Graph Neural Networks for Node Classification and Link Prediction

MTD 701 - MSc Thesis Report

Debargha Ghosh
Entry No. 2022MAS7143

Project Supervisor : Dr. BS Panda

Department of Mathematics
Indian Institute of Technology Delhi

March 2024

1 Introduction

1.1 What is a Graph?

Before we discuss machine learning on graphs, it is necessary to give a bit more formal description of what exactly we mean by "graph data". Formally, a graph $G = (V, E)$ is defined by a set of nodes V and a set of edges E between these nodes. We denote an edge going from node $u \in V$ to node $v \in V$ as $(u, v) \in E$. In many cases, we will be concerned only with simple graphs, where there is at most one edge between each pair of nodes, no edges between a node and itself, and where the edges are all undirected, i.e., $(u, v) \in E \leftrightarrow (v, u) \in E$.
A convenient way to represent graphs is through an adjacency matrix $A \in \mathbb{R}^{|V|\times|V|}$. To represent a graph with an adjacency matrix, we order the nodes in the graph so that every node indexes a particular row and column in the adjacency matrix. We can then represent the presence of edges as entries in this matrix: $A[u, v] = 1$ if $(u, v) \in E$ and $A[u, v] = 0$ otherwise.

1.2 Feature Information

In many cases, we also have attribute or feature information associated with a graph (e.g., a profile picture associated with a user in a social network). Most often, these are node-level attributes that we represent using a real-valued matrix $X \in \mathbb{R}^{|V|\times m}$, where we assume that the ordering of the nodes is consistent with the ordering in the adjacency matrix. In heterogeneous graphs, we generally assume that each different type of node has its own distinct type of attributes.

2 Machine learning on graphs

Machine learning is inherently a problem-driven discipline. We seek to build models that can learn from data in order to solve particular tasks, and machine learning models are often categorized according to the type of task they seek to solve: Is it a supervised task? Is it an unsupervised task, where the goal is to infer patterns, such as clusters of points, in the data?

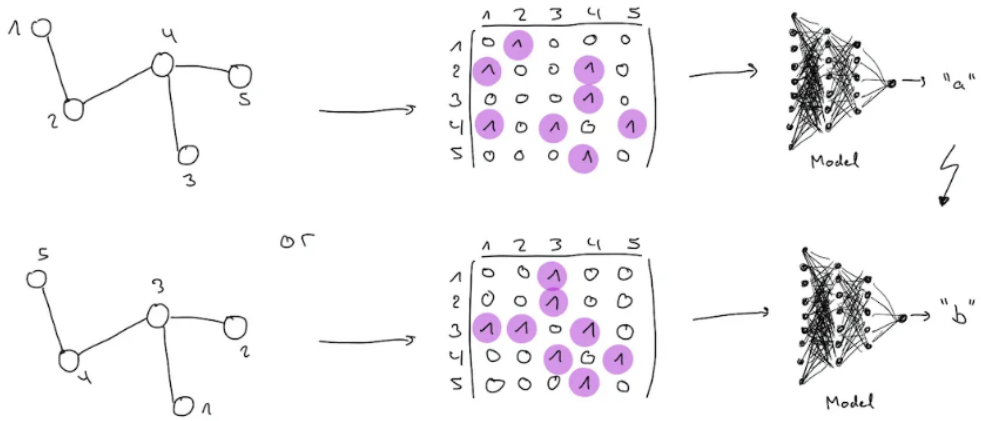
Machine learning with graphs is no different, but the usual categories of supervised and unsupervised are not necessarily the most informative or useful when it comes to graphs.

2.1 Node classification

Suppose we are given a large social network dataset with millions of users, but we know that a significant number of these users are actually bots. Identifying these bots could be important for many reasons: a company might not want to advertise to bots or bots may actually be in violation of the social network's terms of service. Manually examining every user to determine if they are a bot would be prohibitively expensive, so ideally we would like to have a model that could classify users as a bot (or not) given only a small number of manually labeled examples. This is a classic example of node classification, where the goal is to predict the label y_u—which could be a type, category, or attribute—associated with all the nodes $u \in V$, when we are only given the true labels on a training set of nodes $V_{\text{train}} \subset V$.
At first glance, node classification appears to be a straightforward variation of standard supervised classification, but there are in fact important differences.

The most important difference is that the nodes in a graph are not independent and identically distributed (i.i.d.). Usually, when we build supervised machine learning models we assume that each datapoint is statistically independent from all the other datapoints; otherwise, we might need to model the dependencies between all our input points. We also assume that the datapoints are identically distributed; otherwise, we have no way of guaranteeing that our model will generalize to new datapoints. Node classification completely breaks this i.i.d. assumption.

Also, graphs do not have a natural order or reference point. Thus there does not exist a top and a bottom or a left and right. This is different compared to the type of data we can explain with linear regression, CNNs or RNNs.
Any attempt to count through the network and making a matrix fails to generalize patterns in networks. Since adjacency matrices for a given graph are not unique. (permutation invariance and equivariance)

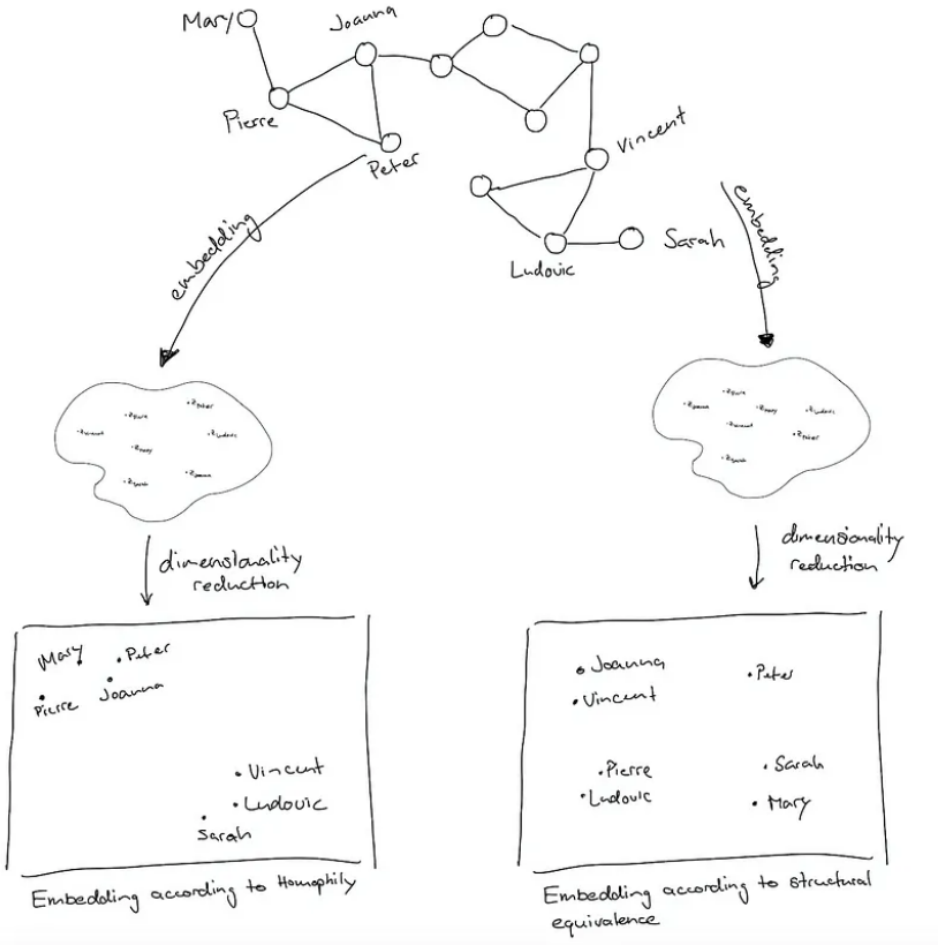


2.2 Node representation learning

Finding a vector for a node is possible with node representation learning. The result of such representation learning is a node embedding (and so does graph embedding, edge embedding exist).
But what rules should node embeddings obey in order to allow inference?

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.

In fact, the key insight behind many of the most successful node classification approaches is to explicitly leverage the connections between nodes. One particularly popular idea is to exploit **homophily**, which is the tendency for nodes. Beyond homophily there are also concepts such as **structural equivalence**, which is the idea that nodes with similar local neighborhood structures will have similar labels, as well as heterophily, which presumes that nodes will be preferentially connected to nodes with different labels. to share attributes with their neighbors in the graph.
There are well established algorithms like node2vec that have managed to have a generic notion of similarity to achieve node embeddings that could follow either more the concept of homophily or the concept of structural equivalence or a mix of both.


Mary
Joanna
Pierre
Peter
Vincent
Ludovic
Sarah
embedding
embedding
dimensionality reduction
dimensionality reduction
Mary
Peter
Pierre
Joanna
Vincent
Ludovic
Sarah
Joanna
Peter
Vincent
Pierre
Ludovic
Sarah
Mary
Embedding according to Homophily
Embedding according to structural equivalence

2.3 An Encoder-Decoder Perspective

In the encoder-decoder framework, we view the graph representation learning problem as involving two key operations. First, an encoder model maps each node in the graph into a low-dimensional vector or embedding. Next, a decoder model takes the low-dimensional node embeddings and uses them to reconstruct information about each node's neighborhood in the original graph.

2.3.1 Encoder

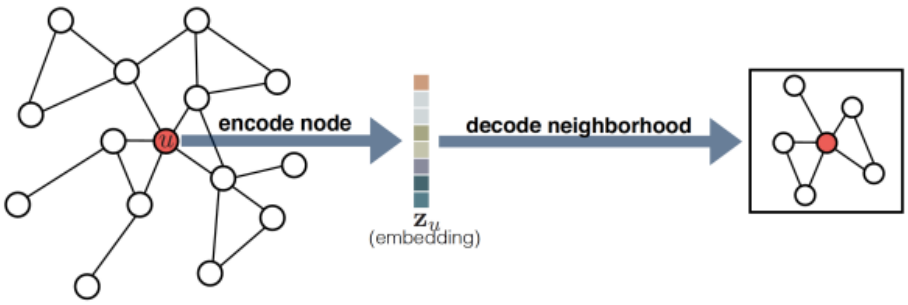
Formally, the encoder is a function that maps nodes $v \in V$ to vector embeddings $\mathbf{z}_v \in \mathbb{R}^d$ (where $\mathbf{z}_v$ corresponds to the embedding for node $v \in V$). In the simplest case, the encoder has the following signature:

$$\text{enc} : V \to \mathbb{R}^d,$$

meaning that the encoder takes node IDs as input to generate the node embeddings. In most work on node embeddings, the encoder relies on what we call the shallow embedding approach, where this encoder function is simply an embedding lookup based on the node ID. In other words, we have that

$$\text{enc}(v) = \mathbf{Z}[v],$$

where $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$ is a matrix containing the embedding vectors for all nodes and $\mathbf{Z}[v]$ denotes the row of $\mathbf{Z}$ corresponding to node v.



2.3.2 Decoder

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 $\mathbf{z}_u$ of a node u, the decoder might attempt to predict u's set of neighbors $N(u)$ or its row $A[u]$ in the graph adjacency matrix.

While many decoders are possible, the standard practice is to define pairwise decoders, which have the following signature:

$$\text{dec} : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+.$$

Pairwise decoders can be interpreted as predicting the relationship or similarity between pairs of nodes. For instance, a simple pairwise decoder could predict whether two nodes are neighbors in the graph.

Applying the pairwise decoder to a pair of embeddings $(\mathbf{z}_u, \mathbf{z}_v)$ results in the reconstruction of the relationship between nodes u and v. The goal is to optimize the encoder and decoder to minimize the reconstruction loss so that

$$\text{dec}(\text{enc}(u), \text{enc}(v)) = \text{dec}(\mathbf{z}_u, \mathbf{z}_v) \approx S[u, v].$$

Here, we assume that $S[u, v]$ is a graph-based similarity measure between nodes. For example, the simple reconstruction objective of predicting whether two nodes are neighbors would correspond to $S[u, v] = A[u, v]$.

2.3.3 Optimizing an Encoder-Decoder Model

To achieve the reconstruction objective, the standard practice is to minimize an empirical reconstruction loss $\mathcal{L}$ over a set of training node pairs D:

$$\mathcal{L} = \sum_{(u,v) \in D} \ell(\text{dec}(\mathbf{z}_u, \mathbf{z}_v), S[u, v]),$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is a loss function measuring the discrepancy between the decoded (i.e., estimated) similarity values $\text{dec}(\mathbf{z}_u, \mathbf{z}_v)$ and the true values $S[u, v]$.

Depending on the definition of the decoder (dec) and similarity function (S), the loss function ℓ might be a mean-squared error or even a classification loss, such as cross-entropy. Thus, the overall objective is to train the encoder and the decoder so that pairwise node relationships can be effectively reconstructed on the training set D.

2.4 Factorization methods

One way of viewing the encoder-decoder idea is from the perspective of matrix factorization. Indeed, the challenge of decoding local neighborhood structure from a node's embedding is closely related to reconstructing entries in the graph adjacency matrix. More generally, we can view this task as using matrix factorization to learn a low-dimensional approximation of a node-node similarity matrix $\mathbf{S}$, where $\mathbf{S}$ generalizes the adjacency matrix and captures some userdefined notion of node-node similarity.

2.4.1 Laplacian Eigenmaps

In this approach, we define the decoder based on the L_2-distance between the node embeddings:

$$\text{dec}(\mathbf{z}_u, \mathbf{z}_v) = \|\mathbf{z}_u - \mathbf{z}_v\|_2^2.$$

The loss function then weighs pairs of nodes according to their similarity in the graph:

$$\mathcal{L} = \sum_{(u,v)\in D} \text{dec}(\mathbf{z}_u, \mathbf{z}_v) \cdot S[u, v].$$

The intuition behind this approach is that the loss penalizes the model when very similar nodes have embeddings that are far apart.

If 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. In particular, if we assume the embeddings $\mathbf{z}_u$ are d-dimensional, then the optimal solution that minimizes loss is given by the d smallest eigenvectors of the Laplacian (excluding the eigenvector of all ones).

2.4.2 Inner Product Methods

Following the Laplacian eigenmaps technique, more recent work generally employs an inner-product-based decoder, defined as follows:

$$\text{dec}(\mathbf{z}_u, \mathbf{z}_v) = \mathbf{z}_u^\top \mathbf{z}_v.$$

Here, we assume that the similarity between two nodes—e.g., the overlap between their local neighborhoods—is proportional to the dot product of

their embeddings. These methods are referred to as matrix-factorization approaches, since their loss functions can be minimized using factorization algorithms, such as the singular value decomposition (SVD). Indeed, by stacking the node embeddings $\mathbf{z}_u \in \mathbb{R}^d$ into a matrix $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$, the reconstruction objective for these approaches can be written as

$$\mathcal{L} \approx \|\mathbf{Z}\mathbf{Z}^\top - \mathbf{S}\|_2^2,$$

which corresponds to a low-dimensional factorization of the node-node similarity matrix $\mathbf{S}$.

Intuitively, the goal of these methods is to learn embeddings for each node such that the inner product between the learned embedding vectors approximates some deterministic measure of node similarity.

2.5 Random Walk Embeddings

Here the aim is to use stochastic measures of neighborhood overlap. The key innovation in these approaches is that node embeddings are optimized so that two nodes have similar embeddings if they tend to co-occur on short random walks over the graph.

2.5.1 DeepWalk and node2vec

Mathematically, the goal is to learn embeddings so that the following (roughly) holds:

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

where $p_{G,T}(v|u)$ is the probability of visiting v on a length-T random walk starting at u, with T usually defined to be in the range $T \in \{2, \ldots, 10\}$. To train random walk embeddings, the general strategy is maximize the likelihood and thus equivalently minimize the following cross-entropy loss:

$$\mathcal{L} = \sum_{(u,v) \in D} -\log(\text{dec}(\mathbf{z}_u, \mathbf{z}_v)).$$

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

2.6 Limitations of shallow Embedding

1. The first issue is that shallow embedding methods do not share any parameters between nodes in the encoder, since the encoder directly optimizes a unique embedding vector for each node. This lack of parameter sharing is both statistically and computationally inefficient. From a statistical perspective, parameter sharing can improve the efficiency of learning and also act as a powerful form of regularization. From the computational perspective, the lack of parameter sharing means that the number of parameters in shallow embedding methods necessarily grows as $O(|V|)$, which can be intractable in massive graphs.

2. A second key issue with shallow embedding approaches is that they do not leverage node features in the encoder. Many graph datasets have rich feature information, which could potentially be informative in the encoding process.

3. Lastly—and perhaps most importantly—shallow embedding methods are inherently transductive. These methods can only generate embeddings for nodes that were present during the training phase. Generating embeddings for new nodes—which are observed after the training phase—is not possible unless additional optimizations are performed to learn the embeddings for these nodes. This restriction prevents shallow embedding methods from being used on inductive applications, which involve generalizing to unseen nodes after training.

To alleviate these limitations, shallow encoders can be replaced with more sophisticated encoders that depend more generally on the structure and attributes of the graph.

3 Graph Neural Networks

To define a deep neural network over general graphs, we need to define a new kind of deep learning architecture.

One reasonable idea for defining a deep neural network over graphs would be to simply use the adjacency matrix as input to a deep neural network. For example, 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):

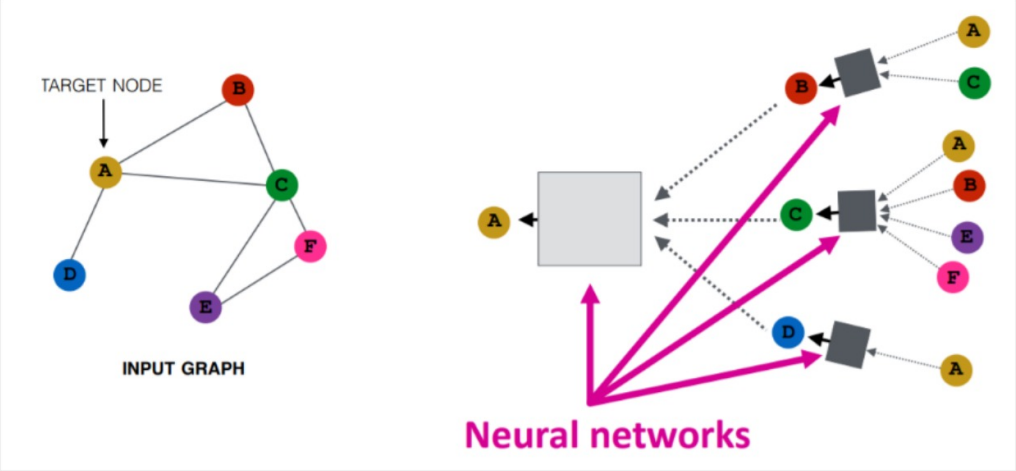
$$Z_G = \text{MLP}\left(A[1] \oplus A[2] \oplus \ldots \oplus A[|V|]\right)$$

where:

$A[i] \in \mathbb{R}^{|V|}$ denotes a row of the adjacency matrix

The issue with this approach is that it depends on the arbitrary ordering of nodes that we used in the adjacency matrix.In other words, such a model is not permutation invariant. So, We consider the following-

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



3.1 The Basic GNN

In order to translate the abstract GNN framework defined above into something we can implement, we must give concrete instantiations to these UPDATE and AGGREGATE functions. We begin here with the most basic GNN framework, which is a simplification of the original GNN models proposed by Merkwirth and Lengauer [2005] and Scarselli et al. [2009].
The basic GNN message passing is defined as

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

Where, $\mathbf{W}_{\text{self}}^{(k)}, \mathbf{W}_{\text{neigh}}^{(k)} \in \mathbb{R}^{d^{(k)} \times d^{(k-1)}}$

We can equivalently define the basic GNN through the UPDATE and AGGREGATE functions:

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

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

where we use,

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

In General the graph level equation becomes :- $\mathrm{H}^{(t)} = \sigma\left(AH^{(k-1)}W_{\text{neigh}}^{(k)} + H^{(k-1)}W_{\text{self}}^{(k)}\right)$

where, $H^{(k)} \in \mathbb{R}^{|V| \times d}$ and A is the graph adjacency matrix.

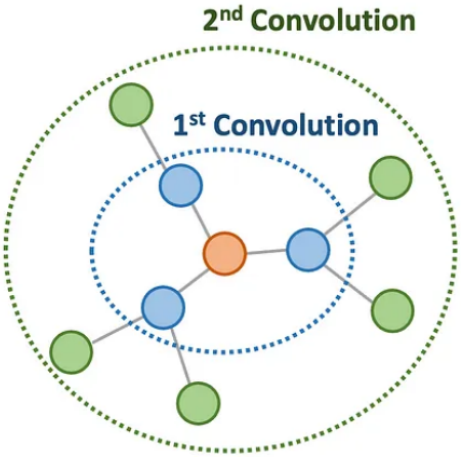
3.2 Graph Convolution Network

The convolution in GCN is the same as a convolution in convolutional neural networks. It multiplies neurons with weights (filters) to learn from data features.
It acts as sliding windows on whole images to learn features from neighboring cells. The filter uses weight sharing to learn various facial features in image recognition systems.

Now transfer the same functionality to Graph Convolutional networks where a model learns the features from neighboring nodes. The major difference between GCN and CNN is that it is developed to work on non-euclidean data structures where the order of nodes and edges can vary.
One of the most popular baseline graph neural network models—the graph convolutional network (GCN)—employs the symmetric-normalized aggrega-

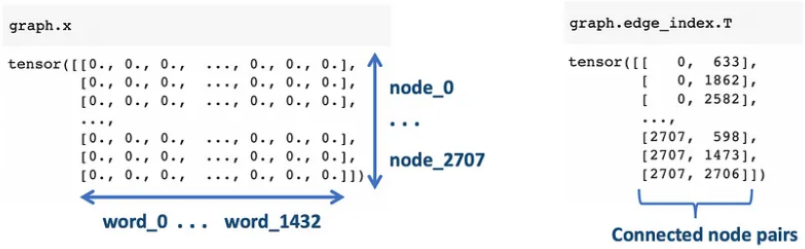
tion as well as the self-loop update approach. The GCN model thus defines the message passing function as

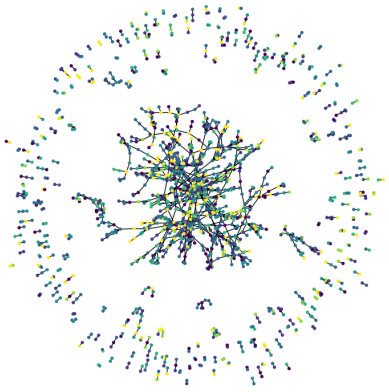
$$\mathrm{h}_u^{(k)} = \sigma\left(\left(W^{(k)}\right)\sum_{v\in N(u)\cup\{u\}}\frac{1}{\sqrt{|N(u)||N(v)|}}h_v\right)$$



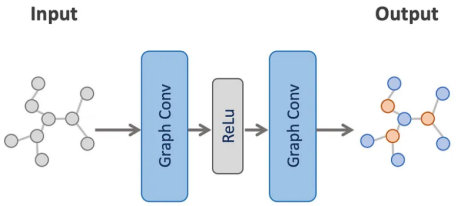
3.3 Node classification with GCN

Now, we will review GCN code implementations. Before we dive into them, let us get familiar with the dataset we are going to use. The Cora dataset is a paper citation network data that consists of 2,708 scientific publications. Each node in the graph represents each publication and a pair of nodes is connected with an edge if one paper cites the other.
The node features are 1433 word vectors indicating the absence (0) or the presence (1) of the words in each publication. The edges are represented in adjacency lists.



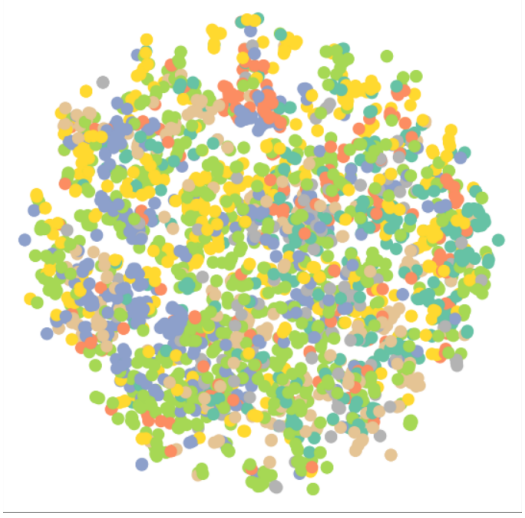


(a) CORA Dataset



(b) GCN Architechture

Initial Node embedding after 2 GCN Layers

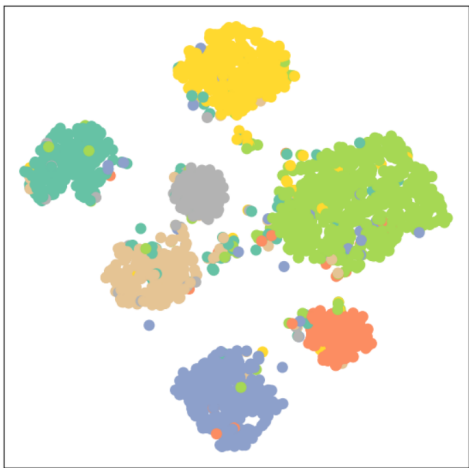


```python
def train_node_classifier(model, graph, optimizer, criterion, n_epochs=200):


    for epoch in range(1, n_epochs + 1):
        model.train()
        optimizer.zero_grad()
        out = model(graph)
        loss = criterion(out[graph.train_mask], graph.y[graph.train_mask])
        loss.backward()
        optimizer.step()

        pred = out.argmax(dim=1)
        acc = eval_node_classifier(model, graph, graph.val_mask)

        if epoch % 10 == 0:
            print(f'Epoch: {epoch:03d}, Train Loss: {loss:.3f}, Val Acc: {acc:.3f}')
            if model == mlp:
              val_acc_mlp.append(acc)
            else:
              val_acc_gcn.append(acc)

    return model
```

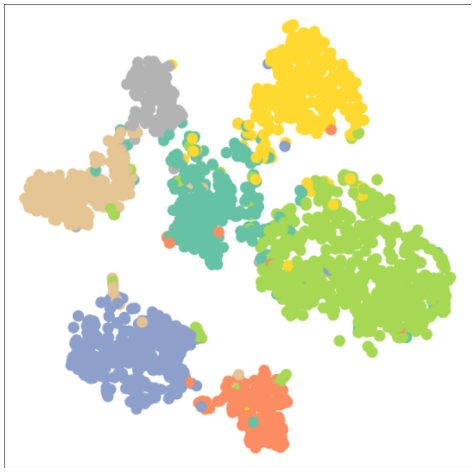
Figure 2: Train node function

```python
from torch_geometric.nn import GCNConv
import torch.nn.functional as F

val_acc_gcn = []
class GCN(torch.nn.Module):
    def __init__(self,hidden_channels):
        super().__init__()
        torch.manual_seed(1234567)
        self.conv1 = GCNConv(dataset.num_node_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index

        x = self.conv1(x, edge_index)
        x = F.relu(x)
        output = self.conv2(x, edge_index)

        return output


gcn = GCN(16).to(device)
optimizer_gcn = torch.optim.Adam(gcn.parameters(), lr=0.01, weight_decay=5e-4)
criterion = nn.CrossEntropyLoss()
gcn = train_node_classifier(gcn, graph, optimizer_gcn, criterion)

test_acc = eval_node_classifier(gcn, graph, graph.test_mask)
print(f'Test Acc: {test_acc:.3f}')
```

Figure 3: GCN Class

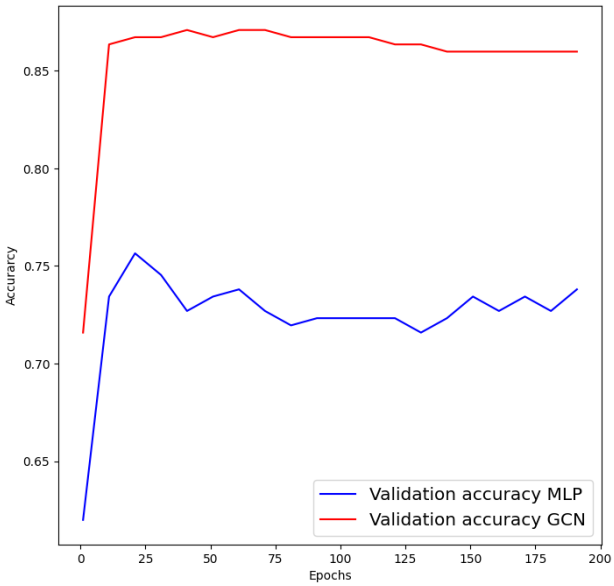


(a) MLP Embedding



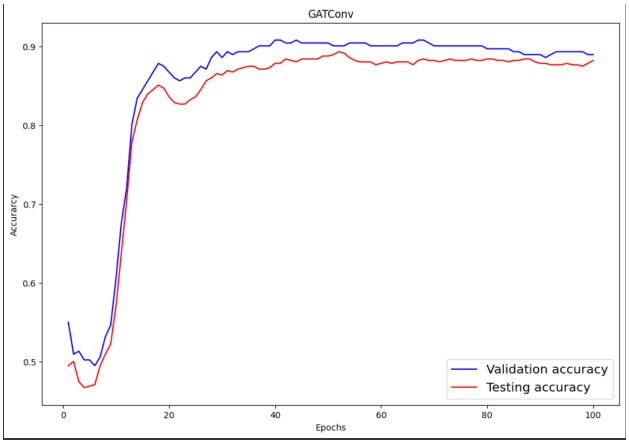
(b) GCN Embedding

3.4 Results using GCN after training



Performance on unseen data -
MLP Test set accuracy - 73 %
GCN Test Set accuracy - 87.5 %

3.5 Gated Network using Attention layers



GAT Accuracy with 8 attention heads : 0.89

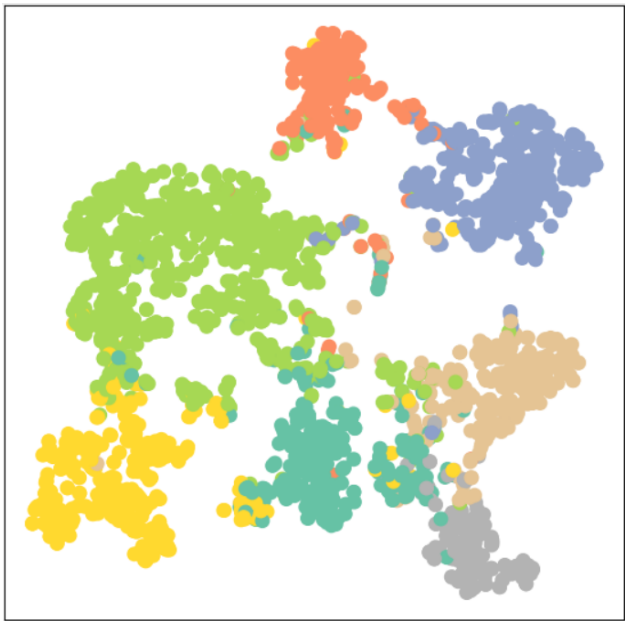


Figure 5: GAT Embedding

4 Graph Neural Networks: Link Prediction

4.1 Abstract

Link prediction is an important application of graph neural networks. By predicting missing or future links between pairs of nodes, link prediction is widely used in social networks, citation networks, biological networks, recommender systems, and security, etc. Traditional link prediction methods rely on heuristic node similarity scores, latent embeddings of nodes, or explicit node features. Graph neural network (GNN), as a powerful tool for jointly learning from graph structure and node/edge features, has gradually shown its advantages over traditional methods for link prediction.
We first introduce the link prediction problem and review traditional link prediction methods. Then, we introduce two popular GNN-based link prediction paradigms, node-based and subgraph-based approaches, and discuss their differences in link representation power.

4.2 Introduction

Link prediction is the problem of predicting the existence of a link between two nodes in a network (Liben-Nowell and Kleinberg, 2007). The term "link prediction" often refers to predicting links in homogeneous graphs, where nodes and links both only have a single type. This is the simplest setting and most link prediction works focus on this setting. Link prediction in bipartite user-item networks is referred to as matrix completion or recommender systems, where nodes have two types (user and item) and links can have multiple types corresponding to different ratings users can give to items. Link prediction in knowledge graphs is often referred to as knowledge graph completion, where each node is a distinct entity and links have multiple types corresponding to different relations between entities. In most cases, a link prediction algorithm designed for the homogeneous graph setting can be easily generalized to heterogeneous graphs (e.g., bipartite graphs and knowledge graphs) by considering heterogeneous node type and relation type information.

5 Traditional Link Prediction Methods

In this section, we review traditional link prediction methods. They can be categorized into three classes: heuristic methods, latent-feature methods, and contentbased methods.

5.1 Heuristic Methods

Heuristic methods use simple yet effective node similarity scores as the likelihood of links (Liben-Nowell and Kleinberg, 2007; Lu and Zhou, 2011). We use ¨ x and y to denote the source and target node between which to predict a link. We use γ (x) to denote the set of x's neighbors.

5.1.1 Local Heuristics

One simplest heuristic is called common neighbors (CN), which counts the number of neighbors two nodes share as a measurement of their likelihood of having a link:

$$f_{\mathrm{CN}}(x,y) = |\Gamma(x) \cap \Gamma(y)|.$$

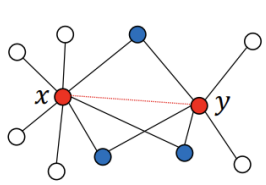
CN is widely used in social network friend recommendation. It assumes that the more common friends two people have, the more likely they themselves are also friends.

$$f_{\mathrm{Jaccard}}(x,y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}.$$

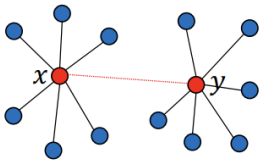
There is also a famous preferential attachment (PA) heuristic (Barabasi and Albert, 1999), which uses the product of node degrees to measure the link likelihood:

$$f_{\mathrm{PA}}(x,y) = |\Gamma(x)| \cdot |\Gamma(y)|.$$

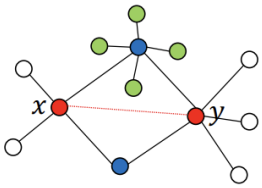
PA assumes x is more likely to connect to y if y has a high degree. For example, in citation networks, a new paper is more likely to cite those papers which already have a lot of citations.



common neighbors (CN):
$|\Gamma(x) \cap \Gamma(y)|$



preferential attachment (PA):
$|\Gamma(x)| \cdot |\Gamma(y)|$



Adamic-Adar (AA):
$\sum_{z \in \Gamma(x) \cap \Gamma(y)} \frac{1}{\log |\Gamma(z)|}$

5.1.2 Global Heuristics

There are also high-order heuristics which require knowing the entire network. Examples include Katz index.
Katz index uses a weighted sum of all the walks between x and y where a longer walk is discounted more:

$$f_{\text{Katz}}(x,y) = \sum_{l=1}^{\infty} \beta^l |\text{walks}^{\langle l \rangle}(x,y)|.$$

5.2 Latent Methods

The second class of traditional link prediction methods is called latent-feature methods. In some literature, they are also called latent-factor models or embedding methods. Latent-feature methods compute latent properties or representations of nodes, often obtained by factorizing a specific matrix derived from the network, such as the adjacency matrix and the Laplacian matrix.

5.2.1 Matrix Factorization

One most popular latent feature method is matrix factorization (Koren et al, 2009; Ahmed et al, 2013), which is originated from the recommender systems

literature. Matrix factorization factorizes the observed adjacency matrix A of the network into the product of a low-rank latent-embedding matrix Z and its transpose. That is, it approximately reconstructs the edge between i and j using their k-dimensional latent embedding $z_i and z_j$:

$$\hat{A}_{i,j} = \mathbf{z}_i^\top \mathbf{z}_j,$$

It then minimizes the mean-squared error between the reconstructed adjacency matrix and the true adjacency matrix over the observed edges to learn the latent embeddings:

$$\mathscr{L} = \frac{1}{|\mathscr{E}|} \sum_{(i,j)\in\mathscr{E}} (A_{i,j} - \hat{A}_{i,j})^2.$$

Finally, we can predict new links by the inner product between nodes' latent embeddings.

5.3 Content-Based Methods

Both heuristic methods and latent-feature methods face the cold-start problem. That is, when a new node joins the network, heuristic methods and latent-feature methods may not be able to predict its links accurately because it has no or only a few existing links with other nodes. In this case, content-based methods might help. Content-based methods leverage explicit content features associated with nodes for link prediction, which have wide applications in recommender systems (Lops et al, 2011). For example, in citation networks, word distributions can be used as content features for papers. In social networks, a user's profile, such as their demographic information and interests, can be used as their content features (however, their friendship information belongs to graph structure features because it is calculated from the graph structure). However, content-based methods usually have worse performance than heuristic and latent-feature methods due to not leveraging the graph structure.

6 GNN Methods for Link Prediction

In the last section, we have covered three types of traditional link prediction methods. In this section, we will talk about GNN methods for link prediction. GNN methods combine graph structure features and content features by learning them together in a unified way, leveraging the excellent graph representation learning ability of GNNs.
There are mainly two GNN-based link prediction paradigms, node-based and subgraph-based. Node-based methods aggregate the pairwise node representations learned by a GNN as the link representation. Subgraph-based methods extract a local subgraph around each link and use the subgraph representation learned by a GNN as the link representation.

6.1 Node-Based Methods

The most straightforward way of using GNNs for link prediction is to treat GNNs as inductive network embedding methods which learn node embeddings from local neighborhood, and then aggregate the pairwise node embeddings of GNNs to construct link representations. We call these methods node-based methods.

6.1.1 Graph AutoEncoder

The pioneering work of node-based methods is Graph AutoEncoder (GAE) (Kipf and Welling, 2016). Given the adjacency matrix A and node feature matrix X of a graph, GAE (Kipf and Welling, 2016) first uses a GCN (Kipf and Welling, 2017b) to compute a node representation z_i for each node i, and then uses $\sigma(\mathrm{z}_i, z_j)$ to predict link (i, j).

$$\hat{A}_{i,j} = \sigma(\mathbf{z}_i^\top \mathbf{z}_j), \text{ where } \mathbf{z}_i = Z_{i,:}, Z = \mathrm{GCN}(X, A)$$

where Z is the node representation (embedding) matrix output by the GCN with the ith row of Z being node i's representation z_i, $\hat{A}_{i,j}$ is the predicted probability for link (i, j), and σ is the sigmoid function. If X is not given, GAE can use the one-hot encoding matrix I instead. The model is trained to minimize the cross-entropy between the reconstructed adjacency matrix and the true adjacency matrix:

$$\mathscr{L} = \sum_{i \in \mathscr{V}, j \in \mathscr{V}} (-A_{i,j} \log \hat{A}_{i,j} - (1 - A_{i,j}) \log(1 - \hat{A}_{i,j})).$$

In practice, the loss of positive edges (Ai, j = 1) is up-weighted by k, where k is the ratio between negative edges (Ai, j = 0) and positive edges. The purpose is to balance the positive and negative edges' contribution to the loss. Otherwise, the loss might be dominated by negative edges due to the sparsity of practical networks.

6.1.2 Variational Graph AutoEncoder

The variational version of GAE is called VGAE, or Variational Graph AutoEncoder (Kipf and Welling, 2016). Rather than learning deterministic node embeddings z_i, VGAE uses two GCNs to learn the mean μ_i and variance σ^2 of z_i, respectively. VGAE assumes the adjacency matrix A is generated from the latent node embeddings Z through $p(A|Z)$, where Z follows a prior distribution $p(Z)$. Similar to GAE, VGAE uses an inner-product-based link reconstruction model as $p(A|Z)$:

$$p(A|Z) = \prod_{i \in \mathscr{V}} \prod_{j \in \mathscr{V}} p(A_{i,j}|\mathbf{z}_i, \mathbf{z}_j), \text{ where } p(A_{i,j} = 1|\mathbf{z}_i, \mathbf{z}_j) = \sigma(\mathbf{z}_i^\top \mathbf{z}_j).$$

And the prior distribution p(Z) takes a standard Normal distribution:

$$p(Z) = \prod_{i \in \mathscr{V}} p(\mathbf{z}_i) = \prod_{i \in \mathscr{V}} \mathscr{N}(\mathbf{z}_i|0, I).$$

Thus, given the adjacency matrix A and node feature matrix X, VGAE uses graph neural networks to approximate the posterior distribution of the node embedding matrix Z:

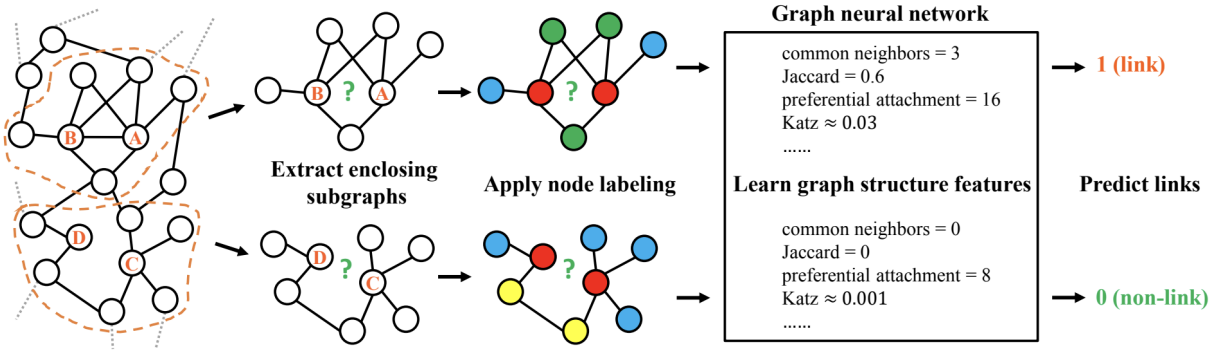
$$q(Z|X,A) = \prod_{i \in \mathscr{V}} q(\mathbf{z}_i|X,A), \text{ where } q(\mathbf{z}_i|X,A) = \mathscr{N}(\mathbf{z}_i|\boldsymbol{\mu}_i, \text{diag}(\boldsymbol{\sigma}_i^2)).$$

6.2 Subgraph-Based Methods

Subgraph-based methods extract a local subgraph around each target link and learn a subgraph representation through a GNN for link prediction.

6.2.1 The SEAL Framework

The pioneering work of subgraph-based methods is SEAL (Zhang and Chen, 2018b). SEAL first extracts an enclosing subgraph for each target link to predict, and then applies a graph-level GNN (with pooling) to classify whether the subgraph corresponds to link existence. The enclosing subgraph around a node set is defined as follows.



Definition : (Enclosing subgraph) For a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, given a set of nodes $S \subseteq \mathcal{V}$, the h-hop enclosing subgraph for S is the subgraph $\mathcal{G}_S^h$ induced from $\mathcal{G}$ by the set of nodes $\cup_{j \in S}\{i \mid d(i,j) \leq h\}$, where $d(i,j)$ is the shortest path distance between nodes i and j.

The motivation for extracting an enclosing subgraph for each link should be that SEAL aims to automatically learn graph structure features from the network. Observing that all first-order heuristics can be computed from the 1-hop enclosing subgraph around the target link and all second-order heuristics can be computed from the 2-hop enclosing subgraph around the target link, SEAL aims to use a GNN to learn general graph structure features (supervised heuristics) from the extracted hhop enclosing subgraphs instead of using predefined heuristics. After extracting the enclosing subgraph $G^h x, y$,

the next step is node labeling. SEAL applies a Double Radius Node Labeling (DRNL) to give an integer label to each node in the subgraph as its additional feature. The purpose is to use different labels to differentiate nodes of different roles in the enclosing subgraph. For instance, the center nodes x and y are the target nodes between which the target link is located, thus they are different from the rest nodes and should be distinguished. Similarly, nodes at different hops w.r.t. x and y may have different structural importance to the link existence, thus can also be assigned different labels.

DRNL works as follows: First, assign label 1 to x and y. Then, for any node i with radius $(d(i,x), d(i,y)) = (1,1)$, assign label 2. Nodes with radius $(1,2)$ or $(2,1)$ get label 3. Nodes with radius $(1,3)$ or $(3,1)$ get 4. Nodes with $(2,2)$ get 5. Nodes with $(1,4)$ or $(4,1)$ get 6. Nodes with $(2,3)$ or $(3,2)$ get 7. And so on. In other words, DRNL iteratively assigns larger labels to nodes with a larger radius with respect to the two center nodes.

DRNL satisfies the following criteria: 1) The two target nodes x and y always have the distinct label "1" so that they can be distinguished from the context nodes. 2) Nodes i and j have the same label if and only if their "double radius" are the same, i.e., i and j have the same distances to (x,y). This way, nodes of the same relative positions within the subgraph (described by the double radius $(d(i,x), d(i,y))$) always have the same label. DRNL has a closed-form solution for directly mapping $(d(i,x), d(i,y))$ to labels:

$$l(i) = 1 + \min(d_x, d_y) + (d/2)[(d/2) + (d\%2) - 1]$$

For nodes with $d(i,x) = \infty$ or $d(i,y) = \infty$, DRNL gives them a null label 0. After obtaining the DRNL labels, SEAL transforms them into one-hot encoding vectors or feeds them to an embedding layer to get their embeddings. These new feature vectors are concatenated with the original node content features (if any) to form the new node features.
Finally, SEAL feeds these enclosing subgraphs as well as their new node feature vectors into a graph-level GNN, to learn a graph classification function. The groundtruth of each subgraph is whether the two center nodes really have a link. To train this GNN, SEAL randomly samples N existing links from the network as positive training links, and samples an equal number of unobserved links (random node pairs) as negative training links.

7 Link prediction on large datasets

Here, we have used implementation of the GraphSAGE algorithm to build a model that predicts citation links in the Cora dataset (see below). The problem is treated as a supervised link prediction problem on a homogeneous citation network with nodes representing papers (with attributes such as binary keyword indicators and categorical subject) and links corresponding to paper-paper citations.
To address this problem, we build a model with the following architecture. First we build a two-layer GraphSAGE model that takes labeled node pairs (citing-paper $\rightarrow$ cited-paper) corresponding to possible citation links, and outputs a pair of node embeddings for the citing-paper and cited-paper nodes of the pair. These embeddings are then fed into a link classification layer, which first applies a binary operator to those node embeddings (e.g., concatenating them) to construct the embedding of the potential link. Thus obtained link embeddings are passed through the dense link classification layer to obtain link predictions - probability for these candidate links to actually exist in the network. The entire model is trained end-to-end by minimizing the loss function of choice (e.g., binary cross-entropy between predicted link probabilities and true link labels, with true/false citation links having labels 1/0) using stochastic gradient descent (SGD) updates of the model parameters, with minibatches of 'training' links fed into the model.

```
StellarGraph: Undirected multigraph
 Nodes: 2708, Edges: 5429

 Node types:
  paper: [2708]
    Features: float32 vector, length 1440
    Edge types: paper-cites->paper

 Edge types:
    paper-cites->paper: [5429]
```

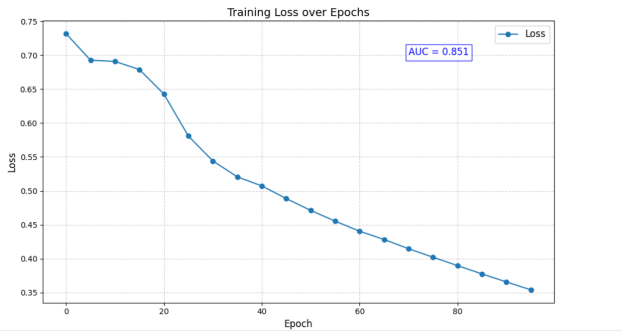
1. Treat the edges in the graph as positive examples.

2. Sample a number of non-existent edges (i.e. node pairs with no edges between them) as negative examples.

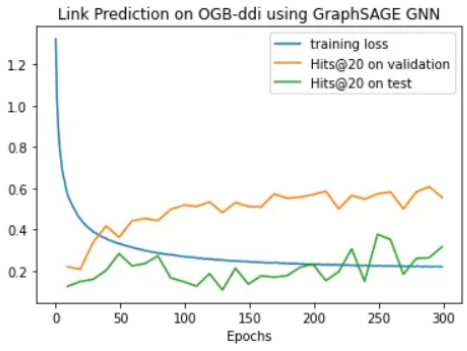
3. Divide the positive examples and negative examples into a training set and a test set.

4. Evaluate the model with any binary classification metric such as Area Under Curve (AUC)

Results :



Now We'll be exploring the inductive power of GNNs on online link prediction by using the **ogb-ddi (drug-drug interaction)** dataset. Specifically, we'll demonstrate GraphSAGE's ability to predict new links. The dataset contains an unweighted, undirected graph with 4,267 nodes and 1,334,889 edges representing the drug-drug interaction network. The nodes of the graph represent different drugs.



The metric here works as follows: we rank each positive edge in the test set against 3,000,000 randomly-sampled negative edges, and count the ratio of

positive edges that are ranked at K-th place or above. Hits@K = Fraction of correct links in the top K links (with respect to their scores)

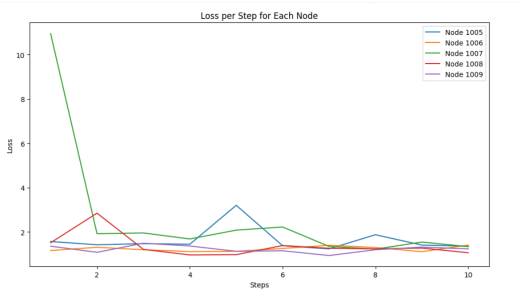
7.1 Online Learning

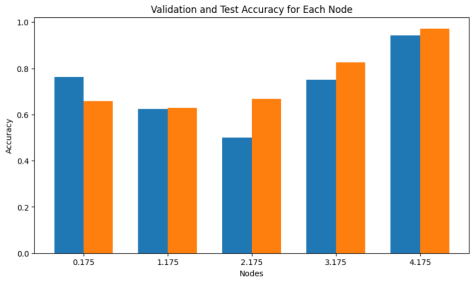
We have treated the online learning problem in the following phases-

- **Preprocessing**
  - Initialize the graph with existing nodes and edges.
  - Prepare the dataset by generating positive and negative edges for training.
- **Training for New Node**
  - For each new node:
    - Update node embeddings using existing graph structure.
    - Predict link probabilities for positive edges.
    - Generate and predict link probabilities for negative edges.
    - Compute loss from positive and negative edge predictions.
    - Update model parameters using the optimizer.
- **Model Evaluation**
  - For each evaluation edge:
    - Predict link probabilities for positive edges and evaluate true positives and false negatives.
    - Predict link probabilities for negative edges and evaluate false positives and true negatives.

Here are our results with the following initial set of parameters:

- Initial size/nodes: init_size = 1000
- Number of online nodes: num_online = 10
- ratio of nodes used for message passing: split_train_msg = 0.4
- ratio of nodes used for training: split_train_sp = 0.4
- ratio of nodes used for validation: split_val = 0.1





Testing over different combinations of hyperparameters,we found that richer prior knowledge, whether in terms of the number of initial nodes or the number of edges used for training, leads to better performance for the GNN.

8 References

1. Graph Representation Learning By William L. Hamilton
2. Stanford-cs224w
3. **Graph Attention Networks**: Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, Yoshua Bengio
   Paper
4. **Link Prediction Based on Graph Neural Networks**: Muhan Zhang, Yixin Chen
   Paper