Social Network Report

Ahmed Fawzy 2205156

1. Objective

The code builds and trains a small **Graph Neural Network (GNN)** using **GraphSAGE** (from `torch_geometric`) to perform **node classification** on a toy graph.

Each node represents a user, and the task is to classify users as:

- `0` → benign
- `1` → malicious

The graph encodes how users are connected (their relationships), and the model uses both **node features** and **graph structure** to infer whether a node is benign or malicious.

---

2. Dataset / Graph Construction

2.1 Node Features

You define a graph with **6 nodes**, each having **2 features**:

```
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,
)
```

Interpretation:

- Benign users: feature vector `[1, 0]`
- Malicious users: feature vector `[0, 1]`

So the model can, in principle, distinguish benign vs malicious just by features, but the idea is to also leverage **graph connectivity**.

2.2 Graph Structure: `edge_index`

You build an **undirected graph** encoded using `edge_index`:

- Nodes `0, 1, 2` (benign) form a fully connected triangle.
- Nodes `3, 4, 5` (malicious) form another fully connected triangle.
- There is **one cross-edge** between node `2` (benign) and node `3` (malicious).

You encode edges in both directions to simulate an undirected graph:

```
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],
    ],
    dtype=torch.long,
).t().contiguous()
```

2.3 Labels

True labels for each node:

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

- Nodes 0, 1, 2 → benign (`0`)
- Nodes 3, 4, 5 → malicious (`1`)

2.4 Packaging into a `Data` Object

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

`Data` is the fundamental data structure in `torch_geometric`, grouping features, connectivity, and labels into one object.

---

3. Model: GraphSAGE Network

You define a **2-layer GraphSAGE network**:

```
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):
        x = self.conv1(x, edge_index)
        x = F.relu(x)          # non-linearity
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

Key points:

- **Input dimension (`in_channels=2`)**

  Two features per node (benign indicator, malicious indicator).

- **Hidden layer (`hidden_channels=4`)**

  First `SAGEConv` layer computes 4-dimensional embeddings from the original features, aggregating information from neighbors.

- **Output layer (`out_channels=2`)**

  Second `SAGEConv` layer maps embeddings to scores for the two classes (benign vs malicious).

- `F.log_softmax(..., dim=1)` returns **log-probabilities** over the 2 classes for each node.

Why GraphSAGE?

GraphSAGE works by **sampling neighbors and aggregating** their features. Even though in this small toy example all neighbors are used.

---

4. Training Procedure

You set up and train the model end-to-end on all nodes:

```
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)
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)
    loss.backward()
    optimizer.step()
```

Details:

- **Optimizer**: Adam, learning rate `0.01`.
- **Loss**: `F.nll_loss` matches the log probabilities from `log_softmax` with integer labels in `y`.
- **Training data**: all 6 nodes are used for training (no train/test split in this example).

Because the graph is tiny and perfectly labeled, the model will usually learn a near-perfect classifier.

---

5. Evaluation and Predictions

After training, you switch to evaluation mode and compute predictions:

```
model.eval()
pred = model(data.x, data.edge_index).argmax(dim=1)
print("Predicted labels:", pred.tolist())
```

- `model(data.x, data.edge_index)` gives log-probabilities for each node.
- `argmax(dim=1)` picks the most likely class for each node.
- The result is a list of 6 predicted labels, one per node.

In an ideal case, this will match `[0, 0, 0, 1, 1, 1]`.
