

# Explanation of GraphSAGE Implementation using PyTorch Geometric

Code Analysis

December 3, 2025

## 1 Introduction

This document explains a small Graph Neural Network (GNN) implementation using the **PyTorch Geometric** library. The example demonstrates the GraphSAGE operator for binary node classification on a tiny synthetic graph. The goal is to classify nodes into two classes: “benign” and “malicious.”

I wrote this explanation to be practical and human-readable: not just a dry list of facts, but a short guide that points out the important engineering choices and pitfalls you should be aware of.

## 2 Data Construction

The example graph contains 6 nodes arranged into two tightly connected clusters with a single bridge edge between them. The graph is designed to be simple so that the demonstration focuses on the pipeline rather than dataset complexity.

### 2.1 Node Features ( $X$ )

The feature matrix has size  $6 \times 2$ . There are two separable patterns:

- **Benign nodes (0,1,2):** feature vector  $[1.0, 0.0]$ .
- **Malicious nodes (3,4,5):** feature vector  $[0.0, 1.0]$ .

### 2.2 Graph Topology (Edge Index)

Connectivity is defined in the `edge_index` tensor. Conceptually:

- Nodes 0,1,2 form a densely connected benign cluster.
- Nodes 3,4,5 form a densely connected malicious cluster.
- A single cross-edge connects node 2 to node 3 (bridge).

**Important implementation detail:** If the graph is intended to be undirected, each edge should be added in both directions. PyTorch Geometric does not automatically assume undirected edges; message passing will be asymmetric unless you explicitly include both directions.

Example snippet to convert to undirected edges:

```
1 # assume 'edge_index' is a 2 x E tensor
2 from torch_geometric.utils import to_undirected
3 edge_index = to_undirected(edge_index)
```

Listing 1: Make `edge_index` undirected

### 2.3 Labels ( $Y$ )

The ground-truth label vector is:

$$Y = [0, 0, 0, 1, 1, 1]$$

where 0 denotes benign and 1 denotes malicious.

## 3 Model Architecture: GraphSAGE

The model (named `GraphSAGENet`) is a straightforward two-layer GraphSAGE network.

### 3.1 High-level description

- First `SAGEConv`: maps input features (2) to hidden channels (4).
- Non-linearity: `ReLU`.
- Second `SAGEConv`: maps hidden channels (4) to output logits (2 classes).
- Final activation: `log_softmax` to get log-probabilities (works with `NLLLoss`).

### 3.2 Mathematical form

GraphSAGE aggregates neighbor representations and combines them with the node's own representation. A compact form of one layer's update:

$$h_v^{(l)} = \sigma\left(W \cdot \text{MEAN}\left(\{h_u^{(l-1)} : u \in \mathcal{N}(v)\} \cup \{h_v^{(l-1)}\}\right)\right), \quad (1)$$

where  $\mathcal{N}(v)$  denotes neighbors of  $v$ , and  $W$  is a learnable matrix. Implementation details (exact concat vs. mean, normalization, etc.) are handled by the `SAGEConv` module depending on parameters.

## 4 Training and Evaluation

### 4.1 Training Loop (practical)

The example trains for 50 epochs using Adam (`lr=0.01`) and `NLLLoss` over log-probabilities. Below is a typical training loop pattern you should follow and include in your code:

```
1 import torch
2 from torch_geometric.nn import SAGEConv
3 import torch.nn.functional as F
4
5 # reproducibility
6 torch.manual_seed(42)
7
8 model.train()
9 optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
10 loss_fn = torch.nn.NLLLoss() # used when model outputs log_softmax
11
12 for epoch in range(50):
13     optimizer.zero_grad()
14     out = model(data.x, data.edge_index) # log-probabilities
15     loss = loss_fn(out, data.y)
16     loss.backward()
17     optimizer.step()
18     # optional: print or log the loss every few epochs
```

Listing 2: Training loop pattern

### 4.2 Evaluation

Switch to evaluation mode, compute predictions using `argmax` on model outputs, and compute accuracy. Example:

```
1 model.eval()
2 with torch.no_grad():
3     out = model(data.x, data.edge_index) # log-probs
4     preds = out.argmax(dim=1)
5     print("Predicted labels:", preds.tolist())
```

Listing 3: Evaluation pattern

### 4.3 Loss function note

The implementation uses `log_softmax + NLLLoss`. That is perfectly fine. As an alternative, you can return raw logits and use `CrossEntropyLoss`, which internally applies `log_softmax` for numerical stability:

```
1 # alternative
2 loss_fn = torch.nn.CrossEntropyLoss()
3 # model returns logits (no log_softmax)
4 # loss = loss_fn(logits, labels)
```

## 5 Results and Interpretation

For this toy graph, the trained model predicts:

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

which corresponds to 100% accuracy on the six nodes.

**Crucial interpretation:** Achieving perfect accuracy on this dataset is expected because:

- The node features are linearly separable and align with graph clusters.
- The small size of the dataset allows the model to perfectly memorize the pattern.

Therefore, 100% here does not imply generalization to realistic or noisy graphs. Always test on held-out nodes or separate graphs to measure true generalization.

## 6 Additional Notes and Practical Considerations

Below are practical details and recommendations that make the code more robust and the explanation clearer to readers.

### 6.1 Graph Directionality

Explicitly state whether the graph is intended to be undirected. If so, ensure each edge is represented in both directions (use `to_undirected` or manually duplicate edges). Asymmetric edges change the message passing behavior and therefore the learned embeddings.

### 6.2 Training / Validation / Test split

In small demonstrative scripts it's common to run training and evaluation on the same nodes for clarity. However, when reporting performance, always clarify if you used the same nodes for training and testing. For realistic evaluation:

- Create boolean masks: `train_mask`, `val_mask`, and `test_mask`.
- Compute loss and metrics separately for each mask.

Example mask usage:

```
1 # compute loss only on training nodes
2 loss = loss_fn(out[train_mask], data.y[train_mask])
```

### 6.3 Reproducibility

Set random seeds to make experiments reproducible:

```
1 import random
2 import numpy as np
3 torch.manual_seed(42)
4 np.random.seed(42)
5 random.seed(42)
6 # if using CUDA:
7 # torch.cuda.manual_seed_all(42)
```

## 6.4 Why GraphSAGE here?

GraphSAGE is a good pedagogical choice because it highlights neighborhood aggregation and supports inductive settings (generating embeddings for nodes unseen during training). Although the present graph is tiny, understanding GraphSAGE prepares you for larger, real-world scenarios.

## 6.5 Hyperparameter choices

Choices like `hidden_channels=4`, `epochs=50`, and `lr=0.01` are simple, practical defaults for a toy example. For real datasets you would tune these via validation.

## 6.6 Device (CPU/GPU)

Make your code device-aware:

```
1 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
2 model.to(device)
3 data = data.to(device)
```

# 7 Caveats and Final Words

This notebook is an educational demonstration: it shows the full pipeline from constructing a graph to defining a GraphSAGE model and training it. The final 100% accuracy simply confirms the pipeline is correct for this synthetic, clean example. Do not mistake this for a proof of strength in realistic settings.