Graph Neural Networks for Clustering Irregular Antenna Arrays

A Complete Step-by-Step Guide from First Principles

Technical Reference Document

January 24, 2026

Abstract

This document provides a comprehensive, step-by-step guide to clustering irregular 16×16 antenna arrays using Graph Neural Networks (GNNs). We begin with a brief explanation of why Physics-Informed Neural Networks (PINNs) are unsuitable for this task, then develop the GNN approach from absolute fundamentals—assuming no prior knowledge of graph theory or neural networks. Every concept is built incrementally with detailed explanations, visualizations, and concrete examples.

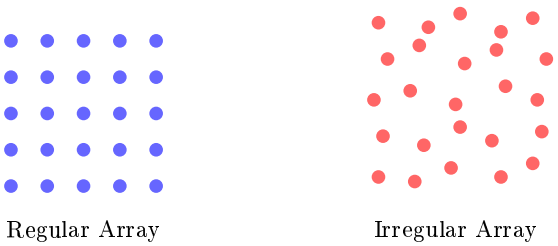
Contents

1 Understanding the Problem

1.1 What Do We Have?

You have an **irregular** 16×16 antenna array. This means:

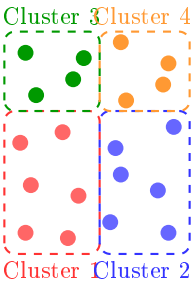
- $N = 256$ individual antenna elements
- Each element has a position (x_i, y_i) in 2D space
- "Irregular" means the elements are **not** on a perfect grid—they have varying spacing, possible rotations, or non-uniform placement



Regular Array Irregular Array

1.2 What Do We Want?

Clustering means grouping the 256 elements into K distinct clusters (groups) based on some criteria. For example, if $K = 4$:



1.3 Why Cluster Antenna Arrays?

Clustering serves practical purposes in antenna system design:

1. **Subarraying:** Instead of controlling 256 elements individually, control K groups (reduces computational complexity from $O(N)$ to $O(K)$)
2. **Beamforming simplification:** Apply the same weights to elements in a cluster
3. **Fault tolerance:** If one element fails, its cluster can compensate
4. **Hardware mapping:** Assign clusters to physical processing units or RF chains

2 Why Not Use Physics-Informed Neural Networks?

Warning

Short Answer: PINNs solve the *wrong type of problem*. They are designed for differential equations, not geometric partitioning.

2.1 What PINNs Are Designed For

Physics-Informed Neural Networks (PINNs) embed physical laws—typically **Partial Differential Equations (PDEs)**—into the training loss. They excel at:

- Solving PDEs (heat equation, wave equation, Navier-Stokes)
- Discovering unknown PDE parameters from data
- Surrogate modeling for physical simulations

The core PINN loss function looks like:

$$\mathcal{L}_{\text{PINN}} = \underbrace{\|u_{\text{predicted}} - u_{\text{data}}\|^2}_{\text{data fitting}} + \lambda \underbrace{\left\| \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} - f \right\|^2}_{\text{PDE residual (e.g., Laplace equation)}} \tag{1}$$

2.2 Why This Fails for Clustering

PINN Requirements	Clustering Reality
Continuous output field $u(x, y)$	Discrete labels $\{1, 2, \ldots, K\}$
Physical law as PDE	No governing PDE exists
Spatial derivatives ∇u, $\nabla^2 u$	Derivatives are meaningless for labels
Smooth solutions preferred	Cluster boundaries are discontinuous

Bottom line: Clustering is a *combinatorial optimization* problem over a *discrete space*. PINNs operate in *continuous function spaces* governed by *differential equations*. These are fundamentally incompatible paradigms.

The right tool: Graph Neural Networks naturally handle discrete structures, learn from local neighborhoods, and are designed for exactly this type of problem.

3 Graph Neural Networks: Building from Zero

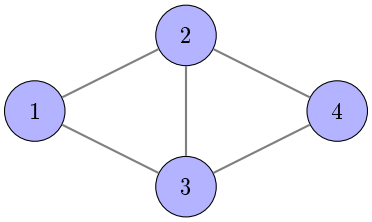
We will now build up the entire GNN framework step by step, assuming you have never seen a graph or neural network before.

3.1 Step 1: What is a Graph?

Step 1: Understanding Graphs

A **graph** is a mathematical structure consisting of:

- **Nodes** (also called vertices): The "things" we care about
- **Edges**: Connections between nodes



A simple graph with 4 nodes and 5 edges

We write this mathematically as:

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}) \tag{2}$$

$$\mathcal{V} = \{1, 2, 3, 4\} \quad \text{(set of nodes)} \tag{3}$$

$$\mathcal{E} = \{(1,2), (1,3), (2,3), (2,4), (3,4)\} \quad \text{(set of edges)} \tag{4}$$

3.2 Step 2: Your Antenna Array as a Graph

Step 2: Converting Antennas to a Graph

Each antenna element becomes a **node**. We create **edges** between antennas that are "related" (e.g., physically close to each other).

For your 16×16 array:

- $|\mathcal{V}| = N = 256$ nodes (one per antenna)
- Each node i has a **feature vector** $\mathbf{x}_i$ (at minimum, its position)

Node Features

The **feature vector** of node i contains all the information we know about that antenna:

$$\mathbf{x}_i = \begin{pmatrix} x_i \\ y_i \end{pmatrix} \in \mathbb{R}^2 \quad \text{(just position)} \tag{5}$$

Or with more information:

$$\mathbf{x}_i = \begin{pmatrix} x_i \\ y_i \\ \theta_i \\ P_i \\ \vdots \end{pmatrix} \in \mathbb{R}^d \quad \text{(position + orientation + power + ...)} \tag{6}$$

3.3 Step 3: How to Create Edges

Step 3: Building the Edge Set

We must decide which antennas are "connected." There is no single correct answer—it depends on your application. Here are common strategies:

3.3.1 Strategy A: k-Nearest Neighbors (k-NN)

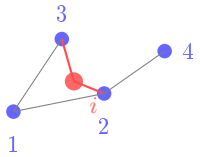
Connect each node to its k closest neighbors by Euclidean distance.

k-NN Edge Construction

For each node i:

$$\mathcal{N}_k(i) = \underset{j \neq i}{\arg\,\text{top-}k} \left(-\|\mathbf{p}_i - \mathbf{p}_j\|_2\right) \tag{7}$$

Then: $\mathcal{E} = \{(i, j) : j \in \mathcal{N}_k(i)\}$
Typical choice: $k = 6$ to 12 for a 256-element array.



Node i with $k = 2$ nearest neighbors

3.3.2 Strategy B: Radius-Based (ϵ-ball)

Connect nodes within distance ϵ:

$$\mathcal{E} = \{(i, j) : \|\mathbf{p}_i - \mathbf{p}_j\|_2 < \epsilon,\ i \neq j\} \tag{8}$$

3.3.3 Strategy C: Mutual Coupling Threshold

If you have computed the mutual coupling matrix $\mathbf{M}$ from electromagnetic simulation:

$$\mathcal{E} = \{(i, j) : |M_{ij}| > \tau\} \tag{9}$$

where τ is a threshold (connect strongly-coupled elements).

3.4 Step 4: The Adjacency Matrix

Step 4: Matrix Representation of Graphs

For computation, we represent the graph as an **adjacency matrix** $\mathbf{A} \in \mathbb{R}^{N \times N}$.

Adjacency Matrix Definition

$$A_{ij} = \begin{cases} 1 & \text{if edge } (i, j) \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Example: Adjacency Matrix

For the 4-node graph from Step 1:

$$\mathbf{A} = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix} \tag{11}$$

Row i, column j: is there an edge from i to j?

We also define the **degree matrix**:

$$D_{ii} = \sum_{j=1}^{N} A_{ij} = \text{number of neighbors of node } i \tag{12}$$

3.5 Step 5: What is a Neural Network? (Brief Review)

Step 5: Neural Network Basics

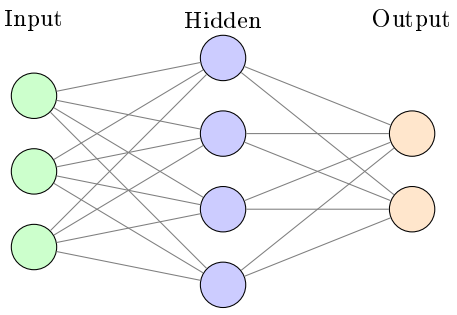
A neural network is a function f_θ that transforms inputs to outputs through layers of linear transformations and non-linear activations.

A single layer:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{13}$$

where:

- $\mathbf{x} \in \mathbb{R}^{d_{\text{in}}}$: input vector
- $\mathbf{W} \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$: learnable weight matrix
- $\mathbf{b} \in \mathbb{R}^{d_{\text{out}}}$: learnable bias vector
- σ: non-linear activation function (e.g., ReLU, tanh)
- $\mathbf{h} \in \mathbb{R}^{d_{\text{out}}}$: output vector



3.6 Step 6: The Key Idea of GNNs—Message Passing

Step 6: Message Passing (The Core Concept)

In a regular neural network, each input is processed independently. In a GNN, each node **aggregates information from its neighbors** to update its representation. This is called **message passing**.

Key Insight

The fundamental insight: *A node's representation should depend on its neighbors.* In antenna arrays, this means: an element's cluster assignment should be influenced by the elements around it.

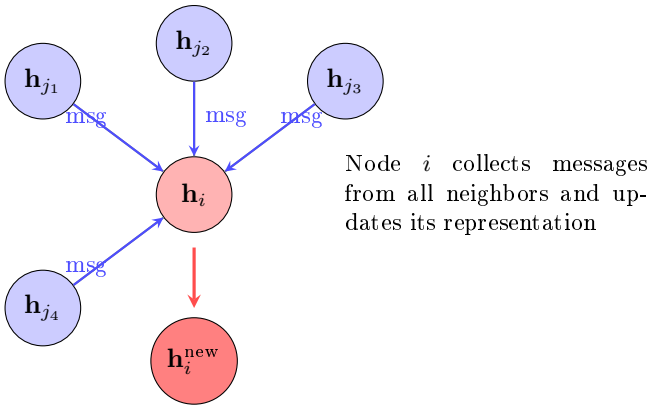
The general message passing formula for one layer:

Message Passing Equation

$$\boxed{\mathbf{h}_i^{(\ell+1)} = \text{UPDATE}\left(\mathbf{h}_i^{(\ell)}, \text{AGGREGATE}\left(\left\{\mathbf{h}_j^{(\ell)} : j \in \mathcal{N}(i)\right\}\right)\right)} \tag{14}$$

where:

- $\mathbf{h}_i^{(\ell)}$: representation (embedding) of node i at layer ℓ
- $\mathcal{N}(i)$: the neighbors of node i (nodes connected by an edge)
- AGGREGATE: combines neighbor information (sum, mean, max, attention)
- UPDATE: combines self-information with aggregated neighbor information



3.7 Step 7: A Concrete GNN Layer—Graph Convolutional Network (GCN)

Step 7: The GCN Layer

The Graph Convolutional Network (GCN) is the simplest and most foundational GNN architecture.

GCN Layer Equation

$$\mathbf{H}^{(\ell+1)} = \sigma\left(\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}\mathbf{H}^{(\ell)}\mathbf{W}^{(\ell)}\right) \tag{15}$$

Let's break this down piece by piece:

- $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$: adjacency matrix with self-loops added
- $\tilde{\mathbf{D}}$: degree matrix of $\tilde{\mathbf{A}}$
- $\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$: normalized adjacency (prevents exploding/vanishing values)
- $\mathbf{H}^{(\ell)} \in \mathbb{R}^{N \times d_\ell}$: all node embeddings at layer ℓ
- $\mathbf{W}^{(\ell)} \in \mathbb{R}^{d_\ell \times d_{\ell+1}}$: learnable weight matrix
- σ: activation function (ReLU or similar)

What does this actually do? For a single node i:

$$\mathbf{h}_i^{(\ell+1)} = \sigma\left(\mathbf{W}^{(\ell)\top} \cdot \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{\mathbf{h}_j^{(\ell)}}{\sqrt{|\mathcal{N}(i)| \cdot |\mathcal{N}(j)|}}\right) \tag{16}$$

In plain English: *"Average your neighbors' features (including yourself), transform with a learned matrix, apply non-linearity."*

3.8 Step 8: Graph Attention Networks (GAT)—Learning Which Neighbors Matter

Step 8: Attention Mechanism

GCN treats all neighbors equally. But some neighbors are more important than others! GAT learns **attention weights** to focus on relevant neighbors.

GAT Layer Equations

Step 8.1: Compute attention coefficients between node i and neighbor j:

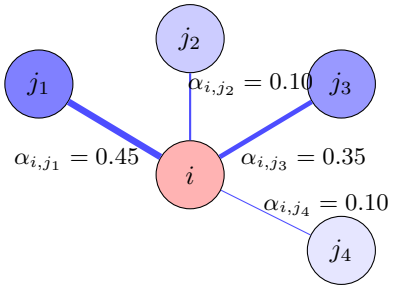
$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_i \,\|\, \mathbf{W}\mathbf{h}_j]\right) \tag{17}$$

where $\|$ denotes concatenation, $\mathbf{a}$ is a learnable attention vector.
Step 8.2: Normalize with softmax over neighbors:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})} \tag{18}$$

Now $\sum_{j \in \mathcal{N}(i)} \alpha_{ij} = 1$.
Step 8.3: Update node representation:

$$\mathbf{h}_i^{(\ell+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{W}\mathbf{h}_j^{(\ell)}\right) \tag{19}$$



Edge thickness $\propto$ attention weight

Multi-head attention: Run K independent attention mechanisms and concatenate:

$$\mathbf{h}_i^{(\ell+1)} = \Big\|_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(k)} \mathbf{W}^{(k)} \mathbf{h}_j^{(\ell)}\right) \tag{20}$$

3.9 Step 9: Edge Features—Incorporating Physics

Step 9: Adding Edge Information

So far, edges just indicate connectivity (binary: connected or not). But edges can carry **information**! For antennas, this includes distance, mutual coupling, etc.

Physics-Informed Edge Features

For each edge (i, j), define a feature vector:

$$\mathbf{e}_{ij} = \begin{pmatrix} d_{ij} \\ |M_{ij}| \\ \angle M_{ij} \\ \cos(\theta_i - \theta_j) \end{pmatrix} = \begin{pmatrix} \text{Euclidean distance} \\ \text{Mutual coupling magnitude} \\ \text{Mutual coupling phase} \\ \text{Orientation similarity} \end{pmatrix} \tag{21}$$

Modified message passing with edge features:

$$\mathbf{h}_i^{(\ell+1)} = \sigma\left(\sum_{j\in\mathcal{N}(i)} \alpha_{ij} \cdot \text{MLP}\left(\left[\mathbf{W}\mathbf{h}_j^{(\ell)} \,\|\, \mathbf{e}_{ij}\right]\right)\right) \tag{22}$$

This allows the network to learn that, for example, *"strongly coupled elements should be weighted more heavily"* or *"distant elements contribute less."*

3.10 Step 10: From Node Embeddings to Cluster Assignments

Step 10: The Output Layer

After L GNN layers, each node has a rich embedding $\mathbf{h}_i^{(L)} \in \mathbb{R}^{d_L}$. Now we must convert this to a cluster assignment.

Soft Cluster Assignment

Apply a final linear layer + softmax to get cluster probabilities:

$$\mathbf{z}_i = \text{softmax}\left(\mathbf{W}_{\text{out}}\mathbf{h}_i^{(L)} + \mathbf{b}_{\text{out}}\right) \in \mathbb{R}^K \tag{23}$$

where:

- $\mathbf{W}_{\text{out}} \in \mathbb{R}^{K \times d_L}$: output weight matrix
- $\mathbf{z}_i = (z_{i1}, z_{i2}, \ldots, z_{iK})$: probability distribution over K clusters
- $z_{ik} = P(\text{node } i \text{ belongs to cluster } k)$
- $\sum_{k=1}^{K} z_{ik} = 1$ (guaranteed by softmax)

We collect all assignments into a matrix:

$$\mathbf{Z} = \begin{pmatrix} z_{11} & z_{12} & \cdots & z_{1K} \\ z_{21} & z_{22} & \cdots & z_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ z_{N1} & z_{N2} & \cdots & z_{NK} \end{pmatrix} \in \mathbb{R}^{N \times K} \tag{24}$$

Row i is the cluster probability distribution for antenna i.

3.11 Step 11: The Loss Function—How Do We Train Without Labels?

Step 11: Unsupervised Clustering Loss

In **unsupervised** clustering, we have no ground truth labels. We must design a loss function that encourages "good" clusters based on the graph structure alone.

The key principle: **good clusters minimize edges cut between them.**

Good Clustering **Bad Clustering**

1 edge cut 5 edges cut

3.11.1 Loss Component 1: MinCut Loss

The **normalized cut** measures how many edges cross cluster boundaries:

MinCut Loss

$$\mathcal{L}_{\text{cut}} = -\frac{\text{Tr}(\mathbf{Z}^\top \mathbf{A}\mathbf{Z})}{\text{Tr}(\mathbf{Z}^\top \mathbf{D}\mathbf{Z})} \tag{25}$$

Intuition:

- $\mathbf{Z}^\top\mathbf{A}\mathbf{Z}$: measures edges *within* clusters (we want this HIGH)
- $\mathbf{Z}^\top\mathbf{D}\mathbf{Z}$: normalization by cluster sizes

- Negative sign: we minimize the loss, so we maximize within-cluster edges

Detailed derivation:

$$(\mathbf{Z}^\top \mathbf{A}\mathbf{Z})_{kk} = \sum_{i=1}^{N}\sum_{j=1}^{N} z_{ik} \cdot A_{ij} \cdot z_{jk} \tag{26}$$

$$= \sum_{\text{edge } (i,j)} z_{ik} \cdot z_{jk} \tag{27}$$

$$= \text{“soft count” of edges where both endpoints are in cluster } k \tag{28}$$

3.11.2 Loss Component 2: Orthogonality Loss

Prevent the trivial solution where all nodes go to one cluster:

Orthogonality Loss

$$\mathcal{L}_{\text{ortho}} = \left\| \frac{\mathbf{Z}^\top \mathbf{Z}}{N} - \frac{\mathbf{I}_K}{K} \right\|_F^2 \tag{29}$$

Intuition:

- $\mathbf{Z}^\top \mathbf{Z} \in \mathbb{R}^{K \times K}$: measures cluster overlap
- We want $\mathbf{Z}^\top \mathbf{Z} \approx \frac{N}{K}\mathbf{I}_K$ (diagonal, equal sizes)
- This encourages balanced, non-overlapping clusters

3.11.3 Loss Component 3: Entropy Regularization (Optional)

Encourage confident (non-uniform) assignments:

$$\mathcal{L}_{\text{entropy}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K} z_{ik} \log(z_{ik} + \epsilon) \tag{30}$$

Low entropy = confident predictions (one z_{ik} close to 1, others close to 0).

3.11.4 Total Loss Function

Complete Loss Function

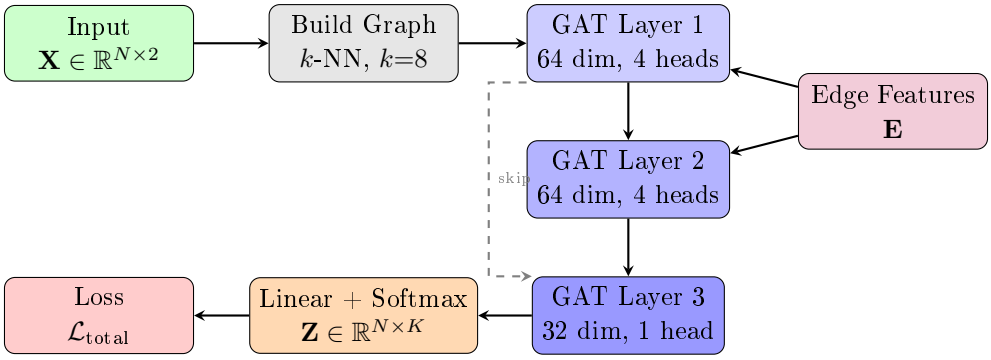
$$\boxed{\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{cut}} + \lambda_1 \mathcal{L}_{\text{ortho}} + \lambda_2 \mathcal{L}_{\text{entropy}}} \tag{31}$$

Typical values: $\lambda_1 = 1.0$, $\lambda_2 = 0.1$

3.12 Step 12: Complete Architecture Summary

Step 12: Putting It All Together

Here is the complete GNN architecture for antenna array clustering:



Layer-by-layer breakdown:

1. **Input:** Antenna positions $\mathbf{X} \in \mathbb{R}^{256\times 2}$
2. **Graph Construction:** Build k-NN graph ($k = 8$), compute edge features
3. **GAT Layer 1:** $2 \to 64$ dimensions, 4 attention heads, output: $\mathbb{R}^{256\times 256}$
4. **GAT Layer 2:** $256 \to 64$ dimensions, 4 attention heads, output: $\mathbb{R}^{256\times 256}$
5. **GAT Layer 3:** $256 \to 32$ dimensions, 1 attention head, output: $\mathbb{R}^{256\times 32}$
6. **Output Layer:** Linear $32 \to K$ + Softmax, output: $\mathbb{R}^{256\times K}$

3.13 Step 13: Training Algorithm

Step 13: How to Train the Network

Algorithm 1 GNN Training for Antenna Array Clustering

Require: Positions $\mathbf{P} \in \mathbb{R}^{N\times 2}$, mutual coupling $\mathbf{M}$ (optional), clusters K
Ensure: Trained model, cluster assignments

// **Step A: Preprocessing**
Normalize positions: $\mathbf{P} \leftarrow (\mathbf{P} - \mu)/\sigma$
Build k-NN graph: $\mathcal{E} \leftarrow \text{kNN}(\mathbf{P}, k = 8)$
Compute adjacency $\mathbf{A}$ and degree $\mathbf{D}$
Compute edge features: $\mathbf{e}_{ij} \leftarrow [d_{ij}, |M_{ij}|, \ldots]$ for $(i, j) \in \mathcal{E}$

// **Step B: Initialize Network**
Initialize GAT layers with Xavier/Glorot initialization
Set learning rate $\eta = 0.001$, weight decay $= 0.0005$

// **Step C: Training Loop**
for epoch $= 1$ to 500 **do**
    // **Forward pass**
    $\mathbf{H}^{(0)} \leftarrow \mathbf{P}$
    for $\ell = 1$ to L **do**
        $\mathbf{H}^{(\ell)} \leftarrow \text{GATLayer}_\ell(\mathbf{H}^{(\ell-1)}, \mathbf{A}, \mathbf{E})$
        $\mathbf{H}^{(\ell)} \leftarrow \text{ELU}(\mathbf{H}^{(\ell)})$ ▷ Activation
        $\mathbf{H}^{(\ell)} \leftarrow \text{Dropout}(\mathbf{H}^{(\ell)}, p = 0.1)$ ▷ Regularization
    end for
    $\mathbf{Z} \leftarrow \text{Softmax}(\text{Linear}(\mathbf{H}^{(L)}))$

    // **Compute loss**
    $\mathcal{L}_{\text{cut}} \leftarrow -\text{Tr}(\mathbf{Z}^\top \mathbf{A}\mathbf{Z})/\text{Tr}(\mathbf{Z}^\top \mathbf{D}\mathbf{Z})$
    $\mathcal{L}_{\text{ortho}} \leftarrow \|\mathbf{Z}^\top \mathbf{Z}/N - \mathbf{I}_K/K\|_F^2$
    $\mathcal{L} \leftarrow \mathcal{L}_{\text{cut}} + \lambda_1 \mathcal{L}_{\text{ortho}}$

    // **Backward pass**
    $\nabla_\Theta \mathcal{L} \leftarrow \text{Backpropagation}(\mathcal{L})$
    $\Theta \leftarrow \Theta - \eta \cdot \text{Adam}(\nabla_\Theta \mathcal{L})$
end for

// **Step D: Extract Final Clusters**
$c_i \leftarrow \arg\max_k z_{ik}$ for all $i = 1, \ldots, N$
return cluster assignments $\{c_1, c_2, \ldots, c_N\}$

3.14 Step 14: From Soft to Hard Assignments

Step 14: Final Cluster Assignment

The network outputs soft probabilities. For the final clustering, we take the hard assignment:

$$c_i = \arg \max_{k \in \{1,\ldots,K\}} z_{ik} \tag{32}$$

Example

If node 42 has $\mathbf{z}_{42} = (0.05, 0.82, 0.08, 0.05)$ for $K = 4$ clusters:

$$c_{42} = \arg\max(0.05, 0.82, 0.08, 0.05) = 2 \tag{33}$$

Antenna element 42 is assigned to Cluster 2.

4 Implementation Guide

4.1 Recommended Libraries

- **PyTorch Geometric (PyG):** Most popular GNN library

```
pip install torch-geometric
```

- **Deep Graph Library (DGL):** Alternative, good for large graphs

```
pip install dgl
```

4.2 Complete PyTorch Geometric Code

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_geometric.nn import GATConv, knn_graph
from torch_geometric.data import Data

class AntennaClusteringGNN(nn.Module):
    def __init__(self, in_dim=2, hidden_dim=64,
                 num_clusters=4, heads=4, dropout=0.1):
        super().__init__()

        # GAT layers
        self.conv1 = GATConv(in_dim, hidden_dim, heads=heads,
                             dropout=dropout)
        self.conv2 = GATConv(hidden_dim * heads, hidden_dim,
                             heads=heads, dropout=dropout)
        self.conv3 = GATConv(hidden_dim * heads, hidden_dim,
                             heads=1, dropout=dropout)

        # Output layer
        self.classifier = nn.Linear(hidden_dim, num_clusters)
        self.dropout = dropout

    def forward(self, x, edge_index):
        # Layer 1
        h = self.conv1(x, edge_index)
        h = F.elu(h)
        h = F.dropout(h, p=self.dropout, training=self.training)
```

```
        # Layer 2
        h = self.conv2(h, edge_index)
        h = F.elu(h)
        h = F.dropout(h, p=self.dropout, training=self.training)

        # Layer 3
        h = self.conv3(h, edge_index)
        h = F.elu(h)

        # Soft cluster assignments
        z = F.softmax(self.classifier(h), dim=-1)
        return z


def mincut_loss(z, adj, deg):
    """Normalized MinCut loss."""
    # z: (N, K) soft assignments
    # adj: (N, N) adjacency matrix
    # deg: (N, N) degree matrix (diagonal)

    num = torch.trace(z.T @ adj @ z)
    denom = torch.trace(z.T @ deg @ z)
    return -num / (denom + 1e-8)


def orthogonality_loss(z):
    """Encourage balanced, non-overlapping clusters."""
    n, k = z.shape
    identity = torch.eye(k, device=z.device) / k
    cluster_sim = (z.T @ z) / n
    return torch.norm(cluster_sim - identity, p='fro') ** 2


def train_clustering(positions, k_neighbors=8, num_clusters=4,
                     epochs=500, lr=0.001):
    """
    Main training function.

    Args:
        positions: (N, 2) tensor of antenna positions
        k_neighbors: number of neighbors for graph construction
        num_clusters: K, number of clusters
        epochs: training iterations
        lr: learning rate

    Returns:
        cluster_assignments: (N,) tensor of cluster labels
    """
    device = torch.device('cuda' if torch.cuda.is_available()
                          else 'cpu')
```

```
# Normalize positions
positions = (positions - positions.mean(0)) / positions.std(0)
positions = positions.to(device)

# Build k-NN graph
edge_index = knn_graph(positions, k=k_neighbors, loop=False)

# Build adjacency and degree matrices
n = positions.shape[0]
adj = torch.zeros(n, n, device=device)
adj[edge_index[0], edge_index[1]] = 1
adj = (adj + adj.T) / 2  # Symmetrize
deg = torch.diag(adj.sum(dim=1))

# Initialize model
model = AntennaClusteringGNN(
    in_dim=2,
    hidden_dim=64,
    num_clusters=num_clusters,
    heads=4
).to(device)

optimizer = torch.optim.Adam(model.parameters(), lr=lr,
                             weight_decay=5e-4)

# Training loop
model.train()
for epoch in range(epochs):
    optimizer.zero_grad()

    # Forward pass
    z = model(positions, edge_index)

    # Compute losses
    loss_cut = mincut_loss(z, adj, deg)
    loss_ortho = orthogonality_loss(z)
    loss = loss_cut + 1.0 * loss_ortho

    # Backward pass
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 50 == 0:
        print(f"Epoch {epoch+1}: Loss = {loss.item():.4f}")

# Extract hard assignments
model.eval()
with torch.no_grad():
    z = model(positions, edge_index)
    clusters = z.argmax(dim=1).cpu().numpy()
```

```
    return clusters


# Example usage
if __name__ == "__main__":
    # Generate sample irregular 16x16 array
    import numpy as np
    np.random.seed(42)

    # Base grid with random perturbations
    x = np.linspace(0, 15, 16)
    y = np.linspace(0, 15, 16)
    xx, yy = np.meshgrid(x, y)
    positions = np.stack([xx.flatten(), yy.flatten()], axis=1)
    positions += np.random.randn(256, 2) * 0.3  # Add noise

    positions = torch.tensor(positions, dtype=torch.float32)

    # Train and get clusters
    clusters = train_clustering(
        positions,
        k_neighbors=8,
        num_clusters=4,
        epochs=300
    )

    print(f"Cluster distribution: {np.bincount(clusters)}")
```

4.3 Hyperparameter Recommendations

Table 1: Recommended Hyperparameters for 16×16 Array

Parameter	**Value**	**Reasoning**
k (neighbors)	6–10	Balance local/global connectivity
Hidden dim	32–128	Larger for complex patterns
GAT heads	4–8	More heads = diverse attention
Layers	2–4	Deeper = larger receptive field
Learning rate	10^{-3}	Standard for Adam
Weight decay	5×10^{-4}	Regularization
Dropout	0.1–0.2	Prevent overfitting
Epochs	200–500	Until loss stabilizes
λ_1 (ortho)	0.5–2.0	Higher = more balanced clusters

5 Summary: The Complete Pipeline

Key Insight

The GNN Clustering Pipeline:

1. **Represent** your antenna array as a graph (nodes = elements, edges = proximity)
2. **Compute** node features (positions) and edge features (distance, coupling)
3. **Apply** multiple GNN layers (GAT recommended) to learn node embeddings
4. **Transform** embeddings to cluster probabilities via softmax
5. **Train** using MinCut + Orthogonality loss (no labels needed!)
6. **Extract** hard cluster assignments via arg max

This approach is:

- ✓ **Unsupervised** (no labeled data required)
- ✓ **Physics-aware** (through edge features)
- ✓ **Scalable** ($O(|\mathcal{E}|)$ complexity)
- ✓ **Flexible** (works for any irregular geometry)
- ✓ **Differentiable** (end-to-end trainable)

A Mathematical Notation Reference

Symbol	Meaning
N	Number of antenna elements (256)
K	Number of clusters
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Graph (vertices, edges)
$\mathbf{A} \in \mathbb{R}^{N \times N}$	Adjacency matrix
$\mathbf{D} \in \mathbb{R}^{N \times N}$	Degree matrix (diagonal)
$\mathcal{N}(i)$	Neighbors of node i
$\mathbf{x}_i \in \mathbb{R}^d$	Feature vector of node i
$\mathbf{h}_i^{(\ell)} \in \mathbb{R}^{d_\ell}$	Embedding of node i at layer ℓ
α_{ij}	Attention weight from j to i
$\mathbf{Z} \in \mathbb{R}^{N \times K}$	Soft cluster assignment matrix
z_{ik}	Probability: node i in cluster k
$c_i \in \{1, \ldots, K\}$	Hard cluster assignment for node i
$\mathbf{W}^{(\ell)}$	Learnable weight matrix at layer ℓ
$\|\cdot\|_F$	Frobenius norm
$\mathrm{Tr}(\cdot)$	Matrix trace

B Troubleshooting Common Issues

1. **All nodes assigned to one cluster:**

- Increase λ_1 (orthogonality weight)
- Add entropy regularization
- Check if learning rate is too high

2. **Very unbalanced clusters:**
   - Increase orthogonality loss weight
   - Try different k for graph construction
3. **Loss not decreasing:**
   - Reduce learning rate
   - Check for NaN in gradients
   - Normalize input positions
4. **Clusters don't respect spatial proximity:**
   - Increase k in k-NN
   - Add more GNN layers
   - Use edge features (distance)