Mathematics for Intelligence Systems – 6

Graph Neural Network for Node Classification

Team 3

Adhithan P	CB.EN.U4AIE19003
Anuvarshini S P	CB.EN.U4AIE19011
Kabilan N	CB.EN.U4AIE19033
Sivamaran M A C	CB.EN.U4AIE19061

Introduction

Graphs are a ubiquitous data structure and a universal language for describing complex systems. It's a collection of nodes (objects or entities or states) along with the set of interactions (edges) between pairs of nodes. This can be used to represent a social network, where nodes represent the peoples and edges represent the relationship between the individuals. In the biological domain, proteins represented as graphs and biological interaction between the proteins are defined by the weighted edges; this can also be used to find the connections between terminals in a telecommunications network and many other different types of datas can be represented as graphs like event graphs, computer networks, disease pathway, food webs, underground networks, economic networks, internet graph, 3D meshes etc,. Every complex domain that has rich relational structure can be represented as a relational graph

Graph

A graph can be represented as $G = (V, \varepsilon)$. It is defined as a set of nodes V and a set of edge ε between these nodes. Each edge in the graph goes from node u $(u \in \varepsilon)$ to node v $(v \in \varepsilon)$ as $(u, v \in \varepsilon)$. A graph can be represented using an adjacency matrix.

Adjacency matrix

The adjacency matrix, sometimes also called the connection matrix, of a simple labeled graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (u, v) according to whether u and v are adjacent or not. For a simple graph with no self-loops, the adjacency matrix must have 0s on the diagonal. For an undirected graph, the adjacency matrix is symmetric. The presence of edge between nodes u and v are represented as $A[u, v] = 1$ where $(u, v \in \varepsilon)$

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. 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.

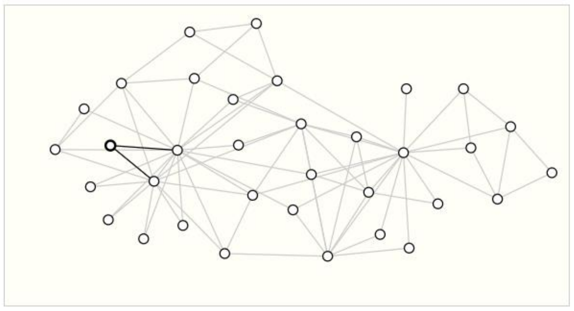
Different Tasks in ML:

- Node level prediction
- Edge level prediction
- Graph level prediction

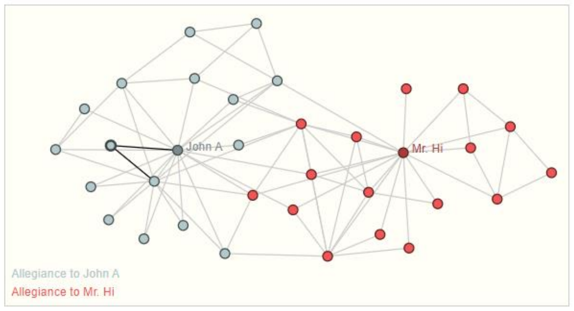
In a graph-level task, we predict a single property for a whole graph. For a node-level task, we predict some property for each node in a graph. For an edge-level task, we want to predict the property or presence of edges in a graph.

Node level prediction

Predicting the identification or purpose of each node in a graph is the focus of node-level tasks. Consider the following scenario: We are given a large social network dataset with millions of users, but we know that a significant number of these users are bots. For a variety of reasons, including the fact that the company may not want to promote bots, identifying these bots is critical. Because manually evaluating each user to determine if they are a bot or not would be prohibitively expensive,hence we would like to develop a graph model that can classify users as bots or not bots.



Input: graph with unlabled nodes

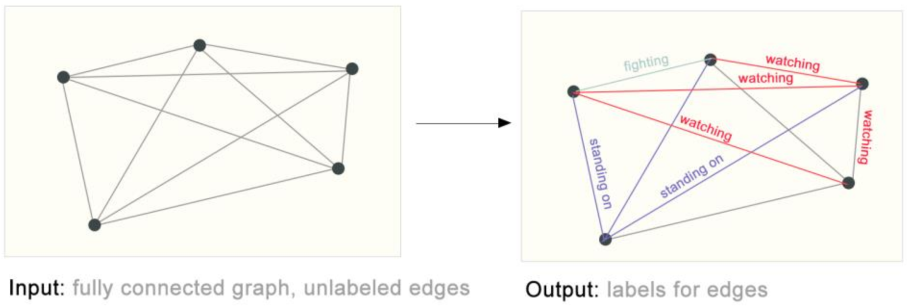


Output: graph node labels

Node-level prediction problems are analogous to **image segmentation**, where we are trying to label the role of each pixel in an image. With text, a similar task would be predicting the parts-of-speech of each word in a sentence (e.g. noun, verb, adverb, etc).

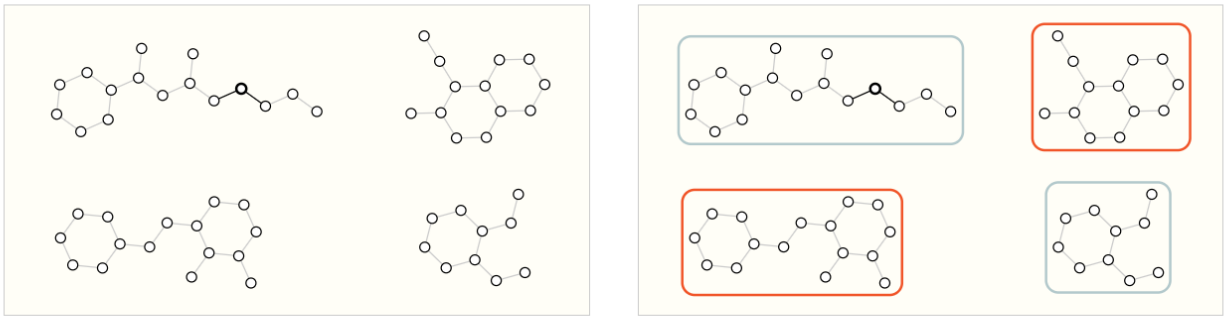
Edge level prediction

Image scene understanding is an example of edge-level inference. Deep learning models can predict the relationship between things in a picture in addition to identifying them. We may think of this as edge-level classification: given a set of nodes that represent the image's objects, we want to predict which of these nodes share an edge and what the value of that edge is. We might consider the graph completely connected and trim edges depending on their anticipated value to arrive at a sparse graph if we want to identify relationships between things.



Graph level prediction

Our goal in a graph-level task is to anticipate an entire graph's property. For example, we could wish to predict what a molecule will smell like or if it will bind to a disease-related receptor using a graph representation. This is similar to MNIST and CIFAR image classification challenges, where we wish to assign a label to a whole image. In text, Sentiment analysis, where we wish to determine the mood or emotion of an entire sentence at once would be an example.


Input: graphs

Output: labels for each graph, (e.g., "does the graph contain two rings?")

Advantages of GNN over CNN

Conventional neural networks require fixed input size whereas GNN are size independent this is because sometimes each node has a different feature size. Each pixel in the image combines information from its neighbors as well as from itself to generate a new value. Because of the unlimited size of the graph and the complicated topology, which implies there is no spatial locality, doing CNN on graphs is extremely challenging. CNN can only work for the datas that can be represented in euclidean space, GNN model has been derived as a generalization
of convolutions to non-Euclidean data.

GNN are also **permutation invariant** . If we use CNN on the same graph data with different node ordering the output of the network also changes. Because the graphs are permutations invariant to node ordering, Whatever the order of the node representation, for the same graph the output should be the same. CNN aggregate feature from spatially defined patched in an image whereas GNN aggregate feature information based on local graph neighborhood.
CNN can only be used for grid structured inputs, RNN can only be used for sequenced inputs, GNN can be used for all generalized graph representations.

Graph Neural Network

Graph Neural Network is a deep learning approach that can be performed on the inference on the graph-based data. The input for this type of neural network is given as graphs and it can do ML tasks for node-level, edge-level, and graph-level events. GNN mainly focuses on learning the embedding for each node containing information of its neighbors, that can be used in many prediction level tasks like node classification, community detection, link prediction, graph classification and network visualization. Basically we map nodes so that similarity in the embedding space approximates similarity in the network. GNN takes node features as input and provides node embeddings that contain features of the neighbor nodes as well as the current embedding of the node.

Graph Convolution

Graph convolution is carried out by the Graph Convolution layer (GCN). The term ‘convolution’ in Graph Convolutional Networks is similar to Convolutional Neural Networks in terms of weight sharing. The main difference lies in the data structure, where GCNs are the generalized version of CNN that can work on data with underlying non-regular structures. The major difference between CNNs and GNNs is that CNNs are specially built to operate on regular (Euclidean) structured data, while GNNs are the generalized version of CNNs where the numbers of nodes connections vary and the nodes are unordered (irregular on non-Euclidean structured data). In GCN, the model learns the features by inspecting neighboring nodes and creates embedding from the relationship between nodes. GCNs can be categorized into 2 major algorithms, Spatial Graph Convolutional Networks and Spectral Graph Convolutional Networks.

Spectral Graph Convolutional Networks

Spectral-based approaches define graph convolutions by introducing filters from the perspective of graph signal processing which is based on graph spectral theory. A drawback of the spectral approach is that it requires the entire graph to be processed simultaneously, which can be impractical for large graphs with billions of nodes and edges such as the social network graphs. Another disadvantage stemming from this requirement of handling the entire graph together is that it is difficult to take advantage of the parallel processing power available today.

Spatial Graph Convolutional Networks

Spatial-based approaches formulate graph convolutions as aggregating feature information from neighbors. Spatial methods perform convolution in the graph domain by aggregating the neighbor nodes’ information. Together with sampling strategies, the computation can be performed in a batch of nodes instead of the whole graph, which has the potential to improve efficiency.

Two main Operations in GNN

- Graph filtering
- Graph Pooling

Graph Filtering: Graph filtering refines the node features

$\boldsymbol{A = \{0, 1\}^{n\times n}, X \in R^{n\times d}}$ **to** $\boldsymbol{A = \{0, 1\}^{n\times n}, X \in R^{n\times d_{new}}}$

where A is the Adjacency matrix with ‘n’ nodes and X matrix contains the feature vectors for each node, ‘d’ is the dimension of the feature vector and ‘d_new’ is the dimension of the refined node feature vector.

Graph Pooling: Helps in dimensionality reduction while retaining the important features.
$\boldsymbol{A = \{0,1\}^{n\times n}, X \in R^{n\times d}}$ **to** $\boldsymbol{A_p = \{0,1\}^{n_p\times n_p}, X \in R^{n_p\times d_{new}}, n_p < n}$
Now the node size and the feature vector size are changed in this process.

Graph approach for image

Image can be represented as a graph in Euclidean space. Pixels in the images can be connected with its neighbors to form a grid. It can be represented as a graph with connection between the adjacent pixels as edges and the pixels as nodes, where the pixel values are the node features. While convolving we perform a dot product between the grid values (i.e. the node embeddings or pixel values in our case) and weight matrix (i.e., filter). In a more generalized term that dot product is called “**AGGREGATION**”. Basically, the aggregation operation summarizes the data to a reduced format. As the Graph Neural Network should be permutation invariant, thus the aggregator operator applied on this set of should be in the way that it should not change it. The most popular choices for aggregation are averaging, summation and normalization. Aggregation is done for all the nodes and updates the node features. As a result, we’ll have the graph with the same structure, but node features will now contain features of neighbors. This process of averaging or summation “convolution”, the sliding from one node to another and applying an aggregator operator, updates all the node features. Thus, the Graph Convolution is the generalized form of CNN.

Neural Message passing in GNN

In Neural message passing, the vector messages are exchanged between the nodes and updated using neural networks. In each message-passing iteration in a GNN, a hidden
embedding $h_u^{(k)}$ corresponding to each node u is updated according to information aggregated from u's graph neighborhood N(u). This message passing can be mathematically represented as\

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

AGGREGATE function takes in the set of neighbor embeddings of the node and generates $m_{N(u)}^{(k)}$ based on information aggregated from their respective neighbors N(u). The UPDATE function combines message $m_{N(u)}^{(k)}$ with the current embedding $h_u^{(k)}$ of node u to generate the updated embedding $h_u^{(k+1)}$. The initial embedding are the node features $h_u^{(0)} = x_{u,}$, which were given as the input. At first update the information about the current neighbor state along with information of the current node. After K-iterations each node gets information of the next k-hop neighbors. K is the number of layers and it ranges from 0 to K-1.

Basic GNN

The basic simplified GNN model is proposed by Merkwirth and Lengauer [2005] and Scarselli et al. [2009]. It is defined as:

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

Here $W_{self}^{(k)}$, $W_{neigh}^{(k)}$ are trainable parameters, $b^{(K)}$ is the bias term, σ denotes the activation function (eg., tanh or ReLU) that imposes elementwise non-linearity to the equation, $h_u^{(k-1)}$ is the previous hidden embedding of the current node and $h_v^{(k-1)}$ is the hidden embedding of the neighborhood of u in the previous step. In the equation above we first sum the messages from the neighbors and combine neighbor information with the previous information using linear combination. Then non-linearity is introduced using the activation function. The aggregate and update steps can be defined as

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

Message Passing with self-loop

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.

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

Instead of a separate update step, the aggregation is taken over the set N(u) U {u} (nodes neighbors as well as the nodes). There are few limitations for this approach; sometimes it might lead to over-fitting or smoothing and the node's neighbors cannot be differentiated from the information from the node itself as the learnable parameters $W_{self}^{(k)}$ and $W_{neigh}^{(k)}$ are shared between them.

Generalized Neighborhood aggregation

There are many aggregation methods available. Here we discuss some of the aggregation operators that can be generalized and improved upon.

Neighborhood normalization

The most basic neighborhood aggregation operation is simply taking the sum of the neighbor embeddings. This approach is highly unstable and highly sensitive to node degrees. Most of the nodes have different degrees, thus summing the values might lead to a drastic difference in magnitude can lead to numerical instabilities as well as difficulties for optimization. Solution for this problem is to simply normalize the aggregation operation based upon the degrees of the nodes involved. The simplest approach is to just take an average rather than sum:

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

Symmetric Normalization

By research, it is found success with other normalization factors, such as the following symmetric normalization,

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

Applying symmetric normalization and self-loop to the basic GNN we get the graph convolutional network (GCN)—employs the symmetric-normalized aggregation as well as the self-loop update approach. The GCN model thus defines the message passing function as

$$h_u^{(k)} = \sigma\left(W^{(k)} + \sum_{v \in N(u) \cup \{u\}} \frac{h_v}{\sqrt{|N(u)||N(v)|}} \right)$$

Aggregation using attention network (GAT)

One of the most popular strategies 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 (GAT), which uses attention weights to define a weighted sum of the neighbors:

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

where $\alpha_{u,v}$denotes the attention on neighbor v ∈ N (u) when we are aggregating information at node u. In the original GAT paper, the attention weights are defined as

$$\alpha_{u,v} = \frac{exp(a^T[Wh_u \oplus Wh_v])}{\sum_{v\prime \in N(u)} exp(a^T[Wh_u \oplus Wh_{v\prime}])}$$

where a is a trainable attention vector, W is a trainable matrix, and ⊕ denotes the concatenation operation.

Generalized Update Method

GNN message passing involves two key steps: aggregation and updating, and in many ways the UPDATE operator plays an equally important role in defining the power and inductive bias of the GNN model. In the basic GNN approach, UPDATE operation involves linear combinations of embeddings of current nodes along with the aggregate of the embeddings of the neighbors. Here we look more into generalizing the update operation.

Issues in updating (over smoothing)

One of the most common issues in updating in GNN is over smoothing. The essential idea of over-smoothing is that after several iterations of GNN message passing, the representations for all the nodes in the graph can become very similar to one another. This issue mostly occurs in the model which incorporates a self-loop approach. This can be overcome by defining the influence of each node's input feature u on the final layer embedding of all the other nodes in the graph. For pairs of nodes u and v, the influence of node u on node v can be quantified by examining the magnitude of the corresponding Jacobian matrix.

$$I_K(u,v) = 1^T \left(\frac{\partial h_u^{(K)}}{\partial h_u^{(0)}} \right) 1,$$

The influence $I_K(u,v)$ values use the sum of the values of the Jacobian matrix as a measure of how much the initial embedding of node u influences the final embedding of node v in the GNN.

Concatenate and skip connections

Sometimes over smoothing occurs when the node information in the embedding gradually washed out or vanished slowly after several iterations of message passing. So, skip connections are also used to overcome this limitation. These concatenation and skip-connection methods can be used in conjunction with most other GNN update approaches.

$$UPDATE_{BASE}\left(h_u, m_{N(u)}\right) = \sigma\left(W_{self} h_u + W_{neigh} m_{N(u)}\right),$$

The base update function is updated as given before, skip connections are updated on top of this base function.

Concatenation

In this we concatenate the output of the base update function with the node's previous-layer representation.

$$UPDATE_{concat}\left(h_v, m_{N(u)}\right) = \left[UPDATE_{base}\left(h_v, m_{N(u)}\right) \oplus h_u\right]$$

Interpolation

There are other forms of skip connections for updating, such as the linear interpolation method.

$$UPDATE_{interpolate}\left(h_v, m_{N(u)}\right) = \alpha_1 o\ UPDATE_{base}\left(h_v, m_{N(u)}\right) + \alpha_1 \odot h_u$$

Where $\alpha_1, \alpha_2 \in [0,1]^d$ are gated vectors with $\alpha_1 = 1 - \alpha_2$ and $\circ$ denotes element wise multiplication. In this approach, the final updated representation is a linear interpolation between the previous representation and the representation that was updated based on the neighborhood information.

GNN for node classification

The standard way to apply GNNs to such a node classification task is to train GNNs in a fully-supervised manner, where we define the loss using a SoftMax classification function and negative log-likelihood loss:

$$L = \sum_{u \in V_{train}} -\log\ (softmax(z_u, y_u))$$

We use SoftMax(zu, yu) to denote the predicted probability that the node belongs to the class yu, computed via the SoftMax function:

$$softmax(z_u, y_u) = \sum_{i=1}^{c} y_u[i] \frac{e^{z_u^T w_i}}{\sum_{j=1}^{c} e^{z_u^T w_j}}$$

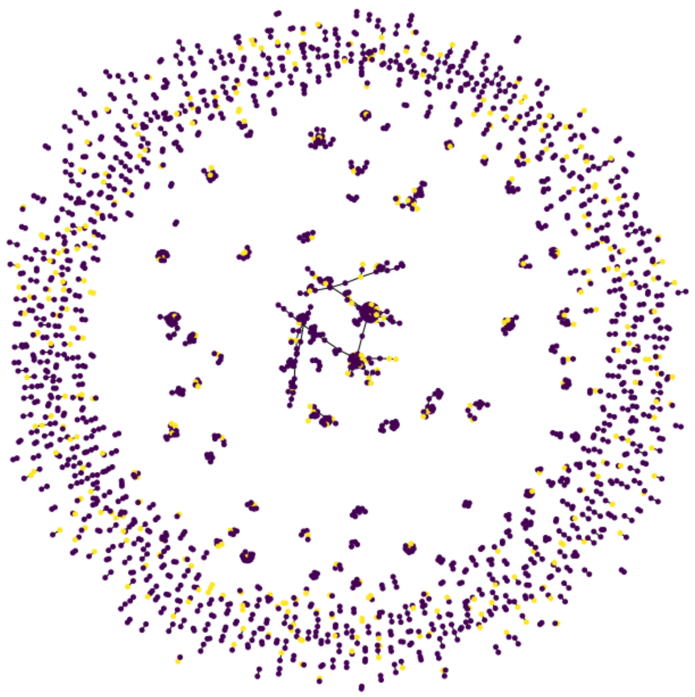
Implementation of node classification for MUSAE GitHub Social Network dataset

Dataset description

A large social network of GitHub developers which was collected from the public API in June 2019. Nodes are developers who have starred at least 10 repositories and edges are mutual follower relationships between them. The vertex features are extracted based on the location, repositories starred, employer and e-mail address.

Implementation steps

In implementation our task is to classify whether the GitHub user is a web or a machine learning developer. This target feature was derived from the job title of each user. It contains 37700 edges, and the maximum length of a feature vector for a certain node is 4005. As it is not possible to feed data of shape (37,700 x 4005), we used truncated SVM for dimensionality reduction after which the shape of the data became (37,700 x 16). The number of web developers in the network are 27961 and number of Machine learning developers are 9739. We have plotted the network graph which shows the interaction between the nodes i.e., GitHub developers.



We can see there are different sub graphs in it. There are few clusters contained only data from one graph, so this can be classified using node classification with GCN method. Feed forward network is defined with a batch normalization layer followed by a dropout layer with dropout rate of 0.2. and a final dense layer with GeLU activation.

The graph convolution layer is custom defined as the method described before. The input parameters for the GCN are hidden units, dropout rate, aggregation type, combination type which is nothing but update and we also specify whether the embedding need to undergo normalization. GCN layer does the aggregation and updating operation. For our use case we define a model with three convolution layers along with a feed forward block defined before. The model is compiled using Adam optimizer with learning rate 0.01. loss is calculated using sparse categorical cross entropy. The model works pretty well with the test dataset as we get the accuracy of **83.71%.**