2}/2 \\ 1/2 \end{bmatrix}$$

Central Node

Eigenvector Centrality: Example 2



$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix} \rightarrow \lambda = (2.68, -1.74, -1.27, 0.33, 0.00)$$

↑
Eigenvalues Vector

$$\lambda_{\max} = 2.68 \rightarrow$$

$$C_e = \begin{bmatrix} 0.4119 \\ 0.5825 \\ 0.4119 \\ 0.5237 \\ 0.2169 \end{bmatrix}$$

Katz Centrality

- A major problem with eigenvector centrality arises when it deals with directed graphs
- Centrality only passes over *outgoing* edges and in special cases such as when a node is in a directed acyclic graph centrality becomes zero
 - The node can have many edge connected to it
- To resolve this problem we add bias term β to the centrality values for all nodes



Elihu Katz

Eigenvector Centrality

$$C_{\text{Katz}}(v_i) = \alpha \sum_{j=1}^n A_{j,i} C_{\text{Katz}}(v_j) + \beta$$

Katz Centrality, cont.

$$C_{\text{Katz}}(v_i) = \alpha \sum_{j=1}^n A_{j,i} C_{\text{Katz}}(v_j) + \beta$$

↑ ↑
Controlling term Bias term

Rewriting equation in a vector form

$$\mathbf{C}_{\text{Katz}} = \alpha A^T \mathbf{C}_{\text{Katz}} + \beta \mathbf{1} \leftarrow \begin{matrix} \text{vector of all 1's} \end{matrix}$$

Katz centrality: $\mathbf{C}_{\text{Katz}} = \beta(\mathbf{I} - \alpha A^T)^{-1} \cdot \mathbf{1}$

Katz Centrality, cont.

$$\mathbf{C}_{\text{Katz}} = \beta(\mathbf{I} - \alpha A^T)^{-1} \cdot \mathbf{1}$$

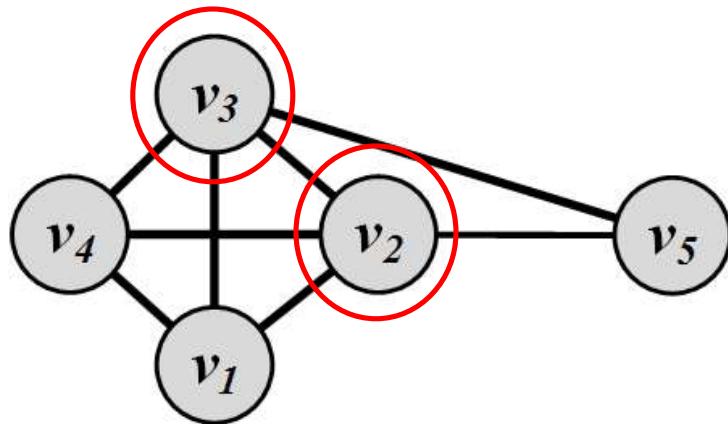
$$C_{\text{Katz}}(v_i) = \alpha \sum_{j=1}^n A_{j,i} C_{\text{Katz}}(v_j) + \beta$$

- When $\alpha=0$, the eigenvector centrality is removed and all nodes get the same centrality value β
 - As α gets larger the effect of β is reduced
- For the matrix $(I - \alpha A^T)$ to be invertible, we must have
 - $\det(I - \alpha A^T) \neq 0$
 - By rearranging we get $\det(A^T - \alpha^{-1} I) = 0$
 - This is basically the characteristic equation,
 - The characteristic equation **first** becomes zero when the largest eigenvalue equals α^{-1}

The largest eigenvalue
is easier to compute
(power method)

In practice we select $\alpha < 1/\lambda$, where λ is the largest eigenvalue of A^T

Katz Centrality Example



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

- The Eigenvalues are -1.68, -1.0, -1.0, 0.35, 3.32
- We assume $\alpha=0.25 < \frac{1}{3.32}$ and $\beta = 0.2$

$$\mathbf{C}_{Katz} = \beta(\mathbf{I} - \alpha A^T)^{-1} \cdot \mathbf{1} = \begin{bmatrix} 1.14 \\ 1.31 \\ 1.31 \\ 1.14 \\ 0.85 \end{bmatrix}$$

Most important nodes!

PageRank

- Problem with Katz Centrality:
 - In directed graphs, once a node becomes an authority (high centrality), it passes **all** its centrality along **all** of its out-links
- This is less desirable since not everyone known by a well-known person is well-known
- **Solution?**
 - We can divide the value of passed centrality by the number of outgoing links, i.e., out-degree of that node
 - Each connected neighbor gets a fraction of the source node's centrality

PageRank, cont.

$$C_p(v_i) = \alpha \sum_{j=1}^n A_{j,i} \frac{C_p(v_j)}{d_j^{\text{out}}} + \beta$$

What if the degree is zero?

$$\begin{cases} d_j^{\text{out}} > 0 \\ D = \text{diag}(d_1^{\text{out}}, d_2^{\text{out}}, \dots, d_n^{\text{out}}) \end{cases} \rightarrow \mathbf{C}_p = \alpha A^T D^{-1} \mathbf{C}_p + \beta \mathbf{1}$$

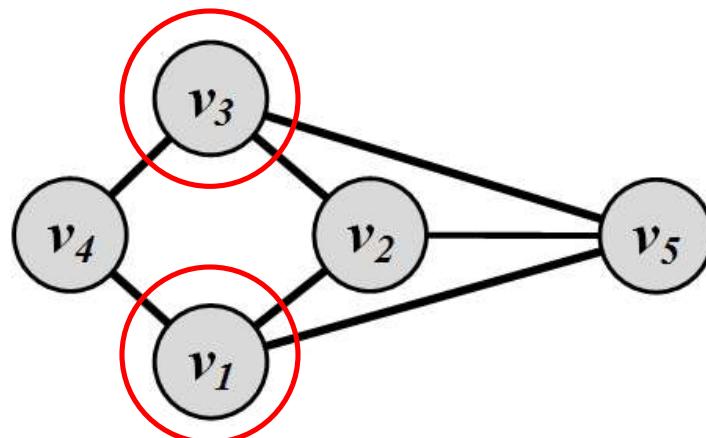


$$\mathbf{C}_p = \beta(\mathbf{I} - \alpha A^T D^{-1})^{-1} \cdot \mathbf{1}$$

Similar to Katz Centrality, in practice, $\alpha < 1/\lambda$, where λ is the largest eigenvalue of $A^T D^{-1}$. In undirected graphs, the largest eigenvalue of $A^T D^{-1}$ is $\lambda = 1$; therefore, $\alpha < 1$.

PageRank Example

- We assume $\alpha=0.95 < 1$ and $\beta = 0.1$

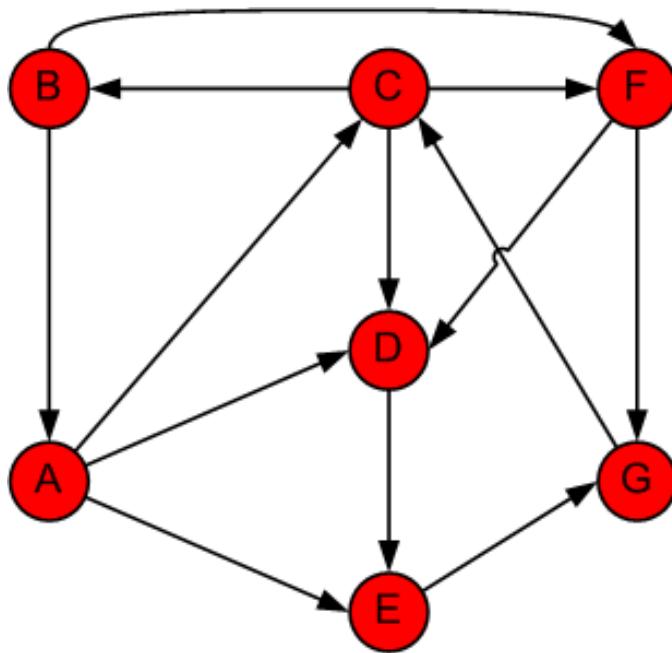


$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \end{bmatrix}$$

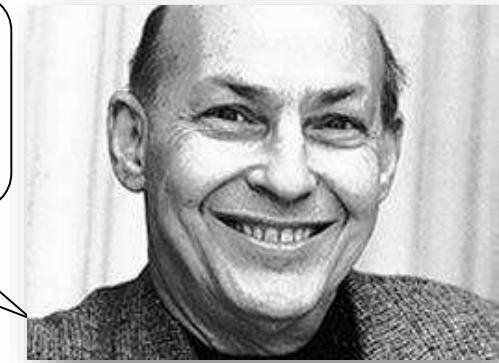
$$\mathbf{C}_p = \beta(\mathbf{I} - \alpha A^T D^{-1})^{-1} \cdot \mathbf{1} =$$

$$\begin{bmatrix} 2.14 \\ 2.13 \\ 2.14 \\ 1.45 \\ 2.13 \end{bmatrix}$$

PageRank Example – Alternative Approach [Markov Chains]



"You don't understand anything until you learn it more than one way"



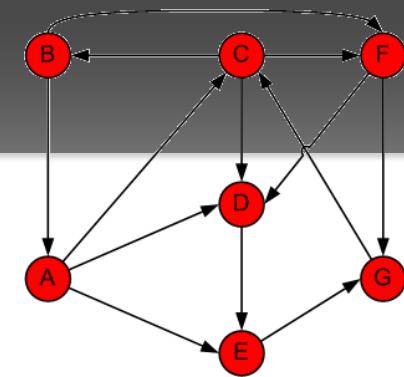
Marvin Minsky (1927-2016)

Using Power Method

$$\alpha=1 \text{ and } \beta = 0?$$

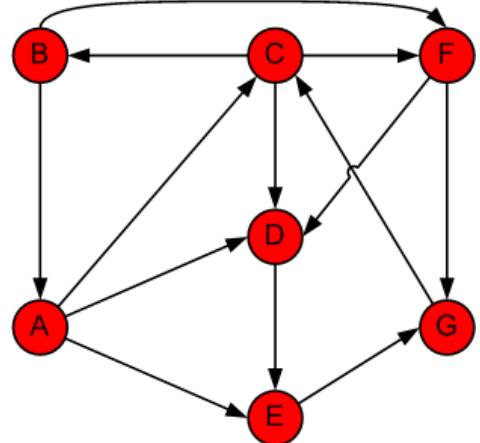
Step	A	B	C	D	E	F	G
0	1/7	1/7	1/7	1/7	1/7	1/7	1/7
1	B/2	C/3	A/3 + G	A/3 + C/3 + F/2	A/3 + D	C/3 + B/2	F/2 + E
	0.071	0.048	0.190	0.167	0.190	0.119	0.214

PageRank: Example



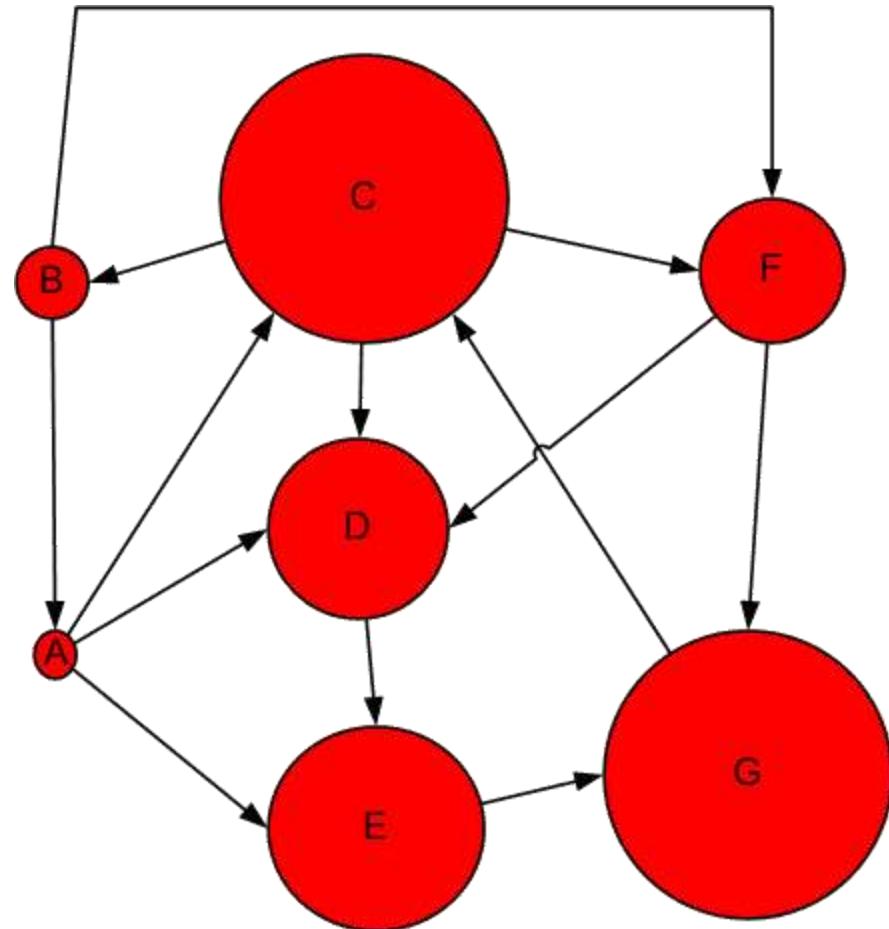
Step	A	B	C	D	E	F	G	Sum
1	0.143	0.143	0.143	0.143	0.143	0.143	0.143	1.000
2	0.071	0.048	0.190	0.167	0.190	0.119	0.214	1.000
3	0.024	0.063	0.238	0.147	0.190	0.087	0.250	1.000
4	0.032	0.079	0.258	0.131	0.155	0.111	0.234	1.000
5	0.040	0.086	0.245	0.152	0.142	0.126	0.210	1.000
6	0.043	0.082	0.224	0.158	0.165	0.125	0.204	1.000
7	0.041	0.075	0.219	0.151	0.172	0.115	0.228	1.000
8	0.037	0.073	0.241	0.144	0.165	0.110	0.230	1.000
9	0.036	0.080	0.242	0.148	0.157	0.117	0.220	1.000
10	0.040	0.081	0.232	0.151	0.160	0.121	0.215	1.000
11	0.040	0.077	0.228	0.151	0.165	0.118	0.220	1.000
12	0.039	0.076	0.234	0.148	0.165	0.115	0.223	1.000
13	0.038	0.078	0.236	0.148	0.161	0.116	0.222	1.000
14	0.039	0.079	0.235	0.149	0.161	0.118	0.219	1.000
15	0.039	0.078	0.232	0.150	0.162	0.118	0.220	1.000
Rank	7	6	1	4	3	5	2	

Effect of PageRank



PageRank

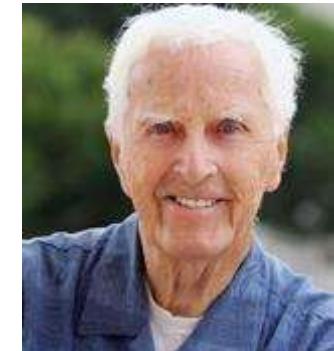
Node	Rank
A	7
B	6
C	1
D	4
E	3
F	5
G	2



Centrality in terms of how you connect others (information broker)

Betweenness Centrality

Another way of looking at centrality is by considering how important nodes are in connecting other nodes



Linton Freeman

$$C_b(v_i) = \sum_{s \neq t \neq v_i} \frac{\sigma_{st}(v_i)}{\sigma_{st}}$$

σ_{st} The number of shortest paths from vertex s to t – a.k.a. **information pathways**

$\sigma_{st}(v_i)$ The number of **shortest paths** from s to t that pass through v_i

Normalizing Betweenness Centrality

- In the best case, node v_i is on all shortest paths from s to t , hence, $\frac{\sigma_{st}(v_i)}{\sigma_{st}} = 1$

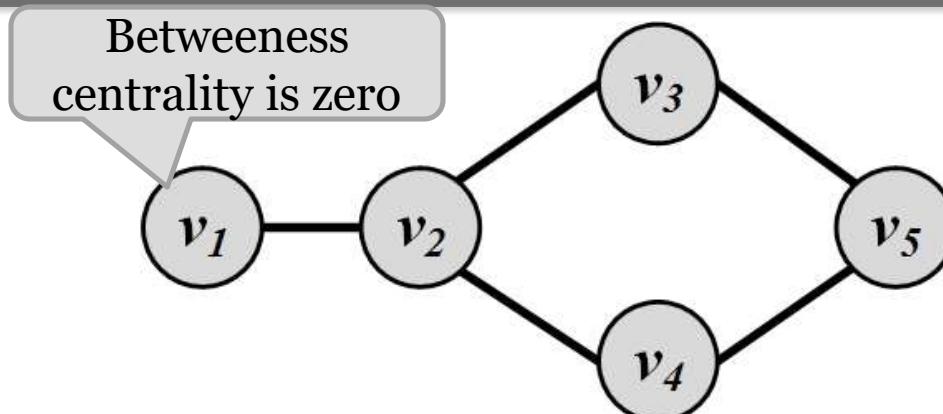
Therefore, the maximum value is $(n - 1)(n - 2)$

$$= \sum_{s \neq t \neq v_i} 1 = 2 \binom{n-1}{2} = (n - 1)(n - 2)$$

Therefore, the maximum value is $(n - 1)(n - 2)$

Betweenness centrality: $C_b^{\text{norm}}(v_i) = \frac{C_b(v_i)}{2 \binom{n-1}{2}}$

Betweenness Centrality: Example 1



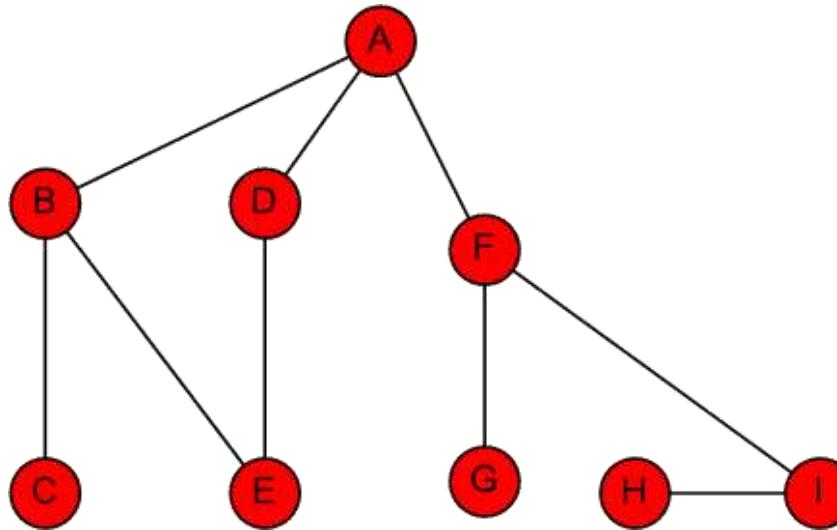
$$C_b(v_2) = 2 \times \left(\underbrace{(1/1)}_{s=v_1,t=v_3} + \underbrace{(1/1)}_{s=v_1,t=v_4} + \underbrace{(2/2)}_{s=v_1,t=v_5} + \underbrace{(1/2)}_{s=v_3,t=v_4} + \underbrace{0}_{s=v_3,t=v_5} + \underbrace{0}_{s=v_4,t=v_5} \right)$$
$$= 2 \times 3.5 = 7,$$

$$C_b(v_3) = 2 \times \left(\underbrace{0}_{s=v_1,t=v_2} + \underbrace{0}_{s=v_1,t=v_4} + \underbrace{(1/2)}_{s=v_1,t=v_5} + \underbrace{0}_{s=v_2,t=v_4} + \underbrace{(1/2)}_{s=v_2,t=v_5} + \underbrace{0}_{s=v_4,t=v_5} \right)$$
$$= 2 \times 1.0 = 2,$$

$$C_b(v_4) = C_b(v_3) = 2 \times 1.0 = 2,$$

$$C_b(v_5) = 2 \times \left(\underbrace{0}_{s=v_1,t=v_2} + \underbrace{0}_{s=v_1,t=v_3} + \underbrace{0}_{s=v_1,t=v_4} + \underbrace{0}_{s=v_2,t=v_3} + \underbrace{0}_{s=v_2,t=v_4} + \underbrace{(1/2)}_{s=v_3,t=v_4} \right)$$
$$= 2 \times 0.5 = 1,$$

Betweenness Centrality: Example 2

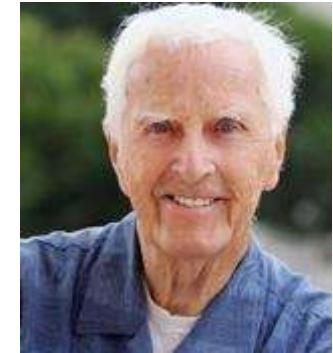


Node	Betweenness Centrality	Rank
A	$16 + 1/2 + 1/2$	1
B	$7+5/2$	3
C	0	7
D	$5/2$	5
E	$1/2 + 1/2$	6
F	$15 + 2$	1
G	0	7
H	0	7
I	7	4

Centrality in terms of how fast you can reach others

Closeness Centrality

- The intuition is that influential/central nodes can quickly reach other nodes
- These nodes should have a ***smaller average shortest path length*** to others



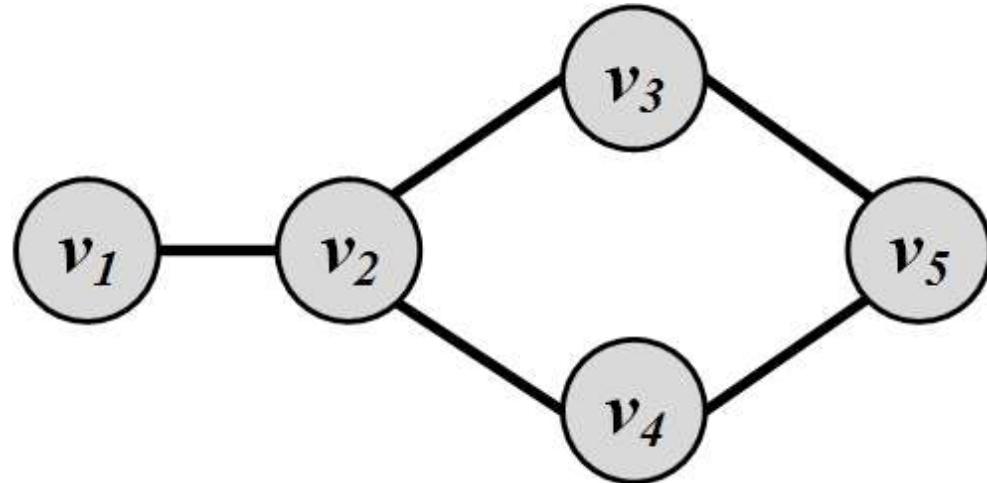
Linton Freeman

Closeness centrality: $C_c(v_i) = \frac{1}{\bar{l}_{v_i}}$

$$\bar{l}_{v_i} = \frac{1}{n-1} \sum_{v_j \neq v_i} l_{i,j}$$

Closeness Centrality: Example 1

$$\bar{l}_{v_i} = \frac{1}{n-1} \sum_{v_j \neq v_i} l_{i,j}$$



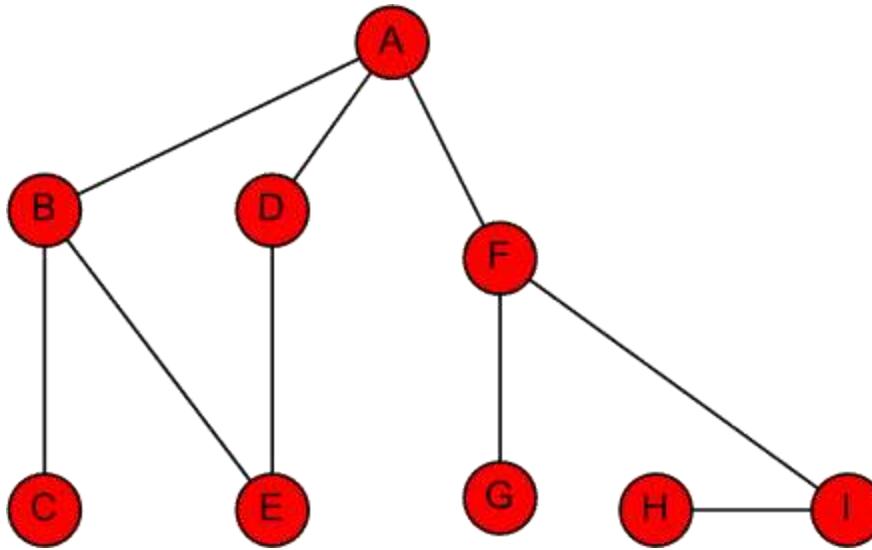
$$C_c(v_1) = 1 / ((1 + 2 + 2 + 3)/4) = 0.5,$$

$$C_c(v_2) = 1 / ((1 + 1 + 1 + 2)/4) = 0.8,$$

$$C_c(v_3) = C_b(v_4) = 1 / ((1 + 1 + 2 + 2)/4) = 0.66,$$

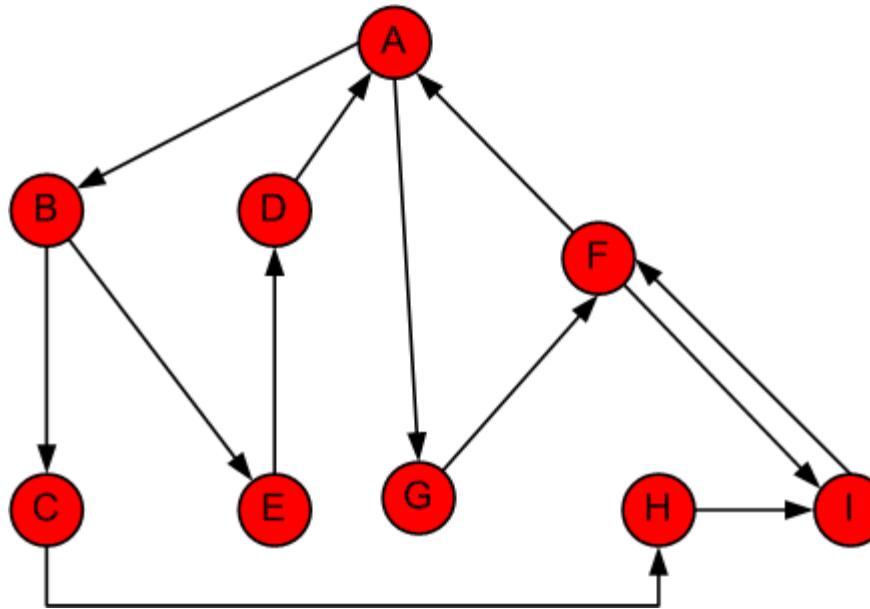
$$C_c(v_5) = 1 / ((1 + 1 + 2 + 3)/4) = 0.57.$$

Closeness Centrality: Example 2 (Undirected)



Node	A	B	C	D	E	F	G	H	I	D_Avg	Closeness Centrality	Rank
A	0	1	2	1	2	1	2	3	2	1.750	0.571	1
B	1	0	1	2	1	2	3	4	3	2.125	0.471	3
C	2	1	0	3	2	3	4	5	4	3.000	0.333	8
D	1	2	3	0	1	2	3	4	3	2.375	0.421	4
E	2	1	2	1	0	3	4	5	4	2.750	0.364	7
F	1	2	3	2	3	0	1	2	1	1.875	0.533	2
G	2	3	4	3	4	1	0	3	2	2.750	0.364	7
H	3	4	5	4	5	2	3	0	1	3.375	0.296	9
I	2	3	4	3	4	1	2	1	0	2.500	0.400	5

Closeness Centrality: Example 3 (Directed)



Node	A	B	C	D	E	F	G	H	I	D_Avg	Closeness Centrality	Rank
A	0	1	2	3	2	2	1	3	3	2.125	0.471	1
B	3	0	1	2	1	4	4	2	3	2.500	0.400	2
C	4	5	0	7	6	3	5	1	2	4.125	0.242	9
D	1	2	3	0	3	3	2	4	5	2.875	0.348	3
E	2	3	4	1	0	4	3	5	5	3.375	0.296	6
F	1	2	3	4	3	0	2	4	4	2.875	0.348	4
G	2	3	4	5	4	1	0	5	2	3.250	0.308	5
H	4	4	5	6	5	2	4	0	1	3.875	0.258	8
I	2	3	4	5	4	1	4	5	0	3.500	0.286	7

An Interesting Comparison!

Comparing three centrality values

- Generally, the 3 centrality types will be positively correlated
- When they are not (or low correlation), it usually reveals interesting information

	Low Degree	Low Closeness	Low Betweenness
High Degree		<i>Node is embedded in a community that is far from the rest of the network</i>	<i>Ego's connections are redundant - communication bypasses the node</i>
High Closeness	<i>Key node connected to important/active alters</i>		<i>Probably multiple paths in the network, ego is near many people, but so are many others</i>
High Betweenness	<i>Ego's few ties are crucial for network flow</i>	<i>Very rare! Ego monopolizes the ties from a small number of people to many others.</i>	

This slide is modified from a slide developed by James Moody

Structural Equivalence

- **Structural Equivalence:**
 - We look at the neighborhood shared by two nodes;
 - The size of this shared neighborhood defines how similar two nodes are.
- **Example:**
 - *Two brothers have in common*
 - *sisters, mother, father, grandparents, etc.*
 - *This shows that they are similar,*
 - *Two random male or female individuals do not have much in common and are dissimilar.*

Structural Equivalence: Definitions

- **Vertex similarity:** $\sigma(v_i, v_j) = |N(v_i) \cap N(v_j)|$

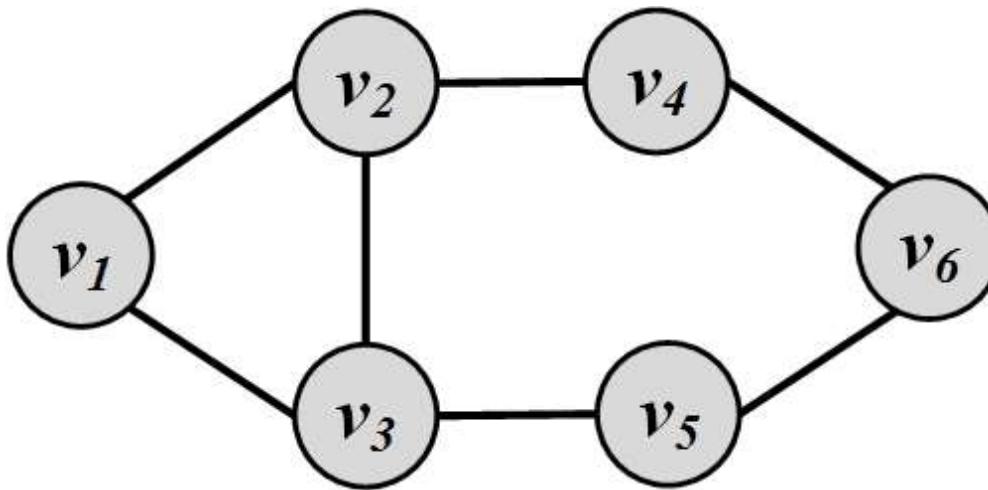
Normalize?

Jaccard Similarity: $\sigma_{Jaccard}(v_i, v_j) = \frac{|N(v_i) \cap N(v_j)|}{|N(v_i) \cup N(v_j)|}$

Cosine Similarity: $\sigma_{Cosine}(v_i, v_j) = \frac{|N(v_i) \cap N(v_j)|}{\sqrt{|N(v_i)||N(v_j)|}}$

- The neighborhood $N(v)$ often excludes the node itself v .
 - **What can go wrong?**
 - Connected nodes not sharing a neighbor will be assigned zero similarity
 - **Solution:**
 - We can assume nodes are included in their neighborhoods

Similarity: Example



$$\sigma_{\text{Jaccard}}(v_2, v_5) = \frac{|\{v_1, v_3, v_4\} \cap \{v_3, v_6\}|}{|\{v_1, v_3, v_4, v_6\}|} = 0.25$$

$$\sigma_{\text{Cosine}}(v_2, v_5) = \frac{|\{v_1, v_3, v_4\} \cap \{v_3, v_6\}|}{\sqrt{|\{v_1, v_3, v_4\}| |\{v_3, v_6\}|}} = 0.40.$$

Similarity Significance

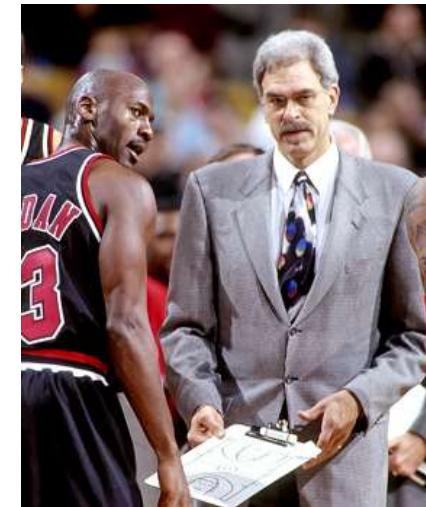
Measuring Similarity Significance: compare the calculated similarity value with its expected value where vertices pick their neighbors at random

- For vertices v_i and v_j with degrees d_i and d_j this expectation is $d_i d_j / n$
 - There is a d_i/n chance of becoming v_i 's neighbor
 - v_j selects d_j neighbors
- We can rewrite neighborhood overlap as

$$\sigma(v_i, v_j) = |N(v_i) \cap N(v_j)| = \sum_k A_{i,k} A_{j,k}$$

Regular Equivalence

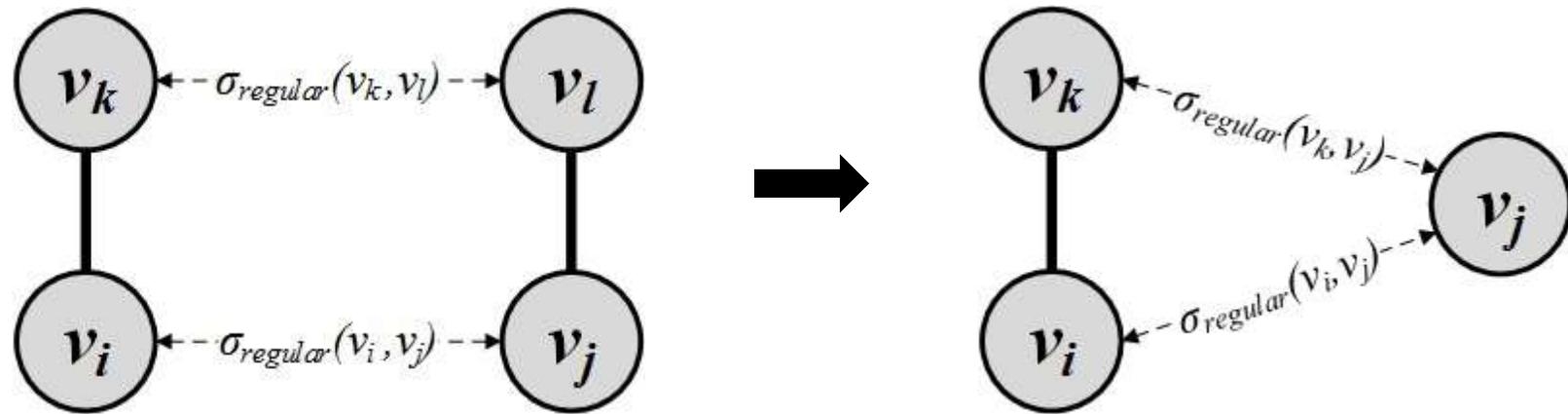
- In regular equivalence,
 - We **do not** look at neighborhoods shared between individuals, but
 - How neighborhoods themselves are similar
- Example:
 - *Athletes are similar not because they know each other in person, but since they know similar individuals, such as coaches, trainers, other players, etc.*



Regular Equivalence

- v_i, v_j are similar when their neighbors v_k and v_l are similar

$$\sigma_{\text{regular}}(v_i, v_j) = \alpha \sum_{k,l} A_{i,k} A_{j,l} \sigma_{\text{regular}}(v_k, v_l)$$



- The equation (left figure) is hard to solve since it is self referential so we relax our definition using the right figure

Regular Equivalence

- v_i and v_j are similar when v_j is similar to v_i 's neighbors v_k

$$\sigma_{regular}(v_i, v_j) = \alpha \sum_k A_{i,k} \sigma_{Regular}(v_k, v_j)$$



$$\sigma_{regular} = \alpha A \sigma_{Regular}$$

- In vector format

A vertex is highly similar to itself, we guarantee this by adding an identity matrix to the equation



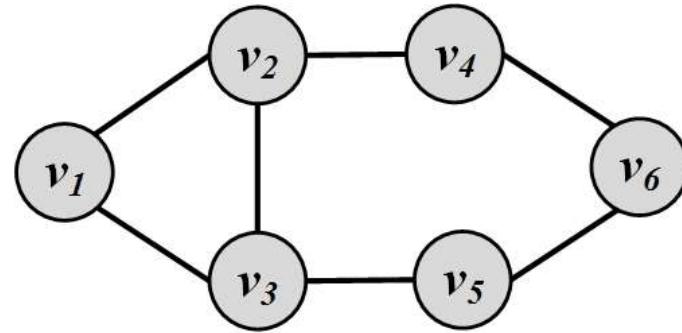
$$\sigma_{regular} = \alpha A \sigma_{Regular} + \mathbf{I}$$



$$\sigma_{regular} = (\mathbf{I} - \alpha A)^{-1}$$

When $\alpha < 1/\lambda_{max}$ the matrix is invertible

Regular Equivalence: Example



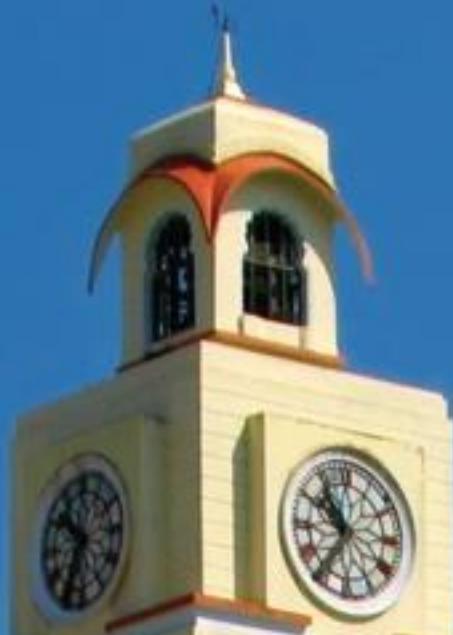
$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

The largest eigenvalue of A is 2.43

Set $\alpha = 0.3 < 1/2.43$

$$\sigma_{\text{regular}} = (I - 0.3A)^{-1} = \begin{bmatrix} 1.43 & 0.73 & 0.73 & 0.26 & 0.26 & 0.16 \\ 0.73 & 1.63 & 0.80 & 0.56 & 0.32 & 0.26 \\ 0.73 & 0.80 & 1.63 & 0.32 & 0.56 & 0.26 \\ 0.26 & 0.56 & 0.32 & 1.31 & 0.23 & 0.46 \\ 0.26 & 0.32 & 0.56 & 0.23 & 1.31 & 0.46 \\ 0.16 & 0.26 & 0.26 & 0.46 & 0.46 & 1.27 \end{bmatrix}$$

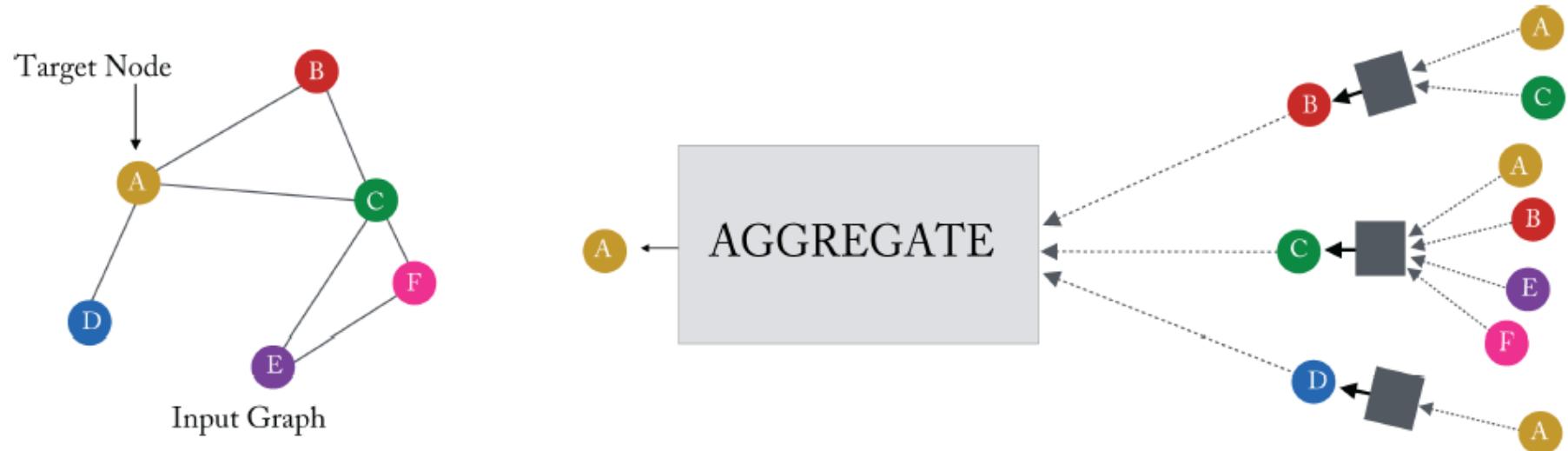
- Any row/column of this matrix shows the similarity to other vertices
- Vertex 1 is most similar (other than itself) to vertices 2 and 3
- Nodes 2 and 3 have the highest similarity (**regular equivalence**)



Graph Neural Networks (S*-**_AIMLCZG514)

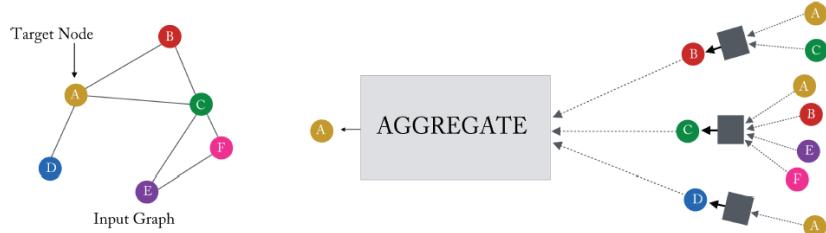
BITS Pilani
Pilani Campus

NEURAL MESSAGE PASSING



Overview of how a single node aggregates messages from its local neighborhood

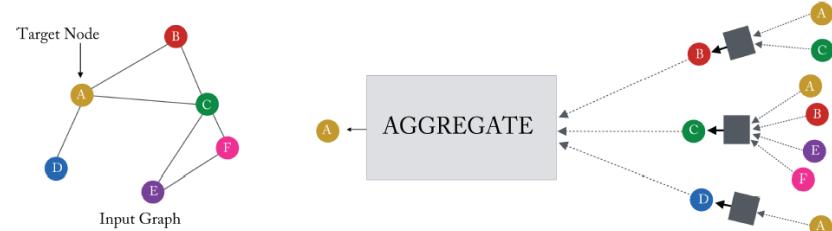
NEURAL MESSAGE PASSING



- ❖ The model aggregates messages from A's local graph neighbors (i.e., B, C, and D), and
- ❖ in turn, the messages coming from these neighbors are based on information aggregated from their respective neighborhoods, and so on.
- ❖ This visualization shows a **two-layer** version of a message-passing model.
- ❖ This is an iterative process.

Notice that the computation graph of the GNN forms a tree structure by unfolding the neighborhood around the target node.

NEURAL MESSAGE PASSING



- ❖ During each message-passing iteration in a GNN, a *hidden embedding* $\mathbf{h}_u^{(k)}$ corresponding to each node $u \in V$ is updated according to information aggregated from u 's graph neighborhood $N(u)$.

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)} \left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in N(u)\}) \right) \\ &= \text{UPDATE}^{(k)} \left(\mathbf{h}_u^{(k)}, \mathbf{m}_{N(u)}^{(k)} \right),\end{aligned}$$

- ❖ **Update** and **Aggregate** are arbitrary differentiable functions (i.e., neural networks) and
- ❖ $\mathbf{m}_{N(u)}$ is the “**message**” that is aggregated from u 's graph neighborhood $N(u)$.

NEURAL MESSAGE PASSING



$$\begin{aligned} \mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \\ &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right), \end{aligned}$$

- ❖ At each iteration k of the GNN, the **aggregate** function takes as input the set of embeddings of the nodes in u 's graph neighborhood $\mathcal{N}(u)$ and generates a message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ based on this aggregated neighborhood information.
- ❖ The **update** function combines the message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ with the previous embedding $\mathbf{h}_u^{(k-1)}$ of node u to generate the updated embedding $\mathbf{h}_u^{(k)}$.
- ❖ The initial embeddings at $k = 0$ are set to the input features for all the nodes, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in V$.

NEURAL MESSAGE PASSING



$$\begin{aligned} h_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(h_u^{(k)}, \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \\ &= \text{UPDATE}^{(k)}\left(h_u^{(k)}, m_{\mathcal{N}(u)}^{(k)}\right), \end{aligned}$$

- ❖ After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings for each node, i.e.,

$$z_u = h_u^{(k)}, \forall u \in V$$

Note that: Since the aggregate function takes a set as input, GNNs defined in this way are permutation equivariant by design.

Node features

- ❖ Unlike the shallow embedding methods, the GNN framework requires that we have node features $x_u, \forall u \in V$ as input to the model.
- ❖ In many graphs, we will have rich node features to use i.e:
 - ❖ Gene expression features in biological networks or
 - ❖ Text features in social networks.
- ❖ However, in cases where no node features are available, we can use **node statistics** (*network measures*) to define features.

Benefits for Message Passing Graph Neural Network (MPGNN)

1. It directly **works on** the original **graph** structure and no pre-processing is needed.
 2. **Graphs** in practice are typically **sparse** ($|E| \ll |V|^2$) and thus MP-GNN is able to scale to very large but sparse graphs.
 3. Each of the three operations ***Message, Aggregate and Update*** can be computed in parallel across all nodes and edges, which is beneficial for parallel computing platforms such as GPUs and map-reduce systems.
 4. Because it is natural and easy to be implemented in practice, most GNN architectures essentially follow the MP-GNN framework by adopting specific ***Message, Aggregate and Update*** operations.
-

MOTIVATIONS AND INTUITIONS

- ❖ The basic intuition behind the GNN message-passing framework is straightforward:
 - ❖ At each iteration, ***every node aggregates information from its local neighborhood.***
 - ❖ As these iterations progress each node embedding contains more and more information from further reaches of the graph.
 - ❖ For $k = 1$, every node embedding contains information from its **1-hop** neighborhood,
 - ❖ For $k = 2$, every node embedding contains information from its **2-hop** neighborhood;
 - ❖ For k^{th} iteration, every node embedding contains information about its **k-hop** neighborhood.

MOTIVATIONS AND INTUITIONS

- ❖ *Kind of information node embedding actually encode:*
 1. *Structural information* about the graph
 - ❖ After k iterations of GNN message passing,
 - ❖ The embedding $h_u^{(k)}$ of node u might encode information about the degrees of all the nodes in u 's **k -hop neighborhood**.
 - ❖ When analyzing molecular graphs, we can use degree information to infer *atom types* and different *structural motifs*, such as benzene rings.

MOTIVATIONS AND INTUITIONS

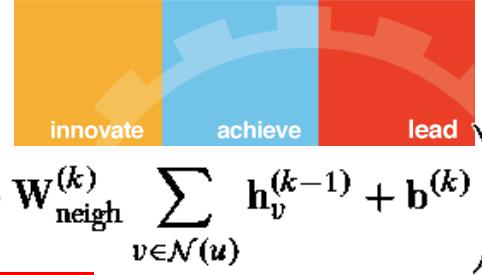
- ❖ *Kind of information node embedding actually encode:*
 2. *Feature-based information* about the graph
 - ❖ After k iterations of GNN message passing,
 - ❖ The embeddings for each node also encode information about all the features in their ***k-hop neighborhood.***
 - ❖ This local feature-aggregation behavior of GNNs is analogous to the behavior of the ***convolutional kernels in CNNs.***
 - ❖ However, whereas CNNs aggregate feature information from spatially defined patches in an image, GNNs aggregate information based on local graph neighborhoods.

THE BASIC GNN

$$\begin{aligned} \mathbf{h}_u^{(k+1)} &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right) \\ &= \text{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right), \end{aligned}$$

- ❖ The GNN framework discussed so far was in a relatively abstract fashion as a series of message-passing iterations using update and aggregate functions defined above.
- ❖ In order to translate the abstract GNN framework into something we can implement, we must give concrete equation to these update and aggregate functions.
- ❖ **The Basic GNN message passing** is defined as

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}_{\text{self}}^{(k)} \mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)} \right)$$



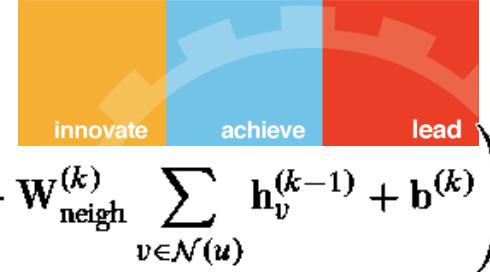
THE BASIC GNN

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}_{\text{self}}^{(k)} \mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)} + \mathbf{b}^{(k)} \right)$$

❖ Here:

- ❖ $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d^{(k)} \times d^{(k-1)}}$ are trainable parameter matrices
- ❖ σ is the activation function denoting an ***elementwise nonlinearity*** (e.g., a tanh or ReLU).
- ❖ The ***bias term*** $\mathbf{b}^{(k)} \in \mathbb{R}^{d^{(k)}}$ is often omitted for notational simplicity, but including the bias term can be important to achieve strong performance. (a bias value allows you to shift the activation function to the left or right, which may be critical for successful learning. You can say the bias makes the equation more flexible to adapt to the best values)

1. We first **sum** the messages incoming from the neighbors;
2. We **combine** the neighborhood information with the node's previous embedding using a **linear combination**;
3. Finally, we apply an elementwise nonlinearity.



THE BASIC GNN

$$h_u^{(k)} = \sigma \left(W_{\text{self}}^{(k)} h_u^{(k-1)} + W_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)} + b^{(k)} \right)$$

- ❖ We can equivalently define the basic GNN through the **update** and **aggregate** functions:

$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} h_v,$$

$$\text{UPDATE}(h_u, m_{\mathcal{N}(u)}) = \sigma (W_{\text{self}} h_u + W_{\text{neigh}} m_{\mathcal{N}(u)}),$$

- ❖ where we recall that we use

$$m_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

In general, the parameters $W_{\text{self}}^{(k)}$, $W_{\text{neigh}}^{(k)}$, and b can be shared across the GNN message passing iterations or trained separately for each layer.

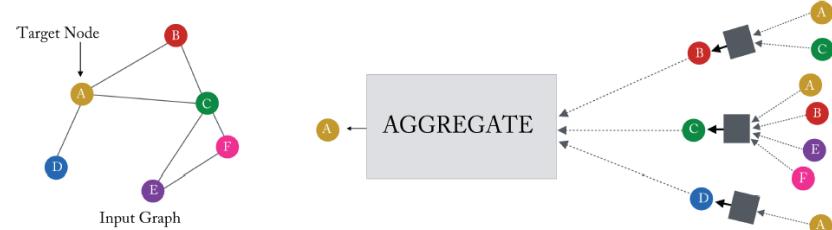
Node vs. graph-level equations

- ❖ Many GNNs can also be succinctly defined using graph-level equations:

$$\mathbf{H}^{(t)} = \sigma \left(\mathbf{A} \mathbf{H}^{(k-1)} \mathbf{W}_{\text{neigh}}^{(k)} + \mathbf{H}^{(k-1)} \mathbf{W}_{\text{self}}^{(k)} \right),$$

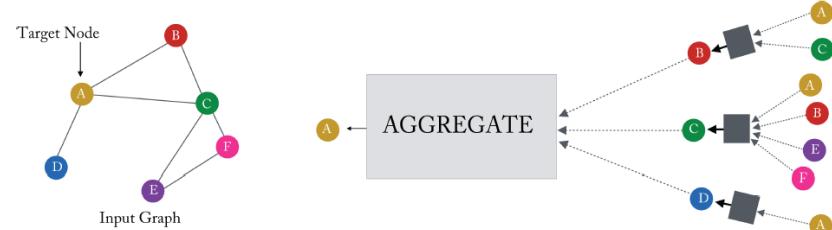
- ❖ Here $\mathbf{H}^{(k)} \in \mathbb{R}^{|V| \times d}$ denotes the matrix of node representations at layer t in the GNN (*with each node corresponding to a row in the matrix*)
- ❖ \mathbf{A} is the graph adjacency matrix, and
- ❖ We have omitted the bias term for notational simplicity

Limitations of MP-GNN



- ❖ MP-GNN iteratively updates the representation of each node by aggregating representations of its neighbors.
- ❖ The obtained node representation essentially encodes the subtree rooted at Node v.
- ❖ However, using this rooted subtree to represent a node may lose useful information, such as:
 1. The information about the distance between multiple nodes is lost.
 2. The information about cycles is lost.

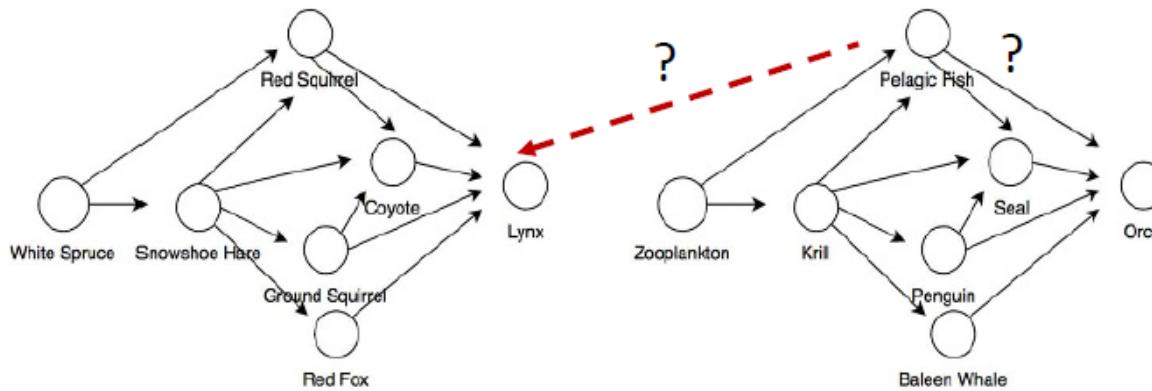
Limitations of MP-GNN



1. The information about the distance between multiple nodes is lost.
 - ❖ MP-GNN has limited power in capturing the position/location of a given node with respect to another node in the graph.
 - ❖ Many nodes may share similar subtrees, and thus, MP-GNN produces the same representation for them although the nodes may be located at different locations in the graph.

Limitations of MP-GNN

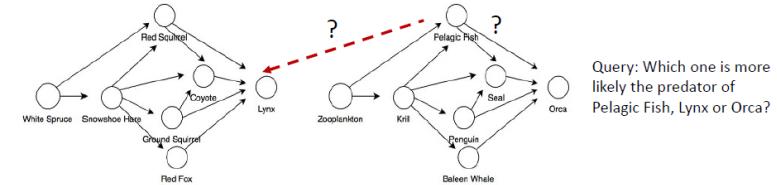
1. The information about the distance between multiple nodes is lost.
 - ❖ This location information of nodes is crucial for the tasks that depend on multiple nodes, such as ***link prediction***, as two nodes that tend to be connected with a link are typically located close to each other.



Query: Which one is more likely the predator of Pelagic Fish, Lynx or Orca?

Liben-Nowell D, Kleinberg J (2007) The link-prediction problem for social networks. Journal of the American society for information science and technology 58(7):1019–1031

Limitations of MP-GNN



- ❖ A foodweb network example that demonstrates limitations of MP-GNN.
- ❖ MP-GNN will associate Lynx and Orca with the same node representations, i.e., $\mathbf{h}_{Lynx}^{(i)} = \mathbf{h}_{Orca}^{(i)}$, as these two nodes hold the same rooted subtree.
- ❖ Note that we do not consider node features.
- ❖ Thus, MP-GNN cannot predict whether Lynx or Orca is more likely to be the predator of Pelagic Fish (alink prediction task).

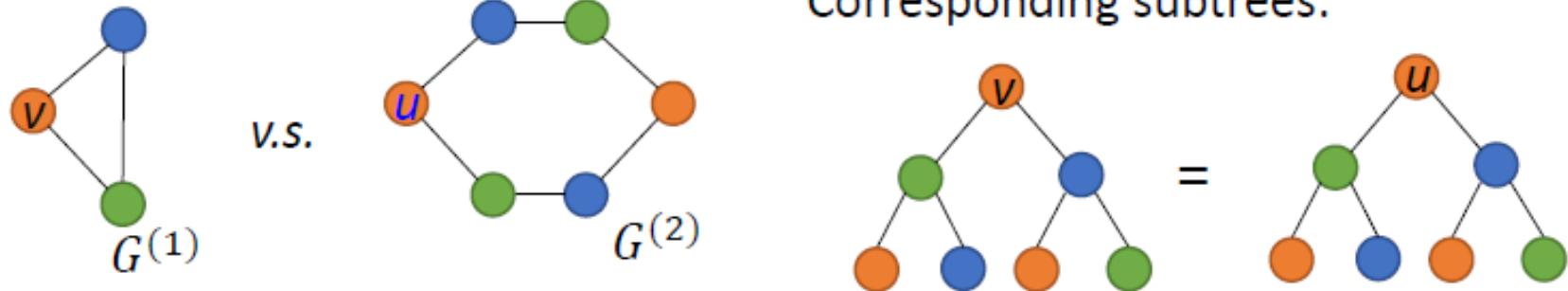
Limitations of MP-GNN

2. The information about cycles is lost.
 - ❖ Particularly, when expanding the subtree of a node, MP-GNN essentially **losses track of the node identities** in the subtrees.
 - ❖ The information about cycles is crucial in applications such as ***subgraph matching and counting*** because loops frequently appear in the queried subgraph patterns of a subgraph matching/counting problem. It was formally proved that MP-GNN is able to count star structures (a particular form of trees) but cannot count connected subgraphs with three or more nodes that **form cycles**.

Srinivasan B, Ribeiro B (2020a) On the equivalence between node embeddings and structural graph representations. In: International Conference on Learning Representations

Limitations of MP-GNN

2. The information about cycles is lost.



- ❖ The node representations $h_v^{(L)} = h_u^{(L)}$, given by MP-GNN are the same, although they belong to different cycles, 3-cycle and 6-cycle, respectively.

Limitations of MP-GNN (Scaling Issue)

- ❖ There are two major challenges while scaling the original implementation of GNN to large graphs.
 1. First, most of the GNN models usually **compute the entire adjacency matrix and node embeddings** of the graph, which demands a huge memory space.
 2. Second, **training** GNN requires **recursively updating** the gradient for **each node** in the graph, which becomes infeasible and ineffective for large graphs.
 - ❖ The gradient descent may take too long to converge on large dataset, and stochastic gradient is introduced to speed up the process towards an optimum.
 - ❖ Current studies propose to tackle these obstacles mainly from sampling paradigms (*we will cover later*)
-

MESSAGE PASSING WITH SELF-LOOPS



$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v,$$

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma (\mathbf{W}_{\text{self}} \mathbf{h}_u + \mathbf{W}_{\text{neigh}} \mathbf{m}_{\mathcal{N}(u)}),$$

$$\mathbf{m}_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

- ❖ As a simplification of the neural message passing approach, it is common to ***add self-loops*** to the input graph and omit the explicit update step.

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

- ❖ The aggregation is taken over the set $\mathcal{N}(u) \cup \{u\}$, i.e., the node's neighbors as well as the node itself.
- ❖ It is equivalent to sharing parameters between the \mathbf{W}_{self} and $\mathbf{W}_{\text{neigh}}$ matrices, which gives the following graph-level update and we call it is ***Self-loop GNN***.

$$\mathbf{H}^{(t)} = \sigma \left((\mathbf{A} + \mathbf{I}) \mathbf{H}^{(t-1)} \mathbf{W}^{(t)} \right).$$

MESSAGE PASSING WITH SELF-LOOPS



$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} h_v,$$

$$\text{UPDATE}(h_u, m_{\mathcal{N}(u)}) = \sigma(W_{\text{self}}h_u + W_{\text{neigh}}m_{\mathcal{N}(u)}),$$

$$m_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

$$h_u^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

❖ Benefits:

- ❖ The benefit is that we no longer need to define an explicit update function.
- ❖ By this we can often alleviate overfitting,

❖ Shortcomings:

- ❖ It also severely limits the expressivity of the GNN, as the information coming from the node's neighbors cannot be differentiated from the information from the node itself.

GENERALIZED NEIGHBORHOOD AGGREGATION

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v,$$

- ❖ The most basic neighborhood aggregation operation simply takes the sum of the neighbor embeddings.
- ❖ One issue with this approach is that it can be unstable and highly sensitive to node degrees.
- ❖ For instance, suppose node u has 100 times more neighbors as node u' , then we would reasonably expect that $\|\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v\| \gg \|\sum_{v' \in \mathcal{N}(u')} \mathbf{h}_{v'}\|$
- ❖ This drastic difference in magnitude can lead to numerical instabilities as well as difficulties for optimization.

GENERALIZED NEIGHBORHOOD AGGREGATION

$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} h_v,$$

- ❖ The solution to this problem is to simply normalize the aggregation operation based upon the degrees of the nodes involved.
 1. The simplest approach is to just take an *average* rather than sum:

$$m_{\mathcal{N}(u)} = \frac{\sum_{v \in \mathcal{N}(u)} h_v}{|\mathcal{N}(u)|},$$

2. The other successful approach is *symmetric normalization*:

$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{h_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$

Symmetric normalization ($D^{-1/2}AD^{-1/2}$), takes into account, both of your number of neighbor, and their own number of neighbor. In this the largest eigen value being 1 (ensured by square root).

GENERALIZED NEIGHBORHOOD AGGREGATION



❖ Example:

❖ In a citation graph:- information from very high-degree nodes (i.e., papers that are cited many times) may not be very useful for inferring community membership in the graph, since these papers can be cited thousands of times across diverse subfields. (e.g Gilmer et.al. 2017)



arXiv

<https://arxiv.org> > cs ::

Neural Message Passing for Quantum Chemistry

by J Gilmer · 2017 · Cited by 7000 — In this paper, we reformulate existing models into a single common framework we call **Message Passing Neural Networks** (MPNNs) and explore ...

Graph Convolutional Networks (GCNs)

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$

- ❖ One of the most popular baseline graph neural network models is the Graph Convolutional Network (GCN) which employs the symmetric-normalized aggregation as well as the self-loop update approach.
- ❖ The GCN model thus defines the message passing function as:

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \right).$$

- ❖ σ is the activation function such as a tanh or ReLU.

Limitations of MP-GNN (Scaling Issue) Re-Visit

- ❖ There are two major challenges while scaling the original implementation of GNN to large graphs.
 1. First, most of the GNN models usually compute the entire adjacency matrix and node embeddings of the graph, which demands a huge memory space.
 2. Second, training GNN requires recursively updating the gradient for each node in the graph, which becomes infeasible and ineffective for large graphs. The gradient descent may take too long to converge on large dataset, and stochastic gradient is introduced to speed up the process towards an optimum.
 - ❖ Current studies propose to tackle these obstacles mainly from sampling paradigms (*we will cover later*)
-

SET AGGREGATORS(SAGE)

- ❖ Neighborhood normalization can be a useful tool to improve GNN performance, but can we do more to improve the aggregate operator?
 - ❖ Is there perhaps something more sophisticated than just summing over the neighbor embeddings?
-

Sampling Paradigms

- ❖ The concept of sampling aims at selecting a partition of all the samples to represent the entire sample distribution.
- ❖ Therefore, the sampling algorithm on large graphs refers to the approach that uses partial graph instead of the full graph to address target problems.
 - ❖ Node-wise sampling,
 - ❖ Layer-wise sampling and
 - ❖ Graphwise sampling.

Sampling Paradigms

- ❖ Node-wise sampling,
 - ❖ Lays a dominant role during the early stage of implementing GCN on large graphs,
 - ❖ such as Graph SAmple and aggreGatE (Graph-SAGE)[Hamilton et al, 2017b] and Variance Reduction Graph Convolutional Networks (VR-GCN) [Chen et al, 2018d]
-

Sampling Paradigms

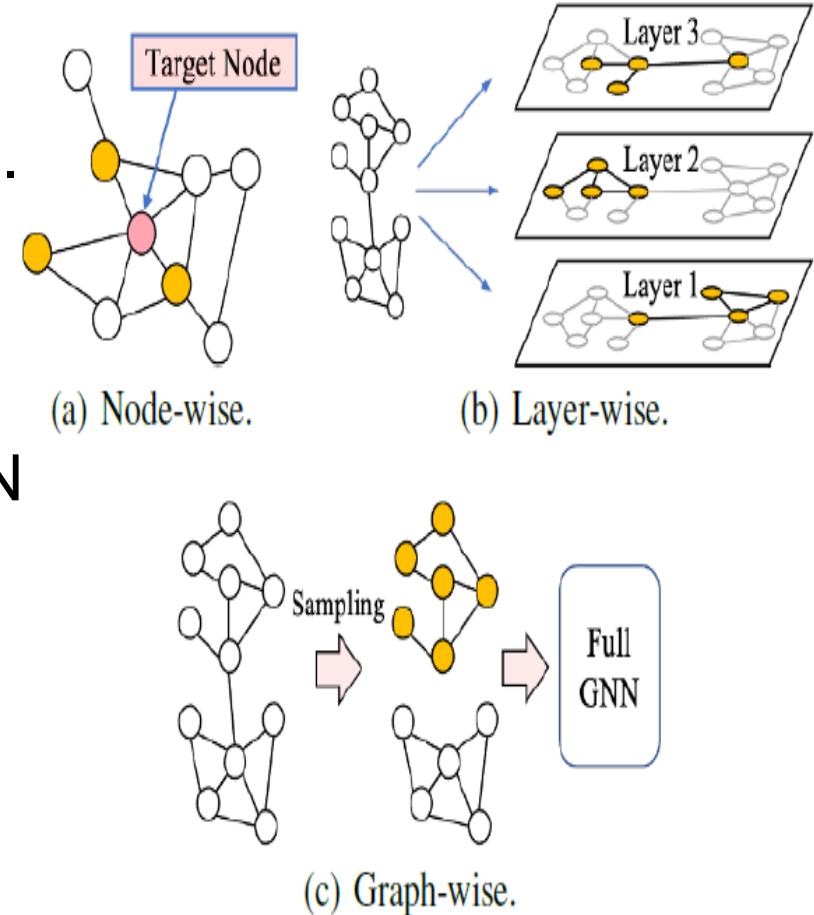
- ❖ Layer-wise sampling and
 - ❖ proposed to address the neighborhood expansion problem occurred during node-wise sampling
 - ❖ Fast Learning Graph Convolutional Networks (Fast-GCN) [Chen et al, 2018c] and Adaptive Sampling Graph Convolutional Networks [Huang et al, 2018]

Sampling Paradigms

-
- ❖ Graphwise sampling.
 - ❖ Designed to further improve the efficiency and scalability, e.g., Cluster Graph Convolutional Networks (Cluster-GCN) [Chiang et al, 2019] and Graph SAmpling based INductive learning meThod (GraphSAINT [Zeng et al, 2020a]

Sampling Paradigms

- ❖ In the ***node-wise sampling***, the nodes are sampled based on the target node in the graph.
- ❖ While in the ***layer-wise sampling***, the nodes are sampled based on the convolutional layers in the GNN models.
- ❖ For the ***graph-wise sampling***, the sub-graphs are sampled from the original graph, and used for the model inference.



References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3
 - Graph Neural Networks: Foundations, Frontiers, and Applications, Editors Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, Springer, 2022.: DOI
<https://doi.org/10.1007/978-981-16-6054-2>
-



Thank You

MESSAGE PASSING WITH SELF-LOOPS



$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} h_v,$$

$$\text{UPDATE}(h_u, m_{\mathcal{N}(u)}) = \sigma(W_{\text{self}}h_u + W_{\text{neigh}}m_{\mathcal{N}(u)}),$$

$$m_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

- ❖ As a simplification of the neural message passing approach, it is common to ***add self-loops*** to the input graph and omit the explicit update step.

$$h_u^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

- ❖ The aggregation is taken over the set $\mathcal{N}(u) \cup \{u\}$, i.e., the node's neighbors as well as the node itself.
- ❖ It is equivalent to sharing parameters between the W_{self} and W_{neigh} matrices, which gives the following graph-level update and we call it is ***Self-loop GNN***.

$$H^{(t)} = \sigma((A + I)H^{(t-1)}W^{(t)}).$$

MESSAGE PASSING WITH SELF-LOOPS



$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} h_v,$$

$$\text{UPDATE}(h_u, m_{\mathcal{N}(u)}) = \sigma(W_{\text{self}}h_u + W_{\text{neigh}}m_{\mathcal{N}(u)}),$$
$$m_{\mathcal{N}(u)} = \text{AGGREGATE}^{(k)}(\{h_v^{(k)}, \forall v \in \mathcal{N}(u)\})$$

$$h_u^{(k)} = \text{AGGREGATE}(\{h_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

❖ Benefits:

- ❖ The benefit is that we no longer need to define an explicit update function.
- ❖ By this we can often alleviate overfitting,

❖ Shortcomings:

- ❖ It also severely limits the expressivity of the GNN, as the information coming from the node's neighbors cannot be differentiated from the information from the node itself.

GENERALIZED NEIGHBORHOOD AGGREGATION



❖ Example:

❖ In a citation graph:- information from very high-degree nodes (i.e., papers that are cited many times) may not be very useful for inferring community membership in the graph, since these papers can be cited thousands of times across diverse subfields. (e.g Gilmer et.al. 2017)

arXiv
<https://arxiv.org> > cs ::

Neural Message Passing for Quantum Chemistry

by J Gilmer · 2017 · Cited by 7000 — In this paper, we reformulate existing models into a single common framework we call **Message Passing Neural Networks** (MPNNs) and explore ...

Graph Convolutional Networks (GCNs)

$$\mathbf{h}_u^{(k)} = \text{AGGREGATE}(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}),$$

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}.$$

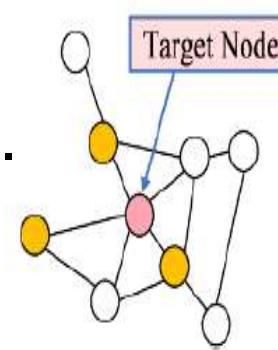
- ❖ One of the most popular baseline graph neural network models is the Graph Convolutional Network (GCN) which employs the symmetric-normalized aggregation as well as the self-loop update approach.
- ❖ The GCN model thus defines the message passing function as:

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \right).$$

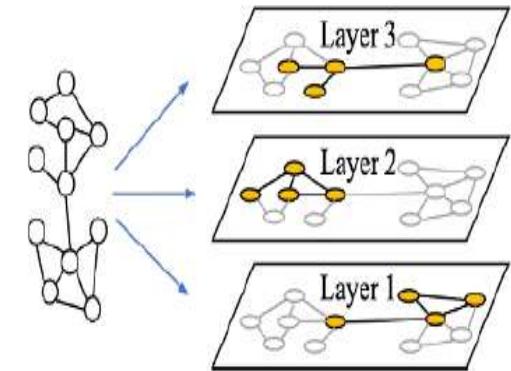
- ❖ σ is the activation function such as a tanh or ReLU.

Sampling Paradigms

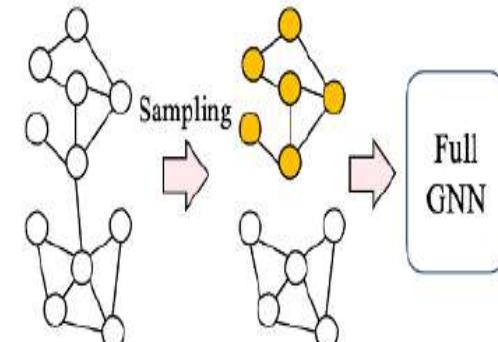
- ❖ In the ***node-wise sampling***, the nodes are sampled based on the target node in the graph.
- ❖ While in the ***layer-wise sampling***, the nodes are sampled based on the convolutional layers in the GNN models.
- ❖ For the ***graph-wise sampling***, the sub-graphs are sampled from the original graph, and used for the model inference.



(a) Node-wise.



(b) Layer-wise.



(c) Graph-wise.

Sampling Paradigms

- ❖ Heterogeneous graph has attracted more and more attention with regards to the rapid growth of data.
- ❖ Large graphs not only include millions of nodes but also various data types.
- ❖ How to train GNNs on such large graphs has become a new domain of interest. Li et al (2019a) proposes a GCN-based Anti-Spam (GAS) model to detect spams by considering both homogeneous and heterogeneous graphs

Li A, Qin Z, Liu R, Yang Y, Li D (2019a) Spam review detection with graph convolutional networks. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management, pp 2703–2711

Node-wise Sampling (GraphSAGE)

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}} \right).$$

- ❖ GraphSAGE can be viewed as an extension of the original Graph Convolutional Network (GCN)
 1. The first extension is the generalized aggregator function.
 - ❖ the embedding generation at the current $(l+1)^{th}$ depth from the target node $v \in V$ can be formulated as, and uses LSTM (Long-short term memory) aggregator and Pooling aggregator to aggregate the information from the neighbors.

$$\mathbf{h}_{\mathcal{N}(v)}^{(l+1)} = \text{AGGREGATE } l \left(\left\{ \mathbf{h}_u^{(l)}, \forall u \in \mathcal{N}(v) \right\} \right),$$

Node-wise Sampling (GraphSAGE)

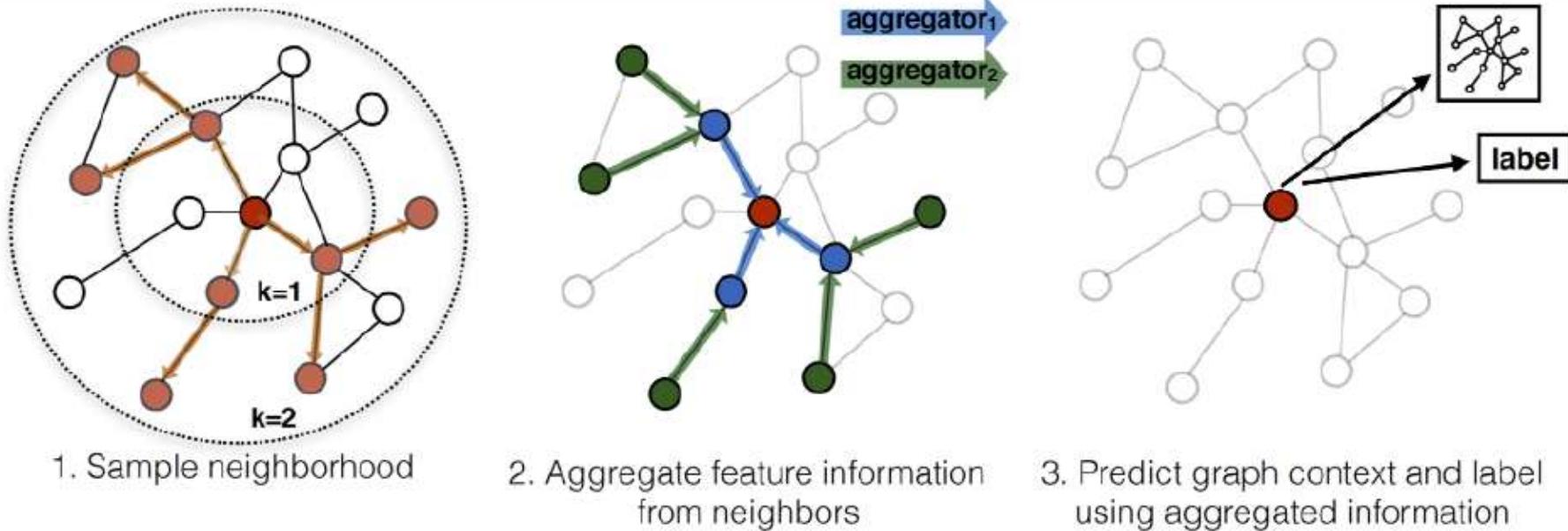
$$\mathbf{h}_{\mathcal{N}(v)}^{(l+1)} = \text{AGGREGATE}_l \left(\left\{ \mathbf{h}_u^{(l)}, \forall u \in \mathcal{N}(v) \right\} \right),$$

- ❖ GraphSAGE can be viewed as an extension of the original Graph Convolutional Network (GCN)
- 2. The second extension is that GraphSAGE applies the concatenation function to combine the information of target node and neighborhoods instead of the summation function:

$$\mathbf{h}_v^{(l+1)} = \sigma \left(\mathbf{W}^{(l+1)} \cdot \text{CONCAT} \left(\mathbf{h}_v^{(l)}, \mathbf{h}_{\mathcal{N}(v)}^{(l+1)} \right) \right),$$

Where $\mathbf{W}^{(l+1)}$ are the weight matrices, and σ is the activation function

Node-wise Sampling (GraphSAGE)



- ❖ Step 1: sample the neighborhoods of the target node;
- ❖ step 2: aggregate feature information from the neighbors;
- ❖ step 3: utilize the aggregated information to predict the graph context or label.

Hamilton WL, Ying R, Leskovec J (2017b) Inductive representation learning on large graphs. In: Advances in Neural Information Processing Systems, pp 1025– 1035



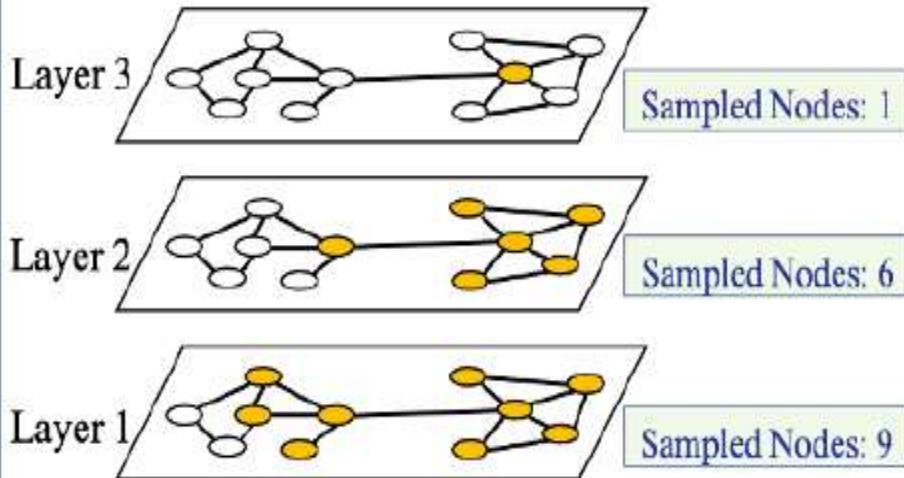
Node-wise Sampling (GraphSAGE)

- ❖ In order to make GNN suitable for the large-scale graphs, GraphSAGE introduces the ***mini-batch training*** strategy to reduce the computation cost during the training phase.
- ❖ Specifically, in each training iteration, only the nodes that are used by computing the representations in the batch are considered, which significantly reduces the number of sampled nodes.

Node-wise Sampling (GraphSAGE)

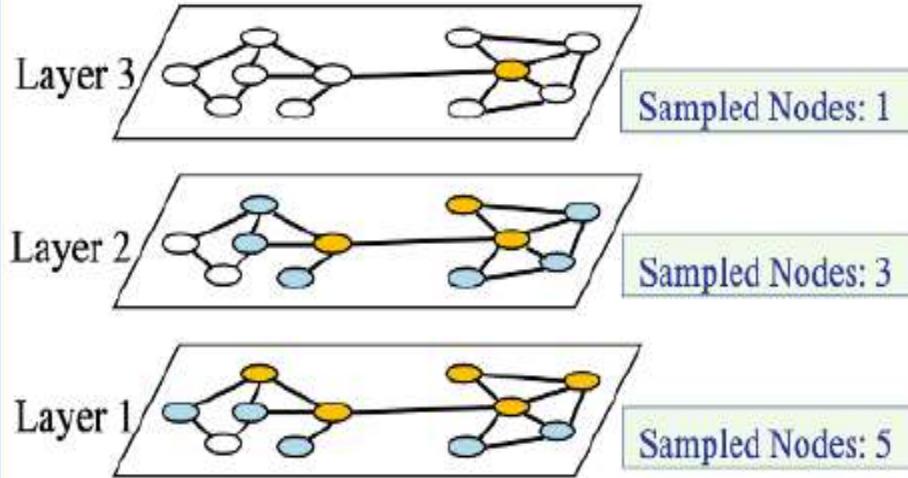
Mini-batch training

● Sampled node
○ Sampling candidates.



(a) Mini-batch training, Batch Size=1

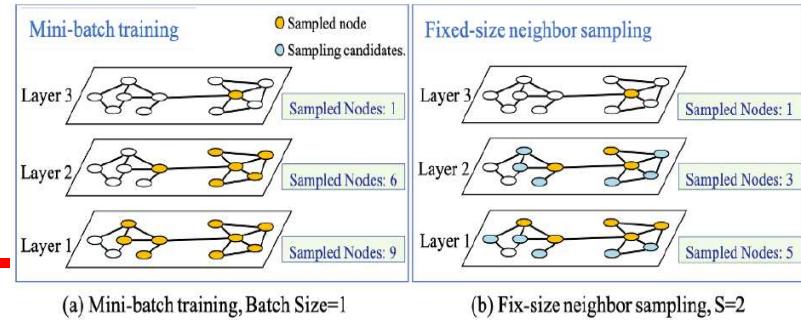
Fixed-size neighbor sampling



(b) Fix-size neighbor sampling, S=2

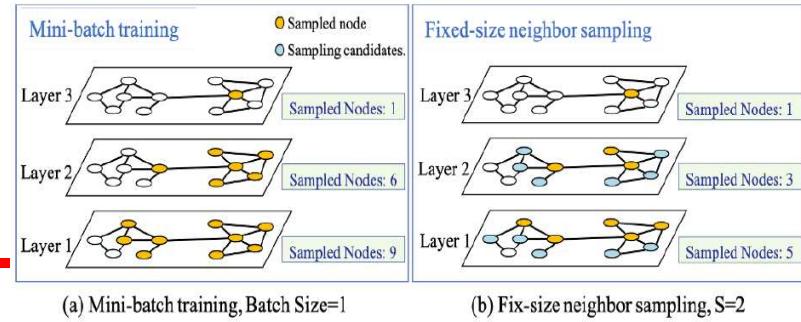
- ❖ Visual comparison between mini-batch training and fixed-size neighbor sampling.

Node-wise Sampling (GraphSAGE)



- ❖ Take layer 2 in Figure (a) as an example, unlike the full-batch training which takes all 11 nodes into consideration, only 6 nodes are involved for mini-batch training.
- ❖ However, the simple implementation of mini-batch training strategy suffers the neighborhood expansion problem.
- ❖ As shown in layer 1 of Figure (a), most of the nodes are sampled since the number of sampled nodes grows exponentially if all the neighbors are sampled at each layer.
- ❖ Thus, all the nodes are selected eventually if the model contains many layers.

Node-wise Sampling (GraphSAGE)



- ❖ To further improve the training efficiency and eliminate the neighborhood expansion problem, GraphSAGE adopts fixed-size neighbor sampling strategy.
- ❖ In specific, a fixed-size set of neighbor nodes are sampled for each layer for computing, instead of using the entire neighborhood sets.
 - ❖ For example, one can set the fixed-size set as two nodes, which is illustrated in Figure (b), the yellow nodes represent the sampled nodes, and the blue nodes are the candidate nodes. It is observed that the number of sampled nodes is significantly reduced, especially for layer

Layer-wise Sampling

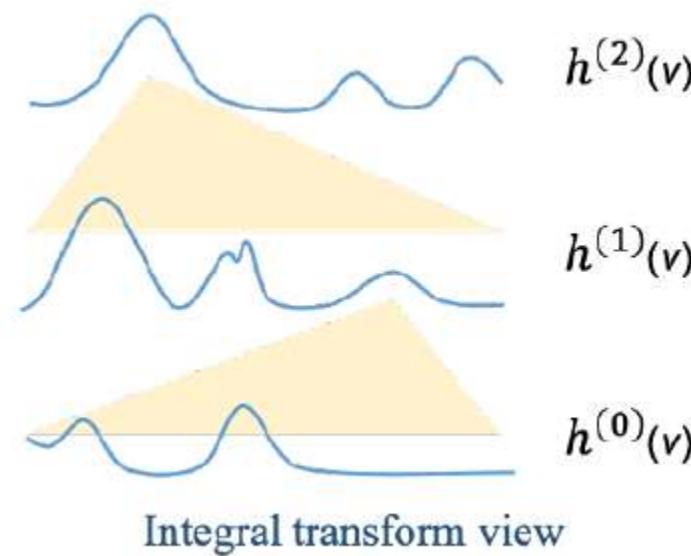
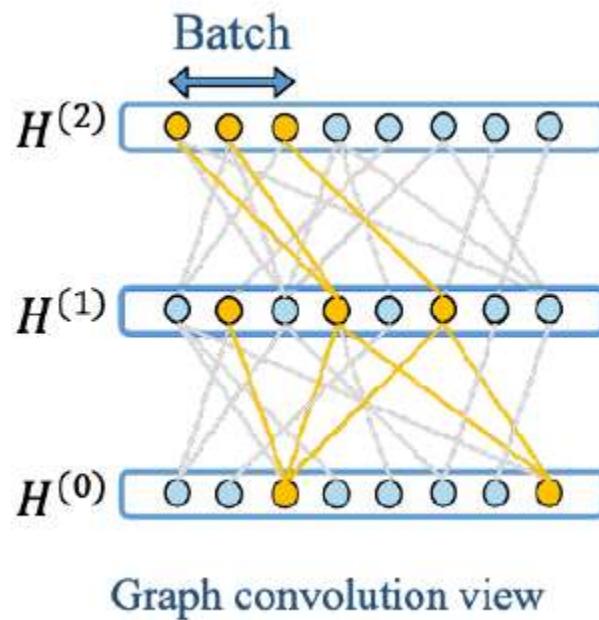
- ❖ Multiple vertices may have some common neighbors so neighbor sampling can result in repeated samples. We can avoid the over-expansion and accelerate the training of GCN by controlling the size of the sampled neighborhoods in each layer.
 - ❖ Since node-wise sampling can only alleviate but not completely solve the neighborhood expansion problem, layer-wise sampling has been studied to address this obstacle.
 - ❖ FastGCN:
-

FastGCN

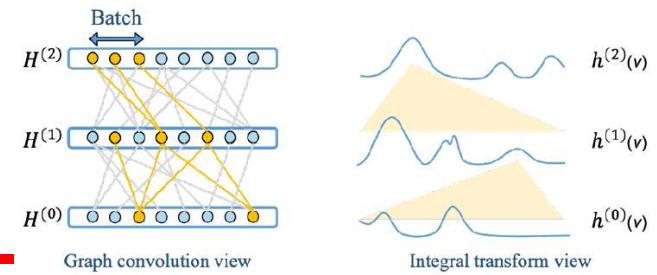
- ❖ First proposes to understand the GNN from the functional generalization perspective.
 - ❖ The researcher point out that training algorithms such as stochastic gradient descent are implemented according to the ***additivity of the loss function*** for independent data samples.
- ❖ FastGCN converts the common graph convolution view to an integral transform view by introducing a probability measure for each node.

FastGCN

- ❖ Two views of GCN. The circles represent the nodes in the graph, while the yellow circles indicate the sampled nodes. The lines represent the connection between nodes.



FastGCN



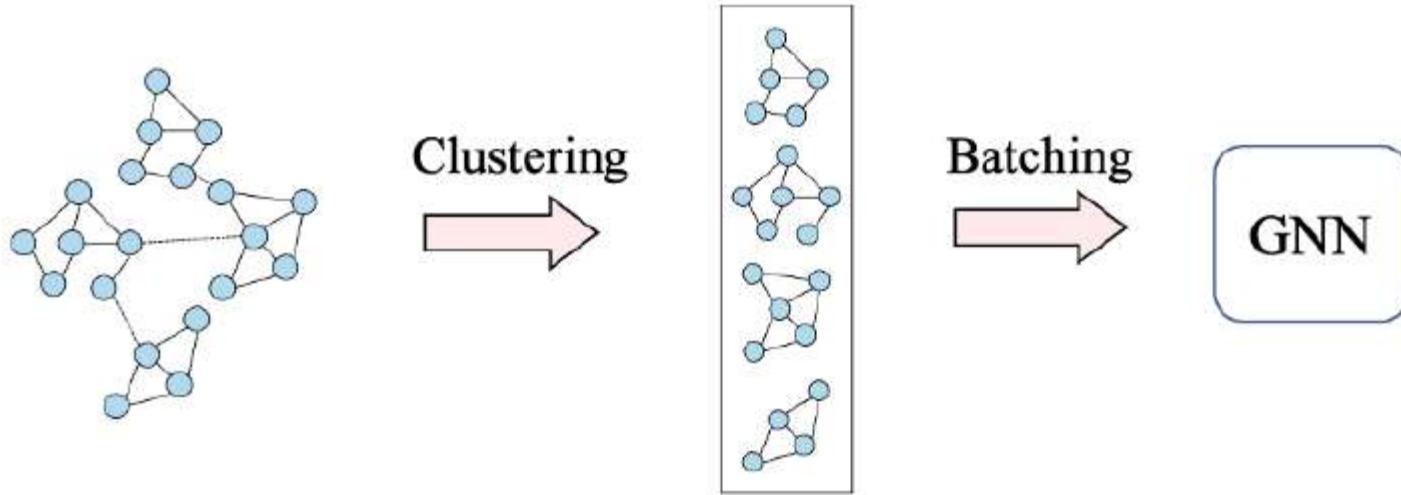
- ❖ Figure shows the conversion between the traditional graph convolution view and the integral transform view.
- ❖ In the graph convolution view, a fixed number of nodes are sampled in a bootstrapping manner in each layer, and are connected if there is a connection exists.
 - ❖ Each convolutional layer is responsible for integrating the node embeddings.
- ❖ The integral transform view is visualized according to the probability measure, and the integral transform (demonstrated in the yellow triangle form) is used to calculate the embedding function in the next layer.

More details can be found in (Chen et al, 2018c).

Chen J, Ma T, Xiao C (2018c) Fastgcn: Fast learning with graph convolutional networks via importance sampling. In: International Conference on Learning Representations

Graph-wise Sampling

An illustration of graph-wise sampling on large-scale graph.



- ❖ Graph-wise sampling is used to accomplish efficient training on large-scale graphs.
- ❖ Whole graph can be sampled into several sub-graphs and fit into the GNN models, in order to reduce the computational cost.

Graph-wise Sampling (Cluster-GCN)

- ❖ Cluster-GCN first proposes to extract small graph clusters based on efficient graph clustering algorithms.
 - ❖ The intuition is that the mini-batch algorithm is correlated with the number of links between nodes in one batch. Hence, Cluster-GCN constructs mini-batch on the sub-graph level, while previous studies usually construct mini-batch based on the nodes
- ❖ Cluster-GCN extracts small clusters based on any clustering algorithms.

Graph-wise Sampling (Cluster-GCN)

A graph $G(V, E)$ can be divided into c portions by grouping its nodes, where $V = [V_1, \dots, V_c]$. The extracted sub-graphs can be defined as:

$$\bar{g} = [g_1, \dots, g_c] = [\{V_1, E_1\}, \dots, \{V_c, E_c\}];$$

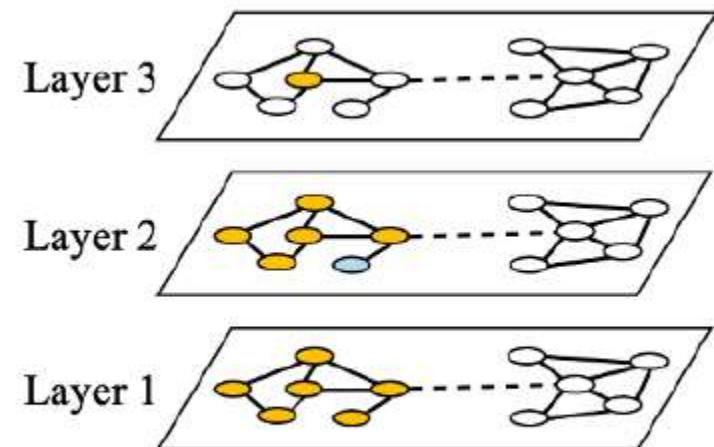
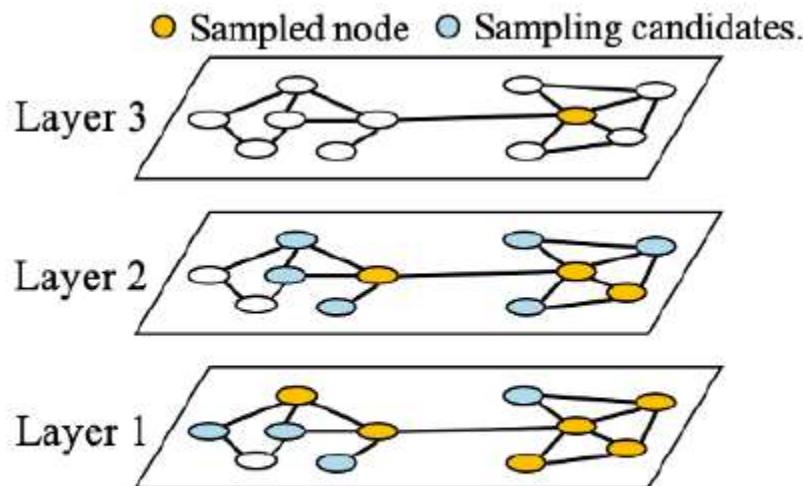
$\{V_t, E_t\}$, represent the nodes and links within the t^{th} partition, $t \in (1, c)$ and re-ordered adjacency matrix can be written as:

$$A = \bar{A} + \Delta = \begin{bmatrix} A_{11} & \cdots & A_{1c} \\ \vdots & \ddots & \vdots \\ A_{c1} & \cdots & A_{cc} \end{bmatrix}; \quad \bar{A} = \begin{bmatrix} A_{11} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & A_{cc} \end{bmatrix}, \Delta = \begin{bmatrix} 0 & \cdots & A_{1c} \\ \vdots & \ddots & \vdots \\ A_{c1} & \cdots & 0 \end{bmatrix}.$$

Graph-wise Sampling (Cluster-GCN)



Comparison between GraphSAGE and Cluster-GCN. In Cluster-GCN, it only samples the nodes in each sub-graph.



GraphSAGE: Fix-size neighbor sampling $S=2$

Graph-wise Sampling (Cluster-GCN)

- ❖ Obviously, this strategy can avoid the neighbor expansion problem since it only samples the nodes in the clusters, as shown in Figure.
- ❖ For Cluster-GCN, since there is no connection between the sub-graphs, the nodes in other sub-graphs will not be sampled when the layer increases.
- ❖ In such a manner, the sampling process establishes a neighbor expansion control by sampling over the sub-graphs, while in layer-wise sampling the neighbor expansion control is implemented by fixing the neighbour sampling size.

Graph-wise Sampling (Cluster-GCN)

- ❖ In summary, Cluster-GCN is a practical solution based on the sub-graph batching.
- ❖ It has good performance and good memory usage, and can alleviate the neighborhood expansion problem in traditional mini-batch training.
- ❖ However, Cluster-GCN does not analyze the sampling quality, e.g., the bias and variance of this sampling strategy.
- ❖ In addition, the performance is highly correlated to the clustering algorithm.

Graph Attention Network

- ❖ In addition to more general forms of set aggregation, a popular strategy for improving the aggregation layer in GNNs is to apply *attention*.
- ❖ The basic idea is to assign an ***attention weight*** or *importance* to each neighbor, *which is used to weigh this neighbor's influence during the aggregation step.*

Graph Attention Network

- ❖ In GCNs, for a target node i , the importance of a neighbor j is determined by the weight of their edge A_{ij} (normalized by their node degrees).
- ❖ However, in practice, the input graph may be noisy. The edge weights may not be able to reflect the true strength between two nodes, so we need to learn the importance of each neighbour, based on the Attention mechanism.

Graph Attention Network

- ❖ Graph Attention Network (GAT), which uses attention weights to define a weighted sum of the neighbors:

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v,$$

- ❖ where $\alpha_{u,v}$ denotes the attention on neighbor $v \in N(u)$ when we are aggregating information at node u .



Graph Attention Network

$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} h_v,$$

- ❖ In the original GAT paper, the attention weights are defined as

$$\alpha_{u,v} = \frac{\exp(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v])}{\sum_{v' \in \mathcal{N}(u)} \exp(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}])},$$

- ❖ where \mathbf{a} is a trainable attention vector, \mathbf{W} is a trainable matrix, and \oplus denotes the concatenation operation.
- ❖ Next, we will understand, how attention is used in graph neural networks?

Graph Attention Layer

- ❖ The graph attention layer defines how to transfer the hidden node representations at layer $k-1$ (denoted as $H^{k-1} \in R^{N \times F}$) to the new node representations $H^k \in R^{N \times F'}$
- ❖ To transform the lower-level node representations to higher-level node representations, a shared linear transformation is applied to every node, denoted as $W \in R^{F \times F'}$.
- ❖ Afterwards, self-attention is defined on the nodes, which measures the attention coefficients for any pair of nodes through a shared attentional mechanism $a : R^{F'} \times R^{F'} \rightarrow R$

$$e_{ij} = a(WH_i^{k-1}, WH_j^{k-1}).$$



Graph Attention Layer

$$e_{ij} = a(WH_i^{k-1}, WH_j^{k-1}).$$

- ❖ e_{ij} indicates the relationship strength between node i and j .
- ❖ For each node, we can theoretically allow it to attend to every other node on the graph, which however will ignore the graph structural information.
- ❖ A more reasonable solution would be only to attend to the neighbors for each node.
- ❖ In practice, the first-order neighbors are only used (including the node itself).

Graph Attention Layer

- ❖ And to make the coefficients comparable across different nodes, the attention coefficients are usually normalized with the softmax function:

$$\alpha_{ij} = \text{Softmax}_j(\{e_{ij}\}) = \frac{\exp(e_{ij})}{\sum_{l \in N(i)} \exp(e_{il})}.$$

- ❖ We can see that for a node i , α_{ij} essentially defines a multinomial distribution over the neighbors,
- ❖ It can also be interpreted as the transition probability from node i to each of its neighbors.

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v,$$

$$\alpha_{u,v} = \frac{\exp(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v])}{\sum_{v' \in \mathcal{N}(u)} \exp(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}])},$$

Graph Attention Network

- ❖ Is a useful strategy ***for increasing the representational capacity*** of a GNN model, especially in cases *where you have prior knowledge to indicate that some neighbors might be more informative than others*.
- ❖ For example,
 - ❖ In case of classifying papers into topical categories, based on, citation networks. Often there are papers that span topical boundaries, or that are highly cited across various different fields.
 - ❖ Ideally, an attention-based ***GNN would learn to ignore these papers*** in the neural message passing, as such numerous neighbors would likely be uninformative when trying to identify the topical category of a particular node.

Multi-head Attention

- ❖ In practice, instead of only using one single attention mechanism, multi-head attention can be used, each of which determines a different similarity function over the nodes.
- ❖ For each attention head, we can independently obtain a new node representation according to Equation

$$H_i^k = \sigma \left(\sum_{j \in N(i)} \alpha_{ij} W H_j^{k-1} \right)$$

- ❖ The final node representation will be a concatenation of the node representations learned by different attention heads. Mathematically, we have

$$H_i^k = \left\|_{t=1}^T \sigma \left(\sum_{j \in N(i)} \alpha_{ij}^t W^t H_j^{k-1} \right) \right\|$$

- ❖ σ is the Leaky ReLU activation function



Multi-head Attention

$$H_i^k = \left\|_{t=1}^T \sigma \left(\sum_{j \in N(i)} \alpha_{ij}^t W^t H_j^{k-1} \right) \right\|$$

- ❖ where T is the total number of attention heads, $\alpha_{i,j}^t$ is the attention coefficient calculated from the t^{th} attention head, W^t is the linear transformation matrix of the t^{th} attention head.
- ❖ As per the paper by Veličković et al (2018) is that in the final layer, when trying to combine the node representations from different attention heads, instead of using the operation concatenation, other pooling techniques could be used, e.g. *simply taking the average node representations from different attention heads*.

Over-smoothing

- ❖ One common issue with GNNs is *over-smoothing*.
- ❖ After several iterations of GNN message passing, the representations for all the nodes in the graph can become very similar to one another and makes it impossible to build deeper GNN models.
- ❖ PairNorm is method for alleviating the problem of over-smoothing when GNNs go deep.
- ❖ It keep the ***Total Pairwise Squared Distance*** (TPSD) of node representations unchanged, which is the same as that of the original node feature X .

Over-smoothing

- ❖ Let \tilde{H} be the output of the node representations by the graph convolution, which will be the input of *PairNorm*, and \hat{H} is the output of *PairNorm*.
- ❖ The goal of *PairNorm* is to normalize the \tilde{H} such that after normalization $\text{TPSD}(\hat{H}) = \text{TPSD}(X)$

$$\sum_{(i,j) \in \mathcal{E}} \|\hat{H}_i - \hat{H}_j\|^2 + \sum_{(i,j) \notin \mathcal{E}} \|\hat{H}_i - \hat{H}_j\|^2 = \sum_{(i,j) \in \mathcal{E}} \|X_i - X_j\|^2 + \sum_{(i,j) \notin \mathcal{E}} \|X_i - X_j\|^2.$$

Over-smoothing

- ❖ It is very computationally expensive, which is quadratic to the number of nodes N .
- ❖ So in the proposed update, in method, ***PairNorm*** can be divided into two steps: center and scale,

$$\tilde{H}_i^c = \tilde{H}_i - \frac{1}{N} \sum_{i=1}^N \tilde{H}_i \quad (\text{Center})$$

$$\hat{H}_i = s \cdot \frac{\tilde{H}_i^c}{\sqrt{\frac{1}{N} \sum_{i=1}^N \|\tilde{H}_i^c\|_2^2}} = s\sqrt{N} \cdot \frac{\tilde{H}_i^c}{\sqrt{\|\tilde{H}^c\|_F^2}} \quad (\text{Scale}),$$



References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3
- Graph Neural Networks: Foundations, Frontiers, and Applications, Editors Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, Springer, 2022.: DOI
<https://doi.org/10.1007/978-981-16-6054-2>



BITS Pilani
Pilani Campus



Thank You



Graph Neural Networks (S*-**_AIMLCZG514)

BITS Pilani
Pilani Campus



Module V

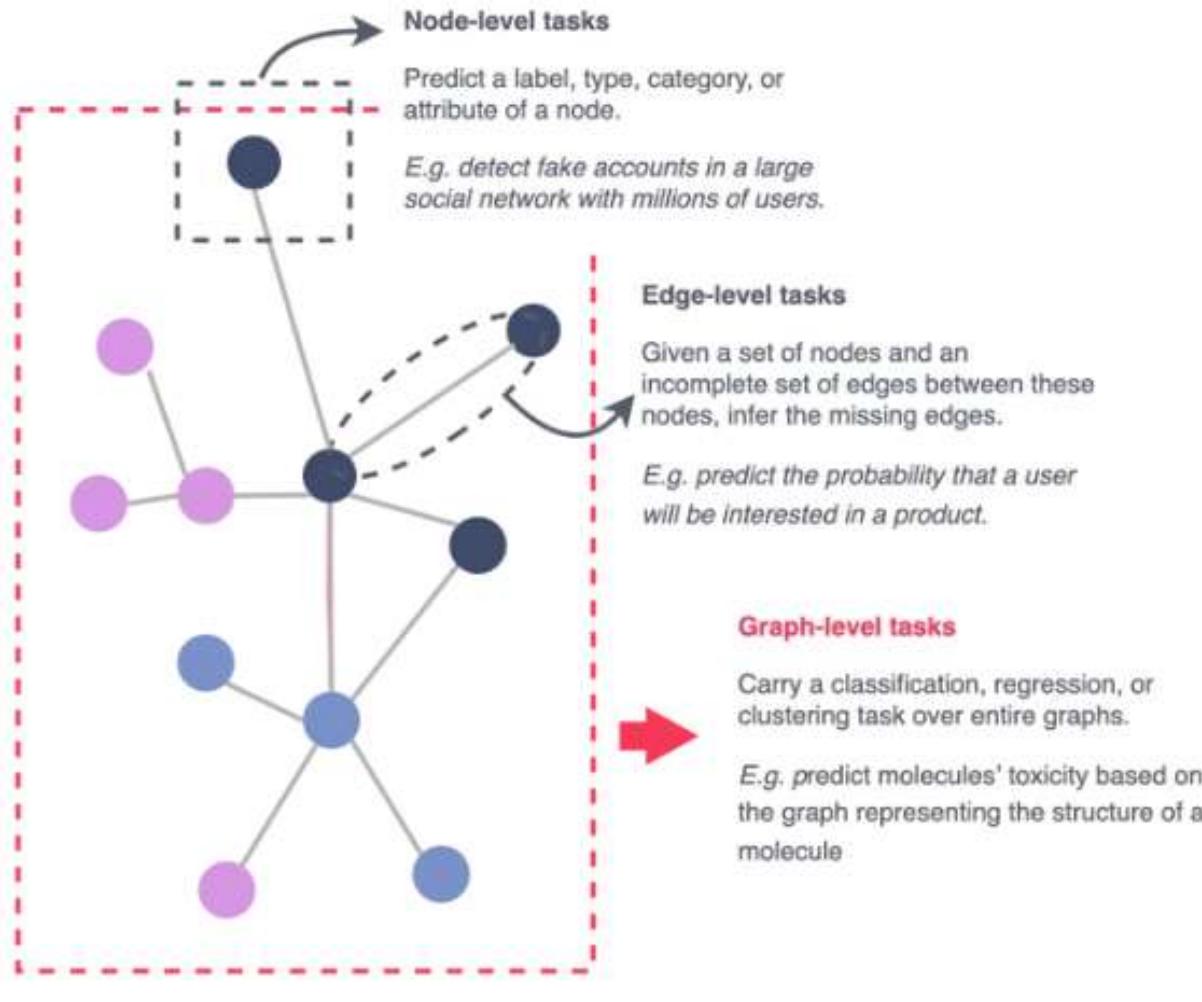


Node & Graph Classification (Application)

Introduction

- ❖ Recently, **Graph Neural Networks (GNNs)** have received a lot of attention. From marketing to social science to biology, they have been widely promoted as the new way of learning “smartly” from data. It’s more than a trend, though, as many research papers have proven that they can actually lead to more accurate and robust models.
- ❖ The most common applications of GNN models to real-world problems are :
 - ❖ Node classification,
 - ❖ Link prediction and
 - ❖ Graph classification

Types of supervised tasks for graph-based models.



Prediction Tasks on Graphs

❖ Node-level tasks: Node classification and regression

- ❖ **Goal:** Predict a label, type, category, or attribute of a node.
- ❖ **Example:** Given a large social network with millions of users, detect fake accounts.

❖ Edge-level tasks: Link prediction

- ❖ **Goal:** Given a set of nodes and an incomplete set of edges between these nodes, infer the missing edges.
- ❖ **Example:** Predict biological interactions between proteins.

❖ Graph-level tasks: Graph classification, regression, and clustering

- ❖ **Goal:** Carry a classification, regression, or clustering task over entire graphs.
- ❖ **Example:** Given a graph representing the structure of a molecule, predict molecules' toxicity.

Example:

- ❖ You like to classify products sold in a store.
- ❖ You would probably gather information about the products (description, price, brand, etc.) and use it as input to train a model. But what if this information is non-existent or too poor to build a robust model, or too big to handle for a deterministic process?

Example:

- ❖ You like to classify products sold in a store.
 - ❖ In this context, you can leverage the graphical aspect of the problem.
 - ❖ Each product can be represented as a node and each pair of products frequently bought together can be linked.
 - ❖ A graph-based model is then likely to perform better than a “traditional” machine learning algorithm, as it would learn not only from information about products but also from the relationships between them.
 - ❖ In fact, instead of considering each product independently, it would leverage this additional information to detect valuable patterns.

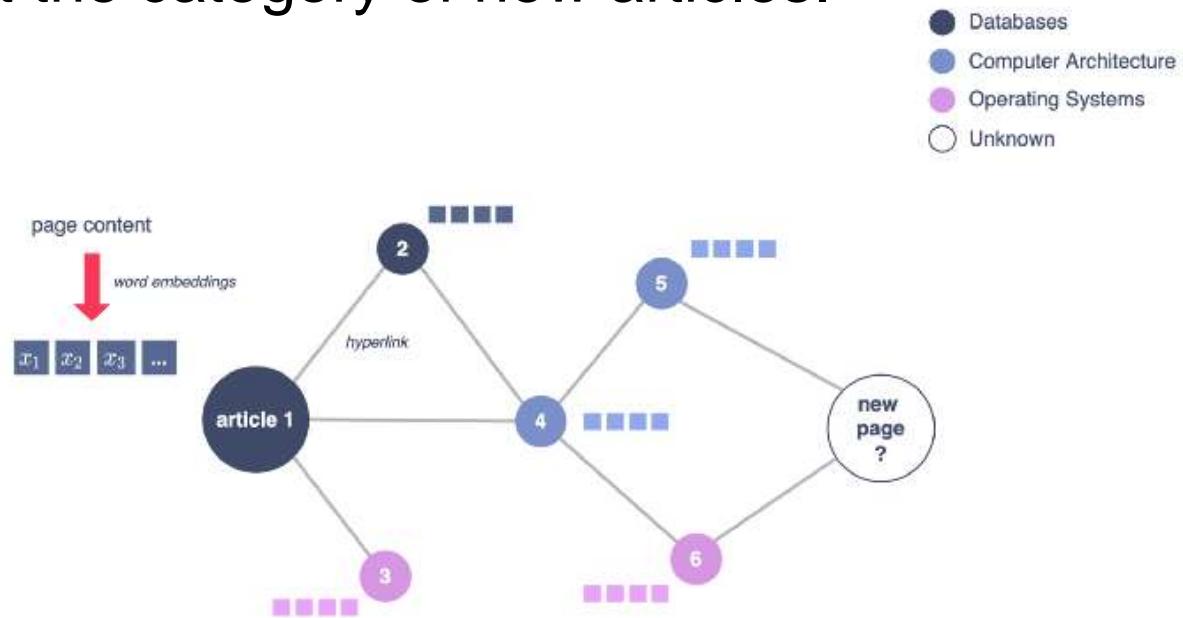


Node Classification With GNN

- ❖ Imagine that you run a large online knowledge-sharing platform such as Wikipedia. Every day, thousands of scientific articles are published.
- ❖ To help your readers easily navigate the platform and find the content they are interested in, you need to make sure that each article is classified into the right category quickly after its publication.

Node Classification With GNN

- ❖ In this context, the problem can be modeled as a graph where each node represents an article and has as an attribute an embedding of the content. Two articles are linked if one of them contains a link to the other. The goal is to predict the category of new articles.



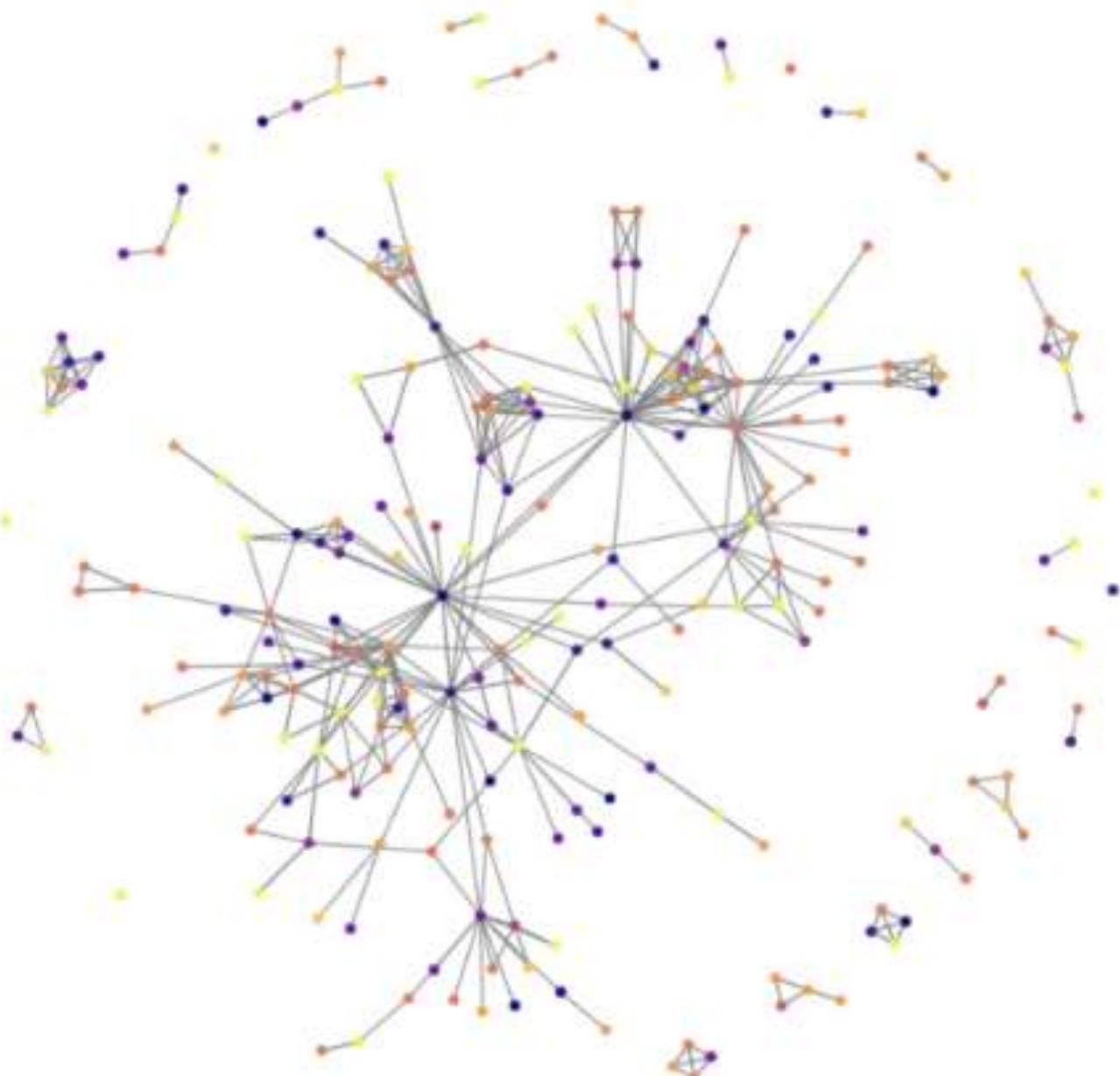


Node Classification With GNN

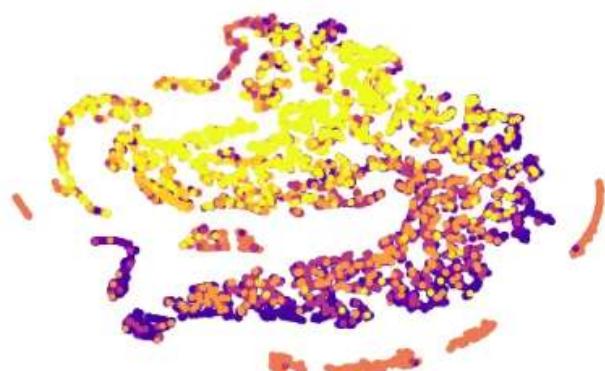
- ❖ The researcher with the name Peter et. al. (2020) had published a work (Dataset) for Node Classification in “*Wiki-CS: A Wikipedia-Based Benchmark for Graph Neural Networks*”.
- ❖ It consists of nodes corresponding to computer science articles, with edges based on hyperlinks and 10 classes that represent different branches of the field.

-
- article 0
 - article 1
 - article 2
 - article 3
 - article 4
 - article 5
 - article 6
 - article 7
 - article 8
 - article 9

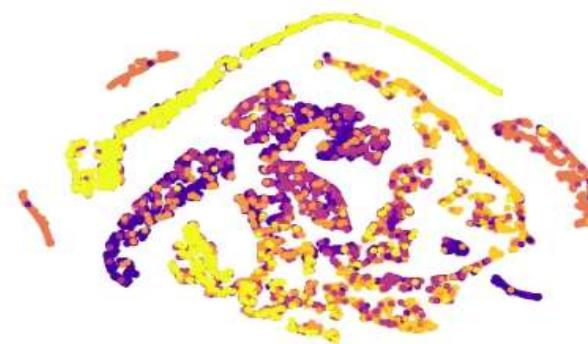
lead



Embeddings epoch 10

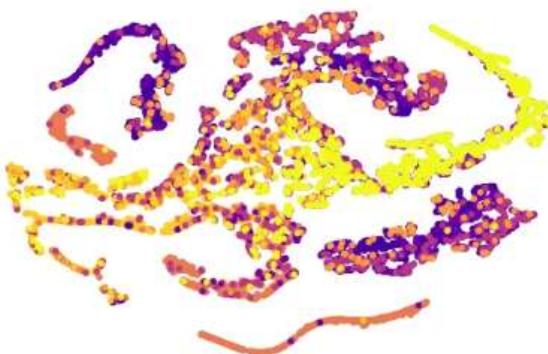


embeddings epoch 30



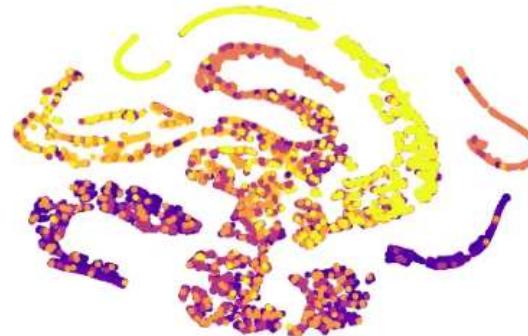
- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9

Embeddings epoch 20



- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9

Embeddings epoch 40

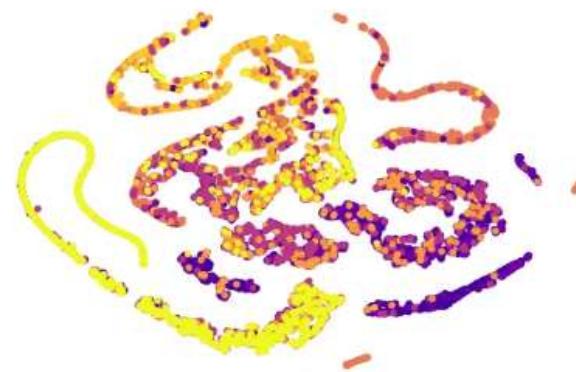


- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9

Embeddings epoch 50



Embeddings epoch 80

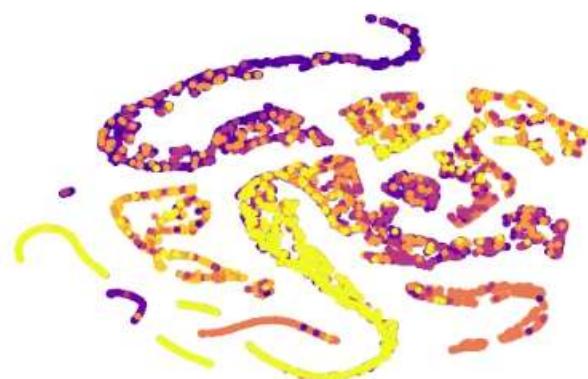


- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9

Embeddings epoch 60



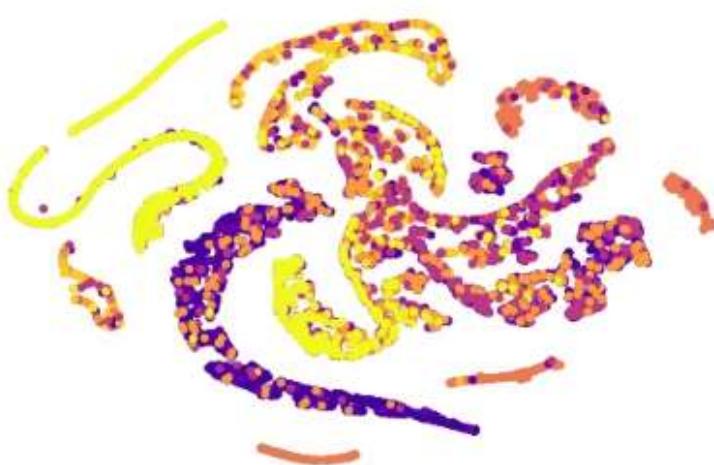
Embeddings epoch 70



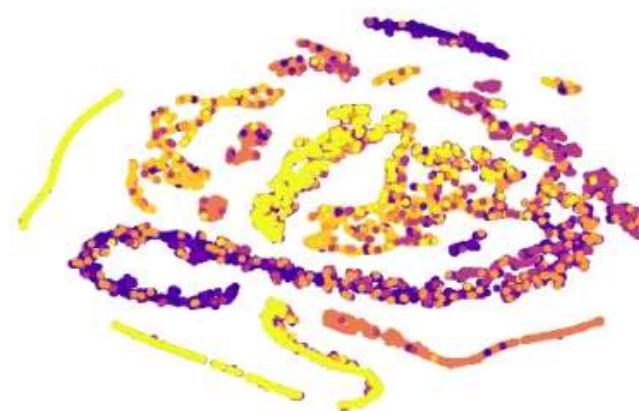
- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9

- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9

Embeddings epoch 90



- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9



- Class 0
- Class 1
- Class 2
- Class 3
- Class 4
- Class 5
- Class 6
- Class 7
- Class 8
- Class 9



Graph Classification

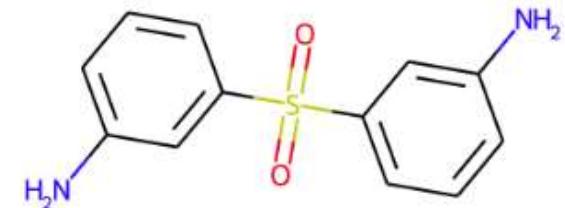
Graph Classification

- ❖ Many real-life situations can be modeled as graphs, but turning the relational structure of these graphs into valuable information that can help solve complex tasks is a real challenge.
- ❖ Take the example of **drug discovery**.
 - ❖ In the early stages of drug development, scientists need to screen large libraries often composed of hundreds of thousands of compounds (drug candidates) against targets (biological events). This requires the use of an arsenal of tools such as robotics, data processing and control software, and sensitive detectors. Such approaches can be very time consuming and expensive, with a very low hit rate (a typical hit rate is less than 1% in most assays!).

Graph Classification

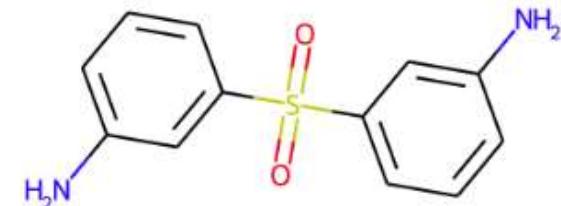
- ❖ we will develop a use case **predicting the toxicity of a molecule**. We'll use its representation as a graph where the nodes are atoms connected by edges corresponding to chemical bonds.
- ❖ The dataset used for the experiments contains graphs of 7,831 molecules. It comes from MoleculeNet (Tox-21) with a node and edge enrichment introduced by the Open Graph Benchmark.

Graph Classification

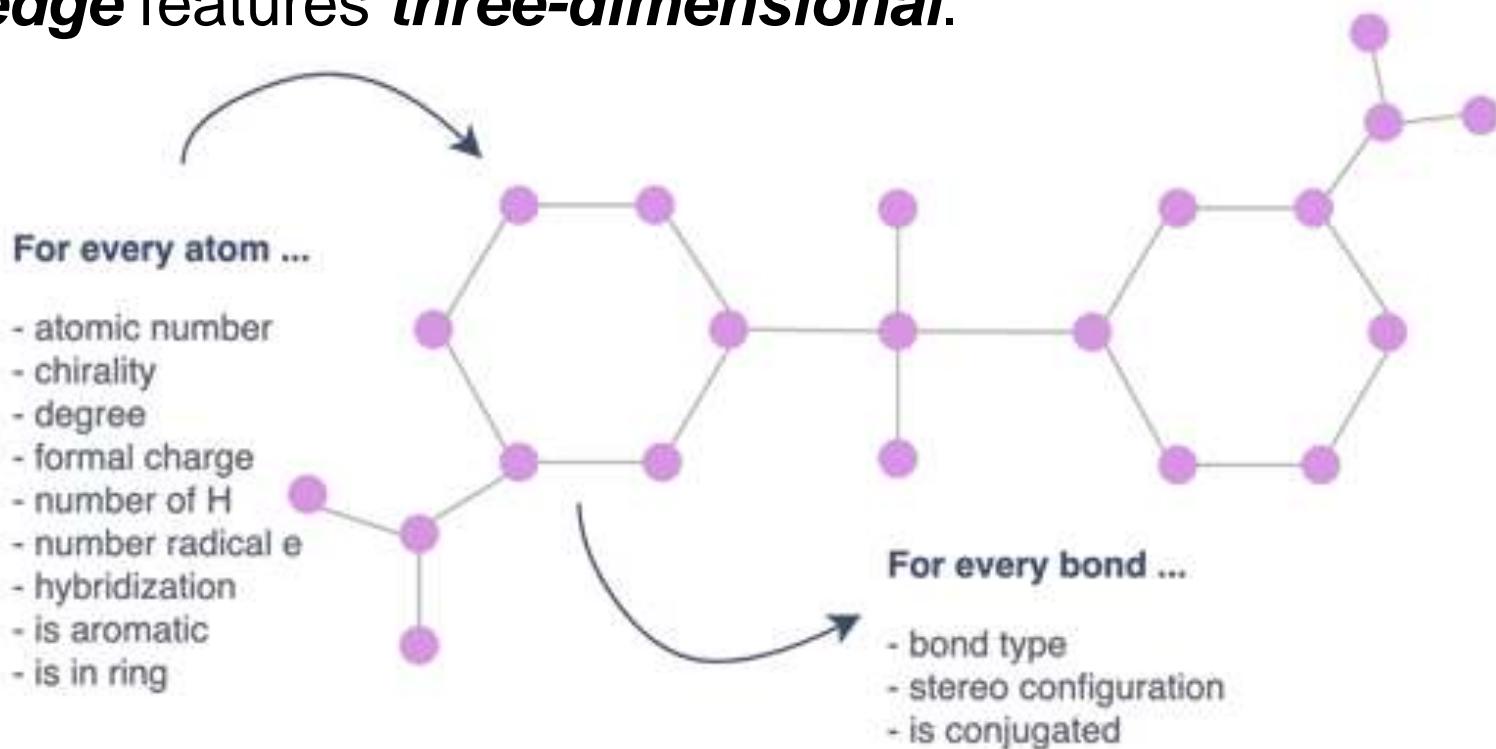


- ❖ Converting this into a graph consists mainly of representing each atom by a node and replacing the bonds with edges.
- ❖ These nodes and edges are further enriched with various features to avoid losing valuable information such as the name of the atom or the type of bond.
- ❖ In total, input **node** features are ***nine-dimensional*** and **edge** features ***three-dimensional***.

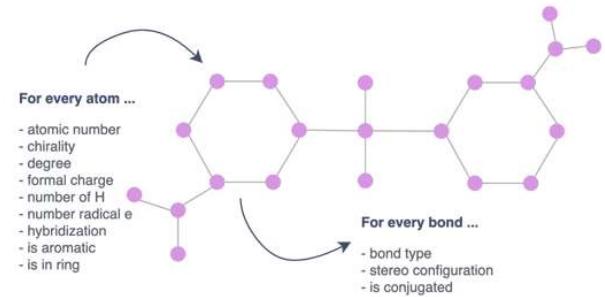
Graph Classification



- ❖ In total, input **node** features are ***nine-dimensional*** and **edge** features ***three-dimensional***.



Graph Classification



❖ Node features

- ❖ **Atomic number:** Number of protons in the nucleus of an atom. It's characteristic of a chemical element and determines its place in the periodic table.
- ❖ **Chirality:** A molecule is chiral if it is distinguishable from its mirror image by any combination of rotations, translations, and some conformational changes. Different types of chirality exist depending on the molecule and the arrangement of the atoms.
- ❖ **Degree:** Number of directly-bonded neighbors of the atom.
- ❖ **Formal charge:** Charge assigned to an atom. It reflects the electron count associated with the atom compared to the isolated neutral atom.
- ❖ **Number of H:** Total number of hydrogen atoms on the atom.
- ❖ **Number of radical e:** Number of unpaired electrons of the atom.
- ❖ **Hybridization:** Atom's hybridization.
- ❖ **Is aromatic:** Whether it is included in a cyclic structure with pi bonds. This type of structure tends to be very stable in comparison with other geometric arrangements of the same atoms.
- ❖ **Is in ring:** Whether it is included in a ring (a simple cycle of atoms and bonds in a molecule).

❖ Edge features:

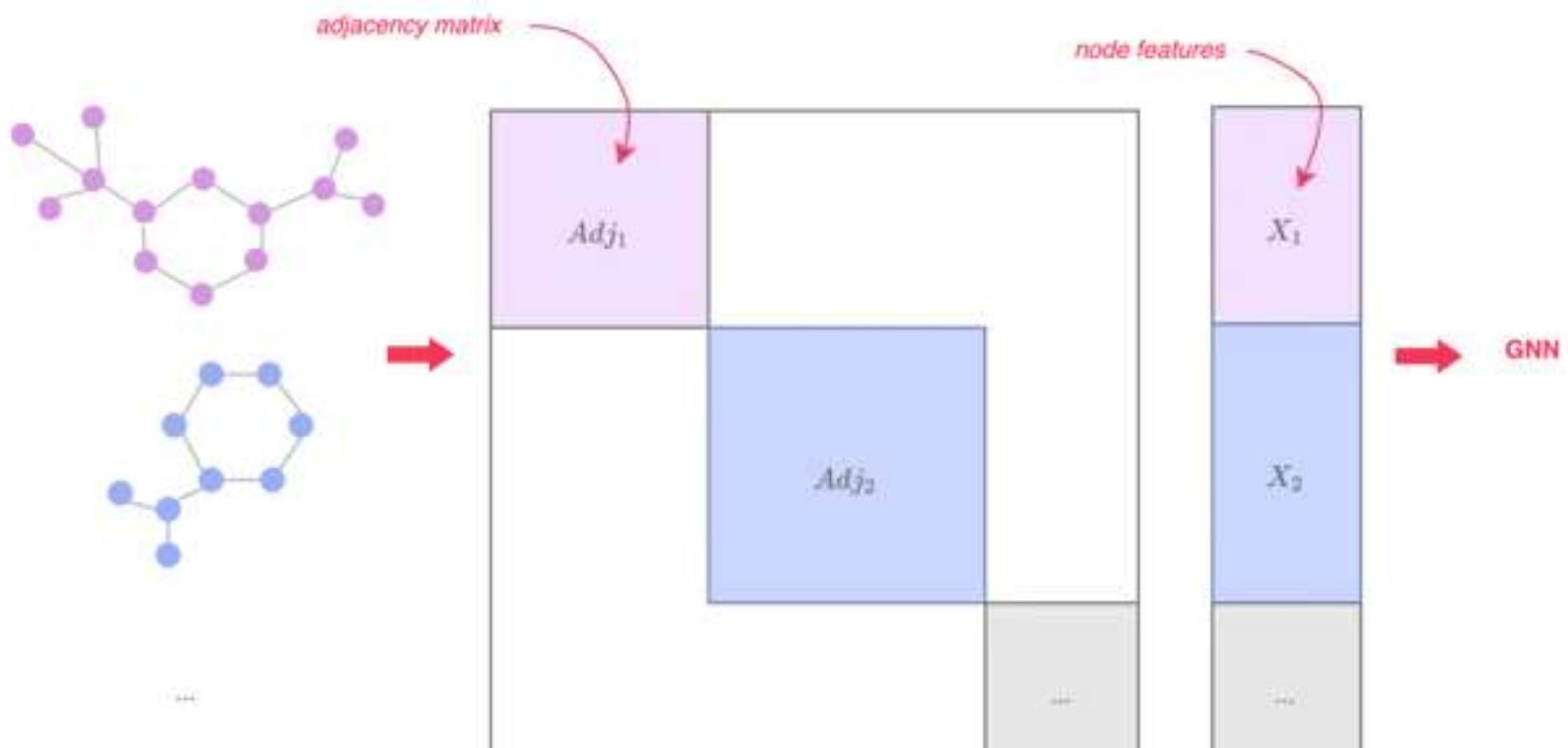
- ❖ **Bond type:** Whether the bond is single, double, triple, or aromatic.
- ❖ **Stereo configuration** of the bond.
- ❖ **Is conjugated:** Whether or not the bond is considered to be conjugated.

Learning From Multiple Graphs at Once



- ❖ As graphs tend to be small, it's better to use **batches of graphs** instead of individual graphs before inputting them into a GNN.
- ❖ In NLP or computer vision, this is typically done by rescaling or padding each element into a set of equally-sized shapes. For graphs, those approaches are not feasible. Instead, we can:
 - ❖ Stack adjacency matrices in a diagonal manner leading to a large graph with multiple isolated subgraphs.
 - ❖ Concatenate node features and the target.

Learning From Multiple Graphs at Once



Mini-batching of graphs,

Learning From Multiple Graphs at Once



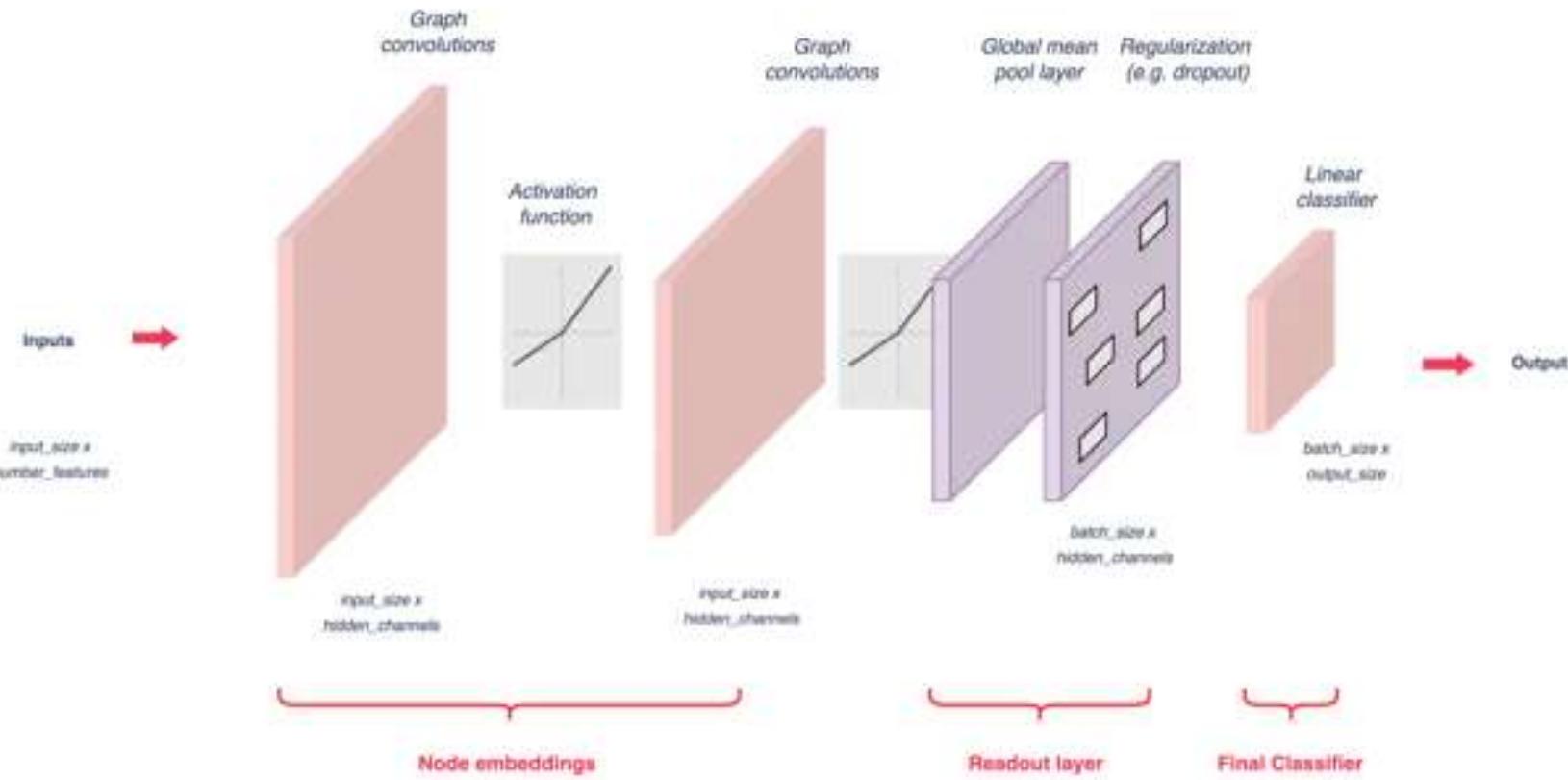
- ❖ It has two main advantages:
 - ❖ It does not require changing GNN operators using a message passing scheme as, by construction, messages are not exchanged between two nodes of different graphs.
 - ❖ There is no risk of having a computational or memory overload since the adjacency matrices are saved sparsely (only the non-zero entries which correspond to the edges are kept).



Model Architecture

- ❖ The model learns to classify graphs using three main steps:
 1. Embed nodes using several rounds of message passing.
 2. Aggregate these node embeddings into a single graph embedding.
 3. Train a classifier based on graph embeddings.

Model Architecture



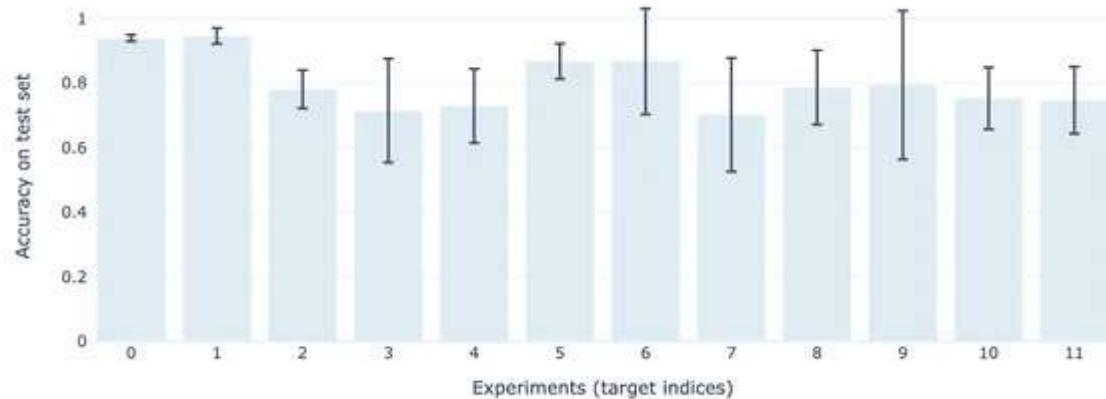
GNN model architecture,

Methodology

- ❖ **Target.** As seen in the first section, the molecules are labeled depending on the experiment. For this reason, each experiment outcome is taken as an individual classification task.
- ❖ **Train / Test sets.** The data is divided into two datasets: a training set, which contains around 70% of the total number of graphs, and a test set that contains the rest of the graph. This split is randomly done three times for each model.
- ❖ **Model evaluation.** The models are trained using cross-entropy loss with class weights. They are evaluated according to the mean accuracy measured on the test sets, as well as other common classification metrics.
- ❖ **Hyperparameters.** For each target, multiple parameters are tested such as the type and number of GNN convolutions used for node embedding (e.g., GCN, GAT, etc.), the latent dimension of node embedding, and the learning rate.
- ❖ **Features.** Node features are all used when training the model. Concerning the edge features, the possibility of using the type of bonds (single, double, or triple) was also tested.

Results

- ❖ The results show satisfactory accuracy on average. However, the accuracy depends to a large extent on the number of labeled molecules (the higher the better) and the percentage of positive outcomes (the lower the worse).
- ❖ As for the hyperparameters, the best combination also differs depending on the target. Hence, it is difficult to draw a general rule.





Graph Classification (Theory & Concept)



BITS Pilani
Pilani Campus



Thank You



Lecture 9

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- ▶ We shall look at theoretical motivations for Graph Neural Networks arising out of considerations of signal processing.
- ▶ There are actually a wide variety of approaches which all converge in the development of the theory of graph neural networks.
- ▶ Convolution-based approaches to the developing the theory of GNN owe their origin to convolutions in the signal processing domain.
- ▶ Another source for the theory of neural message passing in GNNs comes from message passing in the context of probabilistic graphical models.
- ▶ A third approach to GNN theory is the connection to Weisfeiler-Lehman graph isomorphism test.

- ▶ We shall borrow heavily from the approach of Alejandro Ribeiro from UPenn - the URL for Lecture 3 which we will heavily use is
<https://gnn.seas.upenn.edu/lectures/lecture-3/>
- ▶ We shall first start with the notion of a Graph Shift Operator (GSO)
- ▶ A GSO \mathbf{S} is a standin for any of the matrix representations of a graph - the adjacency matrix \mathbf{A} , the Laplacian matrix \mathbf{L} , the normalized adjacency matrix $\tilde{\mathbf{A}}$, or the normalized Laplacian matrix $\tilde{\mathbf{L}}$.
- ▶ Most of the results we shall present below will hold for any choice of \mathbf{S} .

- ▶ Next we need the concept of a graph signal x .
- ▶ A graph signal is a vector of size $n \times 1$ where n is the number of nodes in the graph.
- ▶ Each component of the graph signal is associated with a node of the graph.
- ▶ The graph and its signal should be viewed together. Therefore we need to talk of (S, x) to emphasize that the graph is an expression of proximity or similarity between the components of the graph signal x .

- ▶ The product $\mathbf{S}\mathbf{x}$ diffuses the signal \mathbf{x} over the graph by which we mean that each node influences its neighbours through \mathbf{S} .
- ▶ To see this consider the i th component of the product $\mathbf{y} = \mathbf{S}\mathbf{x}$.
- ▶ We have $y_i = \sum w_{ij}x_j$.
- ▶ A strong w_{ij} means that the j th component exerts significant influence on the signal at node i .
- ▶ The product $\mathbf{S}\mathbf{x}$ signifies a local operation where components are mixed with the components of neighboring nodes.

Diffusion sequence

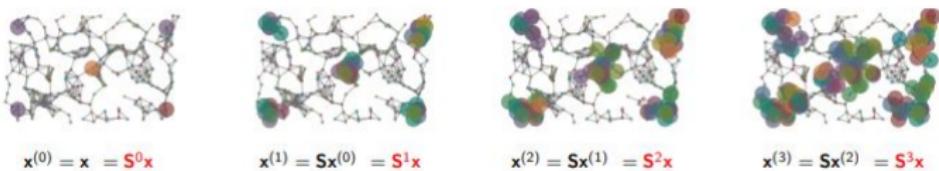


- ▶ We can compose the diffusion operator repeatedly to come up with a diffusion sequence.
- ▶ The diffusion sequence starts with x^0 . Then we have $x^1 = Sx^0$, and so on.
- ▶ The sequence can be expressed recursively as $x^{k+1} = Sx^k$ with $x^0 = x$.
- ▶ What does this mean physically?

Illustration of diffusion sequence



- ▶ The figure below shows how the signal x spreads through diffusion.



Some observations about the diffusion sequence

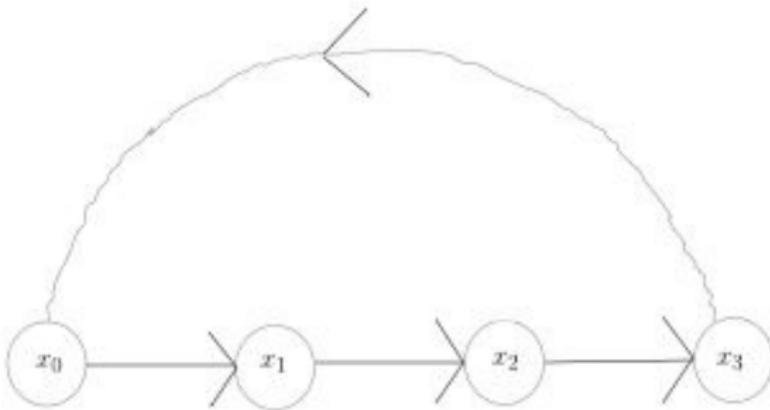


- ▶ The k th element of the diffusion sequence \mathbf{x}^k diffuses information to k -hop neighbourhoods.
- ▶ We will use the diffusion sequence to define graph convolutions.
- ▶ From an analysis point of view we can represent $\mathbf{x}^k = \mathbf{S}^k \mathbf{x}_0$. Writing it in this fashion allows us to see that the evolution of the diffusion sequence is connected to the properties of the matrix \mathbf{S} , specifically its eigenvalues.

- ▶ Given a graph shift operator \mathbf{S} and coefficients h_k , a graph filter is a polynomial series on \mathbf{S} .
- ▶ We can write $H(\mathbf{S}) = \sum_{k=0}^{\infty} h_k \mathbf{S}^k$.
- ▶ The result of applying $H(\mathbf{S})$ on the signal \mathbf{x} is $H(\mathbf{S})\mathbf{x} = \sum_{k=0}^{k=\infty} h_k \mathbf{S}^k \mathbf{x}$.
- ▶ We say $\mathbf{y} = \mathbf{h} *_{\mathbf{S}} \mathbf{x}$ is the graph convolution of the filter $\mathbf{h} = \{h_k\}_{k=0}^{k=\infty}$ with the signal \mathbf{x} .
- ▶ As we consider powers of \mathbf{S} from small values of k to large values, we use information from the local neighbourhood around each node to the global neighbourhood to influence the signal at the node. The magnitude of the coefficients h_k control the amount of information flow from the different neighbourhoods.

- ▶ Let x represent a periodic time signal with period M , i.e $x_{i+M} = x_i = x_{i-M}$ for any value of i . Therefore the periodic signal may be viewed as having M distinct values $x_0, x_1, x_2 \dots x_{M-1}$ which repeat ad-nauseam both backwards and forwards in time.
- ▶ We would like to perform the convolution
$$y_n = \sum_{k=0}^{k=M-1} h_k x_{n-k}.$$
- ▶ The question is the following: can we frame this time convolution computation as a graph convolution with a suitable matrix S ?
- ▶ The answer to the question above is yes, and we shall set about constructing the matrix S in the subsequent slides.

We create a matrix \mathbf{S} that is the adjacency matrix for the graph below representing the cyclic shift operation that shifts the signal to the right modulo the period M .



The adjacency matrix S can be expressed as

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

Applying S on the vector x we see that it accomplishes a right shift on the signal x .

$$S \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_3 \\ x_0 \\ x_1 \\ x_2 \end{bmatrix}$$

- We can see that S^2x will shift the signal x two places to the right, ie.

$$S^2 \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = S \begin{bmatrix} x_3 \\ x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ x_3 \\ x_0 \\ x_1 \end{bmatrix}$$

- We now have the tools we need to express the original time-convolution in terms of a graph-based convolution. Note that we can write

$$S^k \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_{0-k} \\ x_{1-k} \\ x_{2-k} \\ x_{3-k} \end{bmatrix}$$

- ▶ Recall that time convolution is expressed as

$y_n = \sum_{k=0}^{M-1} h_k x_{n-k}$. So each filter coefficient operates on a component of the original signal x right-shifted by an amount k .

- ▶ If we write the output of the convolution as a vector by varying n , we see that the time convolution expression can be obtained using the following vector computation for different components of the output vector y , shown here for the period $M = 4$.

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = h_0 \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} + h_1 \begin{bmatrix} x_3 \\ x_0 \\ x_1 \\ x_2 \end{bmatrix} + h_2 \begin{bmatrix} x_2 \\ x_3 \\ x_0 \\ x_1 \end{bmatrix} + h_3 \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_0 \end{bmatrix}$$

- ▶ The last equation in the previous slide can be reexpressed as

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = h_0 \begin{bmatrix} x_{0-0} \\ x_{1-0} \\ x_{2-0} \\ x_{3-0} \end{bmatrix} + h_1 \begin{bmatrix} x_{0-1} \\ x_{1-1} \\ x_{2-1} \\ x_{3-1} \end{bmatrix} + h_2 \begin{bmatrix} x_{0-2} \\ x_{1-2} \\ x_{2-2} \\ x_{3-2} \end{bmatrix} + h_3 \begin{bmatrix} x_{0-3} \\ x_{1-3} \\ x_{2-3} \\ x_{3-3} \end{bmatrix}$$

- ▶ As previously mentioned we can write

$$\mathbf{S}^k \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} x_{0-k} \\ x_{1-k} \\ x_{2-k} \\ x_{3-k} \end{bmatrix}$$

- ▶ Substituting $\mathbf{S}^k \mathbf{x}$ in place of each shifted \mathbf{x} we see that
 $y = \sum_{k=0}^{M-1} h_k \mathbf{S}^k \mathbf{x}.$

- ▶ We can let \mathbf{S} be the graph shift operator for an arbitrary graph. This will give us graph convolution.
- ▶ Next we shall study the Graph Fourier Transform which is a tool for analyzing graph information processing systems.
- ▶ We shall use tools from linear algebra quite heavily, in particular the diagonalization of symmetric matrices.

- ▶ Our graph shift operator will be taken to be symmetric, i.e $\mathbf{S} = \mathbf{S}^H$.
- ▶ Let us denote the eigenvectors of \mathbf{S} as \mathbf{v}_i and the eigenvalues of \mathbf{S} as λ_i , so that $\mathbf{S}\mathbf{v}_i = \lambda_i\mathbf{v}_i$.
- ▶ By the spectral theorem, a symmetric matrix has real eigenvalues.
- ▶ Let us order these eigenvalues as follows: $\lambda_0 \leq \lambda_1 \leq \dots \leq \lambda_n$.
- ▶ If we define the eigenvector matrix as $\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \dots, \mathbf{v}_n]$, i.e the eigenvectors of \mathbf{S} arranged as columns of the matrix \mathbf{V} , we observe that diagonalization of \mathbf{S} gives us $\mathbf{S} = \mathbf{V}\Lambda\mathbf{V}^H$.

- ▶ Given a graph shift operator \mathbf{S} which can be diagonalized as $\mathbf{S} = \mathbf{V}\Lambda\mathbf{V}^H$, we can define the Graph Fourier transform (GFT) as $\tilde{\mathbf{x}} = \mathbf{V}^H\mathbf{x}$.
- ▶ Note that the matrix \mathbf{V}^H contains the eigenvectors of \mathbf{S} as its rows. Therefore what we are doing here is to take the inner product of \mathbf{x} with each eigenvector to obtain a component of $\tilde{\mathbf{x}}$. This is essentially projecting \mathbf{x} onto the space of the eigenvectors of \mathbf{S} .
- ▶ Consider the analogy from the Fourier series representation of a periodic function: $x(t) = \sum_{k=-\infty}^{k=\infty} a_k e^{jk\omega_0 t}$ where ω_0 is the fundamental frequency such that $\omega_0 = \frac{2\pi}{T}$.
- ▶ The coefficient $a_k = \frac{1}{T} \int_0^T x(t) e^{-jk\omega_0 t} dt$.

- ▶ The preceding computation for a_k , the coefficient of the k th term of the Fourier series can be viewed as an inner product of the "vector" $x(t)$ with the vector $e^{-jk\omega_0 t}$. We are doing the same kind of computation here in our context with respect to the given vector \mathbf{x} and each eigenvector \mathbf{v}_i to compute the relevant Fourier coefficient which is a component of $\tilde{\mathbf{x}}$.
- ▶ We can compute the inverse graph Fourier transform (iGFT) as follows: $\tilde{\mathbf{x}} = \mathbf{V}\tilde{\mathbf{x}}$
- ▶ Since \mathbf{V} is a unitary matrix which means $\mathbf{V}^H \mathbf{V} = \mathbf{I}$ we can see that the inverse Fourier transform of the Fourier transform of a signal \mathbf{x} recovers \mathbf{x} . We have
$$\tilde{\mathbf{x}} = \mathbf{V}\tilde{\mathbf{x}} = \mathbf{V}\mathbf{V}^H\mathbf{x} = \mathbf{I}\mathbf{x} = \mathbf{x}.$$

Theorem Consider a graph filter \mathbf{h} with coefficients h_k , a graph signal \mathbf{x} and the filtered signal $\mathbf{y} = \sum_{k=0}^{k=\infty} h_k \mathbf{S}^k \mathbf{x}$. The GFT $\tilde{\mathbf{x}} = \mathbf{V}^H \mathbf{x}$ and $\tilde{\mathbf{y}} = \mathbf{V}^H \mathbf{y}$ are related by $\tilde{\mathbf{y}} = \sum_{k=0}^{k=\infty} h_k \mathbf{\Lambda}^k \tilde{\mathbf{x}}$

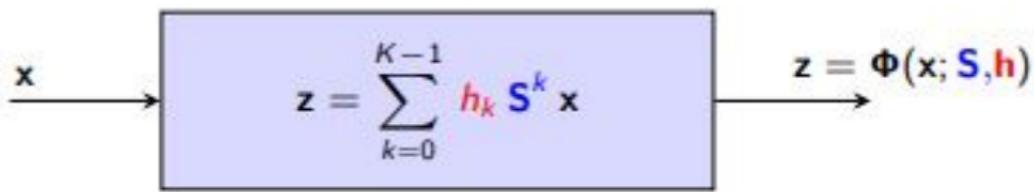
- ▶ Before we delve into the proof of the theorem, let us observe its significance: we are using the same filter coefficients in a polynomial to relate \tilde{y} and \tilde{x} as we did to relate y and x .
- ▶ However the polynomial now is in terms of Λ whereas in the previous case it was in terms of S .
- ▶ Now let us look at the proof of this theorem ...

- ▶ Since $\mathbf{S} = \mathbf{V}\Lambda\mathbf{V}^H$, we can write $\mathbf{S}^k = \mathbf{V}\Lambda^k\mathbf{V}^H$.
- ▶ The filter output \mathbf{y} can then be written as
$$\mathbf{y} = \sum_{k=0}^{\infty} h_k \mathbf{S}^k \mathbf{x} = \sum_{k=0}^{\infty} h_k \mathbf{V}\Lambda^k\mathbf{V}^H \mathbf{x}.$$
- ▶ Multiply both sides of the above equation by \mathbf{V}^H . Then we get $\mathbf{V}^H \mathbf{y} = \mathbf{V}^H \sum_{k=0}^{\infty} h_k \mathbf{V}\Lambda^k\mathbf{V}^H \mathbf{x}$.
- ▶ Now we identify $\mathbf{V}^H \mathbf{y} = \tilde{\mathbf{y}}$, $\mathbf{V}^H \mathbf{V} = \mathbf{I}$, and $\mathbf{V}^H \mathbf{x} = \tilde{\mathbf{x}}$ to get the result we want: $\tilde{\mathbf{y}} = \sum_{k=0}^{\infty} h_k \Lambda^k \tilde{\mathbf{x}}$

- ▶ In the graph frequency domain, the graph filter becomes a diagonal matrix: $\tilde{\mathbf{y}} = \sum_{k=0}^{k=\infty} h_k \Lambda^k \tilde{\mathbf{x}}$.
- ▶ Thus the graph convolutions are point-wise in the GFT domain. This implies $\tilde{y}_i = \sum_{k=0}^{k=\infty} h_k \lambda_i^k \tilde{x}_i$.
- ▶ Thus we have $\tilde{y}_i = \tilde{h}(\lambda_i) \tilde{x}_i$.
- ▶ The above gives us the following definition for a graph filter: given a graph filter with coefficients $\mathbf{h} = \{h_k\}_{k=1}^{k=\infty}$, the graph frequency response is the polynomial $\tilde{h}(\lambda) = \sum_{k=0}^{k=\infty} h_k \lambda^k$
- ▶ The frequency response is the same polynomial that defines the graph filter, but the polynomial is defined over a scalar variable rather than a matrix.

- ▶ The graph frequency response is independent of the graph. It depends only on the filter coefficients.
- ▶ The graph enters the picture only to determine the eigenvalues on which the response is instantiated.
- ▶ Eigenvectors do not enter the frequency response. They determine the meaning of frequencies.

- ▶ The graph convolutional filter can be represented as follows:



- ▶ Optimization is over the filter coefficients that minimize some loss function.

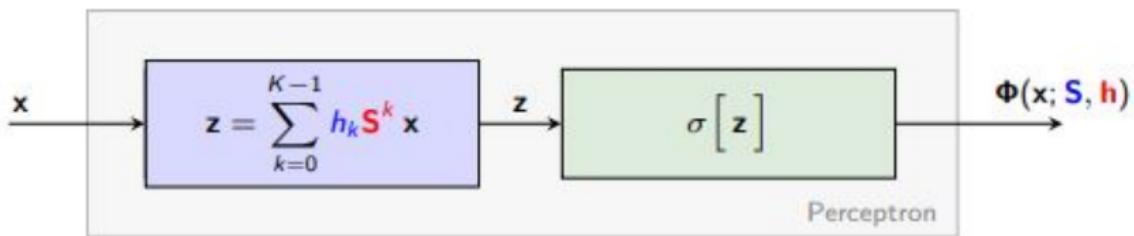
$$\mathbf{h}^* = \operatorname{argmin}_{\mathbf{h}} \sum_{(x,y) \in T} l(y, \phi(x; \mathbf{S}, \mathbf{h}))$$

- ▶ To build a Graph Neural Network we need the concept of pointwise nonlinearities.
- ▶ A pointwise nonlinearity applied to a vector is a nonlinearity applied component-wise to every component of the vector.
- ▶ We can denote a pointwise function σ and its action on a

vector x as follows: $\sigma[x] = \sigma \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_n) \end{bmatrix}$.

- ▶ A pointwise nonlinearity is the simplest nonlinear function that we can apply to a vector. Examples of pointwise nonlinearities include $ReLU : \sigma(x) = \max(0, x)$, hyperbolic tangent $\sigma(x) = \frac{e^{2x}-1}{e^{2x}+1}$ and absolute value $\sigma(x) = |x|$.

- ▶ Graph filters have limited expressive power since they can only express linear maps.
- ▶ Pointwise nonlinearities increase the expressive power of graph filters, and enable graph perceptrons (shown below) to learn nonlinear maps.



- ▶ To define a GNN we compose several graph perceptrons, layer by layer.
- ▶ Layer 1 processes the input signal x with the perceptron $\mathbf{h} = [h_{10}, h_{11}, \dots, h_{1,K-1}]$ to produce the output x_1 according to the following equation:

$$x_1 = \sigma[z_1] = \sigma\left[\sum_{k=0}^{K-1} h_{1k} S^k x\right]$$

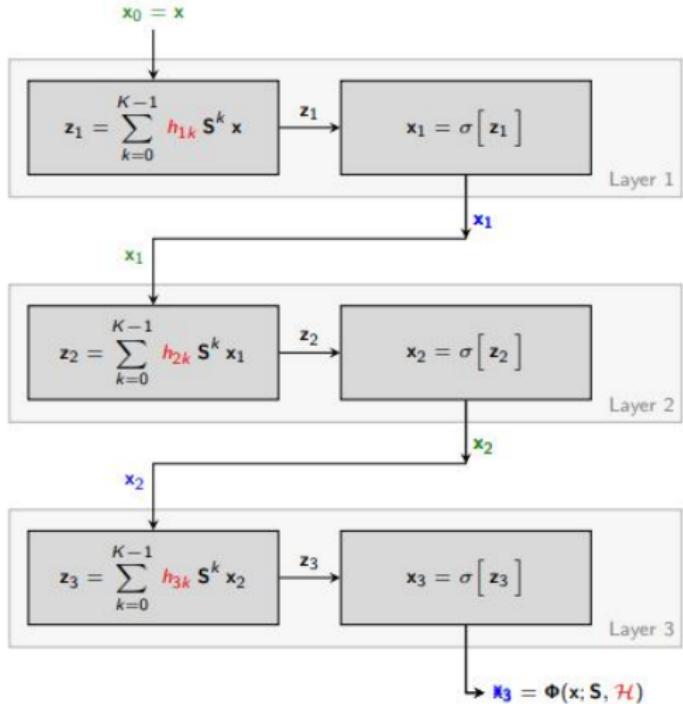
- ▶ The output of the layer x_1 is used as the input to the next layer, and so on.

- ▶ A generic layer l of the GNN takes as input the output of the previous layer \mathbf{x}_{l-1} and produces output \mathbf{x}_l using the following equation:

$$\mathbf{x}_l = \sigma[\mathbf{z}_l] = \sigma\left[\sum_{k=0}^{K-1} h_{lk} \mathbf{S}^k \mathbf{x}_{l-1}\right]$$

- ▶ With the convention that the layer 1 input \mathbf{x}_0 is the original graph signal \mathbf{x} , this gives a recursive definition of a graph neural network.
- ▶ if the GNN has L layers, the GNN output $\mathbf{x}_L = \phi(\mathbf{x}; \mathbf{S}, \mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_L) = \phi(\mathbf{x}; \mathbf{S}, \mathbf{H})$.

GNN block diagram





Lecture 10

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- ▶ In the last lecture, we looked at theoretical motivations for Graph Neural Networks arising out of the signal processing operation of convolution.
- ▶ We showed how time convolution can be implemented in terms of a graph shift operator.
- ▶ We then generalized the graph shift operator to represent an arbitrary adjacency matrix, or Laplacian of a graph.
- ▶ We now look at some other theoretical motivations for Graph Neural Networks arising out of their connection to probabilistic graphical models.

- ▶ We shall establish a connection between GNNS and the problem of variational inference in PGMs.
 - ▶ In the probabilistic perspective, we view the embeddings z_u for each node u as latent variables to be inferred.
- graph* ▶ We assume that we are given the adjacency matrix of the graph A , the input node features X .
- ▶ Our goal then is to infer the latent variables, i.e the embeddings z_u that can explain the observed data.

- ▶ We shall show that the message passing algorithm underpinning graph neural networks is the same kind of algorithm that is used for variational inference to infer distributions over latent variables.
- ▶ This connection was first noted by Dai et al in their paper "Discriminative embeddings of latent variable models for structured data", published in ICML, 2016.

- ▶ To understand the link between GNNs and probabilistic graphical models we need to undertake a brief detour into Hilbert spaces.
- ▶ We are used to vector spaces of finite dimension.
- ▶ A Hilbert space essentially is a vector-space of infinite dimension with an inner product associated with it.
- ▶ This part of the lecture is drawn from notes from CMU:
[https://www.cs.cmu.edu/~epxing/Class/10708-14/
scribe_notes/scribe_note_lecture22.pdf](https://www.cs.cmu.edu/~epxing/Class/10708-14/scribe_notes/scribe_note_lecture22.pdf)

- ▶ A Hilbert space is an extension to a vector-space where the vectors are infinite-dimensional and should be treated as functions.
- ▶ The inner product in a Hilbert space could be defined as follows: $\langle f, g \rangle = \int_a^b f(x)g(x)dx$. However, there is no requirement that the inner product be defined as an integral.
- ▶ The inner product defined on a Hilbert space needs to follow certain basic properties.

- ▶ Symmetry: $\langle f, g \rangle = \langle g, f \rangle$.
- ▶ Linearity: $\langle \alpha f_1 + \beta f_2, g \rangle = \alpha \langle f_1, g \rangle + \beta \langle f_2, g \rangle$.
- ▶ Non-negativity: $\langle f, f \rangle \geq 0$
- ▶ Zero: $\langle f, f \rangle = 0 \implies f = 0$.

Reproducing Kernel Hilbert Space (RKHS)



- ▶ A reproducing kernel Hilbert space is a Hilbert space where each point in the space is a continuous linear function.
- ▶ An RKHS is constructed on the basis of a Mercer kernel.
- ▶ A Mercer kernel is a function of two variables $K(x, y)$ such that $\int \int K(x, y)f(x)f(y)dxdy > 0, \forall f$.
- ▶ The most common kernel that we use is a Gaussian kernel.

Reproducing Kernel Hilbert Space



- ▶ The Gaussian RBF Kernel can be expressed as $\exp\left(\frac{\|x-y\|_2^2}{\sigma^2}\right)$.
- ▶ Let us hold one element of the function fixed, and let the other variable vary.
- ▶ The result is a function of one variable which we call a feature function.
- ▶ The feature function has the form $\phi_x = K(x, :)$

$$\begin{aligned}\phi_x(y) &= k(x, y) \\ \phi_x(z) &= k(x, z)\end{aligned}$$

- ▶ For a Gaussian kernel the feature functions become un-normalized Gaussians.
- ▶ We see that $\phi_1(y) = \exp\left(\frac{\|y - y\|_2^2}{\sigma^2}\right)$.
- ▶ We can define an inner product on this space as follows:
 $\langle \phi_x, \phi_y \rangle = \langle K(x, :), K(y, :) \rangle = K(x, y)$.
- ▶ Intuitively this quantity is a scalar because it is the dot product of two vectors. It can be seen to be symmetric since K is a symmetric function.

- ▶ Consider the space of all functions that are a linear combination of the feature functions, i.e
 $F_0 = \{f(z) : \sum_{j=1}^{j=K} \alpha_j \phi_{x_j}(z) \forall k \in N_+, \text{ and } x_j \in X\}.$
- ▶ We then define the Reproducing Kernel Hilbert Space (RKHS) F to be the completion of the set F_0 .
- ▶ We can now define the Reproducing property that gives RKHS its name.

- ▶ We shall show that $f(x)$ can be computed by the inner product $\langle f, \phi_x \rangle$.
- ▶ To see this, we first note that f can be rewritten as
$$f = \sum_j \alpha_j \phi_{x_j}.$$
- ▶ Because of the linearity of the inner product we can write
$$\langle f, \phi_x \rangle = \langle \sum_j \alpha_j \phi_{x_j}, \phi_x \rangle = \sum_j \alpha_j \langle \phi_{x_j}, \phi_x \rangle.$$
- ▶ Each inner product term can be rewritten in terms of the kernel function: $\langle \phi_{x_j}, \phi_x \rangle = K(x_j, x)$, so that we have
$$\langle f, \phi_x \rangle = \sum_j \alpha_j K(x_j, x) = f(x)$$
 as needed.

- ▶ We shall look at the theory of embedding distributions of one variable.
- ▶ The basic equation can be written as follows:

$$\mu_X(\cdot) = E_{X \sim D}[\phi_X] = \int p_D(X)\phi_X(\cdot)dX$$

- ▶ This statistic is computed over the feature function mappings of the distribution into the RKHS.
- ▶ If the distribution over the x is discrete, then this statistic can be computed as follows: $\hat{\mu}_X = \frac{1}{N} \sum_{i=1}^{i=N} \phi_{x_i}$ where the x_i are sampled from the distribution p_X

- ▶ As a simple example consider the *RKHS* embedding for a discrete distribution that takes values from 1 to 4.
- ▶ The moments of this distribution can be computed from the data but it will lead to a loss of information as we lose the distinction between values contained in the discrete distribution.
- ▶ The computed statistic is also just a number.
- ▶ Let us now consider an *RKHS* mapping of the distribution into a vector in R^4 .

- ▶ The feature functions become

$$\phi_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, \phi_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, \phi_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, \phi_4 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

- ▶ The calculation for the mean becomes a summation since the distribution is discrete - hence we have $\mu_X = E_X[\phi_X] = P(X = 1)\phi_1 + P(X = 2)\phi_2 + P(X = 3)\phi_3 + P(X = 4)\phi_4$.

- ▶ In vector form we have $\mu_X = \begin{bmatrix} P(X = 1) \\ P(X = 2) \\ P(X = 3) \\ P(X = 4) \end{bmatrix}$.
- ▶ Thus the embedding of the distribution in the discrete case is a vector carrying the marginal probabilities.

- ▶ We need to connect the preceding theory to graphical neural networks.
- ▶ The bridge that will enable us to do this is to view a graphical neural network as a probabilistic graphical model.
- ▶ Taking a probabilistic view of graph data, we assume that the graph structure given to us (in the form of the matrix S) represents the dependencies between nodes.
- ▶ This informal relationship between nodes is already assumed to exist in graph neural networks.
- ▶ We shall model the relationship using a probability theory treatment.

- ▶ Formally we say that a graph $G = (V, E)$ denotes a Markov random field.
- ▶ We can write the joint distribution of node features $\{x_v\}$ and node embeddings $\{z_v\}$ as follows:

$$P(\{x_v\}, \{z_v\}) \propto \prod_{v \in V} \phi(x_v, z_v) \prod_{(u,v) \in E} \psi(z_u, z_v)$$

- ▶ Here ϕ and ψ are non-negative potential functions, and the set $\{x_v\}$ is a shorthand for $\{x_v, v \in V\}$.

- ▶ The equation given on the previous slide says that the distribution $p(\{x_v\}, \{z_v\})$ over node features and node embeddings factorizes according to the graph structure.
- ▶ Intuitively $\phi(x_v, z_v)$ indicates the likelihood of a node feature vector x_v given its latent node embedding z_v .
- ▶ The potential function ψ controls the dependence between connected nodes.
- ▶ Thus we assume that node features are determined by their latent embeddings, and that latent embeddings for connected nodes are related to each other.
- ▶ In a probabilistic setting ϕ and ψ are usually defined as parametric functions based on domain knowledge and are usually assumed to come from the exponential family for tractability.

- ▶ We seek to implicitly learn the functions ϕ and ψ by leveraging the Hilbert space embedding idea that we have already presented. The exact form of ϕ and ψ will not be relevant to us.
- ▶ Given the Markov random field given by the equation presented a few slides ago, our goal is to infer the distribution of latent embeddings $p(z_v)$ for all nodes $v \in V$ while also implicitly learning the functions ϕ and ψ .
- ▶ We would like to infer the latent representations for all nodes in the graph that can explain the observed dependencies between node features.
- ▶ A key step in this regard is to learn the posterior $p(\{z_v\} | \{x_v\})$, i.e computing the likelihood of a particular set of latent representations given the observed features.

- ▶ Computing $p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})$ is computationally intractable, so we need to resort to using approximate methods. This is the case even if ϕ and ψ are known and well-defined.
- ▶ We shall exploit an idea called mean-field variational inference where we approximate the posterior using some functions q_v based on the assumption given below:

$$p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\}) \approx q(\{\mathbf{z}_v\}) = \prod_{v \in V} q_v(\mathbf{z}_v)$$

- ▶ Here each q_v is a valid density.

- ▶ The key intuition in mean-field inference is that we assume that the posterior distribution of latent embeddings given node features factorizes into V independent distributions, one per node.
- ▶ To obtain the approximating functions q_v that are optimal in the mean-field approximation, the standard approach is to minimize the KL-divergence between the approximate posterior and the true posterior using the following equation

$$KL(q(\{\mathbf{z}_v\})|p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})) \\ = \int_{(R^d)^V} \prod_{v \in V} q(\mathbf{z}_v) \log \left(\frac{\prod_{v \in V} q(\mathbf{z}_v)}{p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})} \right) \prod_{v \in V} d\mathbf{z}_v$$

- ▶ The KL -divergence is a canonical way of measuring the distance between two probability distributions.
- ▶ We have set up the integral so as to minimize the distance between the approximate posterior and the true posterior under the mean-field assumption
- ▶ There is no way to actually minimize the KL -divergence integral since we do not know the true posterior.
- ▶ We can use techniques from variational inference to solve this problem.

- ▶ It can be shown that any solution for $q_v(\mathbf{z}_v)$ that minimizes the KL -divergence must satisfy the following fixed-point equation:

$$\log(q_v(\mathbf{z}_v)) = c_v + \log(\phi(\mathbf{x}_v, \mathbf{z}_v)) + \sum_{u \in N(v)} \int_{R^d} q_u(\mathbf{z}_u) \log(\psi(\mathbf{z}_u, \mathbf{z}_v)) d\mathbf{z}_u$$

- ▶ Here c_v is a constant that does not depend on $q_v(\mathbf{z}_v)$ or \mathbf{z}_v .

- ▶ We will not prove that any solution that minimizes the KL -divergence must satisfy the fixed-point equation given in the previous slide.
- ▶ Solving the fixed-point equation can be done by starting with an initial guess q_v^0 and then computing

$$\log(q_v^t(\mathbf{z}_v)) = c_v + \log(\phi(\mathbf{x}_v, \mathbf{z}_v)) + \sum_{u \in N(v)} \int_{R^d} q_u^{t-1}(\mathbf{z}_u) \log(\psi(\mathbf{z}_u, \mathbf{z}_v)) d\mathbf{z}_u$$

- ▶ Our purpose is to use the fixed-point equation to establish the link to graph neural networks.

- ▶ Looking at the fixed-point equation we see that the updated marginal distribution q_v^t is a function of the node features x_v through the potential function ϕ as well as a function of set of neighbour marginals $q_u^{t-1}(z_u) \forall u \in N(v)$ from the previous iteration through the function ψ
- ▶ This form of message passing is analogous to what happens in GNNs.
- ▶ We update the values at each node based on the values of nodes in the given node's neighbourhood.
- ▶ The key difference between GNNs and the mean-field message passing is that here we operate over distributions rather than embeddings as in the case of GNNs.

- ▶ We can make the connection between GNNs and mean-field inference even tighter by leveraging Hilbert-space embeddings.
- ▶ Suppose we have some injective feature map ϕ and can represent all the marginals $q_v(\mathbf{z}_v)$ as embeddings

$$\mu_v = \int_{R^d} q_v(\mathbf{z}_v) \phi(\mathbf{z}_v) d\mathbf{z}_v \in R^d$$

- ▶ We can now rewrite the fixed-point equations in terms of embeddings as follows:

$$\mu_v^t = \mathbf{c} + f(\mu_v^{t-1}, \mathbf{x}_v, \{\mu_u, \forall u \in N(v)\})$$



Lecture 11

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

- ▶ We have studied a variety of methods for learning representations of graphs.
- ▶ In this lecture we turn our attention to the question of graph generation.
- ▶ The goal of graph generation is to build models that can generate realistic graph structures.
- ▶ The graph generation problem can be viewed as a mirror image of the graph embedding problem in which we treat the graph itself as an input to the problem.
- ▶ In graph generation we want the graph to be the output of our model.

Overview of traditional approaches



- ▶ We shall begin with traditional approaches to this problem.
- ▶ Traditionally graph generation involves specifying some kind of generative process involving the creation of graph edges.
- ▶ Usually we frame this generative process as a way of specifying the probability $P(A[u, v] = 1)$ of the existence of an edge between vertices u and v in a graph.
- ▶ This simplistic model has some problems in that it might fail to generate other characteristics that the generated graphs must have, such as a certain community structure.
- ▶ This makes graph generation more non-trivial than it might initially appear to be.

- ▶ In this model we define the likelihood of an edge between any pair of nodes u , and v in a graph G as $P(A[u, v] = 1) = r$.
- ▶ The parameter $r \in [0, 1]$ controls the density of the graph.
- ▶ The *ER* model is attractive due to its simplicity.
- ▶ Generating a random *ER*-graph simply requires choosing or sampling the number of nodes we want, set the density parameter r , and then use the probability equation given above to generate the adjacency matrix.
- ▶ Since all the probabilities are independent, the time taken to generate a graph in this model is $O(V^2)$ where V is the number of vertices in the graph.

A closer look at the ER model



- ▶ There are two variants of the *ER* model - the $G(n, M)$ model and the $G(n, p)$ model.
- ▶ In the $G(n, M)$ model a graph is chosen uniformly at random from the set of all graphs with n vertices and M edges.
- ▶ For example, if we consider $G(3, 2)$, there 3 graphs which have 3 vertices and 2 edges. Each of the graphs looks like a line with two segments in it. The graphs are only distinguished by the identity of the middle vertex.
- ▶ Note that if we choose a particular label for the middle vertex, we do not get a different graph by permuting the labels of the end vertices. In the $G(3, 2)$ case, each of the given 3 graphs is selected with probability $1/3$.

- ▶ The other variant of the *ER* model is the $G(n, p)$ model which is identical to the one we initially presented.
- ▶ Here each edge in a graph of n nodes is selected with a probability p independently of any other edge.
- ▶ The probability of generating an arbitrary graph on n nodes with M edges is $p^M(1 - p)^{\binom{n}{2} - M}$
- ▶ Note that when $p = \frac{1}{2}$, all $2^{\binom{n}{2}}$ graphs on n vertices are chosen with the same probability.

A closer look at the ER model



- ▶ When $n \rightarrow \infty$, we have the result that nearly all graphs in $G(n, \frac{2 \ln n}{n})$ are connected.
- ▶ The expected number of edges in $G(n, p)$ is $\binom{n}{2}p$.
- ▶ By the law of large numbers, as $n \rightarrow \infty$, almost all graphs in $G(n, p)$ will almost surely have approximately $\binom{n}{2}$ edges, provided the expected number of edges goes to infinity.
- ▶ The distribution of the degree of vertices is binomial:

$$P(\deg(v) = k) = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

- ▶ Here n is the total number of vertices in the graph. The distribution becomes Poisson for large n , and $np = \text{const.}$

Properties of $G(n,p)$



- ▶ A 1960 paper of Erdős and Rényi showed many interesting results for graphs in $G(n, p)$.
- ▶ If $np < 1$, then a graph in $G(n, p)$ will almost surely have no connected components of size larger than $O(\log n)$
- ▶ If $np = 1$, then a graph in $G(n, p)$ will almost surely have a component of size $n^{\frac{2}{3}}$.
- ▶ If $np \rightarrow c > 1$, then graph in $G(n, p)$ will almost surely have a unique giant component. No other component will contain more than $O(\log n)$ vertices.

Properties of $G(n,p)$



- ▶ If $p < \frac{(1-\epsilon) \ln n}{n}$ then a graph in $G(n, p)$ will almost surely contain isolated vertices and hence be disconnected.
- ▶ If $p > \frac{(1+\epsilon) \ln n}{n}$, then a graph in $G(n, p)$ will almost surely be connected.
- ▶ Thus $\frac{\ln n}{n}$ is a sharp threshold for the connectedness of $G(n, p)$.

- ▶ We shall follow the reference
https://www.ndsu.edu/pubweb/~novozhil/Teaching/767%20Data/chapter_3.pdf for some of the results described here.
- ▶ Let us look at Theorem 3.3 from the above source.
- ▶ Theorem 3.3: Let $\alpha : N \rightarrow R$ be a function such that $\alpha(n) \rightarrow 0$ as $n \rightarrow \infty$. Let $p(n) = \frac{\alpha(n)}{n}$ for each $n \in N$. Then $T_{3,n} = 0$ with high probability.

- ▶ $T_{3,n}$ is a random variable on the space $G(n, p)$ which is equal to the number of triangles in a random graph.
- ▶ For example $T_{3,n}(K_n) = \binom{n}{3}$
- ▶ For any graph G with only two edges, $T_{3,n}(G) = 0$.
- ▶ To prove Theorem 3.3, we shall first need Theorem 3.2 which we state as follows: Let X_n be an non-negative integer valued random variable. IF $E(X_n) \rightarrow 0$, then $X_n = 0$ with high probability as $n \rightarrow \infty$.

Some background



- ▶ To prove Theorem 3.2, we need Markov's inequality which states the following $P(X \geq a) \leq \frac{E(X)}{a}$.
- ▶ Markov's inequality can be shown quite easily:
 $E(X) = \int_{-\infty}^{\infty} xf(x)dx = \int_0^{\infty} xf(x)dx$ since x is positive-valued. Then, we have $E(X) = \int_0^{\infty} xf(x)dx \geq \int_a^{\infty} xf(x)dx \geq \int_a^{\infty} af(x)dx \geq aP(X \geq a)$.
- ▶ Setting $a = 1$ in Markov's inequality gives us $P(X_n \geq 1) \leq E(X_n)$ which proves Theorem 3.2.

Theorem 3.3



- ▶ Now we can prove Theorem 3.3. The goal is to show that $P(T_{3,n} = 0) \rightarrow 1$ as $n \rightarrow \infty$.
- ▶ This is the same as saying $P(T_{3,n} \geq 1) \rightarrow 0$.
- ▶ By Markov's inequality $P(T_{3,n} \geq 1) \leq E(T_{3,n})$.
- ▶ We therefore need to estimate $E(T_{3,n})$. Let us do so in the following slide.

Theorem 3.3



- ▶ We can write $T_{3,n} = \mathbf{1}_{\tau_1} + \mathbf{1}_{\tau_2} + \dots + \mathbf{1}_{\tau_k}$ where $k = \binom{n}{3}$.
- ▶ What we are doing above is that we are using an indicator variable for each of the $\binom{n}{3}$ possible triples of vertices, and τ_i is the i th triple of vertices. The indicator variable $\mathbf{1}_{\tau_i}$ is 1 or 0 in a particular graph instance drawn from $G(n, p)$ depending on whether the particular triple τ_i is a complete triangle or not in a random graph instance.
- ▶ Using the linearity of expectation we have
$$E(T_{3,n}) = E(\mathbf{1}_{\tau_1}) + E(\mathbf{1}_{\tau_2}) + \dots + E(\mathbf{1}_{\tau_k}),$$

Theorem 3.3



- ▶ The last step consists of noting that the expectation of each of the indicator variables constitutes a probability, i.e $E(\mathbf{1}_{\tau_i}) = P(\tau_i)$
- ▶ The probability $P(\tau_i)$ is the probability that in a random graph the particular triple represented by τ_i is a complete triangle. This probability is p^3 .
- ▶ Then we see that $E(T_{3,n}) = \binom{n}{3} p^3$.
- ▶ Finally we see that $E(T_{3,n}) = \binom{n}{3} p^3 = \frac{n!}{(n-3)!3!} \frac{\alpha^3(n)}{n^3} = \frac{n(n-1)(n-2)\alpha^3(n)}{6n^3} \approx \frac{\alpha^3(n)}{6} \rightarrow 0$.

- ▶ Let us look at a result on the connectivity of *ER* random graphs.
- ▶ We shall state and prove Theorem 3.10 from the notes on the *ER* model mentioned previously.
- ▶ Theorem 3.10: Let $p = \frac{c \log n}{n}$. If $c \geq 3$ and $n \geq 100$ then $P(\{G(n, p) \text{ is connected}\}) \rightarrow 1$.
- ▶ Let us look at the intuition behind this result.

Theorem 3.10



- ▶ Consider a random variable X on $G(n, p)$ such that $X(G) = 0$ if G is connected, and $X(G) = k$ if G has k components.
- ▶ Note that $X(G)$ is never equal to 1.
- ▶ We need to show that $P(X = 0) \rightarrow 1$. This is the same as showing that $P(X \geq 1) \rightarrow 0$.
- ▶ By Markov's inequality $P(X \geq 1) \leq E(X)$

Theorem 3.10



- ▶ We can represent X in the following way:
$$X = X_1 + X_2 + \dots + X_{n-1}$$
, where X_j is the number of components that have exactly j vertices. Note that we do not have X_n in the sum since when we have a component with n vertices, the graph is connected.
- ▶ We order all j element subsets of the set of vertices and label them from 1 to $\binom{n}{j}$.
- ▶ Consider the event K_i^j to be one where the i th j -element subset forms a component in G .
- ▶ We can write
$$X_j = \sum_{i=1}^{i=\binom{n}{j}} \mathbf{1}_{K_i^j}$$

Theorem 3.10



- ▶ $E(X) = \sum_{j=1}^{n-1} \sum_{i=1}^{\binom{n}{j}} \mathbf{1}_{K_i^j} = \sum_{j=1}^{n-1} \sum_{i=1}^{\binom{n}{j}} P(K_i^j)$
- ▶ Now we can see that for the i th j -element subset to form a component, at the very least it must be the case that there are no edges from any of the vertices in K_i^j to vertices outside K_i^j .
- ▶ We say "at the very least" because even if it turns out that there are no edges between K_i^j and $V \setminus K_i^j$ it does not mean that i th j -element subset forms a component.

Theorem 3.10



- We have

$P(K_i^j) \leq P(\text{ there are no edges connecting vertices in}$

$$K_i^j \text{ and those in } V \setminus K_i^j) = (1 - p)^{j(n-j)}$$

- Thus we have

$$E(X) \leq \sum_{j=1}^{n-1} \sum_{i=1}^{i=\binom{n}{j}} (1 - p)^{j(n-j)} = \sum_{j=1}^{n-1} \binom{n}{j} (1 - p)^{j(n-j)}$$

- This last sum is symmetric in the sense that the terms with j and $n - j$ are the same. Taking $j = 1$, we see that the first term in the summation is $n(1 - p)^{n-1}$.

$$n(1 - p)^{n-1} \leq n e^{-p(n-1)} \leq e^{-\frac{3(n-1) \log n}{n}}$$

Theorem 3.10



- ▶ Take $\frac{n-1}{n} > 0.9$. This is true for the value of n assumed in the statement of the theorem.
- ▶ We can see that $n(1-p)^{n-1} \leq ne^{-p(n-1)} \leq e^{-2.7\log n}$ where in the last inequality we substitute $p = \frac{3\log n}{n}$ and use the fact that $n = e^{\log n}$.
- ▶ $e^{-2.7\log n} = \frac{1}{n^{2.7}}$. Thus the first term is no greater than $\frac{1}{n^{2.7}}$.
- ▶ Consider the quotient between two successive terms in the series. We must consider

$$\frac{\binom{n}{j+1}(1-p)^{(j+1)(n-j-1)}}{\binom{n}{j}(1-p)^{j(n-j)}} = \frac{n-j}{j+1}(1-p)^{n-2j-1}$$

Theorem 3.10



- ▶ To round out the argument, we consider two conditions for j :
 $j \leq \frac{n}{8}$ and $j > \frac{n}{8}$.
- ▶ If $j \leq \frac{n}{8}$ we can show that $\frac{n-j}{j+1}(1-p)^{n-2j-1} \leq \frac{1}{n}$.
- ▶ If $j > \frac{n}{8}$ we can show that each term of the form
 $\binom{n}{j}(1-p)^{j(n-j)} \leq 2^n n^{-\frac{3n}{16}}$ which for large enough n can be shown to be smaller than $\frac{1}{n^{2.7}}$
- ▶ Thus all the binomial terms in the series are smaller than $\frac{1}{n^{2.7}}$ which means $E(X) \leq \sum_{j=1}^{n-1} \frac{1}{n^{2.7}} < \frac{n}{n^{2.7}} \rightarrow 0$.

- ▶ A downside of the *ER* model is that it does not generate realistic graphs.
- ▶ The parameter r or p controls only the average degree in the graph.
- ▶ Other properties such as degree distribution, existence of community structures, node clustering coefficients etc are not captured by the *ER* model.
- ▶ We can use stochastic block models to generate graphs with community structure.

- ▶ In a basic *SBM* model we first specify a number of blocks $C_1, C_2, \dots, C_\gamma$.
- ▶ Every node $u \in V$ has a probability p_i of belonging to block C_i . We can write $p_i = P(u \in C_i), \forall u \in V, i = 1, \dots, \gamma$,
 $\sum_{i=1}^{i=\gamma} p_i = 1$.
- ▶ Edge probabilities are specified in a block-to-block probability matrix $C \in [0, 1]^{\gamma \times \gamma}$ where $C[i, j]$ specifies the probability of an edge between a node in block C_i and node in block C_j .

Generative process for the SBM model



- ▶ The generative process for the basic *SBM* model is as follows:
 - ▶ For every vertex $u \in V$ we assign u to a block C_i by sampling from a categorical distribution defined by $(p_1, p_2, \dots, p_\gamma)$.
 - ▶ For every pair of nodes $u \in C_i$ and $v \in C_j$ we sample an edge according to $P(A[u, v] = 1) = C[i, j]$.
- ▶ The key innovation in *SBM* is that we can control the edge probabilities within and between different blocks.
- ▶ This allows us to generate graphs with community structure.

- ▶ The matrix C of size $\gamma \times \gamma$ plays a key role in the SBM generative process.
- ▶ Normally we set the diagonal elements of this matrix to equal to some constant α .
- ▶ We can then set the off-diagonal entries to a value $\beta < \alpha$.
- ▶ The semantics of this is that nodes have a probability α of having an edge with another node of the same community and a probability $\beta < \alpha$ of having an edge with another node that is assigned to a different community.
- ▶ Can we envision setting entries in C so as to make the *SBM* model an *ER* model?

- ▶ If we let $\beta = \alpha$ in the matrix C , then *SBM* becomes the *ER* model as the probability that there is an edge between a pair of vertices remains the same no matter the identity of the pair of vertices.
- ▶ If all diagonal elements of the matrix C are equal to p and all off-diagonal elements are equal to q such that $p > q$, the model is called assortative.
- ▶ If $p < q$, the model is called disassortative.

Typical statistical tasks



- ▶ Much of the literature on algorithmic community detection addresses three statistical tasks: detection, partial recovery and exact recovery.
- ▶ The goal of detection algorithms is to determine, given a sampled graph, whether it has a latent community structure.
- ▶ A graph might be generated with some known prior probability from a known stochastic block model or from the *ER* model.
- ▶ The algorithmic detection task is to identify which of the two models generated the graph.

Typical statistical tasks



- ▶ In partial recovery the goal is to approximately determine the latent partition into communities in the sense of finding a partition that is correlated with the true partition better than a random guess.
- ▶ In exact recovery the goal is to recover the latent partition exactly even if the community sizes and probability matrix may be unknown.

- ▶ Suppose we allow the size of the graph to grow by letting the number of vertices n increase.
- ▶ We still keep the community sizes in the same proportions.
- ▶ If the probability matrix remains fixed, tasks such as partial and exact recovery become feasible for all non-degenerate parameter settings.
- ▶ However if we scale down the probability matrix at a suitable rate as n increases, we observe a sharp phase transition: for certain settings of the parameters we can achieve recovery with probability tending to 1. For settings of the parameters on the other side the threshold, recovery probability tends to 0 no matter what algorithm is used.

- ▶ Consider the issue of threshold behaviour on the problem of partial recovery.
- ▶ We take $C_{ij} = \frac{\tilde{C}_{ij}}{n}$, where \tilde{C}_{ij} is a fixed constant.
- ▶ This gives us graphs of constant average degree.
- ▶ Consider the case of two equal-sized communities, in the assortative planted partition matrix with probability matrix

$$C = \begin{bmatrix} \frac{\tilde{p}}{n} & \frac{\tilde{q}}{n} \\ \frac{\tilde{p}}{n} & \frac{\tilde{q}}{n} \end{bmatrix}$$

- ▶ In the case of two equal-sized communities introduced in the last slide, partial recovery is possible with probability $1 - o(1)$ whenever $(\tilde{p} - \tilde{q})^2 > 2(\tilde{p} + \tilde{q})$.
- ▶ On the other hand, any estimator fails partial recovery with probability $1 - o(1)$ whenever $(\tilde{p} - \tilde{q})^2 < 2(\tilde{p} + \tilde{q})$.
- ▶ For exact recovery the appropriate scaling is to take $C_{ij} = \frac{\tilde{C}_{ij} \log n}{n}$ resulting in logarithmic average degree.
- ▶ For exact recovery, using log-scaling, we find that the assortative planted partition model with r equal-sized communities, the threshold lies at $\sqrt{\tilde{p}} - \sqrt{\tilde{q}} = \sqrt{r}$.



Lecture 12

GNN Team



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Motivation for preferential attachment



- ▶ The *SBM* (stochastic block model) framework does generate graphs with community structures.
- ▶ However the SBM model still falls short in that it is unable to capture real-world characteristics of some graphs, in that some individual nodes have very high degree.
- ▶ In an *SBM* model all nodes have the same degree distribution.
- ▶ This means that in an *SBM* model, the nodes have similar structural properties including degree and clustering coefficients.

Motivation for preferential attachment



- ▶ In real-world graphs we see much heterogeneous and varied degree-distributions.
- ▶ It is not uncommon to see many low-degree nodes and a few very high-degree nodes which serve the role of hubs.
- ▶ This kind of degree distribution cannot be produced by *SBM*.
- ▶ Preferential attachment is a model that attempts to capture this specific property of real-world graphs.

- ▶ Many real world graphs exhibit power-law degree distributions.
- ▶ The probability of a node u having degree d_u is roughly given by the following equation: $P(d_u = k) \propto k^{-\alpha}$ where $\alpha > 1$ is a parameter.
- ▶ Power-law distributions have the property that they are heavy-tailed.
- ▶ A heavy-tailed distribution means that the probability values go to zero for extreme values slower than an exponential distribution.
- ▶ This means that heavy-tailed distributions assign a non-trivial probability mass to events that are essentially impossible under a standard exponential distribution.

- ▶ The heavy-tailed nature means that there is a non-zero probability of encountering high-degree nodes in a graph.
- ▶ This allows power-law distributions to capture a crucial aspect of real-world graphs which is that there are a small number of nodes with very high degree.
- ▶ Real-world graphs also have a large number of nodes with small degrees.
- ▶ A simple generative model can be used to generate graphs following the preferential attachment model.
- ▶ We shall detail steps of the model and provide a simple analysis to explain why the degrees of the nodes follow a power-law distribution.

- ▶ First we initialise a fully connected graph wth m_0 nodes.
- ▶ Next we add iteratively $n - m_0$ nodes to this graph. For each new node that we add at iteration t , we connect it to $m < m_0$ existing nodes in the graph. We choose the m neighbours by sampling without replacement according to the following distribution:

$$P(A[u, v] = 1) = \frac{d_v^{(t)}}{\sum_{v' \in V^{(t)}} d_{v'}^{(t)}}$$

- ▶ Here $d_v^{(t)}$ denotes the degree of node v at iteration t and $V^{(t)}$ denotes the set of nodes that have been added up to iteration t .

- ▶ The key idea is that the generative model connects new nodes to existing nodes with a probability that is proportional to the degrees of the existing nodes.
- ▶ High degree nodes will therefore accumulate more and more neighbours in a rich get richer phenomenon as the graph grows.
- ▶ We will show that the degrees of the nodes follow a power-law distribution.
- ▶ The generation process is autoregressive unlike the *SBM* and *ER* models. Instead of specifying edge probabilities for the entire graph in one step, the *PA* model relies on an iterative approach, where the edge probabilities at step t depend on the edges added at step $t - 1$.

- ▶ We shall refer to the work of Albert and Barabasi in <https://arxiv.org/abs/cond-mat/0106096> for the analysis below that shows that the degree distribution in the *PA* model follows a power-law.
- ▶ The technique that we shall use to derive the power-law degree distribution is called the Continuum Theory in the work of Albert and Barabasi.
- ▶ There are other approaches to this problem which yield similar results which we shall not cover such as the master equation approach and the rate equation approach.

- ▶ In continuum theory we calculate the time dependence of the degree k_i of a given node i .
- ▶ This degree will increase every time a new node comes into the system and links to node i . The probability of this happening is $\Pi(k_i)$ where $\Pi(k_i) = \frac{k_i}{\sum_j k_j}$.
- ▶ The preferential attachment algorithm considered by Albert and Barabasi starts with a complete graph on m_0 nodes at time $t = 0$. At time t we end up having $m_0 + t$ nodes and mt edges. Note that each time we add a new node to the system, the new node will add m edges to the system. The m edges will sit between the new nodes and randomly selected nodes among those already added to the system.

- ▶ We shall assume that k_i is a continuous variable and the rate at which k_i changes is given by $\frac{\partial k_i}{\partial t} = m\Pi(k_i) = m \frac{k_i}{\sum_{j=1}^{N-1} k_j}$.
- ▶ The sum in the denominator goes over all nodes in the system except the newly added one. Thus we have $\sum_{j=1}^{N-1} k_j = 2mt - m$.
- ▶ Thus we have $\frac{\partial k_i}{\partial t} = \frac{k_i}{2t}$.
- ▶ We need to find a solution to this equation to obtain the dependence between k_i the degree of node i and time t .

- ▶ The solution of the equation given in the preceding slide with the initial condition that every node at its introduction has $k_i(t_i) = m$ is $k_i(t) = m \left(\frac{t}{t_i} \right)^{\beta}$ where $\beta = \frac{1}{2}$.
- ▶ This solution indicates that the degrees of all nodes evolve in the same way following a power-law with the only difference being the intercept of the power law.
- ▶ We can now calculate the probability that node i has degree $k_i(t)$ smaller than k as follows: $P(k_i(t) < k) = P(t_i > \frac{m^{1/\beta} t}{k^{1/\beta}})$.

- ▶ Let us assume that the $m_0 + t$ nodes were all added at equal time intervals, and node i is added at time t_i which is uniformly distributed in the range $[0, m_0 + t]$. Thus the probability density function of t_i can be written as $P(t_i) = \frac{1}{m_0+t}$. Note that node i is not necessarily the i th node added in sequence, it can be any one of the nodes added. We just take the time at which it was added to be t_i .
- ▶ We can calculate $P(t_i < \frac{m^{1/\beta} t}{k^{1/\beta}}) = \frac{m^{1/\beta} t}{k^{1/\beta}} \frac{1}{m_0+t}$.
- ▶ Thus $P(t_i > \frac{m^{1/\beta} t}{k^{1/\beta}}) = 1 - \frac{m^{1/\beta} t}{k^{1/\beta}} \frac{1}{m_0+t}$.

- ▶ Thus we can see that
$$P(k_i(t) \leq k) = P(t_i > \frac{m^{1/\beta} t}{k^{1/\beta}}) = 1 - \frac{m^{1/\beta} t}{k^{1/\beta}} \frac{1}{m_0 + t}.$$
- ▶ The degree distribution of the i th node can be obtained from the cumulative distribution function by differentiating it, so we have $P(k) = \frac{\partial P(k_i(t) < k)}{\partial k} = \frac{1/\beta m^{1/\beta} t}{m_0 + t} \frac{1}{k^{1/\beta+1}}.$
- ▶ As $t \rightarrow \infty$, we find $P(k) \rightarrow \frac{1}{\beta} m^{1/\beta} \frac{1}{k^{1/\beta+1}}.$
- ▶ Setting $\beta = 1/2$, we can see that $P(k) \rightarrow 2m^2 k^{-\gamma}$, where $\gamma = 3$.

- ▶ We have over the course of these two lectures reviewed the *ER*, *SBM* and *PA* models.
- ▶ Let us now examine some concrete applications of these models.
- ▶ The first useful application of these models is that they can be used to generate synthetic graphs for benchmarking and analysis tasks.
- ▶ Suppose we have developed a community detection algorithm - how do we test it?
- ▶ We can generate a graph consisting of communities using the *SBM* model and test out whether our community detection algorithm is able to detect the communities present in the graph.

Applications of generative models



- ▶ Suppose we have generated a network analysis engine that is supposed to scale to very large graphs.
- ▶ We can test the framework on synthetic graphs generated by the *PA* model to see if our model can handle heavy-tailed degree distributions.
- ▶ The two examples presented show that in certain specific applications we need to generate graphs to test out specific algorithms.
- ▶ The second key application for generative models is the generation of null models.

Applications of generative models



- ▶ Suppose we have a social network dataset in which we are computing different graph statistics such as degree distributions and clustering coefficients.
- ▶ We might want to ask how typical is the graph on which we are computing these statistics.
- ▶ Generative graph models give us a precise way of answering such questions.
- ▶ We can investigate the extent to which different graph statistics are possible under different generative models.
- ▶ For example if our sample graph shows heavy-tailed degree distributions, we might check this statistic against graphs generated using a preferential attachment model



Graph Neural Networks (S*-**_AIMLCZG514)

BITS Pilani
Pilani Campus



BITS Pilani
Pilani Campus



Module IX



Deep Generative Models

Quick Recap of Graph Generation

❖ Traditional Graph Generation Approaches

❖ *Erdös–Rényi (ER) model*

❖ $P(A[u, v] = 1) = r, \forall u, v \in V, u \neq v.$

❖ Here $r \in [0, 1]$, Controlling density of the graph.

❖ *G(N,r) model*

❖ N is the number of node.

❖ r is probability of an edge between any pair of nodes.

❖ *G(n,m) model*

❖ N is the number of node

❖ m is the number of edges to be presented in a graph.

Quick Recap of Graph Generation



- ❖ *Erdös–Rényi (ER) model*
 - ❖ **Simple** to implement.
 - ❖ Time complexity to generate a graph is $O(V^2)$
 - ❖ **Connectedness of the graph need to be checked and ensured by the programmer.**
 - ❖ Does **not** generate very realistic graphs
 - ❖ Does **not** Capture degree distribution, existence of community structures, node clustering coefficients, and the occurrence of structural motifs
-

Quick Recap of Graph Generation

❖ Stochastic Block Models (SBMs)

- ❖ Generate graphs with community structure.
- ❖ The number of blocks/communities required is specified, say γ , $(C_1, C_2, \dots, C_\gamma)$
- ❖ *Every node $u \in V$ has a probability p_i of belonging to block/community i , i.e.:*

$$p_i = P(u \in C_i), \forall u \in V, (i = (1, \dots, \gamma)), s.t \sum_{i=1}^{\gamma} P_i = 1.$$

- ❖ *Edge probabilities are then specified by a block-to-block probability matrix $P \in [0,1]^{\gamma \times \gamma}$, where $C[i, j]$, gives the probability of an edge occurring between a node in block C_i and a node in block C_j*

Quick Recap of Graph Generation



❖ Stochastic Block Models (SBMs)

- ❖ The generative process for the basic SBM model is as follows:
 - ❖ For every node $u \in V$, we assign u to a block C_i by sampling from the categorical distribution defined by $(p_1, p_2, \dots, p_\gamma)$
 - ❖ For every pair of nodes $u \in C_i$ and $v \in C_j$ we sample an edge according to $P(A[u, v] = 1) = C[i, j]$.
- ❖ Here we control the edge probabilities (constants) within (α) and between different blocks (β), and to generate graphs that exhibit community structure.

$$\begin{bmatrix} \alpha & \beta & \beta \\ \beta & \alpha & \beta \\ \beta & \beta & \alpha \end{bmatrix}, \text{and } \alpha > \beta$$

Quick Recap of Graph Generation

❖ Stochastic Block Models (SBMs)

- ❖ Like the simple ER model, the SBM approach is also **limited**, and **fails to capture the structural characteristics** of individual nodes that are present in most real-world graphs.
- ❖ All nodes within a community have the **same degree distribution**.
- ❖ This means that the **structure** of individual communities is relatively **homogeneous**, in that, all the nodes, have similar structural properties (e.g., *similar degrees and clustering coefficients*).
- ❖ Unfortunately, however, this homogeneity is quite unrealistic in the real world.
- ❖ In **real-world graphs** we often see much more **heterogeneous** and **varied degree distributions**,

Quick Recap of Graph Generation

- ❖ Preferential Attachment models (**PA**) [*incremental generation*]
 - ❖ They are built around the assumption that many real-world graphs exhibit **power law degree distributions**,
 - ❖ i.e. the probability of a node u having degree d_u is roughly given by the following equation:
$$P(d_u = k) \propto k^{-\alpha}, \text{here } \alpha > 1.$$
 - ❖ Power law distributions have the property that they are **heavy tailed**.
 - ❖ i.e there is a non-zero chance of encountering a small number of very high-degree nodes, which also implies, that a large number of nodes have small degrees.

Quick Recap of Graph Generation

- ❖ Preferential Attachment models (**PA**)
- ❖ First, we initialize a **fully connected** graph with m_0 nodes
- ❖ Next, we iteratively add $n - m_0$ nodes to this graph.
- ❖ For each new node u that we add at iteration t ,
 - ❖ we connect it to $m < m_0$ existing nodes in the graph, and we choose its m neighbors by sampling **without** replacement according to the following probability distribution: $P(A[u, v]) = \frac{d_v^{(t)}}{\sum_{v' \in V^{(t)}} d_{v'}^{(t)}}$
 - ❖ here $d_v^{(t)}$ is degree of node v at iteration t and $V^{(t)}$ is the set of nodes that have been added to the graph upto the iteration t .



Deep Generative Models

Deep Generative Models

- ❖ The traditional graph generation approaches can be used to efficiently generate synthetic graphs that have certain properties, and they can be used to give us insight into how certain graph structures might arise in the real world.
- ❖ However, a key limitation of those traditional approaches is that they rely on a ***fixed, hand-crafted*** generation process.
 - ❖ In short, the traditional approaches can generate graphs, but they lack the ability to ***learn a generative model*** from data.

Deep Generative Models

- ❖ Deep Generative approaches will seek to **learn** a generative model of graphs based on a set of **training graphs**,
 - ❖ which avoid hand-coding particular properties, such as community structure or degree distributions, into a generative model.
- ❖ Here, we wish to design models that can **observe** a set of graphs $\{G_1; \dots G_n\}$ and **learn** to generate graphs with similar characteristics as this training set.



Deep Generative Models

- ❖ These models will adapt three of the most popular approaches to building general deep generative models:
- ❖ **Variational autoencoders (VAEs),**
 - ❖ We seek to generate an entire graph ***all-at-once*** in an **autoencoder** style.

Deep Generative Models

- ❖ These models will adapt three of the most popular approaches to building general deep generative models:
- ❖ **Generative Adversarial Networks (GANs),**
 - ❖ GAN-based objectives can be used in place of Variational losses, but still in the setting where the graphs are generated ***all-at-once***

These all-at-once generative models are analogous to the ER and SBM generative models, in that we sample all edges in the graph simultaneously.

Deep Generative Models

- ❖ These models will adapt three of the most popular approaches to building general deep generative models:
- ❖ **Autoregressive models.**
 - ❖ Generate a graph ***incrementally*** instead of all-at-once (e.g., generating a graph node-by-node).
 - ❖ It is similar to the preferential attachment model (**PA**), in that the probability of adding an edge at each step during generation depends on what edges were previously added to the graph.



Deep Generative Models

-
- ❖ For simplicity, all the methods we discuss will only focus on generating graph structures (i.e., adjacency matrices) and **not on generating node or edge features.**

Keywords & Definitions

❖ Latent Variable

- ❖ In statistics, **latent variables** (from Latin word “*lateo*” means “lie hidden”) are variables that can only be inferred indirectly through a mathematical model from other **observable variables** that can be directly observed or measured (*i.e. IQ level, Customer Satisfaction index, Employee Dedication, Quality of Life, Happiness level etc.*).

In this situation, the term hidden variables is commonly used (*reflecting the fact that the variables are meaningful, but not observable*).

Keywords & Definitions

❖ Latent Space

- ❖ In the context of machine learning, latent space refers to a ***lower-dimensional space*** where the essential ***features of the original high-dimensional data are preserved.***
- ❖ The term “***latent***” implies that the space ***captures*** the underlying structure or ***hidden relationships*** within the data.
- ❖ It is particularly useful in ***unsupervised learning*** techniques, such as dimensionality reduction, clustering, and generative modeling.



Keywords & Definitions

❖ Latent Space

- ❖ Latent spaces are often used to reduce the complexity of data, making it easier to work with and understand.
- ❖ These methods aim to find a lower-dimensional representation of the data that retains as much of the original information as possible.
- ❖ There are several methods for constructing latent spaces, one of them is **autoencoder**

Keywords & Definitions

❖ Latent Space (Challenges)

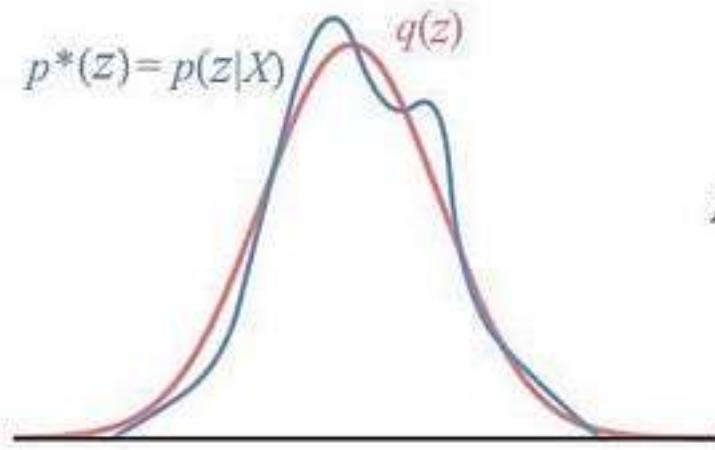
- ❖ **Interpretability:** The transformation from the original data space to the latent space can be complex and difficult to interpret. This can make it challenging to understand the relationships between the original features and the latent space representation.
- ❖ **Loss of Information:** The compression of the original data into a lower-dimensional space can lead to a loss of important features or relationships in the data, which may impact the performance of downstream learning models.

Despite these challenges, latent spaces remain a powerful tool, for enabling the analysis and manipulation of complex, high-dimensional data in a more efficient and interpretable way.

Keywords & Definitions

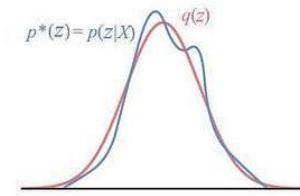
❖ Variational Inferences

- ❖ In **variational inferencing**, given the observation X , we build a probability model q for latent variables z , i.e. $q \approx p(z|X)$.



$$p^*(z) = p(z|X) = \frac{p(X, z)}{p(X)}$$

Keywords & Definitions



$$p^*(z) = p(z|X) = \frac{p(X, z)}{p(X)}$$

❖ Variational Inferences

- ❖ The marginal $p(X)$ above can be computed as

$$p(X) = \int \int \dots \int p(X, z_1, z_2, z_3, \dots) dz_1 dz_2 dz_3 \dots$$

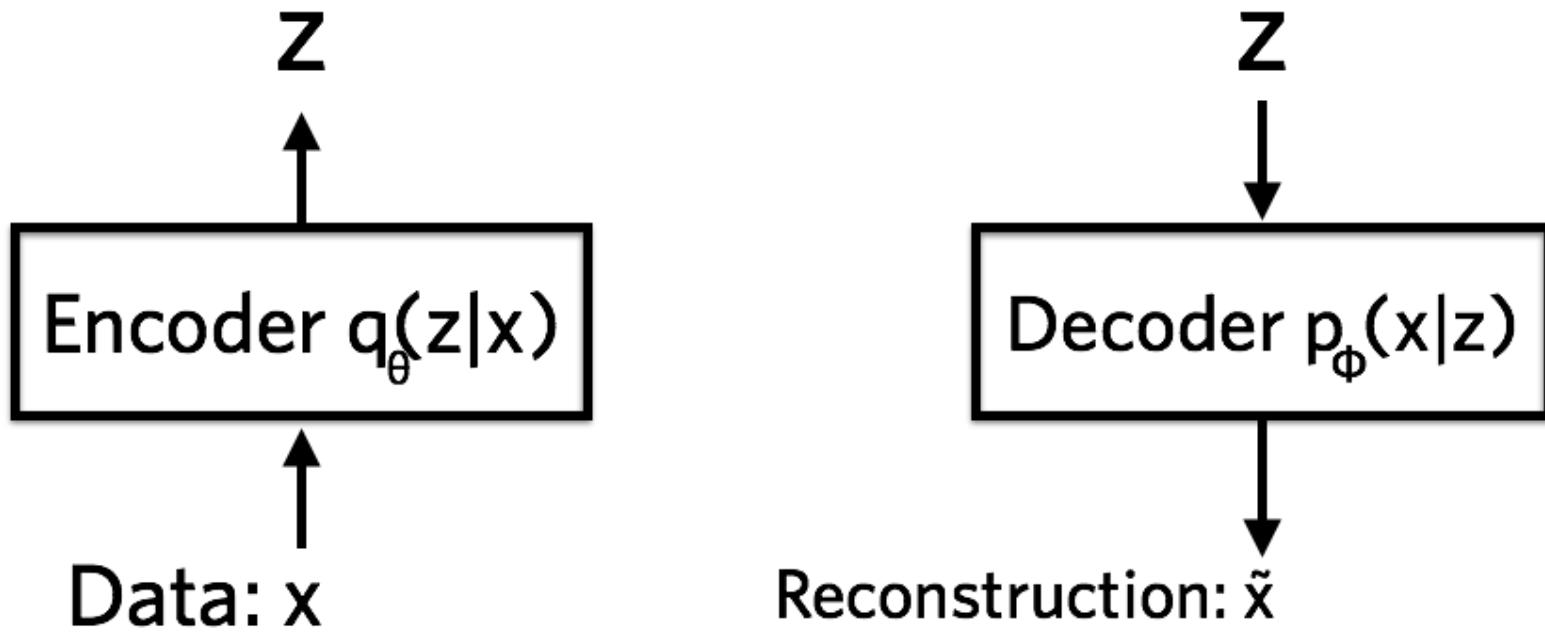
- ❖ In variational inference, ***we avoid computing the marginal $p(X)$.*** This partition function is usually nasty. Instead, we select some tractable families of distribution q to approximate p .

$$p(z_1, z_2, \dots | X) \approx q(z_1, z_2, \dots | \theta)$$

- ❖ We fit q with sample data to ***learn*** the distribution parameters θ .

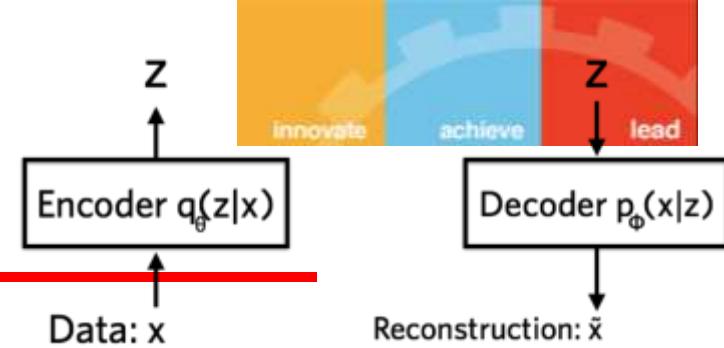
Keywords & Definitions

❖ Variational Autoencoder



- ❖ The Encoder compresses the data into Latent Space (Z).
- ❖ The Decoder reconstructs the data given the hidden representation.

Keywords & Definitions



❖ Variational Autoencoder

- ❖ Variational Autoencoders (VAEs) are a type of generative model that *combines aspects of deep learning and probabilistic modeling to learn compact, structured representations of high-dimensional data.*
- ❖ VAEs consist of an **encoder** network, which maps input data to a latent space, and a **decoder** network, which reconstructs the input data from the latent space representation.
- ❖ VAEs are particularly useful for tasks such as unsupervised learning, data compression, and generative tasks like image synthesis and text generation.



Keywords & Definitions

- ❖ **Variational Autoencoders** perform the following tasks:
 - ❖ **Representation learning:** VAEs learn low-dimensional structured representations, of high-dimensional data, allowing for efficient storage and manipulation of complex data.
 - ❖ **Probabilistic modeling:** VAEs model the underlying probability distribution of the data, which enables them to generate new samples that are similar to the training data.
 - ❖ **Reconstruction:** VAEs can reconstruct input data from the learned latent space representation, which can be useful for tasks like denoising or data compression.

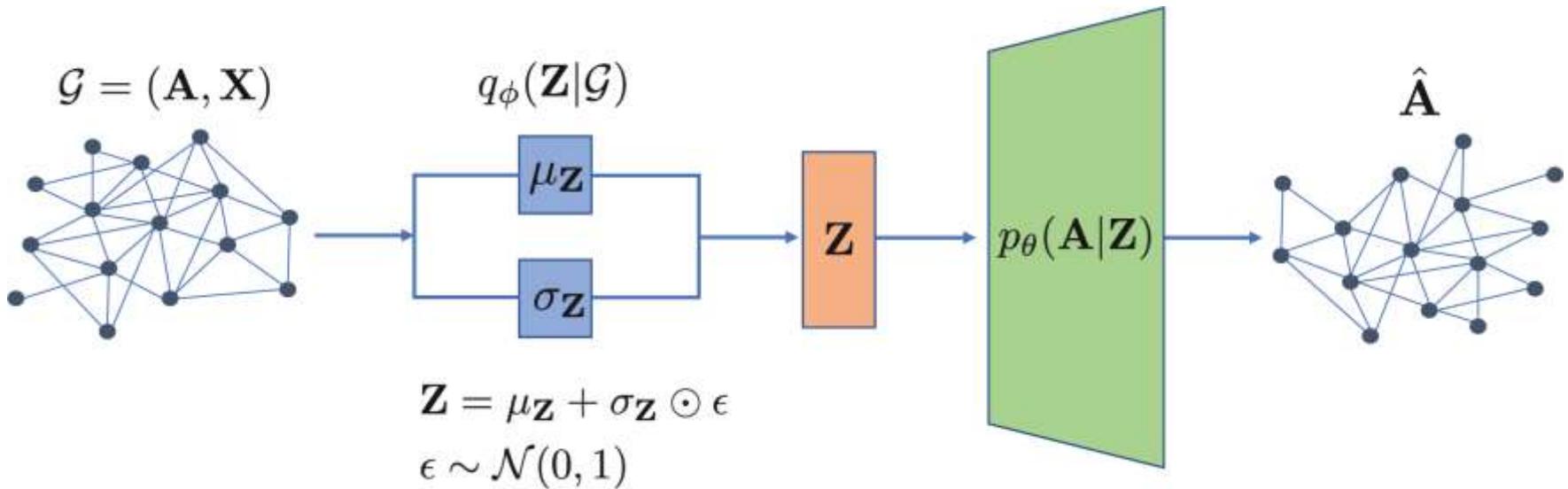
Keywords & Definitions

- ❖ **Variational Autoencoders** offer several **benefits** for machine learning tasks:
 - ❖ **Unsupervised learning:** VAEs can learn useful representations of data without the need for labeled data, making them suitable for unsupervised learning tasks.
 - ❖ **Generative capabilities:** VAEs can generate new data samples that are similar to the training data, which is useful for tasks like data augmentation, image synthesis, and text generation.
 - ❖ **Robustness:** VAEs can be more robust to overfitting and noise than other deep learning models, due to their probabilistic nature and the use of regularization during training.

VARIATIONAL AUTOENCODER APPROACHES (Graph)

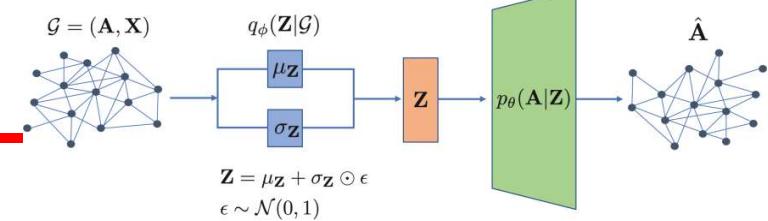


- ❖ Illustration of a standard VAE model applied to the graph setting



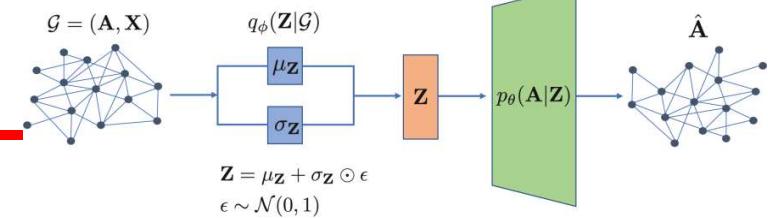
- ❖ An **encoder** neural network maps the input graph $G = (\mathbf{A}, \mathbf{X})$ to a *posterior distribution* $q_\phi(\mathbf{Z}|\mathcal{G})$ over latent variables Z .
- ❖ Given a sample from this posterior, the **decoder** model $p_\theta(\mathbf{A}|Z)$ attempts to reconstruct the adjacency matrix.

VARIATIONAL AUTOENCODER APPROACHES (Graph)



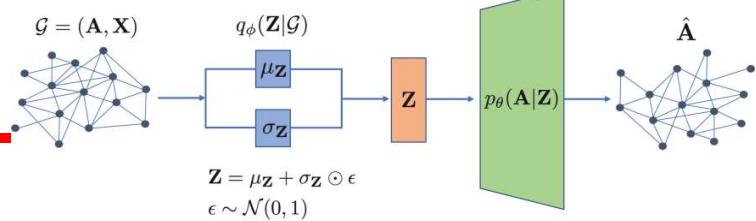
- ❖ our goal is to train a ***probabilistic decoder*** model $p_\theta(\mathbf{A}|\mathbf{Z})$, from which we can sample realistic graphs (*i.e.*, *adjacency matrices*) $\hat{\mathbf{A}} \sim p_\theta(\mathbf{A}|\mathbf{Z})$ by conditioning on a latent variable \mathbf{Z} .
- ❖ In a probabilistic sense, we aim to learn a conditional distribution over adjacency matrices (*with the distribution begin conditioned on some latent variable*).
- ❖ In order to train a VAE, we combine the probabilistic decoder with a ***probabilistic encoder*** model $q_\phi(\mathbf{Z}|\mathcal{G})$.
- ❖ This encoder model maps an input graph \mathcal{G} to a posterior distribution over the latent variable \mathbf{Z} .

VARIATIONAL AUTOENCODER APPROACHES (Graph)



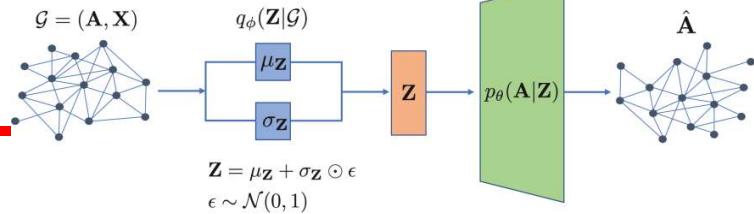
- ❖ The idea is that we jointly train the encoder and decoder so that the decoder is able to reconstruct training graphs given a latent variable $\mathbf{Z} \sim q_\theta(\mathbf{Z}|\mathcal{G})$ sampled from the encoder.
- ❖ Then, after training, we can discard the encoder and generate new graphs by sampling latent variables $\mathbf{Z} \sim p(\mathbf{Z})$ from some (unconditional) prior distribution and feeding these sampled latents to the decoder.

VARIATIONAL AUTOENCODER APPROACHES (Graph)



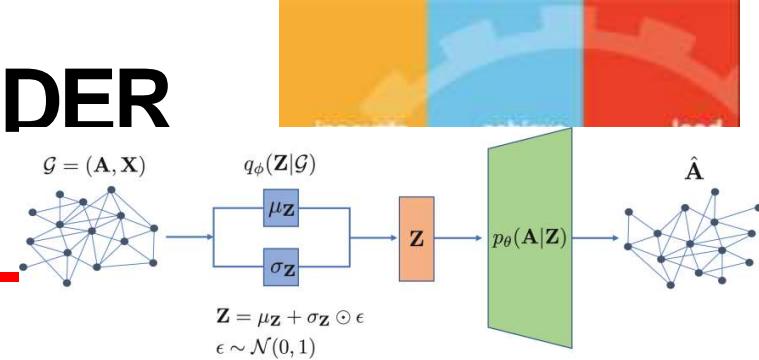
- ❖ In more formal and mathematical detail, to build a VAE for graphs we must specify the following key components
- ❖ A *probabilistic encoder* model q_ϕ :
- ❖ A *probabilistic decoder* model p_θ .
- ❖ A *prior distribution* $p(Z)$ over the latent space.

VARIATIONAL AUTOENCODER APPROACHES (Graph)



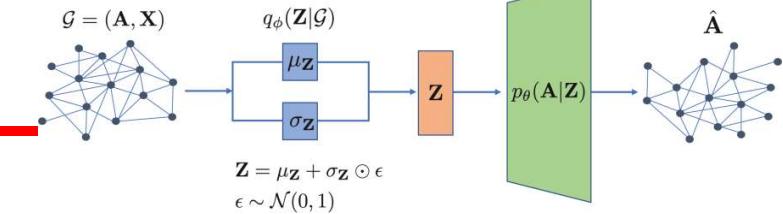
- ❖ A *probabilistic encoder* model q_ϕ :
 - ❖ It takes a graph G as input. From this input q_ϕ , then defines a distribution $q_\phi(\mathbf{Z}|G)$ over *latent representations*.
 - ❖ Generally, in VAEs the *reparameterization trick* with Gaussian random variables is used to design a probabilistic q_ϕ function, i.e., we specify the latent conditional distribution as $\mathbf{Z} \sim N(\mu_\phi(G), \sigma_\phi(G))$, where μ_ϕ and σ_ϕ are neural networks that generate the mean and variance parameters for a normal distribution, from which we sample latent embeddings Z .

VARIATIONAL AUTOENCODER APPROACHES (Graph)



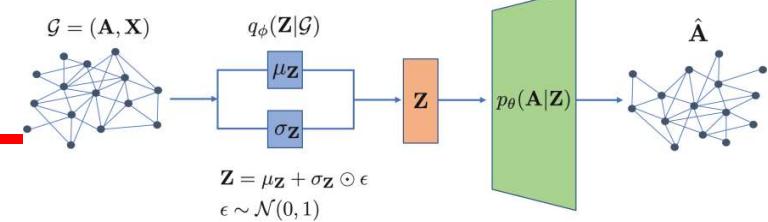
- ❖ A *probabilistic decoder* model p_θ :
 - ❖ The decoder takes a latent representation \mathbf{Z} as input and uses this input to specify a conditional distribution over graphs.
 - ❖ Here, we will assume that p_θ defines a conditional distribution over the entries of the adjacency matrix,
 - ❖ i.e., we can compute $P_\theta(A[u, v] = 1 | \mathbf{Z})$

VARIATIONAL AUTOENCODER APPROACHES (Graph)



- ❖ A *prior distribution* $p(\mathbf{Z})$ over the latent space:
 - ❖ We will adopt the standard Gaussian prior $\mathbf{Z} \sim N(\mathbf{0}, \mathbf{1})$, which is commonly used for VAEs.

VARIATIONAL AUTOENCODER APPROACHES (Graph)

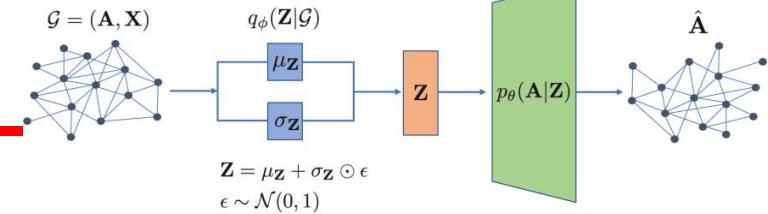


- ❖ Given these components and a set of training graphs (G_1, \dots, G_n), we can train a VAE model by minimizing the evidence likelihood lower bound (**ELBO**):

$$\mathcal{L} = \sum_{G_i \in \{G_1, \dots, G_n\}} \mathbb{E}_{q_\theta(\mathbf{Z}|G_i)} [p_\theta(G_i|\mathbf{Z}) - KL[q_\theta(\mathbf{Z}|G_i) || p(\mathbf{Z})]]$$

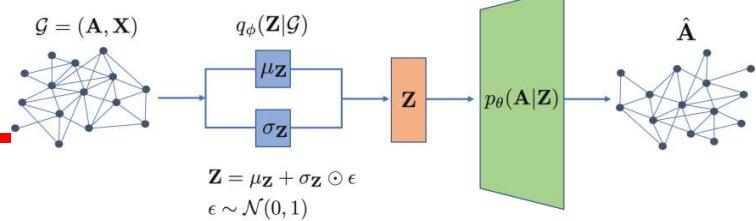
- ❖ The basic idea is that we seek to **maximize** the reconstruction ability of our decoder—i.e., the likelihood term $\mathbb{E}_{q_\theta(\mathbf{Z}|G_i)} [p_\theta(G_i|\mathbf{Z})]$
- ❖ while **minimizing** the KL-divergence between our posterior latent distribution $q_\theta(\mathbf{Z}|G_i)$ and the prior $p(\mathbf{Z})$.

VARIATIONAL AUTOENCODER APPROACHES (Graph)



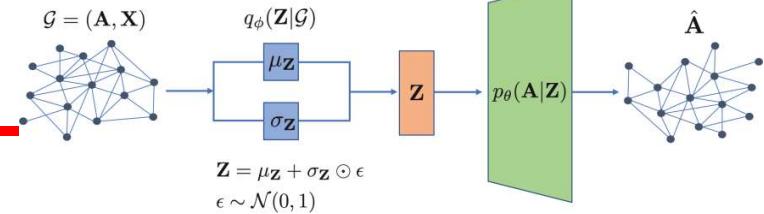
- ❖ The key intuition is that, we want to generate a distribution over latent representations so that the following two (conflicting) goals are satisfied.
 1. The sampled latent representations encode enough information to allow our decoder to reconstruct the input.
 2. The latent distribution is as close as possible to the prior.

VARIATIONAL AUTOENCODER APPROACHES (Graph)



1. The sampled latent representations encode enough information to allow our decoder to reconstruct the input.
 - ❖ Ensures that we learn to decode meaningful graphs from the encoded latent representations, when we have training graphs as input.
2. The latent distribution is as close as possible to the prior.
 - ❖ It acts as a ***regularizer*** and ensures that we can decode meaningful graphs even when, we sample latent representations from the prior ***p(Z)***.

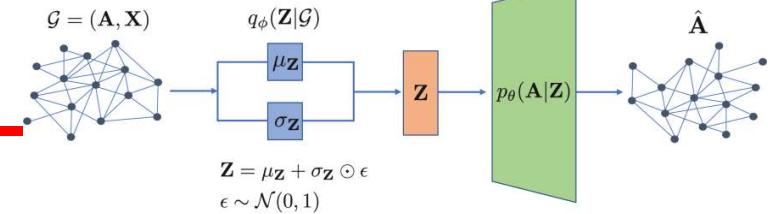
VARIATIONAL AUTOENCODER APPROACHES (Graph)



- ❖ This second goal is critically important:
 - ❖ if we want to generate new graphs after training:
 - ❖ we can generate new graphs by sampling from the prior and feeding these latent embeddings to the decoder.

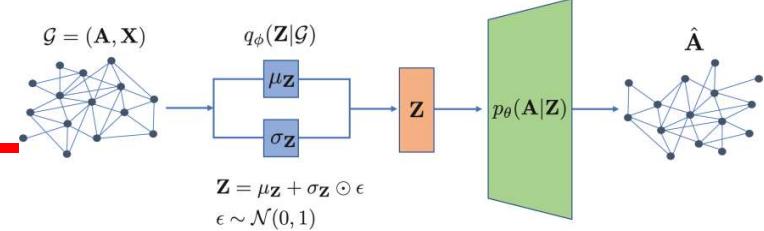
This process will only work if this second goal is satisfied.

VARIATIONAL AUTOENCODER APPROACHES (Graph)



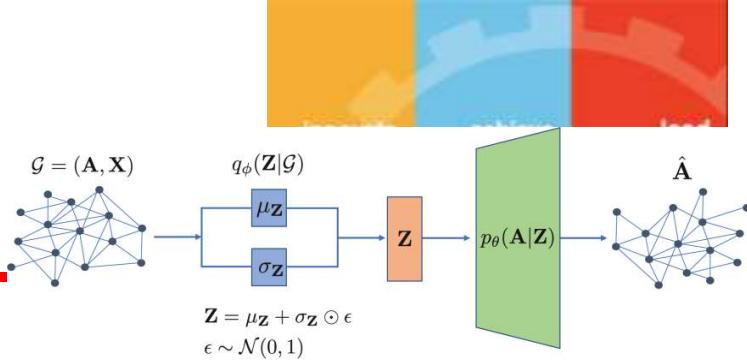
- ❖ Now we will cover two different ways in which the VAE idea can be instantiated for graphs.
- ❖ The approaches differ in how they define the encoder, decoder, and the latent representations.
 - ❖ NODE-LEVEL LATENTS
 - ❖ GRAPH-LEVEL LATENTS

NODE-LEVEL LATENTS



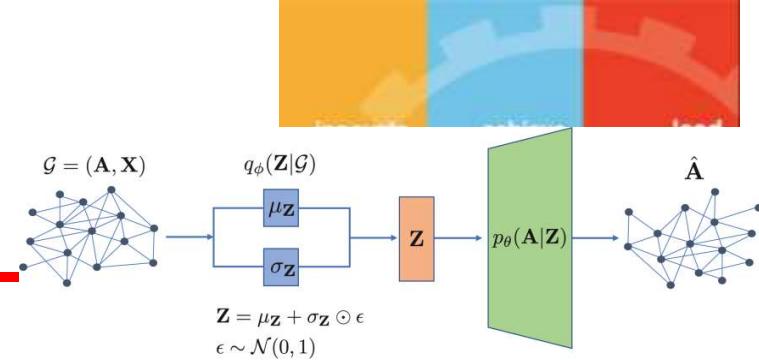
- ❖ This idea was first proposed by **Kipf and Welling [2016b]** and termed the **Variational Graph Autoencoder (VGAЕ)**.
- ❖ It is a framework for ***unsupervised*** learning on graph-structured data based on the Variational Auto-Encoder (VAE).
- ❖ This model makes use of latent variables and is capable of learning interpretable latent representations for undirected graphs.

NODE-LEVEL LATENTS



- ❖ The key idea in this approach is that ***the encoder generates latent representations for each node in the graph.***
- ❖ The ***decoder*** then takes pairs of embeddings as input and uses these embeddings to ***predict*** the ***likelihood*** of an ***edge*** occurring between the two nodes.

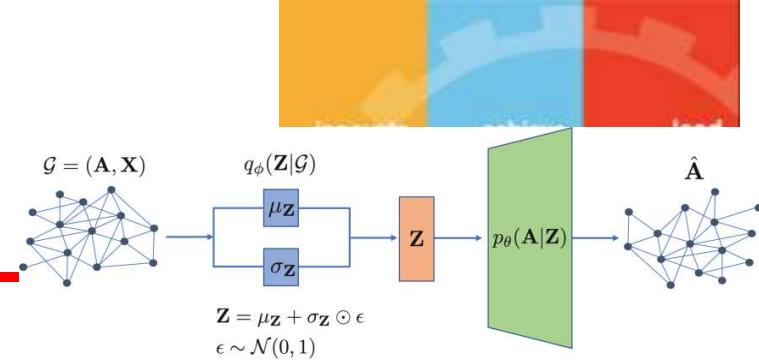
NODE-LEVEL LATENTS



❖ Definitions:

- ❖ They work on an undirected, unweighted graph $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ with $N = |\mathbf{V}|$ nodes.
- ❖ An adjacency matrix \mathbf{A} of \mathbf{G} , is introduced (*assuming diagonal elements set to 1, i.e. every node is connected to itself*) and its degree matrix \mathbf{D} .
- ❖ They further introduce stochastic latent variables \mathbf{z}_i , summarized in an $N \times F$ matrix \mathbf{Z} .
- ❖ Node features are summarized in an $N \times D$ matrix \mathbf{X} .
- ❖ This model uses a graph convolutional network (**GCN**) encoder and a simple *inner product* as *decoder*.

NODE-LEVEL LATENTS



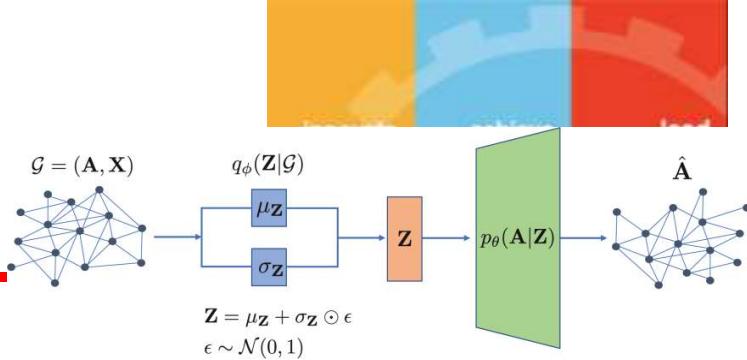
❖ Encoder Model

- ❖ This model uses a graph convolutional network (**GCN**) encoder.
- ❖ In particular, given an adjacency matrix \mathbf{A} and node features \mathbf{X} as input, they use **two** separate **GNNs** to generate **mean** and **variance** parameters, respectively, conditioned on this input:

$$\boldsymbol{\mu}_z = \text{GNN}_\mu(\mathbf{A}, \mathbf{X}) \text{ & } \log \sigma_z = \text{GNN}_\sigma(\mathbf{A}, \mathbf{X})$$

- ❖ Here, $\boldsymbol{\mu}_z$ is a $|V| \times d$ -dimensional matrix, which specifies a mean embedding value *for each node in the input graph*.
- ❖ The $\log \sigma_z \in \mathbb{R}^{|V| \times d}$ matrix similarly specifies the log-variance for the latent embedding of each node.

NODE-LEVEL LATENTS



❖ Encoder Model

- ❖ Latent node embeddings is computed as

$$\mathbf{Z} = \epsilon \circ \exp(\log \sigma_z) + \mu_z$$

- ❖ Here $\epsilon \sim \mathcal{N}(0, 1)$ is a $|v| \times d$ imensional matrix with independently sampled unit **normal** entries, given the encode parameters as, μ_z & $\log \sigma_z$

NODE-LEVEL LATENTS

$$\mathcal{L} = \sum_{G_i \in \{G_1, \dots, G_n\}} \mathbb{E}_{q_\theta(\mathbf{Z}|G_i)} [p_\theta(G_i|\mathbf{Z}) - KL q_\theta(\mathbf{Z}|G_i) || p(\mathbf{Z})]$$

❖ Decoder Model

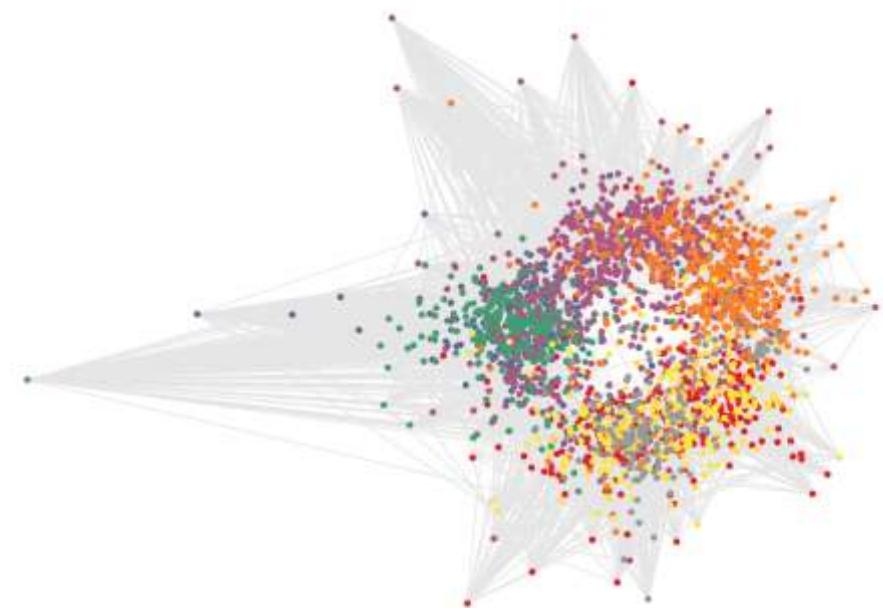
- ❖ Simple *inner product* is used as the decoder:
 $P_\theta(A[u, v] = 1 | z_u, z_v) = \sigma(z_u^T z_v).$
- ❖ To compute the *reconstruction loss*, they assume independence between edges and define the posterior $p_\theta(\mathbf{G}|\mathbf{Z})$ over the full graph as follows:

$$p_\theta(\mathbf{A}|\mathbf{Z}) = \prod_{(u,v) \in \mathcal{V}^2} p_\theta(A[u, v] = 1 | z_u, z_v),$$

- ❖ which corresponds to a binary cross-entropy loss over the edge probabilities.
- ❖ To generate discrete graphs after training, we can sample edges based on the posterior Bernoulli distributions

NODE-LEVEL LATENTS

- ❖ Latent space of unsupervised VGAE model trained on Cora citation network dataset.
- ❖ Grey lines denote citation links.
- ❖ Colors denote document class (not provided during training).



GRAPH-LEVEL LATENTS

$$\mathcal{L} = \sum_{G_i \in \{G_1, \dots, G_n\}} \mathbb{E}_{q_\theta(\mathbf{Z}|G_i)} [p_\theta(G_i|\mathbf{Z}) - KL q_\theta(\mathbf{Z}|G_i) || p(\mathbf{Z})]$$

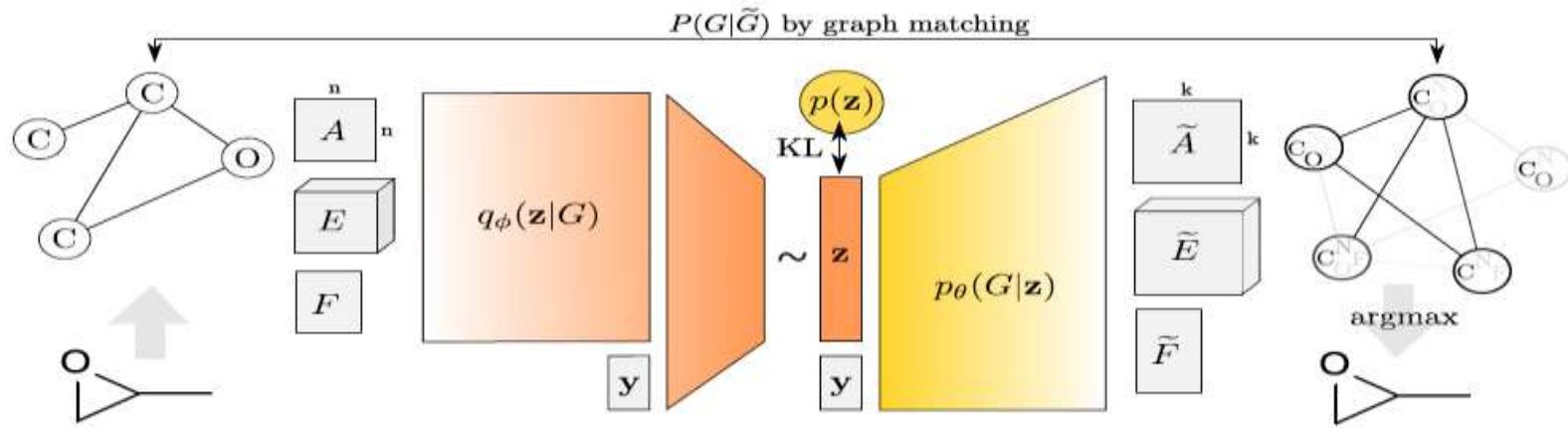
- ❖ As an alternative to the node-level VGAE approach researchers had also define Variational autoencoder based on **graph-level latent representations**.
- ❖ In this approach, they uses the same, ELBO loss to train a VAE model.
- ❖ However, the encoder and decoder functions, are modified to work with graph-level latent representations \mathbf{z}_G .
- ❖ The graph-level VAE was first proposed by ***Simonovsky and Komodakis [2018]***, under the name GraphVAE.

GRAPH-LEVEL LATENTS

- ❖ They coined **GraphVAE**, in cheminformatics on the task of molecule generation.
- ❖ Molecular datasets are a challenging but convenient testbed for the generative model, as they easily allow for both qualitative and quantitative tests of decoded samples.
- ❖ While **GraphVAE** method is applicable for generating smaller graphs only and its performance leaves space for improvement, which can be improved further.
- ❖ As **GraphVAE** outputs a probabilistic graph, and do not assume having a prescribed order of construction transformations available and they formulate the learning problem as an autoencoder.

GRAPH-LEVEL LATENTS

$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_\phi(Z|G)}[-\log p_\theta(G|Z)] - KL[q_\theta(Z|G)||p(Z)]$$



- ❖ Discrete attributed graph $\mathbf{G} = (\mathbf{A}, \mathbf{E}, \mathbf{F})$ on n nodes (e.g. a representation of propylene oxide), adjacency matrix \mathbf{A} , edge attribute tensor \mathbf{E} , and node attribute matrix \mathbf{F} .
- ❖ Stochastic graph encoder $q_\phi(\mathbf{Z}|G)$ embeds the graph into continuous representation \mathbf{z} .
- ❖ Given a point in the latent space, the novel graph decoder $p_\theta(\mathbf{G}|\mathbf{z})$ outputs a probabilistic fully-connected graph $\tilde{\mathbf{G}} = (\tilde{\mathbf{A}}, \tilde{\mathbf{E}}, \tilde{\mathbf{F}})$ on predefined $k \geq n$ nodes, from which discrete samples may be drawn.
- ❖ ϕ, θ are learnable parameters, $p(\mathbf{z})$ as regularization;
- ❖ The process can be conditioned on label \mathbf{y} for controlled sampling at test time. Reconstruction ability of the autoencoder is facilitated by approximate graph matching for aligning \mathbf{G} with $\tilde{\mathbf{G}}$.

GRAPH-LEVEL LATENTS

$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_{\phi}(Z|G)}[-\log p_{\theta}(G|Z)] - KL[q_{\theta}(Z|G) || p(Z)]$$

- ❖ The first term of L , the reconstruction loss, enforces high similarity of sampled generated graphs to the input graph G .
- ❖ The second term, ***KL-divergence***, regularizes the code space to allow for sampling of z directly from $p(z)$ instead from $q_{\phi}(Z|G)$ later.
- ❖ The dimensionality of z is usually fairly small so that the autoencoder is encouraged to learn a high-level compression of the input instead of learning to simply copy any given input.
- ❖ While the regularization is independent on the input space, the reconstruction loss must be specifically designed for each input modality.

GRAPH-LEVEL LATENTS

$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_{\phi}(Z|G)}[-\log p_{\theta}(G|Z)] - KL[q_{\theta}(Z|G) || p(Z)]$$

- ❖ The critical **difference** between this ***graph level*** encoder and the ***node-level*** encoder from the previous section:
 - ❖ Here, we are generating a mean $\mu_{z_G} \in \mathbb{R}^d$ and variance parameter $\log \sigma_{z_G} \in \mathbb{R}^d$ for a single ***graph-level*** embedding $Z_G \sim N(\mu_{z_G}, \sigma_{z_G})$, whereas in the previous section we defined posterior distributions for each individual node.

Summary

- ❖ Learning to generate graphs is a difficult problem for methods based on gradient optimization, as graphs are discrete structures.
 - ❖ Issue resolved by having the decoder output a probabilistic graph (distribution) of a predefined size directly at once.
 - ❖ On the other hand, learning the order for incremental construction involves discrete decisions, which are not differentiable.
 - ❖ Issue resolved by using the reparameterization trick
 - ❖ in a probabilistic graph, the existence of nodes and edges, as well as their attributes, are modeled as independent random variables.
 - ❖ The method is formulated in the framework of Variational autoencoders (VAE)
-

Summary

- ❖ VAEs are a popular framework for deep generative models not just for graphs, but for images, text, and a wide-variety of data domains.
- ❖ VAEs have a well-defined probabilistic motivation, and there are many works that leverage and analyze the structure of the latent spaces learned by VAE models.
- ❖ However, VAEs are also known to suffer from serious limitations such as:
 - ❖ The tendency for VAEs to produce blurry outputs in the image domain.
 - ❖ Many recent state-of-the art generative models leverage alternative generative frameworks, with GANs being one of the most popular

References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3
 - Graph Neural Networks: Foundations, Frontiers, and Applications, Editors Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, Springer, 2022.: DOI
<https://doi.org/10.1007/978-981-16-6054-2>
-



BITS Pilani
Pilani Campus



Thank You



Graphs Neural Network

BITS Pilani
Pilani Campus



Reconstruction Loss in Graph based Variational AutoEncoders

Reconstruction Loss in Graph Generation



$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_\phi(Z|G)}[-\log p_\theta(G|Z)] + KL[q_\theta(Z|G) \| p(Z)]$$

- ❖ Graph reconstruction involves predicting the original graph or a similar graph from a set of learned representations
- ❖ The **reconstruction loss** measures the **difference** between the **predicted** graph and the **original** graph.
- ❖ The goal is to train the GNN, such that, the **reconstruction loss is minimized** i.e. *to encourage the generated graphs to be similar to the input graphs.*
 - ❖ Node-level Reconstruction.
 - ❖ Graph-level Reconstruction.

Reconstruction Loss in Graph Generation



❖ Node-level Reconstruction:

- ❖ For node-level reconstruction, the GNN aims to reconstruct the attributes or embeddings of each node in the graph.
- ❖ The reconstruction loss can be defined using a suitable distance metric, such as mean squared error (MSE) or binary cross-entropy, between the predicted and true node representations.

Reconstruction Loss in Graph Generation



❖ Graph-level Reconstruction:

- ❖ In graph-level reconstruction, the goal is to reconstruct the entire graph structure.
- ❖ The reconstruction loss can be formulated based on graph adjacency matrices, edge predictions, or other graph representations.
- ❖ Commonly used loss functions include binary cross-entropy loss for edge prediction or graph similarity metrics.



Binary Cross Entropy (Log Loss)

Binary Cross Entropy (BCE)

$$BCE = -\frac{1}{N} \sum_{i=1}^n \log(p_i)$$

- ❖ **Binary Cross Entropy (BCE)** is a commonly used loss function in machine learning, particularly in binary classification problems.
- ❖ It measures the difference between the predicted probability distribution and the true distribution for a binary outcome.
- ❖ Binary cross entropy compares each of the predicted probabilities to actual class output which can be either 0 or 1. It then calculates the score that penalizes the probabilities based on the distance from the expected value. ***That means how close or far from the actual value.***

Binary Cross Entropy (LogLoss)

$$BCE = -\frac{1}{N} \sum_{i=1}^n \log(p_i)$$

- ❖ Binary Cross Entropy is the **negative average of the log of corrected predicted probabilities**.
- ❖ The reason behind using the log value is,
 - ❖ The log value offers less penalty for small differences between predicted probability and corrected probability.
 - ❖ When the difference is large the penalty will be higher.
- ❖ We calculate log values for all the corrected probabilities. Since all the corrected probabilities lie between 0 and 1, all the log values are negative.
- ❖ *In order to compensate for this negative value, we will use a negative average of the values.*

Binary Cross Entropy (Graph)

- ❖ In the context of a graph generation, we have an adjacency matrix, BCE is often used when dealing with binary graph edges (*presence or absence of edges between nodes*).
- ❖ Here's a general overview of how BCE can be applied to an adjacency matrix:
- ❖ Let's denote:
 - ❖ \mathbf{A} as the predicted adjacency matrix (output of your model),
 - ❖ \mathbf{Y} as the true adjacency matrix (ground truth).
- ❖ For a single entry in the matrices, let A_{ij} represent the predicted probability of the edge between node i and node j , and Y_{ij} represent the true label (*1 if there is an edge, 0 otherwise*).

Binary Cross Entropy (BCE)

- ❖ The BCE loss for a single entry is calculated as follows:
$$BCE(A_{ij}, Y_{ij}) = -(Y_{ij} * \log(A_{ij})) + (1 - Y_{ij}) * \log(1 - A_{ij})$$
- ❖ When the observation belongs to class 1 the first part of the formula becomes active and the second part vanishes and vice versa in the case observation's actual class are 0. This is how we calculate the Binary cross-entropy.
- ❖ This expression penalizes the model more if it predicts a high probability for a non-existent edge or a low probability for an existing edge.

Binary Cross Entropy (BCE)

$$BCE(A_{ij}, Y_{ij}) = -(Y_{ij} * \log(A_{ij}) + (1 - Y_{ij}) * \log(1 - A_{ij}))$$

- ❖ The overall BCE loss for the entire adjacency matrix can be computed by averaging the losses over all the entries:

$$BCE_{Loss(A,Y)} = \frac{1}{N} \sum_{i=1}^N \sum_{j=1}^N BCE(A_{ij}, Y_{ij})$$

In practice, many deep learning frameworks, such as TensorFlow or PyTorch, provide built-in functions to compute BCE loss, making it easy to incorporate into your model training process



Graph Similarity

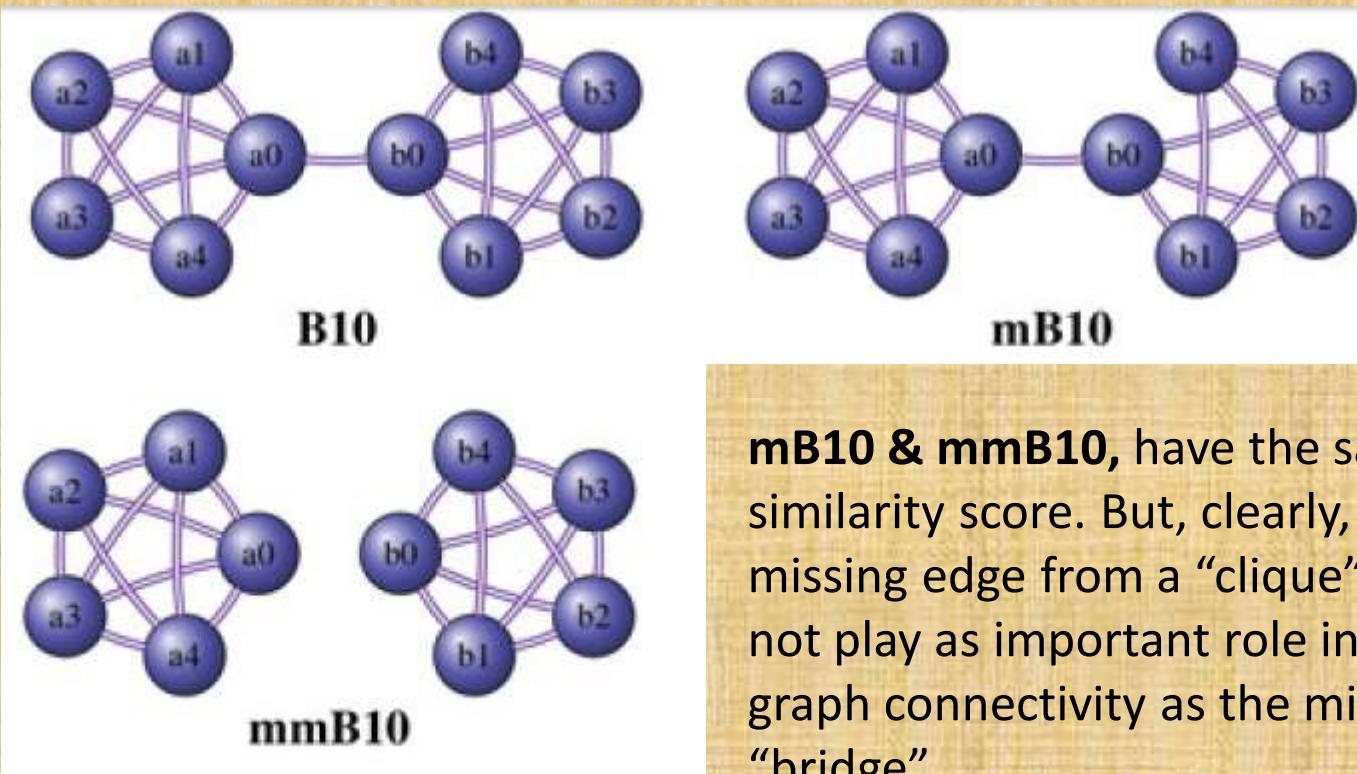
Similarity(Definition)

- ❖ We are Given with:
 - i. Two graphs $G_1(\nu, \varepsilon_1)$ and $G_2(\nu, \varepsilon_2)$ with the **same** node set ν , but **different** edge set ε_1 and ε_2 , and
 - ii. The Node correspondence.
- ❖ Find the similarity score, $sim(G_1, G_2) \in [0, 1]$, between the input graph.

A score of the value **0**, means totally **different**, while **1** means **identical** graphs.

Graph Similarity

- ❖ The obvious way to solve this problem is by measuring the overlap of their edges.
- ❖ Why does this often not work in practice?



mB10 & mmB10, have the same similarity score. But, clearly, a missing edge from a “clique”, does not play as important role in the graph connectivity as the missing “bridge”.

2-Step Process for Graph similarity

- ❖ The **first** conceptual step of is to compute the **pairwise node affinities** in the first graph, and compare them with the ones in the second graph.
 - ❖ For notational compactness, we store them in a $n \times n$ similarity matrix \mathbf{S} .
 - ❖ The s_{ij} entry of the matrix indicates the influence node i has on node j .
- ❖ The **second** conceptual step is to measure the differences in the corresponding node affinity scores of the **two graphs** and report the result as their similarity score.

DESIRED PROPERTIES FOR SIMILARITY MEASURES



- ❖ Let two graphs $G_1(\nu, \varepsilon_1)$ and $G_2(\nu, \varepsilon_2)$ be the two graphs, and $\text{sim}(G_1, G_2) \in [0, 1]$, denote their similarity score.
 - ❖ Then, we want the similarity measure to obey the following axioms:
 - ❖ **A₁**: *Identity property*: $\text{sim}(G_1; G_1) = 1$.
 - ❖ **A₂**: *Symmetric property*: $\text{sim}(G_1; G_2) = \text{sim}(G_2; G_1)$.
 - ❖ **A₃**: *Zero property*: $\text{sim}(G_1; G_2) \rightarrow 0$ for $n \rightarrow \infty$, where G_1 is the **complete** graph (K_n), and G_2 is the **empty** graph (i.e., the edge sets are complementary).

DESIRED PROPERTIES FOR SIMILARITY MEASURES



Moreover, the measure must be as follows:

a) **Intuitive** It should satisfy the following desired properties.

❖ **P₁**. [*Edge Importance*]:

❖ For unweighted graphs, changes that create disconnected components should be penalized more than changes that maintain the connectivity properties of the graphs.

❖ **P₂**. [*Edge-“Submodularity”*]

❖ For unweighted graphs, a specific change is more important in a graph with few edges than in a much denser, but equally sized graph.

DESIRED PROPERTIES FOR SIMILARITY MEASURES



Moreover, the measure must be as follows:

a) Intuitive It should satisfy the following desired properties.

❖ **P₃**. [*Weight Awareness*]

❖ In weighted graphs, the bigger the weight of the removed edge is, the greater the impact on the similarity measure should be.

❖ **IP**. [*Focus Awareness*]

❖ “Random” changes in graphs are less important than “targeted” changes of the same extent.

(b) Scalable: The huge size of the generated graphs, as well as their abundance require a similarity measure that is computed fast and handles graphs with billions of nodes.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ Edit distance/graph isomorphism,
 - ❖ Feature extraction, and
 - ❖ Iterative methods.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ Edit distance/**graph isomorphism**:
 - ❖ Two graphs are similar if they are isomorphic [2], or one is isomorphic to a subgraph of the other, or they have isomorphic subgraphs.
 - ❖ The drawback is, the exact versions of the algorithms are exponential and, thus, not applicable to the large graphs that are of interest today.

[2] M Pelillo. Replicator equations, maximal cliques, and graph isomorphism. *Neural Computation*, 11(8):1933– 1955, 1999.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Edit distance/graph isomorphism:**
 - ❖ The graph edit distance is a generalization of the graph isomorphism problem, where the target is to transform one graph to the other by doing a number of operations (additions, deletions, substitutions of nodes or edges, and reversions of edges).
 - ❖ This method associates each operation with a cost and it attempts to find the sequence of operations that minimizes the cost of matching the two graphs.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Feature extraction:**
 - ❖ The key idea behind these methods is that similar graphs probably share certain properties, such as degree distribution, diameter, eigenvalues [3].
 - ❖ After extracting these features, a similarity measure [4] is applied in order to assess the similarity between the aggregated statistics and, equivalently, the similarity between the graphs.

[3] Duncan J Watts. **Small Worlds**, volume 19. Princeton University Press, 1999.

[4] Sung-Hyuk Cha. **Comprehensive survey on distance / similarity measures between probability density functions**. International Journal of Mathematical Models and Methods in Applied Sciences, 1(4):300–307, 2007.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Feature extraction:**
 - ❖ These methods are powerful and scale well, as they map the graphs to several statistics that are much smaller in size than the graphs.
 - ❖ However, depending on the statistics that are chosen, it is possible to get results that are not intuitive.
 - ❖ For instance, **it is possible to get high similarity between two graphs that have very different node set size, which is not always desirable.**

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:

- ❖ **Iterative methods :**

- ❖ Two nodes are similar “*if their neighborhoods are also similar*”.
 - ❖ In each iteration, the nodes exchange similarity scores and this process ends when convergence is achieved. Such as **similarity flooding algorithm by Melnik et al. [6]**
 - ❖ This algorithm solves the “matching” problem, i.e., *it attempts to find the correspondence between the nodes of two given graphs.*

[6] Sergey Melnik, Hector Garcia-Molina, and Erhard Rahm. **Similarity flooding: A versatile graph matching algorithm and its application to schema matching.** In 18th International Conference on Data Engineering (ICDE 2002), 2002.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Iterative methods :**
 - ❖ **Similarity flooding algorithm**
 - ❖ The way the algorithm is evaluated: humans check whether the matchings are correct, and the accuracy of the algorithms is computed based on the number of adaptations that have to be done in the solutions in order to get the right ones.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Iterative methods :**
 - ❖ Another algorithm is **SimRank [7]**,
 - ❖ It measures the **self-similarity of a graph**, ie. *it assesses the similarities between all pairs of nodes in one graph*, it is based on the notion that **similar nodes have similar neighborhoods**.
 - ❖ It computes iteratively *all pairs similarity scores*, by propagating similarity scores in the A^2 matrix, where A is the adjacency matrix of the graph; the process ends when convergence is achieved.

[7] Glen Jeh and Jennifer Widom. **SimRank: a measure of structural-context similarity**. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD '02, pages 538–543, New York, NY, USA, 2002. ACM.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:

- ❖ **Iterative methods** : (with different # nodes)

- ❖ Another iterative approaches is due to Kleinberg [9], further generalized by Blondel et al. [10].

- ❖ In the method of Blondel et al. the update rule for x_{ij} in step $k + 1$ is given by: (for two directed graphs)

$$x_{ij}^{k+1} \leftarrow \sum_{(p,i) \in E_A, (q,j) \in E_B} x_{pq}^k + \sum_{(i,p) \in E_A, (j,q) \in E_B} x_{pq}^k$$

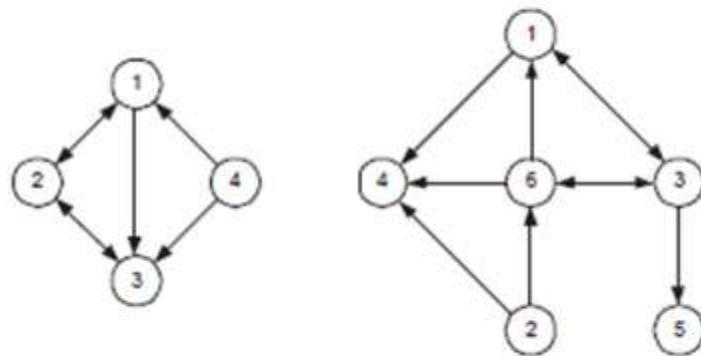
- ❖ The similarity matrix X is **normalized** by $X \leftarrow \|X\|_2$ after each step

[9] J.M. Kleinberg, Authoritative sources in a hyperlinked environment, *Journal of the ACM* **46** (1999), 604–632.

[10] V.D. Blondel, A. Gajardo, M. Heymans, P. Snellart and P. Van Dooren, A measure of similarity between graph vertices: Applications to synonym extraction and web searching, *SIAM Review* **46** (2004), 647–666..

Graph Similarity (Survey)

SIMILARITY IN GRAPHS



0.2720	0.2840	0.2700	0.1440
0.1400	0.1390	0.0670	0.1390
0.2700	0.3120	0.2560	0.1650
0.2300	0.2440	0.2870	0.0720
0.0660	0.0720	0.1030	0
0.2540	0.2480	0.2340	0.1460

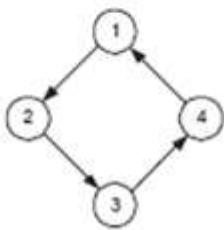
FIG. 1.2. Two graphs G_A, G_B and their similarity matrix. The vertex of G_A which is most similar to vertex 5 in G_B is vertex 3.

[10] V.D. Blondel, A. Gajardo, M. Heymans, P. Snellart and P. Van Dooren, **A measure of similarity between graph vertices: Applications to synonym extraction and web searching**, SIAM Review **46** (2004), 647–666..

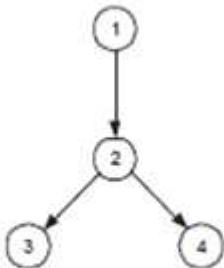
Graph Similarity (Survey)



$$\begin{bmatrix} 0.408 & 0 & 0 \\ 0 & 0.816 & 0 \\ 0 & 0 & 0.408 \end{bmatrix}$$



$$\begin{bmatrix} 0.250 & 0.250 & 0.250 & 0.250 \\ 0.250 & 0.250 & 0.250 & 0.250 \\ 0.250 & 0.250 & 0.250 & 0.250 \\ 0.250 & 0.250 & 0.250 & 0.250 \end{bmatrix}$$



$$\begin{bmatrix} 0.182 & 0 & 0 & 0 \\ 0 & 0.912 & 0 & 0 \\ 0 & 0 & 0.182 & 0.182 \\ 0 & 0 & 0.182 & 0.182 \end{bmatrix}$$

FIG. 5.1. Graphs and their corresponding self-similarity matrix. The self-similarity matrix of a graph gives a measure of how similar vertices are to each other.

[10] V.D. Blondel, A. Gajardo, M. Heymans, P. Snellart and P. Van Dooren, **A measure of similarity between graph vertices: Applications to synonym extraction and web searching**, *SIAM Review* **46** (2004), 647–666..

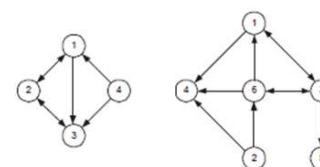
Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Iterative methods :**
 - ❖ Another recursive algorithm related to graph similarity and matching is the algorithm proposed by **Zager and Verghese [8]**;
 - ❖ This method introduces the idea of coupling the similarity scores of nodes and edges in order to compute the similarity between two graphs:

[8] L Zager and G Verghese. **Graph similarity scoring and matching.** Applied Mathematics Letters, 21(1):86–94, 2008.

Graph Similarity (Survey)

SIMILARITY IN GRAPHS



0.2720	0.2840	0.2700	0.1440
0.1400	0.1390	0.0670	0.1390
0.2700	0.3120	0.2560	0.1650
0.2300	0.2440	0.2870	0.0720
0.0660	0.0720	0.1030	0
0.2540	0.2480	0.2340	0.1460

FIG. 1.2. Two graphs G_A, G_B and their similarity matrix. The vertex of G_A which is most similar to vertex 5 in G_B is vertex 3.

- ❖ The proposed techniques can be classified into three main categories:

- ❖ **Iterative methods :**

- ❖ The update rule for the edge similarity matrix $Y = [y_{uv}]$, where $u \in E_A$ and $v \in E_B$, is given by:

$$y_{uv}^{k+1} \leftarrow x_{s(u)s(v)}^k + x_{t(u)t(v)}^k$$

- ❖ The update rule for similarity of nodes is then given in terms of similarities of the edges

$$x_{ij}^{k+1} \leftarrow \sum_{t(u)=i, t(v)=j} y_{uv}^k + \sum_{s(u)=i, s(v)=j} y_{uv}^k$$

- ❖ Matrix normalization of the similarity scores is applied in this approach too

Graph Similarity (Survey)

L.A. Zager, G.C. Verghese / Applied Mathematics Letters 21 (2008) 86–94

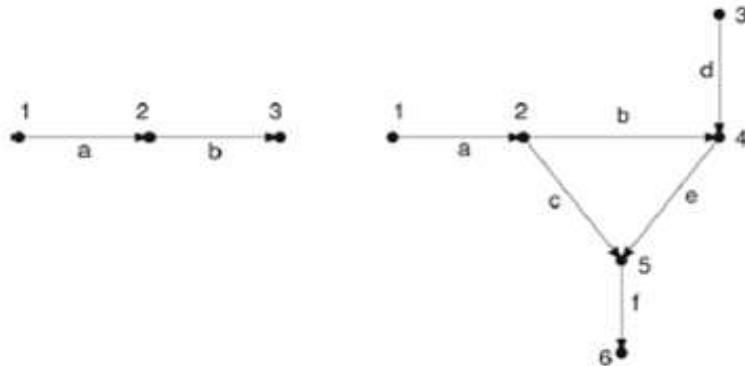


Fig. 2. Two directed graphs.

Table 1
 X , the node similarity scores, and Y , the edge similarity scores for the graphs of Fig. 2

X				Y		
Nodes	1	2	3	Edges	a	b
1	0.124	0	0	a	0.265	0
2	0.348	0.445	0	b	0.426	0.297
3	0.157	0.054	0	c	0.320	0.389
4	0.094	0.563	0.193	d	0.336	0.115
5	0	0.338	0.340	e	0.202	0.445
6	0	0	0.094	f	0	0.202

As discussed in Section 2, higher scores denote greater similarity.

Graph Similarity (Survey)

- ❖ The proposed techniques can be classified into three main categories:
 - ❖ **Iterative methods :**
 - ❖ Koutra et. al [9] described a new approach “*the method of neighbor matching*” for measuring similarity of nodes of graphs, then, they define a measure of similarity of whole graphs based on the similarities of their nodes.
 - ❖ *Two nodes $i \in V_A$ and $j \in V_B$ are considered to be similar if neighbor nodes of i can be matched to similar neighbor nodes of j (hence the name neighbor matching).*

Graph Similarity (Survey)

- ❖ They had differentiated between *in-similarity* s_{in} and *out-similarity* s_{out} and gave them equal weights.
- ❖ In order to calculate *in-similarity*, the matching of *in-neighbors* with maximal sum of similarities has to be constructed, and analogously for out-similarity. More formally, the update rule is given by:

Graph Similarity (Survey)

$$x_{ij}^{k+1} \leftarrow \frac{s_{in}^{k+1}(i,j) + s_{out}^{k+1}(i,j)}{2}, \text{ where as;}$$

$$s_{in}^{k+1}(i,j) \leftarrow \frac{1}{m_{in}} \sum_{l=1}^{n_{in}} x_{f_{ij}^{in}(l)g_{ij}^{in}(l)}^k;$$

$$s_{out}^{k+1}(i,j) \leftarrow \frac{1}{m_{out}} \sum_{l=1}^{n_{out}} x_{f_{ij}^{out}(l)g_{ij}^{out}(l)}^k$$

$$m_{in} = \max(id(i), id(j)); \quad m_{out} = \max(od(i), od(j))$$

$$n_{in} = \min(id(i), id(j)); \quad n_{out} = \min(od(i), od(j))$$

- ❖ f_{ij}^{in} and g_{ij}^{in} , are the enumeration functions of the optimal matching of in-neighbors of nodes i and j with weight function $w(a, b) = x_{ab}^k$, and analogously for f_{ij}^{out} and g_{ij}^{out}

Graph Similarity (Survey)

- ❖ We define $\frac{0}{0} = 1$.
 - ❖ This convention ensures that the similarity of nodes with no *in* or no *out* neighbors is recognized.
- ❖ If there is a difference in the number of in or out neighbors, that difference is penalized when calculating corresponding similarities since m_{in} and m_{out} are greater than the number of terms in the summation.
- ❖ The first iteration is:

$$s_{in}^1 = \frac{\min(id(i), id(j))}{\max(id(i), id(j))} \quad \& \quad s_{out}^1 = \frac{\min(od(i), od(j))}{\max(od(i), od(j))}$$

References

1. Danai Koutra, Christos Faloutsos, “***Individual and collective Graph Mining: principles, Algorithm, and Application***”, 2017.
2. Laura A.ZagerGeorge C.Verghese, “**Graph similarity scoring and matching**”, Applied Mathematics Letters; Volume 21, Issue 1, January 2008, Pages 86-94
3. Mladen Nikolić, “**Measuring similarity of graph nodes by neighbor matching**”, in Intelligent Data Analysis, Volume 16, Issue 6, 2012.
4. Danai Koutra, Ankur Parikh, Aaditya Ramdas, Jing Xiang, “**Algorithms for Graph Similarity and Subgraph Matching**”, 2011.



BITS Pilani
Pilani Campus



Thank You

Summery (VAE)

$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_{\phi}(Z|G)}[-\log p_{\theta}(G|Z)] - KL[q_{\theta}(Z|G) || p(Z)]$$

- ❖ Learning to generate graphs is a difficult problem for methods based on **gradient optimization**, as graphs are discrete structures.
 - ❖ **Issue resolved** by having the decoder output a ***probabilistic graph*** (distribution) of a predefined size directly at once.
- ❖ On the other hand, learning the order for incremental construction involves discrete decisions, which are **not differentiable**.
 - ❖ **Issue resolved** by using the ***reparameterization trick***.

Summery (VAE)

$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_{\phi}(Z|G)}[-\log p_{\theta}(G|Z)] - KL[q_{\theta}(Z|G) || p(Z)]$$

- ❖ in a probabilistic graph, the existence of nodes and edges, as well as their attributes, are modeled as independent random variables
- ❖ The method is formulated in the framework of Variational autoencoders (VAE).
 - ❖ VAEs are a popular framework for deep generative models not just for graphs, but for images, text, and a wide-variety of data domains.
 - ❖ VAEs have a well-defined probabilistic motivation, and there are many works that leverage and analyze the structure of the latent spaces learned by VAE models.

Summery (VAE)

$$\mathcal{L}(\phi, \theta, G) = \mathbb{E}_{q_\phi(Z|G)}[-\log p_\theta(G|Z)] - KL[q_\theta(Z|G) || p(Z)]$$

- ❖ However, VAEs are also known to suffer from serious **limitations** (*image processing domain*) such as:
 - ❖ The tendency for VAEs to produce blurry outputs in the image domain.
- ❖ Many recent state-of-the art generative models leverage alternative generative frameworks, with GANs being one of the most popular



Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs)



- ❖ Generative Adversarial Networks (GANs) are a class of neural networks that are trained to generate new data that is similar to a training dataset.
- ❖ They consist of two neural networks: **a generator network** that creates new data samples, and a **discriminator network** that tries to distinguish between the generated samples and the real samples.
- ❖ GANs have been successful in generating realistic images, videos, and audio, and have been applied to a variety of applications, including art generation, data augmentation, and synthetic data generation.

Generative Adversarial Networks (GANs)



- ❖ GANs are used for generating new data that is similar to a training dataset, and can be used for a variety of machine learning tasks:
 - ❖ **Image generation:** GANs can be used to generate new images that are similar to a training dataset, for applications such as art generation, data augmentation, and synthetic data generation.
 - ❖ **Video generation:** GANs can be used to generate new videos that are similar to a training dataset, for applications such as video prediction and video synthesis.
 - ❖ **Text generation:** GANs can be used to generate new text that is similar to a training dataset, for applications such as language translation and text summarization.

Generative Adversarial Networks (GANs)



- ❖ GANs offer several **benefits** for machine learning and artificial intelligence:
 - ❖ **Data augmentation:** GANs can be used to generate synthetic data that can be used for data augmentation, improving the performance and accuracy of machine learning models.
 - ❖ **Art generation:** GANs can be used to generate art and other creative works, demonstrating the potential of AI for artistic expression.
 - ❖ **Synthesis of data:** GANs can be used to synthesize new data that is similar to a training dataset, expanding the available data for machine learning and AI applications.

GAN-based generative models

- ❖ First, we define a trainable generator network $g_\theta: \mathbb{R}^d \rightarrow X$.
- ❖ This **generator** network is trained to generate realistic (*but fake*) data samples $\tilde{x} \in X$ by taking a random seed $z \in \mathbb{R}^d$ as input (e.g., *a sample from a normal distribution*).
- ❖ At the same time, we define a **discriminator** network $d_\theta: x \rightarrow [0, 1]$.
- ❖ The goal of the discriminator is to distinguish between real data samples $x \in X$ and samples generated by the generator $\tilde{x} \in X$.
- ❖ Here, we will assume that discriminator outputs the probability that a given input is fake.

GAN-based generative models

- ❖ To train a GAN, both the generator and discriminator are optimized jointly in an *adversarial game*:

$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log(1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log(d_{\phi}(g_{\theta}(z)))]$$

- ❖ Here $p_{data}(x)$ denotes the empirical distribution of real data samples (e.g., a *uniform sample over a training set*) and $p_{seed}(z)$ denotes the random seed distribution (e.g., a *standard multivariate normal distribution*).
- ❖ Equation above represents a **minimax** optimization problem.
 - ❖ The generator is attempting to minimize the discriminatory power of the discriminator, while the discriminator is attempting to maximize its ability to detect fake samples

Benefits and Limitations of the GAN Approach



$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log(1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log(d_{\phi}(g_{\theta}(z)))]$$

- ❖ One important benefit of the GAN-based framework is that *it removes the complication of specifying a node ordering in the loss computation.*
- ❖ As long as the discriminator model is *permutation invariant*, which is the case for almost every GNN, then the GAN approach does not require any node ordering to be specified.
- ❖ *The ordering of the adjacency matrix generated by the generator is immaterial if the discriminator is permutation invariant.*
- ❖ The only difficulties involved in GAN is in the minimax optimization

References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3
- Graph Neural Networks: Foundations, Frontiers, and Applications, Editors Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, Springer, 2022.: DOI
<https://doi.org/10.1007/978-981-16-6054-2>



BITS Pilani
Pilani Campus



Thank You

GAN-based generative models

- ❖ To train a GAN, both the generator and discriminator are optimized jointly in an *adversarial game*:

$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log(1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log(d_{\phi}(g_{\theta}(z)))]$$

- ❖ Here $p_{data}(x)$ denotes the empirical distribution of real data samples (e.g., a *uniform sample over a training set*) and $p_{seed}(z)$ denotes the random seed distribution (e.g., a *standard multivariate normal distribution*).
- ❖ Equation above represents a **minimax** optimization problem.

❖ The **generator** is attempting to **minimize** the **discriminatory power** of the discriminator, while the **discriminator** is attempting to **maximize** its **ability to detect fake samples**

Benefits and Limitations of the GAN Approach



$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log(1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log(d_{\phi}(g_{\theta}(z)))]$$

- ❖ One important benefit of the GAN-based framework is that *it removes the complication of specifying a node ordering in the loss computation.*
- ❖ As long as the discriminator model is *permutation invariant*, which is the case for almost every GNN, then the GAN approach does not require any node ordering to be specified.
- ❖ *The ordering of the adjacency matrix generated by the generator is immaterial, if the discriminator is permutation invariant.*
- ❖ The only difficulties involved in GAN is in the minimax optimization



MolGAN: An implicit generative model for small molecular graphs

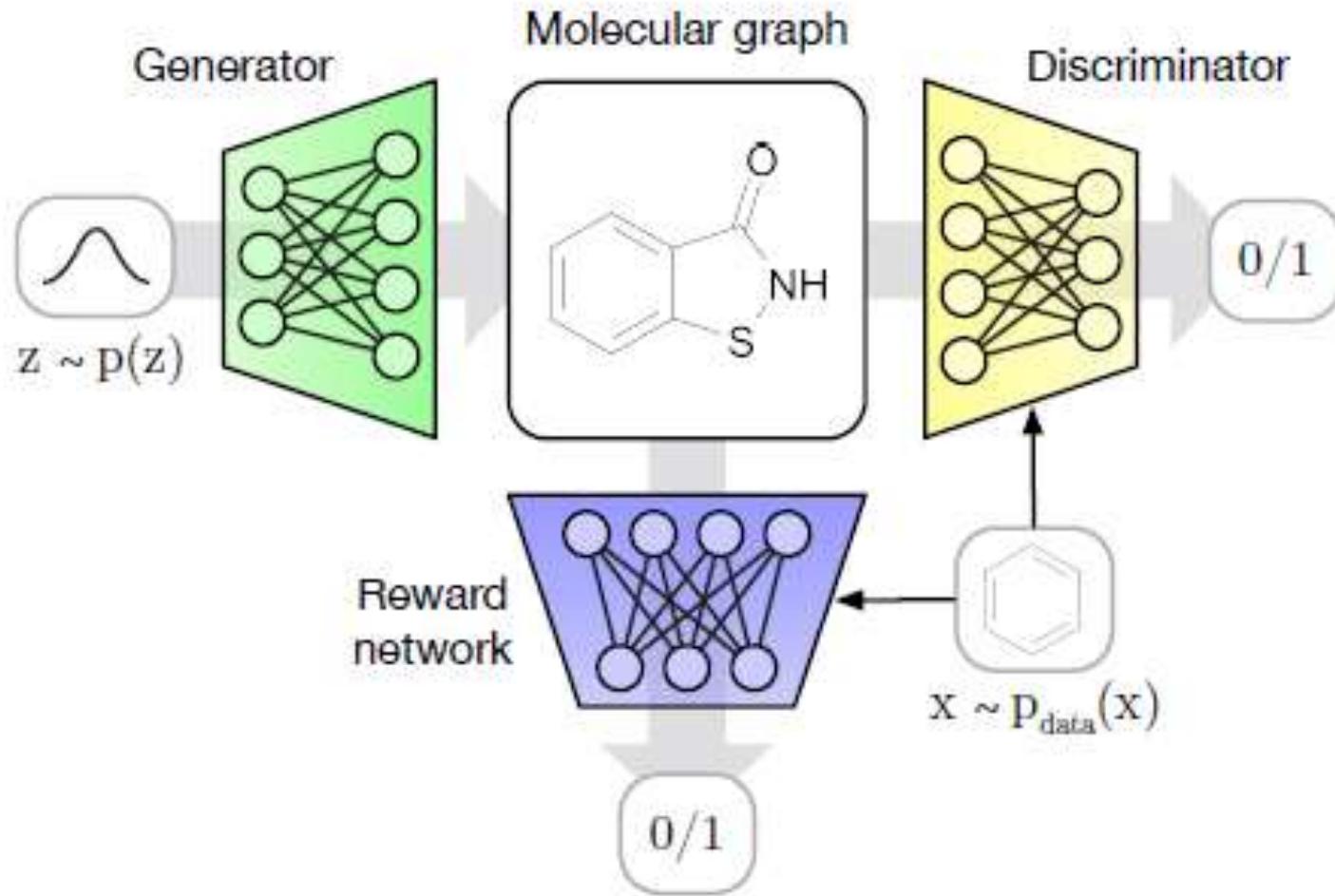
- ❖ They introduce **MolGAN**, an implicit, *likelihoodfree* generative model for ***small molecular graphs*** that circumvents the need for expensive graph matching procedures or node ordering heuristics of previous likelihood-based methods.
- ❖ Their method adapts generative adversarial networks (**GANs**) to operate directly on graph-structured data.
- ❖ They combine their approach with a *reinforcement learning* objective to encourage the generation of molecules with specific desired chemical properties.



MolGAN: An implicit generative model for small molecular graphs

- ❖ The generative model of MolGAN ***predicts discrete graph structure at once*** (i.e., nonsequentially) for computational efficiency, although sequential variants are possible in general.
- ❖ MolGAN further utilizes a **permutation-invariant discriminator and reward network** (*for RL-based optimization towards desired chemical properties*) based on Graph Convolution Layers

MolGAN

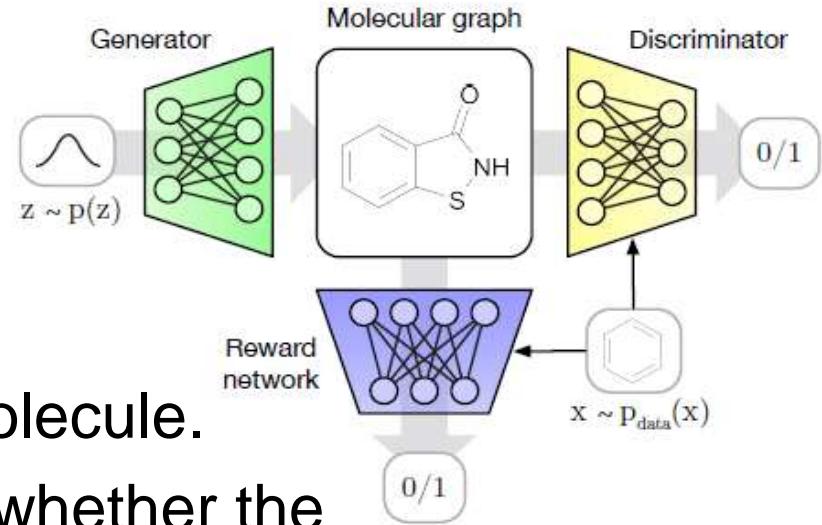


MolGAN

$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log(1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log(d_{\phi}(g_{\theta}(z)))]$$

❖ Schema of MolGAN.

- ❖ A vector z is sampled from a **prior** and passed to the generator which outputs the graph representation of a molecule.
- ❖ The discriminator classifies whether the molecular graph comes from the generator or the dataset.
- ❖ The reward network tries to estimate the reward for the chemical properties of a particular molecule provided by an external software.



MolGAN

$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log (1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log (d_{\phi}(g_{\theta}(z)))]$$

The MolGAN Architecture

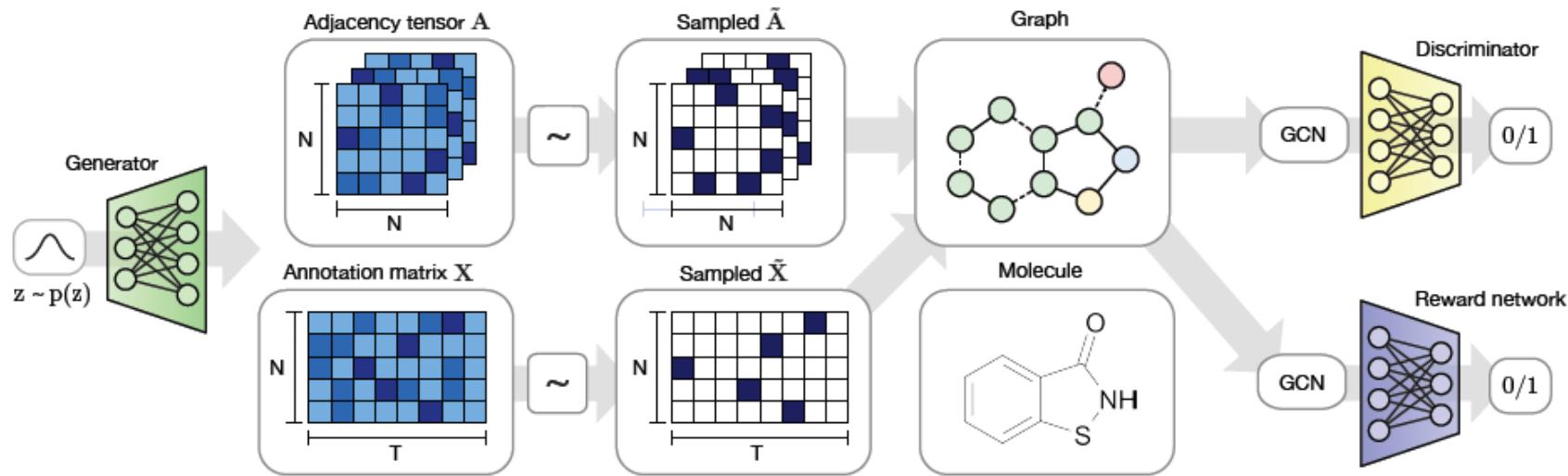


Figure 2. Outline of MolGAN. From left: the generator takes a sample from a prior distribution and generates a dense adjacency tensor A and an annotation matrix X . Subsequently, sparse and discrete \tilde{A} and \tilde{X} are obtained from A and X respectively via categorical sampling. The combination of \tilde{A} and \tilde{X} represents an annotated molecular graph which corresponds to a specific chemical compound. Finally, the graph is processed by both the discriminator and reward networks that are invariant to node order permutations and based on Relational-GCN (Schlichtkrull et al., 2017) layers.

MolGAN

$$\min_{\theta} \max_{\phi} E_{x \sim p_{data}(x)} [\log(1 - d_{\phi}(x))] + E_{z \sim p_{seed}(z)} [\log(d_{\phi}(g_{\theta}(z)))]$$

❖ Summary:

- ❖ MolGAN: is an implicit generative model for molecular graphs of small size.
- ❖ Through joint training with a GAN and an RL objective, the model is capable of generating molecular graphs with both higher validity and novelty than previous comparable VAE-based generative models, while not requiring a permutation-dependent likelihood function.



AUTOREGRESSIVE METHODS

AUTOREGRESSIVE METHODS



- ❖ Both the basic GAN and VAE-based approaches that we discussed used simple MLPs to generate adjacency matrices.
- ❖ Now we will introduce more sophisticated ***autoregressive*** methods that can decode graph structures from latent representations.
- ❖ *Autoregressive* models originated from economics and social science literature on time-series data, where observations from the previous steps are used to predict the value at the current and at future time steps.

AUTOREGRESSIVE METHODS

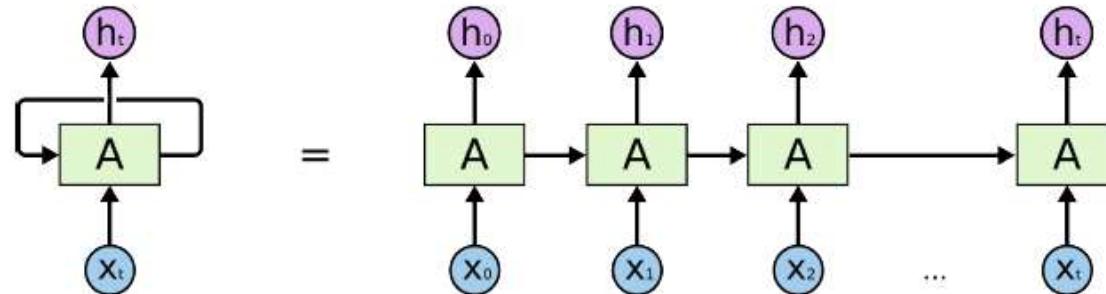


- ❖ The shared underlying idea of these autoregressive models is to characterize the graph generation process as a sequential decision-making process and make a new decision at each step conditioning on all previously made decisions.
- ❖ We can model this, using the concept of Recurrent Neural Network (RNN), as it also works on **sequences**.
- ❖ The Generation of graph can also be considered as the sequence of node & edges generation.

AUTOREGRESSIVE METHODS



- ❖ It can be compared with the **Recurrent Neural Networks** (RNNs), which are far more well-known.
- ❖ Like an RNN, an autoregressive model's output, at time t , depends on, not just x_t , but also x 's from previous time steps.
- ❖ However, unlike an RNN, the previous x 's are not provided via some hidden state, they are given as just another input to the model.



Some Thoughts and Observation

- ❖ Given previous values x_1, x_2, \dots, x_n , these models do not output a value for x_{t+1} , they output the ***predictive probability distribution*** $p(x_{t+1}|x_1, x_2, \dots, x_n)$ for x_{t+1} .
- ❖ Autoregressive models are **supervised**, whereas VAE & GAN, were unsupervised.
 - ❖ This means that, training is stable and highly parallelizable, *i.e.*, it is straightforward to tune hyper parameters, and that inference is computationally inexpensive.
- ❖ Autoregressive models work on both **continuous** and **discrete** data.

Some Thoughts and Observation

- ❖ Autoregressive models are very **amenable to conditioning.**
 - ❖ They can be conditioned:
 - ❖ On both discrete and continuous variables;
 - ❖ At multiple time scales;
 - ❖ On latent embeddings or the outputs of other neural networks.
 - ❖ They can only be conditioned on the data's labels, *i.e.*, unlike a GAN, They can be conditioned on random noise and expect the model to shape the noise space into a semantically (stylistically) meaningful latent space.

Some Thoughts and Observation

- ❖ Autoregressive deep generative models are ***directed and fully observed models***, where the outcome of the data completely depends on, the previous data points.
- ❖ ***PixelCNN*** by *Google DeepMind* was probably the first ***deep autoregressive*** model, that works on ***Image modeling***, and the inspiration/base for most of the other models.

Keywords:

❖ Readouts:

- ❖ are simple and non-adaptive functions designed such that the resulting hypothesis space is ***permutation invariant***, such as sum, mean, and max.
 - ❖ refers to functions that summarize the information of individual nodes to form more high-level information such as forming a subgraph/super-graph or obtaining the representations of the entire graph.
 - ❖ Readout is also called **pooling** and **graph coarsening** in some literature.
-

Keywords:

- ❖ **Teacher forcing algorithm:**
 - ❖ Teacher forcing is an algorithm for training the weights of recurrent neural networks (RNNs).
 - ❖ It involves feeding observed sequence values (i.e. ground-truth samples) back into the RNN after each step, thus forcing the RNN to stay close to the ground-truth sequence
 - ❖ The term "teacher forcing" was introduced in 1989 by **Ronald J. Williams and David Zipser**, who reported that, the technique was already being "frequently used in dynamical supervised learning tasks" around that time.

Keywords:

- ❖ **Teacher forcing algorithm:**
 - ❖ The term "teacher forcing" can be motivated by comparing the RNN to a human student taking a multi-part exam where the answer to each part (*for example a mathematical calculation*) depends on the answer to the preceding part.
 - ❖ In this analogy, rather than grading every answer in the end, with the risk that the student fails every single part, even though, they only made a mistake in the first one, a teacher records the score for each individual part and then tells the student the correct answer, to be used in the next part.

Teacher forcing algorithm

- ❖ It is a network training method critical to the development of deep learning language models used in machine translation, text summarization, and image captioning, among many other applications.
- ❖ *The approach was originally described and developed as an alternative technique to backpropagation through time for training a recurrent neural network.*
- ❖ Teacher forcing works by using the actual or expected output from the training dataset at the current time step $y(t)$ as input in the next time step $X(t+1)$, rather than the output generated by the network.

Teacher forcing algorithm

- ❖ The model's parameters are updated based on the loss between its generated output and the true target output at each step. This helps the model learn to create sequences closer to the desired target.
- ❖ It has several **advantages** during training:
 - ❖ Faster Convergence:..
 - ❖ Reduced Error Propagation:
 - ❖ Stable Training
 - ❖ Explicit Supervision
 - ❖ Controlled Exploration
 - ❖ Easier Evaluation

Teacher forcing algorithm

- ❖ It has several **advantages** during training:
 - ❖ **Faster Convergence:**
 - ❖ During training, it provides the model with accurate and consistent guidance using the true target sequence as input. This helps the model converge to a solution more rapidly than training without teacher forcing.
 - ❖ The direct supervision provided by teacher forcing reduces the need for the model to explore a wide range of possibilities, making it more efficient in learning the correct sequence generation.



Teacher forcing algorithm

- ❖ It has several **advantages** during training:
 - ❖ **Reduced Error Propagation:**
 - ❖ One common issue in sequence generation is error propagation. Mistakes made in earlier steps of sequence generation can accumulate and affect the quality of subsequent predictions.
 - ❖ Teacher forcing helps mitigate this problem by ensuring that each step of the model's training receives accurate inputs, allowing it to correct errors and make better predictions.

Teacher forcing algorithm

- ❖ It has several **advantages** during training:
 - ❖ **Stable Training:**
 - ❖ Training sequence-to-sequence models can be challenging due to the intricacies of sequence data. Teacher forcing provides a stable and consistent training process by maintaining a clear and well-defined input-output relationship.
 - ❖ Compared to models that generate their inputs, the reduced variability in training data contributes to more stable convergence and better training outcomes.

Teacher forcing algorithm

- ❖ It has several **advantages** during training:
 - ❖ **Explicit Supervision:**
 - ❖ It provides explicit, ground truth supervision at every time step during training. This means that the model can access the “correct” answer, which can be particularly valuable for tasks where the target sequence follows a specific structure or pattern.
 - ❖ For tasks like machine translation or speech recognition, teacher forcing can ensure that the model effectively learns the target language’s grammar, vocabulary, and structural nuances.

Teacher forcing algorithm

- ❖ It has several **advantages** during training:
 - ❖ **Controlled Exploration:**
 - ❖ Teacher forcing enables the model to explore the sequence space in a controlled manner. It ensures that the model encounters the correct examples early in training, allowing it to form a solid foundation for generating sequences.
 - ❖ By guiding the model's exploration, teacher forcing helps the model generalize well to unseen examples and produce more coherent and contextually accurate sequences.

Teacher forcing algorithm

- ❖ It has several **advantages** during training:
 - ❖ **Easier Evaluation:**
 - ❖ Since teacher forcing uses the true target sequences during training, evaluating the model's performance is relatively easy. You can directly compare the generated sequences to the target sequences for quantitative assessment.
 - ❖ This simplicity in evaluation makes monitoring the model's progress and making necessary adjustments during training convenient.



Teacher forcing algorithm

❖ It has some **limitations** also:

❖ **Exposure Bias:**

❖ The model may become overly reliant on the teacher-provided signals and struggle to generate sequences independently during inference or real-world use (*i.e.*, *without teacher forcing*).

❖ This leads to a disconnect between training and deployment performance.



Teacher forcing algorithm

-
- ❖ It has some **limitations** also:
 - ❖ **Mismatch Between Training and Inference:**
 - ❖ If the model relies heavily on teacher forcing during training, it may not perform as well during inference, when it has to generate sequences step by step without access to the true target.



Teacher forcing algorithm

- ❖ It has some **limitations** also:
 - ❖ **Lack of Real-World Noise:**
 - ❖ Teacher forcing relies on pristine, error-free target sequences for training. In real-world scenarios, data often contains noise, errors, and variations.
 - ❖ Models trained with teacher forcing may not learn to handle noisy or imperfect input data effectively. This makes them less robust in practical applications where input data may deviate from the ideal.



Teacher forcing algorithm

-
- ❖ It has some **limitations** also:
 - ❖ **Limited Exploration:**
 - ❖ Teacher forcing constrains the model's exploration by consistently guiding it with the true target sequence. This can limit the model's ability to discover novel or creative solutions.
 - ❖ It may not be the best choice in scenarios where creativity and diversity are essential, such as text generation for artistic purposes.



Teacher forcing algorithm

-
- ❖ It has some **limitations** also:
 - ❖ **Incomplete Training Data:**
 - ❖ In some cases, the true target sequences may be incomplete or truncated. For instance, the reference translation may not cover all possible valid translations in language translation.
 - ❖ The model may struggle to handle unseen variations or provide appropriate translations.



Teacher forcing algorithm

❖ It has some **limitations** also:

❖ **Resource-Intensive :**

- ❖ Training with teacher forcing often requires substantial training data, making it resource-intensive. It relies on a large set of paired input-output sequences.
- ❖ Generating or curating high-quality training data can be time-consuming and costly.



Teacher forcing algorithm

- ❖ Teacher forcing is a valuable training technique for sequence generation models. Still, it should be used judiciously, and potential issues like exposure bias should be considered when applying it.
- ❖ To mitigate the limitations of teacher forcing and strike a balance between training and inference performance, researchers have developed a technique known as **“scheduled sampling.”**

Teacher forcing algorithm

❖ Scheduled Sampling:

- ❖ Scheduled sampling is a training strategy that gradually transitions from using teacher forcing to using the model's predictions as input during training. Instead of providing the true target sequence as input at every time step, the transition allows the model to adapt to generating sequences independently.
- ❖ A schedule controls the transition from teacher forcing to the model's predictions. Early in training, the schedule favours teacher forcing, ensuring the model receives significant guidance. As training progresses, the schedule gradually shifts towards using the model's predictions, reducing the reliance on the true target.

Teacher forcing algorithm

❖ Applications of Teacher Forcing

- ❖ Natural Language Processing
- ❖ Machine Translation
- ❖ Text Generation:
- ❖ Speech Recognition
- ❖ Handwriting Recognition
- ❖ Image Captioning
- ❖ Dialogue Systems (in developing chatbots and conversational agents)
- ❖ Text Summarization
- ❖ Music Generation
- ❖ Time Series Forecasting
- ❖ Game Playing Agents
- ❖ Content Recommendation
- ❖ Video Captioning
- ❖ Autonomous Vehicles



GNN-based Autoregressive Model

GNN-based Autoregressive Model

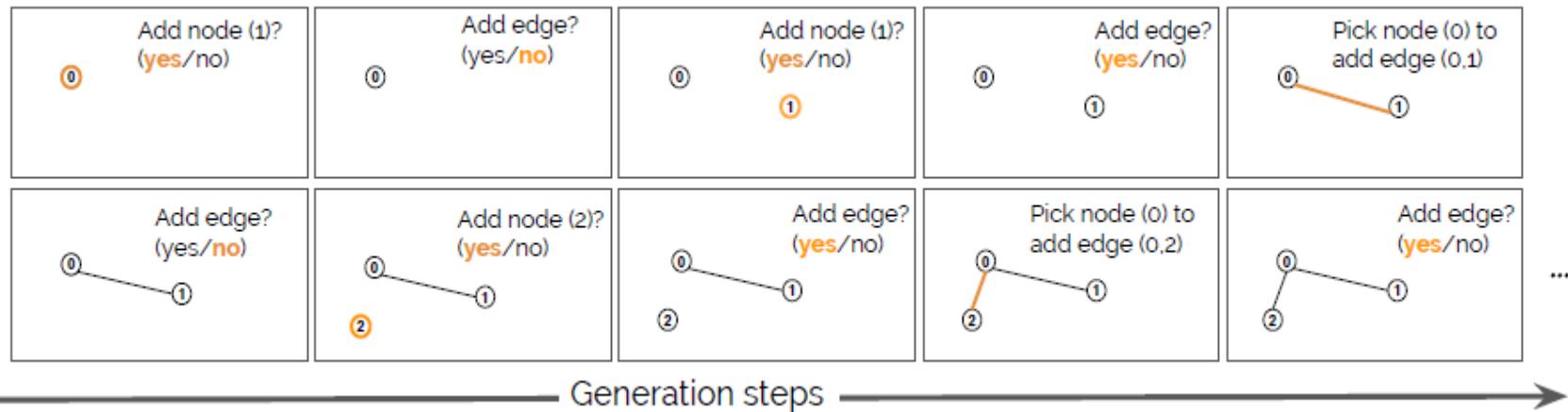


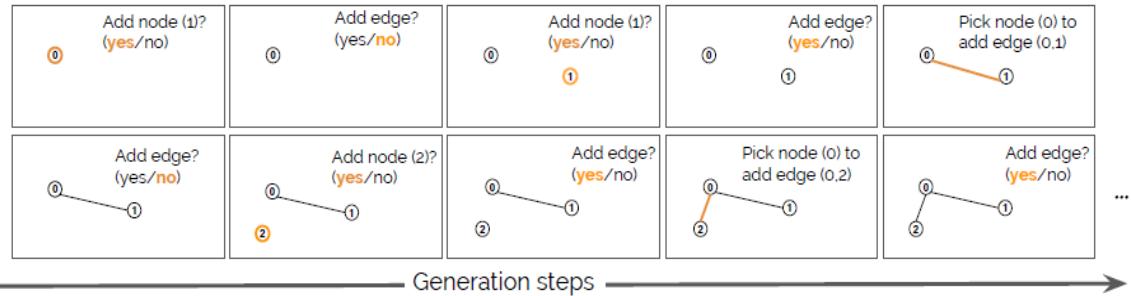
- ❖ The first GNN-based autoregressive model was proposed in (Li et al, 2018).
- ❖ This model can capture both, the structure and attributes, of the graph.
- ❖ Their approach uses graph neural networks to express probabilistic dependencies among a graph's nodes and edges, and can, in principle, learn distributions over any arbitrary graph.
- ❖ After a series of experiments their results show that once trained, the models can generate good quality samples of both synthetic graphs as well as real molecular graphs, both unconditionally and conditioned on data.

GNN-based Autoregressive Model

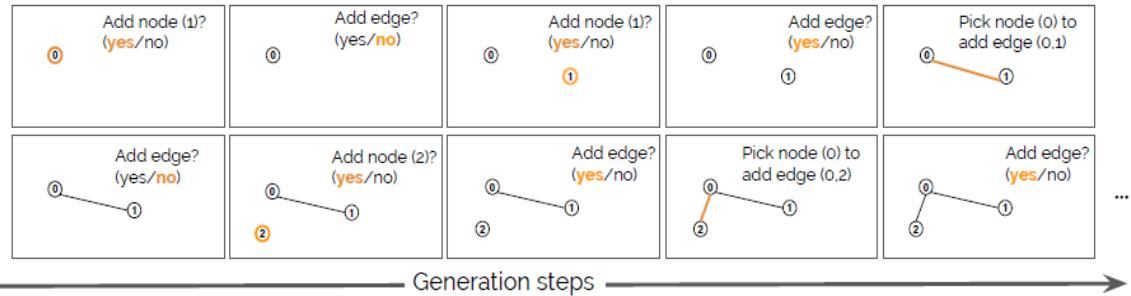


- ❖ The first GNN-based autoregressive model was proposed in (Li et al, 2018).
- ❖ The generative model of graphs uses a sequential process which generates one node at a time and connects each node to the existing partial graph by creating edges one by one.

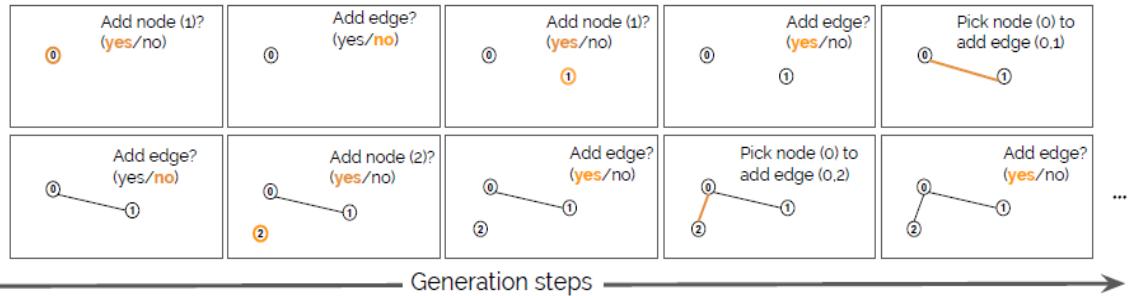




- ❖ Briefly, in this generative process, in each iteration we:
- (1) sample whether to add a new node of a particular type or terminate; if a node type is chosen,
- (2) we add a node of this type to the graph and
- (3) check if any further edges are needed to connect the new node to the existing graph; if yes
- (4) we select a node in the graph and add an edge connecting the new node to the selected node.
- (5) The algorithm goes back to step (3) and repeats until the model decides not to add another edge.
- (6) Finally, the algorithm goes back to step (1) to add subsequent nodes.



- ❖ There are many different ways to tweak this generation process.
- ❖ For example, edges can be made directional or typed by jointly modeling the node selection process with type and direction random variables.
- ❖ Additionally, constraints on certain structural aspects of graphs can be imposed such as forbidding self-loops or multiple edges between a pair of nodes.



- ❖ If node/edge labels are considered, we can further sample from a categorical distribution at each step to specify such labels.
- ❖ ***The key question of this class of methods is, how to build a probabilistic model so that, the current decision depends on, all previous historical choices.***

GNN-based Autoregressive Model



- ❖ The Adjacency matrix of the (generating) graph using the GNN based Autoregressive approach will be a ***lower triangular*** matrix L (*we will use L to denote the lower-triangular portion of the adjacency matrix A .*)
- ❖ The rows of this matrix is increased, as we add new node into the graph.
- ❖ The connectivity of that newly generated node, with the already generated graph, is decided based on some hyper parameters (*explained later*).

GNN-based Autoregressive Model



- ❖ Assuming we are working with simple graphs, \mathbf{A} and \mathbf{L} contain exactly the same information,
 - ❖ But it will be convenient to work with \mathbf{L} in the following equations.
- ❖ We will then use the notation $\mathbf{L}[v_1,:]$ to denote the row of \mathbf{L} corresponding to node v_1 , and we will assume that the rows of \mathbf{L} are indexed by nodes $v_1; \dots v_{|V|}$.
- ❖ Note that due to the lower-triangular nature of \mathbf{L} , we will have that $L[v_i, v_j] = 0 \forall j > i$, meaning that, we only need to be concerned with generating the first i entries for any row $\mathbf{L}[v_i,:]$, the rest can simply be padded with zeros.

GNN-based Autoregressive Model



- ❖ The autoregressive approach amounts to the following decomposition of the overall graph likelihood:

$$P(\mathbf{G}|\mathbf{z}) = \prod_{i=1}^{|\mathcal{V}|} P(\mathbf{L}[\mathbf{v}_{i\cdot} :] | \mathbf{L}[\mathbf{v}_{1\cdot} :], \dots, \mathbf{L}[\mathbf{v}_{i-1\cdot} :], \mathbf{z})$$

- ❖ In other words, when we generate row $\mathbf{L}[\mathbf{v}_{i\cdot} :]$, we conditioned on all the previous generated rows $\mathbf{L}[\mathbf{v}_{j\cdot} :]$ with $j < i$.

- ❖ Suppose at time step $t - 1$, we already generated a partial graph denoted as $\mathbf{G}^{t-1} = (\mathbf{V}^{t-1}, \mathbf{E}^{t-1})$.
- ❖ The corresponding adjacency matrix and node feature matrix are denoted as $(\mathbf{A}^{t-1}, \mathbf{X}^{t-1})$.
- ❖ At time step t , the model needs to decide:
 - 1) whether we add a new node or we stop the generation (denoting the probability as $p_{AddNode}$);
 - 2) whether we add an edge that links any existing node to the newly added node (denoting the probability as $p_{AddEdge}$);
 - 3) choose a existing node to link to the newly added node (denoting the probability as p_{Nodes}).
- ❖ For simplicity, we define $p_{AddNode}$ to be a **Bernoulli distribution**.
- ❖ We can extend it to a categorical one, if node labels types are considered.
- ❖ $p_{AddEdge}$ is yet another Bernoulli distribution whereas p_{Nodes} is a categorical distribution with size $|\mathbf{V}^{t-1}|$ (*i.e., its size will change as the generation goes on*).

GNN-based Autoregressive Model.



- ❖ To construct the above probabilities of decisions, we first build a ***message passing graph neural network*** to learn node representations.
- ❖ The input to the GNN at time step $t - 1$ is (A^{t-1}, H^{t-1}) where H^{t-1} is the node representation (one row corresponds to a node).
 - ❖ Note that, at time 0, since the graph is empty, we need to generate a new node to start. The generation probability p_{AddNode} will be output by the model based on some randomly initialized hidden state.
 - ❖ If we model the node labels/types or node features, we can also use them as additional node representations, e.g., *concatenating them with rows of H^{t-1}* .

GNN-based Autoregressive Model



- ❖ The one-step message passing is shown as below:

$$\begin{aligned} m_{ij} &= f_{msg}(h_i^{t-1}, h_j^{t-1}), & \forall (i, j) \in E \\ \bar{m}_i &= f_{agg}(\{m_{ij} | \forall j \in \Omega_i\}, & \forall i \in V \\ \tilde{h}_i^{t-1} &= f_{Update}(h_i^{t-1}, \bar{m}_i), & \forall i \in V \end{aligned}$$

- ❖ Here f_{msg} , f_{agg} & f_{update} are the message function, the aggregation function, and the node update function respectively.
- ❖ For the message function, we often instantiate f_{msg} as an MLP
 - ❖ if edge features are considered, one can incorporate them as input to f_{msg} .
 - ❖ f_{agg} could simply be an average or summation operator.
 - ❖ Typical examples of f_{Update} include gated recurrent units (GRUs) and long-short term memory (LSTM).

GNN-based Autoregressive Model



- ❖ The one-step message passing is shown as below:

$$\begin{aligned} m_{ij} &= f_{msg}(h_i^{t-1}, h_j^{t-1}), & \forall (i, j) \in E \\ \bar{m}_i &= f_{agg}(\{m_{ij} | \forall j \in \Omega_i\}, & \forall i \in V \\ \tilde{h}_i^{t-1} &= f_{Update}(h_i^{t-1}, \bar{m}_i), & \forall i \in V \end{aligned}$$

- ❖ Ω_i denotes the set of neighboring nodes of the node i
- ❖ h_i^{t-1} is the input node representation at time step $t - 1$
- ❖ \tilde{h}_i^{t-1} is the updated node representation which serves as the input node representation for the next message passing step.

The above message passing process is typically executed for a fixed number of steps, which is tuned as a hyperparameter.

GNN-based Autoregressive Model



Output Probabilities:

$$h_{G^{t-1}} = f_{readout}(H^t)$$

$$p_{AddNode} = \text{Bernoulli}(\sigma(MLP_{AddNode}(h_{G^{t-1}})))$$

$$p_{AddEdge} = \text{Bernoulli}\left(\sigma\left(MLP_{AddEdge}(h_{G^{t-1}}, h_v)\right)\right)$$

$$s_{uv} = MLP_{Nodes}(h_u^t, h_v) \quad \forall u \in V^{t-1}$$

$$p_{Node} = \text{Categorical}(\text{Softmax}(s)).$$

GNN-based Autoregressive Model

$$\begin{aligned} h_{G^{t-1}} &= f_{readout}(H^t) \\ p_{AddNode} &= \text{Bernoulli}(\sigma(MLP_{AddNode}(h_{G^{t-1}}))) \\ p_{AddEdge} &= \text{Bernoulli}\left(\sigma\left(MLP_{AddEdge}(h_{G^{t-1}}, h_v)\right)\right) \\ s_{uv} &= MLP_{Nodes}(h_u^t, h_v) \quad \forall u \in V^{t-1} \\ p_{Node} &= \text{Categorical}(\text{Softmax}(s)). \end{aligned}$$

- ❖ Here we first summarize the graph representation $\mathbf{h}_{G^{t-1}}$ (a vector) by reading out from the node representation H^t via $f_{ReadOut}$, (which could be an average operator or an attention-based one).
- ❖ Based on $\mathbf{h}_{G^{t-1}}$, we predict the probability of adding a new node $p_{AddNode}$ where σ is the sigmoid function.
- ❖ If we decide to add a new node by sampling $\mathbf{1}$ from the Bernoulli distribution $p_{AddNode}$, we denote the new node as v .
- ❖ We can initialize its representation \mathbf{h}_v as random features by sampling either from $\mathcal{N}(\mathbf{0}, \mathbf{I})$ or learned distribution over node type/label if provided.

GNN-based Autoregressive Model

$$\begin{aligned} h_{G^{t-1}} &= f_{readout}(H^t) \\ p_{AddNode} &= \text{Bernoulli}(\sigma(MLP_{AddNode}(h_{G^{t-1}}))) \\ p_{AddEdge} &= \text{Bernoulli}\left(\sigma\left(MLP_{AddEdge}(h_{G^{t-1}}, h_v)\right)\right) \\ s_{uv} &= MLP_{Nodes}(h_u^t, h_v) \quad \forall u \in V^{t-1} \\ p_{Node} &= \text{Categorical}(\text{Softmax}(s)). \end{aligned}$$

- ❖ Then we compute similarity scores between every existing node u in G^{t-1} and v as s_{uv}
- ❖ s is the concatenated vector of all **similarity scores**.
- ❖ Finally, we normalize the scores using softmax to form the categorical distribution from which we sample an existing node to obtain the new edge.
- ❖ By sampling from all these probabilities, we could either stop the generation or obtain a new graph with a new node and/or a new edge.
- ❖ We repeat this procedure by carrying on the node representations along with the generated graphs until the model generates a stop signal from $p_{AddNode}$

GNN-based Autoregressive Model



- ❖ The overall model design is well motivated. It also achieves good empirical performances in generating small graphs like molecules (e.g., *less than 40 nodes*).
- ❖ However, since the model only generates at most one new node and one new edge per step, the total number of generation steps scales with the number of nodes quadratically for dense graphs.
- ❖ **It is thus inefficient to generate moderately large graphs** (e.g., *with a few hundreds of nodes*).



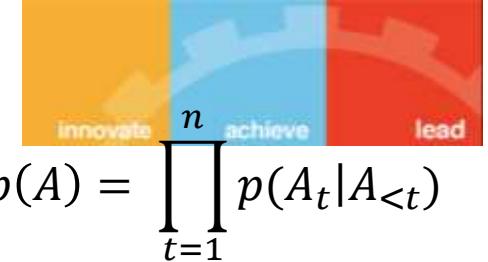
Graph Recurrent Neural Networks (GraphRNN)

Graph Recurrent Neural Networks (GraphRNN)

- ❖ Graph Recurrent Neural Networks (**GraphRNN**) (You et al, 2018b) is another **deep autoregressive** model which has a similar sequential decision-making formulation and leverages RNNs to construct the conditional probabilities.
- ❖ We again rely on the adjacency matrix representation of a graph, i.e $G \equiv \{PAP^T | P \in \pi_G\}$
- ❖ GraphRNN starts with an autoregressive decomposition of the probability of an adjacency matrix as follows:

$$p(A) = \prod_{t=1}^n p(A_t | A_{<t})$$

Graph Recurrent Neural Networks (GraphRNN)



- ❖ where A_t is the t -th column of the adjacency matrix A and
- ❖ $A_{<t}$ is a matrix formed by columns A_1, A_2, \dots, A_{t-1} .
- ❖ n is the maximum number of nodes.
- ❖ If a graph has less than n nodes, we pad dummy nodes.
- ❖ Then we can construct the conditional probability as an *edge-independent Bernoulli distribution*.

$$p(A_t | A_{<t}) = \text{Bernoulli}(\Theta_t) = \prod_{i=1}^n \Theta_{t,i}^{1[A_{i,t}=1]} (1 - \Theta_{t,i})^{1[A_{i,t}=0]}$$
$$\Theta_t = f_{out}(h_t)$$
$$h_t = f_{trans}(h_{t-1}, A_{t-1})$$

Graph Recurrent Neural Networks (GraphRNN)



$$p(A_t | A_{<t}) = \text{Bernoulli}(\boldsymbol{\theta}_t) = \prod_{i=1}^n \boldsymbol{\theta}_{t,i}^{1[A_{i,t}=1]} (1 - \boldsymbol{\theta}_{t,i})^{1[A_{i,t}=0]}$$
$$\boldsymbol{\theta}_t = f_{\text{out}}(h_t)$$
$$h_t = f_{\text{trans}}(h_{t-1}, A_{t-1})$$

- ❖ $\boldsymbol{\theta}_t$ is a size-n vector of Bernoulli parameters.
- ❖ $\boldsymbol{\theta}_{t,i}$ denotes its i^{th} element
- ❖ $A_{i,t}$ denotes the i^{th} element of the column vector A_t .
- ❖ f_{out} could be an MLP which takes the hidden state h_t as input and outputs $\boldsymbol{\theta}_t$.
- ❖ f_{trans} is the RNN cell function which takes the $(t-1)^{\text{th}}$ column of the adjacency matrix A_{t-1} and the hidden state h_{t-1} as input and outputs the current hidden state h_t .
- ❖ We can use an LSTM or GRU as the RNN cell function.

Note that the conditioning on $A_{<t}$ is implemented via the recurrent use of the hidden state in an RNN. The hidden state can be initialized as zeros or randomly sampled from a standard normal distribution.

References

- William L. Hamilton. (2020). Graph Representation Learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, Vol. 14, No. 3
 - Graph Neural Networks: Foundations, Frontiers, and Applications, Editors Lingfei Wu, Peng Cui, Jian Pei, Liang Zhao, Springer, 2022.: DOI
<https://doi.org/10.1007/978-981-16-6054-2>
-



BITS Pilani
Pilani Campus



Thank You