## Malak Mohamed Osman Mohamed-220509

---

1. These imports load PyTorch, PyG Data container, GraphSAGE layer and common neural network functions needed to build and train a graph model.

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

2.

```
[3]
0s
x = torch.tensor(
    [
        [1.0, 0.0],
        [1.0, 0.0],
        [1.0, 0.0],
        [0.0, 1.0],
        [0.0, 1.0],
        [0.0, 1.0]
    ],
    dtype=torch.float,
)
```

- We define a feature matrix x that contains one row per node (6 rows total).
- Each row is a 2-dimensional feature vector describing whether the node is benign or malicious.
- Benign nodes (0, 1, 2) are assigned the feature [1.0, 0.0], meaning benign = 1, malicious = 0.
- Malicious nodes (3, 4, 5) are assigned [0.0, 1.0], meaning benign = 0, malicious = 1.

```
[4]
0s
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()
)
```

- We define all graph connections where each row represents one directed edge between two nodes.
- PyTorch Geometric only works with directed edges. So to represent an undirected connection, we write it twice (0→1 and 1→0) to make sure both nodes can exchange information during message passing.
- Nodes 0, 1, and 2 are fully connected to each other, forming a small benign subgraph and Nodes 3, 4, and 5 are also fully connected, forming the malicious subgraph.

```
[5]
✓ 0s
y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)
data = Data(x=x, edge_index=edge_index, y=y)
```

- y stores the true class label for each node: 0 for benign nodes (0, 1, 2) and 1 for malicious nodes (3, 4, 5).
- The Data object bundles everything into one graph structure:
  x → node features
  edge_index → the graph's connections
  y → node labels

```
[6]
✓ 0s
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)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

- The forward function describes how data flows through the model every time the graph is passed into it.
- The first layer conv1 takes the original 2-dimensional node features and transforms them into 4-dimensional embeddings (conv1: 2 → 4).
- A ReLU activation is applied to help the model learn more complex patterns.
- The second layer conv2 receives the 4-dimensional vectors and maps them to 2 output values per node one for each class (benign or malicious) (conv2: 4 → 2).
- log_softmax converts these two output scores into log-probabilities so the network can correctly learn which class each node belongs to.

```
[7]
✓ 0s
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)
```

- This line creates an instance of the GraphSAGENet model and specifies the size of each layer.
- in_channels=2 tells the model that each node has 2 input features.
- hidden_channels=4 sets the hidden layer to produce 4-dimensional embeddings.
- out_channels=2 means the final layer outputs 2 class scores for each node (benign vs malicious).

```
[9]
✓ 0s
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()
```

- We create an Adam optimizer to update the model's weights with a learning rate of 0.01.
- model.train() -> we are in training mode.
- The loop runs for 50 epochs, meaning we train the model 50 times over the same graph.
- optimizer.zero_grad() -> clears old gradients from the previous step.
- out = model(data.x, data.edge_index) -> runs the model to get the predicted class scores for each node.
- loss = F.nll_loss(out, data.y) -> compares the predictions to the true labels and computes how wrong the model is.
- loss.backward() -> computes the corrections needed for each model weight by showing how much each one contributed to the prediction error.
- optimizer.step() -> updates the model's weights to reduce the loss on the next iteration.


Switch the model to evaluation mode, run it on the graph and take the class with the highest score for each node using argmax.