A Graph Neural Network (GNN) is a type of neural network designed to work with graph-structured data.

In graphs:

- Nodes represent entities.
- Edges represent relationships between entities.
- Features can be associated with nodes (node features) or edges (edge features).

GNNs process these graphs by learning to represent nodes, edges, or the entire graph in a way that captures the graph structure and feature information. They are widely used in applications like social networks, molecular chemistry, and recommendation systems.

Key Concepts of GNNs

- Graph Representation: A graph is represented by:
  - An adjacency matrix that defines connections between nodes
  - A feature matrix that represents node-specific data
- Message Passing: GNNs operate by exchanging information ("messages") between neighboring nodes in the graph. Nodes aggregate information from their neighbors to update their own features.
- Layers: Each layer of a GNN updates the node features based on:
  - The node's current features.
  - The aggregated features of its neighbors.
- Output:
  - Node-level outputs (e.g., classification of individual nodes).
  - Edge-level outputs (e.g., predicting relationships between nodes).
  - Graph-level outputs (e.g., predicting a property of the entire graph).

```
#!pip install torch torchvision torchaudio torch-geometric
```

```
import torch
import torch.nn.functional as F
from torch_geometric.nn import GCNConv
from torch_geometric.datasets import Planetoid
```

```
# Load the Cora dataset (common benchmark dataset for GNNs)
dataset = Planetoid(root="/tmp/Cora", name="Cora")
```

Dataset:

In this example, we use the Cora dataset:

1. Nodes represent research papers.
2. Edges represent citation relationships.
3. Node features are word vectors from the paper abstracts.
4. Labels represent the research field of each paper.

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

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        # First Graph Convolution + ReLU
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        # Second Graph Convolution + Softmax
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

GCNConv: A Graph Convolutional Layer (from PyTorch Geometric). It implements message passing to update node features using neighbors' features. * in_channels: Number of features for each node. * hidden_channels: Intermediate representation size. * out_channels: Number of classes for classification.

- data.x: Node features (input to the model).
- data.edge_index: Connectivity of the graph.
- Message Passing:
  - In self.conv1, node features are updated by aggregating features from neighbors.
  - In self.conv2, the aggregated features are transformed again to predict class probabilities.

```
# Load dataset and model
data = dataset[0]  # Cora has only one graph
model = GNN(in_channels=dataset.num_node_features, hidden_channels=16, out_channels=dataset.num_classes)
```

```
# Define optimizer and loss function
optimizer = torch.optim.Adam(model.parameters(), lr=0.01, weight_decay=5e-4)
criterion = torch.nn.NLLLoss()
```

```
# Training loop
def train():
    model.train()
    optimizer.zero_grad()
    out = model(data)
    loss = criterion(out[data.train_mask], data.y[data.train_mask])
    loss.backward()
    optimizer.step()
    return loss.item()
```

- Loss Function: NLLLoss is used because the output is log probabilities (log_softmax).
- Train Mask: Specifies which nodes to use for training.

```
# Test function
def test():
    model.eval()
    out = model(data)
    pred = out.argmax(dim=1)  # Get predictions
    correct = pred[data.test_mask] == data.y[data.test_mask]  # Compare with true labels
    acc = int(correct.sum()) / int(data.test_mask.sum())
    return acc
```

```
# Training the model
for epoch in range(200):
    loss = train()
    acc = test()
    if epoch % 10 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}, Test Accuracy: {acc:.4f}")

print("Training complete!")
```

```
Epoch 0, Loss: 1.9455, Test Accuracy: 0.5200
Epoch 10, Loss: 0.6057, Test Accuracy: 0.8040
Epoch 20, Loss: 0.1160, Test Accuracy: 0.7950
Epoch 30, Loss: 0.0300, Test Accuracy: 0.7920
Epoch 40, Loss: 0.0156, Test Accuracy: 0.7950
Epoch 50, Loss: 0.0136, Test Accuracy: 0.8020
Epoch 60, Loss: 0.0148, Test Accuracy: 0.8090
Epoch 70, Loss: 0.0164, Test Accuracy: 0.8070
Epoch 80, Loss: 0.0171, Test Accuracy: 0.8090
Epoch 90, Loss: 0.0167, Test Accuracy: 0.8090
Epoch 100, Loss: 0.0158, Test Accuracy: 0.8120
Epoch 110, Loss: 0.0149, Test Accuracy: 0.8130
Epoch 120, Loss: 0.0141, Test Accuracy: 0.8120
Epoch 130, Loss: 0.0134, Test Accuracy: 0.8060
Epoch 140, Loss: 0.0128, Test Accuracy: 0.8040
Epoch 150, Loss: 0.0123, Test Accuracy: 0.8050
Epoch 160, Loss: 0.0118, Test Accuracy: 0.8050
Epoch 170, Loss: 0.0114, Test Accuracy: 0.8060
Epoch 180, Loss: 0.0110, Test Accuracy: 0.8050
Epoch 190, Loss: 0.0107, Test Accuracy: 0.8060
Training complete!
```

How is the GNN Implemented?

1. Graph Representation: The input graph is represented using:
    - data.x: Feature matrix.
    - data.edge_index: Connectivity matrix (edges).
2. Message Passing: Implemented through GCNConv layers. Each layer aggregates information from neighbors to update node features.
3. Node Classification: The model learns to classify nodes into categories by training on labeled nodes (train_mask) and testing on a separate set of nodes (test_mask).
4. Training Pipeline:
    - Forward pass through the GNN.
    - Compute the loss for labeled nodes.
    - Backpropagate to update parameters.