GNN Tutorial: Node Classification & Link Prediction with PyTorch Geometric

ACM India Summer School 2025: AI for Social Good

Tutorial by Shubhajit Roy

Goals of this tutorial:

1. Understand the basics of GNNs and their representations.
2. Learn how to use PyTorch Geometric for graph data handling and model building.
3. Implement and train GNN models for node classification.
4. Implement and train GNN models for link prediction.
5. Discuss concepts, ask questions, and explore variations throughout the session.

1. Introduction to Graph Neural Networks

What are Graphs? Graphs are structures used to represent relationships (edges) between entities (nodes).
What are Graph Neural Networks? GNNs are a type of neural network designed to work directly with graph-structured data. They learn representations (embeddings) for nodes, edges, or entire graphs by aggregating information from their local neighborhoods.
Why GNNs?

- Capture complex relationships and dependencies in data.
- Achieve state-of-the-art results on various graph-related tasks.
- Can generalize to unseen nodes and graphs.

PyTorch Geometric (PyG): PyG is a library built upon PyTorch to easily write and train GNNs for a wide range of applications related to structured data. It provides:

- Data handling for graphs.
- Implementations of many popular GNN layers and models.
- Benchmark datasets.
- Useful utilities for graph machine learning.

2. Setup and Installation

If you haven't installed the necessary libraries, you can do so by running the following commands in your terminal or a code cell (remove the `!` if running in a terminal):

In [ ]:

```
!pip install torch torchvision torchaudio
!pip install torch_geometric
!pip install tqdm
!pip install scikit-learn
```

In [2]:

```
# Import Libraries
import warnings
warnings.filterwarnings("ignore")
import torch
import torch.nn.functional as F
import torch_geometric
from tqdm import tqdm
import matplotlib as mpl
from sklearn.manifold import TSNE
from torch_geometric.nn import GCNConv, SAGEConv, GATConv # Common GNN layers
from torch_geometric.data import Data, DataLoader
from torch_geometric.datasets import Planetoid, TUDataset # Benchmark datasets
from torch_geometric.utils import train_test_split_edges, negative_sampling, to_networkx
import torch_geometric.transforms as T

import numpy as np
import matplotlib.pyplot as plt
import networkx as nx # For graph visualization (optional)
```

In [ ]:

```python
# Helper function for printing dataset information
def print_dataset_summary(dataset_name, data):
    print(f"--- {dataset_name} Dataset Summary ---")
    print(f"Number of graphs: {len(dataset) if isinstance(dataset, list) else 1}")
    print(f"Number of nodes: {data.num_nodes}")
    print(f"Number of edges: {data.num_edges}")
    print(f"Number of node features: {data.num_node_features}")
    if hasattr(data, 'num_classes'):
        print(f"Number of classes: {data.num_classes}")
    print(f"Contains isolated nodes: {data.has_isolated_nodes()}")
    print(f"Contains self-loops: {data.has_self_loops()}")
    print(f"Is undirected: {data.is_undirected()}")
    print(f"Data object: {data}")
    print("-----------------------------------")

print(f"PyTorch version: {torch.__version__}")
print(f"PyG version: {torch_geometric.__version__}")

# Set device (GPU/MPS if available, otherwise CPU)
device = torch.device('cuda' if torch.cuda.is_available() else 'mps' if torch.backends.mps.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
PyTorch version: 2.7.0
PyG version: 2.6.1
Using device: mps
```

3. Understanding Graph Data in PyTorch Geometric

PyG represents graphs using the `torch_geometric.data.Data` object. Key attributes include:

Attribute	Description
`x`	Node features matrix of shape `[num_nodes, num_features]`.
`edge_index`	Graph connectivity in COO format `[2, num_edges]`.
`y`	Node or graph labels (can be for classification/regression).

Let's create a simple example:

In [3]:

```
# Example: A simple graph with 3 nodes and 2 edges
# Nodes: 0, 1, 2
# Edges: (0,1), (1,2)

x = torch.tensor([
    [-1, 1], # Node 0 features
    [0, 0],  # Node 1 features
    [1, -1]  # Node 2 features
], dtype=torch.float)

edge_index = torch.tensor([
    [0, 1],  # Source nodes
    [1, 2]   # Target nodes
], dtype=torch.long)

# Create a Data object
simple_data = Data(x=x, edge_index=edge_index)
print(simple_data)
```

```
Data(x=[3, 2], edge_index=[2, 2])
```

3.1 Attributes of Graphs

In [4]:

```
# For the previous graph, suppose we want to assign some more information to nodes or edges
node_info = torch.tensor([
    True, # Node 0 information
    False, # Node 1 information
    False # Node 2 information
])

edge_attr = torch.tensor([
    1.5, # Edge between 0 and 1
    9.0, # Edge between 1 and 2

])
simple_data.node_info = node_info
simple_data.edge_attr = edge_attr
print(simple_data)
```

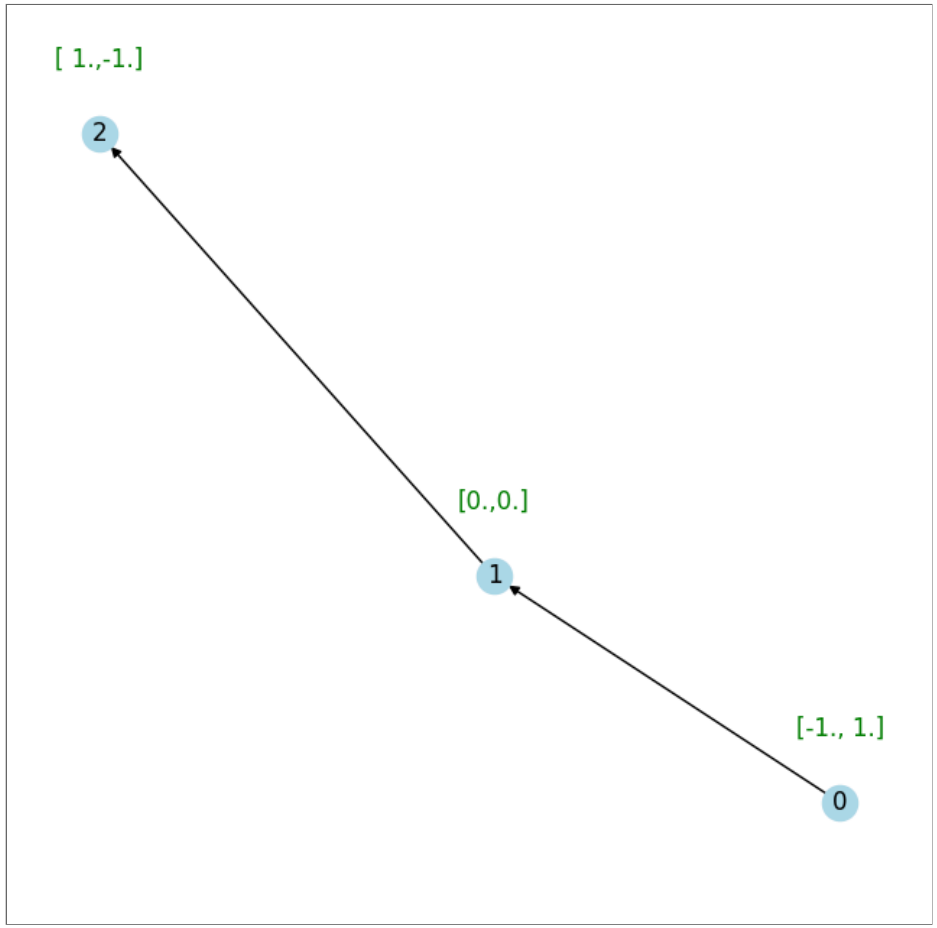
```
Data(x=[3, 2], edge_index=[2, 2], node_info=[3], edge_attr=[2])
```

In [5]:

```python
# Helper functions for plotting graph
def nudge(pos, x_shift, y_shift):
    return {n: (x + x_shift, y + y_shift) for n, (x, y) in pos.items()}

def draw_graph_with_attributes(data, show_x=True, figsize = (8, 8), x_nudge = 0.0, y_nudge = 0.07,
    ax = None,
    font_color = "green",
    edge_color = "black",
    node_color = "lightblue"):
    """Draw a graph with node labels and attributes"""
    if ax is None:
        fig, ax = plt.subplots(1, 1, figsize=figsize)

    G = to_networkx(data)
    pos = nx.spring_layout(G)
    nx.set_node_attributes(G, {n:{"x": data.x[n].tolist()} for n in range(data.num_nodes)})

    nx.draw_networkx(G, pos=pos, with_labels=True, ax=ax, edge_color=edge_color, node_color=node_color)
    pos_nudged = nudge(pos, x_nudge, y_nudge)
    if show_x:
        props = nx.get_node_attributes(G, "x")
        props = {
            node_id: np.array2string(np.array(x), precision=2, separator=",")
            for node_id, x in props.items()
        }
        nx.draw_networkx_labels(
            G, pos=pos_nudged, labels=props, ax=ax, font_color=font_color
            )
    ax.set_ylim(tuple(i * 1.1 for i in ax.get_ylim()))
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)
    ax.spines["bottom"].set_visible(False)
    ax.spines["left"].set_visible(False)
```

In [6]:

```
draw_graph_with_attributes(simple_data)
```



🔷 Discussion Point 🔷

- What other attributes can a `Data` object hold (e.g., `train_mask`, `test_mask`)? How are they used?

- The `torch_geometric.data.Data` object is flexible and can store any attribute related to the graph. You can even add custom fields like:

Attribute	Description
`train_mask` / `val_mask` / `test_mask`	Boolean masks for supervised learning on subsets of nodes.
`pos_edge_label_index`, `neg_edge_label_index`	Used in link prediction tasks.
`edge_attr`	Edge features (e.g., weights, distances).
`batch`	Assigns each node to a graph when dealing with batched inputs.

These attributes are automatically tracked and transferred across operations and can be used in model design or evaluation.

🔷 Coding Point 🔷

- Create a directed graph of 10 nodes with atleast 15 edges
- Assign random features of dimension 5
- Assign edge weights to each edge
- Assign class to each node (Binary class)
- Create a `train_mask`, `val_mask`, `test_mask` for nodes
- Visualise the graph `draw_graph_with_attributes()` function

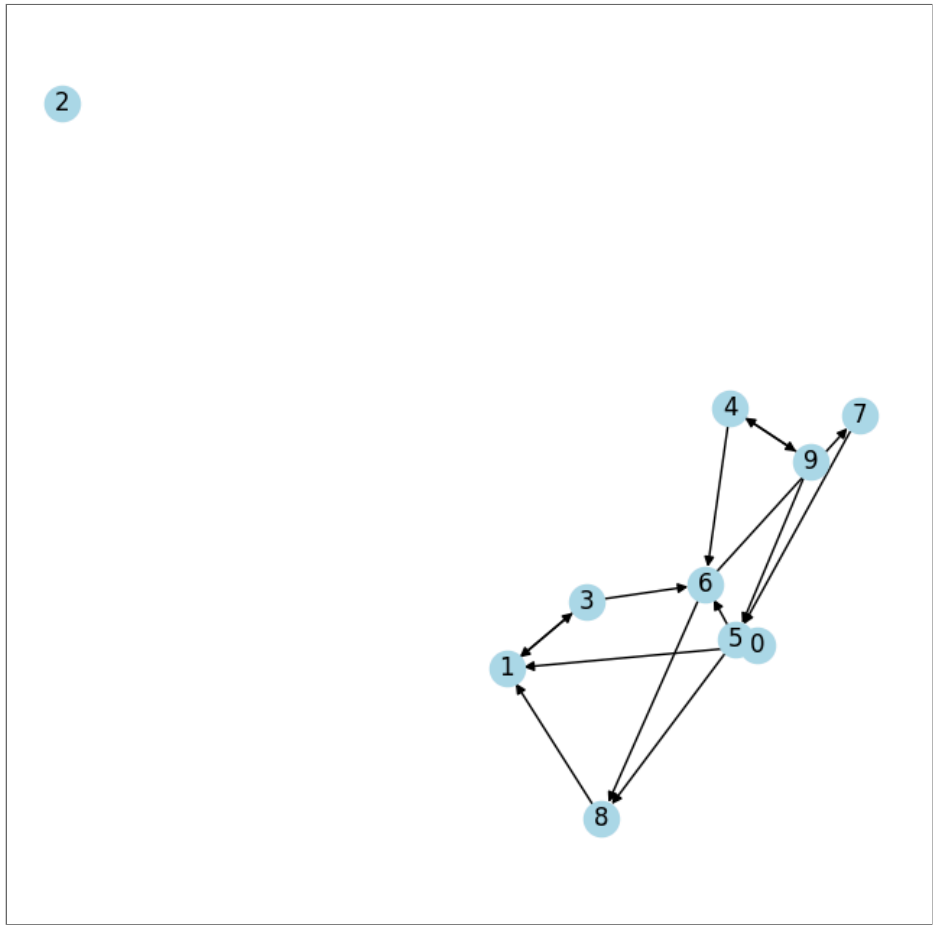
In [ ]:

```
new_x = torch.rand(10,5) # Node feature
new_edge_index = set()
while len(new_edge_index)<15:
    u = torch.randint(0, 10, (1,)).item()
    v = torch.randint(0, 10, (1,)).item()
    if u != v:
        new_edge_index.add((u, v))
new_edge_index = torch.tensor(list(new_edge_index), dtype=torch.long)
new_edge_index = new_edge_index[new_edge_index[:, 0].argsort()].T # Edge Index
new_edge_attr = torch.rand(new_edge_index.shape[1],1) # Edge Attributes

new_y = torch.randint(0, 2, (10,)) # Node Labels

train_mask, val_mask = torch.zeros(10, dtype=torch.bool), torch.zeros(10, dtype=torch.bool)
test_mask = torch.zeros(10, dtype=torch.bool)
train_mask[:7] = True
val_mask[7:9] = True
test_mask[9] = True
new_data = Data(x=new_x, edge_index=new_edge_index, edge_attr=new_edge_attr,
                        y=new_y, train_mask=train_mask, val_mask=val_mask, test_mask=test_mask)
print(new_data)
```

```
Data(x=[10, 5], edge_index=[2, 15], edge_attr=[15, 1], y=[10], train_mask=[10], val_mask=[10], test_mask
=[10])
```

In [ ]:

```
draw_graph_with_attributes(new_data, show_x=False)
```



🔷 Coding Point 🔷

- How would you represent a directed graph vs. an undirected graph using `edge_index`?
- Modify the previously create graph to make it undirected

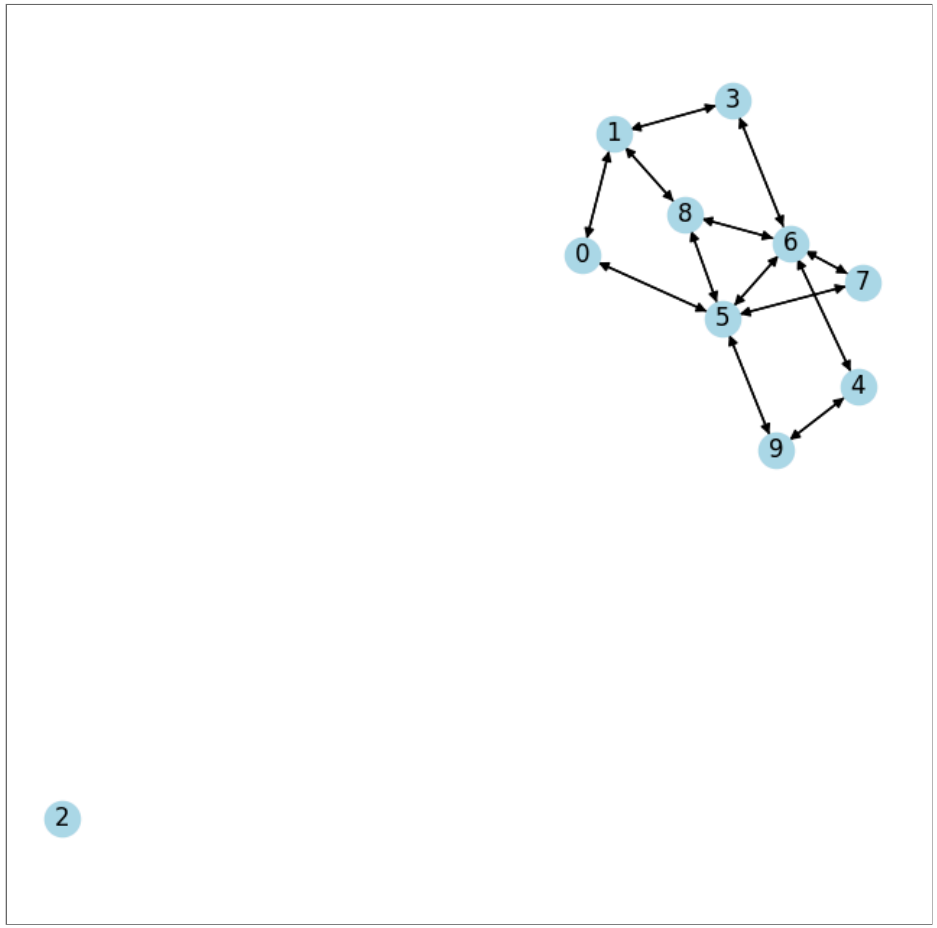
In [ ]:

```
from torch_geometric.utils import to_undirected
new_data.edge_index, new_data.edge_attr= to_undirected(edge_index=new_data.edge_index,
                                                       edge_attr=new_data.edge_attr, num_nodes=new_data.num_node
print(new_data)
```

```
Data(x=[10, 5], edge_index=[2, 26], edge_attr=[26, 1], y=[10], train_mask=[10], val_mask=[10], test_mask
=[10])
```

In [ ]:

```
draw_graph_with_attributes(new_data, show_x=False)
```



4. Part 1: Node Classification

Task: Predict the category (label) of each node in a graph.
Example: In a citation network, classify research papers into different academic fields based on their citation links and content (node features).

4.1. Dataset: Cora

The Cora dataset is a standard benchmark for node classification. It's a citation network where:

- **Nodes:** Scientific publications.
- **Edges:** Citation links between publications.
- **Node Features:** A binary word vector (bag-of-words) indicating the presence/absence of words from a dictionary.
- **Labels:** The academic field of each publication (7 classes).

PyG provides easy access to this dataset through `torch_geometric.datasets.Planetoid`.

In [4]:

```
# Load the Cora dataset
dataset_name = 'Cora'
dataset = Planetoid(root='/tmp/Cora', name=dataset_name, transform=T.NormalizeFeatures())
data = dataset[0] # Get the single graph object

# Print dataset summary
print_dataset_summary(dataset_name, data)

# Explore masks for training, validation, and testing
print(f"\nTrain mask sum: {data.train_mask.sum().item()}")
print(f"Validation mask sum: {data.val_mask.sum().item()}")
print(f"Test mask sum: {data.test_mask.sum().item()}")
```

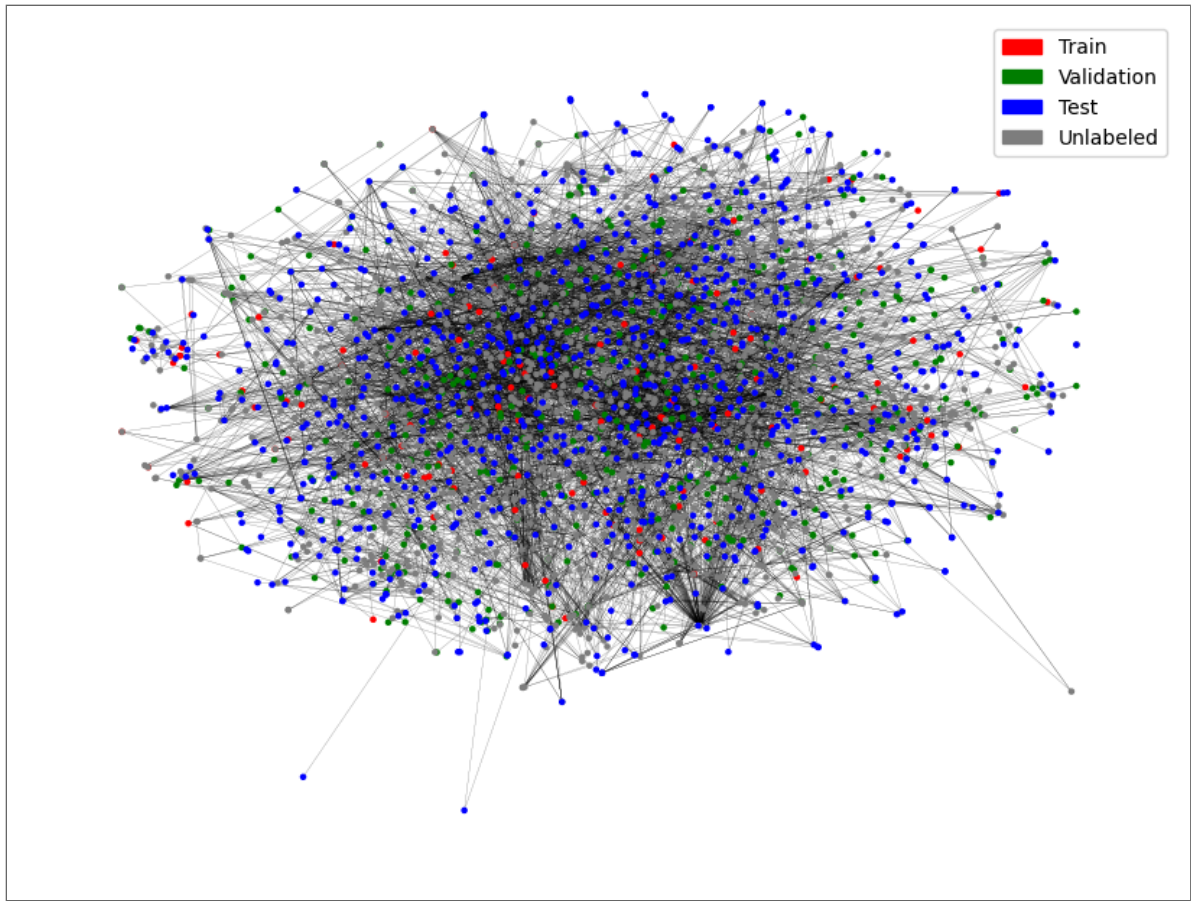
```
--- Cora Dataset Summary ---
Number of graphs: 1
Number of nodes: 2708
Number of edges: 10556
Number of node features: 1433
Contains isolated nodes: False
Contains self-loops: False
Is undirected: True
Data object: Data(x=[2708, 1433], edge_index=[2, 10556], y=[2708], train_mask=[2708], val_mask=[2708], t
est_mask=[2708])
---------------------------------

Train mask sum: 140
Validation mask sum: 500
Test mask sum: 1000
```

In [12]:

```
import matplotlib.patches as mpatches
G = to_networkx(data, to_undirected=True)

pos = TSNE(
        n_components=2, learning_rate="auto", init="random", perplexity=30
    ).fit_transform(data.x.detach().cpu().numpy())

colors = [
    'red' if data.train_mask[i] else
    'green' if data.val_mask[i] else
    'blue' if data.test_mask[i] else
    'gray'
    for i in range(data.num_nodes)
]# Assign colors based on mask
```

In [13]:

```
plt.figure(figsize=(8, 6))
nx.draw(G, pos, node_color=colors, with_labels=False, node_size=5, width=0.1)
legend_handles = [
    mpatches.Patch(color='red', label='Train'),
    mpatches.Patch(color='green', label='Validation'),
    mpatches.Patch(color='blue', label='Test'),
    mpatches.Patch(color='gray', label='Unlabeled')]
plt.legend(handles=legend_handles, loc='best')
plt.show()
```

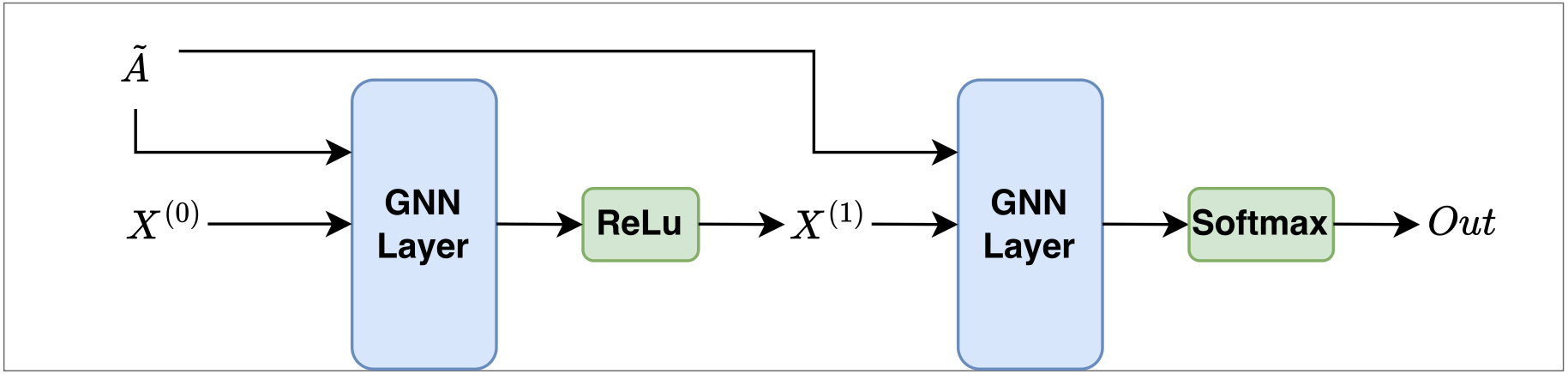


4.2. Defining a GNN Model (GCN)

We'll use a simple Graph Convolutional Network (GCN) model. A GCN layer updates a node's representation by aggregating feature information from its neighbors.
A 2-layer GCN can be defined as: $Z = f(X, A) = \text{softmax}(\hat{A}\,\text{ReLU}(\hat{A}XW^{(0)})W^{(1)})$
Where:

- X: Node feature matrix.
- A: Adjacency matrix (often normalized, $\hat{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$).
- $W^{(0)}, W^{(1)}$: Trainable weight matrices.
- ReLU: Activation function.
- softmax: For multi-class classification output.

In PyG, we use `GCNConv` layers.

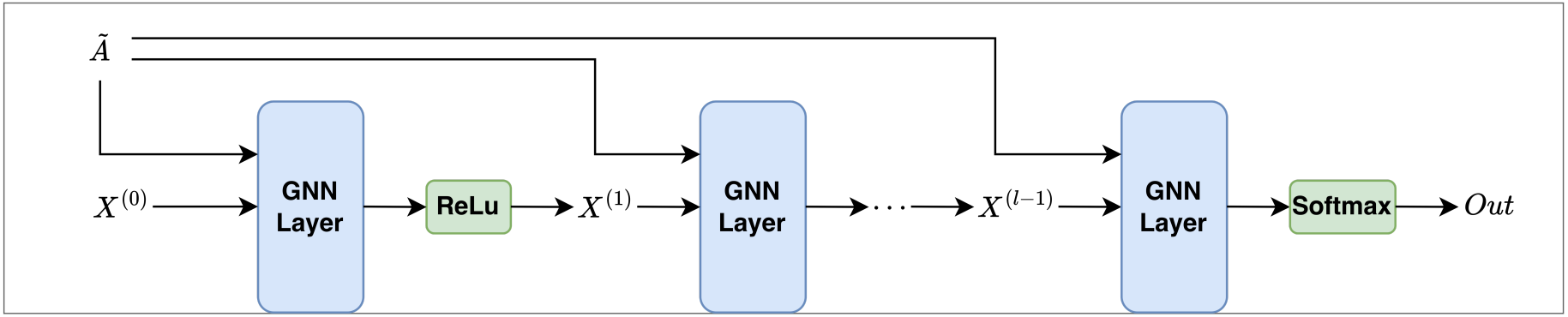

$\tilde{A}$
$X^{(0)}$
GNN Layer
ReLu
$X^{(1)}$
GNN Layer
Softmax
Out

In [5]:

```
class GCN_2Layers(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, dropout_rate=0.5):
        super(GCN_2Layers, self).__init__()
        torch.manual_seed(12345) # For reproducibility
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)
        self.dropout_rate = dropout_rate

    def forward(self, x, edge_index):
        # First GCN layer
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout_rate, training=self.training) # Apply dropout

        x = self.conv2(x, edge_index) # Second GCN layer

        return F.log_softmax(x, dim=1) # Output layer (log_softmax for NLLLoss)
```

🔷 Coding Point 🔷

- Create a new class `GCN_nLayers` to create a GNN model with `n` number of layers


$\tilde{A}$
$X^{(0)}$
GNN Layer
ReLu
$X^{(1)}$
GNN Layer
$\cdots$
$X^{(l-1)}$
GNN Layer
Softmax
Out

In [14]:

```
class GCN_nLayers(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, dropout_rate=0.5, num_layers=2):
        super(GCN_nLayers, self).__init__()
        torch.manual_seed(12345) # For reproducibility
        self.conv = torch.nn.ModuleList()
        if num_layers < 2:
            num_layers = 2
        self.num_layers = num_layers
        self.conv.append(GCNConv(in_channels, hidden_channels))
        for i in range(num_layers-2):
            self.conv.append(GCNConv(hidden_channels, hidden_channels))
        self.conv.append(GCNConv(hidden_channels, out_channels))
        self.dropout_rate = dropout_rate

    def forward(self, x, edge_index):
        for i in range(self.num_layers-1):
            x = self.conv[i](x, edge_index)
            x = F.relu(x)
            x = F.dropout(x, p=self.dropout_rate, training=self.training)
        x = self.conv[-1](x, edge_index)

        return F.log_softmax(x, dim=1) # Output layer (log_softmax for NLLLoss)
```

4.3. Training the Node Classification Model

In [6]:

```
def train_node_classifier(model, criterion, optimizer):
    model.train() # Set the model to training mode
    model.zero_grad()  # Clear gradients
    out = model(data.x, data.edge_index)  # Perform a single forward pass
    # Compute loss only on training nodes
    loss = criterion(out[data.train_mask], data.y[data.train_mask])
    loss.backward()  # Derive gradients
    optimizer.step()  # Update parameters based on gradients
    return loss.item()

@torch.no_grad() # Decorator to disable gradient calculation for evaluation
def test_node_classifier(model, criterion, mask):
    model.eval() # Set the model to evaluation mode
    out = model(data.x, data.edge_index)
    loss = criterion(out[mask], data.y[mask])
    pred = out.argmax(dim=1)  # Use the class with highest probability
    correct = pred[mask] == data.y[mask]  # Check against ground-truth labels
    acc = int(correct.sum()) / int(mask.sum())  # Derive accuracy
    return acc, loss.item()
```

In [7]:

```
# Instantiate the model
node_model_2 = GCN_2Layers(
    in_channels=dataset.num_node_features,
    hidden_channels=128, # You can experiment with this
    out_channels=dataset.num_classes,
).to(device)
print("Node Classification Model (2-Layer GCN):")
print(node_model_2)

# Optimizer
optimizer_2 = torch.optim.Adam(node_model_2.parameters(), lr=0.01, weight_decay=5e-4)

# Loss function (Negative Log Likelihood Loss for multi-class classification)
criterion = torch.nn.NLLLoss()

# Move data to device
data = data.to(device)

# Training loop
epochs = 200
train_losses_2, val_losses_2 = [], []
val_accuracies_2 = []
best_val_loss_2 = float('inf')
best_val_acc_2 = 0
```

```
Node Classification Model (2-Layer GCN):
GCN_2Layers(
  (conv1): GCNConv(1433, 128)
  (conv2): GCNConv(128, 7)
)
```

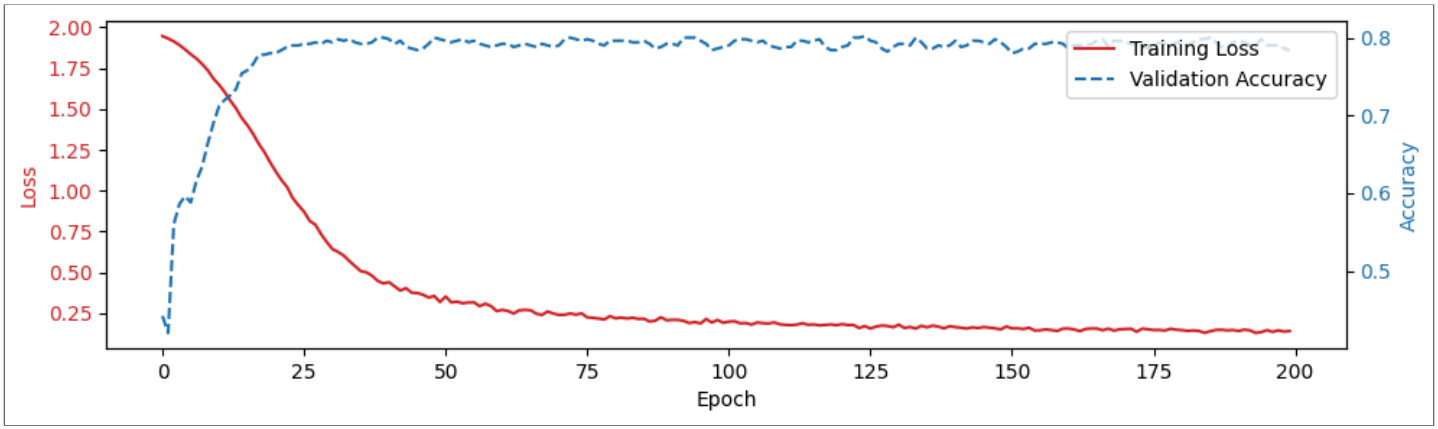
In [8]:

```
print("--- Training Node Classification Model (2-Layer)---")
for epoch in range(1, epochs + 1):
    loss = train_node_classifier(node_model_2, criterion, optimizer_2)
    train_losses_2.append(loss)

    # Validation
    val_acc, val_loss = test_node_classifier(node_model_2, criterion, data.val_mask)
    val_accuracies_2.append(val_acc); val_losses_2.append(val_loss)

    if epoch == 1 or val_loss <= best_val_loss:
        best_val_loss = val_loss
        best_val_acc = val_acc
        torch.save(node_model_2.state_dict(), 'node_model_2.pt') # Save the best model

    if epoch % 50 == 0 or epoch == 1:
        print(f'Epoch: {epoch:03d}, Train Loss: {loss:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}')
print("--- Training Complete ---")
```

```
--- Training Node Classification Model (2-Layer)---
Epoch: 001, Train Loss: 1.9462, Val Loss: 1.9365, Val Acc: 0.4420
Epoch: 050, Train Loss: 0.3192, Val Loss: 0.8655, Val Acc: 0.7980
Epoch: 100, Train Loss: 0.1916, Val Loss: 0.7585, Val Acc: 0.7880
Epoch: 150, Train Loss: 0.1690, Val Loss: 0.7310, Val Acc: 0.7880
Epoch: 200, Train Loss: 0.1416, Val Loss: 0.7272, Val Acc: 0.7840
--- Training Complete ---
```

In [9]:

```
# Plotting training loss and validation accuracy
fig, ax1 = plt.subplots(figsize=(10, 3))
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss', color='tab:red')
ax1.plot(train_losses_2, color='tab:red', label='Training Loss')
ax1.tick_params(axis='y', labelcolor='tab:red')

ax2 = ax1.twinx()
ax2.set_ylabel('Accuracy', color='tab:blue')
ax2.plot(val_accuracies_2, '--', color='tab:blue', label='Validation Accuracy')
ax2.tick_params(axis='y', labelcolor='tab:blue')

fig.tight_layout()
fig.legend(loc='upper right', bbox_to_anchor=(1,1), bbox_transform=ax1.transAxes)
plt.show()
```



4.4. Evaluating the Model

After training, we evaluate the model's performance on the unseen test data.

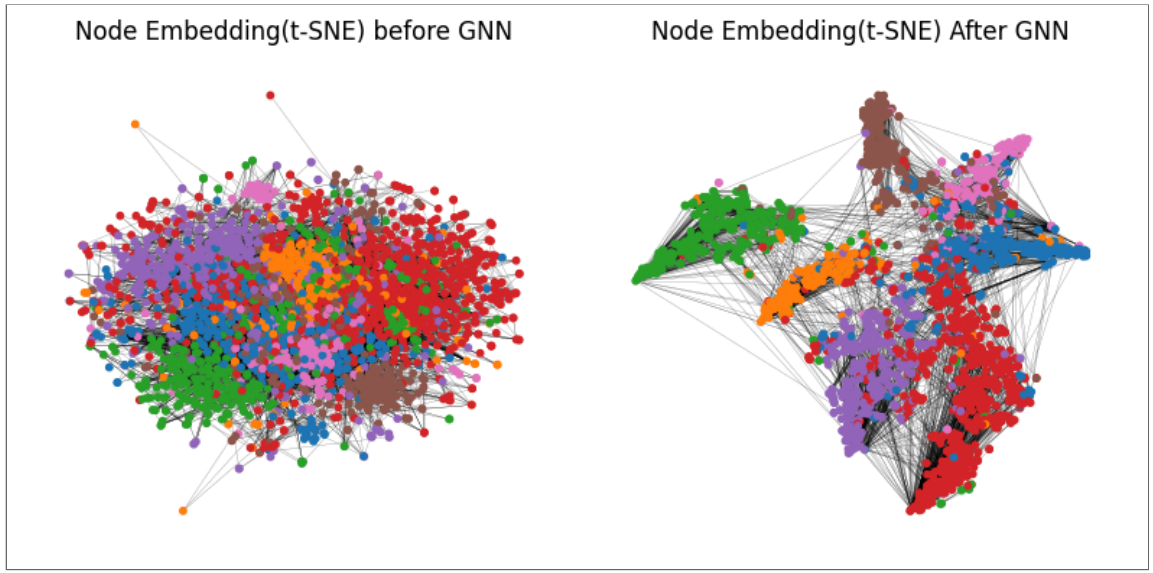
In [10]:

```
# Re-Instantiate model
node_model_2 = GCN_2Layers(
    in_channels=dataset.num_node_features,
    hidden_channels=128,
    out_channels=dataset.num_classes,
).to(device)

node_model_2.load_state_dict(torch.load('node_model_2.pt', weights_only=True)) # Load the best model learnt

final_test_acc_2, final_test_loss_2 = test_node_classifier(node_model_2, criterion, data.test_mask)
print(f'Final Test Accuracy using 2-Layer GCN: {final_test_acc_2:.4f}')
```

```
Final Test Accuracy using 2-Layer GCN: 0.8120
```

In [11]:

```
out = node_model_2(data.x, data.edge_index)
G = to_networkx(data, to_undirected=True)
pos_learned = TSNE(n_components=2, learning_rate="auto",
                   init="random", perplexity=30).fit_transform(out.detach().cpu().numpy())
pos_dataset = TSNE(n_components=2, learning_rate="auto",
                   init="random", perplexity=30).fit_transform(data.x.detach().cpu().numpy())
color_list = list(mpl.colormaps["tab10"].colors)[0 : dataset.num_classes]
fig, axes = plt.subplots(1, 2, figsize=(8, 4))
nx.draw(G, pos_dataset, ax=axes[0], with_labels=False,
        node_color=[color_list[community] for community in data.y], edge_color='black',
        width=0.1, node_size=10); axes[0].set_title("Node Embedding(t-SNE) before GNN")
nx.draw(G, pos_learned, ax=axes[1], with_labels=False,
        node_color=[color_list[community] for community in data.y], edge_color='black',
       width=0.1, node_size=10); axes[1].set_title("Node Embedding(t-SNE) After GNN")
plt.tight_layout(); plt.show()
```



🔷 Discussion Point 🔷

- How could we improve this model further?

We can improve the model further by following ways:

- Using different GNN layers such as `GATConv`, `GraphTransformer` which utilises attention mechanism
- Use different complex architecture, such as **Jumping Knowledge**
- Better feature engineering

🔷 Discussion Point 🔷

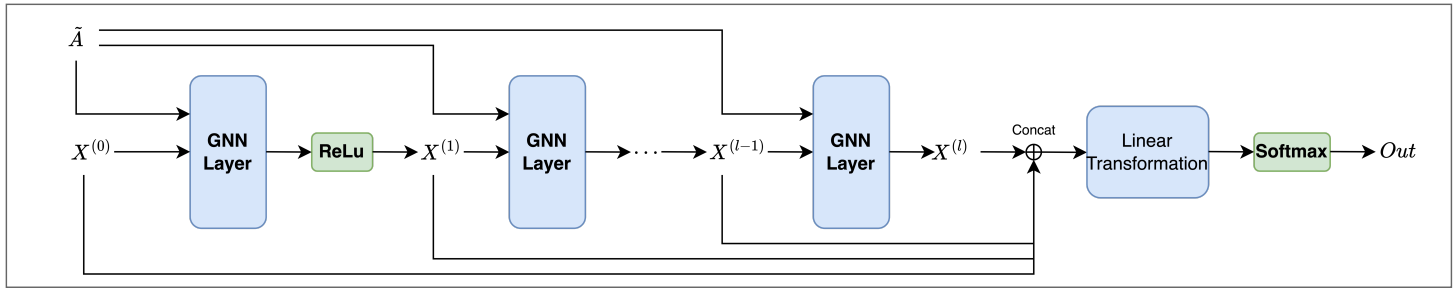
- What are common pitfalls in training GNNs?

Problems related to GNNs are as follows:

- Cannot capture longer dependencies (But Why?)
- Oversmoothening

🔷 Coding Point 🔷

- Create a new class `GNN_JumpingKnowledge` with `n` many layers
- The pipeline is as follows:



- Compare the performance of `GNN_2Layers`, `GNN_nLayers`, and `GNN_JumpingKnowledge`

In [12]:

```
class GCN_JumpingKnowledge(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, dropout_rate=0.5, num_layers=2):
        super(GCN_JumpingKnowledge, self).__init__()
        torch.manual_seed(12345) # For reproducibility
        self.conv = torch.nn.ModuleList()
        if num_layers < 2:
            num_layers = 2
        self.num_layers = num_layers
        self.conv.append(GCNConv(in_channels, hidden_channels))
        for i in range(num_layers-1):
            self.conv.append(GCNConv(hidden_channels, hidden_channels))
        self.lt1 = torch.nn.Linear((num_layers*hidden_channels)+in_channels, out_channels)
        self.dropout_rate = dropout_rate

    def forward(self, x, edge_index):
        #X = torch.tensor([], device=x.device)
        X = x.detach().clone()
        for i in range(self.num_layers):
            x = self.conv[i](x, edge_index)
            x = F.relu(x)
            x = F.dropout(x, p=self.dropout_rate, training=self.training)
            X = torch.cat((X,x), dim=1)
        X = self.lt1(X)
        return F.log_softmax(x, dim=1) # Output layer (log_softmax for NLLLoss)
```

In [15]:

```
# Instantiate the JK model
node_model_JK = GCN_JumpingKnowledge(
    in_channels=dataset.num_node_features,
    hidden_channels=128, # You can experiment with this
    out_channels=dataset.num_classes,
    num_layers=2
).to(device)
print("Node Classification Model (2-Layer GCN with JK):")
print(node_model_JK)
# Instantiate the 3-Layer model
node_model_3 = GCN_nLayers(
    in_channels=dataset.num_node_features,
    hidden_channels=128, # You can experiment with this
    out_channels=dataset.num_classes,
    num_layers=3
).to(device)
print("Node Classification Model (3-Layer GCN):")

# Optimizer
optimizer_JK = torch.optim.Adam(node_model_JK.parameters(), lr=0.01, weight_decay=5e-4)
optimizer_3 = torch.optim.Adam(node_model_3.parameters(), lr=0.01, weight_decay=5e-4)

# Loss function (Negative Log Likelihood Loss for multi-class classification)
criterion = torch.nn.NLLLoss()

# Move data to device
data = data.to(device)
```

```
Node Classification Model (2-Layer GCN with JK):
GCN_JumpingKnowledge(
  (conv): ModuleList(
    (0): GCNConv(1433, 128)
    (1): GCNConv(128, 128)
  )
  (lt1): Linear(in_features=1689, out_features=7, bias=True)
)
Node Classification Model (3-Layer GCN):
```

In [16]:

```
epoch = 200
train_losses_JK, val_losses_JK, train_losses_3, val_losses_3 = [], [], [], []
val_accuracies_JK, val_accuracies_3 = [], []
best_val_loss_JK, best_val_loss_3 = float('inf'), float('inf')
best_val_acc_JK, best_val_acc_3 = 0, 0

print("--- Training Node Classification Model (JK) ---")
for epoch in range(1, epochs + 1):
    loss = train_node_classifier(node_model_JK, criterion, optimizer_JK)
    train_losses_JK.append(loss)

    # Validation
    val_acc, val_loss = test_node_classifier(node_model_JK, criterion, data.val_mask)
    val_accuracies_JK.append(val_acc); val_losses_JK.append(val_loss)

    if epoch == 1 or val_loss <= best_val_loss:
        best_val_loss = val_loss
        best_val_acc = val_acc
        torch.save(node_model_JK.state_dict(), 'node_model_JK.pt') # Save the best model

    if epoch % 50 == 0 or epoch == 1:
        print(f'Epoch: {epoch:03d}, Train Loss: {loss:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}')
print("--- Training Complete ---")
```

```
--- Training Node Classification Model (JK) ---
Epoch: 001, Train Loss: 4.8513, Val Loss: 4.8172, Val Acc: 0.1880
Epoch: 050, Train Loss: 2.9139, Val Loss: 3.1270, Val Acc: 0.7880
Epoch: 100, Train Loss: 3.1979, Val Loss: 2.6644, Val Acc: 0.7880
Epoch: 150, Train Loss: 2.4539, Val Loss: 2.5937, Val Acc: 0.7840
Epoch: 200, Train Loss: 2.5855, Val Loss: 2.5112, Val Acc: 0.7860
--- Training Complete ---
```

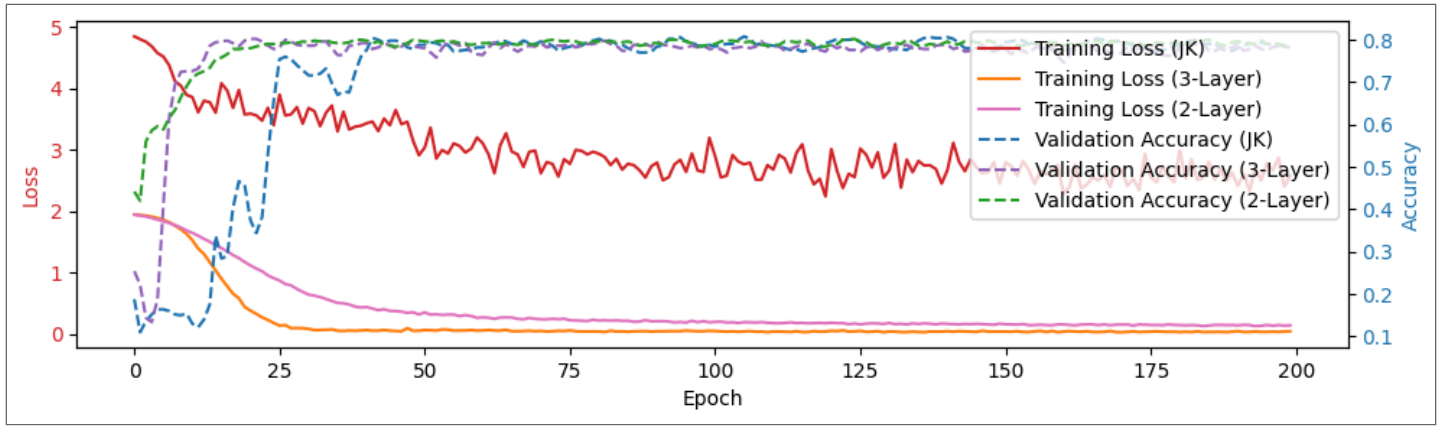
In [17]:

```python
# Training loop
print("--- Training Node Classification Model (3-Layer) ---")
for epoch in range(1, epochs + 1):
    loss = train_node_classifier(node_model_3, criterion, optimizer_3)
    train_losses_3.append(loss)

    # Validation
    val_acc, val_loss = test_node_classifier(node_model_3, criterion, data.val_mask)
    val_accuracies_3.append(val_acc); val_losses_3.append(val_loss)

    if epoch == 1 or val_loss <= best_val_loss:
        best_val_loss = val_loss
        best_val_acc = val_acc
        torch.save(node_model_3.state_dict(), 'node_model_3.pt') # Save the best model

    if epoch % 50 == 0 or epoch == 1:
        print(f'Epoch: {epoch:03d}, Train Loss: {loss:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}')
print("--- Training Complete ---")
```

```
--- Training Node Classification Model (3-Layer) ---
Epoch: 001, Train Loss: 1.9464, Val Loss: 1.9437, Val Acc: 0.2540
Epoch: 050, Train Loss: 0.0539, Val Loss: 0.8316, Val Acc: 0.7820
Epoch: 100, Train Loss: 0.0529, Val Loss: 0.7103, Val Acc: 0.7780
Epoch: 150, Train Loss: 0.0417, Val Loss: 0.7715, Val Acc: 0.7780
Epoch: 200, Train Loss: 0.0446, Val Loss: 0.6960, Val Acc: 0.7820
--- Training Complete ---
```

In [18]:

```
# Plotting training loss and validation accuracy
fig, ax1 = plt.subplots(figsize=(10, 3))
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss', color='tab:red')
ax1.plot(train_losses_JK, color='tab:red', label='Training Loss (JK)')
ax1.plot(train_losses_3, color='tab:orange', label='Training Loss (3-Layer)')
ax1.plot(train_losses_2, color='tab:pink', label='Training Loss (2-Layer)')
ax1.tick_params(axis='y', labelcolor='tab:red')

ax2 = ax1.twinx()
ax2.set_ylabel('Accuracy', color='tab:blue')
ax2.plot(val_accuracies_JK, '--', color='tab:blue', label='Validation Accuracy (JK)')
ax2.plot(val_accuracies_3, '--', color='tab:purple', label='Validation Accuracy (3-Layer)')
ax2.plot(val_accuracies_2, '--', color='tab:green', label='Validation Accuracy (2-Layer)')
ax2.tick_params(axis='y', labelcolor='tab:blue')

fig.tight_layout()
fig.legend(loc='upper right', bbox_to_anchor=(1,1), bbox_transform=ax1.transAxes)
plt.show()
```



In [19]:

```
#Comparison Between Different Models
node_model_JK = GCN_JumpingKnowledge(
    in_channels=dataset.num_node_features,
    hidden_channels=128,
    out_channels=dataset.num_classes,
    num_layers=2
).to(device)

node_model_3 = GCN_nLayers(
    in_channels=dataset.num_node_features,
    hidden_channels=128,
    out_channels=dataset.num_classes,
    num_layers=3
).to(device)

node_model_JK.load_state_dict(torch.load('node_model_JK.pt', weights_only=True))
node_model_3.load_state_dict(torch.load('node_model_3.pt', weights_only=True))

final_test_acc_3, final_test_loss_3 = test_node_classifier(node_model_3, criterion, data.test_mask)
final_test_acc_JK, final_test_loss_JK = test_node_classifier(node_model_JK, criterion, data.test_mask)
print(f'Final Test Accuracy using 2-Layer GCN: {final_test_acc_2:.4f}')
print(f'Final Test Accuracy using 3-Layer GCN: {final_test_acc_3:.4f}')
print(f'Final Test Accuracy using 2-Layer GCN with Jumping Knowledge: {final_test_acc_JK:.4f}')
```

```
Final Test Accuracy using 2-Layer GCN: 0.8120
Final Test Accuracy using 3-Layer GCN: 0.8150
Final Test Accuracy using 2-Layer GCN with Jumping Knowledge: 0.8060
```

🔷 Discussion Point 🔷

- How would you handle a dataset with a very large number of nodes (scalability)?

One possible way to handle will be dicussed in later sessions

5. Part 2: Link Prediction

Task: Predict whether an edge (link) exists or is likely to form between two nodes in a graph.
Example: In a social network, recommend new friends (predict missing links). In a protein-protein interaction network, predict undiscovered interactions.

5.1. Preparing Data for Link Prediction

For link prediction, we need:

1. **Positive edges:** Existing edges in the graph, split into training, validation, and test sets.
2. **Negative edges:** Pairs of nodes that are *not* connected. These are sampled from the graph.

PyG provides `torch_geometric.utils.train_test_split_edges` to help with this. This function modifies the `data` object by adding `train_pos_edge_index`, `val_pos_edge_index`, `val_neg_edge_index`, etc.

In [21]:

```
link_pred_dataset = Planetoid(root='/tmp/Cora_LP', name='Cora', transform=T.NormalizeFeatures())
link_pred_data = link_pred_dataset[0]

# To avoid modifying the original data.edge_index for node classification, we operate on a copy
lp_data = Data(x=link_pred_data.x, edge_index=link_pred_data.edge_index)

# We need to ensure the graph is undirected and does not contain self-loops for train_test_split_edges
from torch_geometric.utils import to_undirected, remove_self_loops
lp_data.edge_index = to_undirected(lp_data.edge_index)
lp_data.edge_index, _ = remove_self_loops(lp_data.edge_index)

# Apply the split. This also generates negative edges for validation and test.
lp_data.train_mask = lp_data.val_mask = lp_data.test_mask = lp_data.y = None # Not needed for link prediction
transform = T.RandomLinkSplit(num_val=0.1, num_test=0.2, is_undirected=True, add_negative_train_samples=True)
train_lp_data, val_lp_data, test_lp_data = transform(lp_data)

print("--- Link Prediction Data Split ---")
print("Training Data:", train_lp_data)
print("Validation Data:", val_lp_data)
print("Test Data:", test_lp_data)

# train_lp_data.edge_label contains 1s for positive edges and 0s for negative edges
# train_lp_data.edge_label_index contains the actual edges (positive and negative)
print(f"\nNumber of training edges (pos+neg): {train_lp_data.edge_label_index.size(1)}")
print(f"Number of positive training edges: {train_lp_data.edge_label.sum().item()}")
```

```
--- Link Prediction Data Split ---
Training Data: Data(x=[2708, 1433], edge_index=[2, 7392], edge_label=[7392], edge_label_index=[2, 7392])
Validation Data: Data(x=[2708, 1433], edge_index=[2, 7392], edge_label=[1054], edge_label_index=[2, 105
4])
Test Data: Data(x=[2708, 1433], edge_index=[2, 8446], edge_label=[2110], edge_label_index=[2, 2110])

Number of training edges (pos+neg): 7392
Number of positive training edges: 3696.0
```

5.2. Defining a GNN Model for Link Prediction

A common approach for link prediction is:

1. **Encoder:** Use a GNN (like GCN, GraphSAGE) to learn node embeddings based on the *message passing edges* (`train_lp_data.edge_index`).
2. **Decoder:** Use these node embeddings to predict the existence of an edge between any two nodes. A simple decoder is the dot product of the embeddings of the two nodes, followed by a sigmoid function.

Score $(u, v) = \sigma(z_u^T z_v)$
Where z_u and z_v are the embeddings for nodes u and v.

In [22]:

```python
class LinkPredictionGCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, dropout_rate=0.5):
        super(LinkPredictionGCN, self).__init__()
        # Encoder (GCN layers)
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels) # out_channels is embedding dimension
        self.dropout_rate = dropout_rate

    def encode(self, x, edge_index):
        x = self.conv1(x, edge_index).relu()
        x = F.dropout(x, p=self.dropout_rate, training=self.training)
        x = self.conv2(x, edge_index) # Final embeddings
        return x

    def decode(self, z, edge_label_index):
        # z: node embeddings [num_nodes, embedding_dim]
        # edge_label_index: edges to score [2, num_edges_to_score]
        node_i_emb = z[edge_label_index[0]] # Embeddings for source nodes
        node_j_emb = z[edge_label_index[1]] # Embeddings for target nodes
        # Dot product decoder
        return (node_i_emb * node_j_emb).sum(dim=-1) # Sum along embedding dimension


    def forward(self, x, edge_index, edge_label_index):
        z = self.encode(x, edge_index)
        return self.decode(z, edge_label_index)
```

5.3. Training the Link Prediction Model

We'll use Binary Cross-Entropy with Logits Loss (`BCEWithLogitsLoss`) as we're predicting probabilities for edge existence (a binary classification task for each potential edge).

In [23]:

```python
def train_link_predictor():
    link_model.train()
    lp_optimizer.zero_grad()

    # Use message passing edges from train_lp_data for encoding
    # These are the original edges used to learn node representations
    z = link_model.encode(lp_data_x, train_lp_data.edge_index)

    # Decode using the supervision edges (positive and negative samples)
    out = link_model.decode(z, train_lp_data.edge_label_index)

    loss = lp_criterion(out, train_lp_data.edge_label.float())
    loss.backward()
    lp_optimizer.step()
    return loss.item()

@torch.no_grad()
def test_link_predictor(current_lp_data):
    link_model.eval()
    # Important: For evaluation, the GNN encoder should use the graph structure
    # defined by train_lp_data.edge_index (the message passing graph)
    # The edges we are predicting (current_lp_data.edge_label_index) should not be part of message passing.
    z = link_model.encode(lp_data_x, train_lp_data.edge_index)
    out = link_model.decode(z, current_lp_data.edge_label_index)
    loss = lp_criterion(out, current_lp_data.edge_label.float())
    # Calculate AUC (Area Under ROC Curve)
    from sklearn.metrics import roc_auc_score
    auc = roc_auc_score(current_lp_data.edge_label.cpu().numpy(), torch.sigmoid(out).cpu().numpy())
    return auc, loss.item()
```

In [24]:

```
# Instantiate the link prediction model
embedding_dim = 64 # Dimension of node embeddings
link_model = LinkPredictionGCN(
    in_channels=dataset.num_node_features,
    hidden_channels=128,
    out_channels=embedding_dim,
    dropout_rate=0.3
).to(device)

print("Link Prediction Model:")
print(link_model)

# Optimizer
lp_optimizer = torch.optim.Adam(link_model.parameters(), lr=0.005)

# Loss function
lp_criterion = torch.nn.BCEWithLogitsLoss()

# Move data to device
train_lp_data = train_lp_data.to(device)
val_lp_data = val_lp_data.to(device)
test_lp_data = test_lp_data.to(device)
lp_data_x = lp_data.x.to(device) # Node features are shared
```

```
Link Prediction Model:
LinkPredictionGCN(
  (conv1): GCNConv(1433, 128)
  (conv2): GCNConv(128, 64)
)
```

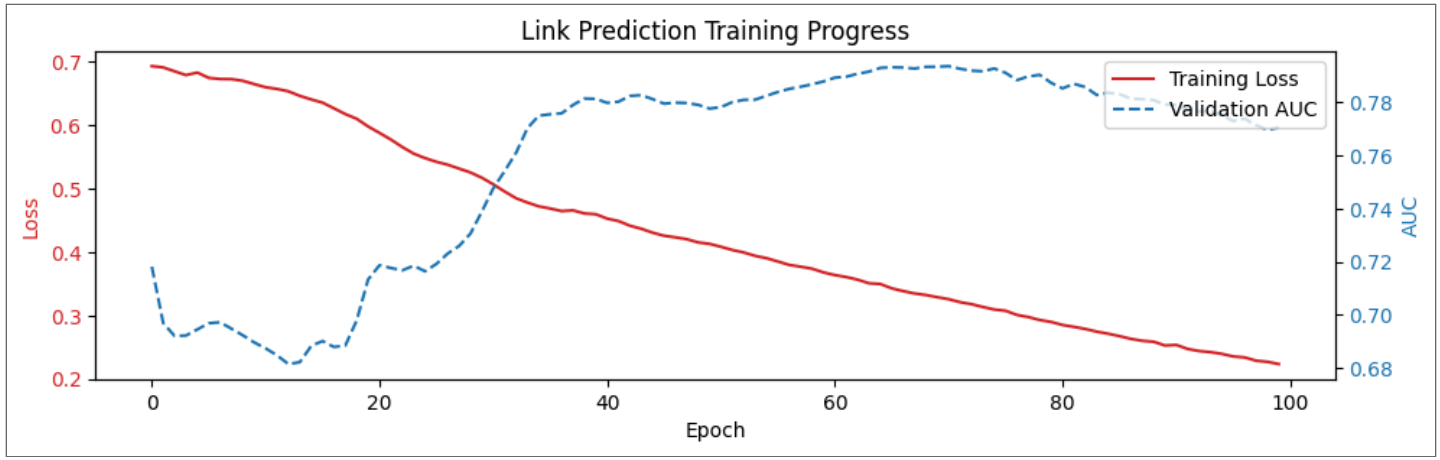
In [25]:

```python
# Training loop for link prediction
lp_epochs = 100
lp_train_losses = []
lp_val_aucs, lp_val_losses = [], []
best_val_loss = float('inf')
best_val_acc = 0

print("--- Training Link Prediction Model ---")
for epoch in range(1, lp_epochs + 1):
    train_loss = train_link_predictor()
    lp_train_losses.append(train_loss)
    # Validation
    val_acc, val_loss = test_link_predictor(val_lp_data)
    lp_val_aucs.append(val_acc); lp_val_losses.append(val_loss)

    if epoch == 1 or val_loss <= best_val_loss:
        best_val_loss = val_loss
        best_val_acc = val_acc
        torch.save(link_model.state_dict(), 'link_model.pt') # Save the best model

    if epoch % 20 == 0 or epoch == 1:
        print(f'Epoch: {epoch:03d}, Train Loss: {loss:.4f}, Val Loss: {val_loss:.4f}, Val Acc: {val_acc:.4f}')
print("--- Link Prediction Training Complete ---")
```

```
--- Training Link Prediction Model ---
Epoch: 001, Train Loss: 0.0446, Val Loss: 0.6916, Val Acc: 0.7182
Epoch: 020, Train Loss: 0.0446, Val Loss: 0.6462, Val Acc: 0.7133
Epoch: 040, Train Loss: 0.0446, Val Loss: 0.6390, Val Acc: 0.7813
Epoch: 060, Train Loss: 0.0446, Val Loss: 0.6707, Val Acc: 0.7879
Epoch: 080, Train Loss: 0.0446, Val Loss: 0.7509, Val Acc: 0.7874
Epoch: 100, Train Loss: 0.0446, Val Loss: 0.9197, Val Acc: 0.7705
--- Link Prediction Training Complete ---
```

In [26]:

```
# Plotting training loss and validation AUC
fig, ax1 = plt.subplots(figsize=(10, 3))

color = 'tab:red'
ax1.set_xlabel('Epoch')
ax1.set_ylabel('Loss', color=color)
ax1.plot(lp_train_losses, color=color, label='Training Loss')
ax1.tick_params(axis='y', labelcolor=color)

ax2 = ax1.twinx()
color = 'tab:blue'
ax2.set_ylabel('AUC', color=color)
ax2.plot(lp_val_aucs, color=color, linestyle='--', label='Validation AUC')
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()
plt.title('Link Prediction Training Progress')
fig.legend(loc='upper right', bbox_to_anchor=(1,1), bbox_transform=ax1.transAxes)
plt.show()
```



5.4. Evaluating the Link Prediction Model

The primary metric for link prediction is often AUC (Area Under the ROC Curve) or Average Precision (AP).

In [27]:

```
# Re-Instantiate model
link_model = LinkPredictionGCN(
    in_channels=dataset.num_node_features,
    hidden_channels=128,
    out_channels=embedding_dim,
    dropout_rate=0.3
).to(device)

link_model.load_state_dict(torch.load('link_model.pt', weights_only=True)) # Load the best model learnt

final_test_auc, final_test_loss = test_link_predictor(test_lp_data)
print(f'\nFinal Test AUC for Link Prediction: {final_test_auc:.4f}')
```

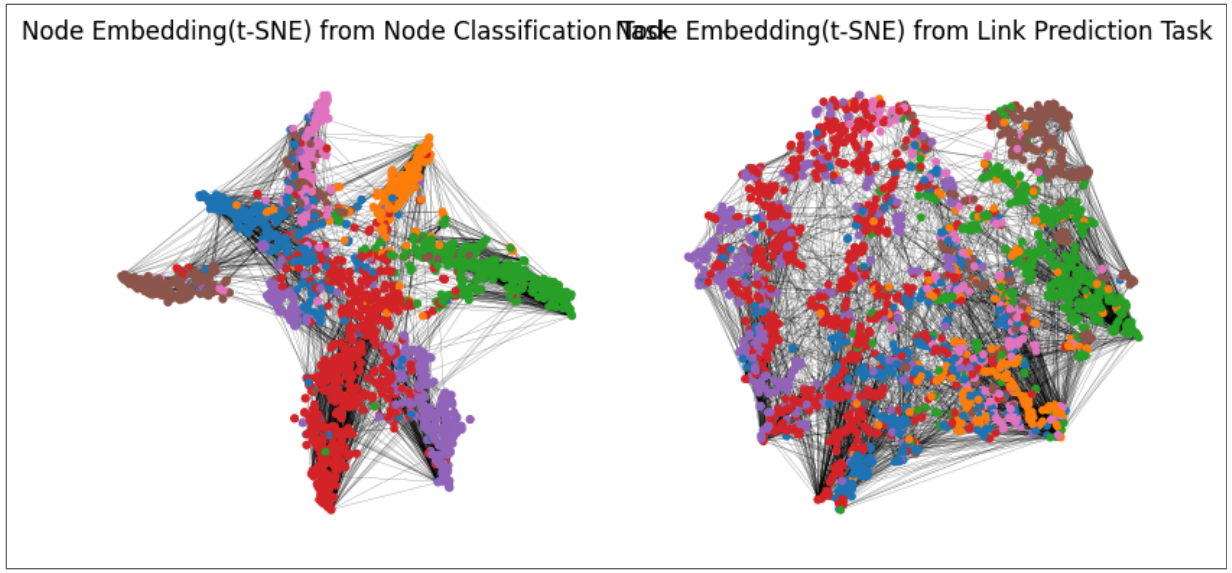
```
Final Test AUC for Link Prediction: 0.7944
```

🔷 Coding Point 🔷

- Generate t-SNE plot of node embeddings based on Node Classification task and Link Prediction task? What are your observations?

In [28]:

```
z = link_model.encode(lp_data_x, train_lp_data.edge_index)
out = node_model_3(data.x, data.edge_index)
G = to_networkx(data, to_undirected=True)
pos_link = TSNE(n_components=2, learning_rate="auto",
                init="random", perplexity=30).fit_transform(z.detach().cpu().numpy())
pos_node = TSNE(n_components=2, learning_rate="auto",
                init="random", perplexity=30).fit_transform(out.detach().cpu().numpy())
color_list = list(mpl.colormaps["tab10"].colors)[0 : dataset.num_classes]
fig, axes = plt.subplots(1, 2, figsize=(8, 4))
nx.draw(G, pos_node, ax=axes[0], with_labels=False,
        node_color=[color_list[community] for community in data.y], edge_color='black',
        width=0.1, node_size=10); axes[0].set_title("Node Embedding(t-SNE) from Node Classification Task")
nx.draw(G, pos_link, ax=axes[1], with_labels=False,
        node_color=[color_list[community] for community in data.y], edge_color='black',
       width=0.1, node_size=10); axes[1].set_title("Node Embedding(t-SNE) from Link Prediction Task")
plt.tight_layout(); plt.show()
```



🔷 Coding Point 🔷

- Conduct Link Prediction on `Citeseer` dataset
- Refer to **https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.datasets.Planetoid.html#torch_geometri** for downloading the dataset

---

In [ ]:

```
```

6. Conclusion and Further Exploration

Congratulations! You've learned how to:

- Represent graph data using PyTorch Geometric.
- Build, train, and evaluate GNN models for **node classification**.
- Build, train, and evaluate GNN models for **link prediction**.

Further Exploration:

- **Different GNN Layers:** Experiment with `GATConv` (Graph Attention Networks), `SAGEConv` (GraphSAGE), and others.
- **Graph-level Tasks:** Try graph classification or regression (e.g., using `TUDataset` and pooling layers).
- **Heterogeneous Graphs:** Work with graphs that have different types of nodes and edges (`torch_geometric.data.HeteroData`).
- **Scalability:** Investigate techniques for training GNNs on very large graphs (e.g., neighbor sampling, graph partitioning).
- **Dynamic Graphs:** Explore models for graphs that change over time.
- **Advanced Link Prediction:** Look into knowledge graph embedding methods (e.g., TransE, ComplEx) or more sophisticated decoders.

Thank you for participating in this tutorial! Feel free to ask any remaining questions.