Comp 6730 Advanced Database Systems Project

GNN Implementation

Due 11:59pm Sunday April 30, in Blackboard

Graph Neural Networks (GNNs) are a class of neural network architectures used for deep learning on graph-structured data. Broadly, GNNs aim to generate high-quality embeddings of nodes by iteratively aggregating feature information from local graph neighborhoods using neural networks; embeddings can then be used for recommendations, classification, link prediction or other downstream tasks. Two important types of GNNs are GCNs (graph convolutional networks) and GraphSAGE (graph sampling and aggregation).

Let $G = (V, E)$ denote a graph with node feature vectors X_u for $u \in V$. To generate the embedding for a node u, we use the neighborhood of the node as the computation graph. At every layer l, for each pair of nodes $u \in V$ and its neighbor $v \in V$, we compute a message function via neural networks, and apply a convolutional operation that aggregates the messages from the node's local graph neighborhood (Figure 1), and updates the node's representation at next layer. By repeating this process through K GNN layers, we capture feature and structural information from the local K-hop neighborhood. For each of the message computation, aggregation and update functions, the learnable parameters are shared across all nodes in the same layer.

We initialize the feature vector of each node X_u based on the data we have available. If we already have outside information about the nodes, we can embed that as a feature vector. Otherwise we can use constant feature (vector of 1) or the degree of the node as the feature vector.

These are the key steps in each layer of a GNN:

- **Message computation:** We use a neural network to learn a message function between nodes. For each pair of nodes u and its neighbor v, the neural network message function can be expressed as $M(h_u^k, h_v^k, e_{u,v})$. In GCN and GraphSAGE, this can simply be $\sigma(Wh_v + b)$, where W and b are the weights are bias of a neural network linear layer. Here h_u^k refers to the hidden representation of node u at layer k, and $e_{u,v}$ denotes available information about the edge (u, v), like the edge weight or other features. For GCN and GraphSAGE, the neighbors of u are simply defined as nodes that are connected to u. However, many other variants of GNNs have different definitions of neighborhood.

- **Aggregation:** At each layer, we apply a function to aggregate information from all of the neighbors of each node. The aggregation function is usually permutation invariant, to reflect the fact that nodes' neighbors have no canonical ordering. In a GCN, the aggregation is done by a weighted sum, where the weight for aggregating from v to u corresponds to the (u, v) entry of the normalized adjacency matrix $D^{-1/2}AD^{-1/2}$.

- **Update:** We update the representation of a node based on the aggregated representation of the neighborhood. For example, in GCNs, a multi-layer perceptron (MLP) is used; GraphSAGE combines a skip layer with the MLP.

- **Pooling:** The representation of an entire graph can be obtained by adding a pooling layer at the end. The simplest pooling methods are just taking the mean, max or sum of all of the individual node representations. This is usually done for the purposes of graph classification

We can formulate the Message computation, Aggregation and Update steps for a GCN as a layer-wise propagation rule given by:

$$h^{k+1} = \sigma\left(D^{-\frac{1}{2}}AD^{-\frac{1}{2}}h^kW^k\right) \tag{4}$$

where h^k represents the matrix of activations in the k-th layer, $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the normalized adjacency of graph G, W^k is a layer-specific learnable matrix, and σ is a non-linearity function. Dropout and other forms of regularization can also be used.

We provide the pseudo-code for GraphSAGE embedding generation below.

```
Algorithm 1: Pseudo-code for forward propagation in GraphSAGE
  Input  : Graph $G(V, E)$; input features $\{x_v, \forall v \in V\}$; depth $K$; non-linearity $\sigma$;
           weight matrices $\{W^k, \forall k \in [1, K]\}$; neighborhood function $\mathcal{N} : v \to 2^V$;
           aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$
  Output: Vector representations $z_v$ for all $v \in V$
  $h_v^0 \leftarrow x_v, \forall v \in V$ ;
  for $k = 1...K$ do
      for $v \in V$ do
          $h_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ // aggregation
          $h_v^k \leftarrow \sigma\left(W^k \cdot \text{CONCAT}(h_v^{k-1}, h_{\mathcal{N}(v)}^k)\right)$ // MLP with skip connection
      $h_v^k \leftarrow h_v^k / \|h_v^k\|_2, \forall v \in V$ // update step
  $z_v \leftarrow h_v^K, \forall v \in V$
```

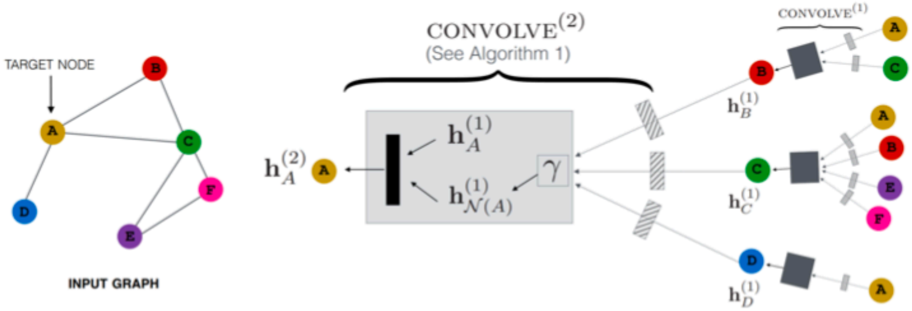

Figure 1: GNN architecture

Attention mechanisms have become the state-of-the-art in many sequence-based tasks such as machine translation and learning sentence representations. One of the major benefits of attention-based mechanisms is their ability to focus on the most relevant parts of the input to make decisions. In this problem, we will see how attention mechanisms can be used to perform node classification of graph-structured data through the usage of Graph Attention Networks (GATs).

The building block of the Graph Attention Network is the graph attention layer, which is a variant of the aggregation function (see step 2 in Question 3). Let N be the number of nodes and F be the dimension of the feature vector for each node. The input to each graph attentional layer is a set of node features: $\mathbf{h} = \{\overrightarrow{h_1}, \overrightarrow{h_2}, \dots, \overrightarrow{h_N}\}$, $\overrightarrow{h_i} \in R^F$. The output of each graph attentional layer is a new set of node features, which may have a new dimension F': $\mathbf{h'} = \{\overrightarrow{h'_1}, \overrightarrow{h'_2}, \dots, \overrightarrow{h'_N}\}$, with $\overrightarrow{h'_i} \in \mathbb{R}^{F'}$.

We will now describe this transformation of the input features into higher-level features performed by each graph attention layer. First, a shared linear transformation parametrized by the weight matrix $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is applied to every node. Next, we perform self-attention on the nodes. We use a shared attentional mechanism:

$$a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}. \tag{5}$$

This mechanism computes the attention coefficients that capture the importance of node j's features to node i:

$$e_{ij} = a(\mathbf{W}\overrightarrow{h_i}, \mathbf{W}\overrightarrow{h_j}) \tag{6}$$

The most general formulation of self-attention allows every node to attend to all other nodes which drops all structural information. To utilize graph structure in the attention mechanisms, we can use masked attention. In masked attention, we only compute e_{ij} for nodes $j \in \mathcal{N}_i$ where $\mathcal{N}_i$ is some neighborhood of node i in the graph.

To easily compare coefficients across different nodes, we normalize the coefficients across j using a softmax function:

$$\alpha_{ij} = \text{softmax}_j(e_{ij}) = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}$$

For this problem, our attention mechanism a will be a single-layer feedforward neural network parametrized by a weight vector $\overrightarrow{a} \in \mathbb{R}^{2F'}$, followed by a LeakyReLU nonlinearity (with negative input slope 0.2). Let $\cdot^T$ represent transposition and $||$ represent concatenation. The coefficients computed by our attention mechanism may be expressed as:

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\overrightarrow{a}^T[\mathbf{W}\overrightarrow{h_i}||\mathbf{W}\overrightarrow{h_j}]\right)\right)}{\sum_{k \in \mathcal{N}_i} \exp\left(\text{LeakyReLU}\left(\overrightarrow{a}^T[\mathbf{W}\overrightarrow{h_i}||\mathbf{W}\overrightarrow{h_k}]\right)\right)} \tag{7}$$

Now, we use the normalized attention coefficients to compute a linear combination of the features corresponding to them. These aggregated features will serve as the final output features for every node.

$$h_i' = \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\overrightarrow{h_j}. \tag{8}$$

To stabilize the learning process of self-attention, we use multi-head attention. To do this we use K independent attention mechanisms, or "heads" compute output features as in Equation 8. Then, we concatenate these output feature representations:

$$\overrightarrow{h_i}' = ||_{k=1}^{K}\Big(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^{(k)}\mathbf{W}^{(k)}\overrightarrow{h_j}\Big) \tag{9}$$

where $||$ is concentation, $\alpha_{ij}^{(k)}$ are the normalized attention coefficients computed by the k-th attention mechanism (a^k), and $\mathbf{W}^{(k)}$ is the corresponding input linear transformation's weight matrix. Note that for this setting, $\mathbf{h}' \in \mathbb{R}^{KF'}$.

In this problem, we will be implementing a general Graph Neural Network Stack, as well as GraphSAGE and GATs. We will use our implementations to complete graph and node classification on two benchmarks:

- CORA is a standard citation network benchmark dataset. In this dataset, nodes correspond to documents and edges correspond to undirected citations. Each node has a class label. The node features are elements of a bag-or-words representation of a document. For the Cora dataset, there are 2708 nodes, 5429 edges, 7 prediction classes for nodes, and 1433 features per node. We will be performing node classification on CORA.
- ENZYMES is a relatively small graph classification benchmark, containing graphs representing the protein tertiary structures from the BRENDA enzyme database. The task is to correctly assign each enzyme to one of the 6 EC top-level classes. It has 600 graphs, classified into 6 classes. The average number of nodes 32.63; average number of edges 62.14. Every node has attribute with 18 dimensions. We will be performing graph classification on ENZYMES.

For the questions below, you will need to modify the files **train.py** and **models.py**. We've left TODO blocks in **model.py** indicating what you need to modify. To implement GraphSage and GAT, we will be extending the **MessagePassing** base class of PyTorch geometric. You may find the **MessagePassing** documentation found here to be useful. In this documentation, you will find an example implementation of GCNs by extending the **MessagePassing** base class. We will be doing a similar extension for the implementations of **GraphSage** and **GAT**.

Before starting the implementation, we recommend reading over **train.py** and **models.py** and understanding the structure of the code. In **train.py**, take a look at the training and testing loops and what we do differently for node and graph classification.

1. **Warm-up. [10 Points]** First let's examine the data. To do this, you will need use train.py. How many nodes are there in the test set of CORA? How many graphs are there in the test set of ENZYMES?
2. **Coding. [10 Points]** Now we will implement the GNN Stack. Complete the forward method for GNNStack in models.py. For the purpose of this exercise, each layer of the GNN stack should contain a CONV → RELU → DROPOUT structure, where the CONV layer is specified by the model type. This forward call will be used for both the node classification task and graph classification task later.
3. **Coding. [30 Points]** Complete the GraphSage implementation in models.py by filling in the **__init__** , **forward**, and **message** methods. Use Algorithm 1 for reference. For the aggregate

function, we will be using a dense layer followed by a RELU non-linearity, and a mean aggregator.

4. **Coding. [30 Points]** Complete the GAT implementation by filling in **__init__, forward**, and **message** methods. In **__init__** will need to define the layers we need for the attention mechanism and for aggregating the final features. In **forward**, we will apply a linear transformation to the node feature matrix before starting message propagation. For **message**, we will implement GAT message passing following the equation 7.
5. **Coding. [10 Points]** Run GNN training for node classification task on the CORA dataset, and graph classification task on the ENZYMES dataset. Do this for GCN, GraphSage, and GAT. You will be using the Torch Geometric implementation of GCN, and your implementations of GraphSage and GAT. You can specify the model type to use in training by the flag –model type of train.py. Write code to plot the validation accuracy over number of epochs. For each dataset, please plot the validation accuracy vs epochs for each of the models.
6. **[10 Points]** Describe the performance differences between GCN, GraphSage, and GAT on both tasks. Which model performed best for each task?

What to Submit

- Number of nodes in CORA.
- Number of graphs in ENZYMES.
- Submit **models.py** and **train.py** in the code zip file.
- Two plots, one for each dataset. Each plot will be of validation accuracy vs epoch for each of the three models.
- Description of the performance differences between the three models. For each task, which model performed the best?