CS 760: Machine Learning Spring 2024

Graph Neural Networks

AUTHORS: Parla Surendra Mani Kumar

DO NOT POLLUTE! AVOID PRINTING, OR PRINT 2-SIDED MULTIPAGE.

Graph Neural Networks (GNNs) are powerful tools for graph-structured data, enabling tasks like node and graph classification, and link prediction. GNNs can capture complex relationships and patterns. We give an overview of GNNs and an abstraction to represent different flavours of GNN like Graph Convolutional Network and Graph Attention Networks. Finally, an example, with code and explanations, is also given.

1 Introduction

Graph Neural Networks (GNN) can be thought as a subset of neural networks which are modelled as graphs. Graphs are non-euclidean data structures where objects are represented as nodes and their relationship is captured as edges. GNN has vast uses cases because there are many datasets which are naturally expressed as graphs, like social media networks, molecules, DNA, internet traffic, knowledge graphs and more.

Historically, Convolutional Neural Networks (CNN) have been highly successful in extracting the spacial features from images and n-D euclidean data structures, but they struggle to capture the relationship present between the nodes as graph can have irregular structures and unequal number of neighbours making convolution operation difficult.

GNNs can be broadly categorized into 4 types:

- Recurrent GNN
- Graph Convolutional Neural Networks
- Graph Autoencoders (Attension Networks)
- Spatial-temporal Graphs (Recursive Networks)

2 Basic Graph Concepts

A graph G = (V, E) consists of a set of nodes (vertices) V and a set of edges E

Each edge connects two nodes and represents the relationship between the two nodes. Graphs can be directed (edges have a direction) as shown in Figure 1 (page 2) or undirected (edges have no direction) as shown in Figure 2 (page 2). Additionally, graphs can be weighted, where edges have an associated weight representing the strength of the connection between the nodes.

Graphs are represented using the adjacency matrix A where each element in the matrix A_{ij} represent the weight of edge from $node_i$ to $node_j$.

A degree of node can be defined as the number of neighbour nodes $N(v)$.

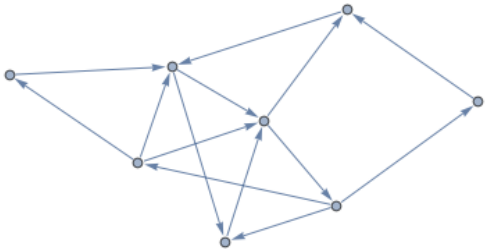

Figure 1: Directed Graph

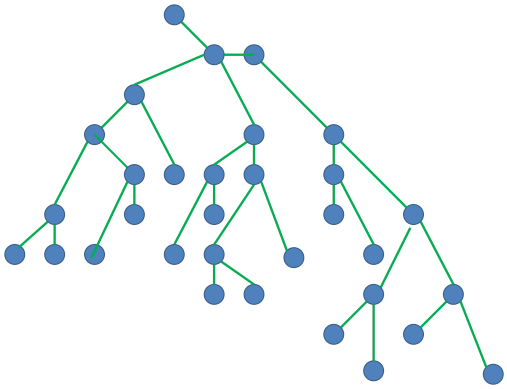

Figure 2: Undirected Graph

Note : *Nodes* and *edges* are very general terms and can mean a lot of things. For example, a node itself could be a Deep Neural Network and an edge could represent a producer-consumer relationship, evolutionary relationship, a chemical bond, a dataflow relationship or even a mechanical connection.

2.1 Modelling Using Graphs

Graphs allow for the most flexible and accurate representation of a lot of modelling problems. In general, engineering involves the creation of a model with certain (reasonable) assumptions. The objective of creating a model is to ultimately utilise the model for its predictive capabilities. Thus, the modelling framework or approach we take, influences both the way we see a problem and also its solution. A good number of problems that we are trying to solve can be represented as a graph model -

1. Social Media Networks - where every individual is modelled as node and their relation is the edge between them
2. Molecules - Every atom is a node and the bond between them is an edge
3. Proteins - Proteins are represented as a graph (G) of Aminoacids (N) connected to each other by an edge (E)

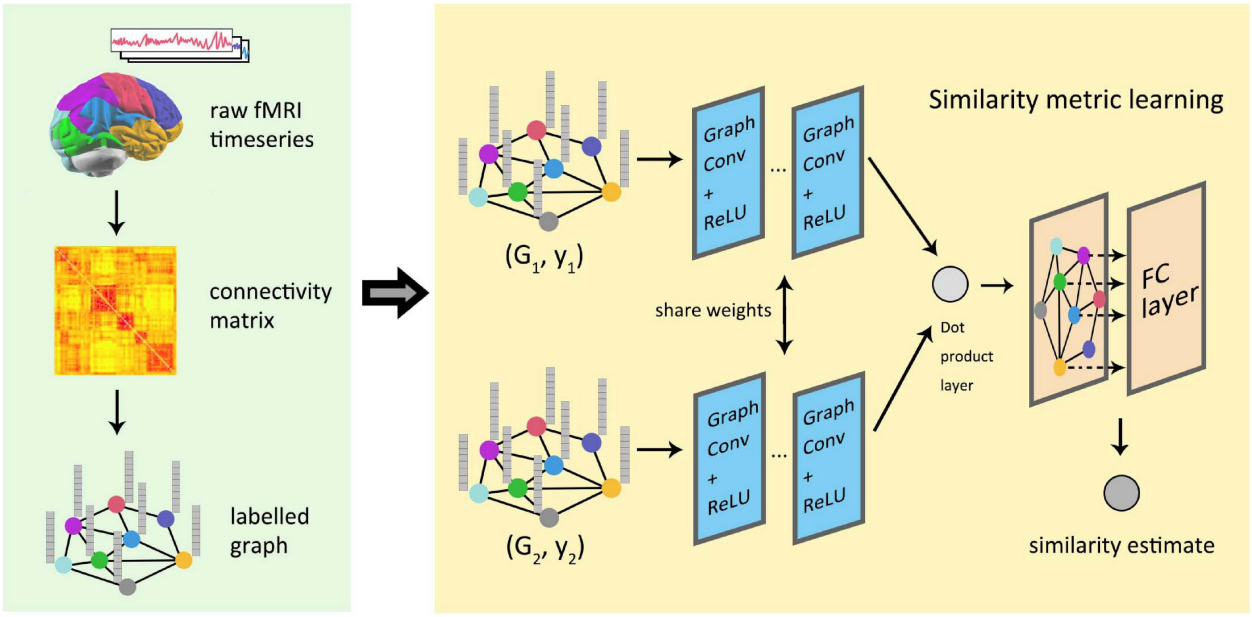

Figure 3: Similarity measure learning of graph neural network in functional brain graphs. The information contained in the fMRI image was integrated into the graph through the graph partition and connection matrix. Two graphs were, respectively, used as the input of the two graph convolutional networks with sharing parameters, and the output of the network was combined by the inner product layer. Finally, a fully connected layer outputs the estimated similarity between the graphs [1]

3 Graph Neural Network

The function of the graph convolution layer in a GNN is very similar to that of the convolution layer in a CNN, in that the objective is to derive a relationship amongst the nodes. Then, the output from the graph layer is fed into, say, an inference layer for further processing. In supervised learning, we are effectively trying to find a function f(x) which models the output y. The function f, is a neural network which incorporates the graph information.

A GNN layer makes the nodes learn about their neighbours (and all extended neighbours, eventually) over time, by changing each nodes' hidden state. The number of times the hidden state needs to updated is application specific.

Figure 3 (page 3), shows a real world application of a GNN. It also shows how the GNN layer is separate from the classification layer (FC Layer), as described above.

3.1 Mathematics Behind a GNN

A single GNN layer consists of two important parts -

1. **Aggregation** - The information from the neighbouring nodes ($N(v)$) us transformed and aggregated. Eg : SUM, MAX, AVERAGE.
2. **Update** - The embeddings of the $node(v)$ is updated using the aggregated information from nodes and linearly transformed current node embeddings. Eg : SUM, CONCAT, MLP (Multi Layer Perceptron)

The hidden state of a node at a particular layer l is

$$h_v^0 = X_v$$

$$h_v^{(l+1)} = \sigma\Big(W_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}\Big)$$

The hidden state of all the nodes are transformed using a linear transformation of weight matrix W_l and aggregated using weighted average. The current state of the node is also linearly transformed using a weight matrix B_l. σ is a non linear function, such as the sigmoid function.

Matrix Representation

Let D be a diagonal matrix where

$$D_{v,v} = Degree(v) = |N(v)|$$

$$D_{v,v}^{-1} = \frac{1}{|N(v)|}$$

The term

$$\sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|}$$

can be written as

$$D^{-1} A H^{(l)}$$

Therefore,

$$H^{(l+1)} = \sigma(D^{-1} A H^{(l)} W_l^T + H^{(l)} B_l^T)$$

Where $H^{(\ell)}$ is the hidden state matrix at layer ℓ and A is the adjacency matrix of the graph.
The parameters in the weight matrices W and B are tuned by back propagation using MSE (Mean Square Error) or Log Likelihood depending on the ML task.

$$\Theta_{\ell,t} = \Theta_{\ell,t-1} - \eta \sum_{i \in \Omega_t} \nabla_i \Theta_{\ell,t-1}$$

3.2 Classification

As a reminder, the GNN layers are used to extract information on how the nodes relate to each other according to our modelling of the problem as a graph. This information is then, commonly, fed into a classification layer.

Three types of classification exist -

1. Node Classification - here, we try to ascertain what a particular node could be, provided that we have a graph with a good number of nodes labelled

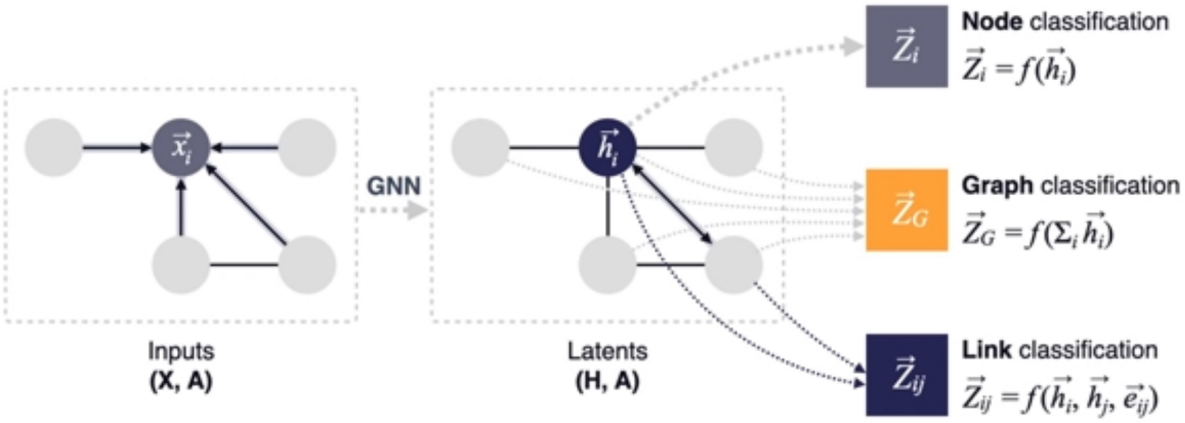

Figure 4: Three Types of GNN Classfication [2]

2. Graph Classification - here, we try to ascertain what the graph as a whole could mean. If we consider an image as a graph, then this would translate to image classification
3. Link Classification - here, we try to find out if there could be a relationship between two nodes, provided we already have a few links between nodes on the graph

3.3 General Recipe

So, the basic recipe for a GNN is

1. Construct the initial graph structure to model the data as nodes and the relationship between data as edges
2. Based on the task i.e, node level, edge level as described earlier, create the GNN model's classification layer
3. Define the loss function according to the context. Eg: log likelihood, root mean square error etc.
4. Apply gradient descent and optimize the weight parameters by minimizing the loss function

4 Message Passing GNN

The message passing GNN layer captures the essence of a GNN. It is the most potent form of a GNN, but it is very expensive to implement - in terms of memory and processing. Thus, they are limited to smaller graphs.

In message passing GNNs, each node first calculates its message and sends it to its neighbours. Each node then aggregates - the aggregation function depending on the implementation - all the messages that it received from its neighbours. The state of that node is then updated. These happen in lock step across the graph i.e. the state of each node in the graph is updated over time, in each "clock tick". An overview is shown in Figure 5 (page 6).

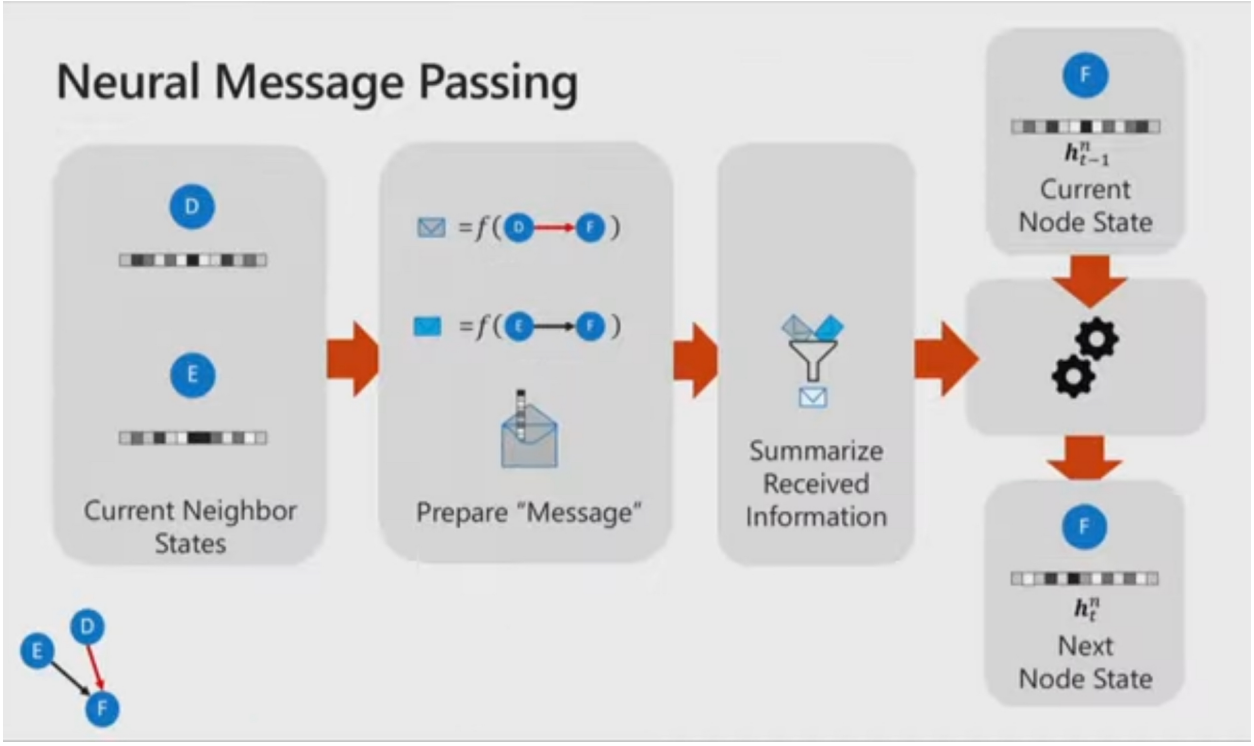

Figure 5: Dataflow in a Message Passing GNN Layer [3]

Note : In certain graph, the edges can be of different types. The edges themselves influence the message that is generated between two nodes

The change in a node's state can be described using the following equation -

$$h_t^n = q(h_{t-1}^n, \bigcup_{\forall n_j : n_j \xrightarrow{k} n} f_t(h_{t-1}^n, k, h_{t-1}^{n_j}))$$

where, h_t^n is the state of node n at time t. q is a non-linear transformation function. $\bigcup$ is any commutative and permutation invariant operation to combine the messages. f_t is the function that produces the messages according to the previous state of node n_j and node n and edge type k. Here, f_t can be dependent on time, but not necessarily.

5 Graph Convolution Networks

In a GCN the neighbour embeddings are normalized and summed. These embeddings are then transformed using a weight matrix and a non linear function is applied. See Figure 6 (page 7).

$$h_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in N(u) \cup \{u\}} \frac{h_v}{\sqrt{|N(u)||N(v)|}}\right)$$

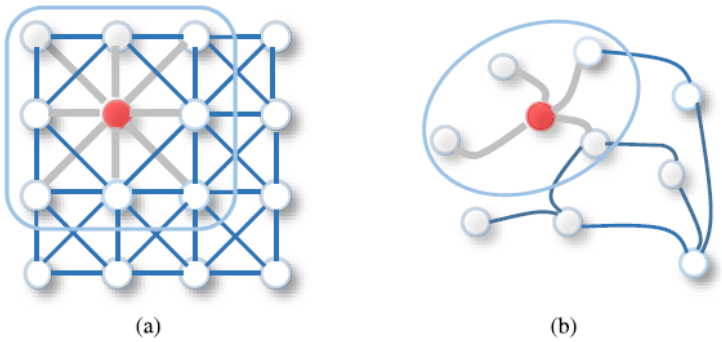

Figure 6: 2D convolution vs Graph Convolution

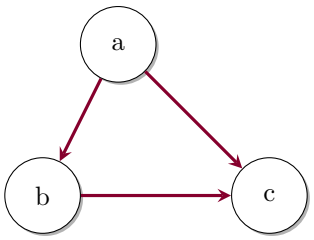

Figure 7: Example Graph

This can be expressed in matrix representation as

$$x *_G g\theta = \theta(I_n + D^{-\frac{1}{2}}AD^{-\frac{1}{2}})x$$

where D is diagonal degree matrix, A is adjacency matrix and x is the input

An example of nodes in a graph and their adjacency matrix is shown in Figure 7 (page 7).

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}, N = \begin{bmatrix} a \\ b \\ c \end{bmatrix}, A^T.N = \begin{bmatrix} 0 \\ a \\ a+b \end{bmatrix}$$

6 Other Flavours of GNN

6.1 GraphSage

GraphSage proposes the use of different aggregating functions like weigthed avegage, Multi Layer Perceptron on all the neighbouring nodes or even a LSTM.

The aggregated messsages are then concatenated with the node embeddings.

$$m_N^{(u)} = \text{MLP}\theta\left(\sum v \in N(u)\text{MLP}_\phi(h_v)\right)$$

Other interesting idea here is ℓ_2 normalization of hidden state in all the layer which seems to improve performance.

$$h_v^{(\ell)} = \frac{h_v^{(\ell)}}{||h_v^{(\ell)}||_2} \quad where, \quad ||u||_2 = \sqrt{\sum_i u_i^2}$$

6.2 Graph Attention Networks

The aggregate function in the GAT is attention operation. All the neighbouring node embeddings are multiplied by weights i.e, attention coefficients. Comparing this to other flavours where this weight is just the inverse of the degree of the node, attention which is learnt from the network is great replacement for this weight.

$$m_v^{(u)} = \sum_{v \in N(u)} \alpha_{u,v} h_v, \quad \alpha_{u,v} = \frac{\exp(a^T[Wh_u \oplus Wh_v])}{\sum_{v' \in N(u)} \exp(a^T[Wh_u \oplus Wh_{v'}])}$$

Where a is linear transformation matrix and the operation $\oplus$ can either be concatenation or a dot product or some distance metric like cosine similarity.

7 Over-Smoothing Problem

Unlike CNN where the performance increase along with the number of layers, GNN with many layers perform poorly. In a highly connected graph network every node is influenced by many neighbours. So, with huge number of layers all the node embeddings will be saturated and become indistinguishable to each other. This is called over smoothing effect. The number layers to reach this saturation in a highly connected graph could be as low as O(log(|v|))

So, the solution for this problem is instead of increasing the number of GNN layers (k) we can add complexity in the aggregation function with the use of Multi Layer Perceptron (MLP) as in Graph Sage model. This would help you aggregate the spacial information with fewer GNN layers.

8 Example

We shall now look at an example, with explanations and code.

> **Objective :** To find the solubility of a molecule in a solvent.

We will be using the MoleculeNet [4] dataset's ESOL dataset.

> "ESOL is a small dataset consisting of water solubility data for 1128 compounds. The dataset has been used to train models that estimate solubility directly from chemical structures (as encoded in SMILES strings)."

Each molecule is represented as a graph, with the nodes representing the atoms. Each node (atom) has a feature vector associated with it. Since the molecule itself is represented as a graph and we would like to find the solubility of the molecule, we are naturally going to make graph level predictions. The GNN architectures vary for different levels of predictions (see section 3.2).

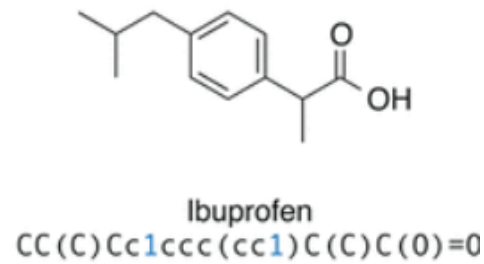

Figure 8: Smile String of Ibuprofen

Since this is a graph level task a Pooling layer combines the node representations of the graph into one representation so that every graph is converted to a vector embedding.

3 simple GCN layers were applied followed by global Max and global Mean pooling of the graph concatenated.

See Appendix in page 10 for some example code.

References

[1] X.-M. Zhang, L. Liang, L. Liu, and M.-J. Tang, "Graph Neural Networks and Their Current Applications in Bioinformatics," *Frontiers in Genetics*, vol. 12, p. 690049, Jul. 2021. [Online]. Available: https://www.frontiersin.org/articles/10.3389/fgene.2021.690049/full

[2] "Intro to graph neural networks (ML Tech Talks)." [Online]. Available: https://www.youtube.com/watch?v=8owQBFAHw7E

[3] "An Introduction to Graph Neural Networks: Models and Applications." [Online]. Available: https://www.youtube.com/watch?v=zCEYiCxrL$_0$

[4] "torch_geometric.datasets." [Online]. Available: https://pytorch-geometric.readthedocs.io/en/latest/modules/datasets.html

[5] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666651021000012

[6] Z. a. Liu, *Introduction to Graph Neural Networks*. 1st ed. 2020. Cham : Springer International Publishing : Imprint: Springer, 2020., 2020. [Online]. Available: https://search.library.wisc.edu/catalog/9913707926802121

[7] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, pp. 1025–1035, event-place: Long Beach, California, USA.

[8] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A Comprehensive Survey on Graph Neural Networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, pp. 4–24, 2021.

[9] "Stanford CS224W: Machine Learning with Graphs | 2021 | Lecture 1.1 - Why Graphs." [Online]. Available: https://www.youtube.com/watch?v=JAB$_p$$lj2rbAlist = PLoROMvodv4rPLKxIpqhjhPgdQy7imNkDn$

Appendix

```
import torch
from torch.nn import Linear
import torch.nn.functional as F
from torch_geometric.nn import GCNConv, TopKPooling, global_mean_pool
from torch_geometric.nn import global_mean_pool as gap, global_max_pool as gmp
embedding_size = 64

class GCN(torch.nn.Module):
    def __init__(self):
        # Init parent
        super(GCN, self).__init__()
        torch.manual_seed(50)

        # GCN layers
        self.initial_conv = GCNConv(data.num_features, embedding_size)
        self.conv1 = GCNConv(embedding_size, embedding_size)
        self.conv2 = GCNConv(embedding_size, embedding_size)
        self.conv3 = GCNConv(embedding_size, embedding_size)

        # Output layer
        self.out = Linear(embedding_size*2, 1)

    def forward(self, x, edge_index, batch_index):
        # First Conv layer
        hidden = self.initial_conv(x, edge_index)
        hidden = F.tanh(hidden)

        # Other Conv layers
        hidden = self.conv1(hidden, edge_index)
        hidden = F.tanh(hidden)
        hidden = self.conv2(hidden, edge_index)
        hidden = F.tanh(hidden)
        hidden = self.conv3(hidden, edge_index)
        hidden = F.tanh(hidden)

        # Global Pooling (stack different aggregations)
        hidden = torch.cat([gmp(hidden, batch_index),
                            gap(hidden, batch_index)], dim=1)

        # Apply a final (linear) classifier.
        out = self.out(hidden)

        return out, hidden

model = GCN()
print(model)

from torch_geometric.data import DataLoader
```

```
import warnings
warnings.filterwarnings("ignore")

# Root mean squared error
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0007)

# Use GPU for training
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = model.to(device)

# Wrap data in a data loader
data_size = len(data)
NUM_GRAPHS_PER_BATCH = 64
loader = DataLoader(data[:int(data_size * 0.8)],
                    batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True)
test_loader = DataLoader(data[int(data_size * 0.8):],
                         batch_size=NUM_GRAPHS_PER_BATCH, shuffle=True)

def train(data):
    # Enumerate over the data
    for batch in loader:
      # Use GPU
      batch.to(device)
      # Reset gradients
      optimizer.zero_grad()
      # Passing the node features and the connection info
      pred, embedding = model(batch.x.float(), batch.edge_index, batch.batch)
      # Calculating the loss and gradients
      loss = loss_fn(pred, batch.y)
      loss.backward()
      # Update using the gradients
      optimizer.step()
    return loss, embedding

print("Starting training...")
losses = []
for epoch in range(2000):
    loss, h = train(data)
    losses.append(loss)
    if epoch % 100 == 0:
      print(f"Epoch {epoch} | Train Loss {loss}")
```