Deep Learning Assignment
Assignment 7: Introduction to Graph Neural Networks

Mohammad Parsa Dini - Std ID: 400101204

May 2025

Introduction

Graph Neural Networks (GNNs) are a class of neural networks designed to operate on graph-structured data, enabling the modeling of relationships and dependencies between entities represented as nodes and edges. This assignment focuses on implementing and evaluating two GNN models, Graph Convolutional Networks (GCN) and GraphSAGE, using the PyTorch Geometric (PyG) framework on the Cora dataset for vertex classification. The goal is to understand the message passing algorithm, implement GCN and GraphSAGE layers, and compare their performance with different aggregation methods (mean, sum, max).

Graph Neural Networks

GNNs extend traditional neural networks to handle graph data, where entities are represented as nodes and their relationships as edges. Unlike standard neural networks that assume independent data points, GNNs leverage the graph structure to aggregate information from neighboring nodes, making them suitable for tasks like node classification, link prediction, and graph classification. The Cora dataset, used in this assignment, is a citation network with 2,708 nodes (papers) and 7 classes (research topics), where edges represent citations, and node features are bag-of-words representations of paper abstracts.

Message Passing Algorithm

The message passing algorithm is the core mechanism of GNNs, enabling nodes to exchange and aggregate information from their neighbors iteratively. It consists of three key steps:

1. **Message**: Compute messages from neighboring nodes, typically based on their features and edge attributes.
2. **Aggregation**: Aggregate messages from neighbors using functions like sum, mean, or max to produce a single representation for each node.
3. **Update**: Update each node's representation using the aggregated messages, often through a neural network layer.

In PyTorch Geometric, the `MessagePassing` base class automates this process, decoupling the message, aggregation, and update steps. The general update rule is:

$$\mathbf{x}_i^{(k)} = \text{UPDATE}\left(\mathbf{x}_i^{(k-1)}, \text{AGGR}_{j\in\mathcal{N}(i)}\text{MESSAGE}^{(k)}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{i,j}\right)\right)$$

where $\mathbf{x}_i^{(k)}$ is the feature vector of node i at layer k, $\mathcal{N}(i)$ is the set of neighbors of node i, and $\mathbf{e}_{i,j}$ represents edge features (if applicable).

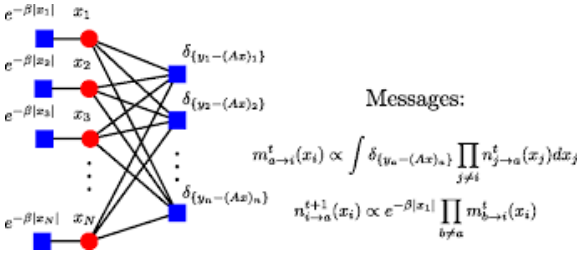

Figure 1: A scheme of Message Passing Algorithm

Architecture and Code Implementation

The notebook implements two GNN models for vertex classification on the Cora dataset: GCN and GraphSAGE. Below, we describe the architecture, implementation details, and key components of the code.

GCN Implementation

The GCN layer, based on Kipf & Welling (ICLR 2017), is implemented as a subclass of `MessagePassing` with the following formula:

$$\mathbf{x}_i^{(k)} = \sum_{j\in\mathcal{N}(i)\cup\{i\}} \frac{1}{\sqrt{\deg(i)}\cdot\sqrt{\deg(j)}} \cdot \left(\mathbf{x}_j^{(k-1)}\mathbf{\Theta}\right)$$

where $\mathbf{\Theta}$ is a learnable weight matrix, and $\deg(i)$ is the degree of node i. The implementation involves:

1. Adding self-loops to the adjacency matrix. 2. Linearly transforming node features using a weight matrix. 3. Normalizing features by the inverse square root of node degrees. 4. Aggregating (summing) normalized features from neighbors. 5. Returning updated node embeddings.

The `GCNConv` class implements these steps, with `forward`, `message`, and `update` methods. The `Net` class constructs a two-layer GCN model: - First layer: `GCNConv(num_features, 16)` with ReLU

activation and dropout (0.5). - Second layer: `GCNConv(16, num_classes)` followed by log-softmax for classification.

```python
class Net(torch.nn.Module):
    def __init__(self, dataset):
        super(Net, self).__init__()
        self.conv1 = GCNConv(dataset.num_features, hidden)
        self.conv2 = GCNConv(hidden, dataset.num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.conv1(x, edge_index))
        x = F.dropout(x, p=dropout, training=self.training)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

GraphSAGE Implementation

GraphSAGE, based on Hamilton et al. (NIPS 2017), is an inductive GNN framework that aggregates neighbor features using mean, sum, or max functions, followed by a linear transformation of concatenated self and aggregated features. The `SAGEConv` class implements:

$$\mathbf{h}_i^{(k)} = \mathbf{W} \cdot \left[\mathbf{x}_i^{(k-1)} \,||\, \text{AGGR}_{j \in \mathcal{N}(i)} \left(\mathbf{x}_j^{(k-1)} \right) \right]$$

where $\mathbf{W}$ is a learnable weight matrix, and AGGR is the aggregation function (mean, sum, or max). The `SAGENet` class constructs a two-layer model similar to the GCN model, with dropout and ReLU between layers.

```python
class SAGENet(torch.nn.Module):
    def __init__(self, dataset, aggr='mean'):
        super(SAGENet, self).__init__()
        self.conv1 = SAGEConv(dataset.num_features, hidden, aggr=aggr)
        self.conv2 = SAGEConv(hidden, dataset.num_classes, aggr=aggr)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = F.relu(self.conv1(x, edge_index))
        x = F.dropout(x, p=dropout, training=self.training)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

Training and Evaluation

Both models are trained on the Cora dataset using the Adam optimizer with a learning rate of 0.01, weight decay of 0.0005, and early stopping after 10 epochs if validation loss does not improve. The training loop involves: - Forward pass to compute logits. - Negative log-likelihood loss on the training mask. - Backpropagation and weight updates. - Evaluation on train, validation, and test masks, computing loss and accuracy.

The `run` function executes 10 runs, reporting the mean and standard deviation of validation loss, test accuracy, and runtime.

Results

The reported results for the models are as follows: - **GCN**: - Validation Loss: 0.7452 - Test Accuracy: 0.798 ± 0.008 - Duration: 6.482 seconds - **GraphSAGE (Mean Aggregation)**: - Validation Loss: 0.7619 - Test Accuracy: 0.790 ± 0.015 - Duration: 0.905 seconds - **GraphSAGE (Sum Aggregation)**: - Validation Loss: 1.0896 - Test Accuracy: 0.738 ± 0.028 - Duration: 0.775 seconds - **GraphSAGE (Max Aggregation)**: - Validation Loss: 0.9350 - Test Accuracy: 0.762 ± 0.015 - Duration: 0.951 seconds

The GCN model achieves the highest test accuracy (0.798), followed by GraphSAGE with mean aggregation (0.790). Sum and max aggregations perform worse, with test accuracies of 0.738 and 0.762, respectively. GraphSAGE models are significantly faster, with runtimes under 1 second compared to GCN's 6.482 seconds, likely due to simpler aggregation mechanisms.

Analysis and Inferences

- **GCN vs. GraphSAGE**: GCN normalizes neighbor contributions by degree, which may help in stabilizing training on graphs with varying node degrees, leading to better accuracy on Cora. GraphSAGE's inductive approach, which does not rely on the entire graph structure during training, makes it faster but less effective on this transductive task. - **Aggregation Methods**: Mean aggregation in GraphSAGE performs best among the three, as it balances neighbor contributions, while sum aggregation may overemphasize high-degree nodes, and max aggregation may lose information by focusing only on the maximum value. - **Code Robustness**: The code includes proper initialization (Glorot for GCN, uniform for GraphSAGE) and handles edge cases (e.g., preventing division by zero in degree normalization). The use of `torch.cuda.synchronize` ensures accurate timing on GPU. - **Improvements**: The GraphSAGE implementation could include normalization (e.g., layer normalization) or additional non-linearities in the update step to improve performance. The GCN model could benefit from additional layers or residual connections for deeper architectures.

Conclusion

This assignment provided hands-on experience with GNNs using PyTorch Geometric, implementing GCN and GraphSAGE for vertex classification on the Cora dataset. The GCN model outperformed GraphSAGE in accuracy, while GraphSAGE was faster, particularly with mean aggregation. The message passing framework was effectively utilized to propagate and aggregate node features, demonstrating the power of GNNs in leveraging graph structure for machine learning tasks.