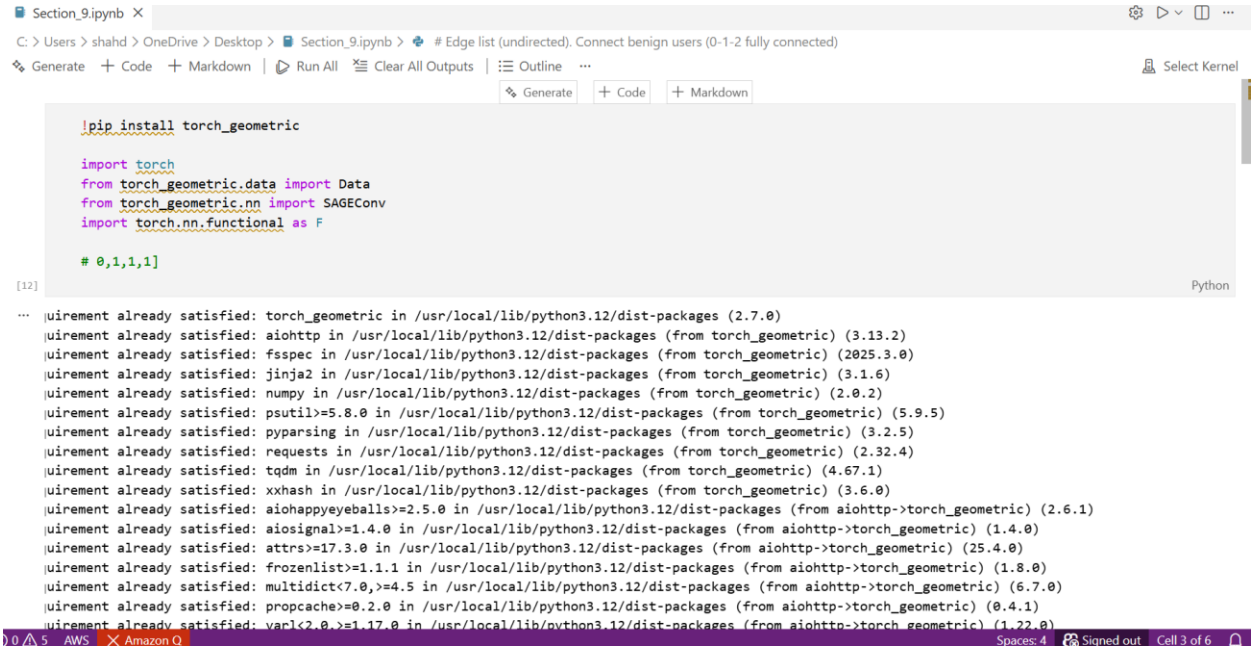
Shahd Mohamed Abd El Sabour

ID : 2205027

Assignment Section 9 : Explaining The Code



Installing the library

```
!pip install torch_geometric
```

- This command installs **PyTorch Geometric**, a library used for working with **Graph Neural Networks (GNNs)**.
- The ! indicates that this command runs in the notebook or Colab environment as a shell command.

Importing required libraries

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

◇ *import torch*

- Imports the main PyTorch library.
- Used for tensors, neural networks, training, etc.

◇ *from torch_geometric.data import Data*

- Imports the **Data** class from PyTorch Geometric.
- `Data` is used to store graph information:
    - node features
    - edge connections
    - labels
    - anything related to a graph structure

Example:

- You might say "this graph has 4 nodes and edges connecting them" → `Data` object holds that.

◇ *from torch_geometric.nn import SAGEConv*

- Imports the **GraphSAGE convolution layer**.
- GraphSAGE is a popular GNN architecture for learning from graphs.
- It updates node embeddings by aggregating information from neighbors.

◇ *import torch.nn.functional as F*

- Imports common neural network functions like:
    - `relu()`
    - `softmax()`

    - `cross_entropy()`
- These functions are often used in forward passes and training.

Output Messages (Requirements already satisfied...)

You see many lines like:

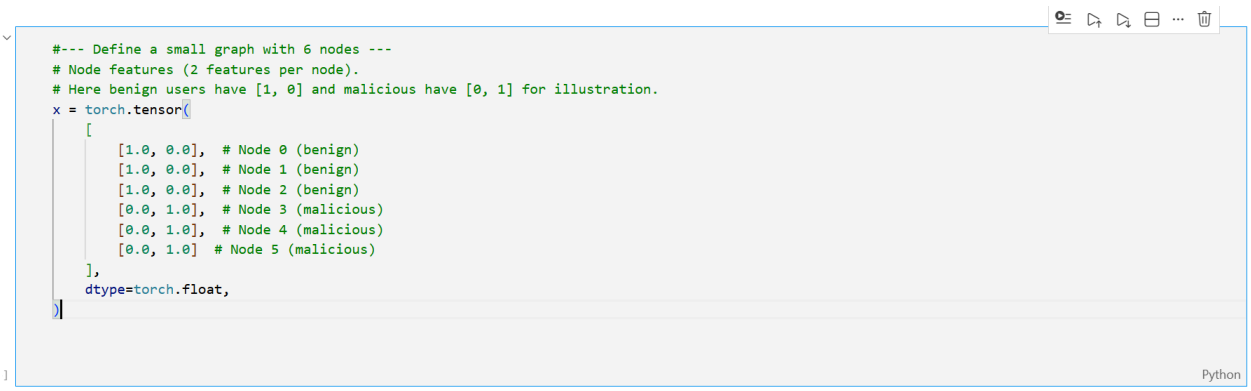
```
Requirement already satisfied: torch_geometric in
/usr/local/lib/python3.12/dist-packages (2.7.0)
Requirement already satisfied: numpy in
/usr/local/lib/python3.12/dist-packages (2.0.2)
...
```

These messages simply mean:

- The packages are **already installed**.
- So `pip` does not reinstall them.

They list all dependencies of `torch_geometric` such as:

- `aiohttp` (for async networking)
- `numpy` (math operations)
- `requests` (HTTP)
- `jinja2` (templating)
- `tqdm` (progress bars)
- etc.



What this part of the code does ?

It creates node feature data for a graph.

- The graph has **6 nodes**.
- Each node has **2 features**.
- These features represent the node's type (benign or malicious).

Representation of Node Types

- **Benign node = `[1.0, 0.0]`**
- **Malicious node = `[0.0, 1.0]`**

This is a very common encoding technique called **one-hot encoding**.

Examples:

- Node 0 → `[1,0]` → benign
- Node 3 → `[0,1]` → malicious

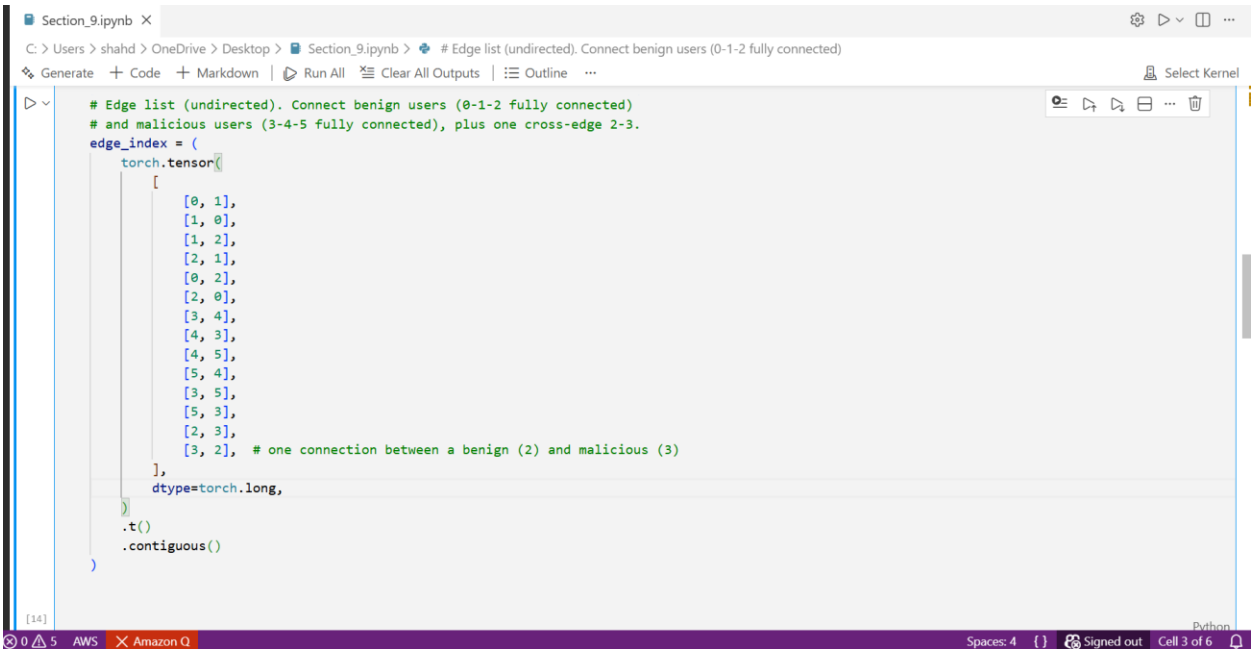
About `torch.tensor()`

- Converts the list of lists into a PyTorch tensor (matrix).
- The tensor shape is:

`6 nodes × 2 features`
Shape: `(6, 2)`

About `dtype=torch.float`

- `Sets the data type to` **`float numbers`**`, not integers.`
- `Neural networks work better with float values, especially during training and gradient updates.`



What this code does ?

It defines the connections (edges) between graph nodes.

In a graph neural network, we must describe how nodes are linked.
This is done using **edge_index**, which lists pairs of connected nodes.

Undirected edges (both directions)

- Each connection is listed **twice**:
  - one direction → u → v
  - reverse direction → v → u

[0, 1] # 0 connected to 1

[1, 0] # 1 connected to 0

PyTorch Geometric treats edges as directed, so to represent an undirected graph, you must include both directions manually.

Node groups

The graph has two fully-connected groups:

1. Benign group (nodes 0, 1, 2)

All 3 benign users are connected to each other:

```
0–1
0–2
1–2
```

Hence edges include:

```
0→1, 1→0
1→2, 2→1
0→2, 2→0
```

2. Malicious group (nodes 3, 4, 5)

They are also fully connected:

```
3–4
3–5
4–5
```

So we add:

```
3→4, 4→3
4→5, 5→4
3→5, 5→3
```

Cross-edge (link between the two communities)

Notice the last pair:

```
[2, 3]
[3, 2]
```

This is a **link between a benign node (2) and a malicious node (3)**.

This simulates a real scenario like:

- user 2 interacts once with malicious user 3
- or a one-off connection across communities

This edge is important:
It allows the GNN to **propagate information between groups**.

Why `.t()`

At the end:

```
.t()
```

This **transposes the tensor**, converting it from shape:

```
(14 edges, 2 columns)
```

into PyTorch Geometric format:

```
2 rows × 14 columns
```

Meaning:

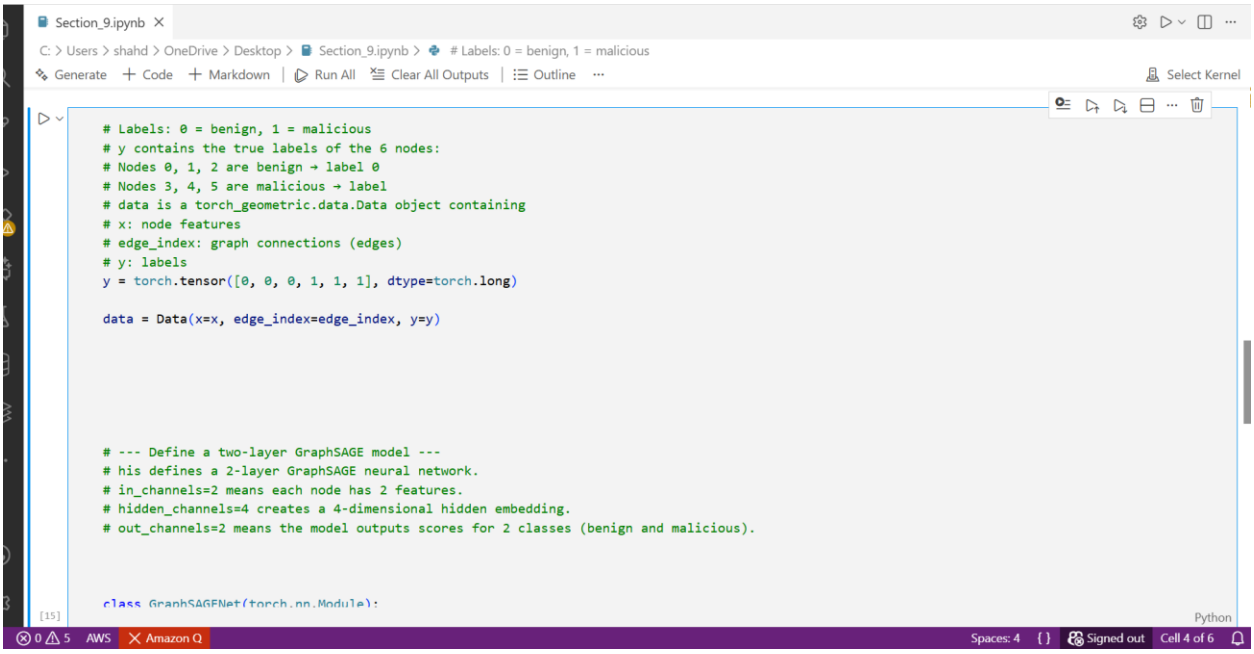
```
[ source nodes
  destination nodes ]
```

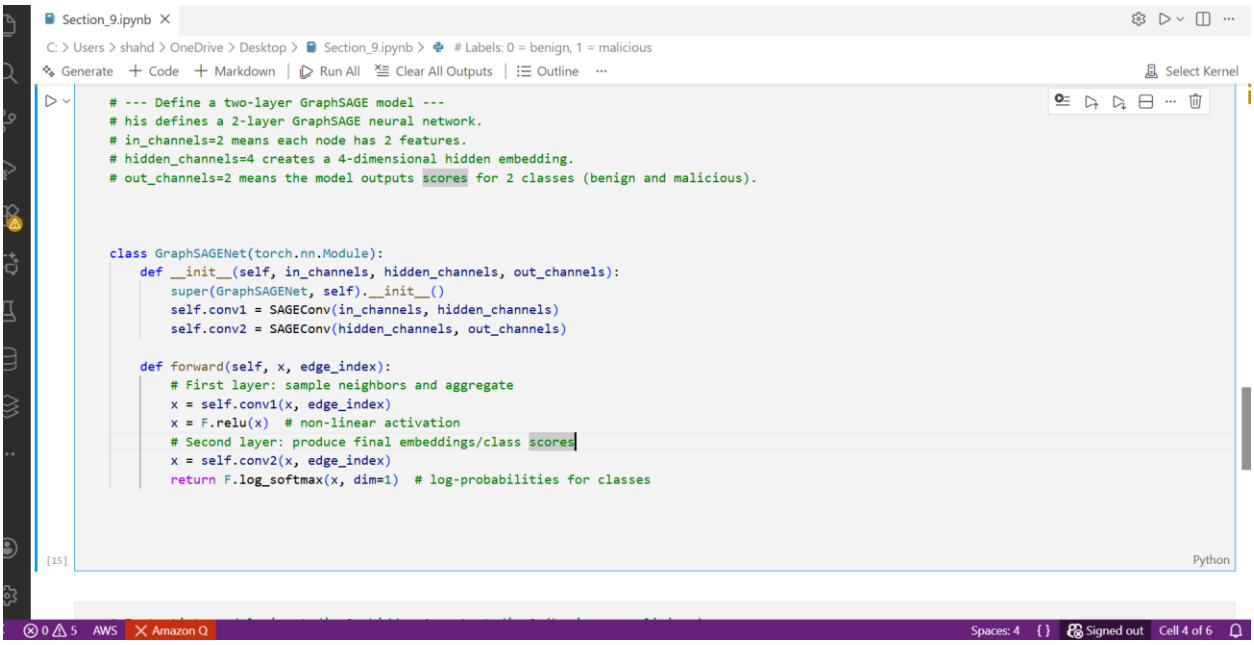
So final format is:

```
edge_index = [
    [0, 1, 1, 2, 0, 2, 3, ...],  # from
    [1, 0, 2, 1, 2, 0, 4, ...]   # to
```

]

This is exactly what torch_geometric expects.





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

What this line does:

It creates a tensor containing the true labels of each node in the graph.

dtype=torch.long means the values are stored as integer class labels, which is required by PyTorch.

Label meaning:

0 = benign

1 = malicious

So:

Nodes 0, 1, 2 → label 0 (benign)

Nodes 3, 4, 5 → label 1 (malicious)

This matches the node features you defined earlier.

Creating the Graph Data Object

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

Explanation:

This creates a **torch_geometric.data.Data** object.

It stores everything needed to represent a graph:

Attribute	Meaning
x	node features

`edge_index`	graph connectivity (edges)
y	node labels

This object will be passed to the GNN model during training or inference.

Graph Neural Network Definition — GraphSAGE Model

The following code defines a **2-layer GraphSAGE neural network**.

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)
```

Explanation:

- The model extends `torch.nn.Module`, which is the base class for neural networks in PyTorch.
- It consists of **two GraphSAGE layers**:

1. `self.conv1`
    a. Input: node features (size = `in_channels`)
    b. Output: hidden node embeddings (size = `hidden_channels`)
2. `self.conv2`
    a. Input: hidden embeddings
    b. Output: class predictions (size = `out_channels`)

Model Architecture Meaning

- `in_channels = 2`
  Because each node feature is a 2-dimensional vector:
  `[1,0]` or `[0,1]`
- `hidden_channels = 4`
  This layer transforms each node into a **4-dimensional representation**.
- `out_channels = 2`
  The model outputs scores for **two classes**:
  - class 0 → benign
  - class 1 → malicious

Forward Pass Explanation

```
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
```

Step-by-step :

First GraphSAGE layer

```
x = self.conv1(x, edge_index)
```

- The layer **gathers information from neighboring nodes**.
- It learns how nodes influence each other via graph connectivity.

Activation

```
x = F.relu(x)
```

- Applies **ReLU**, a non-linear activation function.
- Helps the model learn richer patterns.
- Prevents it from becoming a simple linear model.

Second GraphSAGE layer

```
x = self.conv2(x, edge_index)
```

- Produces final **node embeddings**
- Each node now has **2 output values** → one per class

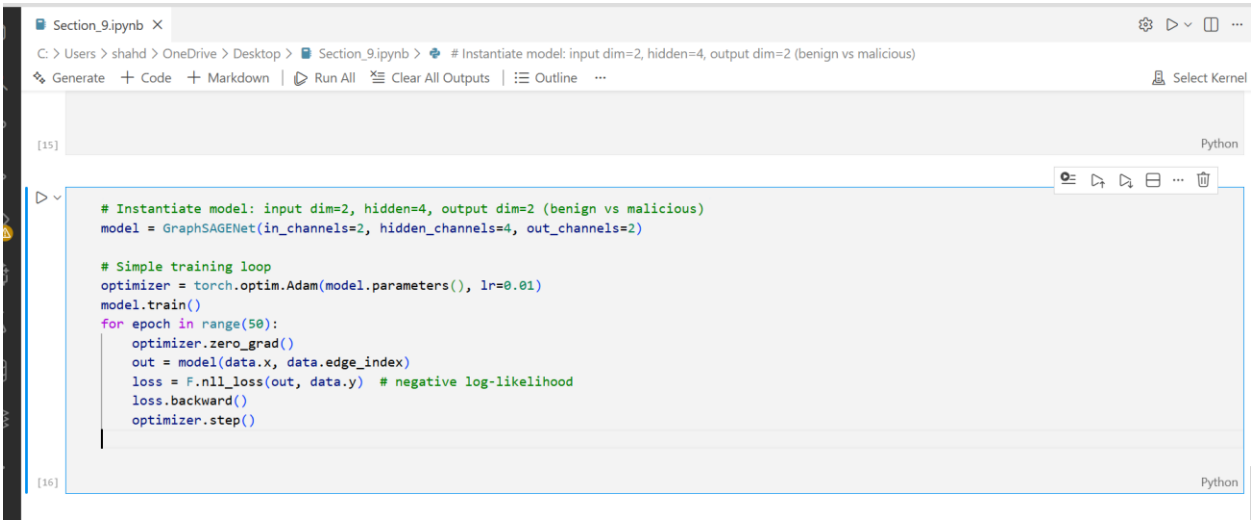
Log-Softmax

```
return F.log_softmax(x, dim=1)
```

- Converts raw scores into **log-probabilities**
- Often used with **negative log likelihood loss** for classification

For each node, output looks like:

```
[score_for_benign, score_for_malicious]
```



nstantiate the Model

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

Meaning:

You create an instance of your GraphSAGE neural network.

- `in_channels=2`
  → each node has 2 input features (e.g., `[1,0]` or `[0,1]`)
- `hidden_channels=4`
  → the first GraphSAGE layer produces 4-dimensional embeddings
- `out_channels=2`
  → the final layer outputs 2 values per node (class scores)
    - class 0 = benign
    - class 1 = malicious

So the model can classify each node into two categories.

Training Setup

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

Explanation:

- Uses the **Adam optimizer**, a common optimization algorithm used in deep learning.
- `model.parameters()` tells Adam which variables to update (weights of the network).
- `lr=0.01` sets the learning rate (how fast the model learns).

Start Training Mode

```
model.train()
```

This switches the model into training mode:

- Enables gradient tracking
- Activates dropout/batchnorm if present
- Prepares the network for learning

Training Loop

```
for epoch in range(50):
```

You train the model for **50 epochs**.
One epoch = one full training step over the data.

Since this is a tiny graph, we train on all nodes at once.

Step-by-step inside the loop :

Reset gradients

```
optimizer.zero_grad()
```

Gradients from the previous epoch are cleared.
Otherwise, new gradients accumulate, which breaks learning.

Forward pass

```
out = model(data.x, data.edge_index)
```

You feed the model the graph:

- `data.x` → node features
- `data.edge_index` → connections between nodes

The model returns predictions:

- Each node gets a vector of size 2
  → `[score_benign, score_malicious]`

These are **log-probabilities** because the model uses `log_softmax`.

Compute loss

```
loss = F.nll_loss(out, data.y)
```

- `F.nll_loss` = **Negative Log Likelihood loss**
- Used when the model outputs **log-probabilities**
- Compares:
  - the model's predictions (`out`)
  - the true labels (`data.y`)

It measures how wrong the predictions are.

Lower loss = better model.

Backpropagation

```
loss.backward()
```

This calculates gradients of the loss with respect to the model weights.

Update weights

```
optimizer.step()
```

Adam updates the model's parameters:

- weights move in the direction that reduces the loss

Switch the Model to Evaluation Mode

```
model.eval()
```

What it does:

- Puts the model into **evaluation mode**.
- This disables training features such as:
  - Dropout
  - Gradient tracking
- Ensures stable, deterministic predictions.

We do this because **we are no longer training**, only testing.

Generate Predictions

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

Step-by-step :

Forward pass

```
model(data.x, data.edge_index)
```

You pass:

- Node features (`data.x`)
- Graph structure (`data.edge_index`)

The model outputs a matrix of shape:

```
(number of nodes) × (number of classes)
```

Example output for a node:

```
[-0.2, -1.8]
```

These are log-probabilities for:

- class 0 → benign
- class 1 → malicious

Take the highest score

```
.argmax(dim=1)
```

- `argmax(dim=1)` selects the index of the highest value in each row.
- The index corresponds to the predicted class.

For example:

```
[0.2, -1.3] → 0 (benign)
[-0.8, 1.1] → 1 (malicious)
```

So you get a vector like:

```
tensor([0,0,0,1,1,1])
```

Print Predictions

```
print("Predicted labels:", pred.tolist())
```

- Converts the tensor to a Python list
- Prints the predicted class per node

Example output:

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

This means:

- Nodes 0, 1, 2 → predicted benign
- Nodes 3, 4, 5 → predicted malicious

Exactly matching the true labels.