



**Social network**

**DR : Reem essam**

**ENG : Marwan**

*Name : Ahmed emad fawzy Mohammed*

*ID : 2205086*

---

## *Introduction*

---

*This notebook demonstrates a basic implementation of a Graph Neural Network (GNN) using PyTorch Geometric, specifically the GraphSAGE (SAGEConv) layer.*

*The primary goal of the code is to perform node classification on a small synthetic graph consisting of 6 nodes. Each node has feature vectors, connections (edges), and a class label.*

***The notebook covers:***

-  Graph construction
  -  Model definition using GraphSAGE
  -  Training loop
  -  Model evaluation and predictions
- 

## *Required Libraries and Installation*

---

**!PIP INSTALL TORCH\_GEOMETRIC**

-  This command attempts to install the torch\_geometric library, which is essential for graph-based deep learning in PyTorch.

-  **Note:**

*In practice, torch\_geometric requires additional dependencies (torch-scatter, torch-sparse, etc.) that must match the installed Torch and CUDA versions.*

## Imports

```
➊ IMPORT TORCH
➋ FROM TORCH_GEOMETRIC.DATA IMPORT DATA
➋ FROM TORCH_GEOMETRIC.NN IMPORT SAGECONV
➋ IMPORT TORCH.NN.FUNCTIONAL AS F
```

*Explanation:*

- ➊ torch: Core PyTorch library for tensors and automatic differentiation.
- ➋ Data: Data structure used to represent a graph.
- ➋ SAGEConv: Implementation of the GraphSAGE convolution layer.
- ➋ torch.nn.functional: Provides neural network activation functions and loss functions.
- ➋ must match the installed Torch and CUDA versions.

---

### 3. Graph Definition

---

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

- ➊ Each row represents a node feature vector.
- ➋ The features are illustrative (e.g.,  $[1,0]$  = benign,  $[0,1]$  = malicious).

## Edge Index

```
edge_index = torch.tensor([
    [0, 1, 1, 2, 2, 3, 4, 5, 0, 2],
    [1, 0, 2, 1, 3, 2, 5, 4, 2, 0],
], dtype=torch.long)
```

- ⊕ Represents graph connectivity in **COO (Coordinate) format**.
- ⊕ Shape: (2, E) where E is the number of edges.
- ⊕ Each column denotes a directed edge (source → target).
- ⊕ Undirected edges are represented by adding edges in **both directions**.

## Node Labels

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

- ⊕ Each value corresponds to the class label of a node.
- ⊕ 0 and 1 represent two different categories.
- ⊕ Labels must be of type long for classification loss functions.

## Graph Object

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

### THE DATA OBJECT STORES:

- ⊕ Node features (x)
- ⊕ Graph topology (edge\_index)
- ⊕ Ground-truth labels (y)

---

#### 4. GraphSAGE Model Architecture

---

```
class GraphSAGE(torch.nn.Module):

    def __init__(self, in_channels, hidden_channels, out_channels):
        super().__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)
```

### Layer-by-Layer Explanation

#### First SAGEConv Layer

- ⊕ Aggregates features from neighboring nodes.
- ⊕ Projects features from in\_channels → hidden\_channels.

#### ReLU Activation

- ⊕ Adds non-linearity to the model.
- ⊕ Helps capture complex relationships in the graph.

#### Second SAGEConv Layer

- ⊕ Produces output logits for classification.

#### Log Softmax

- ⊕ Converts logits into log-probabilities.
- ⊕ Required for compatibility with nll\_loss

## Model Initialization

```
model = GraphSAGE(in_channels=2, hidden_channels=8, out_channels=2)
```

- ✚ Input features: 2
  - ✚ Hidden layer size: 8
  - ✚ Output classes: 2
- 

## 5. Training Process

---

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

- ✚ Adam optimizer is used for adaptive learning rate updates.

## Training Loop

```
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()
```

### Step-by-step:

- ✚ Forward pass through the model.
- ✚ Compute classification loss.
- ✚ Backpropagate gradients.
- ✚ Update model weights

---

## 6. Loss Function

---

*F.nll\_loss(output, target)*

- ✚ Negative Log-Likelihood Loss.
  - ✚ Requires log-probabilities as input.
  - ✚ Commonly used with `log_softmax`.
- 

## 7. Model Evaluation

---

*model.eval()*

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

- ✚ `argmax` selects the class with highest probability.
  - ✚ Produces predicted class labels for each node.
- 

## 8. Tensor Shapes Summary

---

Component	Shape
<code>x</code>	(6, 2)
<code>edge_index</code>	(2, E)
<code>Model output</code>	(6, 2)
<code>Labels y</code>	(6,)