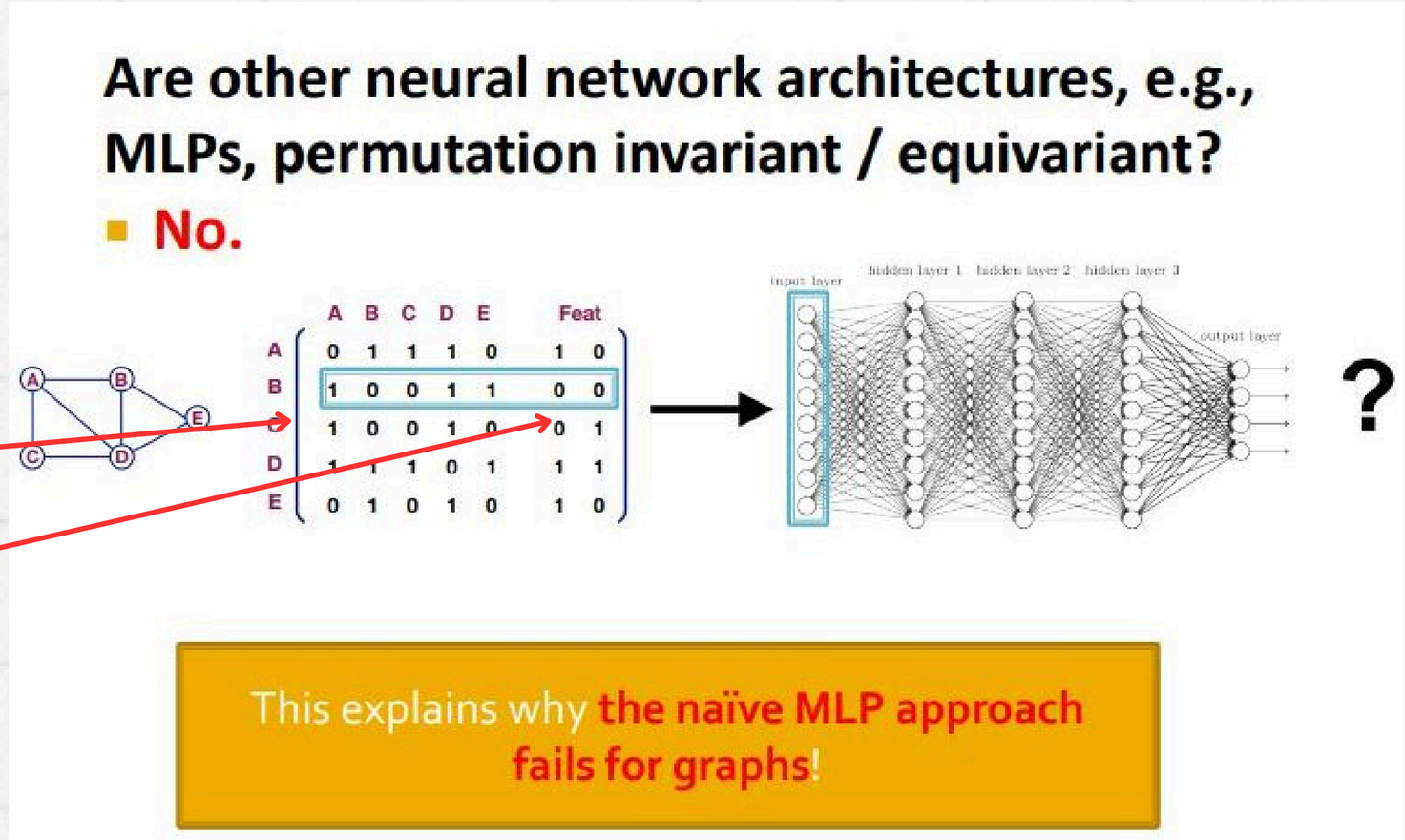
An Introduction to Graph Neural Networks (GNNs)

Amirmahdi Ansaripour

Problems

- Why do we need GNNs?
- What are permutation invariance and equi-variance?



1. Images are taken from Stanford CS224 Machine Learning with Graphs course

Permutation Invariance vs Equivariance

- Permutation: We change the node orders.
  For example for a graph with three nodes:
  (A, B, C) -> (B, A, C)

- If the output labels (predictions) do not change -> Invariance

$$f(A, X) = f(PAP^T, PX) \text{ for any permutation } P.$$

- If they change (permute) in the same way as the input -> equivariance

$$Pf(A, X) = f(PAP^T, PX)$$

- Every invariance is also equivariance.

Example

- Consider this matrix
(not an adjacency matrix for simplicity):

	A	B	C
A	1	2	3
B	4	5	6
C	7	8	9

- A permutation matrix is a matrix which:
  1. Each row and column contain only one 1
  2. It shouldn't be an identity matrix

$$P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Now we compute PAP^T:

$$\begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \end{array} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} =$$

$$\begin{array}{c} \\ C \\ A \\ B \end{array} \begin{array}{c} \begin{array}{ccc} C & A & B \end{array} \\ \begin{bmatrix} 9 & 7 & 8 \\ 3 & 1 & 2 \\ 6 & 4 & 5 \end{bmatrix} \end{array}$$

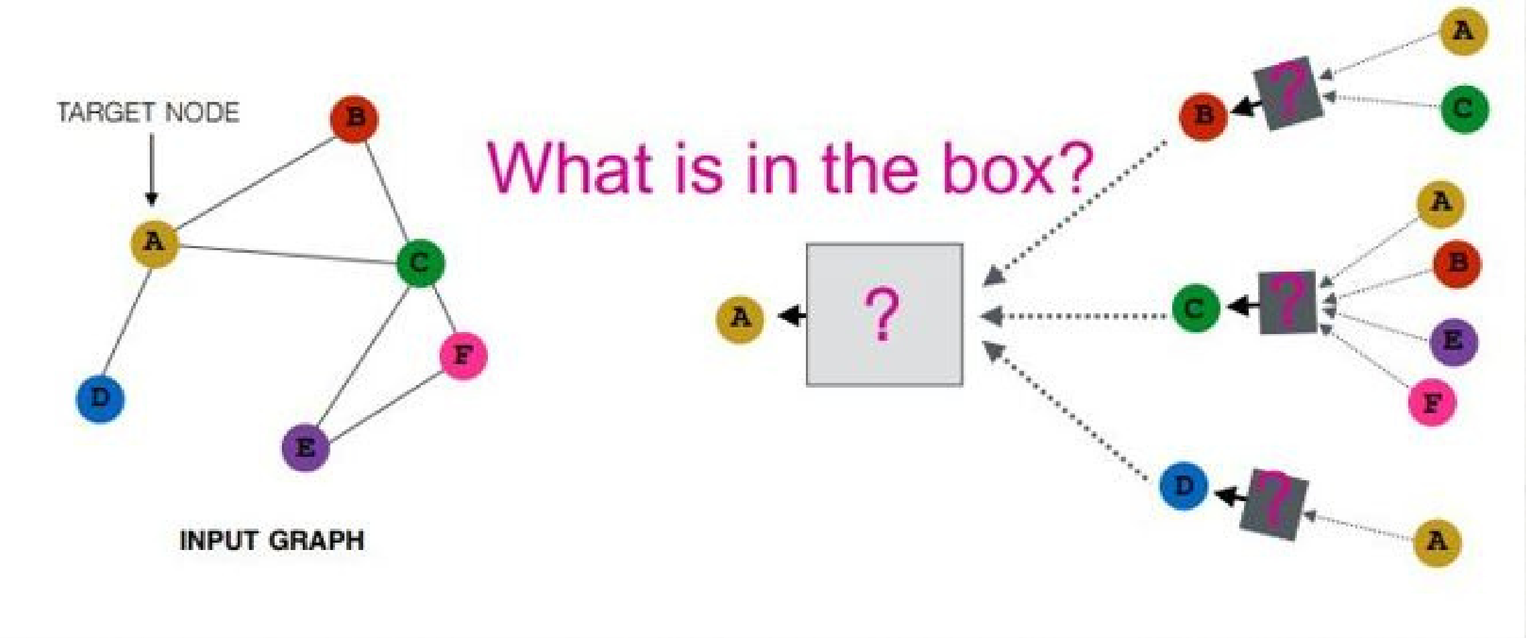
Both rows and columns are permutated

Rows are permutated but columns have no patterns

$$\begin{array}{c} \\ C \\ A \\ B \end{array} \begin{array}{c} \begin{array}{ccc} ? & ? & ? \end{array} \\ \begin{bmatrix} 7 & 8 & 9 \\ 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \end{array}$$

Obtaining Info from Neighbor Nodes

Each node computes an embedding based on its neighbors:



The process is recursive. Node A, which is the target node, has appeared before. Therefore, the initial embedding value is the node feature itself.

$$h_v^0 = x_v$$

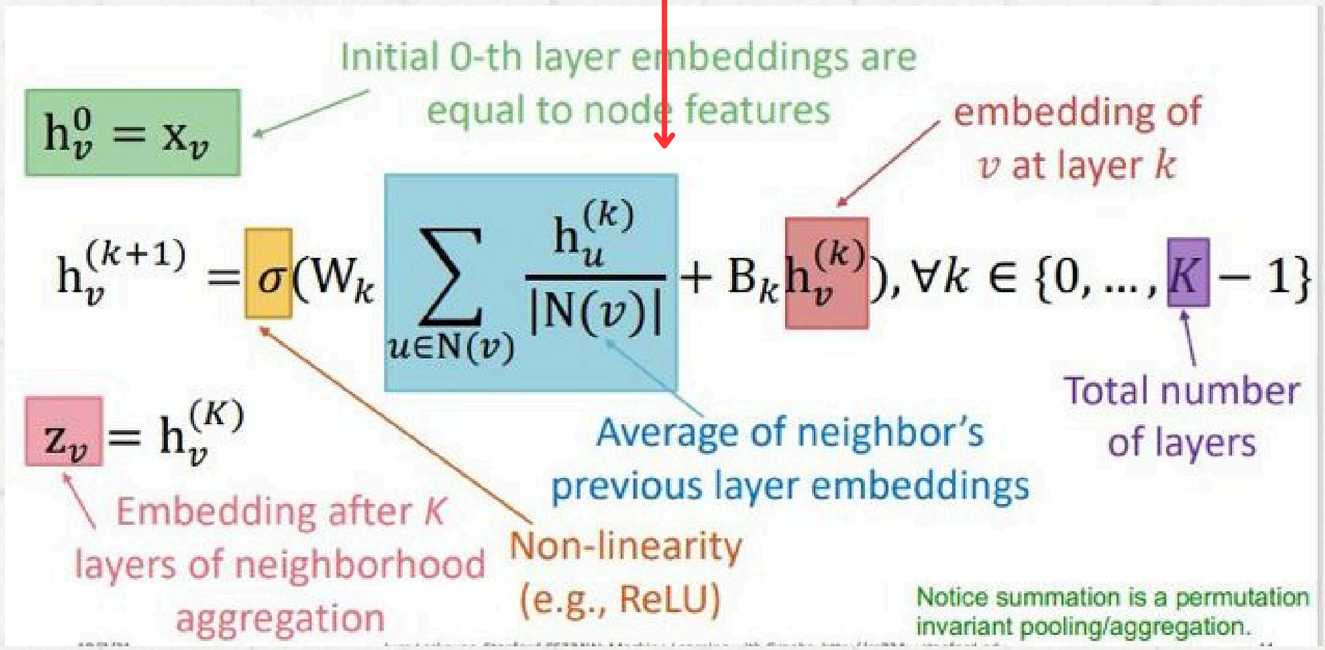
What Happens to the Embeddings?

1. The embeddings from neighbor nodes are summed and multiplied by a weight matrix. (W_k)

2. The target node embedding is also multiplied by a matrix. (B_k)

B_k and W_k are learnable matrices

3. Pay attention that summation is invariance and thus, equivariance.



Another approach: Each GNN has two main operations: AGGREGATE and COMBINE

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right), \quad h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

Coding Part

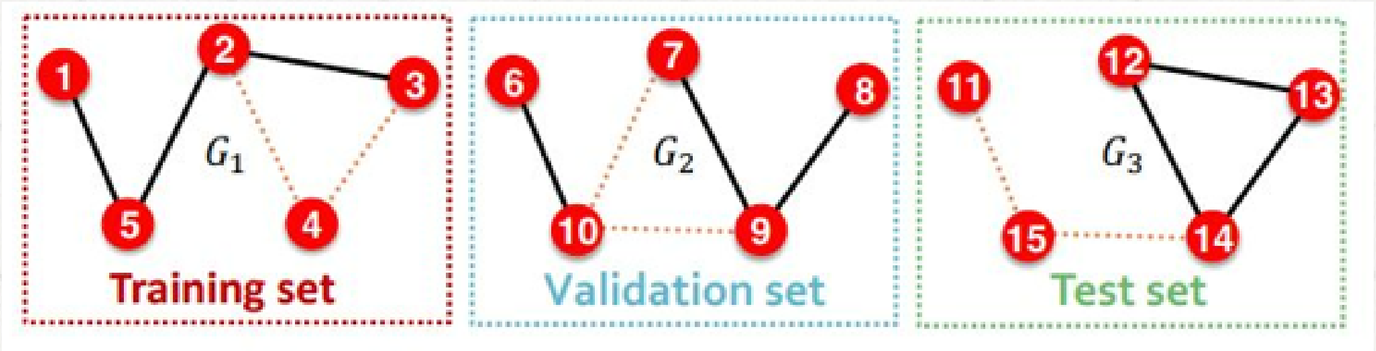
```
self.convs = nn.ModuleList()
self.convs.append(GCNConv(input_dim, hidden_dim))

for i in range(num_layers - 2):
  self.convs.append(GCNConv(hidden_dim, hidden_dim))
self.convs.append(GCNConv(hidden_dim, output_dim))

self.bns = nn.ModuleList()
for i in range(num_layers - 1):
  self.bns.append(BatchNorm1d(hidden_dim))

self.softmax = LogSoftmax(dim = 1)
```

```
for i in range(len(self.convs) - 1):
   x = self.convs[i](x, adj_t)
   x = self.bns[i](x)
   x = F.relu(x)
   x = F.dropout(x, p = self.dropout_rate, training = self.training)

x = self.convs[-1](x, adj_t) # The last block does not have any BN, rel
if self.return_embeds:   # Return embeddings not classes
  return x
else:
  res = self.softmax(x)
  return res     # Return classes
```

GCNConv is the correspondent of Conv2D basic layer in CNNs

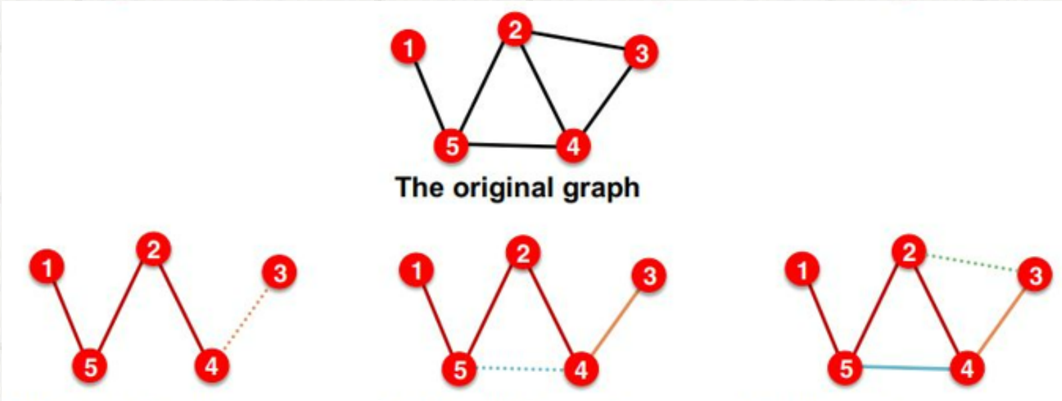
Transductive vs Inductive Setting

In transductive learning, both the training and test datasets are on the same graph. In inductive learning, however, the graphs are different.



Inductive Learning

Subtle point:
Difference between GraphSage and GCN



Transductive Learning

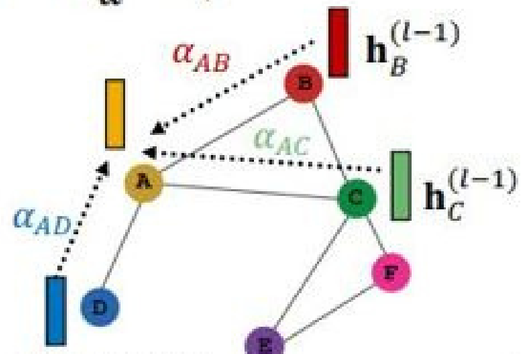
Attention Mechanism in Graphs (GAT)

- How much the message of node v is important for node u? (u and v are neighbors)

$$\mathbf{h}_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}\right)$$

Weighted sum using α_{AB}, α_{AC}, α_{AD}:

$$\mathbf{h}_A^{(l)} = \sigma\left(\alpha_{AB}\mathbf{W}^{(l)}\mathbf{h}_B^{(l-1)} + \alpha_{AC}\mathbf{W}^{(l)}\mathbf{h}_C^{(l-1)} + \alpha_{AD}\mathbf{W}^{(l)}\mathbf{h}_D^{(l-1)}\right)$$



How the Attention Coefficients are obtained?

- The coefficients are calculated by a function whose inputs are weight matrices and previous embedding values. Finally, the coefficients are normalized:

$$e_{vu} = a(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}, \mathbf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

Here, a is the function calculating attention between u and v.

$$\alpha_{vu} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})}$$

Normalized (final) attention values

Coding Part

MLP weights

GNN (any type) learnable matrix

Node Embedding

$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{a_l}^T \mathbf{W}_1 \vec{h_i} + \vec{a_r}^T \mathbf{W}_r \vec{h_j}\right)\right)}{\sum_{k \in N_i} \exp\left(\text{LeakyReLU}\left(\vec{a_l}^T \mathbf{W}_1 \vec{h_i} + \vec{a_r}^T \mathbf{W}_r \vec{h_k}\right)\right)}$$

alpha_i + alpha_j in the code

```
def message(self, x_j, alpha_j, alpha_i, index, ptr, size_i):

    ##############################################################################
    att_weight = F.leaky_relu(alpha_i + alpha_j,
                              negative_slope = self.negative_slope)
    if ptr:
      att_weight = torch_geometric.utils.softmax(att_weight, ptr = ptr)
    else:
      att_weight = torch_geometric.utils.softmax(att_weight, index = index,
                                                 num_nodes = size_i)

    att_weight = F.dropout(att_weight, p=self.dropout)
    out = att_weight * x_j
    ##############################################################################

    return out
```

In this example, the attention function mentioned on slide 8 is a simple MLP with a LeakyReLU activation.

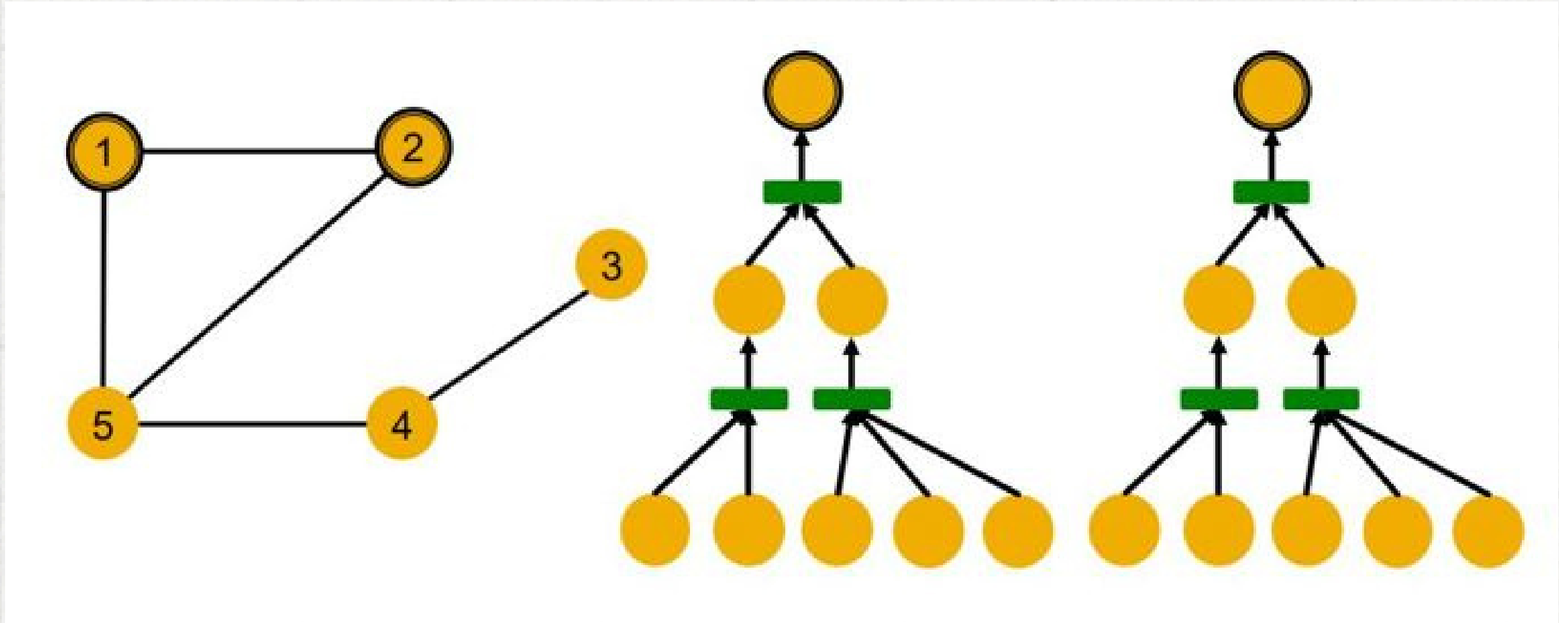
Coding Part

```
def forward(self, x, edge_index, size = None):

    H, C = self.heads, self.out_channels

    ###########################################################################
    x_l = self.lin_l(x).reshape(-1, H, C)
    x_r = self.lin_r(x).reshape(-1, H, C)
    alpha_l = self.att_l * x_l
    alpha_r = self.att_r * x_r
    out = self.propagate(edge_index, x = (x_l, x_r),
                         alpha = (alpha_l, alpha_r), size = size)
    out = out.reshape(-1, H*C)
    ###########################################################################

    return out
```

x_l is the neighbor and x_r is the central node. The attention weights are multiplied by the embeddings and then, the message is propagated from the neighbor (x_l) to the central (x_r).

The Most Expressive GNN

Obviously, the node embedding (output) of these two graphs, belonging to nodes 1 and 2, should be identical. Why? Because nodes 1 and 2 have the same degree and neighbors.

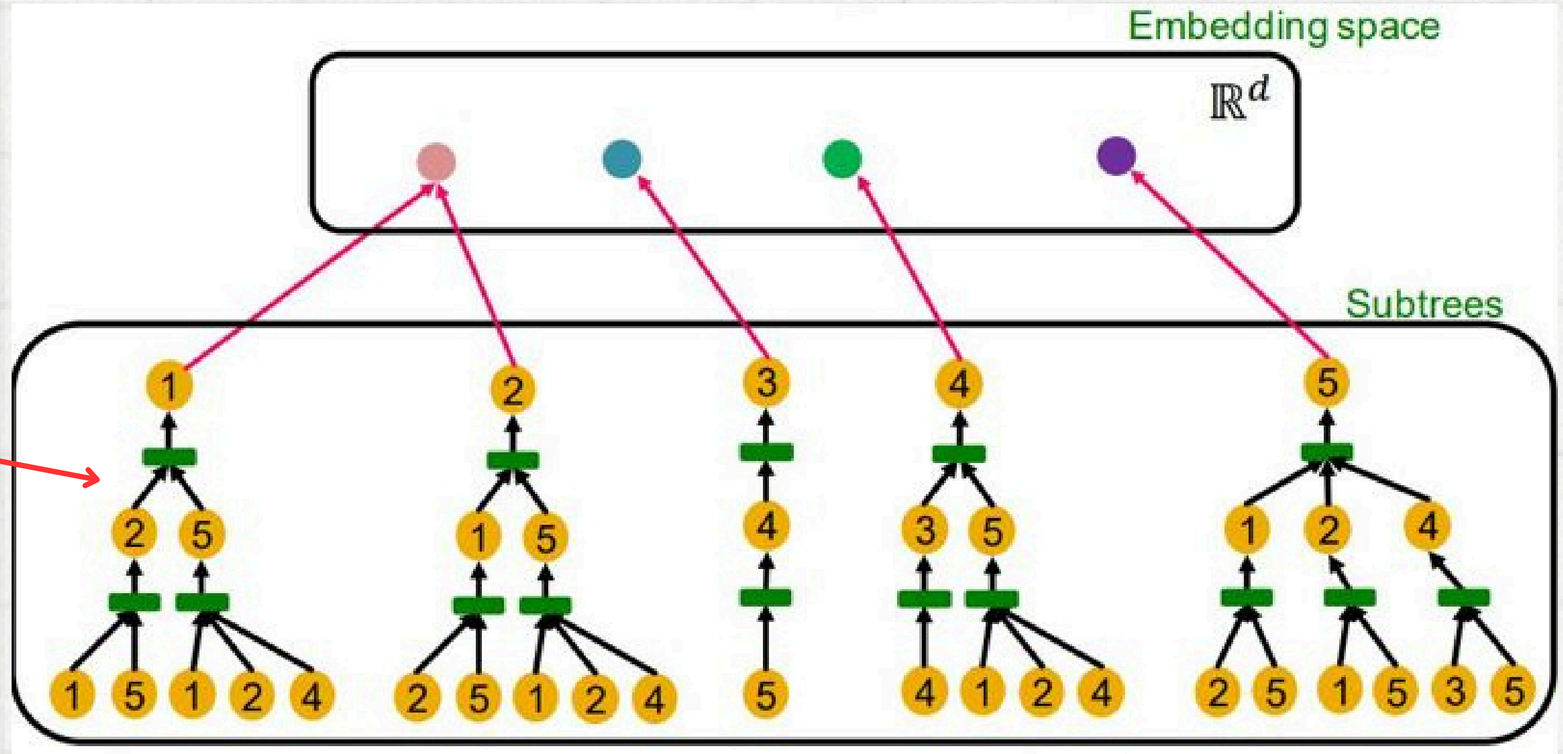


On the other side, the node embedding for nodes 3 and 4 must be different.

The Most Expressive GNN

More specifically: There should be an **injection** between the computation graphs (node embeddings) and their embedding values

These are called computational graphs



The Most Expressive GNN

- The expressive power of a GNN is determined by the aggregation function it uses for aggregating messages from neighbors.

- An injective aggregation function leads to the most expressive GNN.

Now let’s analyze the aggregation functions used in GCN and GraphSage.

The Most Expressive GNN

To study the representational power of a GNN, we analyze when a GNN maps two nodes to the same location in the embedding space. Intuitively, a maximally powerful GNN maps two nodes to the same location *only if* they have identical subtree structures with identical features on the corresponding nodes. Since subtree structures are defined recursively via node neighborhoods (Figure 1), we can reduce our analysis to the question whether a GNN maps two neighborhoods (*i.e.*, two multisets) to the same embedding or representation. A maximally powerful GNN would *never* map two different neighborhoods, *i.e.*, multisets of feature vectors, to the same representation. This means its aggregation scheme must be *injective*. Thus, we abstract a GNN's aggregation scheme as a class of functions over multisets that their neural networks can represent, and analyze whether they are able to represent injective multiset functions.

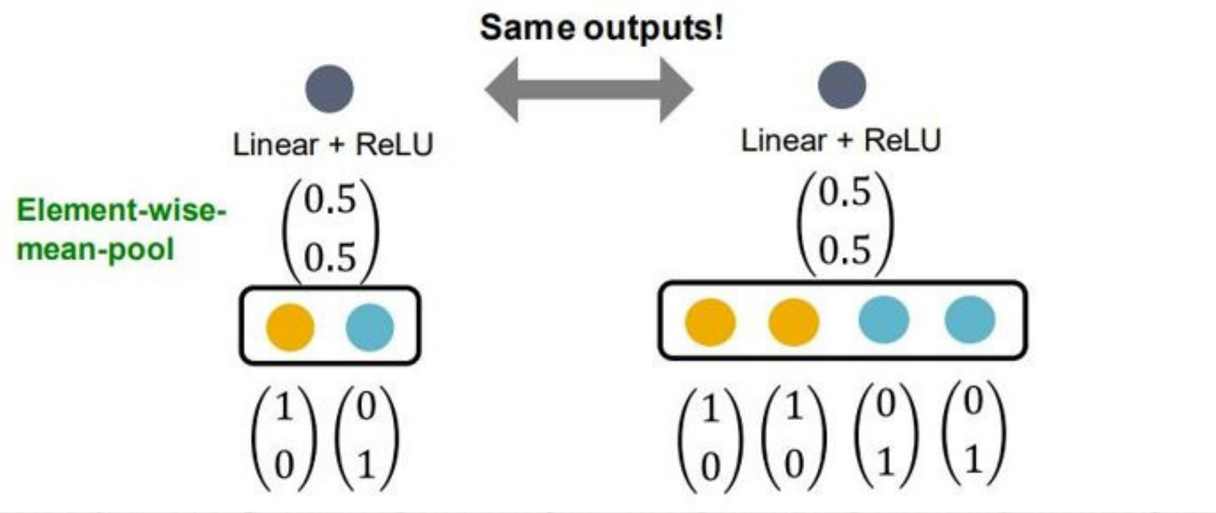
Reference: Xu K., Hu W., Leskovec J., Jegelka S., How Powerful are Graph Neural Network?, ICLR 2019.

The Most Expressive GNN

GCN: Not injective!!

$$h_v^{(k)} = \mathrm{ReLU}\left(W \cdot \mathrm{MEAN}\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v) \cup \{v\}\right\}\right)$$

The COMBINE step is here,
merged inside the AGGREGATE step



The Most Expressive GNN

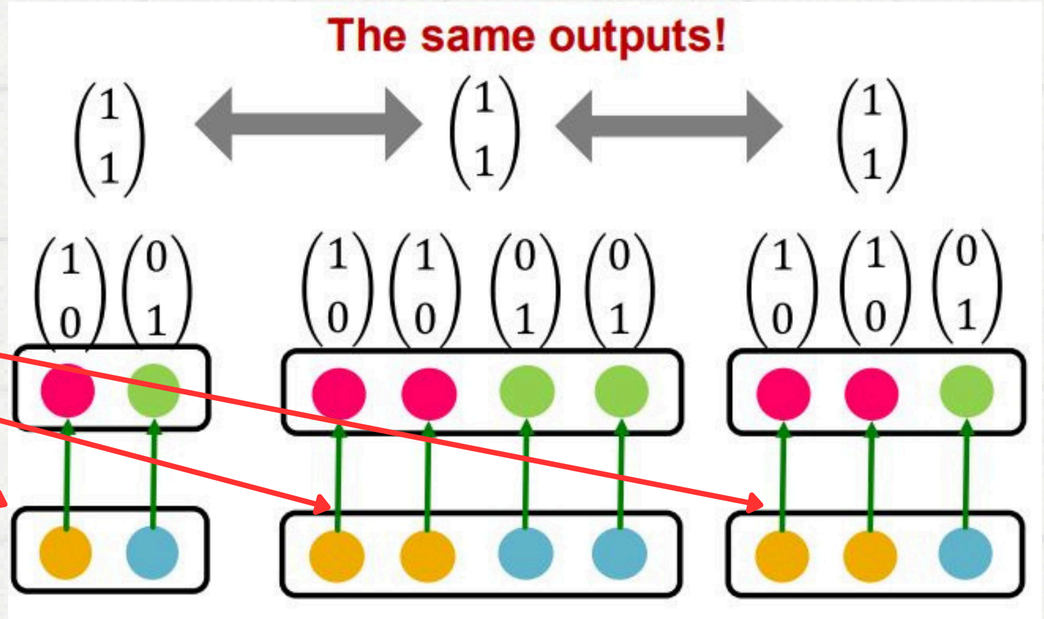
GraphSAGE: Not injective!!

$$a_v^{(k)} = \text{MAX}\left(\left\{\text{ReLU}\left(W \cdot h_u^{(k-1)}\right),\ \forall u \in \mathcal{N}(v)\right\}\right)$$

AGGREGATE step

$$W \cdot \left[h_v^{(k-1)}, a_v^{(k)}\right]$$

COMBINE step: Previous node embedding and neighbors' embeddings



3 different graphs having the same embedding after MAX aggregation

Graph Isomorphism Network (GIN)

When are two graphs called isomorphic?

Two graphs with vertex sets $V(G)$ and $V(H)$ are called isomorphic if and only if there is a bijective[1] function between these sets $f : V(G) \to V(H)$ such that:

$$uv \in E(G) \Leftrightarrow f(u)f(v) \in E(V) \quad \forall u, v \in V(G)$$

We should find a GNN which can embed isomorphic graphs to one point and be injective at the same time.

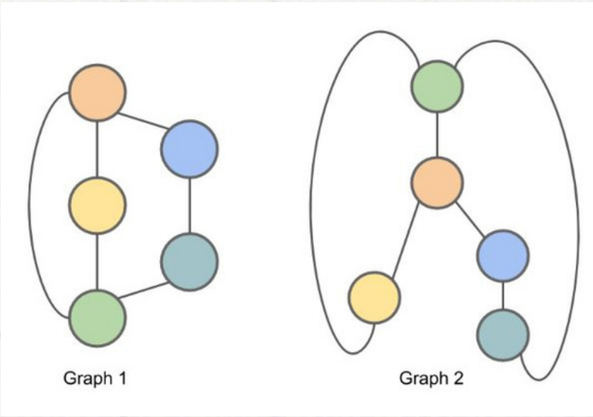
1. یک به یک و پوشا

A Practical Algortihm: Weisfeiler-Lehman (WL) Test

- It can detect whether two graphs are non-isomorphic!
- However, it is a necessary but insufficient[1] algorithm for graph isomorphism.
- In other words, if it succeeds, the input graphs are possibly isomorphic.

1. لازم ولی نه کافی

WL Test: Example

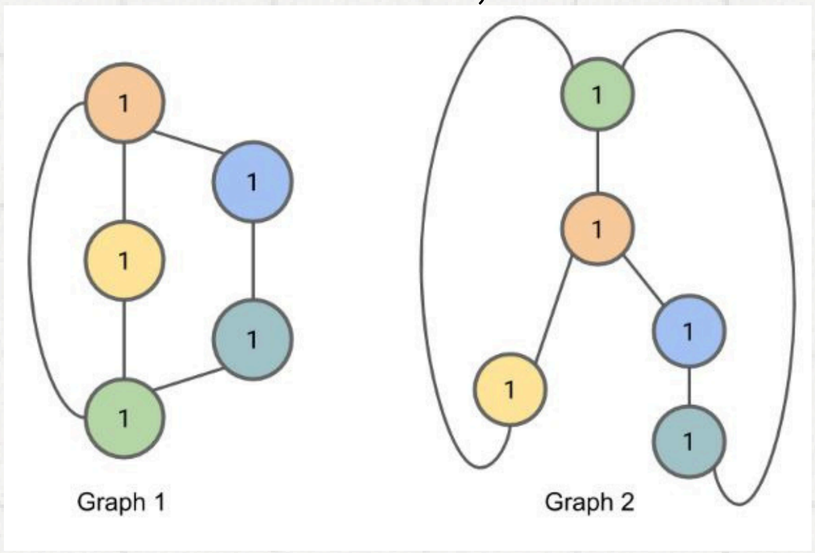


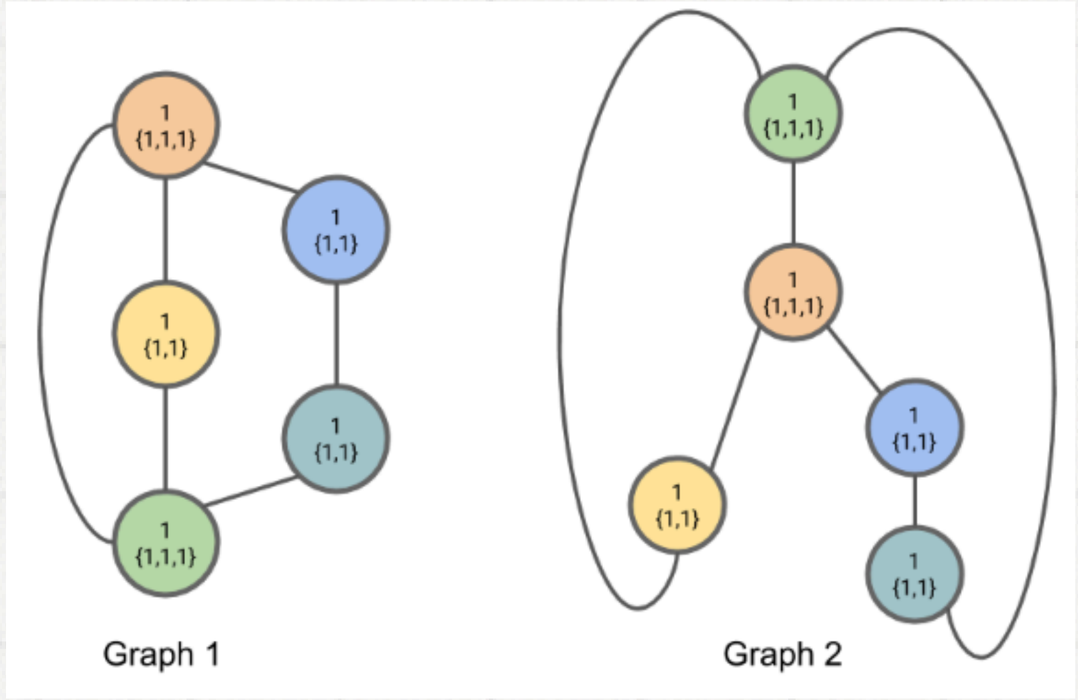
Goal: Identical nodes must have equal hash values, i.e., belong to identical partitions.

We define two kinds of labels for each node: normal label ($L_{t,i}$) and hashed label ($H_{t,i}$). t specifies the current step and i specifies the node index.

$L_{t,i}$ has 2 parts → Construct $L_{0,i}$ based on node i's neighbors

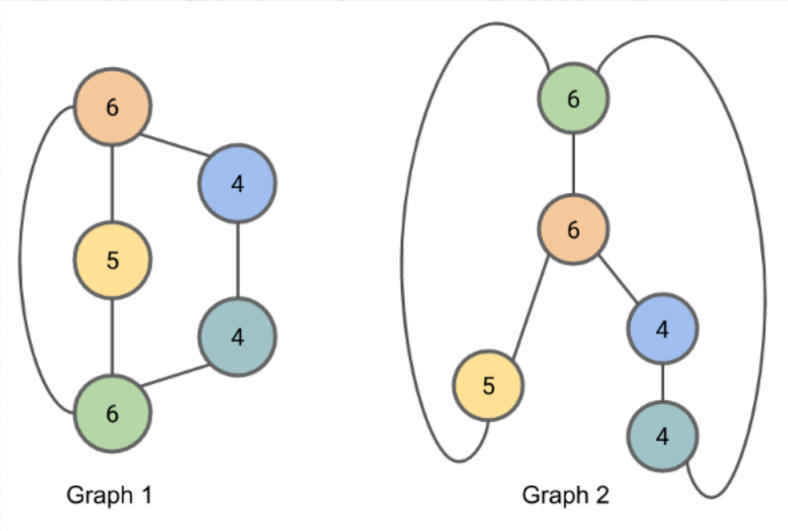
Initialize $H_{0,i} = 1$



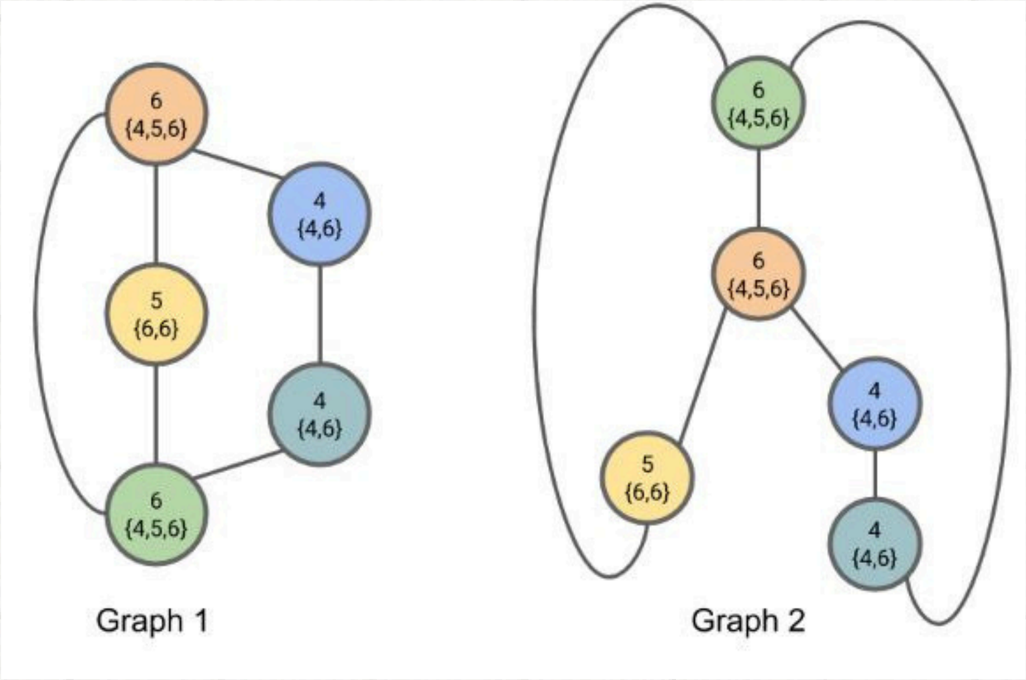


WL Test: Example

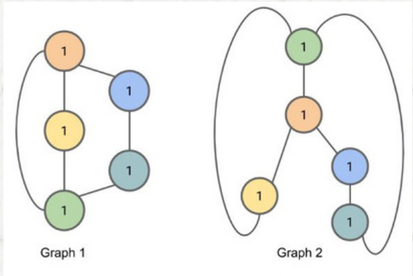
update hashed labels by partitioning identical L_is



compute $L_{1,i}$ based on neighbors

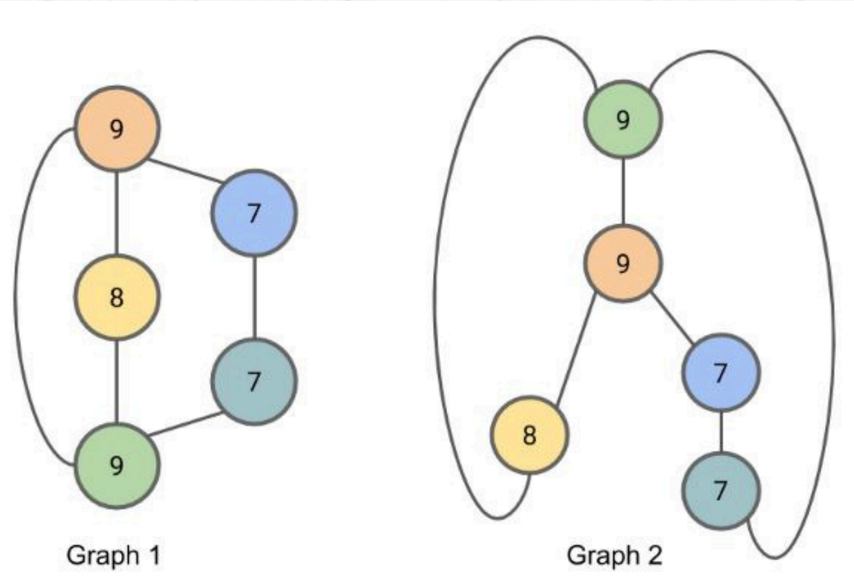


Did partitioning change from prev state? Yes.



WL Test: Example

update hashed labels by partitioning identical L_is



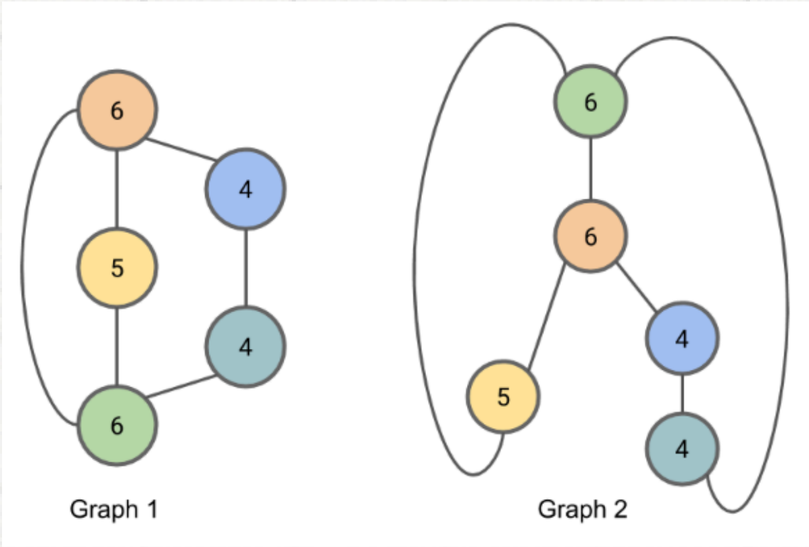
Did partitioning change from prev state? No

-> Finish the algorithm

Now that the algorithm is finished, are Graph1 and Graph2 isomorphic?

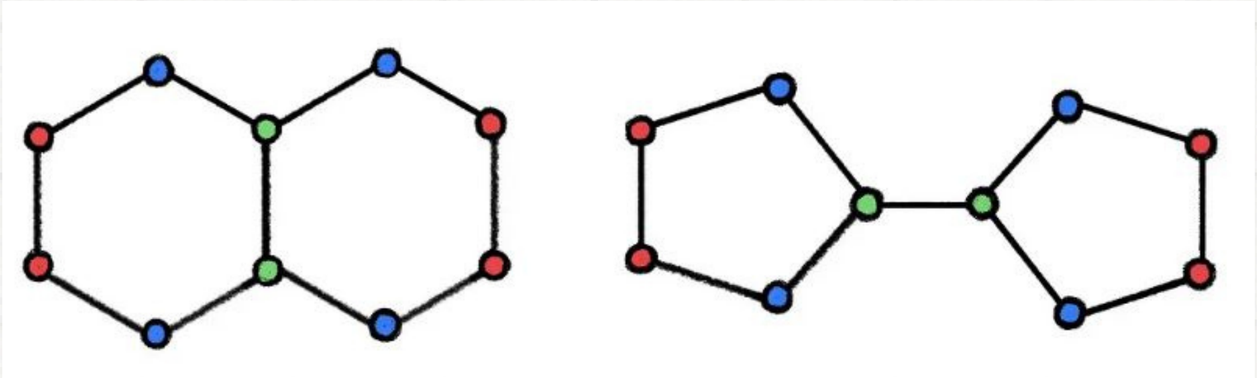
Their final partitioning is the same, so we can possibly say they are isomorphic.

Please note that WL test proves non-isomorphism



WL Test: Example

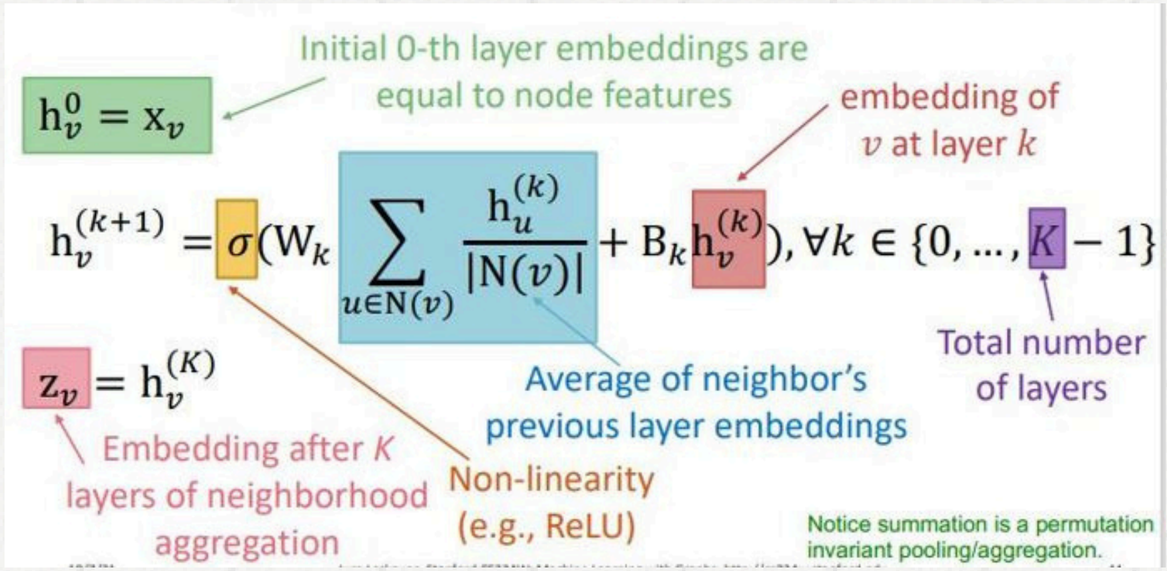
One example for showing why we cannot 100% be sure about isomorphism with WL-Test:

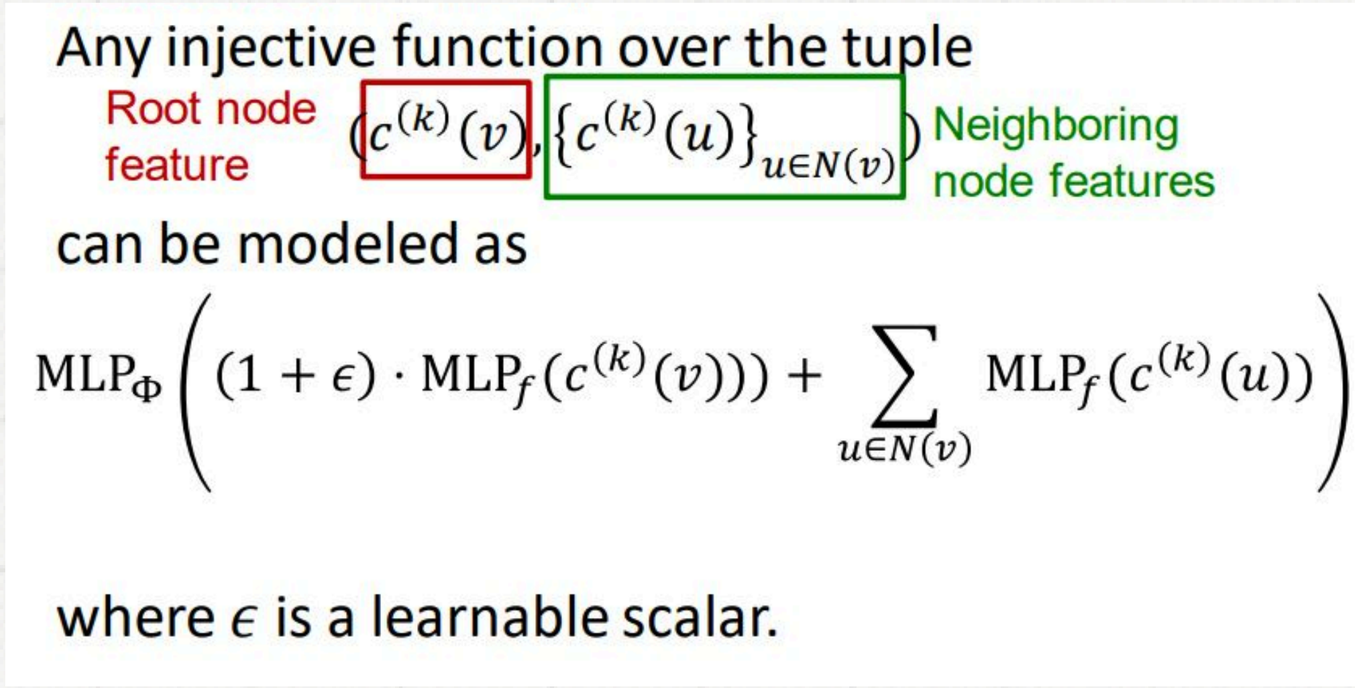


Here, partitioning are the same, but graphs aren’t isomorphic.

Using MLP as the Hash Function of WL Test

Comparison between the AGGREGATE functions of basic GNNs (GCN, GraphSAGE, GAT) and GIN:





The entire $\sigma(\dots)$ is one MLP

Here we have two MLPs. One for embedding (new) and one for combining (like GCN)

Using MLP as the Hash Function of WL Test

The mathematics behind the GINConv formulation and why using MLP for predicting f and ϕ

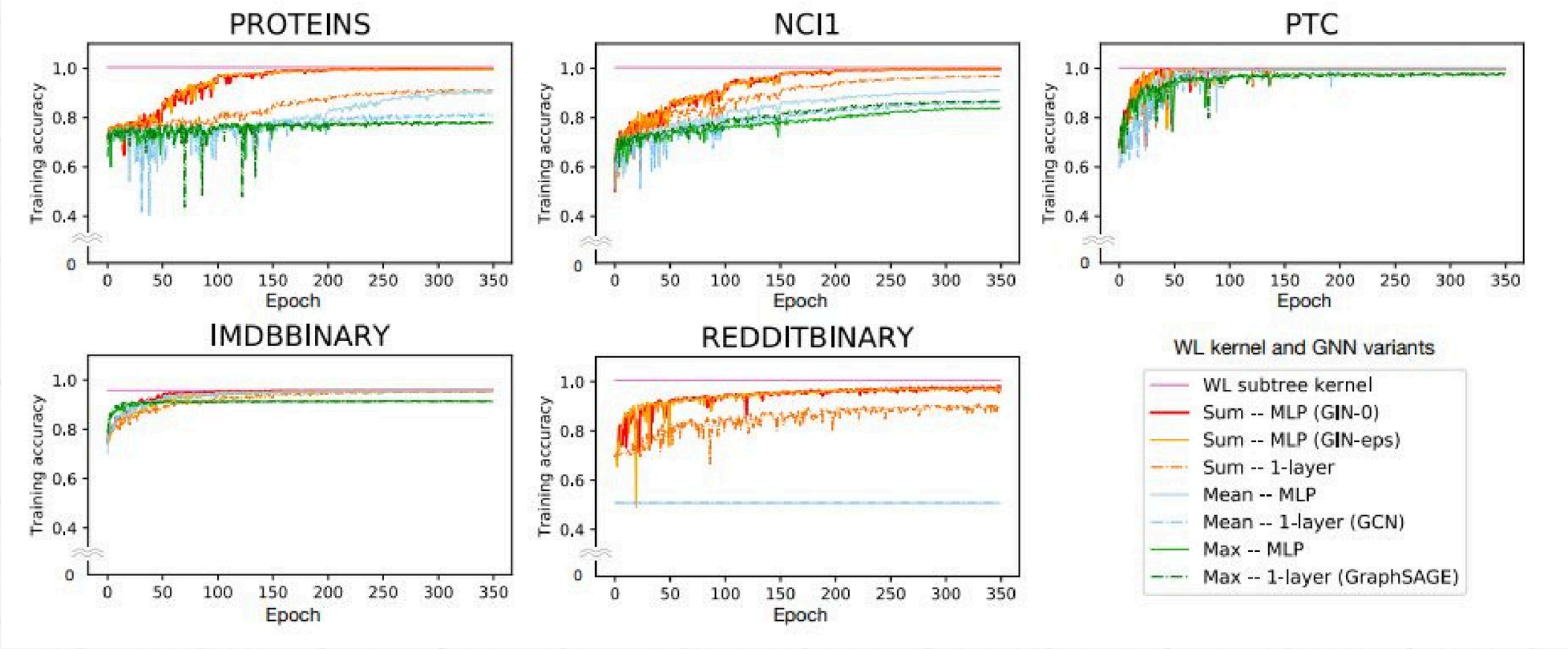
Corollary 6. *Assume $\mathcal{X}$ is countable. There exists a function $f : \mathcal{X} \to \mathbb{R}^n$ so that for infinitely many choices of ϵ, including all irrational numbers, $h(c, X) = (1+\epsilon) \cdot f(c) + \sum_{x \in X} f(x)$ is unique for each pair (c, X), where $c \in \mathcal{X}$ and $X \subset \mathcal{X}$ is a multiset of bounded size. Moreover, any function g over such pairs can be decomposed as $g(c, X) = \varphi\left((1+\epsilon) \cdot f(c) + \sum_{x \in X} f(x)\right)$ for some function φ.*

We can use multi-layer perceptrons (MLPs) to model and learn f and φ in Corollary 6, thanks to the universal approximation theorem (Hornik et al., 1989; Hornik, 1991). In practice, we model

Any continuous function can be approximated (with arbitrary precision) by an MLP with a sufficiently large hidden layer

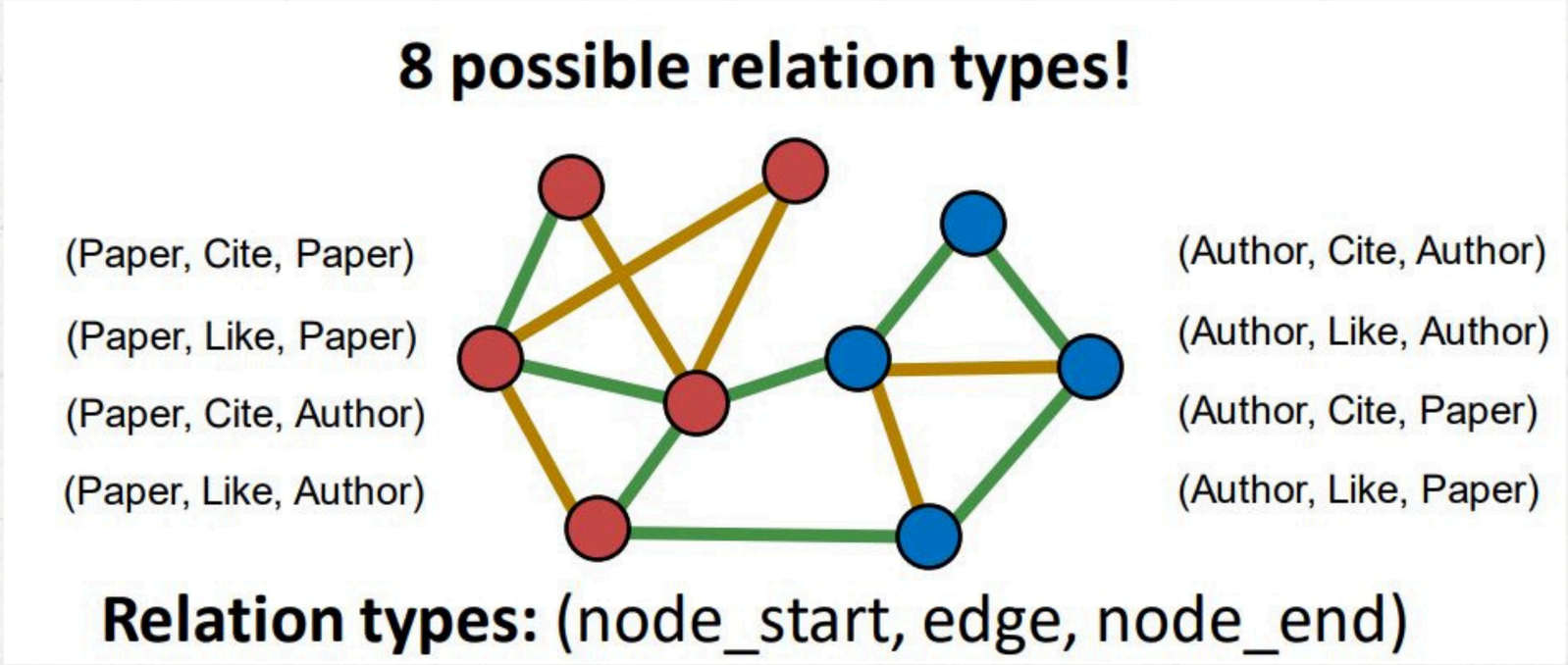
Experiments

GIN-0 and GIN-eps are compared against other GNN architectures on the graph classification task. The datasets are for bioinformatics and social network.



Next Topics

Homogenous graphs



Semi-supervised learning on graphs: Node classification methods (Correct & Smooth, Label Propagation, etc.)