GNN architectures

Think of convolutional operations like taking a snapshot of a scene and smoothing out the details. In images, CNN layers create representations by zooming in on smaller and smaller areas of the image. For GCNs, we're doing the same thing, but instead of pixels, we're looking at neighborhoods of nodes.

We'll be talking about convolutional GNNs here, which is a whole family of GNNs that includes GraphSAGE and GCN. And just so you know, we'll be using GCN to specifically refer to the architecture that Thomas Kipf and Max Welling introduced.

> *Please refer to the Amazon products example notebook in our repository. Feel free to play with it. The main idea is to compare performances of GCN and GraphSAGE. The dataset is here (https://ogb.stanford.edu/), Data preparation steps include:*

- *Subset Graph Initialization: A new graph object is created to store a subset of data.*
- *Subset Graph Data: The subset graph contains edges, features, and labels of nodes with indices 0–9,999 from the original graph.*
- *Node Index Relabeling: Relabeling node indices within the subset graph to ensure consistency and avoid index mismatches.*
- *Importance of Relabeling: Crucial for GNN operations that heavily rely on indexing for processing node and edge information.*
- *Graph Construction: A new graph object is created.*
- *Node Feature and Label Assignment: Node features (x) and labels (y) are assigned to the graph object, derived from the original dataset based on specified subset indices.*
- *Edge Mask Purpose: Identify selected edges during subgraph creation.*
- *Edge Mask Application: Trace back to the original graph structure or perform structural analysis.*
- *Edge Mask Usage: Enable by setting the return_edge_mask option during subgraph extraction.*

GCN model

The GCN model makes use of the GCNConv layer, which is based on the graph convolution operation as explained by Kipf and Welling in their groundbreaking paper. This layer takes advantage of the special properties of graphs to help information move between nodes, enabling the model to learn representations that capture both the local graph structure and the individual node features.

```
class GCN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels,
out_channels):
    super(GCN, self).__init__()
    self.conv1 = GCNConv(in_channels, hidden_channels)
    self.conv2 = GCNConv(hidden_channels, out_channels)

    def forward(self, x, edge_index, \
return_embeds=False):
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        if return_embeds:
             return x

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

GraphSAGE model

While GCN uses the whole graph's adjacency matrix, GraphSAGE is built to learn from just a random sample of its neighbors. This makes it super good at handling big graphs!

GraphSAGE makes use of the SAGEConv layer, which is super versatile and can handle different aggregation functions like mean, pool, and even long short-term memory (LSTM). This means you can easily tweak how node features are combined!

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

```
    def __init__(self, in_channels, hidden_channels,
out_channels):
    super(GraphSAGE, self).__init__()
    self.conv1 = SAGEConv(in_channels, \
    hidden_channels)
    self.conv2 = SAGEConv(hidden_channels, out_channels)

    def forward(self, x, edge_index, \
    return_embeds=False):
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        if return_embeds:
            return x

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

To put together a product bundle, we use node embeddings! We start by picking a single product node and then discover the six products that are most similar to it. It's a four-step process:

```
- First, we'll use our trained GNN to create node embeddings for
each of our nodes.
```

- Next, we'll put these embeddings into a similarity matrix to see how similar each node is to our chosen product.
- Then, we'll sort the top five embeddings based on how similar they are to our product.
- Finally, we'll map the node indices of these top embeddings to product IDs.

```
gcn_model.eval()

with torch.no_grad():
     gcn_embeddings = gcn_model(subset_graph.x, \
subset_graph.edge_index, return_embeds=True)
```

```
gcn_similarity_matrix =
cosine_similarity(gcn_embeddings.cpu().numpy())
```

```
product_idx = 123
top_k = 6
top_k_similar_indices_gcn = np.argsort(-
gcn_similarity_matrix[product_idx])[:top_k]
```

Aggregation methods

- Basic Aggregation Methods: Mean, sum, and max applied over all layers.
- Advanced Aggregation Methods in PyG: Unique aggregations per layer, list aggregations, aggregation functions, and jumping knowledge networks (JK-Nets).

In GCN, we've already got a weighted average aggregation layer built in! If you're looking to tweak it, you can create a custom version of that layer.

In SAGEConv, the aggr parameter lets you choose how to aggregate data. You can pick from a bunch of options, like:

- Sum aggregation—Just a straightforward way to add up all the features of your neighbor nodes.
- Mean aggregation (default)—It calculates the average of the neighbor node features. This is super simple and works well for smoothing out any weird data points.
- Max aggregation—It picks the highest value from each feature among all your neighbors. This is handy when the most important features are more telling than the average ones, so you can grab the biggest signals from your neighbors.
- LSTM aggregation—This method is a bit more work and memory-heavy, but it uses an LSTM network to process the features of your ordered sequence of neighbor nodes.

```
class GraphSAGE(torch.nn.Module):
    def __init__(self, in_channels, \
hidden_channels, out_channels, agg_func='mean'):
        super(GraphSAGE, self).__init__()
        self.conv1 = SAGEConv(in_channels, \
hidden_channels, aggr=agg_func)
        self.conv2 = SAGEConv(hidden_channels, \
out_channels, aggr=agg_func)

    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)
```

List aggregations

How about we put ‘mean’, ‘max’, and ‘sum’ aggregations together in one layer? That way, we can get a snapshot of the neighborhood’s average, most important, and total structural stuff!

```
       self.conv1 = SAGEConv(in_channels,\
 hidden_channels, aggr=['max', 'sum', 'mean'])

       self.conv1 = SAGEConv(in_channels,\
 hidden_channels, aggr=[SoftmaxAggregation(),\
 StdAggregation() ])
```

Dropout

Dropout is a nifty trick in neural networks that helps keep them from getting too cozy with their favorite neurons. It randomly “drops” some units during training, which forces the model to learn from a wider range of data. This way, it’s better at handling new, unseen stuff!

During training, it randomly zeroes out some elements in the input tensor and the hidden-layer activations. The graph's topology is preserved, and only the neural network's activations are affected.

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

class GraphSAGEModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels,
dropout=0.5):
        super().__init__()
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, out_channels)
        self.dropout = dropout  # <-- Dropout rate (default is 0
in PyG)

    def forward(self, x, edge_index):
        # First GraphSAGE layer
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout, training=self.training)
# apply dropout

        # Second GraphSAGE layer
        x = self.conv2(x, edge_index)
        return x

# Example
x = torch.randn(10, 8)  # 10 nodes, 8 features
edge_index = torch.tensor([[0, 1, 2, 3, 4, 5],
                           [1, 2, 3, 4, 5, 0]])
model = GraphSAGEModel(8, 16, 2, dropout=0.5)
out = model(x, edge_index)
print(out.shape)
```

```
from torch_geometric.nn import GCNConv

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

```
    def __init__(self, in_channels, hidden_channels, out_channels,
dropout=0.5):
        super().__init__()
        self.conv1 = GCNConv(in_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, out_channels)
        self.dropout = dropout  # <-- default is 0 (no dropout)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = F.dropout(x, p=self.dropout, training=self.training)
        x = self.conv2(x, edge_index)
        return x

# Example
x = torch.randn(10, 8)
model = GCNModel(8, 16, 2, dropout=0.5)
out = model(x, edge_index)
print(out.shape)
```

Model depth

*Each layer allows nodes to aggregate information from their immediate neighbors, effectively increasing the **receptive field** by one hop per layer.*

Usually, GNNs with just 2 or 3 layers are pretty good at lots of things, like finding the right mix of info from their neighbors without making everything too smooth.

Under the hood

$$\text{Transform}(u) = \sigma\big(W_a * \text{Aggregate}(u)\big)$$

$$\text{Aggregate Function}(u) = \sum_{v \in \mathcal{N}(u)} h_v$$

$$\text{Update}(u) = h'_u = \text{Concat}(h_u, \text{Transform}(u))$$

GCN is a spectral-based GNN, while GraphSAGE uses a spatial approach. To really get a feel for how they differ, let's dive into how we can implement both of them!

$$m_{\mathcal{N}(u)} = \sum_{v \in \mathcal{N}(u)} \frac{h_v}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}}$$

$$\text{GCN Updated Node Embeddings} = h_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in \mathcal{N}(u)} \frac{h_v}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}}\right)$$

$$h_u^{(k)} = \sigma\left(\sum_{v \in \mathcal{N}(u)} \frac{1}{\sqrt{|\mathcal{N}(u)|\,|\mathcal{N}(v)|}}\left(W^{(k)} * h_v\right)\right)$$

Spectral	**Spatial**
Operation: performing a convolution using a graph's eigenvalues	**Operation:** aggregation of node features in node neighborhoods
• Must be undirected • Operation dependent on node features • Generally less computationally efficient	• Not required to be undirected • Operation not dependent on node features • Generally more computationally efficient

Reference

Graph Neural Networks in Action, K. Broadwater, 2025