Hierarchical Graph Neural Networks

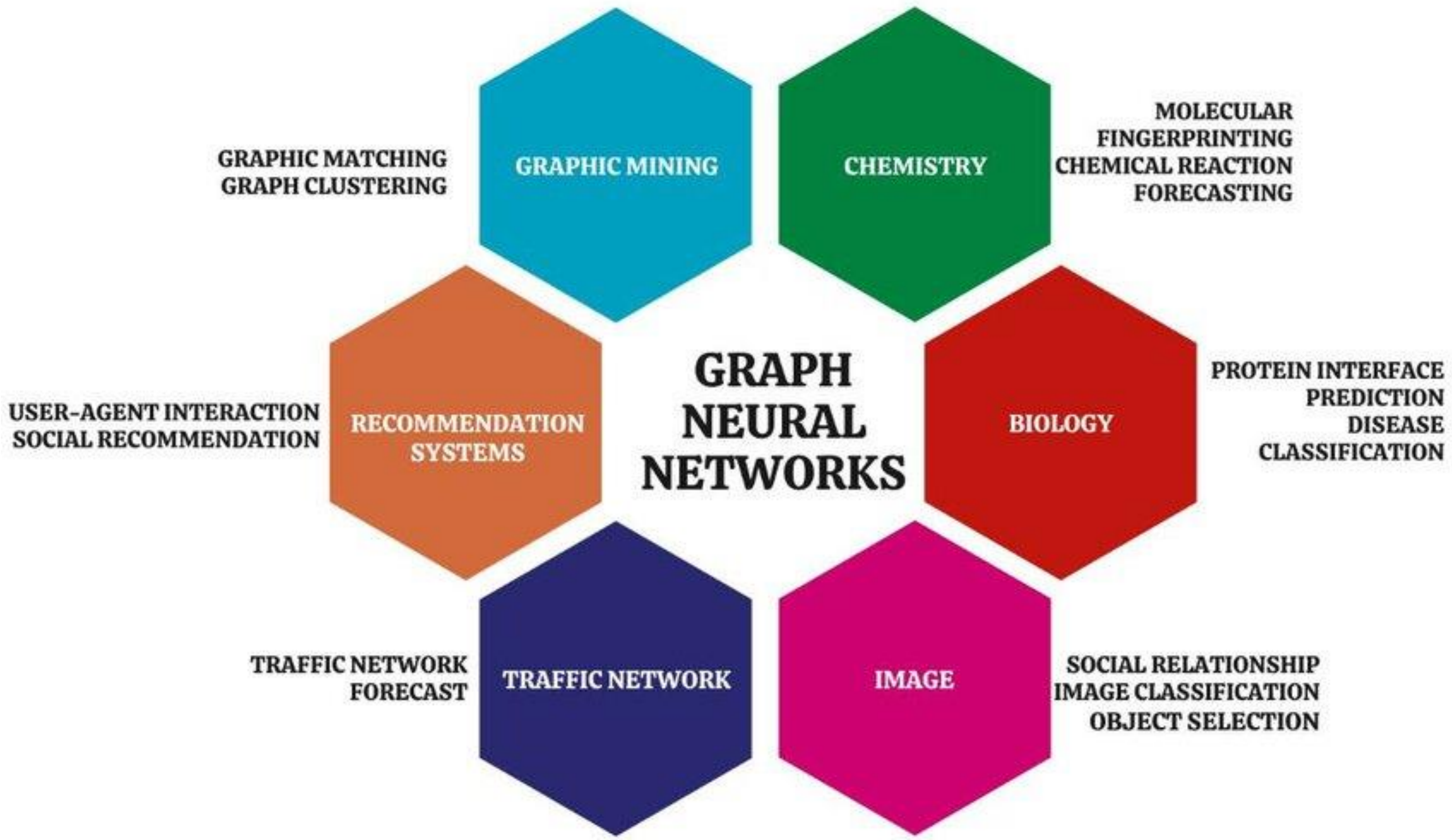
A seminar presentation for HO-GNN WS2024/25

Mohammad Shaique Solanki – 7062950

Content

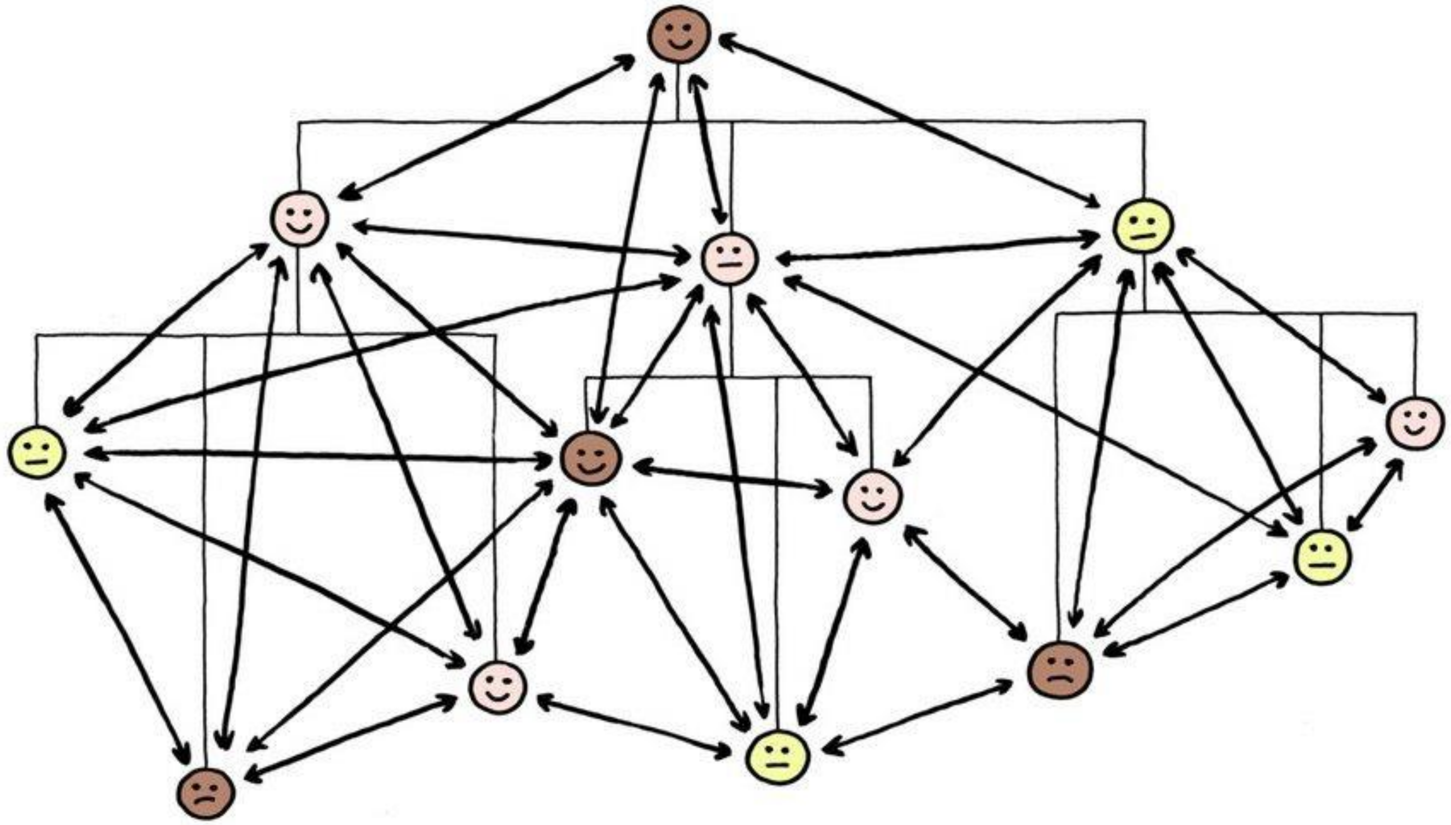
1. Introduction

Motivation == Graphs are everywhere



1. Introduction

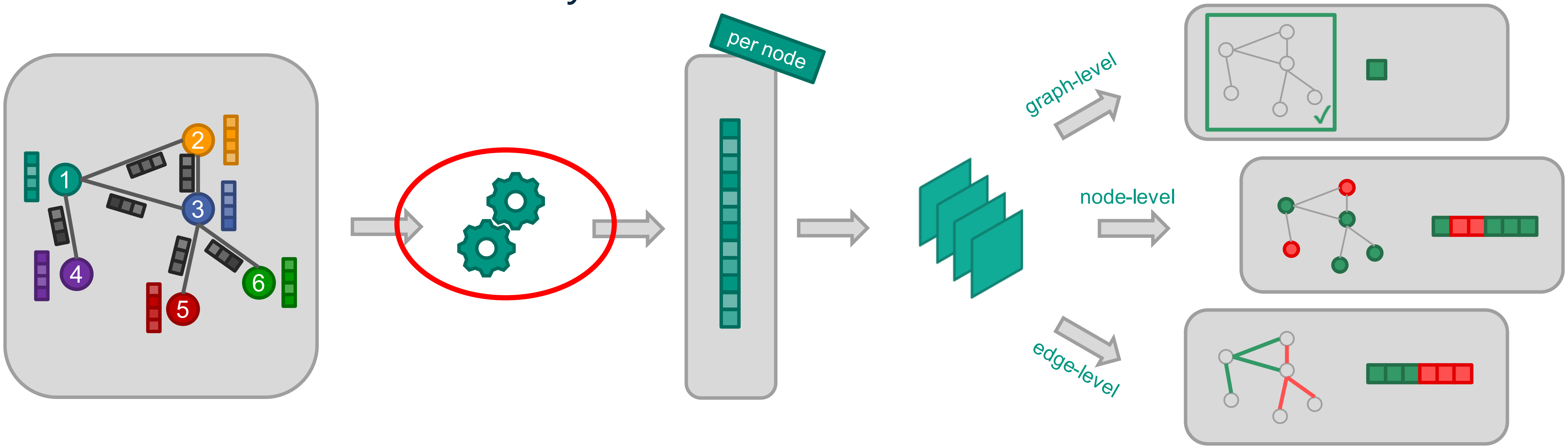
Objective == Learning the hierarchy in the graphs



[Scientists at Indian Institute of Science, Bangalore used Hierarchical GNNs for Speaker diarization problem.]

2. Background

What are GNNs? How do they work?



(X, A)
Input Graph

$(X, A) \rightarrow H$
Graph Encoder
(e.g. GNN)

$H \in \mathbb{R}^{NxM}$
Embedding
(aka. latent representation,
aka. feature vector)

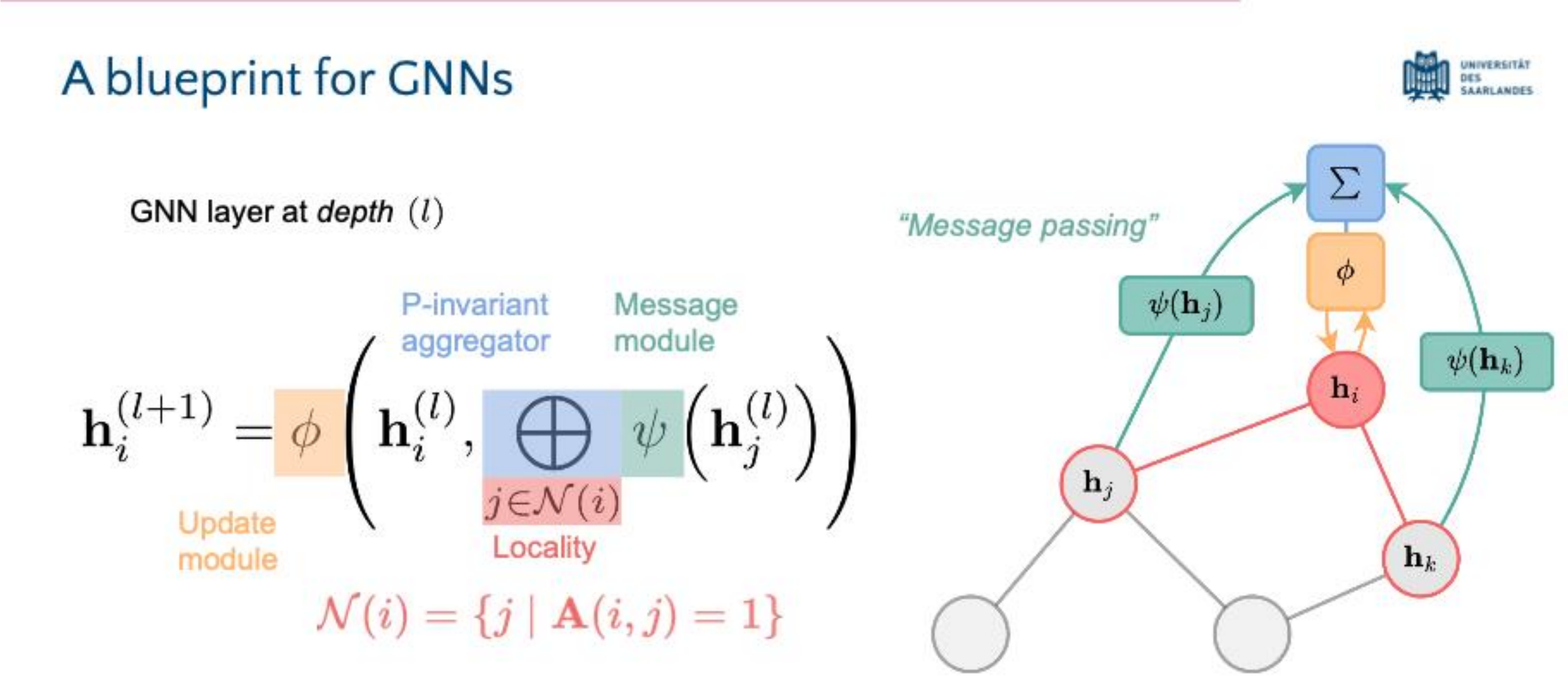
$H \rightarrow \hat{Y}$
Prediction Head
(e.g. MLP)

$\hat{Y}$
Output / Prediction

Feature Extraction

Prediction

2. Background



Uniform Processing Across Layers

Each layer uniformly transforms neighbour node features using ψ (e.g., via a weight matrix) before aggregation, applying the same operations irrespective of the layer depth.

Simple Aggregation Method

Uses a straightforward aggregation function ⊕, typically a mean or weighted sum, without adapting to node features or graph complexity, reinforcing the flat architecture design.

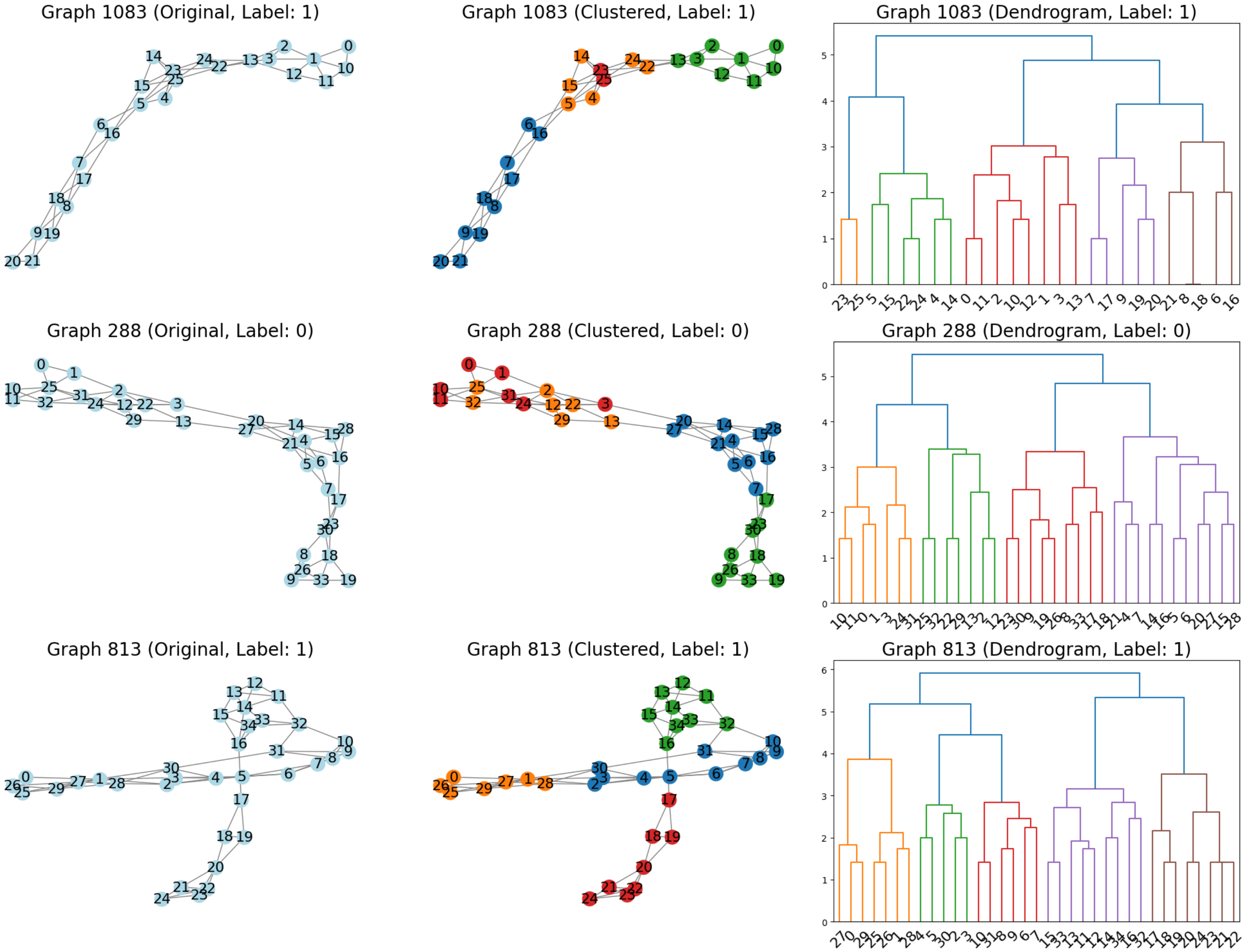
3. Observing Hierarchy in Graph Datasets

TUDataset Docs

Bioinformatics

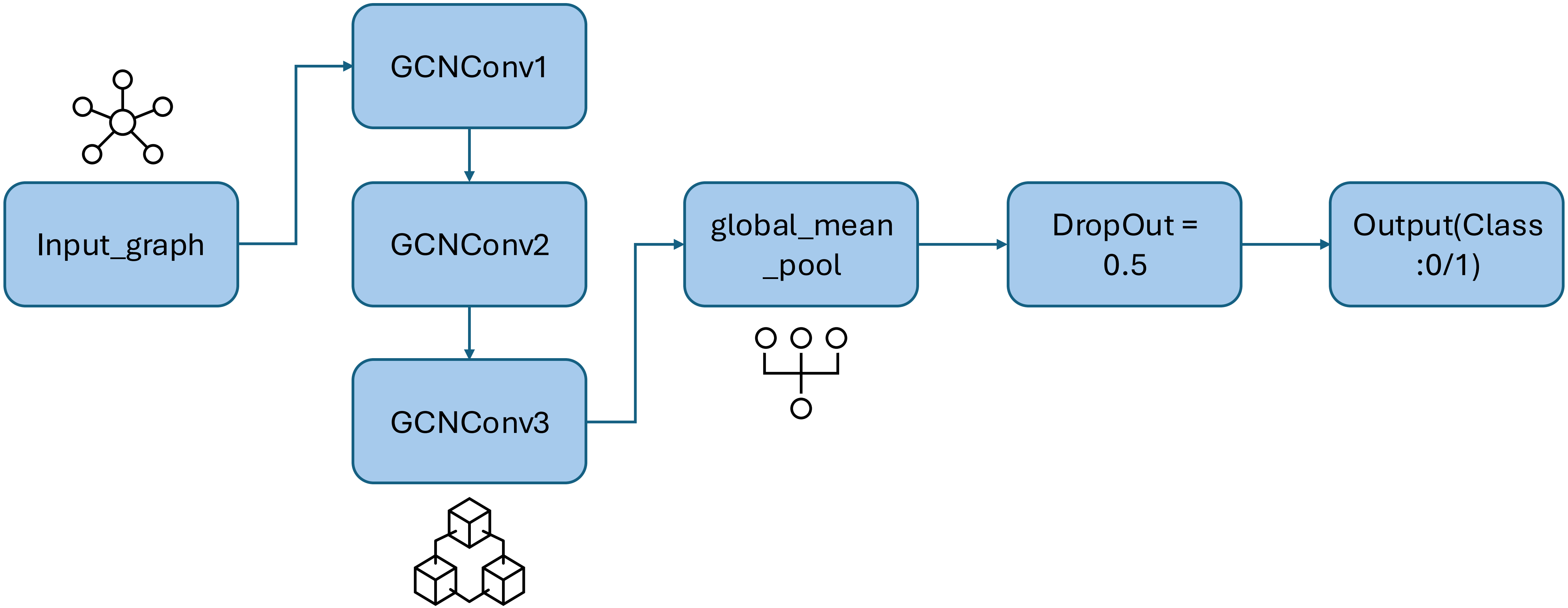
Name	Source	Statistics				Labels/Attributes					Download (ZIP)
		Graphs	*Classes*	*Avg. Nodes*	*Avg. Edges*	*Node Labels*	*Edge Labels*	*Node Attr.*	*Geometry*	*Edge Attr.*	
DD	[6,22]	1178	2	284.32	715.66	+	–	–	–	–	DD
ENZYMES	[4,5]	600	6	32.63	62.14	+	–	+ (18)	–	–	ENZYMES
KKI	[26]	83	2	26.96	48.42	+	–	–	–	–	KKI
OHSU	[26]	79	2	82.01	199.66	+	–	–	–	–	OHSU
Peking_1	[26]	85	2	39.31	77.35	+	–	–	–	–	Peking_1
PROTEINS	[4,6]	1113	2	39.06	72.82	+	–	+ (1)	–	–	PROTEINS
PROTEINS_full	[4,6]	1113	2	39.06	72.82	+	–	+ (29)	–	–	PROTEINS_full

3. Observing Hierarchy in Graph Datasets



3. Observing Hierarchy in Graph Datasets

Model Architecture



3. Observing Hierarchy in Graph Datasets

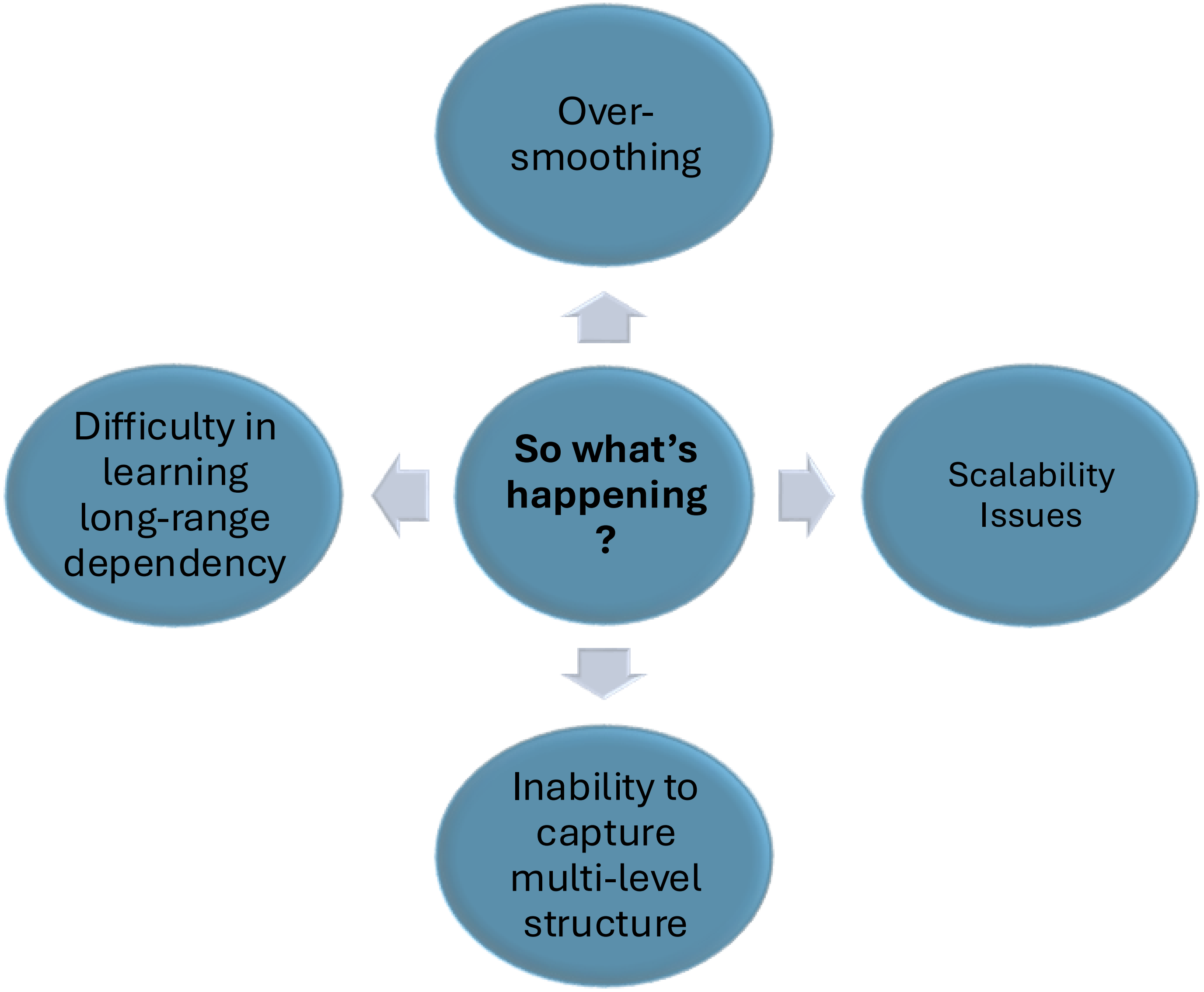
UNIVERSITÄT DES SAARLANDES

Overview of results

Final Test Accuracy: 0.7232, Test Loss: 0.5846

3. Observing Hierarchy in Graph Datasets

Analysis



3. Observing Hierarchy in Graph Datasets

Analysis – Diving Deep

Problem	Culprit

Over-Smoothing	`self.conv1 = GCNConv(dataset.num_node_features, hidden_channels)`
	`self.conv2 = GCNConv(hidden_channels, hidden_channels)`
	`self.conv3 = GCNConv(hidden_channels, hidden_channels)`

Scalability Issues	
Inability to Capture Multi-Scale Structures	
Difficulty in Learning long-range dependencies	

Stacking multiple GCN layers without preserving distinct node features of incorporate different layers of abstraction

3. Observing Hierarchy in Graph Datasets

Analysis – Diving Deep

Problem	Culprit

Over-Smoothing	`self.conv1 = GCNConv(dataset.num_node_features, hidden_channels)`
	`self.conv2 = GCNConv(hidden_channels, hidden_channels)`
	`self.conv3 = GCNConv(hidden_channels, hidden_channels)`

Scalability Issues	`train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)`
	`val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)`
	`test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)`

Inability to Capture Multi-Scale Structures	
Difficulty in Learning long-range dependencies	

Using a static batch size for processing large, complex graphs like proteins, where node and edge counts vary, can cause scalability issues and inefficiencies in memory and computational resources.

3. Observing Hierarchy in Graph Datasets

Analysis – Diving Deep

Problem	Culprit

Over-Smoothing	`self.conv1 = GCNConv(dataset.num_node_features, hidden_channels)`
	`self.conv2 = GCNConv(hidden_channels, hidden_channels)`
	`self.conv3 = GCNConv(hidden_channels, hidden_channels)`

Scalability Issues	`train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)`
	`val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)`
	`test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)`

Inability to Capture Multi-Scale Structures	`def forward(self, x, edge_index, batch):`
	`    x = F.relu(self.conv1(x, edge_index))`
	`    x = F.relu(self.conv2(x, edge_index))`
	`    x = F.relu(self.conv3(x, edge_index))`
	`    x = global_mean_pool(x, batch)`

Difficulty in Learning long-range dependencies	

The model's uniform convolutional layers and global mean pooling lack the ability to capture multi-scale features essential for hierarchical structures like proteins.

3. Observing Hierarchy in Graph Datasets

Analysis – Diving Deep

Problem	Culprit

Over-Smoothing	`self.conv1 = GCNConv(dataset.num_node_features, hidden_channels)`
	`self.conv2 = GCNConv(hidden_channels, hidden_channels)`
	`self.conv3 = GCNConv(hidden_channels, hidden_channels)`

Scalability Issues	`train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)`
	`val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)`
	`test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)`

Inability to Capture Multi-Scale Structures	`def forward(self, x, edge_index, batch):`
	`x = F.relu(self.conv1(x, edge_index))`
	`x = F.relu(self.conv2(x, edge_index))`
	`x = F.relu(self.conv3(x, edge_index))`
	`x = global_mean_pool(x, batch)`

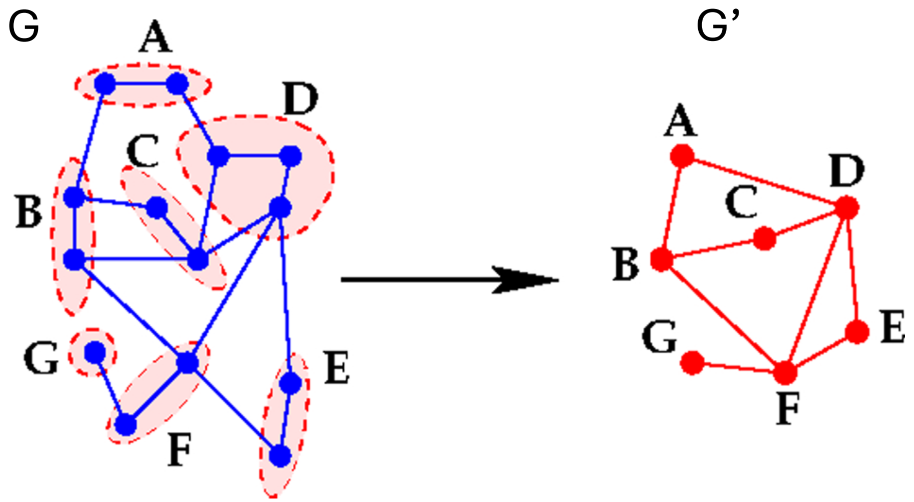
Difficulty in Learning long-range dependencies	`x = F.relu(self.conv1(x, edge_index))`
	`x = F.relu(self.conv2(x, edge_index))`
	`x = F.relu(self.conv3(x, edge_index))`

While graph convolutions theoretically capture long-range dependencies through layer stacking, in practice, the repetitive application of uniform convolutions across entire graphs fails to effectively address long-range interactions in large, complex structures.

4. Fundamentals of Hierarchical GNN

UNIVERSITÄT DES SAARLANDES

Graph Coarsening



$$H^{(l+1)} = C^{(l)}H^{(l)}W^{(l)}$$

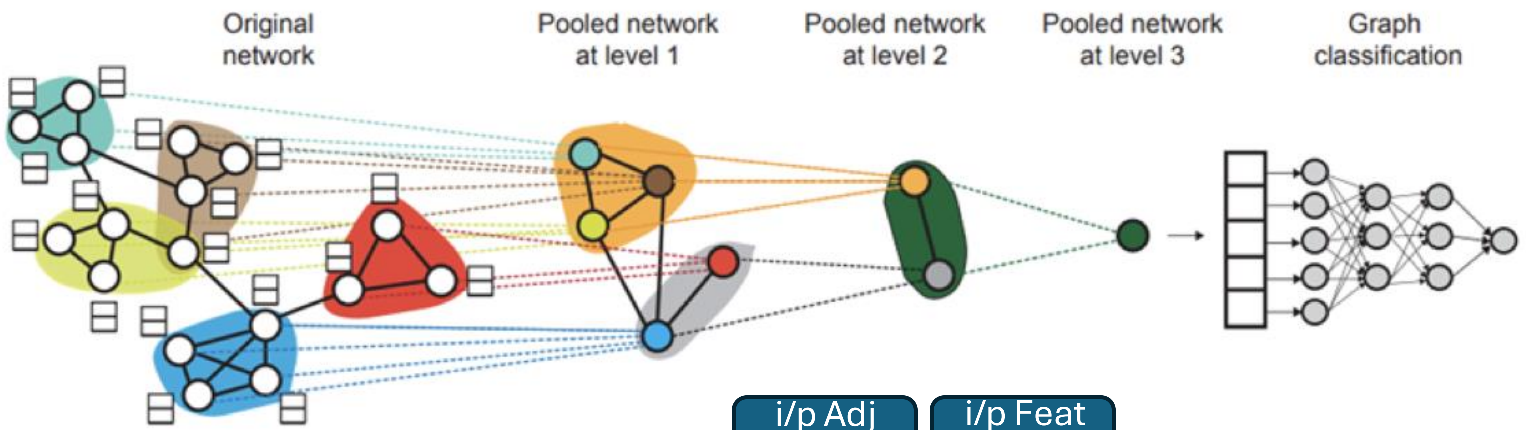
- **$H^{(l)}$** is the matrix of node features at layer (l)
- **$W^{(l)}$** is the weight matrix at layer (l), which transforms the node features.
- $C^{(l)}$ is the coarsening matrix which reduces the number of nodes by merging them

Capturing the Hierarchy

$$H^{(l+1)} = \sigma(Aggregate(C^{(l)}H^{(l)})W^{(l)})$$

- **σ** is a nonlinear activation function that introduces nonlinearity
- **AGGREGATE** could be a function like sum, mean, or more complex learned function, which combines features of the nodes which have been merged together.

5. Diffpool Mechanism



i/p Adj Mat

i/p Feat Mat

Node embeddings at layer l

$$\boxed{Z^{(l)} = \mathrm{GNN}_{l,\mathrm{embed}}(A^{(l)}, X^{(l)})}$$

Learned cluster assignment matrix at layer l

$$\boxed{S^{(l)} = \mathrm{softmax}\left(\mathrm{GNN}_{l,\mathrm{pool}}(A^{(l)}, X^{(l)})\right) \in \mathbb{R}^{n_l \times n_{l+1}}}$$

o/p Feat Mat

$$\boxed{X^{(l+1)} = {S^{(l)}}^T Z^{(l)} \in \mathbb{R}^{n_{l+1} \times d}}$$

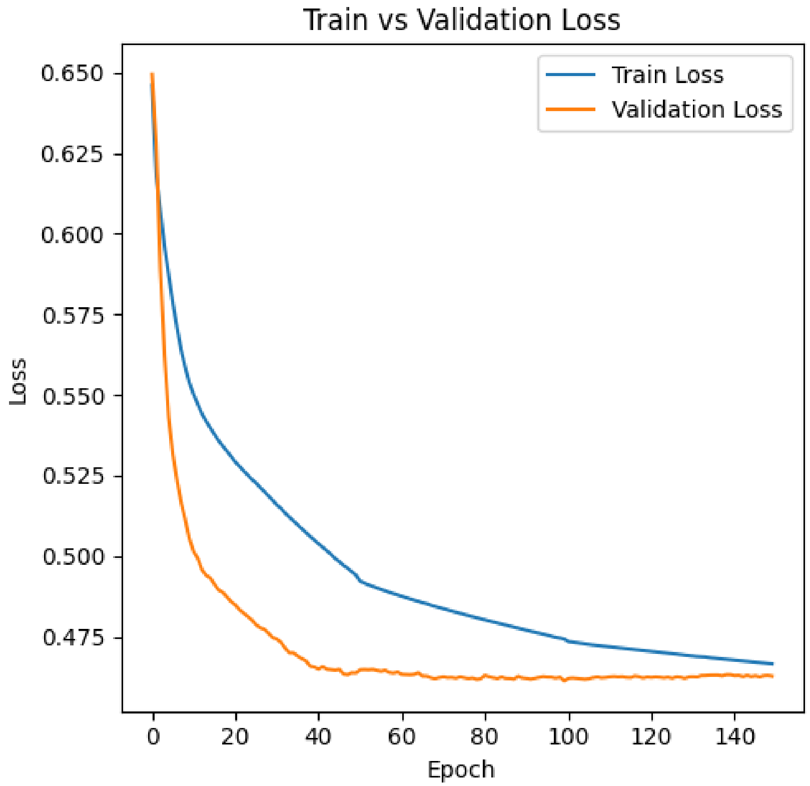
o/p Adj Mat

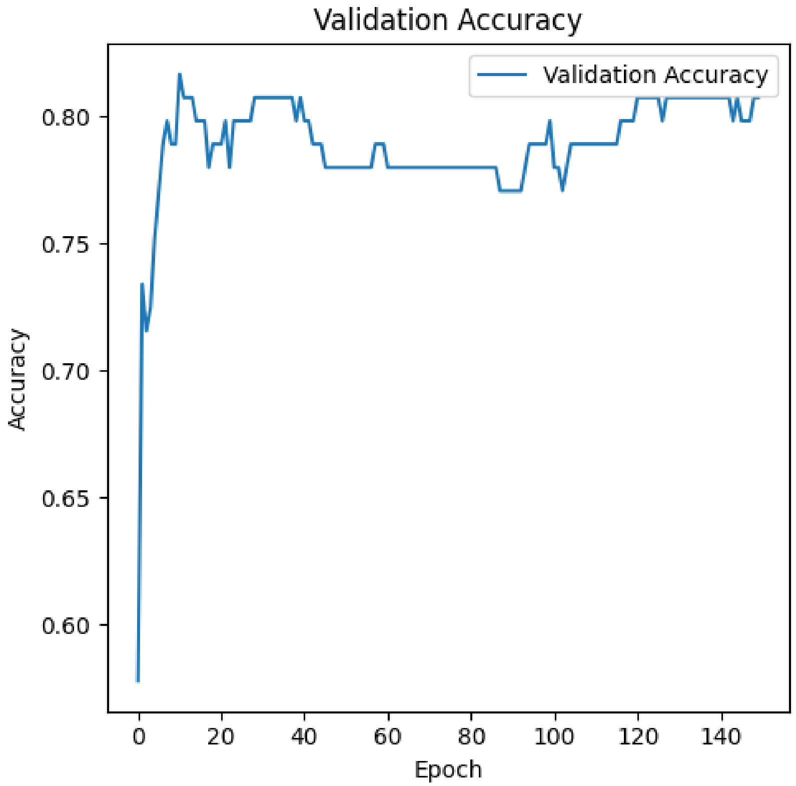
$$\boxed{A^{(l+1)} = {S^{(l)}}^T A^{(l)} S^{(l)} \in \mathbb{R}^{n_{l+1} \times n_{l+1}}}$$

5. Diffpool Mechanism - Implementation

```python
class GNN(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels, normalize=False, lin=True):
        super().__init__()
        self.conv1 = DenseSAGEConv(in_channels, out_channels, normalize)
        self.bn1 = torch.nn.BatchNorm1d(out_channels)
        self.lin = torch.nn.Linear(out_channels, out_channels) if lin else None

    def forward(self, x, adj, mask=None):
        x = self.conv1(x, adj, mask).relu()
        x = self.bn(1, x)
        if self.lin is not None:
            x = self.lin(x).relu()
        return x

    def bn(self, i, x):
        batch_size, num_nodes, num_channels = x.size()
        x = x.view(-1, num_channels)
        x = getattr(self, f'bn{i}')(x)
        x = x.view(batch_size, num_nodes, num_channels)
        return x


class Net(torch.nn.Module):
    def __init__(self, num_features, num_classes):
        super().__init__()
        self.gnn1_pool = GNN(num_features, 64, ceil(0.25 * 640))
        self.gnn1_embed = GNN(num_features, 64, 64, lin=False)

        self.lin1 = torch.nn.Linear(64, 64)
        self.lin2 = torch.nn.Linear(64, num_classes)

    def forward(self, x, adj, mask=None):
        s = self.gnn1_pool(x, adj, mask)
        x = self.gnn1_embed(x, adj, mask)
        x, adj, _, _ = dense_diff_pool(x, adj, s, mask)

        x = x.mean(dim=1)
        x = self.lin1(x).relu()
        x = self.lin2(x)
        return F.log_softmax(x, dim=-1)
```

5. Diffpool Mechanism - Implementation



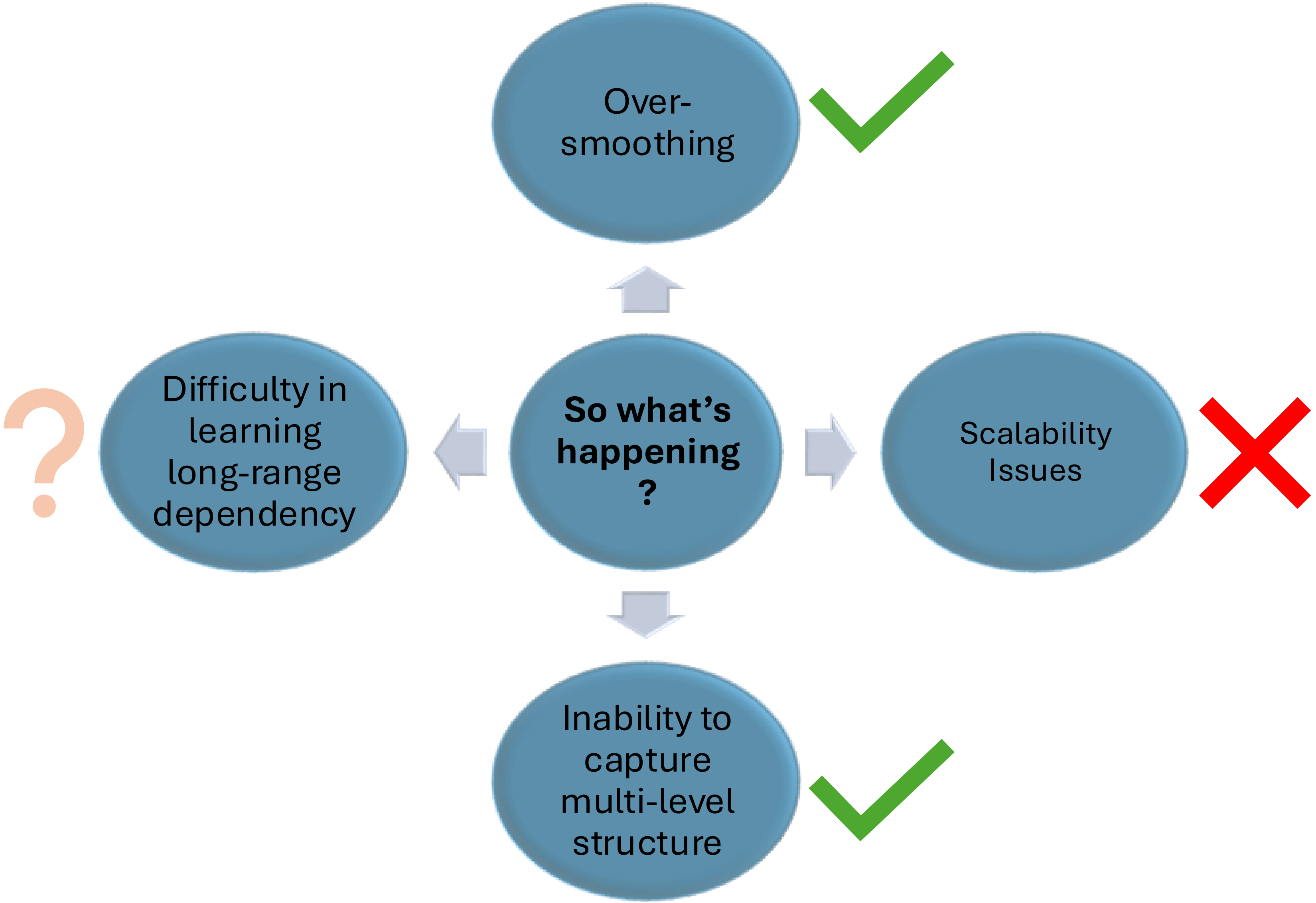


Final Test Accuracy with diffpool : 0.78, Test Loss: 0.48

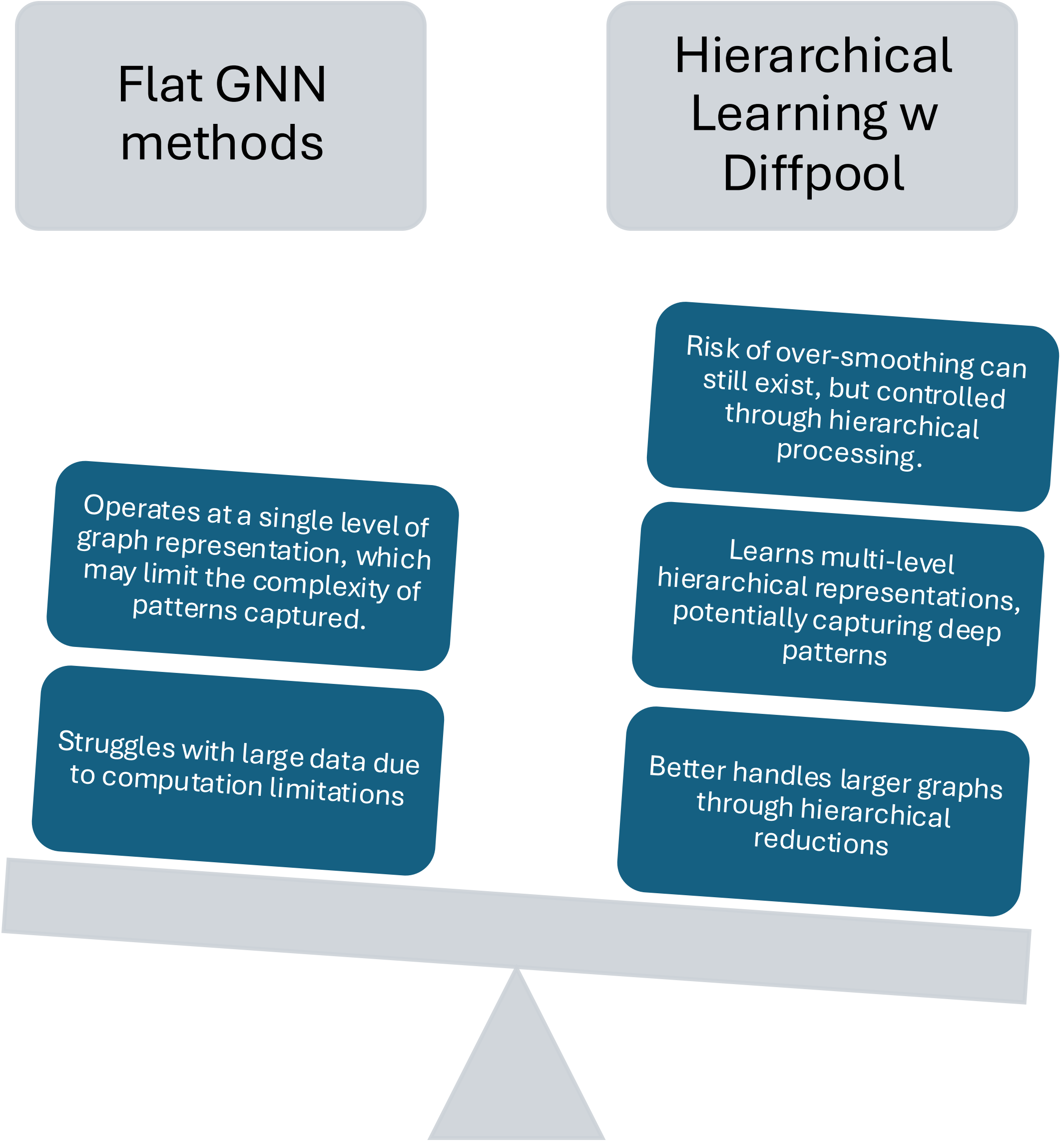
Final Test Accuracy w/o : 0.7232, Test Loss: 0.5846

5. Diffpool Mechanism

Analysis



6. Tying it all up



Questions

Thank you!!!