Graph Neural Network

Yujia Zhu

Node embedding just uses a embedding-lookup to encode node to a low-dimensional vector. It is a type of shallow embedding, thus having some limitations:

1. Every node has its own unique embedding, so there is no shared parameters between nodes
2. Can not generate embeddings for nodes that are not seen during training
3. Can not add node's feature into the embedding

We can use **Graph Neural Network** to solve these problems. Instead of only using an embedding-lookup, we can shift to multiple layers of non-linear transformations based on graph structure. But unlike neural networks' great capability on vision or language tasks, combining neural networks into graphs is more complex because of following reasons:

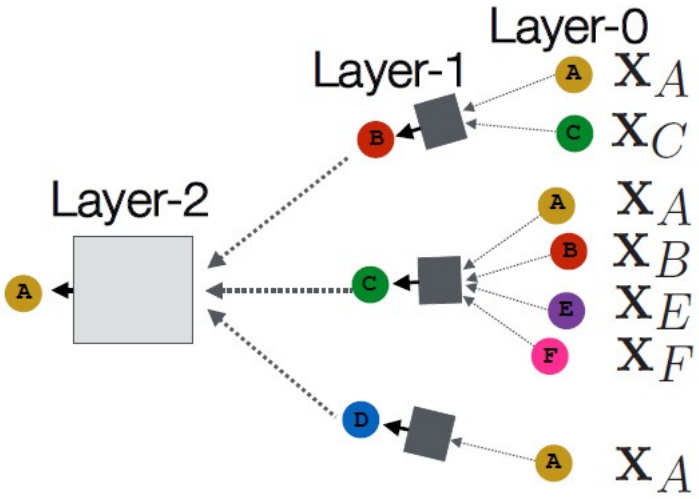
- Graph has arbitrary size and complex topological structure
- There is no fixed order of nodes or reference point in the graph
- Graph is often dymanic and has multimodel features

Network Design

We mainly introduce Graph Convolutional Networks, GraphSAGE and Graph Attention Networks.

Graph Convolutional Networks

Single Convolutional Neural Networks use specified size filters to extract informations of the input features. Actually, the fuction of the filter is doing a linear transformation. When using this idea to the graph, we can transform information at the neighbours and combine it. With this idea, the embedding of the node will be generated based on **local network neighbourhoods**.



Graph Convolutional Network has the architecture shown in the above figure. Nodes will have embeddings in each layer (the embedding in different layers may have different dimensions). Layer-0 embedding of node u

is its input feature vector x_u (by this way, the features of the node can be taken into account). Layer-(k+1) embedding of node u comes from the transformation of node u's neighbours' layer-k embedding. Suppose v is a node in a graph and $h_v^{(l)}$ is the layer-l embedding of node v, then the layer embeddings will have recursive equation:

$$h_v^{(0)} = x_v$$
$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\right), l = 0, 1, ..., L-1$$

$\sigma(.)$ is the non-linear transformation function(e.g. sigmoid, ReLU, etc.), L is the total number of layers. The equation tells that the new embedding is an aggregation of the embedding of itself in the previous layer and the average embedding of its neighbours in the previous layer. W_l, B_l are two learnable parameters. W_l is the weight matrix for neighborhood aggregation and B_l is the weight matrix for transforming hidden vector of the node itself.

GraphSAGE

GraphSAGE makes a little change from Graph Convolutional Network. GCN handles the embeddings of the last layer by a summation operation, but GraphSAGE turns to concatenation operation. So, the recursive equation will change to:

$$h_v^{(0)} = x_v$$
$$h_v^{(l+1)} = \sigma\left(\left[W_l \cdot AGG\left(\{h_u^{(l)}, u \in N(v)\}\right), B_l h_v^{(l)}\right]\right), l = 0, 1, ..., L-1$$

[.] means concatenation function. AGG(.) is also a function to deal with the embedding of the node's neighbours' embedding in the last layers. Since the set $\{h_v^{(l)}, v \in N(u)\}$ has different length for different node, the AGG(.) function is a little complex. There exists several choices:

- Use mean of the embedding, then it is the same as GCN
- Use pool(aka. elementwise-max) of the embedding
- Use a LSTM to reshuffled of neighbors' embedding

Besides using concatenation rather than summation, GraphSAGE also applys L2-normalization to $h_v^{(l)}$ embedding at every layer, $h_v^{(l)} := \frac{h_v^{(l)}}{||h_v^{(l)}||_2}$, where $||u||_2 = \sqrt{\sum_i u_i^2}$. In some cases (not always), normalization of embedding results in performance improvement.

Graph Attention Network

Graph Attention Network uses a weighted sum:

$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \alpha_{vu} h_u^{(l)} + B_l h_v^{(l)}\right)$$

In GCN, $\alpha_{vu} = \frac{1}{|N(v)|}$, so GCN thinks all neighbors $u \in N(v)$ are equally important to node u. Graph Attention Network allocates different weights to different neighbouring nodes and the weight α_{vu} is learned during training. α_{vu} will be calculated by following steps.

1. Define an attention coefficient computing function $a(.)$. Compute attention coefficients $e_{vu} = a(h_v^{(l)}, h_u^{(l)})$, which indicates the importance of node u's message to node v

2. Normalize e_{vu} to α_{vu} using softmax

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

3. Compute the weighted sum to update the embedding

$$h_v^{(l+1)} = \sigma\left(W_l \sum_{u \in N(v)} \alpha_{vu} h_u^{(l)} + B_l h_v^{(l)}\right)$$

The attention function $a(.)$ has multiple choices:

- Dot Product: $a(h_v^{(l)}, h_u^{(l)}) = h_v^{(l)} \cdot h_u^{(l)}$
- Concatenation and Linear Transformation: $a(h_v^{(l)}, h_u^{(l)}) = W[h_v^{(l)}, h_u^{(l)}] + b$, W, b are trainable parameters

In addition, we can use multi-head attention to stabilize the learning process of attention mechanism. We need to create multiple attention scores and aggregate them together by concatenation or summation:

$$h_v^{(l+1)}[1] = \sigma\left(W_l \sum_{u \in N(v)} \alpha_{vu}^1 h_u^{(l)} + B_l h_v^{(l)}\right)$$
$$h_v^{(l+1)}[2] = \sigma\left(W_l \sum_{u \in N(v)} \alpha_{vu}^2 h_u^{(l)} + B_l h_v^{(l)}\right)$$
$$...$$
$$h_v^{(l+1)}[n] = \sigma\left(W_l \sum_{u \in N(v)} \alpha_{vu}^n h_u^{(l)} + B_l h_v^{(l)}\right)$$
$$h_v^{(l+1)} = \left[h_v^{(l+1)}[1], h_v^{(l+1)}[2], ..., h_v^{(l+1)}[n]\right]$$

A Single GNN Layer

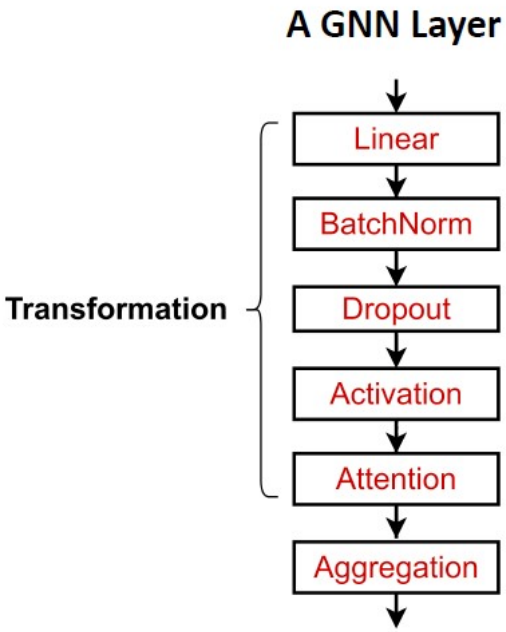
With different types of network design, we will get different graph neural networks. In short, a GNN layer deals the graph information in two procedures:

1. Message: get the message of the node's neighouring nodes
2. Aggregation: aggregate the message from the neighbours

The simpliest GNN will only use a linear transformation and non-linear activation function in a layer. In fact, some much more complicated techniques should be taken into account. They include:

- Batch Normalization: stablize the training process of the neural network
- Dropout: prevent overfitting
- Attention: control the importance of different messages

So, a suggested Graph Neural Network Layer is as follows:



Multi-layer GNN

The standard way to build a multi-layer GNN is to stack GNN layers sequentially. The node embedding $h_u^{(L)}$ will be get after L GNN layers given the input node features $h_u^{(0)} = x_u$. But just stacking multiple GNN layers will lead to the **over-smoothing problem**. The over-smoothing problem means all the node embeddings will converge to the same value. This is really bad because we can not differentiate nodes with node embeddings.

The cause of this problem is related to **receptive field**. For a k-layer GNN, each node embedding will be influenced by the nodes by all the k-hop neighbourhood. When k goes to infinity, all the node embeddings have the same receptive field (all the nodes in the graph). Since the receptive fields of the nodes are the same, their embeddings are inevitably similar.

So, adding more GNN layers do not always help. We need to analyze the necessary receptive field to set a suitable layer number. Do not set layer number L to be unnecessarily large. As is commonly approved, deeper networks will behave better than shallow networks, how can we enhance the expressive power of a GNN under the limitation of over-smoothing problem?

The first solution is to increase the expressive power within each GNN layer. For example, we can change the linear layer to a 3-layers MLP.

The second solution is to add layers that do not pass messages. A GNN does not necessarily only contain GNN layers and it can have some pre-process layers and post-process layers. For example, if the node of a graph is an image, we can add some convolution layers to transform the image to a feature vector and just send the feature vector to the GNN. These convolution layers are pre-process layers.

The third solution is to use skip connections in GNNs. Suppose the transfromation function of the l-th layer is $G^l(.)$, then in traditional GNN, we have $h_u^{(l+1)} = G^l(h_u^{(l)})$. With skip connections, the recursive equation becomes:

$$
\begin{aligned}
h_u^{(l+1)} &= G^l(h_u^{(l)}) + h_u^{(l)} \\
&= \sigma\left(W_l \sum_{v \in N(u)} \frac{h_v^{(l)}}{|N(u)|} + B_l h_u^{(l)}\right) + h_u^{(l)}
\end{aligned}
$$

Another option using skip connection is that the final layer directly aggregates from the all the node embeddings in the previous layers and then makes a transformation to get the final embedding (i.e. $h_u^{(final)} = F(h_u^{(0)}, h_u^{(1)}, ..., h_u^{(L)})$). The transformation can be a linear transformation/sum/Lstm/etc.

Graph Augmentation

Sometimes, we need to do graph augmentation to get the optimal computional graph for embedding. Graph Augmentation can be devided to **feature-level augmentation** and **structure-level augmentation**.

Feature Augmentation

In GNN, the input of node u is the feature vector x_u. But, input graphs do not always have node features and this is common when we only have the adjacency matrix. No input features may lead to problems in some tasks. Suppose we want to learn the length of a cycle that node v resides in. Without node features, the task is impossible to be finished. There are mainly three feature augmentation methods:

1. Assign constant values to nodes
2. Assign unique IDs (i.e. one-hot vector) to nodes
3. Assign hand-craft features (e.g. clustering coefficient, pagerank, centrality, etc.) to nodes

Structure Augmentation

When the graph is too sparse or too dense, we need to use structure augmentation. Structure augmentation include:

1. Add virtual edges: The common approach is to connect 2-hop neighbours via virtual edge.
2. Add virtual nodes: The virtual node will be connected to all the nodes in the graph. After adding the virtual node, all the nodes will have a distance up to 2. So, it greatly improves message passing in sparse graphs.
3. Node Neighbourhood Sampling: This is suitable for dense graph. We can randomly sample a node's neighbourhood for message passing. In expectation, we can get embeddings similar to the case where all the neighbors are used and it greatly reduces computional costs.

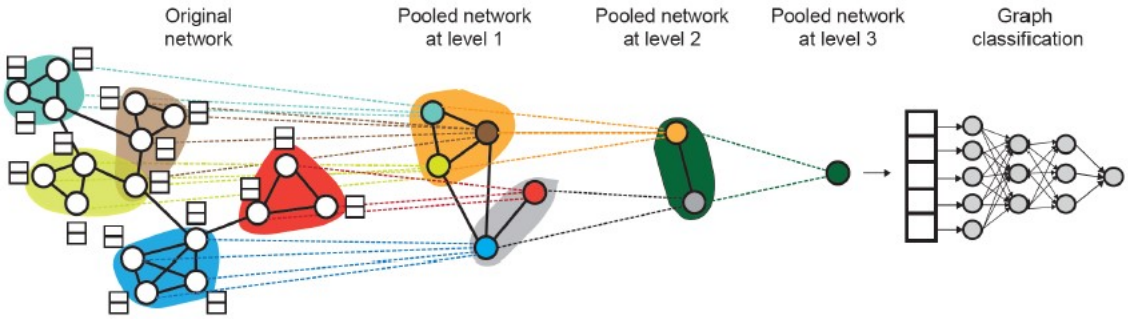
Optimize and Train the Network

Above we have introduced the structure of Graph Neural Network and how to augment graphs. But, we still need to define a loss function and use the loss function to optimize the parameters in neural network in order to train it.

Suppose the output of the GNN is $h_u^{(l)} \in R^d$. For different tasks, we will do different transformations of the embedding and use different loss functions. Suppose k is the dimension of prediction space. If we are doing regression tasks, then k equals to 1 and MSE loss is recommended. If we are doing classification tasks, then k equals to the categories of the variable and cross entropy loss is recommended. Suppose y_u is the label of node u. The label can be get by supervised ways (manual labeling) or unsupervised/self-supervised ways (use the relationship in graphs).

- For node-level prediction, we directly make predictions by node embeddings via a map from $R^d \to R^k$. The loss can be calculated by $\hat{y}_u = M(h_u^{(l)})$ and y_u.

- For edge-level prediction, we make prediction by pairs of node embedding. The loss can be calculated by $\hat{y}_{vu} = M(h_u^{(l)}, h_v^{(l)})$ and y_{vu}. The function $M(.)$ is commonly concatenation+linear-transformation or dot prodction.
- For graph-level prediction, we make prediction by all the node embedding. Global mean/sum/max pooling is the most commonly-used method, but hierarchical global pooling is much more recommended. DiffPool is a representative of hierarchical global pooling. The following figure shows its principles. We pool the subgroups at different phases gradually and get the aggregated representation at last.



Dataset Split

A basic procedure of machine learning is to split data to training/validation/test set. But spliting data on graph is much more complex because data points are not independent. We have two options.

The first choice is **transductive setting**. By transductive setting, the input graph can be observed in all the dataset splits and we only split labels. So, training/validation/test sets are on the same graph. This is applicable to node/edge prediction tasks.

The other choice is **inductive setting**. By inductive setting, we break edges between splits to get multiple graphs. So, training/validation/test sets are on the different graph. Each split can only observe the graph within the split. Compared with transductive setting, inductive setting is a successful model to generalize to unseen graphs. This is applicable to node/edge/graph prediction tasks.

Summary

In this file, we cover Graph Neural Network and its related contents including stacking layers, training, graph augmentation and data splits.

Suppose you are given a graph to do a node/edge/graph prediction task. The steps are as follows:

1. Data splits: Use transductive setting or inductive setting ? Should we use some augmentation methods on the graph ?
2. Design networks: Use GCN or GraphSAGE or GAT ? How many layers should the network have ?
3. Define the loss function and optimize it.