Social Networks

Node Classification Using GraphSAGE on a Synthetic Graph

Name: Sara Ayman Abdelbassir
ID: 2205129

1. Node Feature Definition

```
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,
)
```

explaination

In this experiment, we construct a small synthetic graph composed of 6 nodes.
Each node is represented using a 2-dimensional feature vector:

[1, 0] represents a benign user

[0, 1] represents a malicious user

This simple feature encoding allows the Graph Neural Network to differentiate between normal and malicious behavior based on the node features. These features will be used by the GraphSAGE model to learn patterns from the graph structure and node attributes.

2. Graph Connectivity

```
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()
)
```

Explanation
The graph structure is defined using edge_index, which stores pairs of connected nodes. The graph consists of:

A fully connected cluster of benign nodes (0–1–2)

A fully connected cluster of malicious nodes (3–4–5)

One bridge edge connecting node 2 (benign) to node 3 (malicious)

We include edges in both directions because GraphSAGE expects undirected or bidirectional adjacency.

This structure simulates a realistic scenario where malicious and benign entities form separate communities but occasionally interact. The cross-community edge is important because it challenges the model to distinguish nodes based not only on their neighbors but also on their node features.

3. Node Labels

```
y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)
```

Explanation
Each node is assigned a ground-truth label:
0 → benign
1 → malicious

- ✧ These labels are used by the model during training to learn how to classify nodes based on graph structure and features.

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

- ✧ This structure stores the graph features, adjacency, and labels in a format compatible with PyTorch Geometric.

4.GraphSAGE Model Definition

```
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
```

Explanation

We implement a two-layer GraphSAGE neural network:

Input dimension = 2 (the two node features)

Hidden dimension = 4 (learned embedding size)

Output dimension = 2 (probabilities of benign vs malicious)

The model works as follows:

First GraphSAGE layer
Samples each node's neighbors and aggregates their features.
The result goes through a ReLU activation to introduce non-linearity.

Second GraphSAGE layer
Generates the final node embeddings and produces classification logits.

Log-Softmax
Converts raw scores into log-probabilities suitable for training with NLL loss.

GraphSAGE is chosen because it handles inductive learning, enabling the model to generalize to unseen nodes or graphs.

5. Training Loop

```
# 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()
```

Explaination

We train the GraphSAGE model using the Adam optimizer with a learning rate of 0.01.
For 50 epochs, the following steps occur:

Forward pass:
The model predicts a class distribution for all nodes.

Loss calculation:
We use Negative Log Likelihood (NLL) because the model outputs log-softmax probabilities.

Backpropagation:
Gradients are computed and weights updated.

The model learns to classify nodes correctly based on both their features and how they are connected in the graph.

6. Making Predictions

```
# 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,
```

Explanation

After training, we evaluate the model by:
Running a forward pass without gradient updates.
Selecting the class with the highest probability for each node (argmax).
Printing the predicted labels for all 6 nodes.
This step confirms whether the GraphSAGE model successfully learned to distinguish benign and malicious nodes.

Output:

Predicted labels: [0, 0, 0, 1, 1, 1]

"The model correctly predicted all node labels: nodes 0–2 as benign (0) and nodes 3–5 as malicious (1). This shows that the GraphSAGE model successfully learned the patterns in the node features and graph structure."