Name : Abdelrahman Ayman Saad Abdelhalim Mohamed  Id : 2205033

# Introduction

In this section, I worked on a small example that shows how graph neural networks can be used for classification tasks. The goal was to see how users in a network can be labeled as either benign or malicious, not just based on their own data, but also by looking at how they are connected to others. This is something traditional machine-learning models usually ignore, since they treat all data points independently.

To handle this type of problem, I used a GraphSAGE model from the `torch_geometric` library. GraphSAGE is designed to learn node representations by mixing each node's features with information from its immediate neighbors. This helps the model capture local structure inside the network and improves classification accuracy.

# Libraries and Setup

The notebook begins by installing and importing the main Python libraries needed for this task. PyTorch is used for all tensor calculations and neural-network operations. Torch Geometric is added on top of PyTorch to provide tools that make working with graph data much easier. From this library, `SAGEConv` is imported to create the core GraphSAGE layers. In addition, `torch.nn.functional` supplies activation functions such as ReLU along with the loss function used during training.

Together, these tools provide everything required to construct, train, and test the graph neural network model.

```
!pip install torch_geometric



import torch

from torch_geometric.data import Data

from torch_geometric.nn import SAGEConv

import torch.nn.functional as F
```

## Graph Construction

To keep the example simple, a small graph containing six nodes is created. Each node represents a user and has two features. Benign users are represented by the feature vector `[1, 0]`, while malicious users are represented by `[0, 1]`. Nodes 0, 1, and 2 are considered benign, and nodes 3, 4, and 5 are marked as malicious.

The connections between users are then defined. The benign nodes are fully connected to one another, forming a small group. The same is done for the malicious nodes. To simulate real-world interaction between groups, one edge is added between node 2 and node 3, linking the benign cluster to the malicious cluster. All connections are made undirected so that information can flow both ways across edges.

```
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,
)
```

Each node's true class label is stored in a tensor where `0` stands for benign and `1` stands for malicious. All the graph data — node features, edge connections, and labels — are combined into a Torch Geometric `Data` object so that they can be easily passed into the model.

```
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()
)
```

## Model Design

The classifier is built using a two-layer GraphSAGE neural network. The first layer takes the two input features of each node and produces a four-dimensional hidden representation. After applying a ReLU activation function, the data is passed to the second GraphSAGE layer, which outputs two values per node. These correspond to the two possible classes.

At the end of the forward pass, a log-softmax function converts the raw outputs into log-probabilities, which are required for calculating the training loss.

```
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
```

## Training Process

The training process uses the Adam optimizer with a learning rate of 0.01. The loss function selected is negative log-likelihood loss, which compares the predicted class probabilities with the actual node labels.

```
model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)



# Simple training loop

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

During each training epoch, the following steps occur:

1. All existing gradients are cleared.
2. The model generates predictions for the nodes.
3. The loss value is calculated using the true labels.
4. Backpropagation computes the gradients.
5. The optimizer updates the model's weights.

This loop is repeated for 50 epochs, allowing the model to improve its predictions gradually.

```
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()
```

## Model Evaluation

Once training is complete, the model is switched into evaluation mode. Predictions are generated by selecting the class with the highest probability for each node.

The final predicted labels are:

```
[0, 0, 0, 1, 1, 1]
```

These predicted values match the original labels exactly. All benign nodes were classified correctly, and all malicious nodes were detected without error.

```
model.eval()

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

print("Predicted labels:", pred.tolist())  # e.g. [0,0,
```

# Discussion

This small experiment shows how effective graph neural networks can be, even with simple features. The connections between users allow information to spread through the network so that nodes benefit from their neighbors' data. The single link between the two groups also demonstrates how relationships can influence classification boundaries.

GraphSAGE is especially useful for large networks because it samples neighbors rather than processing the entire graph at once, making it scalable to real-world applications such as fraud detection, cybersecurity monitoring, and social network analysis.

# Conclusion

In this section, a GraphSAGE-based classifier was successfully implemented to distinguish between benign and malicious users in a small graph. By combining node features and neighborhood information, the model achieved perfect classification on the example dataset. This shows why GNN approaches are well suited for problems where relationships between data points play a critical role.