Assignment 3: Edge prediction using a customized Graph Isomorphism Network (GIN) layer

ELTE University

by

Jacopo Manenti

Student Name	Student Number
Jacopo Manenti	G1R5RZ

Professor:	Guettala Walid
Faculty:	Computer Science, Budapest

0.1. Abstract

In this paper, we detail the implementation of a Graph Isomorphism Network (GIN) from scratch focusing on edge-level tasks. We have only used and intentionally avoiding any pre-existing implementations from libraries such as PyTorch Geometric (Chapter 0.3). The model incorporates Custom message-passing mechanism using MLPs, Skip connections for improved gradient flow, Layer normalization for training stability, Dropout for regularization (Chapter 0.4).

Our implementation targets link prediction tasks on the Cora dataset, evaluating both **inductive** and **transductive** learning approaches through `RandomLinkSplit` and `train_test_split_edges` respectively. The Cora dataset represents a citation network where nodes are papers described by feature vectors (e.g., bag-of-words representations) and associated categories, with edges indicating citations (Chapter 0.2).

We have used different dataset splitting strategies, for example employing an 80/20/20 ratio for training, validation, and testing (see Table 1). Notably, the inductive learning approach demonstrated superior performance, achieving an 27% improvement in AUC compared to the transductive method and 22% in Accuracy (Table 2).

We tested various configurations of the GIN model with different splitting ratios, ultimately selecting the best-performing models, as detailed in Chapter 0.6. Performance analysis, evaluated using metrics like AUC-ROC and Precision@k, highlights the superiority of the inductive splitting strategy, likely due to its ability to effectively manage unseen graph structures—an essential advantage for real-world applications where new edges frequently emerge.

0.2. Dataset

The Cora dataset[3] consists of 2,708 scientific publications connected through citation relationships, forming an undirected graph with 5,429 unique citations (10,556 edges when counting both directions). Each publication is represented as a node with 1,433 binary features indicating word presence/absence from a fixed dictionary.

The Cora dataset is a widely recognized benchmark for graph learning tasks, making it an excellent choice for evaluating edge prediction in both inductive and transductive learning contexts.

For **inductive learning**, the ability to generalize to new, unseen nodes or subgraphs is paramount. The Cora dataset facilitates robust evaluation of this capability by enabling graph splits where test nodes or edges are completely disjoint from the training set.

Furthermore, the dataset offers an optimal **complexity**, with manageable graph size, multiple node classes, and feature-rich attributes. This balance makes it computationally feasible while providing meaningful insights into model performance. This efficiency is particularly crucial for edge prediction tasks, which involve operations like matrix multiplication

$$[n_{\text{nodes}}, \text{features}] \times [n_{\text{nodes}}, \text{features}]^T.$$

Since we used Google Colab, large datasets can easily exceed memory limits on the platform. Cora's moderate size mitigates this issue, enabling smoother experimentation without significant hardware bottlenecks.

The graph structure is characterized by an average node degree of 3.90, with no isolated nodes or self-loops. The dataset contains complete edge information through an `edge_index` representing citation relationships between papers. While node features are present, the dataset contains no edge features (`num_edge_features = 0`).

Note: The dataset provides raw binary features and has not been normalized using `NormalizeFeatures` *when imported from the PyTorch dataset, meaning that each feature retains its original binary representation.*

0.2.1. Inductive splitting

Inductive learning in link prediction focuses on enabling models to generalize to unseen graph structures, making it well-suited for dynamic networks where new nodes and edges frequently appear.

In our implementation, we have used an inductive approach by splitting the Cora dataset edges into training, validation, and testing sets with the `RandomLinkSplit` function[1]. This function ensures balanced positive and negative edge samples for all splits. For example, with 10.556 total edges in the graph, and validation and test ratios set to 20% and 20% respectively, RandomLinkSplit generates 1.055 positive and 1.055 negative validation edges (2.110 total) and 1.055 positive and 1.055 negative test edges (2.110 total). The remaining 3.168 positive and 3.168 negative training edges (6.336 total) form the training set. Negative edges were generated explicitly for training using a negative sampling function, which ensures the model learns to distinguish true connections from false ones.

The function also ensures each split retains appropriate ratios while maintaining graph integrity. For example, in the message passing, each edge index is passed in pairs (e.g., [0, 1]) and is duplicated for undirected graphs, resulting in 6.336 edges in training (3.168 positive × 2), 6.336 edges in validation (equal to the training), and 8.446 edges in testing (training message passing (edge_index) + validation message passing undirected (1.055)). This inductive approach helps evaluate the model on unseen edge scenarios, mimicking real-world applications like predicting interactions in a social network or emerging links in recommendation systems.

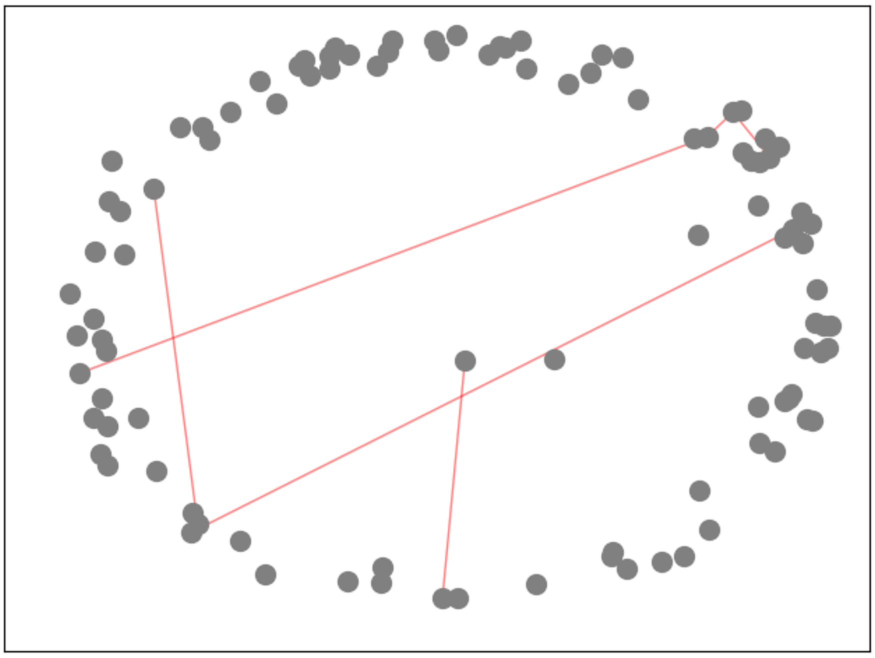


Figure 1: The figure illustrates a subgraph (100 nodes) from the Cora dataset after applying `RandomLinkSplit`. The edges are color-coded as follows: gray represents positive training edges, red denotes negative training edges, blue indicates positive validation edges, and green signifies positive test edges.

0.2.2. Transductive splitting

Transductive learning, by contrast, trains models using the complete graph structure, leveraging all nodes and their relationships to predict missing edges within the same graph. This method is ideal for static networks where all nodes are known, such as citation graphs. We used the the deprecated `train_test_split_edges` function[2] available in `torch_geometric.utils`.

The `train_test_split_edges` function is a method in PyTorch Geometric for splitting a graph's edges into training, validation, and test sets, including both positive and negative edges. It operates on a graph represented by a `torch_geometric.data.Data` object, which contains the graph's nodes, edges, and optionally edge attributes. The function is primarily used for transductive learning scenarios, where all nodes are known during training, and the goal is to predict missing edges within the same graph. The splits are determined by the specified ratios for validation and test edges (`val_ratio` and `test_ratio`).

Once the `train_test_split_edges` function is called on a `torch_geometric.data.Data` object, it processes the original graph and adds several new attributes to the `Data` object, which are crucial for link prediction tasks. These new attributes represent the split edges, including both positive and negative edges for training, validation, and testing:

- `val_pos_edge_index`: This attribute contains the edge indices (in the form of a tensor with shape $[2, \text{num\_edges}]$) of the positive edges in the validation set. These are the edges that exist in the graph, used to validate the model's link prediction performance during training.
- `test_pos_edge_index`: Similar to the validation set, this attribute stores the indices of positive edges in the test set. These edges will be used to evaluate the model after training.
- `train_pos_edge_index`: This contains the positive edges used for training. These edges are split from the original set of edges, ensuring that only a portion is used for training, while the rest are kept for validation and testing.
- `train_neg_adj_mask`: This is a binary mask of size $[\text{num\_nodes}, \text{num\_nodes}]$ representing the negative adjacency matrix for training. It indicates which node pairs are **not** connected in the graph, helping the model distinguish between actual (positive) and non-existent (negative) edges.
- `val_neg_edge_index`: These are the indices of the negative edges for validation. Similar to `val_pos_edge_index`, they represent node pairs that do not have an edge in the original graph and are used to evaluate the model's ability to distinguish true edges from false ones.
- `test_neg_edge_index`: This contains the indices of negative edges in the test set, which are used to test the model's performance on unseen negative samples after training.

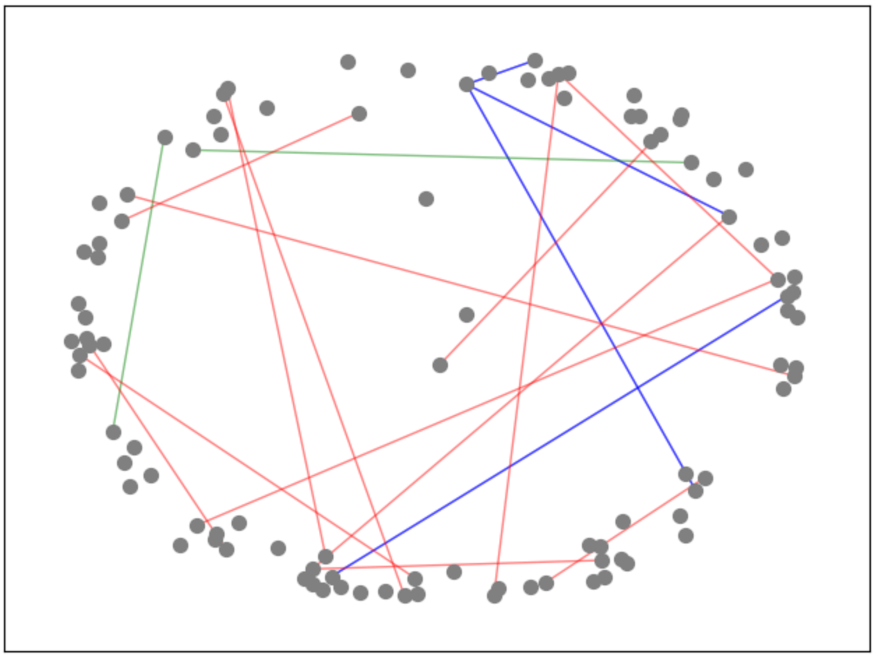


Figure 2: The figure illustrates a subgraph (100 nodes) from the Cora dataset after applying `train_test_split_edges`. The edges are color-coded as follows: gray represents positive training edges, red denotes negative training edges, blue indicates positive validation edges, and green signifies positive test edges.

0.3. GIN Implementation Analysis

The implementation of the Graph Isomorphism Network (GIN) builds upon PyTorch Geometric's `MessagePassing` class, incorporating customized dimensional transformations and parameter management. In this section, we detail the mathematical formulation of the GIN layer. The corresponding code can be accessed on GitHub.

0.3.1. GIN Components Analysis

Message Function

The message function implements a simple identity mapping:

$$\text{message}(x_j) = x_j, \quad x_j \in \mathbb{R}^{F_{in}}$$

This design choice follows GIN's theoretical finding that simple summation of raw neighbor features is sufficient for maximum discriminative power.

Epsilon Parameter

The learnable parameter ϵ adjusts the importance of the central node:

$$\epsilon = \begin{cases} \text{learnable} \in \mathbb{R}^1 & \text{if train\_eps = True} \\ \text{fixed} = \epsilon_0 & \text{otherwise} \end{cases}$$

Where ϵ_0 is the initial value (default 0.0). This parameter helps balance self-information against neighborhood information.

Aggregation Mechanism

The aggregation uses summation over the neighborhood $\mathcal{N}(i)$:

$$\text{aggregate}(\{x_j | j \in \mathcal{N}(i)\}) = \sum_{j \in \mathcal{N}(i)} x_j$$

Summation ensures the layer maintains its injective properties, crucial for the network's discriminative power.

Self-Connection

The self-connection adds weighted self-features:

$$\text{self\_connect}(x_i, \text{agg}) = \text{agg} + (1 + \epsilon) x_i$$

Where:

- agg is the aggregated neighbor features
- $(1 + \epsilon)$ allows adaptive self-feature weighting
- x_i represents the central node's features

Transform Operation

Final feature transformation through an MLP:

$$\text{transform}(h) = \text{MLP}_\theta(h) : \mathbb{R}^{F_{in}} \to \mathbb{R}^{F_{out}}$$

The MLP applies non-linear transformations to combined neighborhood and self-features, producing the final node representation.

0.3.2. Input Dimensions and Parameters

Let N be the number of nodes, E the number of edges, F_{in} the input feature dimension, and F_{out} the output feature dimension:

- **Input Features**: **X** $\in \mathbb{R}^{N \times F_{in}}$
- **Edge Index**: **edge_index** $\in \mathbb{R}^{2 \times E}$
- **Neural Network** (nn): MLP : $\mathbb{R}^{F_{in}} \to \mathbb{R}^{F_{out}}$
- **Epsilon** (ϵ): Either fixed or trainable parameter:

$$\epsilon = \begin{cases} \text{Parameter(empty}(1)) & \text{if train\_eps = True} \\ \text{Buffer(empty}(1)) & \text{otherwise} \end{cases}$$

- **Aggregation**: Default 'add' for message combination

0.3.3. Forward Pass with Dimensional Analysis

The forward propagation performs these transformations:

1. **Message** : $x_j \in \mathbb{R}^{F_{in}} \to \mathbb{R}^{F_{in}}$
2. **Aggregate** : $\sum_{j \in \mathcal{N}(i)} x_j : \mathbb{R}^{N \times F_{in}} \times \mathbb{R}^{2 \times E} \to \mathbb{R}^{N \times F_{in}}$
3. **Self-Connection** : $\text{out} + (1+\epsilon)x_r : \mathbb{R}^{N \times F_{in}} \to \mathbb{R}^{N \times F_{in}}$
4. **Transform** : $\text{nn}(\text{out}) : \mathbb{R}^{N \times F_{in}} \to \mathbb{R}^{N \times F_{out}}$

Parameter Management

The `reset_parameters` method handles initialization:

- Resets MessagePassing parameters: $\mathbb{R}^{F_{in} \times F_{out}}$
- Initializes neural network weights using reset function
- Sets epsilon to initial value: $\epsilon \in \mathbb{R}^1$

The implementation maintains dimensional consistency throughout message passing before the final neural network transformation, ensuring proper feature propagation across the graph structure.

0.4. Model

In this section, we describe the model that incorporates the handcrafted GIN to address the edge prediction task. The model follows an encoder-decoder architecture and combines both fully connected (FC) layers and GIN convolutional layers. The **encoder**, consisting of the GIN layers, learns node representations by applying graph convolutions and aggregating information from neighboring nodes. This is preceded by a projection layer that transforms the raw node features into the `hidden_channels` space, enabling residual connections. The **decoder** then utilizes the learned node embeddings to perform edge prediction. The model can handle either link prediction or edge classification, depending on the task. In link prediction, the model predicts the existence of edges between nodes, whereas in edge classification, it assigns labels to edges based on the learned representations.

0.4.1. Model Architecture

Constructor (`__init__`)

The model's constructor defines several layers and parameters necessary for the learning task, as described below:

- `in_channels` - The number of input features for each node in the graph.
- `hidden_channels` - The number of hidden units in the intermediate layers.
- `out_channels` - The number of output features per node (or embedding dimension).
- `num_classes` - The number of classes for classification tasks (default: None).
- `link_pred` - A boolean flag indicating whether link prediction or edge classification is being performed.
- `is_batch` - A flag that determines whether batch normalization should be applied.

Layers Overview

Fully Connected Layers:

- `nn1`: A sequence of fully connected layers with ReLU activations and batch normalization.
  - `Linear(in_channels, hidden_channels)`: This layer takes the input feature vector of size $[N, \texttt{in\_channels}]$ where N is the number of nodes and transforms it to a hidden representation of size $[N, \texttt{hidden\_channels}]$.

  - `BatchNorm1d(hidden_channels)`: Normalizes the hidden representation with size $[N, \texttt{hidden\_channels}]$.
  - `Linear(hidden_channels, hidden_channels)`: A second fully connected layer maintaining the same size $[N, \texttt{hidden\_channels}]$, followed by ReLU activation.
- `nn2`: Another sequence of fully connected layers similar to `nn1`, but with the output size being `out_channels`.
  - `Linear(hidden_channels, hidden_channels)`: Transforms the hidden representation of size $[N, \texttt{hidden\_channels}]$ to another hidden representation of the same size.
  - `BatchNorm1d(hidden_channels)`: Applies batch normalization.
  - `Linear(hidden_channels, out_channels)`: The final fully connected layer transforms the hidden representation to an output size of $[N, \texttt{out\_channels}]$.

GIN Convolutional Layers: The core feature of the model lies in its use of GIN convolutional layers. The layers are defined as follows:

- `conv1`: This is the first GIN convolutional layer, which initializes an instance of the GIN handcrafted model. It takes as a parameter a neural network (`nn1`), which consists of fully connected layers. The parameter `train_eps=True` indicates whether the epsilon value, used in the aggregation function, should be trained. The default aggregation function is `add`, but this can be modified.
- `conv2`: This is the second GIN convolutional layer, which uses `nn2` for the graph convolution operation. Similar to `conv1`, it performs the convolution but with the updated set of neural network layers (`nn2`), which processes the hidden representations further to obtain the final node embeddings.

Both layers operate on the input feature matrix x and the graph structure (defined by `edge_index`) to learn graph representations. The dimensions of these layers are as follows:

- `conv1`: Takes an input feature matrix of size $[N, \texttt{in\_channels}]$ and outputs a hidden representation of size $[N, \texttt{hidden\_channels}]$.
- `conv2`: Takes an input feature matrix of size $[N, \texttt{hidden\_channels}]$ and outputs the final embedding of size $[N, \texttt{out\_channels}]$.

Projection Layer: The projection layer is a simple fully connected layer:

$$\texttt{projection} : \mathbb{R}^{\texttt{in\_channels}} \rightarrow \mathbb{R}^{\texttt{hidden\_channels}}$$

This layer projects the node features into a hidden space of size $[N, \texttt{hidden\_channels}]$ and it is used for the residual connections.

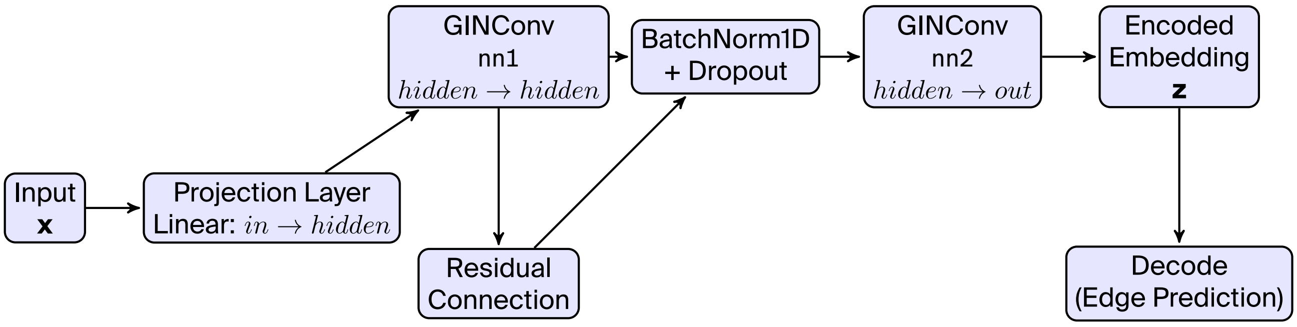
Batch Normalization: The `batch` layer applies batch normalization to the hidden features output by `conv1` before passing it to `conv2`. This normalization layer helps stabilize training.

Dropout: Dropout is applied after the first convolutional layer and after the edge classifier during training to prevent overfitting.

0.4.2. Model Forward Pass

The forward pass of the model consists of two stages:

- `encode`: This function performs the GIN convolution operation by passing the input feature matrix x and the graph structure (edges defined by `edge_index`) through `conv1` and `conv2`. It returns the final node embeddings.
- `decode`: If `link_pred` is True, it computes the dot product between the embeddings of node pairs (edge labels), returning the predicted link probability. If `link_pred` is False, it uses the edge classifier for edge classification.



Model Dimensions Overview

- `Input size`: $[N, \texttt{in\_channels}]$, where N is the number of nodes.
- `After conv1`: $[N, \texttt{hidden\_channels}]$
- `After conv2`: $[N, \texttt{out\_channels}]$

0.5. Model Implementation Details

Model Initialization

The Graph Neural Network model is initialized with the following configuration:

```
model = Net(cora_train.x.shape[1], 128, 64, is_batch=True).to(device)
```

where the parameters represent:

- `in_channels`: `cora_train.x.shape[1]` (matches the node feature dimension)
- `hidden_channels`: 128
- `out_channels`: 64
- Batch processing flag: `True`

0.5.1. Training Configuration

The training process is configured with the following parameters:

$$\begin{aligned}\text{Loss Function} &: \text{Binary Cross Entropy with Logits}\\ \text{Epochs} &: 1000\\ \text{Learning Rate} &: 0.01\\ \text{Weight Decay} &: 5 \times 10^{-4}\end{aligned}$$

Binary Cross Entropy with Logits Loss

The Binary Cross Entropy with Logits Loss combines a sigmoid layer and the Binary Cross Entropy loss in a single class. Mathematically, for a single prediction, it is defined as:

$$\mathcal{L}(x, y) = -w[y \cdot \log(\sigma(x)) + (1 - y) \cdot \log(1 - \sigma(x))] \quad (1)$$

where:

- x is the raw output (logit) from the model
- y is the target binary value (0 or 1)
- $\sigma(x) = \frac{1}{1+e^{-x}}$ is the sigmoid function
- w is an optional weight parameter

For a batch of N predictions, the loss is averaged:

$$\mathcal{L}_{\text{batch}} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(x_i, y_i) \quad (2)$$

Edge Representation and Sampling

Edge Index Terminology

In the context of our graph data:

- **Edge Index** (`data.train_pos_edge_index`): A tensor of shape $(2, |E^+|)$ where $|E^+|$ is the number of positive edges. Each column represents an edge and contains two node indices (i, j), indicating a connection from node i to node j. For example:

$$E^+ = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix}$$

  represents edges $(0, 3)$, $(1, 4)$, and $(2, 5)$.
- **Edge Label Index** (`cora_train.edge_label_index`): A tensor containing both positive and negative edge pairs used for training. Its shape is $(2, |E_L|)$ where $|E_L|$ is the total number of edge labels (both positive and negative). Similar to the edge index format, but includes both types of edges.
- **Number of Positive Edges** (`data.train_pos_edge_index.size(1)`): Represents $|E^+|$, the total count of positive edges in the training set. This is used as a reference for sampling an equal number of negative edges to maintain class balance.

0.5.2. Negative Sampling Strategy

The training implementation supports both transductive and inductive learning approaches, differentiated by a flag in the training method. Both approaches incorporate negative sampling, but with distinct implementations:

Transductive Learning

For the transductive setting, negative edges are sampled using:

$$\texttt{neg\_edge\_index} = \text{negative\_sampling}(E^+, |V|, |E^+|)$$

where:

- E^+: positive edges (`data.train_pos_edge_index`)
- $|V|$: total number of nodes (`data.num_nodes`)
- $|E^+|$: number of positive edges (`data.train_pos_edge_index.size(1)`)

Inductive Learning

For the inductive setting, negative edges are generated using a custom sampling function:

$$\texttt{neg\_samples\_index} = \text{get\_negative\_edges}(E, |V|, |E_L|)$$

where:

- E: all edges (`cora_train.edge_index`)
- $|V|$: total number of nodes (`data.num_nodes`)
- $|E_L|$: number of edge labels (`cora_train.edge_label_index.shape[1]`)

In both approaches, the number of negative samples matches the number of positive edges in the training set, maintaining a balanced dataset for the link prediction task. The key difference lies in how the negative edges are sampled: the transductive approach uses the built-in PyTorch Geometric negative sampling function, while the inductive approach uses a custom implementation to ensure proper sampling for the inductive learning scenario.

0.6. Analysis of Training Dynamics and Testing Performance

Splitting strategies

We have trained the model across different splitting configurations, as reported by the table below.

Splitting Strategy	Validation Ratio	Test Ratio
Strategy #1	0.2	0.2
Strategy #2	0.1	0.2
Strategy #3	0.1	0.1
Strategy #4	0.2	0.1

Table 1: Data splitting configurations used in the GIN experiments. The remaining proportion is used for training (e.g., Strategy #1 uses 0.6 for training, 0.2 for validation, and 0.2 for testing).

Training Dynamics

In Fig. 3 we have reported the training loss behaviour across all the strategies for both Inductive and Transductive. The training curves demonstrate a characteristic rapid initial descent in loss during the first 200 epochs, followed by a stable plateau phase that persists through the remaining training period of 900 epochs. This uniform behavior might superficially suggest comparable model performance across configurations.

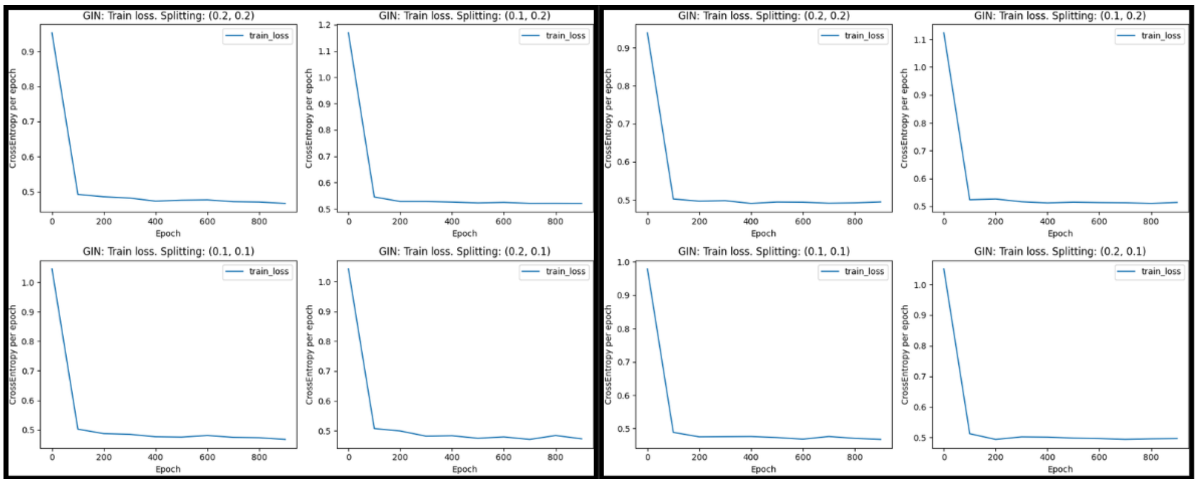


Figure 3: Comparison of GIN training loss curves across different data splitting configurations. **Left panel**: Inductive learning training dynamics with four splitting ratios: $(0.2, 0.2)$, $(0.1, 0.2)$, $(0.1, 0.1)$, and $(0.2, 0.1)$. Right panel: Transductive learning with identical splitting configurations. Both scenarios demonstrate rapid initial convergence within 200 epochs, followed by stable plateaus around 0.5 loss.

Testing

The link prediction process in our implementation follows a binary classification approach, where the model predicts the existence of edges in the graph. The key aspects of the prediction and evaluation process are:

The model outputs logits which are transformed into probabilities using the sigmoid function:

$$p(edge) = \sigma(logits) = \frac{1}{1 + e^{-logits}} \tag{3}$$

A binary prediction is then made using a threshold of 0.5:

$$prediction = \begin{cases} 1 & \text{if } p(edge) > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Altough similar resalut during training, in the testing emerse the real differences betewwn the tow training method. The experimental results from the link prediction task using GIN layers reveal compelling insights into the model's performance characteristics. The most striking observation is the substantial performance gap between inductive and transductive learning approaches (Table 2).

The **inductive learning** setup demonstrates remarkable effectiveness, achieving an average AUC of 0.814 and an accuracy of 0.748, in stark contrast to the **transductive learning** setup's modest performance, with an average AUC of 0.553 and an accuracy of 0.520.

The inductive framework exhibits notable stability across multiple trials, with AUC values consistently ranging from 0.786 to 0.833 and accuracy scores from 0.72 to 0.78. This consistency underscores the GIN architecture's robust ability to learn and generalize graph structural patterns. In contrast, the transductive learning scenario shows greater performance variability, with AUC values fluctuating between 0.520 and 0.589 and accuracy ranging from 0.43 to 0.59. This suggests that the choice of splitting ratio has a greater influence on model generalization than the training curves alone indicate..

These findings indicate that GIN layers excel in inductive link prediction, effectively generalizing to unseen graph structures by capturing transferable patterns. However, their poor transductive performance (AUC 0.55) reveals limitations in learning graph-specific patterns within fixed structures. This highlights the need for architectural improvements for transductive tasks, while affirming GIN's reliability in inductive scenarios.

Method	**Metric**	**Trial 1**	**Trial 2**	**Trial 3**	**Trial 4**
Inductive	AUC	0.805	0.786	0.832	0.833
	Accuracy	0.760	0.730	0.720	0.780
Transductive	AUC	0.539	0.520	0.562	0.589
	Accuracy	0.540	0.520	0.430	0.590
Mean Values	AUC	Inductive: 0.814		Transductive: 0.553	
	Accuracy	Inductive: 0.748		Transductive: 0.520	

Table 2: Comparison of Inductive and Transductive Learning Performance

References

[1] Pytorch Geometric. *RandomLinkSplit*. Available at: `https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.transforms.RandomLinkSplit.html`. 2017. URL: `https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.transforms.RandomLinkSplit.html`.

[2] Pytorch Geometric. *RandomLinkSplit*. Available at: `https://pytorch-geometric.readthedocs.io/en/1.4.3/_modules/torch_geometric/utils/train_test_split_edges.html`. 2017. URL: `https://pytorch-geometric.readthedocs.io/en/1.4.3/_modules/torch_geometric/utils/train_test_split_edges.html`.

[3] Ruslan Salakhutdinov Zhilin Yang William W. Cohen. *Revisiting Semi-Supervised Learning with Graph Embeddings*. Available at: `https://arxiv.org/abs/1603.08861`. 2016. URL: `https://arxiv.org/abs/1603.08861`.