Attentive Graph Neural Networks

Prof. O-Joun Lee

Dept. of Artificial Intelligence,
The Catholic University of Korea
ojlee@catholic.ac.kr

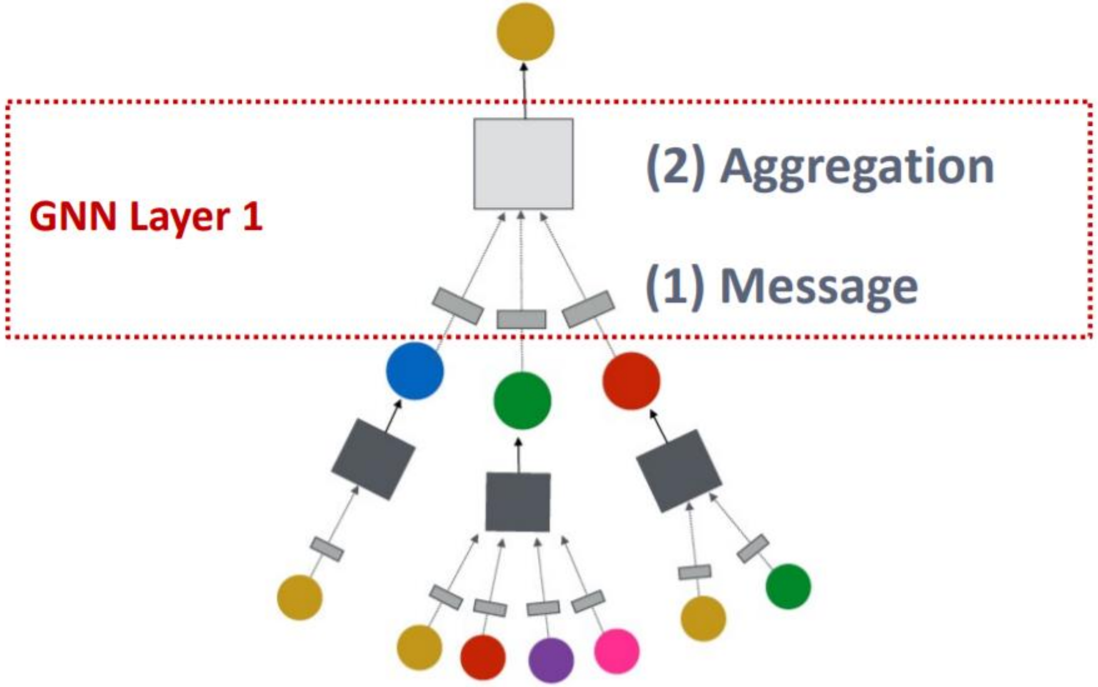
네트워크 과학 연구실 NETWORK SCIENCE LAB

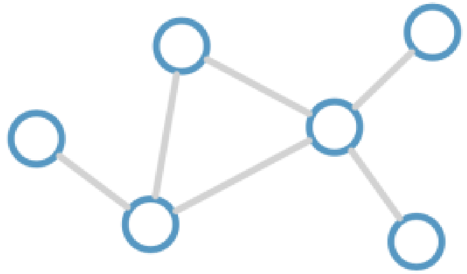
가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA

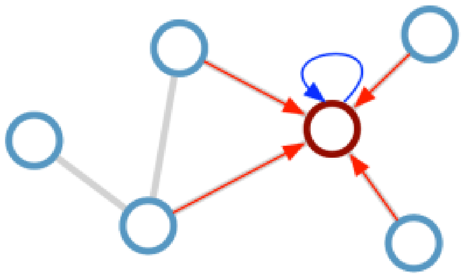
Contents

Recap: Graph Convolutional Networks (GCNs)

- GNN Layer = **Message** + **Aggregation**
  - Message COMPUTATION
    - how to make each neighborhood node as embedding?
  - Message AGGERGATION
    - how to combine those embeddings?







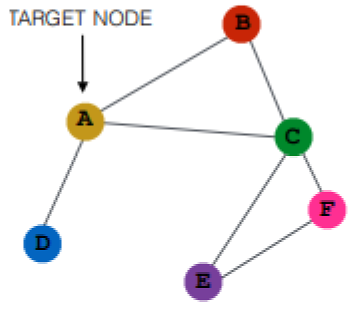
Update rule: $$\mathbf{h}_i^{(l+1)} = \sigma\left(\mathbf{h}_i^{(l)}\mathbf{W}_0^{(l)} + \sum_{j\in\mathcal{N}_i}\frac{1}{c_{ij}}\mathbf{h}_j^{(l)}\mathbf{W}_1^{(l)}\right)$$

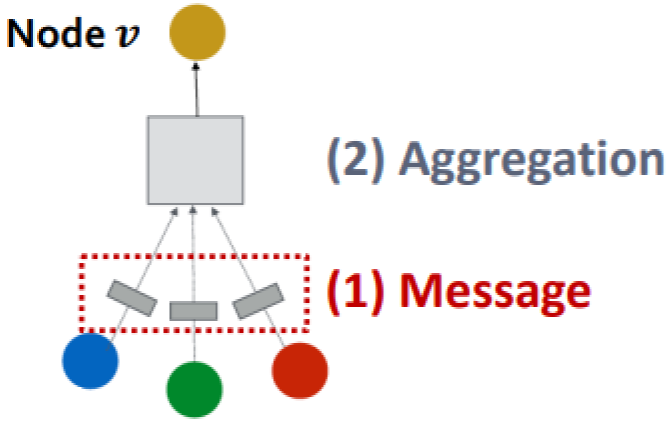
Kipf & Welling (ICLR 2017), related previous works by Duvenaud et al. (NIPS 2015) and Li et al. (ICLR 2016)

GNN Layer: Message Computation

- **Intuition**: Each node will create a message, which will be sent to other nodes later
- **Example**: A Linear layer $\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)}$
  - Multiply node features with weight matrix $\mathbf{W}^{(l)}$

Message function: $\mathbf{m}_u^{(l)} = \mathrm{MSG}^{(l)}\left(\mathbf{h}_u^{(l-1)}\right)$





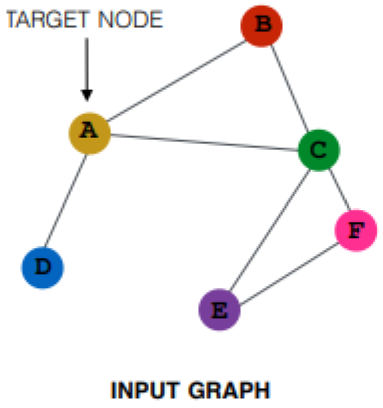
GNN Layer: Message Aggregation

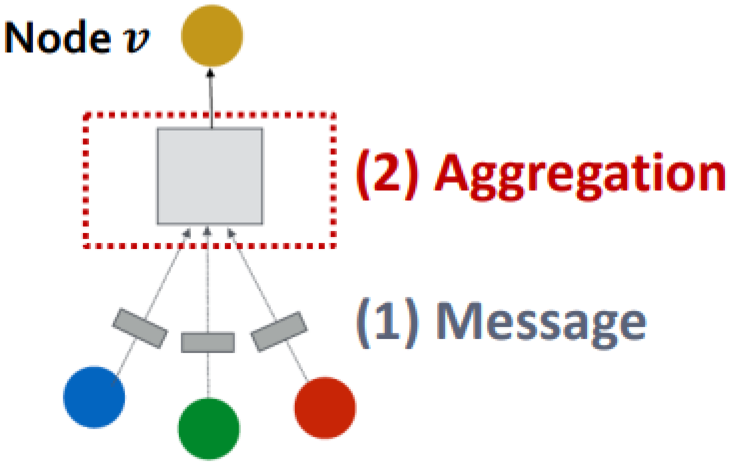
- **Intuition**: Each node will aggregate the messages from node v's neighbors

$$\mathbf{h}_v^{(l)} = \mathrm{AGG}^{(l)}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)$$

- **Example**: Sum(·), Mean(·) or Max(·) aggregator

$$\mathbf{h}_v^{(l)} = \mathrm{Sum}(\{\mathbf{m}_u^{(l)}, u \in N(v)\})$$





GNN Layer: Message Aggregation Issue

- **Issue**: Information from node v itself could get lost
  - Computation of $\mathbf{h}_v^{(l)}$ does not directly depend on $\mathbf{h}_v^{(l-1)}$

- **Solution**: Include $\mathbf{h}_v^{(l-1)}$ when computing $\mathbf{h}_v^{(l)}$
  - (1) **Message**: compute message from node v itself

$$\mathbf{m}_u^{(l)} = \mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)} \qquad \mathbf{m}_v^{(l)} = \mathbf{B}^{(l)}\mathbf{h}_v^{(l-1)}$$

  - (2) **Aggregation**: After aggregating from neighbors, we can aggregate the message from node v itself
    - Via concatenation or summation

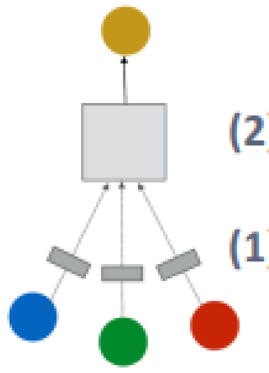
$$\mathbf{h}_v^{(l)} = \mathrm{CONCAT}\Big(\underbrace{\mathrm{AGG}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)}_{\text{First aggregate from neighbors}}, \overbrace{\mathbf{m}_v^{(l)}}^{\text{Then aggregate from node itself}}\Big)$$

Graph Convolutional Networks (GCN)

- Pure Graph Convolutional Networks (GCN)

equally important to v

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



- **Message**: Each neighbour u:

$$\mathbf{m}_u^{(l)} = \boxed{\frac{1}{|N(v)|}} \mathbf{W}^{(l)} \mathbf{h}_u^{(l-1)}$$

- **Aggregation**: Sum over messages from neighbors, then apply activation

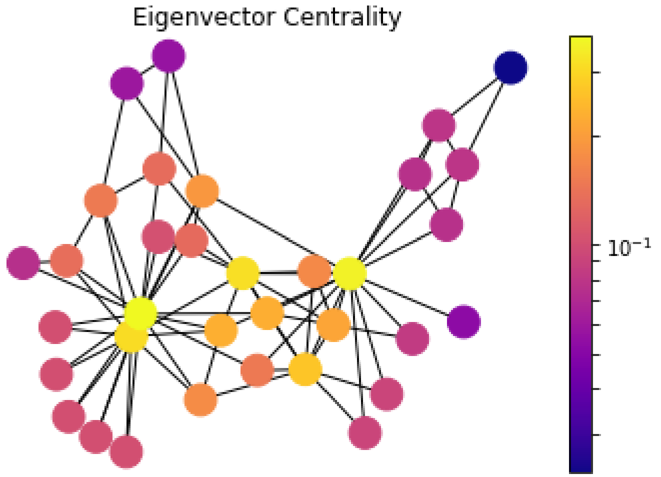
$$\mathbf{h}_v^{(l)} = \sigma\left(\mathrm{Sum}\left(\left\{\mathbf{m}_u^{(l)}, u \in N(v)\right\}\right)\right)$$

→ All neighbors $u \in N(v)$ are equally important to node v

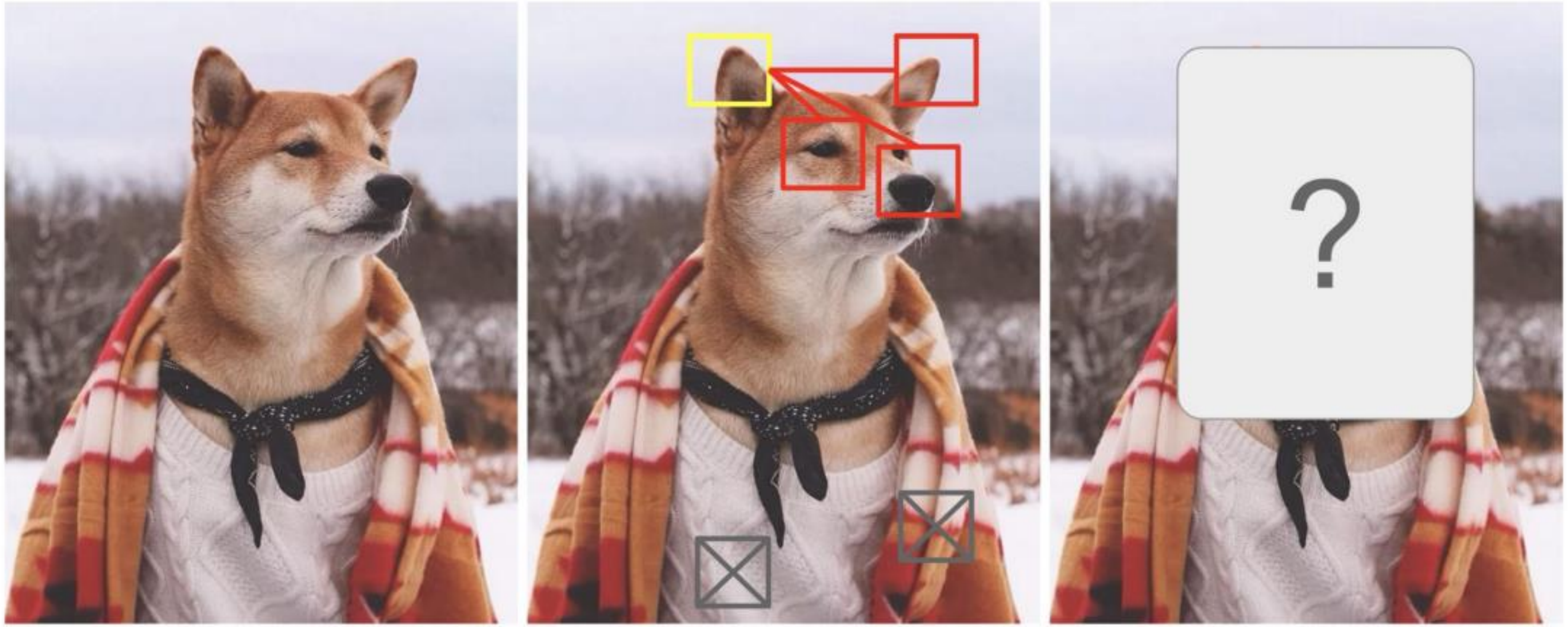
T. Kipf, M. Welling. Semi-Supervised Classification with Graph Convolutional Networks, ICLR 2017

Not all node's neighbors are equally important

- Attention is a mechanism that allows a network to focus on certain parts of the input when processing it
- The attention focuses on the important parts of the input data and fades out the rest.
  - **Idea**: the neural network should devote more computing power on that small but important part of the data.
  - Which part of the data is more important depends on the context and is learned through training.



➢ We deduce something by paying attention to something that is relatively more important.



https://lilianweng.github.io/posts/2018-06-24-attention/

Attention mechanism in NLP (Seq2Seq)

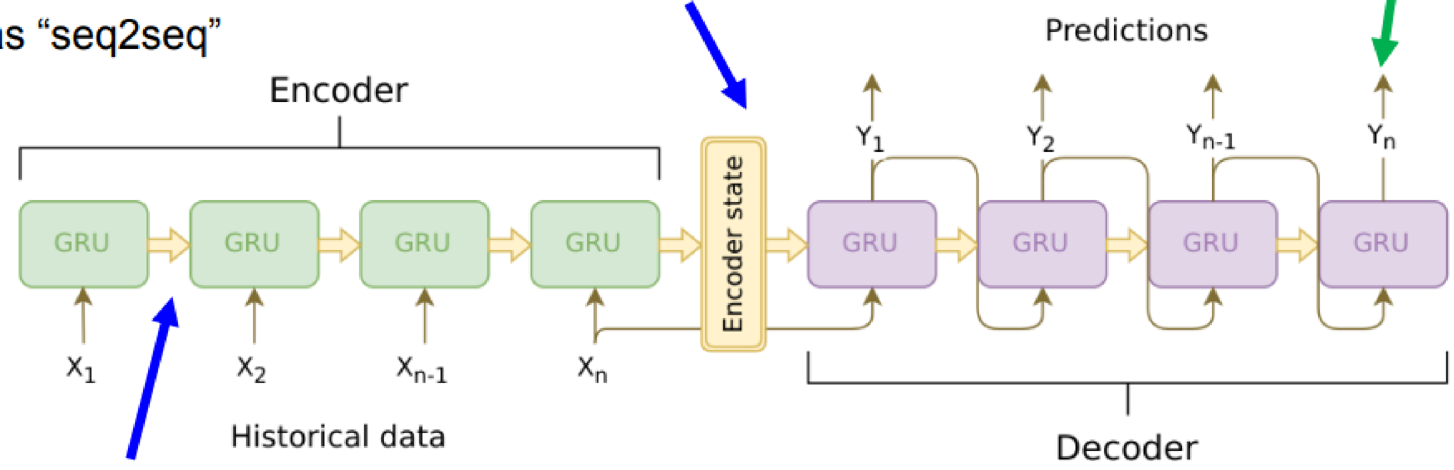
- RNN encoder-decoder for neural machine translation:
  - In capability of remembering long sentences : Often it has forgotten the first part once it completes processing the whole input. The attention mechanism was born to resolve this problem.

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c})$$

$$p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c}) = g(\mathbf{y}_{t-1}, \mathbf{s}_t, \mathbf{c})$$

$$\mathbf{c} = q(\{\mathbf{h}_1, \dots, \mathbf{h}_T\}) = \mathbf{h}_T$$

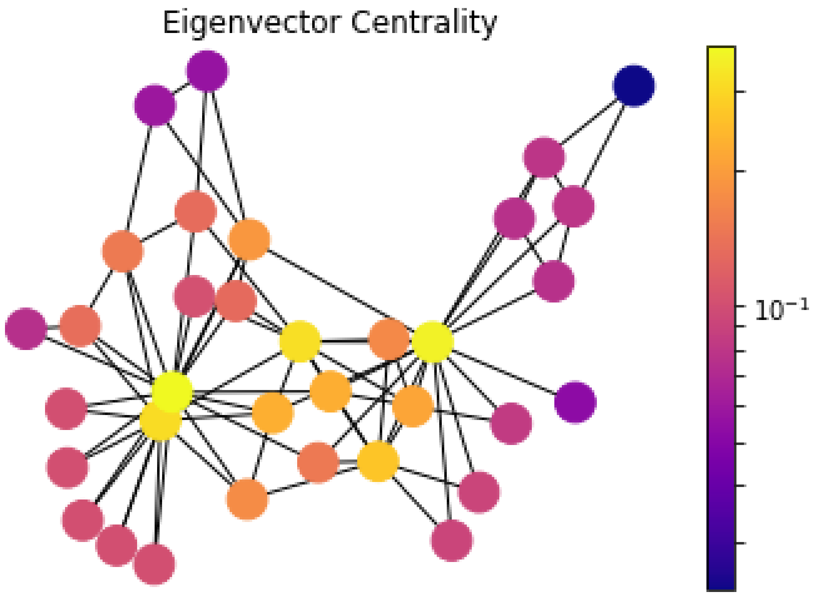
referred as “seq2seq”



$\mathbf{h}_t = f(\mathbf{x}_t, \mathbf{h}_{t-1}) \in \mathbb{R}^n$: hidden state at time t

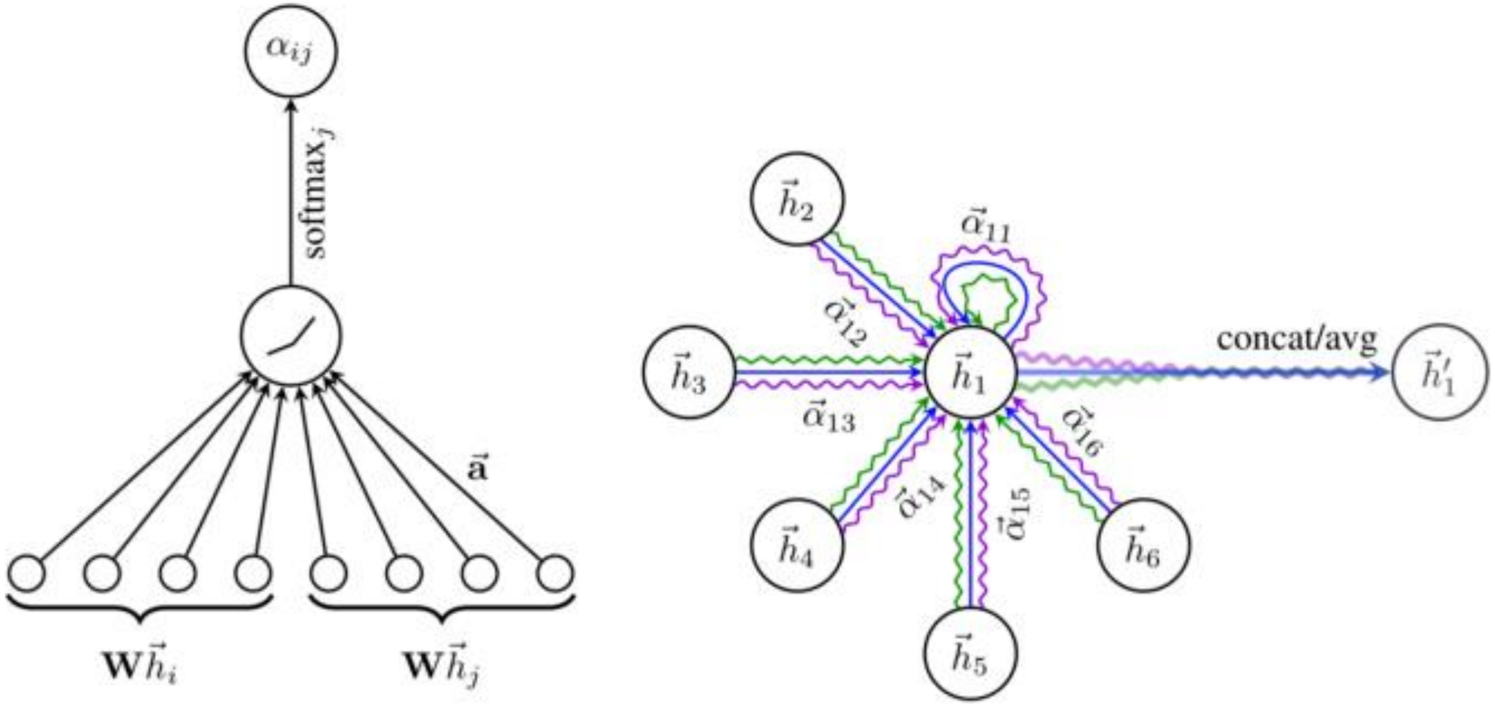
Why Attention in GNN?

- GNN compute node representations from representations of neighbours.
- Nodes can have largely different neighbourhood sizes.
- Not all neighbours have relevant information for a certain node.
- Attention mechanism allow to adaptively weight the contribution of each neighbour when updating a node.



Graph Neural Networks (GNNs) with Attention

- **Attention means:** assign an attention coefficient to each neighbor, indicating the importance of that neighbor's features for the feature update of the node.



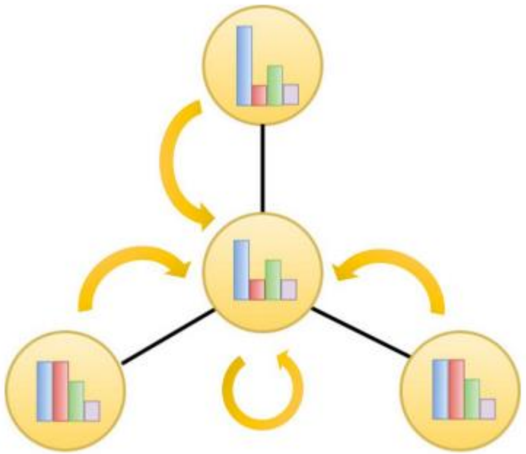
[Figure from Veličković et al. (ICLR 2018)]

Monti et al. (CVPR 2017), Hoshen (NIPS 2017), Veličković et al. (ICLR 2018)

Comparison between GCN and GAT

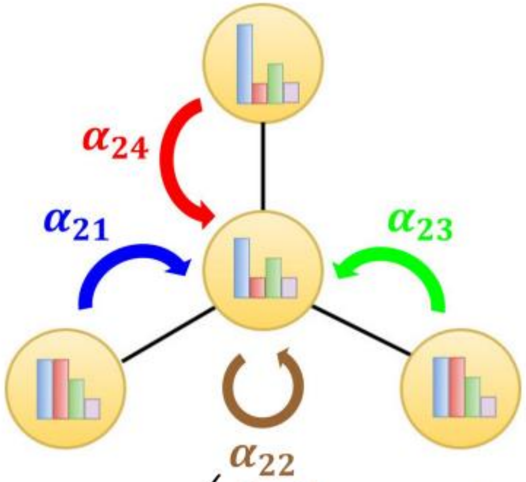
- The key difference between GAT and GCN is how the information from the one-hop neighborhood is aggregated.

Vanilla GCN updates information of neighbor nodes with same importance



$$H^{(l+1)} = \sigma\left(\sum_{j \in N(i)} H_j^{(l)} W^{(l)}\right)$$

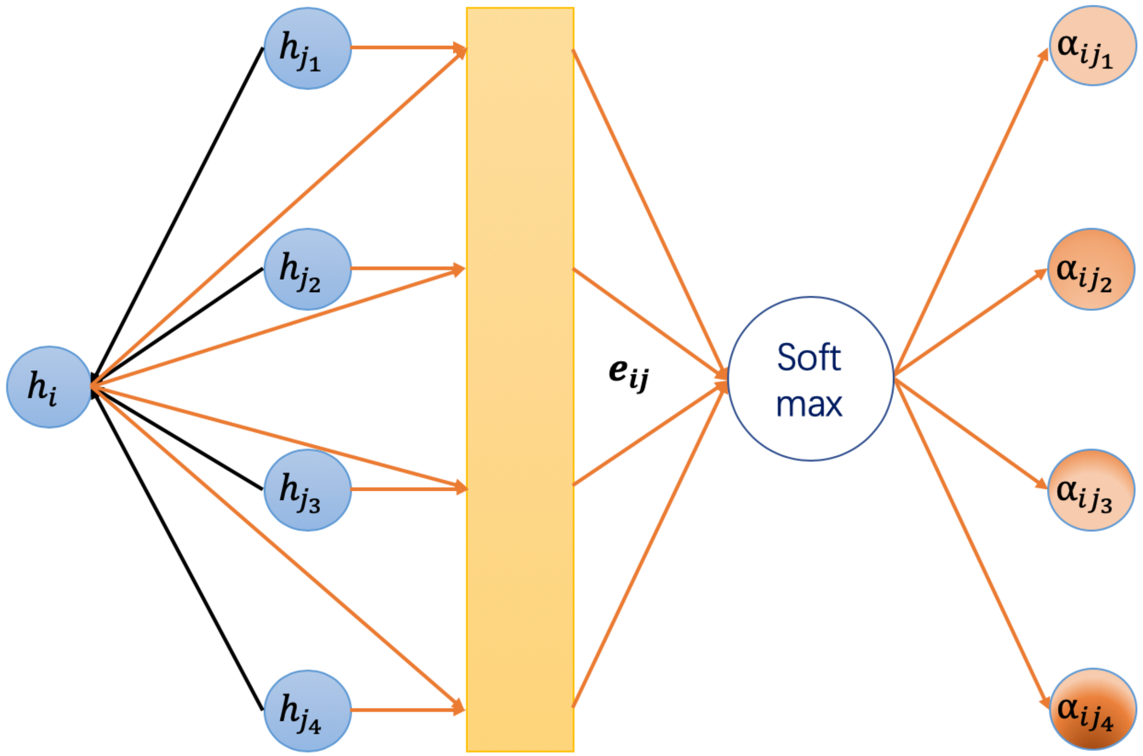
Attention mechanism enables GCN to update nodes with different importance.



$$H^{(l+1)} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij} H_j^{(l)} W^{(l)}\right)$$

Analysis of Graph Attention: Attention Heads

- In Graph Attention Networks (GATs), the concept of multiple attention heads is similar to the idea of multiple filters in Convolutional Neural Networks (CNNs).
- Each attention head can potentially learn to pay attention to different types of neighborhood information.



Analysis of Graph Attention: Computation Steps (1)

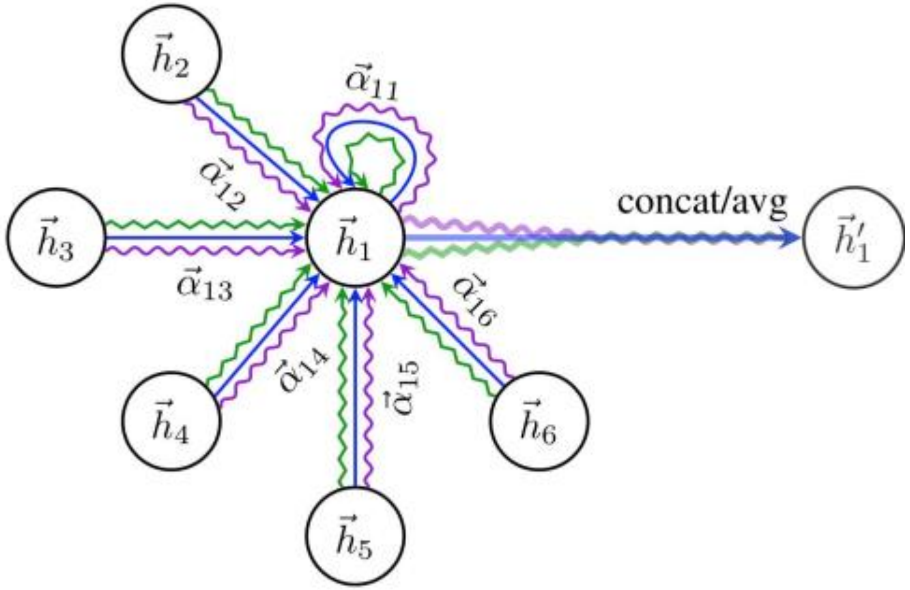
- Input node features: Each node in the graph has a feature vector.

$$\mathbf{h} = \{\vec{h}_1, \vec{h}_2, \ldots, \vec{h}_N\}, \vec{h}_i \in \mathbb{R}^F$$

- Calculate energy (co-efficient) between two nodes

$$e_{ij} = a(\mathbf{W}\vec{h}_i, \mathbf{W}\vec{h}_j)$$

a: attention function



Analysis of Graph Attention: Computation Steps (2)

- Attention score (over the neighbors): Normalize over all the neighbors

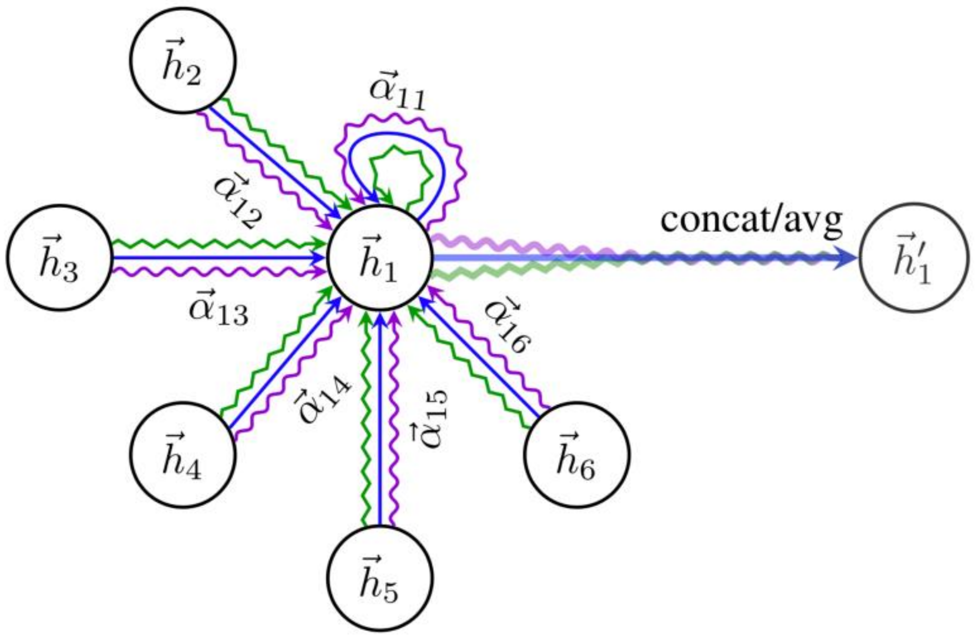
$$\alpha_{ij} = \frac{\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j]\right)\right)}{\sum_{k\in\mathcal{N}_i}\exp\left(\text{LeakyReLU}\left(\vec{\mathbf{a}}^T[\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k]\right)\right)}$$

- Multi-head attention
  - Feature concatenation

$$\vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right)$$

  - Feature averaging (for the final layer)

$$\vec{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j\right)$$

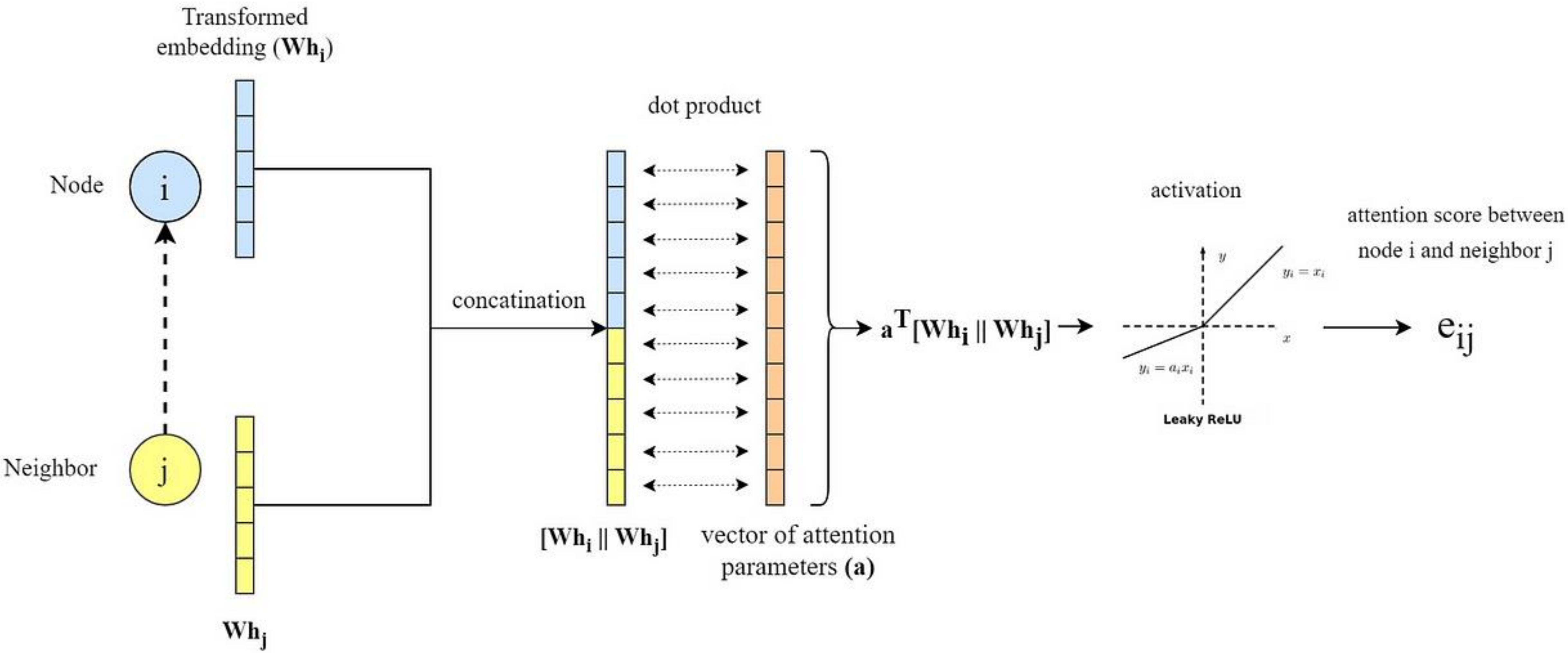


Analysis of Graph Attention

- Pros:
  - No need to score intermediate edge-based activation vectors (when using dot product attention).
  - Slower than GCNs but faster than GNNs with edge embeddings.
- Cons:
  - Can be more difficult to optimize.

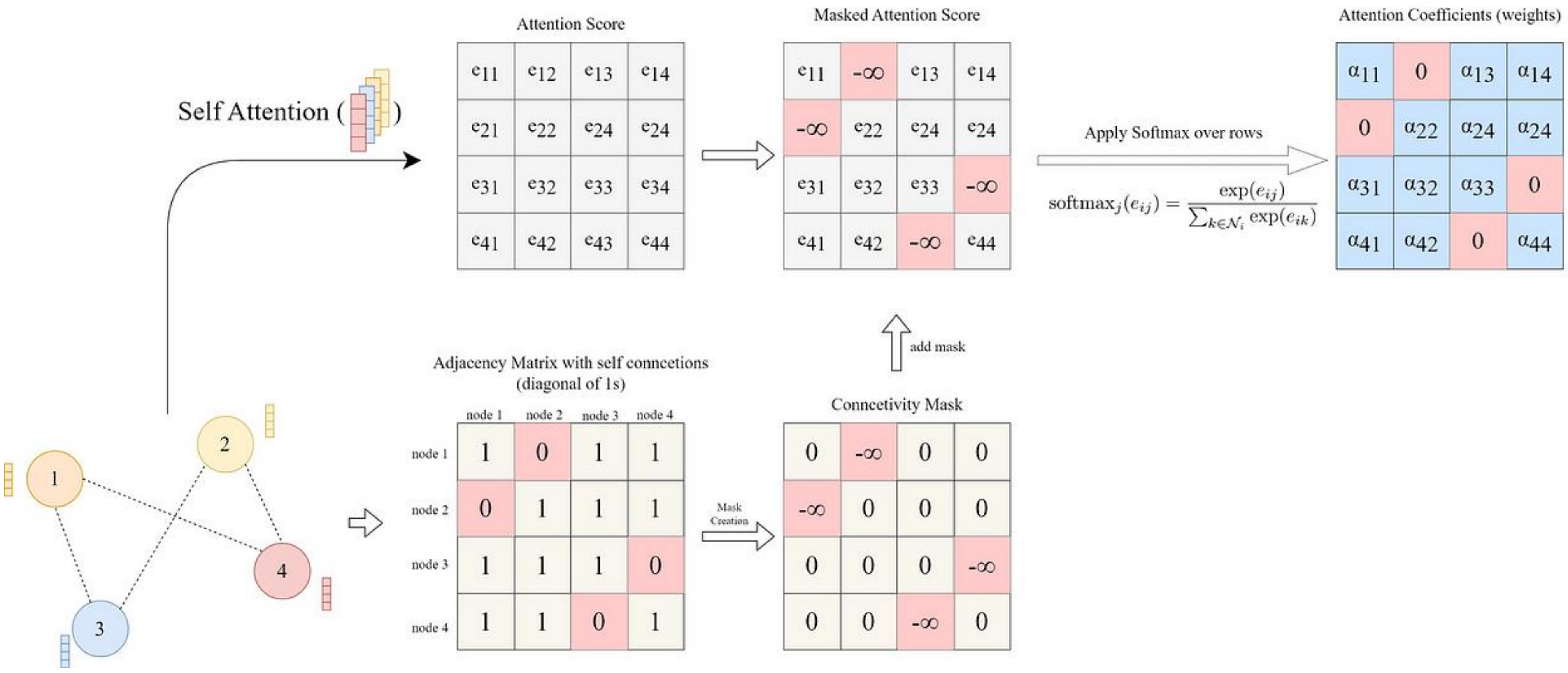
GAT (Graph Attention Networks) Summary

- The whole operation is illustrated below:

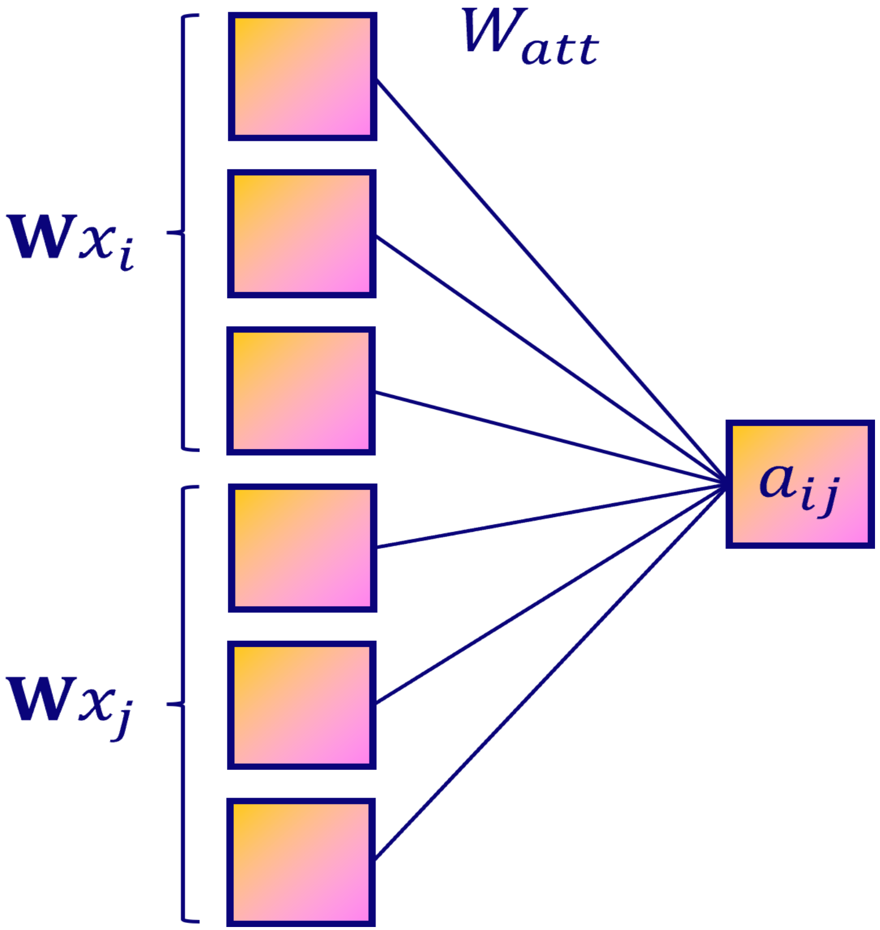


Graph Attention Networks: Example

- Applying masking mechanism to the masked attention score, then apply Softmax function:



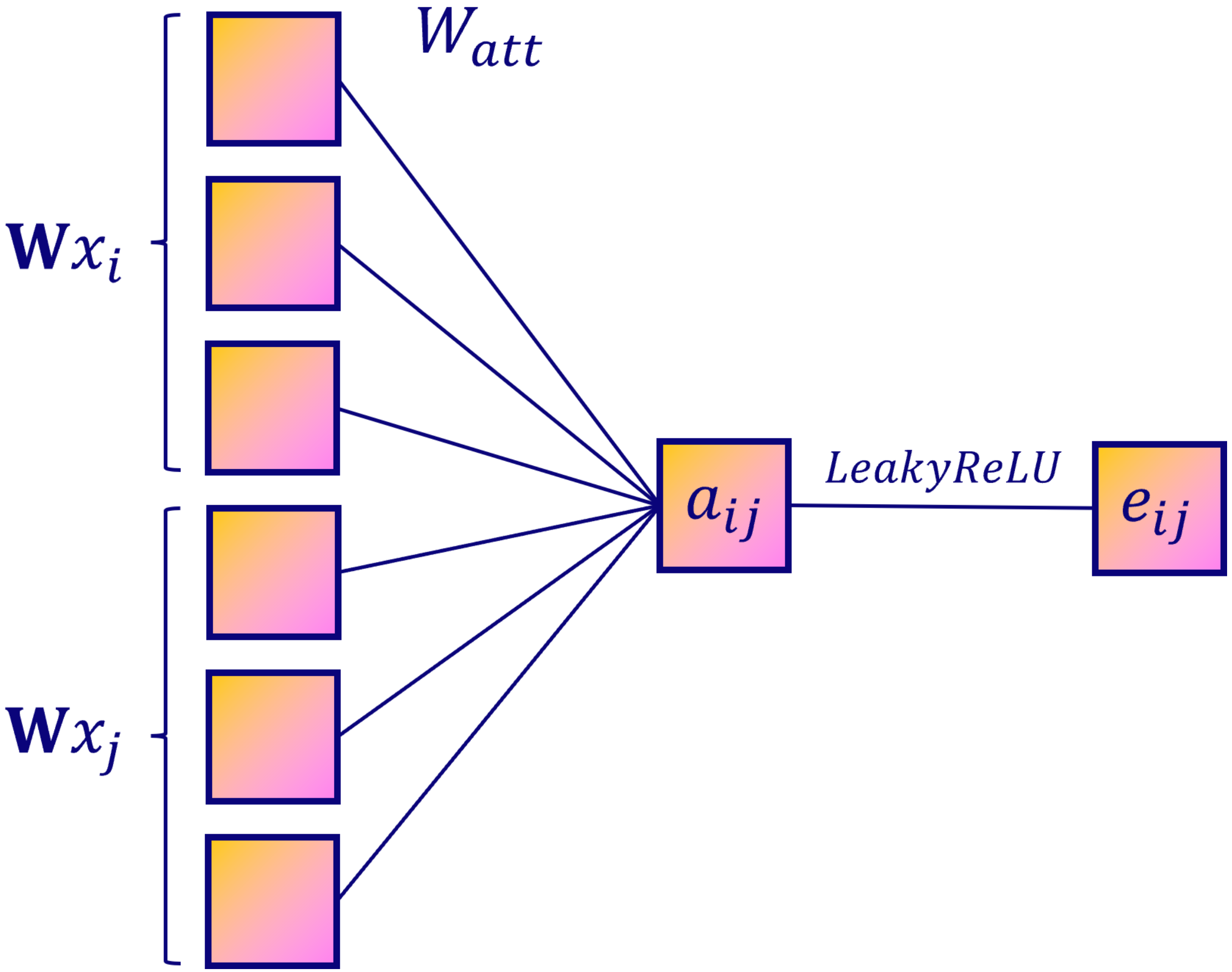
- **Linear transformation:** To calculate the attention coefficient, we need to consider pairs of nodes. An easy way to create these pairs is to concatenate attribute vectors from both nodes.



$$a_{ij} = W_{att}^t[\mathbf{W}x_i \parallel \mathbf{W}x_j]$$

Graph Attention Networks: Steps by steps

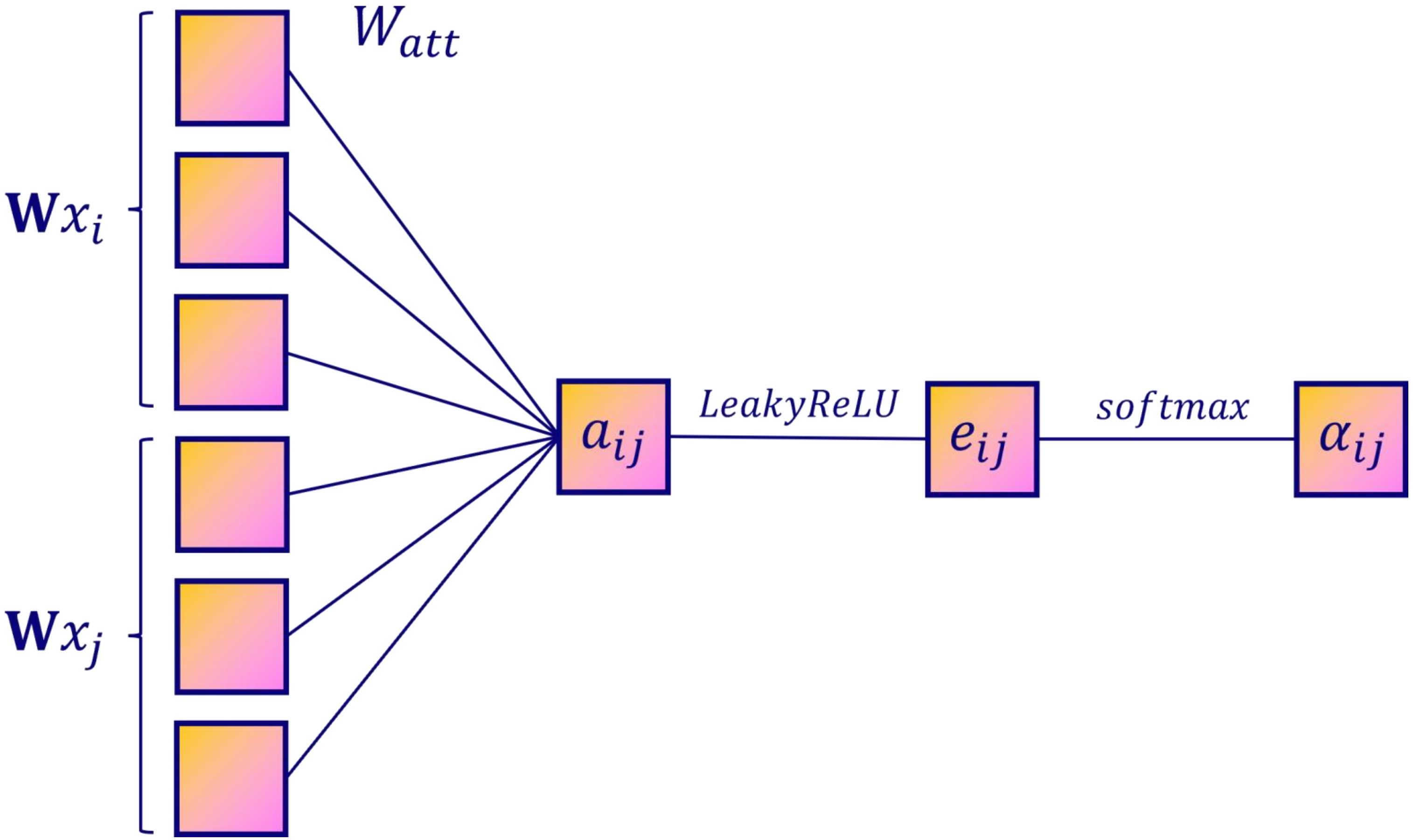
- **Activation function:** add nonlinearity with an activation function. In this case, the paper's authors chose the LeakyReLU function.



$$e_{ij} = \mathrm{LeakyReLU}(a_{ij})$$

Graph Attention Networks: Steps by steps

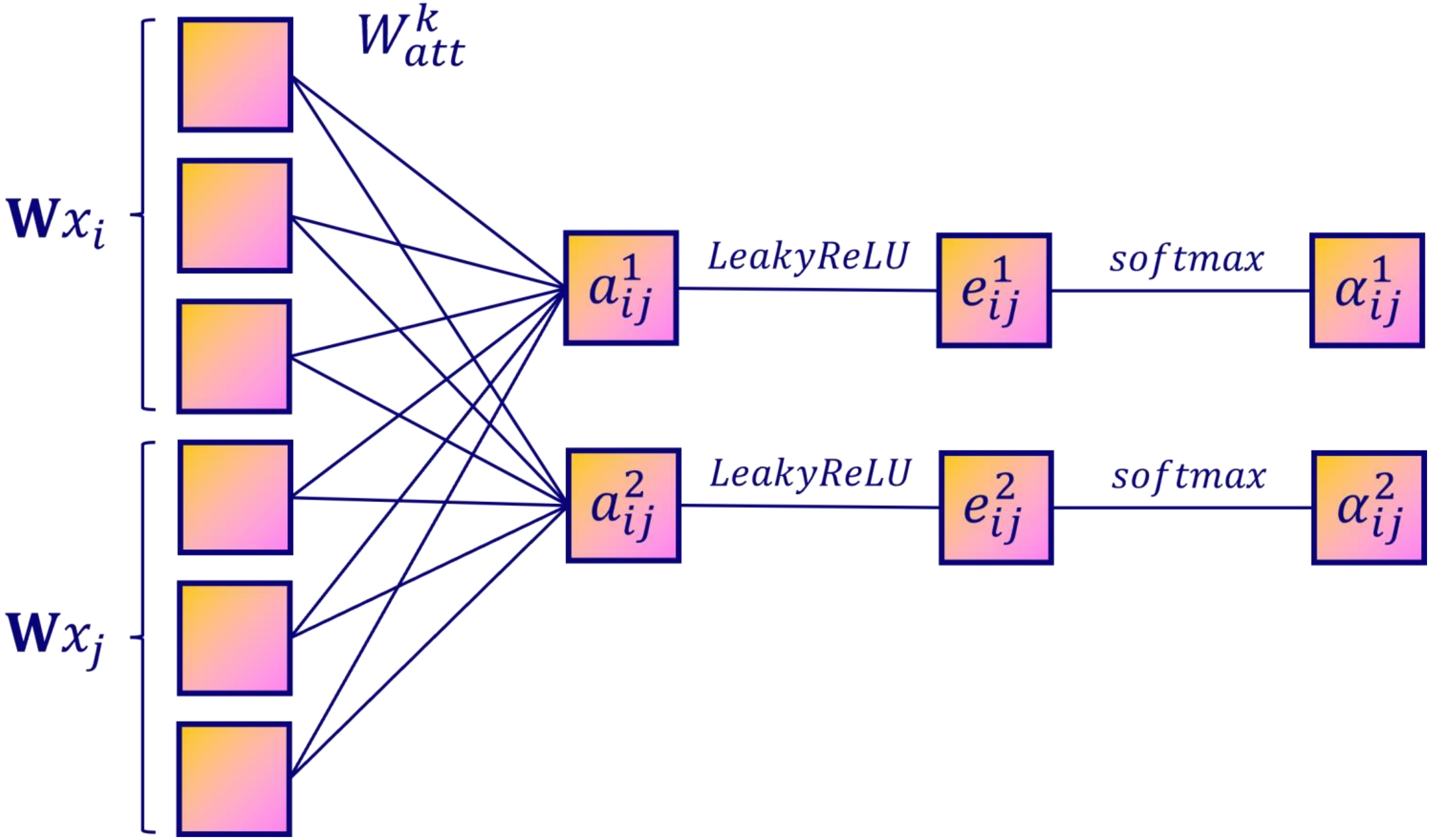
- Softmax normalization: The output of our neural network is not normalized, which is a problem since we want to compare these coefficients.
- A common way to do it with neural networks is to use the softmax function.



$$\alpha_{ij} = \mathrm{softmax}_j(e_{ij}) = \frac{exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} exp(e_{ik})}$$

Graph Attention Networks: Steps by steps

- Multi-head attention: In GATs, multi-head attention consists of replicating the same three steps several times in order to average or concatenate the results.



Average: $$h_i = \frac{1}{n}\sum_{k=1}^{n} h_i^k$$

Concatenation: $$h_i = \|_{k=1}^{n} h_i^k$$

GAT in Torch Geometric

/ torch_geometric.nn / conv.GATConv

conv.GATConv

class `GATConv(in_channels: Union[int, Tuple[int, int]], out_channels: int, heads: int = 1, concat: bool = True, negative_slope: float = 0.2, dropout: float = 0.0, add_self_loops: bool = True, edge_dim: Optional[int] = None, fill_value: Union[float, Tensor, str] = 'mean', bias: bool = True, **kwargs)`

[source]

Bases: `MessagePassing`

The graph attentional operator from the "Graph Attention Networks" paper

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j,$$

where the attention coefficients $\alpha_{i,j}$ are computed as

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}.$$

GAT (Graph Attention Networks): Sample code


```
class GAT(torch.nn.Module):
        def __init__(self,num_features, num_classes, dims, drop=0.0):
                super(GAT, self).__init__()
                heads = 8
                self.conv1 = GATConv(num_features,dims, heads=heads, dropout=0.3, concat=False)
                # On the Pubmed dataset, use heads=8 in conv2.
                self.conv2 = GATConv(dims, num_classes, heads=heads, concat=False,
                                                         dropout=0.3)
                self.drop = torch.nn.Dropout(p=drop)
        def forward(self,x, edge_index):
                x = F.elu(self.conv1(x, edge_index))
                x = self.drop(x)
                x = self.conv2(x, edge_index)
                return F.log_softmax(x, dim=1), x
```

- Let's try some simple GAT code in the sample code file

Graph Attention Networks v2 (GATv2)

- GATv2s is similar to GAT.
- The GATv2 operator fixes the static attention problem of the standard GAT.
  - Static attention is when the attention to the key nodes has the same rank (order) for any query node.
  - GAT computes attention from query node i to key node j:

$$\begin{aligned} e_{ij} &= \mathrm{LeakyReLU}\left(\mathbf{a}^\top \left[\mathbf{W}\vec{h_i} \| \mathbf{W}\vec{h_j}\right]\right) \\ &= \mathrm{LeakyReLU}\left(\mathbf{a}_1^\top \mathbf{W}\vec{h_i} + \mathbf{a}_2^\top \mathbf{W}\vec{h_j}\right) \end{aligned}$$

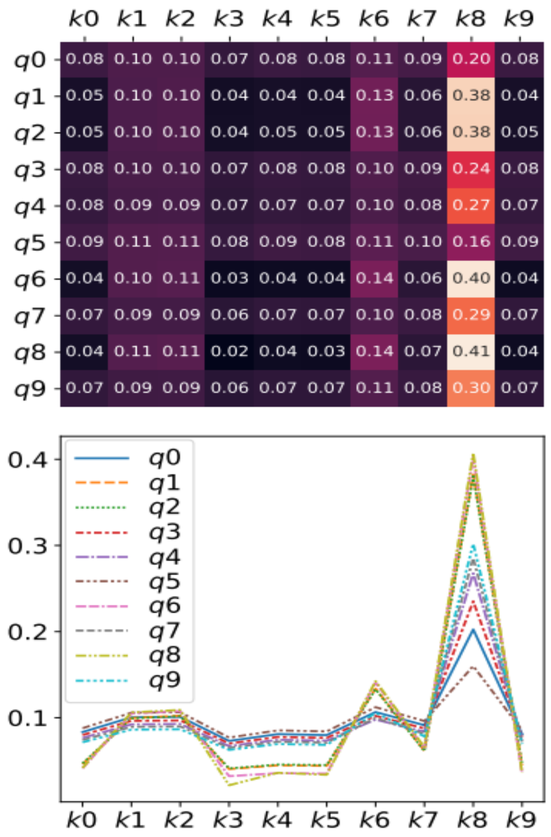
GAT

$$\begin{aligned} e_{ij} &= \mathbf{a}^\top \mathrm{LeakyReLU}\left(\mathbf{W}\left[\vec{h_i} \| \vec{h_j}\right]\right) \\ &= \mathbf{a}^\top \mathrm{LeakyReLU}\left(\mathbf{W}_l\vec{h_i} + \mathbf{W}_r\vec{h_j}\right) \end{aligned}$$

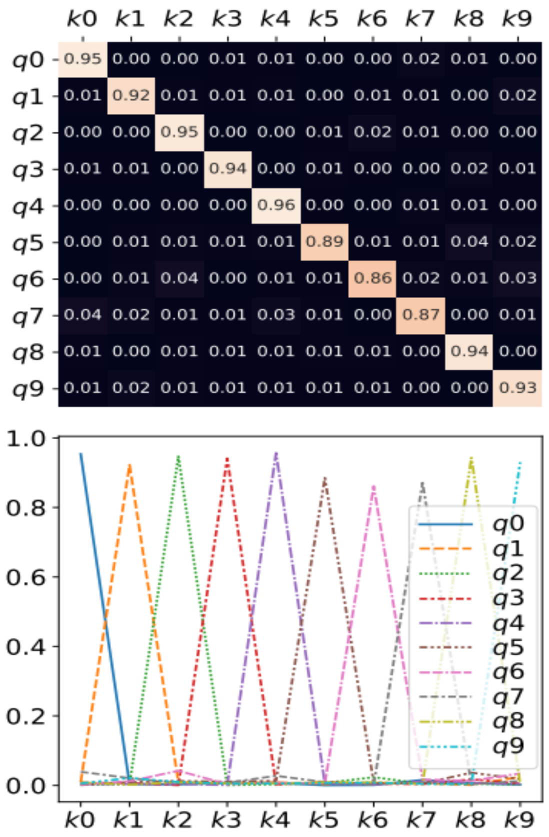
GATv2

Graph Attention Networks v2 (GATv2)

➢ The GATv2 model performs better than the first version GAT, because it uses a dynamic graph attention variant that has a universal approximator attention function, it is more expressive than the other model, based on a static attention



Attention in standard GAT



Attention in GATv2

Brody, Shaked, Uri Alon, and Eran Yahav. "How attentive are graph attention networks?." *ICLR* (2021).

GATv2 in Torch Geometric

- GATv2 is available as part of PyTorch Geometric library

```
from torch_geometric.nn import GATv2Conv
```

/ torch_geometric.nn / conv.GATv2Conv

conv.GATv2Conv

class `GATv2Conv` (**in_channels**: Union[int, Tuple[int, int]], **out_channels**: int, **heads**: int = 1, **concat**: bool = True, **negative_slope**: float = 0.2, **dropout**: float = 0.0, **add_self_loops**: bool = True, **edge_dim**: Optional[int] = None, **fill_value**: Union[float, Tensor, str] = 'mean', **bias**: bool = True, **share_weights**: bool = False, ****kwargs**) [source]

Bases: `MessagePassing`

The GATv2 operator from the "How Attentive are Graph Attention Networks?" paper, which fixes the static attention problem of the standard `GATConv` layer. Since the linear layers in the standard GAT are applied right after each other, the ranking of attended nodes is unconditioned on the query node. In contrast, in `GATv2`, every node can attend to any other node.

GATv2 in Torch Geometric: Sample code

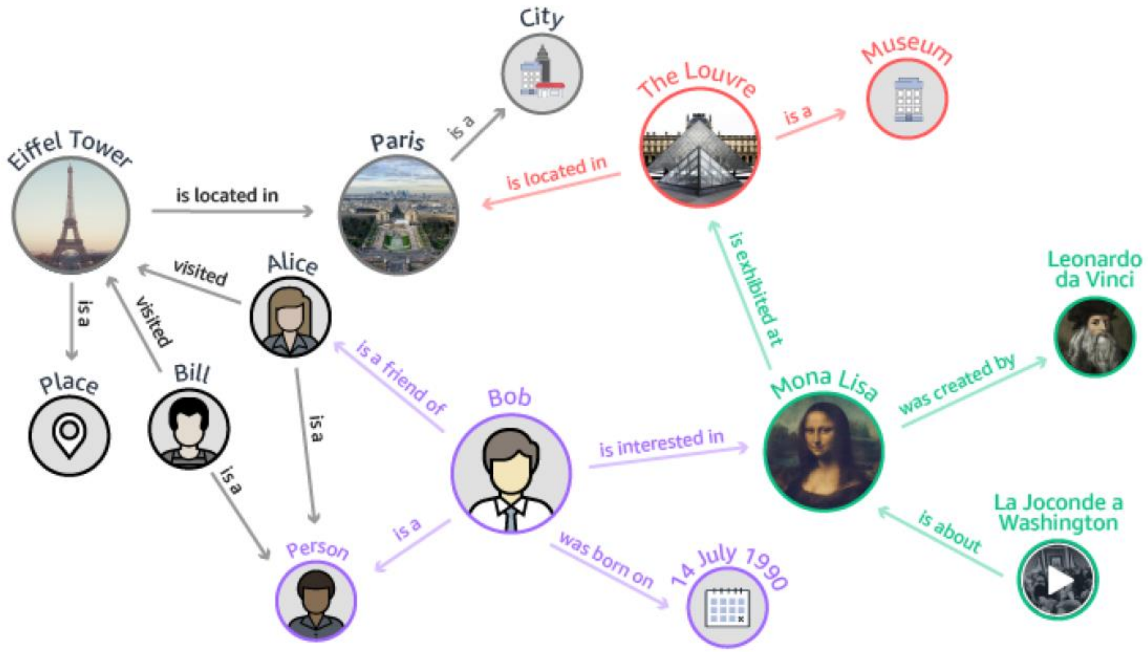
- GATv2 is available as part of PyTorch Geometric library

```
from torch_geometric.nn import GATv2Conv
```

```python
class GATv2(torch.nn.Module):
    def __init__(self,num_features, num_classes, dims, drop=0.0):
        super(GATv2, self).__init__()
        heads = 8
        self.h = None
        self.conv1 = GATv2Conv(num_features,dims, heads=heads, dropout = 0.3, concat=False)
        self.conv2 = GATv2Conv(dims, num_classes, heads=heads, concat=False, dropout=0.3)
        self.drop = torch.nn.Dropout(p=drop)
    def forward(self, x, edge_index,  g, Kindices):
        x = F.elu(self.conv1(x, edge_index))
        x = self.drop(x)
        x = self.conv2(x, edge_index)
        self.h = x
        return F.log_softmax(x, dim=1)
```

Attention in Heterogeneours Graphs

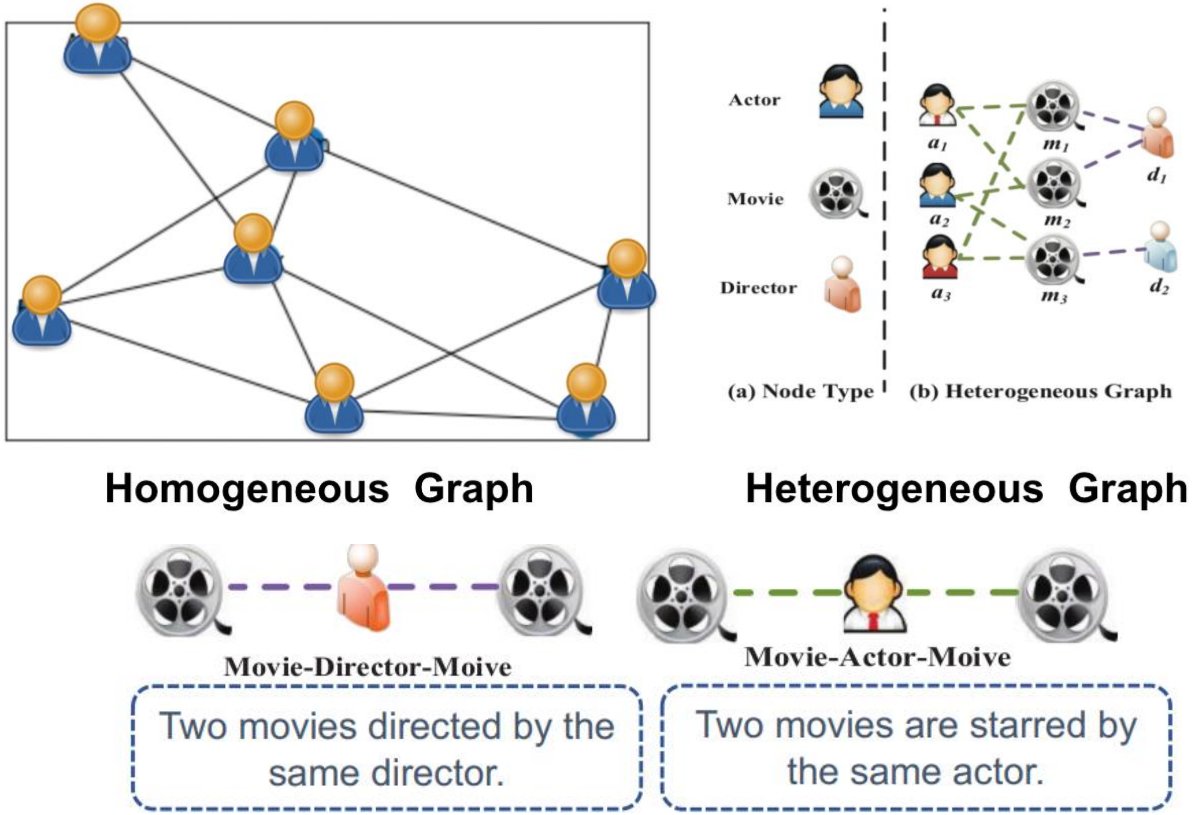
- Graph in real world:
  - Many node types, link types
  - Non- ordered
  - Complex connections



Attention in Heterogeneours Graphs

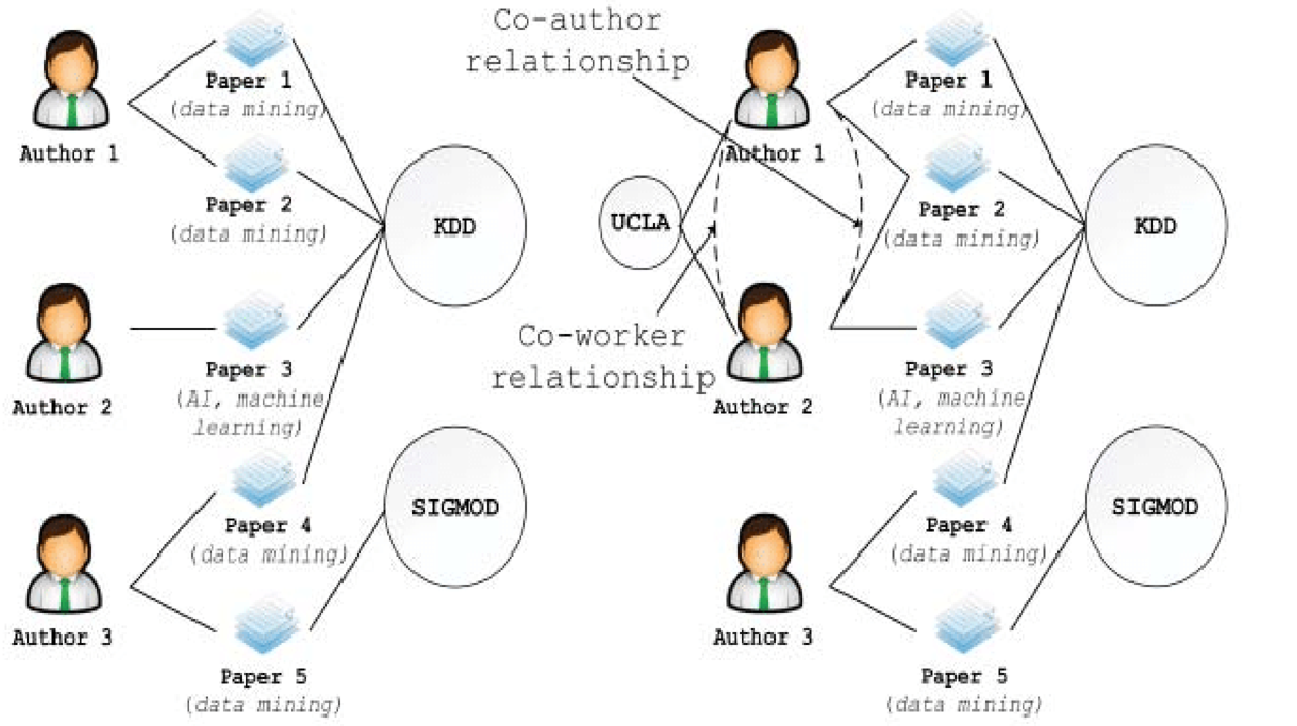
- Multiple types of nodes or links
- Rich semantic information
  - Meta-path: a relation sequence connecting objects

    (e.g., Movie-Actor-Movie).



Meta-paths: High level representation of relationship

- DBLP Bibliographic network
  - Node (type)
    - KDD (Venue)
    - Author 1
  - Link (Type)
    - Write (Author - Paper)
    - Publish (Paper – Venue)

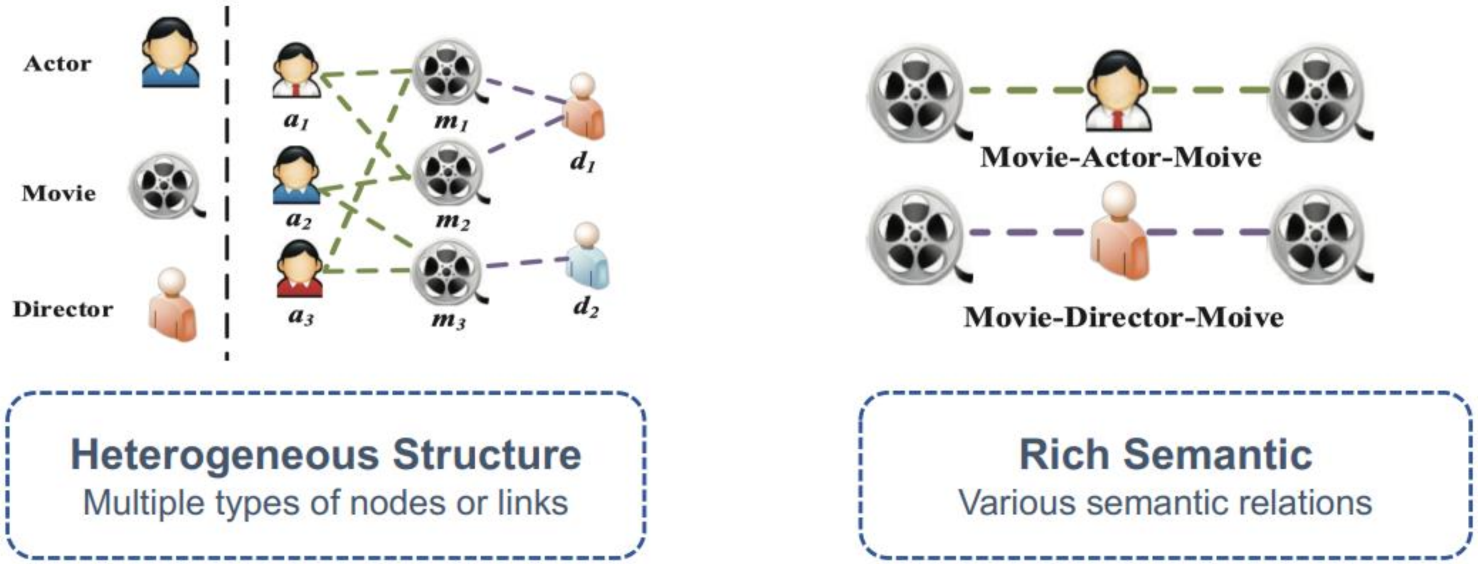


A. Examples of A-P-V-P-A meta-path on DBLP

B. Examples of common neighborhood objects between two authors in DBLP

Why Attention in Heterogeneours Graphs

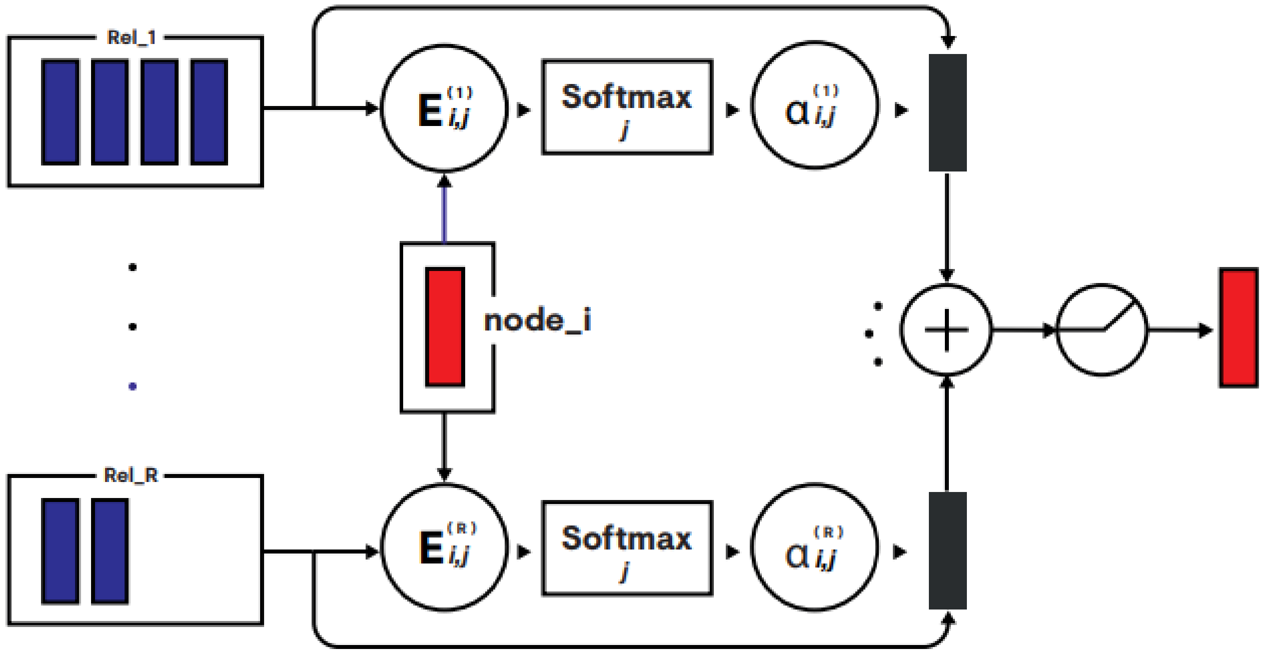
- Existing GNNs focus on homogeneous graphs
  - Cannot handle multiple types of nodes and edges.
  - Cannot capture rich semantic information.



Challenge: How to handle the heterogeneity of graph?

Relational Graph Attention Networks (RGATs)

➢ The objective: Extending attention mechanisms to the relational graph domain



The representations for node i

A target node i have different relations : $Rel_1, Rel_2, \ldots, Rel_R$

The logits $E_{i,j}^{(r)}$ of each relation r: $E_{i,j}^{(r)} = a\left(\boldsymbol{g}_i^{(r)}, \boldsymbol{g}_j^{(r)}\right),$

where: $\boldsymbol{G}^{(r)} = \boldsymbol{H}\,\boldsymbol{W}^{(r)} \in \mathbb{R}^{N \times F'},$ the representation feature matrix under relation r

Busbridge, D., Sherburn, D., Cavallo, P., & Hammerla, N. Y. (2019). Relational graph attention networks. *ICLR 2019*

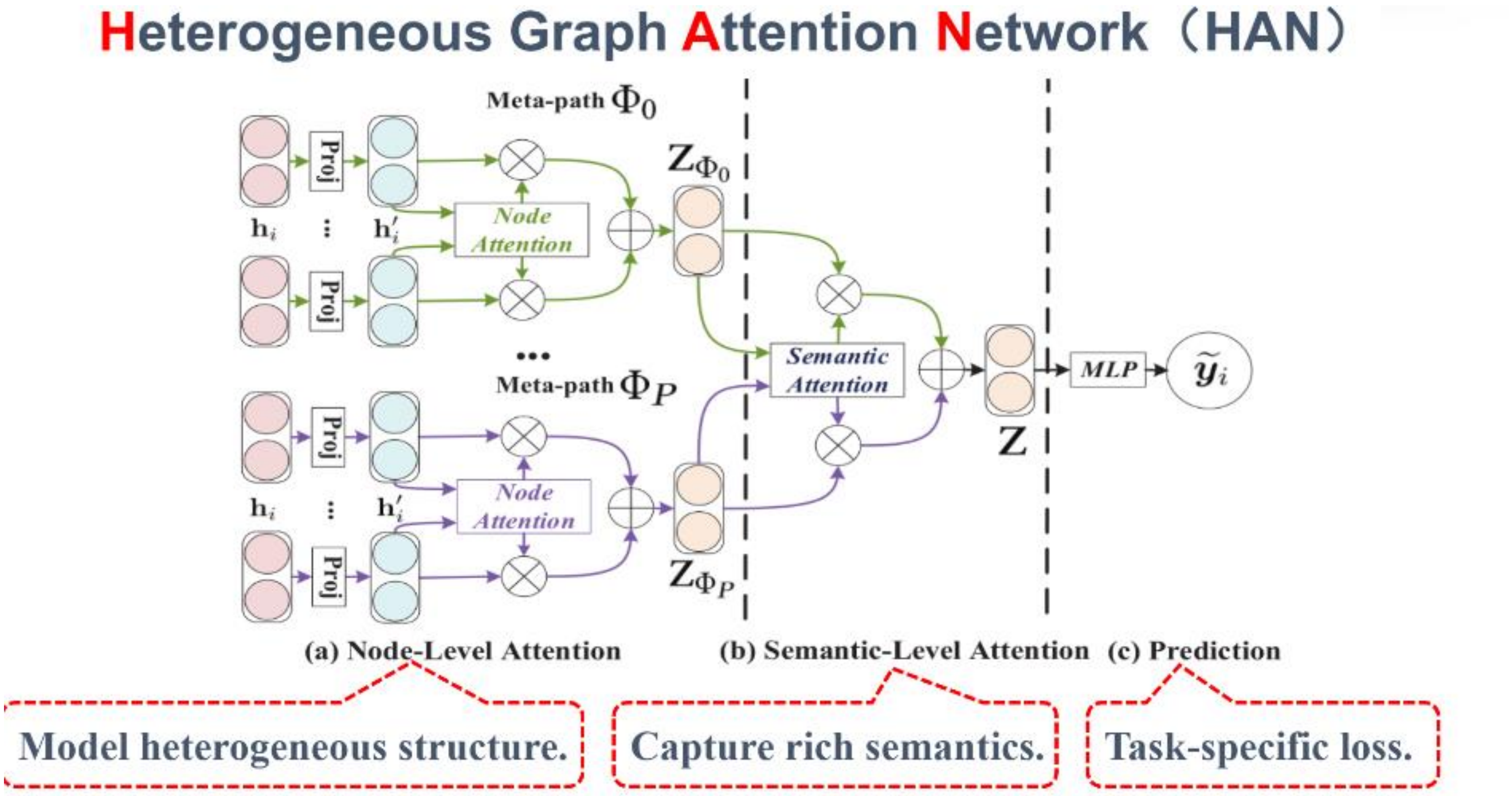
RGATs in Torch Geometric

- RGAT is available as part of PyTorch Geometric library

```
from torch_geometric.nn import RGATConv
```

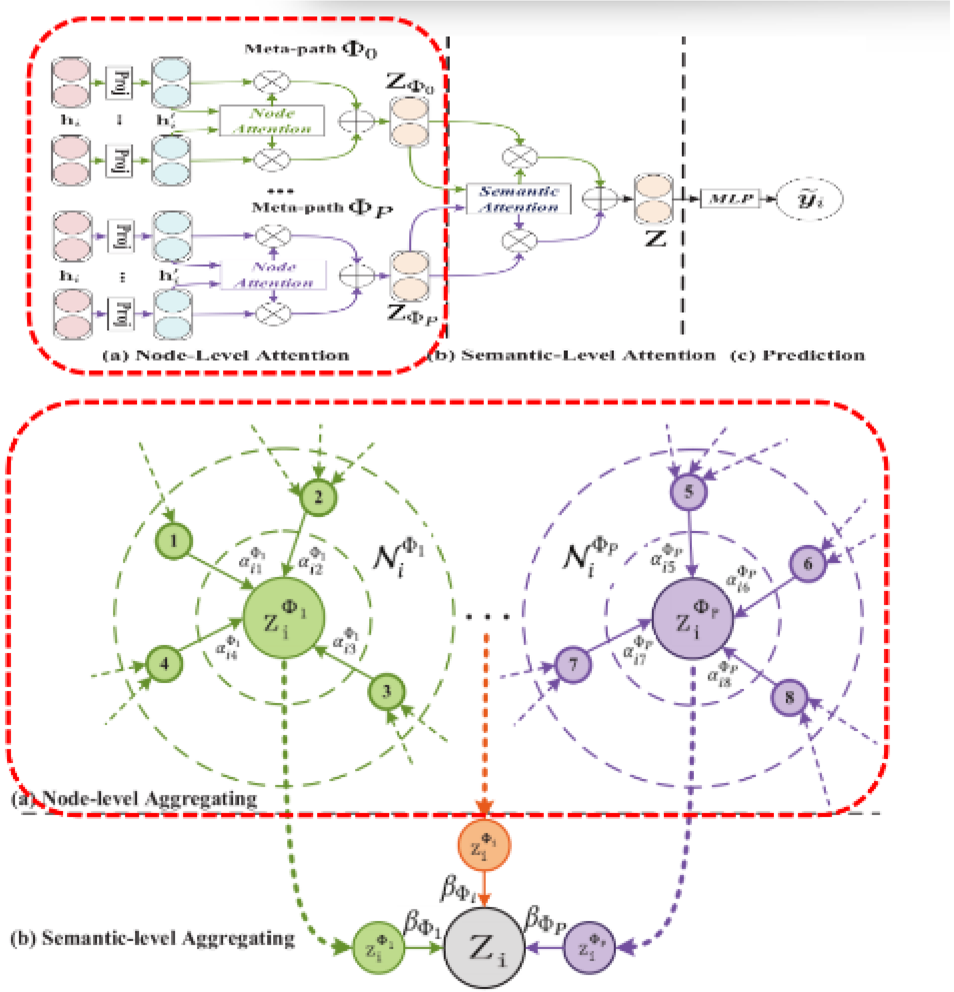
```
class RGAT(torch.nn.Module):
    def __init__(self, in_channels, hidden_channels, out_channels,
                 num_relations):
        super().__init__()
        self.conv1 = RGATConv(in_channels, hidden_channels, num_relations)
        self.conv2 = RGATConv(hidden_channels, hidden_channels, num_relations)
        self.lin = torch.nn.Linear(hidden_channels, out_channels)

    def forward(self, x, edge_index, edge_type):
        x = self.conv1(x, edge_index, edge_type).relu()
        x = self.conv2(x, edge_index, edge_type).relu()
        x = self.lin(x)
        return F.log_softmax(x, dim=-1)
```

Heterogeneous Graph Attention Networks (HAN)



Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., & Yu, P. S. (2019, May). Heterogeneous graph attention network. In *The world wide web conference*

➢ Node-level Attention and Aggregating



➢ Type-specific information

$$\mathbf{h}'_i = \mathbf{M}_{\phi_i} \cdot \mathbf{h}_i,$$

Type-specific transformation matrix

➢ Importance of Neighbors

$$e_{ij}^{\Phi} = att_{node}(\mathbf{h}'_i, \mathbf{h}'_j; \Phi).$$

$$\alpha_{ij}^{\Phi} = softmax_j(e_{ij}^{\Phi}) = \frac{\exp\left(\sigma(\mathbf{a}_{\Phi}^{\mathrm{T}} \cdot [\mathbf{h}'_i \| \mathbf{h}'_j])\right)}{\sum_{k \in \mathcal{N}_i^{\Phi}} \exp\left(\sigma(\mathbf{a}_{\Phi}^{\mathrm{T}} \cdot [\mathbf{h}'_i \| \mathbf{h}'_k])\right)},$$

Node-level attention vector

➢ Node-level Aggregating

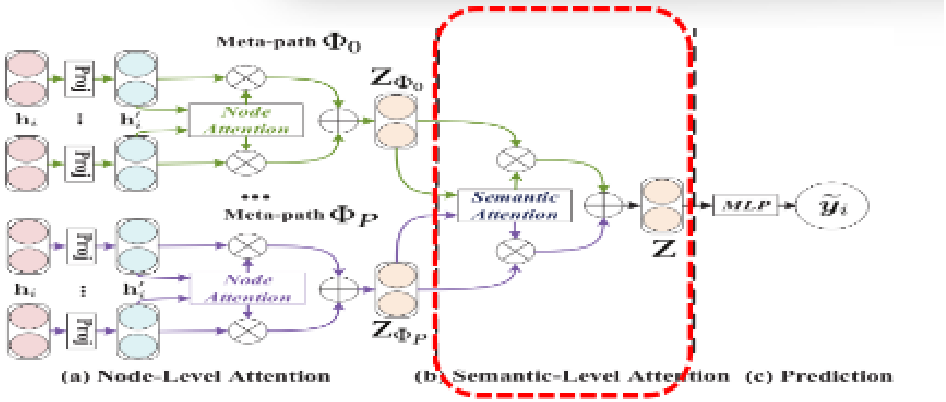
$$\mathbf{z}_i^{\Phi} = \Big\Vert_{k=1}^{K} \sigma\left(\sum_{j \in \mathcal{N}_i^{\Phi}} \alpha_{ij}^{\Phi} \cdot \mathbf{h}'_j \right).$$

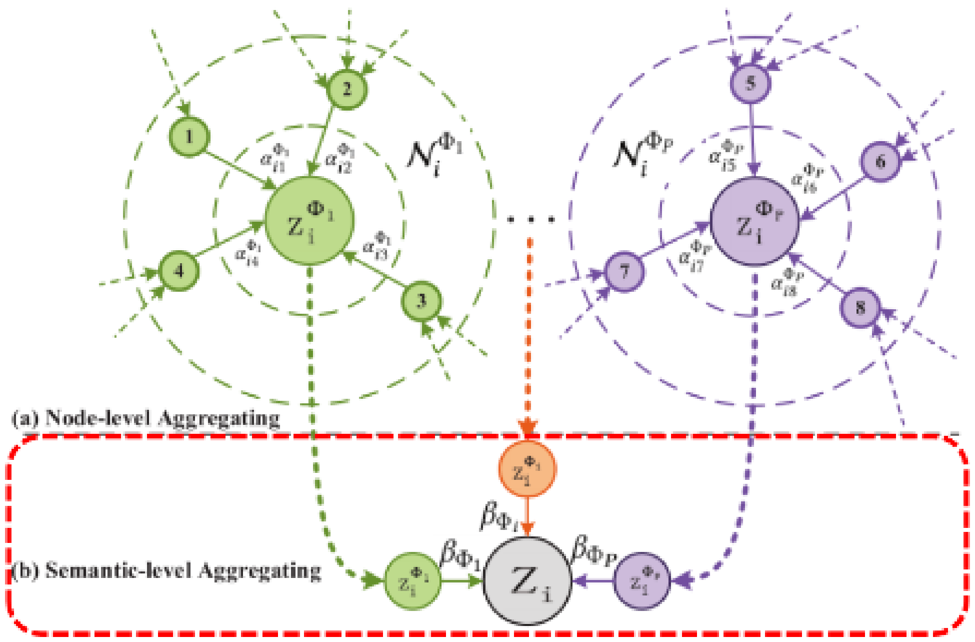
Node weight

Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., & Yu, P. S. (2019, May). Heterogeneous graph attention network. In *The world wide web conference*

Heterogeneous Graph Attention Networks (HAN)

➢ Semantic-level Attention and Aggregating





➢ Semantic-Level Attention

$$(\beta_{\Phi_0}, \beta_{\Phi_1}, \ldots, \beta_{\Phi_P}) = att_{sem}(\mathrm{Z}_{\Phi_0}, \mathrm{Z}_{\Phi_1}, \ldots, \mathrm{Z}_{\Phi_P})$$

➢ Importance of Meta-path

Semantic-level attention vector

$$w_{\Phi_i} = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} \mathbf{q}^{\mathrm{T}} \cdot \tanh(\mathbf{W} \cdot \mathbf{z}_i^{\Phi} + \mathbf{b})$$

$$\beta_{\Phi_i} = \frac{\exp(w_{\Phi_i})}{\sum_{i=1}^{P} \exp(w_{\Phi_i})}$$

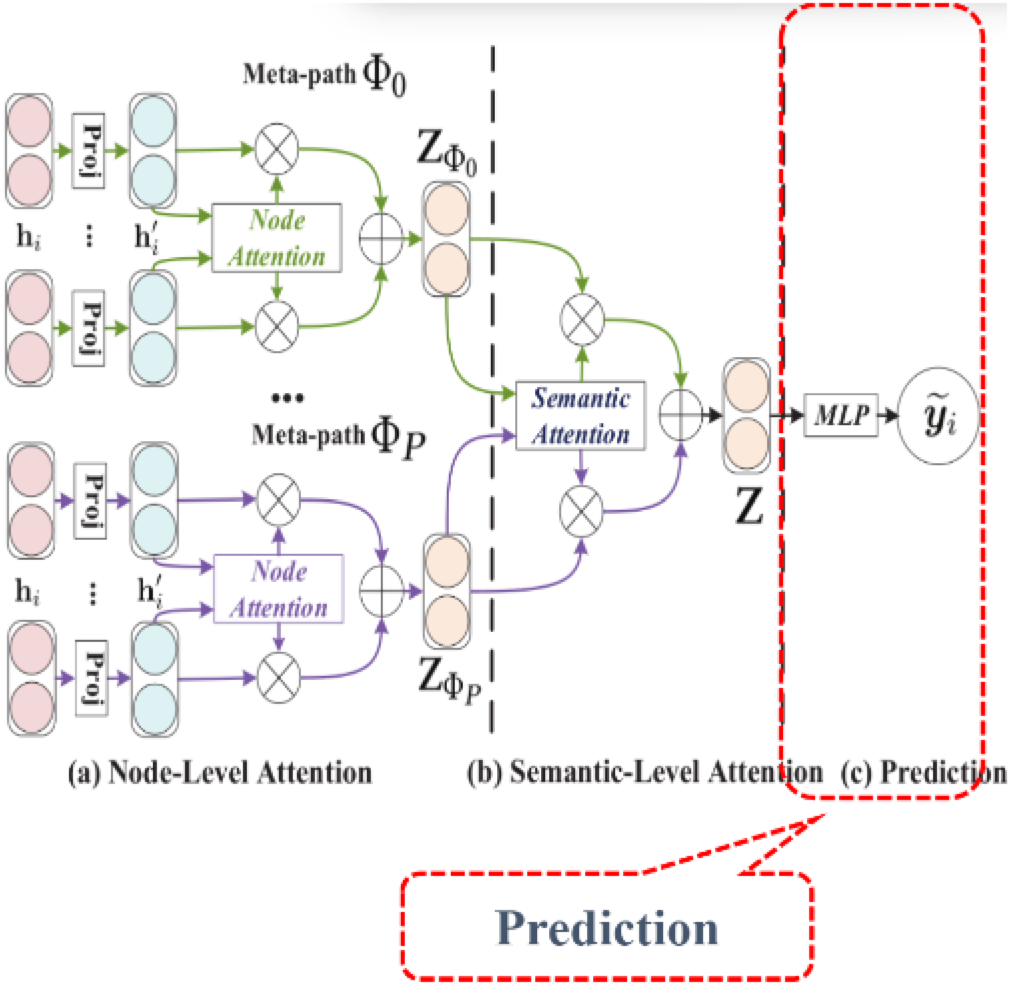
➢ Semantic-Level Aggregating

$$\mathbf{Z} = \sum_{i=1}^{P} \beta_{\Phi_i} \cdot \mathbf{Z}_{\Phi_i}$$

Semantic weight

Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., & Yu, P. S. (2019, May). Heterogeneous graph attention network. In *The world wide web conference*

➢ Prediction



➢ Semi-supervised Loss

Parameter of classifier

$$L = -\sum_{l \in \mathcal{Y}_L} \mathrm{Y}^l \ln(\mathrm{C} \cdot \mathrm{Z}^l)$$

Labeled data

Optimize for the specific task
(e.g. node classification)

Wang, X., Ji, H., Shi, C., Wang, B., Ye, Y., Cui, P., & Yu, P. S. (2019, May). Heterogeneous graph attention network. In *The world wide web conference*

HAN in Torch Geometric

- HAN is available as part of PyTorch Geometric library

```
from torch_geometric.nn import HANConv
```

```
class HAN(torch.nn.Module):
    def __init__(self, in_channels: Union[int, Dict[str, int]],
                 out_channels: int, hidden_channels=128, heads=8):
        super().__init__()
        self.han_conv = HANConv(in_channels, hidden_channels, heads=heads,
                                dropout=0.6, metadata=data.metadata())
        self.lin = Linear(hidden_channels, out_channels)

    def forward(self, x_dict, edge_index_dict):
        out = self.han_conv(x_dict, edge_index_dict)
        out = self.lin(out['movie'])
        return out
```

네트워크 과학 연구실 NETWORK SCIENCE LAB

가톨릭대학교 THE CATHOLIC UNIVERSITY OF KOREA