

1) Node Feature Matrix

```
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,
)
```

This creates a tensor that contains the features for each node in the graph.

Each row represents one node, and each node has two numerical features.

The values are manually assigned to distinguish between two categories of nodes.

2) Edge Index (Graph Connections)

```
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()
)

```

This block creates a tensor that defines all edges in the graph.

Each pair [a, b] represents a directed connection from node a to node b.

The .t() function transposes the tensor to match the graph

library's expected format, making the shape [2,

number_of_edges].

.contiguous() ensures the final tensor is stored in a continuous memory block for better performance.

3) Node Labels

```

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

```

This tensor stores the class label for each node.

The position in the list corresponds to the node index.

The values (0 or 1) represent the category each node belongs to.

4) Creating the Graph Data Object

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

This line creates a Data object that holds the entire graph:
node features, graph structure, and node labels.

This object will be passed into the model during training and
prediction.

5) Defining the GraphSAGE Neural Network

```
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)
```

This defines a custom neural network with two GraphSAGE
convolution layers.

The first layer takes the input features and produces a hidden
representation.

The second layer converts the hidden representation into final
class scores.

6) Forward Pass of the Model

```
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)
```

This function describes how the data flows through the network.

1. The first GraphSAGE layer processes the node features.
2. A ReLU activation is applied to introduce non-linearity.
3. The second GraphSAGE layer produces the final output for each node.
4. `log_softmax` converts the output into log-probabilities for each class

7) Creating the Model Instance

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

This initializes the neural network with:

- 2 input feature values per node,
- 4 hidden units,
- 2 output classes

8) Training Setup

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

The Adam optimizer is created to update the model's parameters during training.

`model.train()` switches the model into training mode.

9) Training Loop

```
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()
```

This loop trains the model for 50 iterations.

- `optimizer.zero_grad()` clears old gradients.
- `out` is the model's prediction for each node.
- `loss` measures how far the predictions are from the true labels.
- `loss.backward()` computes the gradients.
- `optimizer.step()` updates the model parameters based on the gradients.

10) Evaluating the Model

```
model.eval()
```

```
pred = model(data.x,
```

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

```
print("Predicted labels:",
```

```
pred.tolist())
```

This switches the model to evaluation mode.

The model processes the graph and produces predictions.

`argmax(dim=1)` selects the class with the highest score for each node.

Finally, the predicted labels are printed.