Graph Neural Network

Pseudocode

Pre-Requisites:

1. os: Provides a way of using operating system-dependent functionality like interacting with the file system.
2. json: Used for working with JSON data.
3. math: Provides access to mathematical functions.
4. numpy (as np): A fundamental package for numerical computations in Python, used for working with arrays and matrices.
5. time: Used for time-related functions.
6. matplotlib.pyplot (as plt): The primary plotting library, used for creating static, animated, and interactive visualizations.
7. seaborn (as sns): Built on top of Matplotlib, it's used for making statistical graphics more aesthetically pleasing and informative.
8. tqdm: A fast, extensible progress bar for loops.

PyTorch and Related Libraries:

1. torch: The core PyTorch library for building and training neural networks.
2. torch.nn: Contains modules for building neural network layers.
3. torch.nn.functional (as F): Provides a set of stateless functions like activation functions and convolutions.
4. torch.utils.data: Contains classes for data loading and manipulation, such as DataLoader and Dataset.
5. torch.optim: Provides a variety of optimization algorithms like Adam and SGD.
6. torchvision: A package that contains popular datasets, model architectures, and common image transformations for computer vision.
7. pytorch_lightning (as pl): A lightweight PyTorch wrapper that simplifies the training of models by automating boilerplate code.
8. torch_geometric: A library for deep learning on graphs.
9. LearningRateMonitor and ModelCheckpoint: Callbacks from PyTorch Lightning for monitoring and saving models during training.

Graph Layer Definition:

Graph Representation

- Explain graph representation using adjacency matrix and edge list
- Introduce the GCNLayer class
    - Initialize with input and output feature dimensions
    - Implement the forward method:
        - Calculate the number of neighbors
        - Project node features
        - Perform matrix multiplication with adjacency matrix
        - Normalize by the number of neighbors
- Demonstrate GCNLayer with an example graph and features
- Discuss limitations of GCNs (forgetting node-specific information)
- Introduce the GATLayer class
    - Initialize with input/output dimensions, number of heads, concatenation option, and leaky ReLU alpha
    - Implement the forward method:
        - Apply linear projection to node features
        - Reshape features for multi-head attention
        - Get edge indices from adjacency matrix
        - Concatenate features of connected nodes
        - Calculate attention logits using a linear layer and leaky ReLU
        - Map attention values back to a matrix
        - Apply softmax to get attention probabilities
        - Perform weighted average of node features using attention probabilities
        - Concatenate or average head outputs

- Demonstrate GATLayer with an example graph and features

PyTorch Geometric

- Introduce PyTorch Geometric library
- Explain edge representation in PyTorch Geometric (list of index pairs)
- Define a dictionary mapping layer names to PyTorch Geometric classes

Main Execution:

Experiments on Graph Structures

Node-level tasks: Semi-supervised node classification [CORA Dataset]

- Introduce node classification task
- Describe the Cora dataset
- Load the Cora dataset using PyTorch Geometric
- Explain the Data object structure
- Implement the GNNModel class
    - Initialize with input, hidden, and output dimensions, number of layers, layer name, and dropout rate
    - Create a list of graph layers, ReLU activations, and dropout
    - Implement the forward method:
        - Iterate through layers
        - Apply graph layers with edge index
        - Apply other layers (ReLU, Dropout)
- Implement the MLPModel class (baseline)
    - Initialize with input, hidden, and output dimensions, number of layers, and dropout rate
    - Create a sequential model of linear layers, ReLU, and dropout
    - Implement the forward method:

        - Pass input through the sequential model

- Implement the NodeLevelGNN (PyTorch Lightning Module)
    - Initialize with model name and kwargs
    - Select either GNNModel or MLPModel
    - Define the loss function (CrossEntropyLoss)
    - Implement the forward method:
        - Pass data through the model
        - Apply mask based on mode (train, val, test)
        - Calculate loss and accuracy
    - Configure optimizer
    - Implement training, validation, and test steps
- Implement the train_node_classifier function
    - Set seed
    - Create DataLoader
    - Create PyTorch Lightning Trainer
    - Check for and load pre-trained model
    - If no pre-trained model, initialize and train the model
    - Load the best model from the checkpoint
    - Test the model
    - Return model and results
- Implement print_results function for displaying accuracy
- Train and evaluate MLP model on Cora
- Train and evaluate GNN model on Cora

Edge-level tasks: Link prediction

- Briefly describe the link prediction task

Graph-level tasks: Graph classification [MUTAG Dataset]

- Introduce graph classification task
- Describe the MUTAG dataset
- Load the MUTAG dataset using PyTorch Geometric
- Print dataset statistics
- Split dataset into training and testing sets
- Explain the batching strategy for graph datasets in PyTorch Geometric
- Create DataLoaders for training, validation, and testing
- Load and print a batch to show batching in action
- Implement the GraphGNNModel class
    - Initialize with input, hidden, and output dimensions, linear dropout rate, and GNN kwargs
    - Create a GNNModel instance
    - Create a sequential head with dropout and a linear layer
    - Implement the forward method:
        - Pass data through the GNN
        - Apply global mean pooling based on batch index
        - Pass pooled features through the head
- Implement the GraphLevelGNN (PyTorch Lightning Module)
    - Initialize with model kwargs
    - Create a GraphGNNModel instance
    - Define the loss function (BCEWithLogitsLoss or CrossEntropyLoss)
    - Implement the forward method:
        - Pass data through the model
        - Squeeze the output dimension
        - Calculate predictions based on output dimension

        - Calculate loss and accuracy
    - Configure optimizer
    - Implement training, validation, and test steps
- Implement the train_graph_classifier function
    - Set seed
    - Create PyTorch Lightning Trainer
    - Check for and load pre-trained model
    - If no pre-trained model, initialize and train the model
    - Load the best model from the checkpoint
    - Test the model on train and test sets
    - Return model and results
- Train and evaluate GraphConv model on MUTAG
- Print training and test performance