Introduction to GNNs

StructML – Tutorial 6

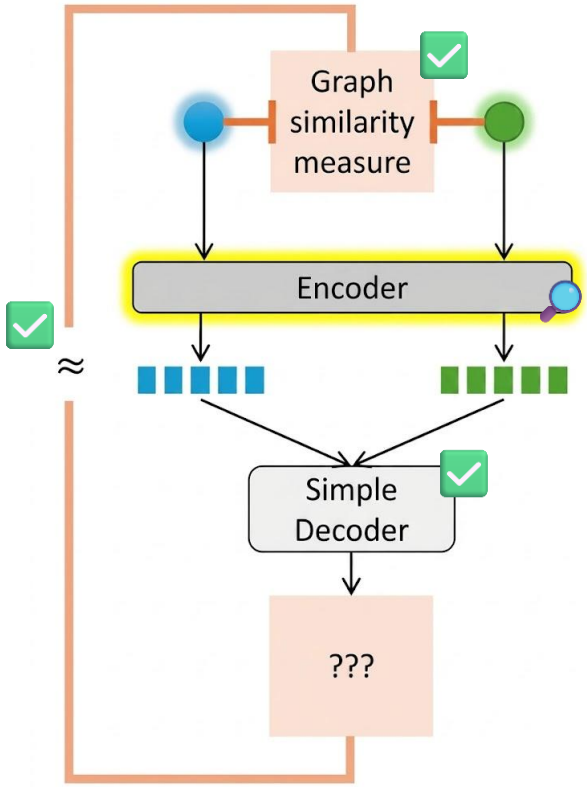
Outline

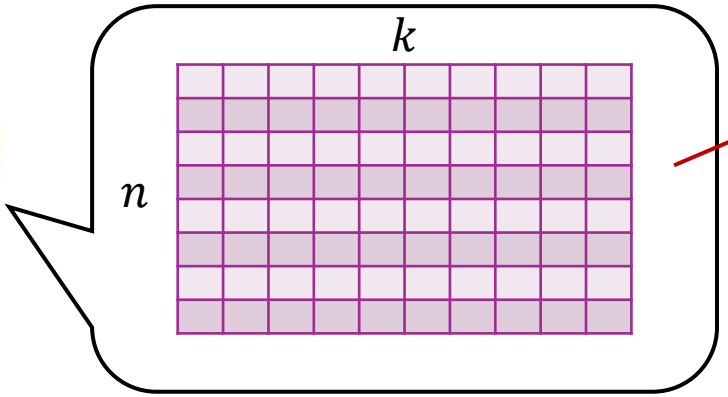
- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

Outline

- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

Previously – Embedding using Encoder Decoder





Shallow Encoder
Direct mapping from a node to its embedding

- ✗ Requires $O(n \cdot k)$ parameters
- ✗ Inherently transductive
- ✗ No capture of structural equivalence

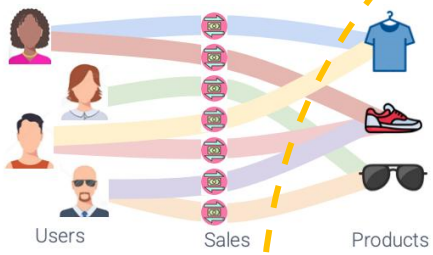
Let's solve these

But... how?

GNNs – Graph Neural Networks

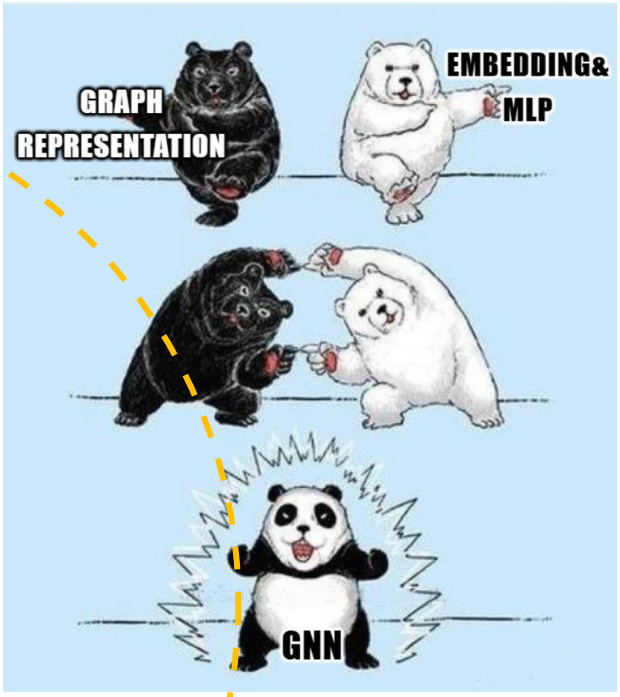
Input Database

Graph Representation



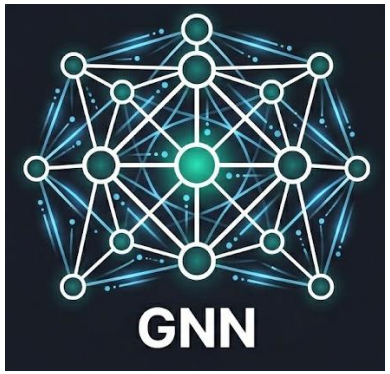
Learning Algorithm

Prediction



GNNs – Graph Neural Networks

Aim: *Incorporate learning of downstream task into graph representation*

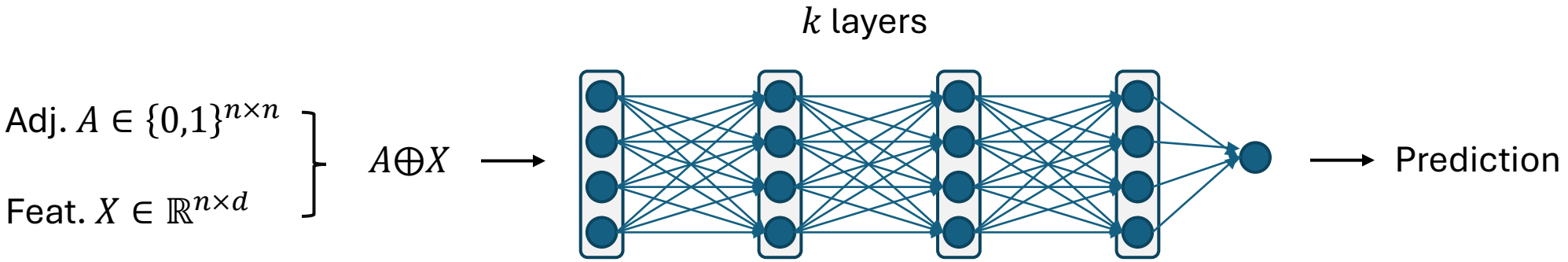
Input: Graph → 

→ Output: Prediction

But… how?

Naive Approach

Idea: Feed the raw feature matrix + adjacency matrix into an MLP

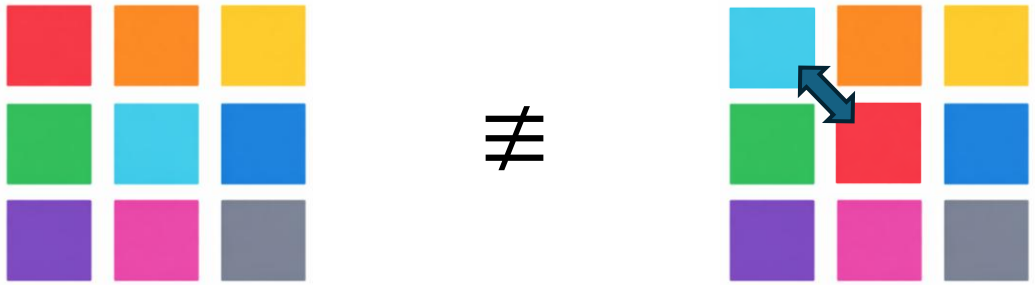


Limitations

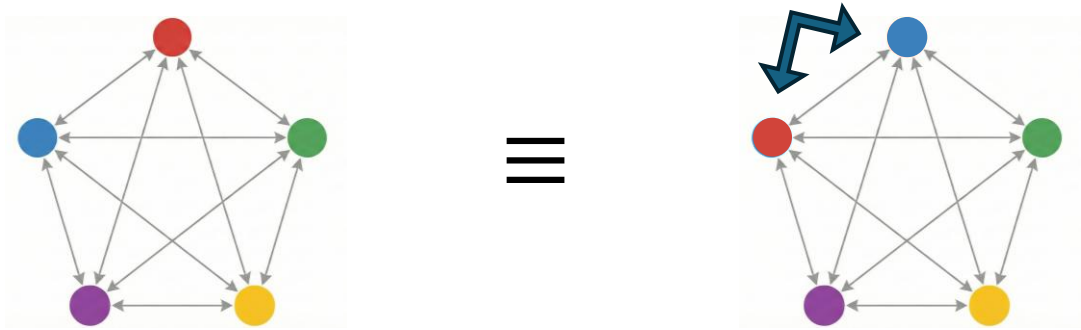
- Doesn't fit graphs of different sizes
- Large parameters count – depends on graph size
- Sensitive to the order of nodes

Data Structure in GNNs

- In text / images, every input has an order
  - Each element ⇒ unique position



- Not in graphs!



Requirements from GNNs - Equivariance

Idea: We require the GNN to be independent of the order.

What is an order?

Given a permutation $\pi: \{1, \dots, n\} \to \{1, \dots, n\}$, the permutation matrix:

$$\mathbf{P}_{i,j} := \begin{cases} 1, & \text{if } \pi(i) = j \\ 0, & otherwise \end{cases}$$

Permutes the matrix $\mathbf{X}$ by multiplying $\mathbf{P} \cdot \mathbf{X}$

Example:

- $n = 3$
- $\pi(1) = 2$
- $\pi(2) = 1$
- $\pi(3) = 3$

$$\mathbf{P} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \text{Row 1} \\ \text{Row 2} \\ \text{Row 3} \end{bmatrix} \quad \Rightarrow \quad \mathbf{P} \cdot \mathbf{X} = \begin{bmatrix} \text{Row 2} \\ \text{Row 1} \\ \text{Row 3} \end{bmatrix}$$

Requirements from GNNs - Equivariance

Notation: A GNN implements a learnable function $F(\mathbf{A}, \mathbf{X}; \theta)$ where:

- $\mathbf{A} \in \mathbb{R}^{n \times n}$ – the adjacency matrix
- $\mathbf{X} \in \mathbb{R}^{n \times d}$ – the nodes feature matrix
- θ – learned parameters
- Output: a new feature matrix $\mathbf{X}' \in \mathbb{R}^{n \times d'}$

Equivariance: The outcome of each node is preserved under permutations

$$F(\mathbf{P}\mathbf{A}\mathbf{P}^{\top}, \mathbf{P}\mathbf{X}; \theta) = \mathbf{P}F(\mathbf{A}, \mathbf{X}; \theta)$$

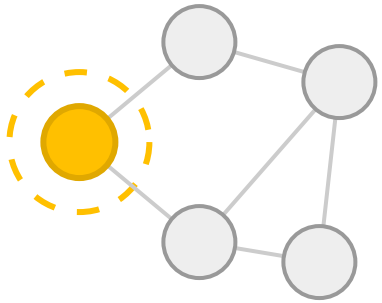
Permuting the adjacency matrix

- Fits node-level tasks

Task Types

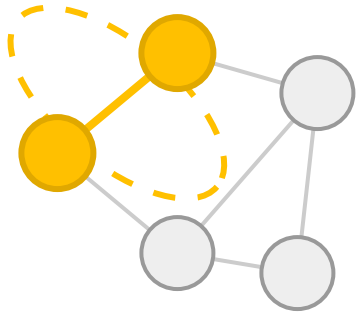
We defined $F(\mathbf{A}, \mathbf{X}; \theta)$ according to a node-level task, but of course we can predict tasks in different levels:

Node-level



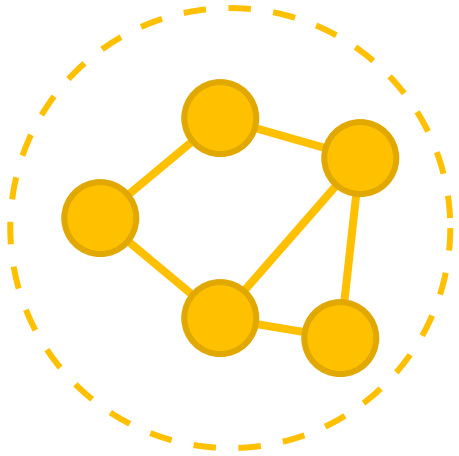
- Document Classification
- Fraud Detection

Link-level



- Friend/Item Recommendation
- Drug-Drug Interaction

Graph-level



- Molecular Property Prediction
- Program Vulnerability

⇒ Here the outcome should be a single vector

$$F(\mathbf{A}, \mathbf{X}; \theta) \in \mathbb{R}^{d'}$$

And similarly to Equivariance we define

Permutation Invariance:

$$F(\mathbf{P}\mathbf{A}\mathbf{P}^{\top}, \mathbf{P}\mathbf{X}; \theta) = F(\mathbf{A}, \mathbf{X}; \theta)$$

⇒ The result (graph prediction) doesn't depend on the order at all

Permutation Equivariance: $F(\mathbf{P}\mathbf{A}\mathbf{P}^{\top}, \mathbf{P}\mathbf{X}; \theta) = \mathbf{P}F(\mathbf{A}, \mathbf{X}; \theta)$

Outline

- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

First Attempt – Spectral CNN

Spectral CNN = Laplacian eigenmaps + Neural Network

- Inspired by Convolutional Neural Networks
- Given:
  - $A \in \mathbb{R}^{n \times n}$ – adjacency
  - $X \in \mathbb{R}^{n \times d}$ – feature matrix
  - $W \in \mathbb{R}^{n \times n}$ – diagonal learnable weights

$$A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad W = \begin{bmatrix} w_1 & 0 & 0 \\ 0 & w_2 & 0 \\ 0 & 0 & w_3 \end{bmatrix}$$

- Calculate:
  - Laplacian: $L = D - A$ where D is the degree matrix
  - Decompose: $L = U \Lambda U^T$ (symmetric)
    - Λ - eigenvalues of L
    - U – eigenvectors of L

$$\Rightarrow L = \begin{bmatrix} 1 & -1 & 0 \\ -1 & 2 & -1 \\ 0 & -1 & 1 \end{bmatrix}$$

$$X_{new} = \sigma(U \cdot W \cdot U^T \cdot X)$$

Limitations:
- Computationally expensive - $O(n^3)$ calculating eigenvalues
  → Due to global perspective
- Needs to be done from scratch for every different graph
  → How is that called? → *Transductive*

→ Can apply multiple W's as in multiple channels in CNNs
→ Can be extended by applying in layers
→ It's equivariant
→ Fed into a NN trained on the prediction task

Reference: Spectral Networks and Locally Connected Networks on Graphs

Summary So Far

Aim: *Incorporate learning process into graph representation*

- Naive Approach: $X \oplus A \Rightarrow$MLP
  - ✗ Rigid graph size
  - ✗ Rigid parameters count
  - ✗ Not equivariant
- Spectral CNN:
  - ✗ Rigid graph size
  - ✗ Rigid parameters count
  - ✓ Equivariant
  - ✗ High cost due to global perspective
  - ✗ Transductive

💡 Let's design a robust deep learning framework for graphs that will be permutation **invariant** & **equivariant**

💡 And also more **local** for hopefully better performance

💡 And optimally **inductive** and not transductive

Outline

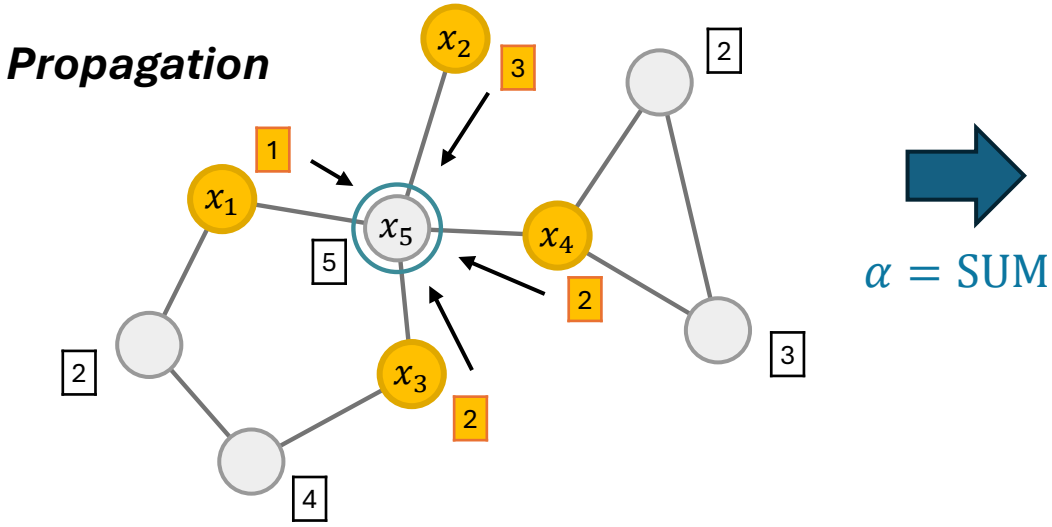
- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

Message Passing

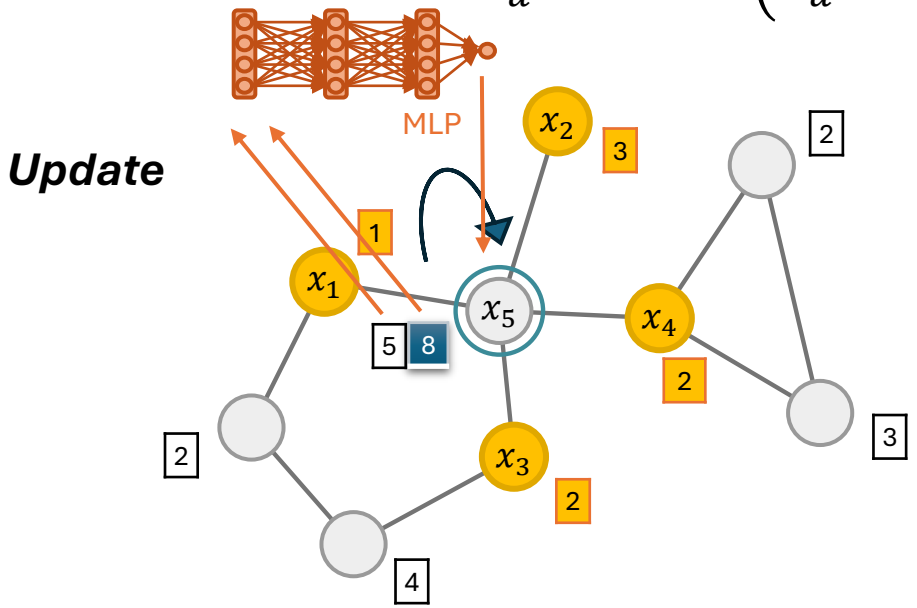
- $N(u)$ - neighbors of u
- $\mathrm{h}_u^{(k)}$ - embedding at "step" k
- $M^{(k)}$ - message function
- $\alpha^{(k)}$ - aggregation function
- $U^{(k)}$ - update function

multiset

$$\mathrm{m}_u^{(k)} := \alpha^{(k)}\left(\left\{\!\left\{ M^{(k)}\left(\mathrm{h}_u^{(k-1)}, \mathrm{h}_v^{(k-1)}\right) \mid v \in N(u) \right\}\!\right\}\right)$$



$$\mathrm{h}_u^{(k)} := U^{(k)}\left(\mathrm{h}_u^{(k-1)}, m_u^{(k)}\right)$$



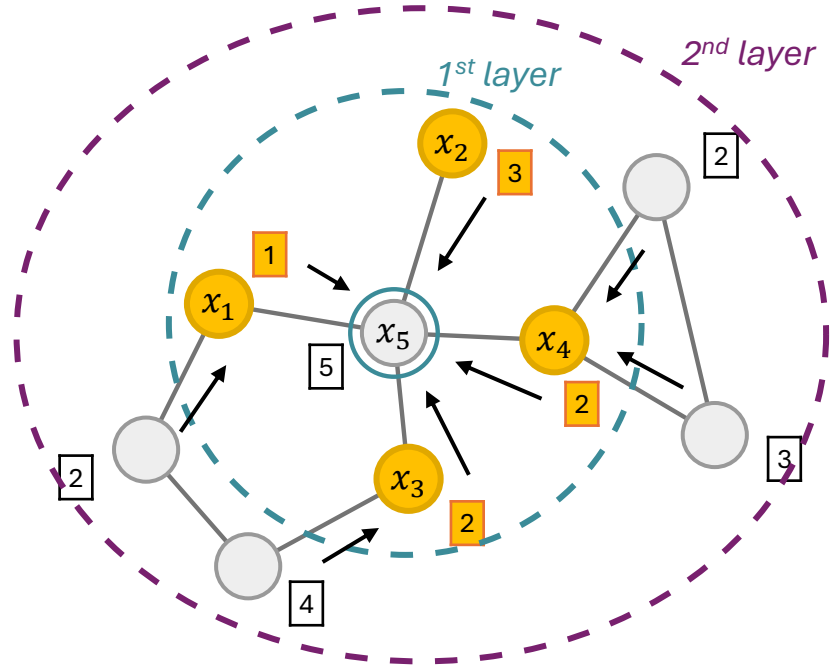
Message Passing Neural Network

MPNN = Message Passing Neural Network

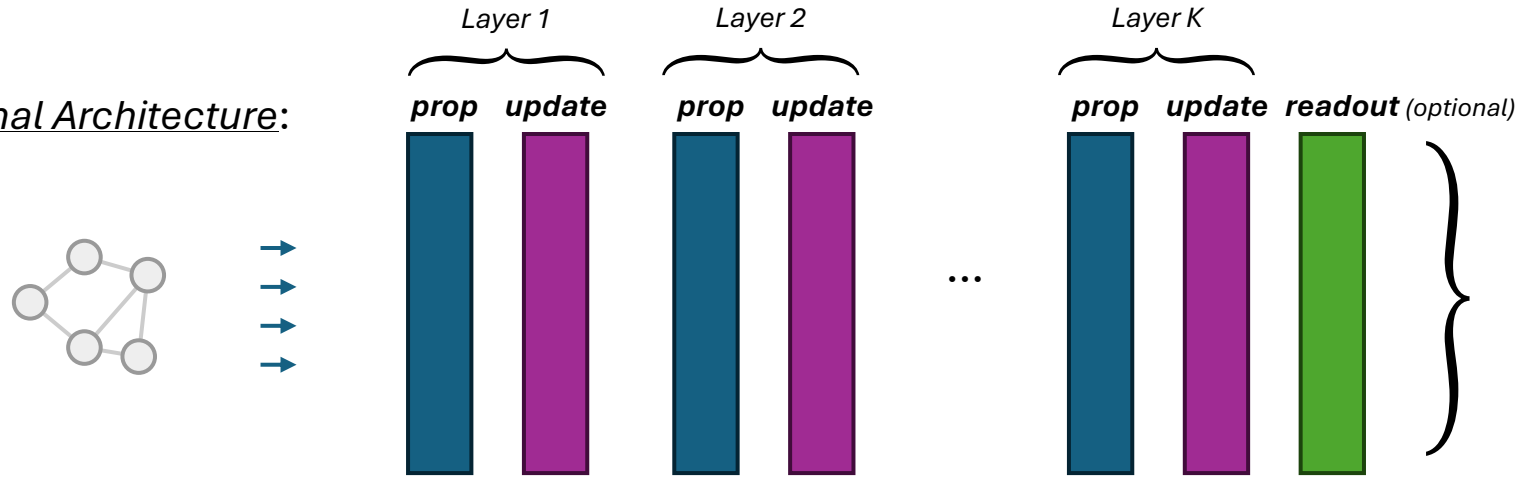
- Apply message passing K times, resulting with $\mathrm{h}_u^{(K)}$
- Each iteration – message passing layer
- For graph-level tasks – add a **readout function** R:

$$h_G := R(\{\{h_u^{(K)} | u \in V\}\})$$

- To make it overall permutation equivariant & invariant we restrict:
  - Aggregation (α), readout (R) - **invariant**
- The output is still an embedding. To get a final prediction ⇒ **prediction head**



Final Architecture:



... and so on ...

Message Passing in Code

Message Passing Layer

PyG

Basic MPNN unit

```python
from torch_geometric.nn import MessagePassing
from torch.nn import Linear

class SimpleMeanLayer(MessagePassing):
    def __init__(self, in_channels, out_channels):
        super().__init__(aggr='mean')
        self.lin = Linear(in_channels * 2, out_channels)

    def message(self, x_j):.
        return x_j

    def update(self, aggr_out, x):
        combined = torch.cat([x, aggr_out], dim=-1)
        return self.lin(combined)

    def forward(self, x, edge_index):
        return self.propagate(edge_index, x=x)
```

Input & output embedding dim

Aggregation function

Update NN

Message function

Update function Called by propagate

Executes propagation & update automatically

❓ *Where are the trainable parameters?*

Message Passing Model (MPNN)

A PyTorch module

Label dim

```python
import torch.nn.functional as F

class SimpleMPNNModel(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, num_classes):
        super().__init__()
        self.mpnn_layer = SimpleMeanLayer(in_channels, hidden_channels)
        self.mpnn_layer2 = SimpleMeanLayer(hidden_channels, hidden_channels)
        self.classifier = Linear(hidden_channels, num_classes)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        h = self.mpnn_layer(x, edge_index)
        h = F.relu(h)
        h = self.mpnn_layer2(h, edge_index)
        out = self.classifier(h)
        return out
```

Prediction head

2 message passing layers + relus in between + prediction head

❓ *Why is ReLU not part of the layer?*

→ *Limits final embedding to be non-negative*

Message Passing in Code

Executing the Model

```
from torch_geometric.data import Data

node_features = torch.tensor([
    [1.0, 0.0, 0.5],  # Node 0
    [0.2, 1.1, 0.0],  # Node 1
    [0.0, 0.0, 1.5],  # Node 2
    [0.8, 0.7, 0.2]   # Node 3
], dtype=torch.float)

edge_list = torch.tensor([
    [0, 1, 1, 2],
    [1, 0, 2, 3]
], dtype=torch.long)

toy_graph = Data(x=node_features, edge_index=edge_list)
model = SimpleMPNNModel(in_channels=3, hidden_channels=8, num_classes=2)
logits = model(toy_graph)

print("Output Shape (num_nodes, num_classes):", logits.shape)
print("Logits:\n", logits)
```

PyG data object

$|V| \times d$ node feature tensor

$|E| \times 2$ edges tensor

Instantiate the model

Execute the model

```
Output Shape (num_nodes, num_classes): torch.Size([4, 2])
Logits:
 tensor([[-0.5742, -0.6403],
        [-0.2172, -0.4693],
        [-0.4978, -0.8056],
        [-0.2545, -0.4351]], grad_fn=<AddmmBackward0>)
```

💡 This was a nice “Hello World” GNN
Now we will go through known **GNN architectures**.
We will answer questions regarding:

- What aggregation function to use?
- What update function to use?
- How well do they perform?
- What are their weaknesses?

Remember: No architecture is perfect

Outline

- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

GCN – Graph Convolutional Network

Inspired by Spectral CNN

- M – **identity** func. (of neighbor): $M^{(k)}\left(h_u^{(k-1)}, h_v^{(k-1)}\right) := h_v^{(k-1)}$
- α – **normalized average**
- U – 1-layer MLP:
  - $U^{(k)}\left(h_u^{(k-1)}, m_u^{(k)}\right) := \sigma(m_u^{(k)} W^{(k)})$
  - $W^{(k)} \in \mathbb{R}^{d_{k-1} \times d_k}$ – learnable parameters
  - σ – non-linearity

$$h_u^{(k)} := U^{(k)}\left(h_u^{(k-1)}, m_u^{(k)}\right)$$

$$m_u^{(k)} := \alpha^{(k)}\left(\left\{\left\{ M^{(k)}\left(h_u^{(k-1)}, h_v^{(k-1)}\right) \mid v \in N(u) \right\}\right\}\right)$$

$$h_u^{(k)} = \sigma\left(\sum_{v \in \widehat{N}(u)} \frac{1}{\sqrt{\widehat{d_u} \cdot \widehat{d_v}}} h_v^{(k-1)} W^{(k)}\right)$$

Includes self-node in agg ←

- $\widehat{N}(u) = N(u) \cup \{u\}$
- $\widehat{d_u} = \deg(u) + 1$
- $\hat{A} = A + I$
- $\widehat{D} = D + I$

Matrix Notation:

$$H^{(k)} = \sigma\left(\widehat{D}^{-\frac{1}{2}} \hat{A} \widehat{D}^{-\frac{1}{2}} H^{(k-1)} W^{(k)}\right)$$

- A – adjacency
- $H^{(k)} = \begin{bmatrix} h_1^{(k)} \\ h_2^{(k)} \\ \vdots \\ h_N^{(k)} \end{bmatrix}$

Reference: Semi-Supervised Classification with Graph Convolutional Networks

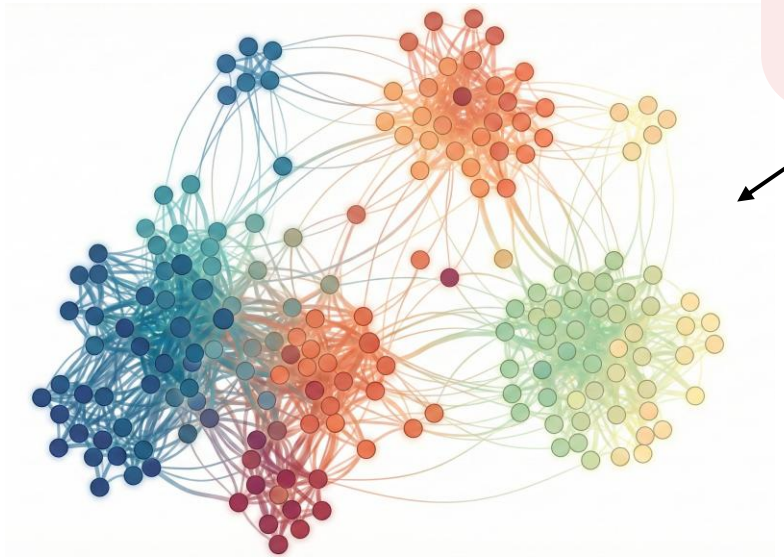
Evaluating GCN

✗ **Transductive**

Adding a new node requires recalculation of its K-hop neighborhood

✗ **Assumes homophilic graphs**

Neighboring nodes are likely to get similar representation (due to average). Doesn't hold for all graphs and tasks – i.e wife/husband



✗ **Homogeneous**

Ignores node and edge types

✓ No need to calculate eigenvalues

✓ Parameter count independent of graph size

✓ Reduced computational complexity

Outline

- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

RGCN – Relational GCN

💡 *Idea*: Extend GCN to treat the graph as heterogeneous

$$h_u^{(k)} := \sigma\left(W_0^{(k)} h_u^{(k-1)} + \sum_{\tau \in L_E} \sum_{v \in N_\tau(u)} \frac{1}{|N_\tau(v)|} W_\tau^{(k)} h_v^{(k-1)}\right)$$

- L_E - Edge labels
- $N_\tau(v)$ - Neighbors of v of edge label $\tau \in L_E$
- $W_\tau^{(k)}$ - Learnable weight matrix per node label $\tau \in L_E$ and layer k

Outline

- GNN Motivation & Setting
- Spectral CNN
- Message Passing
  - GCN
  - RGCN
  - Experimenting GCN vs RGCN

GCN vs RGCN

Experiment Idea: Which is better? GCN or RGCN?

Dataset: AIFB (Angewandte Informatik und Formale Beschreibungsverfahren)

- A real-world knowledge graph representing the internal structure of an academic research institute
- Nodes: ≈ 8,200 entities (Researchers, Publications, Projects, Topics)
- Edges: ≈29K links over 90 relation types

  (*author_of*, *works_on_project*, *is_affiliated_with*,...)
- No node features

Task: Predict the specific research group affiliation of subset of 176 researchers

GCN vs RGCN

Data Retrieval

```
from torch_geometric.datasets import Entities
dataset = Entities(root='./data/Entities', name='AIFB')
data = dataset[0]
```

PyG is equipped with datasets

Retrieve AIFB dataset

Only 1 graph in the dataset

Underlying Data object

GCN impl.

```
from torch_geometric.nn import GCNConv

class GCNNet(torch.nn.Module):
    def __init__(self, num_nodes, hidden_channels, num_classes):
        super().__init__()
        self.node_emb = torch.nn.Embedding(num_nodes, hidden_channels)
        self.conv1 = GCNConv(hidden_channels, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, num_classes)

    def forward(self, edge_index, edge_type=None):
        x = self.node_emb.weight
        x = F.relu(self.conv1(x, edge_index))
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

GCN Layer implemented in PyG

GCNConv requires node features

2 GCN layers + relu + softmax

RGCN impl.

```
from torch_geometric.nn import GCNConv, FastRGCNConv

class RGCNNet(torch.nn.Module):
    def __init__(self, num_nodes, hidden_channels, num_classes,
                 num_relations):
        super().__init__()
        self.conv1 = FastRGCNConv(num_nodes, hidden_channels,
                                  num_relations, num_bases=30)
        self.conv2 = FastRGCNConv(hidden_channels, num_classes,
                                  num_relations, num_bases=30)

    def forward(self, edge_index, edge_type):
        x = F.relu(self.conv1(None, edge_index, edge_type))
        x = self.conv2(x, edge_index, edge_type)
        return F.log_softmax(x, dim=1)
```

Algebraic Trick for parameters reduction

RGCN treats missing feature matrix automatically

2 RGCN layers + relu + softmax

Back to GCN vs RGCN

Training & Eval Functions

```
def train_and_eval(model, data, epochs=50):
    optimizer = torch.optim.Adam(model.parameters(), lr=0.01,
                                 weight_decay=0.0005)

    for epoch in range(1, epochs + 1):
        model.train()
        optimizer.zero_grad()
        out = model(data.edge_index, data.edge_type)
        loss = F.nll_loss(out[data.train_idx], data.train_y)
        loss.backward()
        optimizer.step()

    model.eval()
    with torch.no_grad():
        pred = model(data.edge_index, data.edge_type).argmax(dim=-1)
        train_acc = (pred[data.train_idx] == data.train_y).float().mean().item()
        test_acc = (pred[data.test_idx] == data.test_y).float().mean().item()

    return train_acc, test_acc
```

Models need only edges and their types

Will be ignored in GCN

Train

Eval

Execute Training & Eval

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
data = data.to(device)

print("Training Standard GCN...")
gcn = GCNNet(data.num_nodes, 16, dataset.num_classes).to(device)
gcn_train, gcn_test = train_and_eval(gcn, data)

print("Training Relational GCN (RGCN)...")
rgcn = RGCNNet(data.num_nodes, 16, dataset.num_classes,
dataset.num_relations).to(device)
rgcn_train, rgcn_test = train_and_eval(rgcn, data)

print("\n--- Results ---")
print(f"GCN Test Accuracy:  {gcn_test:.4f}")
print(f"RGCN Test Accuracy: {rgcn_test:.4f}")
```

Param. matrix dim=16

Any guesses?

GCN vs RGCN

- So RGCN is better than GCN
  - Maybe only in this setting?
  - Are there other cases in which GCN can be better?

Training Standard GCN...
Training Relational GCN (RGCN)...

--- Results ---
GCN Test Accuracy: 0.6944
RGCN Test Accuracy: 0.9444

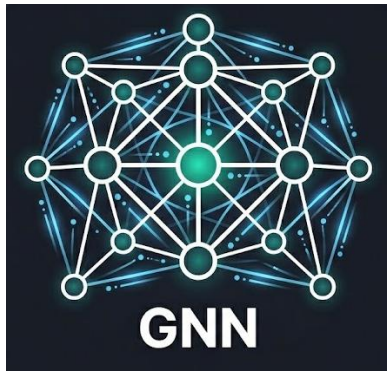
💡 *Conclusion*: Each task might benefit from different architectural properties
Choosing the **right architecture** is a crucial step in designing the solution for a specific learning task

→ In the next tutorials we will discuss more GNN architectures and their limitations

Big Picture

Input Database

Graph Representation



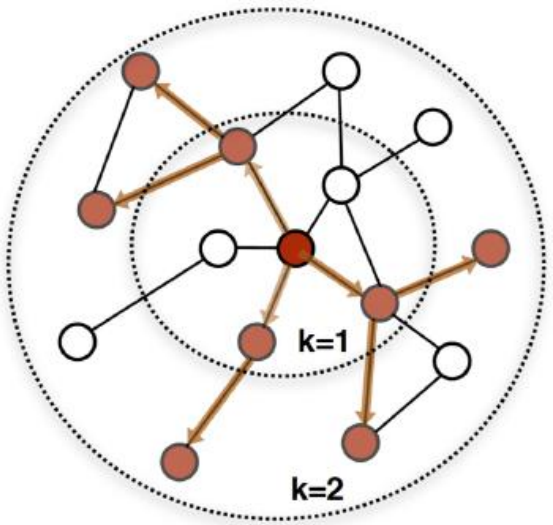
Prediction

GraphSAGE

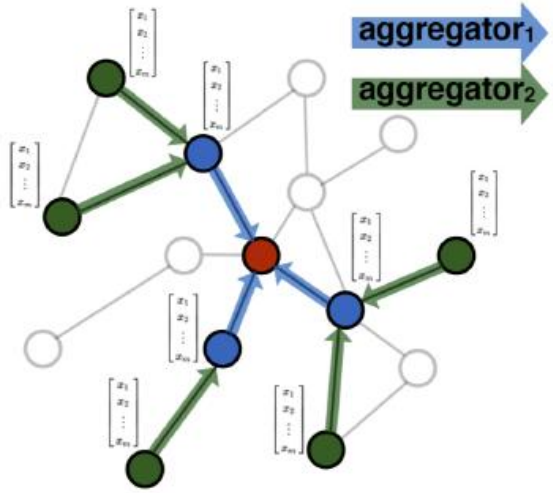
Sample & **AG**gregat**E**

- Instead of a strict aggregation scheme – why not learn it?
- Mechanism: Sample neighbors and apply custom learnable aggregations
- 3 options for aggregations:
  - Elementwise average
  - LSTM
  - Elementwise max +

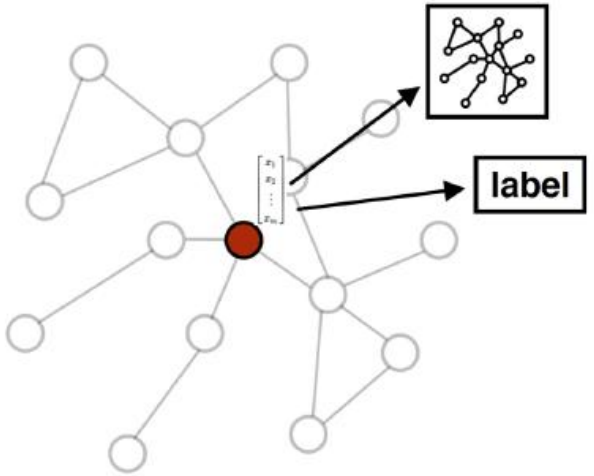
GraphSAGE



1. Sample neighborhood



2. Aggregate feature information from neighbors



3. Predict graph context and label using aggregated information

GraphSAGE

- Mean Aggregation
  - Element-wise mean com
- LSTM
  - Based on RNN
  - Not invariant – therefore a
- Max-pooling Aggregation
  - Element-wise max + MLP

$$h_{\mathcal{N}(v)}^{(k)} \leftarrow \sigma\left(W \cdot \text{MEAN}\left(\{h_v^{(k-1)}\} \cup \{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\}\right)\right)$$

$$h_{\mathcal{N}(v)}^{(k)} \leftarrow \text{LSTM}\left(\text{shuffle}\left(\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\}\right)\right)$$

$$\text{AGGREGATE}_k^{\text{pool}} = \max(\{\sigma\left(\mathbf{W}_{\text{pool}}\mathbf{h}_{u_i}^k + \mathbf{b}\right), \forall u_i \in \mathcal{N}(v)\})$$

GraphSAGE

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

GraphSAGE

- Now it’s inductive
  - No need to see all the nodes for making a forward pass
- More scalable during message passing
  - Only requires the k-hop subgraph to update a node’s embedding
- Flexible aggregation functions

GraphSAGE Limitations

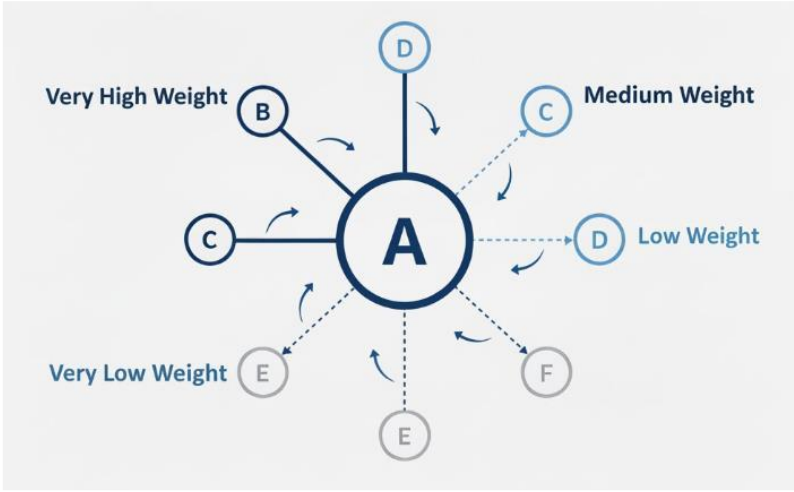
- Information loss in sampling phase
- Still – exponential growth within message passing
- Non-deterministic inference – due to random sampling
- LSTM is not permutation invariant
- Mean & Max aggregations treat all neighbors equally

GAT

- Idea: Stop treating all nodes equally
  - Make a weighted average learnable

- Isotropic vs Anisotropic
  - Treating all neighbors equally vs not

$$\vec{h}_i' = \sigma\left(\sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \vec{h}_j\right)$$



GAT Limitations

- Still exponential growth withing message passing
- Efficient in theory, but practically more expensive
  - Instead of just mean/max – compute attention function
- More parameters (for attention) – more vulnerability to overfit
- Attention is based on node features – and not structural information

Summary

- GCN: "I listen to everyone equally based on how popular they are." (Fast, Rigid).
- GraphSAGE: "I randomly pick a few people and average what they say." (Scalable, Inductive).
- GAT: "I listen to everyone, but I decide who to trust based on what they are saying." (Expressive, Heavy).

Demo (?)