1. `BasicGraphConvolutionLayer`

The `BasicGraphConvolutionLayer` is a custom implementation of a graph convolution layer. Here's a breakdown of the mathematical operations within this layer.

Mathematics Behind the `BasicGraphConvolutionLayer`:

Layer Parameters:

- $\mathbf{W}_1$ and $\mathbf{W}_2$ are the learnable weight matrices:
  - $\mathbf{W}_1$ has dimensions $\text{in\_channels} \times \text{out\_channels}$.
  - $\mathbf{W}_2$ has dimensions $\text{in\_channels} \times \text{out\_channels}$.
- $\mathbf{b}$ is the bias vector, with dimensions out_channels.

Forward Pass:

Given:

- $\mathbf{X}$ is the node feature matrix, with dimensions $N \times \text{in\_channels}$.
- $\mathbf{A}$ is the adjacency matrix, with dimensions $N \times N$.

Step 1: Compute Potential Messages

$$\text{potential\_msgs} = \mathbf{X}\mathbf{W}_2$$

- Here, $\mathbf{X}\mathbf{W}_2$ results in a matrix of dimensions $N \times \text{out\_channels}$.
- This step applies the learnable transformation $\mathbf{W}_2$ to the node features.

Step 2: Propagate Messages

$$\text{propagated\_msgs} = \mathbf{A} \times \text{potential\_msgs}$$

- $\mathbf{A} \times \text{potential\_msgs}$ results in a matrix of dimensions $N \times \text{out\_channels}$.
- This step aggregates messages from neighboring nodes. The adjacency matrix $\mathbf{A}$ determines how messages are propagated, with each node receiving information from its neighbors.

Step 3: Compute Root Update

$$\text{root\_update} = \mathbf{X}\mathbf{W}_1$$

- Similar to the first step, $\mathbf{X}\mathbf{W}_1$ is a transformation of the node features using a different weight matrix $\mathbf{W}_1$.

Step 4: Combine and Add Bias

$$\text{output} = \text{propagated\_msgs} + \text{root\_update} + \mathbf{b}$$

- The final output is obtained by combining the propagated messages, the root update, and the bias term. This output has dimensions $N \times \text{out\_channels}$.

Overall Operation:

The `BasicGraphConvolutionLayer` performs a graph convolution that combines information from neighboring nodes (through the adjacency matrix $\mathbf{A}$) and applies learnable transformations to the node features.

2. `NodeNetwork`

The `NodeNetwork` class defines a GNN model that stacks two `BasicGraphConvolutionLayer` layers, followed by fully connected (FC) layers to produce an output for each graph.

Mathematics Behind the `NodeNetwork`:

Layer Setup:

- **conv_1** and **conv_2** are instances of `BasicGraphConvolutionLayer`:
  - conv_1 transforms the input features from input_features to 32 channels.
  - conv_2 transforms the features from 32 to 32 channels.
- **fc_1** is a fully connected layer that reduces the dimension from 32 to 16.
- **out_layer** is a fully connected layer that reduces the dimension from 16 to 2, which is suitable for a 2-class classification problem.

Forward Pass:

1. **First Graph Convolution:**

$$\mathbf{H}_1 = \text{ReLU}(\text{conv}_1(\mathbf{X}, \mathbf{A}))$$

   - $\mathbf{H}_1$ is the output after applying the first graph convolution layer and ReLU activation.
   - This transforms the node features from input_features to 32 dimensions.

2. **Second Graph Convolution:**

$$\mathbf{H}_2 = \text{ReLU}(\text{conv}_2(\mathbf{H}_1, \mathbf{A}))$$

   - $\mathbf{H}_2$ is the output after the second graph convolution layer and ReLU activation.
   - This retains the 32-dimensional feature representation.

3. **Global Sum Pooling:**

$$\mathbf{h}_{\text{graph}} = \text{global\_sum\_pool}(\mathbf{H}_2, \text{batch\_mat})$$

- $\mathbf{h}_{\text{graph}}$ is the result of summing the node features across each graph in the batch, producing a graph-level feature vector.

4. **Fully Connected Layers:**

$$\mathbf{h}_{\text{fc}} = \text{ReLU}(\mathbf{h}_{\text{graph}} \times \mathbf{W}_3 + \mathbf{b}_3)$$

- This applies a linear transformation followed by ReLU activation, reducing the feature size from 32 to 16.

5. **Output Layer:**

$$\mathbf{output} = \text{Softmax}(\mathbf{h}_{\text{fc}} \times \mathbf{W}_4 + \mathbf{b}_4)$$

- The final output is obtained by applying another linear transformation followed by a softmax activation, which converts the output into probabilities for a 2-class classification.

3. `global_sum_pool(X, batch_mat)`

Mathematics Behind Global Sum Pooling

Let:

- X be the node feature matrix, with dimensions $N \times F$, where N is the total number of nodes across all graphs in the batch, and F is the number of features per node.
- batch_mat be a matrix that helps identify which nodes belong to which graph in a batch. Its dimensions are $B \times N$, where B is the number of graphs in the batch.

Two Cases:

1. **When `batch_mat` is `None` or a 1D vector:**
   In this case, the operation performed is:

$$\text{pooled\_X} = \sum_{i=1}^{N} X_i$$

   Here, X_i refers to the i-th row (node) of the matrix X. This summation gives a single vector of size $1 \times F$, which is the sum of the features across all nodes, producing a global representation of the entire graph (or the entire batch treated as one graph).
2. **When `batch_mat` is a 2D matrix:**
   Here, the pooling operation is:

$$\text{pooled\_X} = \text{batch\_mat} \times X$$

- batch_mat is a binary matrix where:
  - $\text{batch\_mat}[b, i] = 1$ if node i belongs to graph b.
  - $\text{batch\_mat}[b, i] = 0$ otherwise.
- The resulting matrix pooled_X will have dimensions $B \times F$, where each row represents the sum of the node features for each graph in the batch.

Mathematical Insight:

- Matrix multiplication here aggregates node features within each graph separately. For each graph b, it sums the features of all nodes i belonging to that graph:

$$\text{pooled\_X}[b, :] = \sum_{i \text{ where } \text{batch\_mat}[b,i]=1} X_i$$

This operation effectively creates a graph-level feature vector for each graph in the batch.

4. `get_batch_tensor(graph_sizes)`

Mathematics Behind Batch Matrix Creation

Let:

- graph_sizes be a list where each element g_j represents the number of nodes in graph j in the batch.
- B be the number of graphs in the batch, i.e., the length of graph_sizes.
- N be the total number of nodes across all graphs, i.e., $N = \sum_{j=1}^{B} g_j$.

Batch Matrix Construction:

- The batch matrix batch_mat is a $B \times N$ matrix where:
  - Each row corresponds to a graph.
  - Each column corresponds to a node.

For graph j, which starts at node index start_j and ends at stop_j:

$$\text{batch\_mat}[j, i] = \begin{cases} 1 & \text{if } \text{start}_j \leq i < \text{stop}_j \\ 0 & \text{otherwise} \end{cases}$$

- $\text{start}_j = \sum_{k=1}^{j-1} g_k$ is the cumulative sum of node counts up to the previous graph.
- $\text{stop}_j = \text{start}_j + g_j$.

Mathematical Insight:

- The matrix batch_mat ensures that any operation involving this matrix will respect the boundaries of each graph. For example, summing node features using $\text{batch\_mat} \times X$ will sum features only within the nodes of each graph, producing a $B \times F$ matrix where each row is the sum of features for one graph.

5. `collate_graphs(batch)`

Mathematics Behind Graph Collation

This function aggregates various graph-related matrices and vectors into a batch-level representation.

Let:

- A_j be the adjacency matrix of the j-th graph, with dimensions $g_j \times g_j$.
- X_j be the node feature matrix of the j-th graph, with dimensions $g_j \times F$.
- y_j be the label vector of the j-th graph, with dimensions C (e.g., for a classification task).

Collating Process:

1. **Batch Adjacency Matrix batch_adj:**
   - This is a block diagonal matrix where each block is the adjacency matrix A_j of graph j.
   - The combined adjacency matrix batch_adj is of size $N \times N$, where $N = \sum_{j=1}^{B} g_j$.
   - Mathematically, batch_adj is a block matrix:

$$\text{batch\_adj} = \text{diag}(A_1, A_2, \ldots, A_B)$$

     - Each A_j is placed in its corresponding diagonal block, and all off-diagonal blocks are zero matrices.
2. **Batch Feature Matrix feat_mats:**
   - This is simply the vertical concatenation of the node feature matrices:

$$\text{feat\_mats} = \begin{bmatrix} X_1 \\ X_2 \\ \vdots \\ X_B \end{bmatrix}$$

     - Resulting in a matrix of size $N \times F$.
3. **Batch Label Vector labels:**

- The labels are concatenated into a single vector (or matrix, depending on the task):

$$\text{labels} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_B \end{bmatrix}$$

4. **Batch Matrix batch_mat:**
   - Created using the `get_batch_tensor` function, this $B \times N$ matrix indicates which nodes belong to which graph.

Final Output:

- The function returns a dictionary containing:
  - batch_adj: The block diagonal adjacency matrix.
  - feat_mats: The combined node feature matrix.
  - labels: The concatenated labels.
  - batch_mat: The batch matrix indicating node membership.

Mathematical Insight:

- The key mathematical operation here is the block diagonal construction of the adjacency matrix and the concatenation of the feature matrices. This setup allows the GNN to process each graph separately within the batch, but in a single forward pass, which is efficient and scales well with large datasets.

Summary:

- **Graph Convolution:** Combines node features from neighboring nodes, applying learnable transformations through weight matrices.
- **Activation Functions:** Non-linear transformations (ReLU) are applied after each convolution to introduce non-linearity.
- **Global Pooling:** Aggregates node-level features into graph-level features, essential for tasks like graph classification.
- **Fully Connected Layers:** Further process the graph-level features to produce the final output, suitable for classification tasks.
- **Matrix Multiplication (for Sum Pooling):** Used to aggregate node features based on graph membership.
- **Block Diagonal Matrices:** Used to combine adjacency matrices while keeping each graph's structure separate.

- **Concatenation:** Used to merge node features and labels from multiple graphs into single batch-level tensors.