Name:Salma Hussam Ali

ID:2205093-G1

```
!pip install torch_geometric

import torch
from torch_geometric.data import Data
from torch_geometric.nn import SAGEConv
import torch.nn.functional as F

# 0,1,1,1]
```

!pip install torch_geometric
Installs the PyTorch Geometric library

import torch
Imports PyTorch, the main deep-learning framework

from torch_geometric.data import Data
Imports the Data class used to store graphs with features, edges, and labels

from torch_geometric.nn import SAGEConv
Imports the GraphSAGE convolution layer used to build the model.

import torch.nn.functional as F
Imports neural-network functions

```
#--- Define a small graph with 6 nodes ---
# Node features (2 features per node).
# Here benign users have [1, 0] and malicious have [0, 1] for illustration.
x = torch.tensor(
    [
        [1.0, 0.0],  # Node 0 (benign)
        [1.0, 0.0],  # Node 1 (benign)
        [1.0, 0.0],  # Node 2 (benign)
        [0.0, 1.0],  # Node 3 (malicious)
        [0.0, 1.0],  # Node 4 (malicious)
        [0.0, 1.0]  # Node 5 (malicious)
    ],
    dtype=torch.float,
)
```

x = torch.tensor(
Starts creating a tensor called x, which will store the node feature matrix

6 nodes

benign = [1, 0], malicious = [0, 1]

dtype=torch.float: sets the data type to float

```
# Edge list (undirected). Connect benign users (0-1-2 fully connected)
# and malicious users (3-4-5 fully connected), plus one cross-edge 2-3.
edge_index = (
    torch.tensor(
        [
            [0, 1],
            [1, 0],
            [1, 2],
            [2, 1],
            [0, 2],
            [2, 0],
            [3, 4],
            [4, 3],
            [4, 5],
            [5, 4],
            [3, 5],
            [5, 3],
            [2, 3],
            [3, 2],  # one connection between a benign (2) and malicious (3)
        ],
        dtype=torch.long,
    )
    .t()
    .contiguous()
)
```

- **Benign:** 0, 1, 2 (fully connected)
- **Malicious:** 3, 4, 5 (fully connected)

Cross-edge: 2 ↔ 3 (connects benign to malicious cluster)

.t() → Transposes the tensor to shape [2, num_edges]

.contiguous() → Stores the tensor **continuously in memory**

```
# Labels: 0 = benign, 1 = malicious
# y contains the true labels of the 6 nodes:
# Nodes 0, 1, 2 are benign → label 0
# Nodes 3, 4, 5 are malicious → label
# data is a torch_geometric.data.Data object containing
# x: node features
# edge_index: graph connections (edges)
# y: labels
y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)

data = Data(x=x, edge_index=edge_index, y=y)
```

- **y is a tensor of true labels for each node**

Nodes 0, 1, 2 → label 0 (benign)

Nodes 3, 4, 5 → label 1 (malicious)

- Data is a **PyG class** that stores a graph.

Components:

- x → node features ([1,0] for benign, [0,1] for malicious)
- edge_index → connections between nodes (edgeS)
- y → node labels

```
# --- Define a two-layer GraphSAGE model ---
# his defines a 2-layer GraphSAGE neural network.
# in_channels=2 means each node has 2 features.
# hidden_channels=4 creates a 4-dimensional hidden embedding.
# out_channels=2 means the model outputs scores for 2 classes (benign and malicious).



class GraphSAGENet(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels):
        super(GraphSAGENet, self).__init__()
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        # First layer: sample neighbors and aggregate
        x = self.conv1(x, edge_index)
        x = F.relu(x)  # non-linear activation
        # Second layer: produce final embeddings/class scores
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)  # log-probabilities for classes
```

Defines a 2-layer GraphSAGE model

in_channels=2 → each node has 2 features ([1,0] for benign, [0,1] for malicious).

hidden_channels=4 → first layer produces 4-dimensional hidden embeddings.

out_channels=2 → output layer predicts scores for **2 classes** (benign vs malicious)

Layer 1: Aggregates neighbor information and transforms node features → hidden embedding.

ReLU: Introduces non-linearity.

Layer 2: Aggregates again to produce **class scores** for each node.

log_softmax → outputs **log-probabilities**, suitable for torch.nn.NLLLoss

```
# Instantiate model: input dim=2, hidden=4, output dim=2 (benign vs malicious)
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)

# Simple training loop
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
model.train()
for epoch in range(50):
    optimizer.zero_grad()
    out = model(data.x, data.edge_index)
    loss = F.nll_loss(out, data.y)  # negative log-likelihood
    loss.backward()
    optimizer.step()
```

- Creates a **GraphSAGE model** with:
  - 2 input features per node
  - 4-dimensional hidden layer

    - 2 output classes (benign/malicious)
- Uses **Adam optimizer** to update model weights
- Learning rate = 0.01

50 epochs: model trains over the graph 50 times

Forward pass: computes predictions (log-probabilities) for all nodes

Loss computation: compares predictions with true labels using **NLL loss**

Backpropagation: computes gradients

Optimizer step: updates weights

Updates the model to learn embeddings that distinguish **benign vs malicious** nodes

```
# After training, we can check predictions
model.eval()
pred = model(data.x, data.edge_index).argmax(dim=1)
print("Predicted labels:", pred.tolist())  # e.g. [0,0,
```

Set model to evaluation mode:

model.eval()

- Disables training-specific behaviors
- Ensures the model produces deterministic outputs

Compute predictions:

pred = model(data.x, data.edge_index).argmax(dim=1)

- **Forward pass** over the graph with trained model
- argmax(dim=1) selects the class with highest log-probability for each node
- pred → tensor of predicted labels for all nodes.

print("Predicted labels:", pred.tolist())

- Converts tensor to a Python list for easy viewing