

# Graph Network

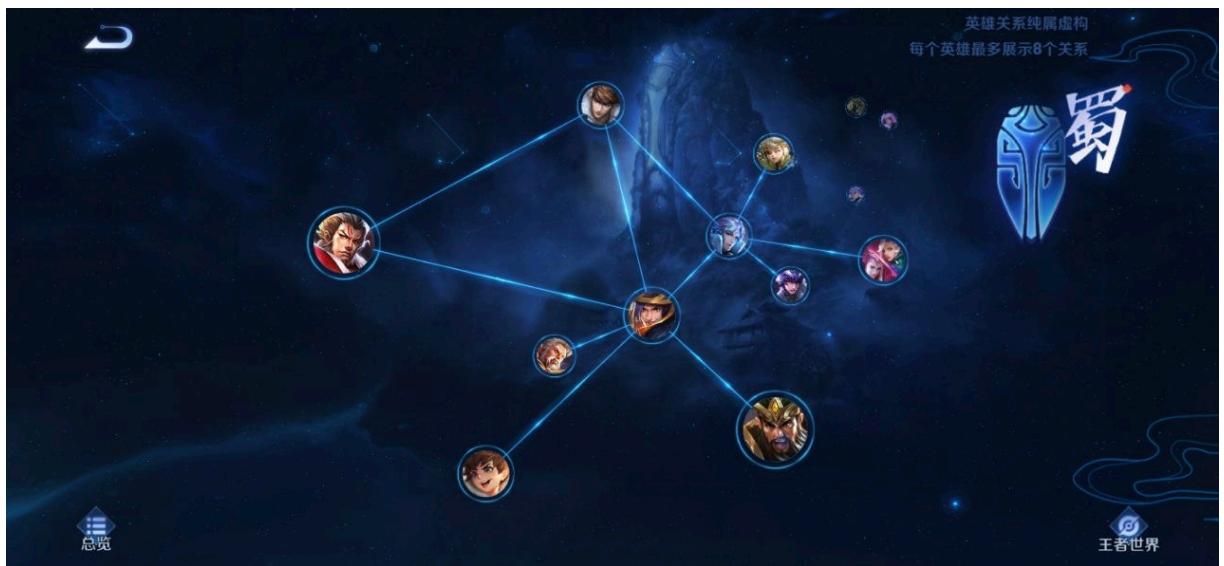
# 目录

- 1. Graph基本介绍
- 2. Graph Embedding
- 3. Graph Network
- 4. Complex Graph Network
- 5. Graph应用

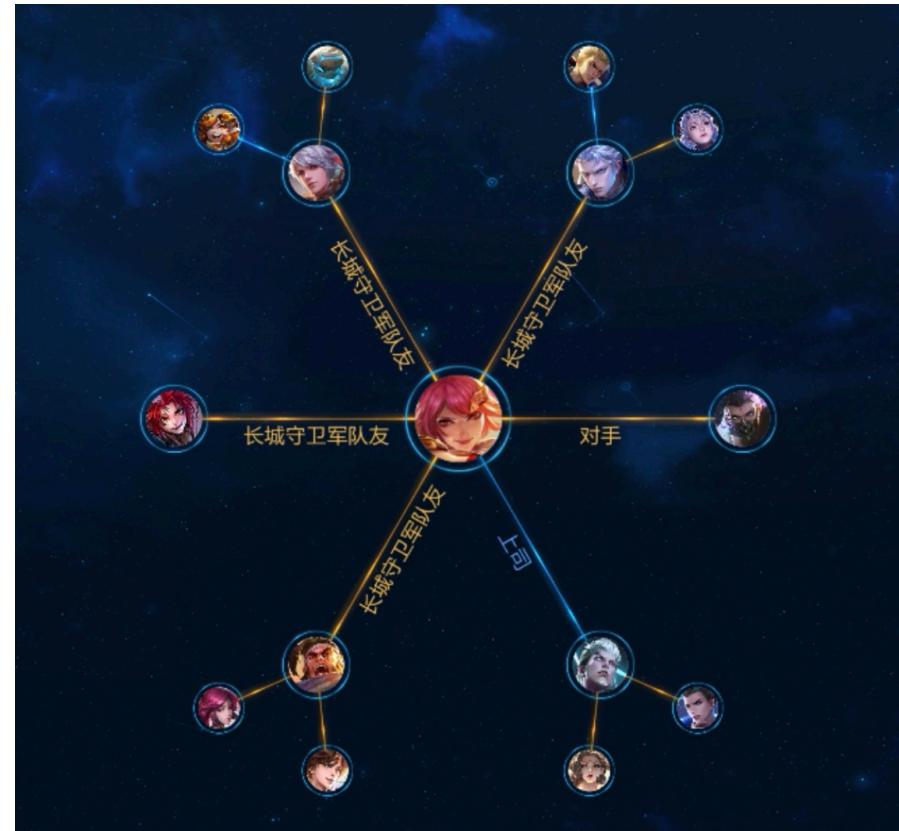
# 1. Graph基本介绍

- 1.1 图的表示
- 1.2 图的特性
  - 子图 Subgraph
  - 连通分量 Connected Component
  - 接通图 Connected Graph
  - 最短路径 Shortest Path
  - 图直径 Diameter
- 1.3 图中心性. Centrality
  - 度中心性 Degree Centrality
  - 特征向量中心性 Eigenvector Centrality
  - 中介中心性 Betweenness Centrality
  - 连接中心性 Closeness Centrality
- 1.4 网页排序算法
  - PageRank
  - HITS

# 1. 什么是图？



## 2. 图的表示？



如何去表示一张图？



	蒙恬	嬴政	芈月	鲁班	镜	白起	扁鹊
蒙恬		1					
嬴政	1		1		1	1	
芈月		1				1	
鲁班							
镜		1					
白起		1	1			1	
扁鹊						1	

邻接矩阵  
→  
Adjacency

0 1 0 0 0 0 0  
1 0 1 0 1 1 0  
0 1 0 0 0 1 0  
0 0 0 0 0 0 0  
0 1 0 0 0 0 0  
0 1 1 0 0 0 1  
0 0 0 0 0 1 0

图的性质：度 (degree)

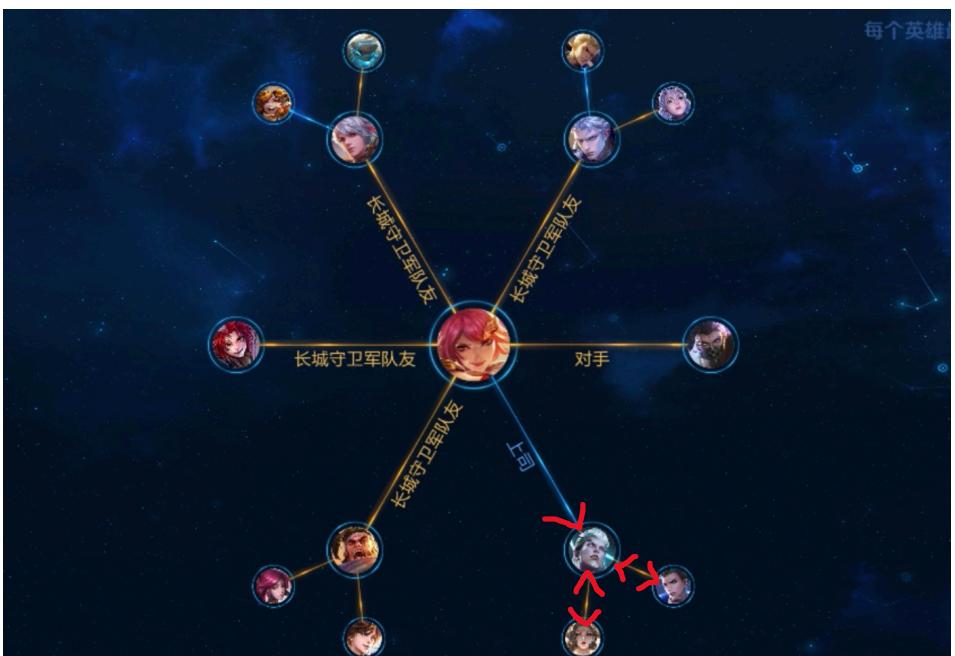


无向图的度：

嬴政 : 4

白起 : 3

鲁班 : 0



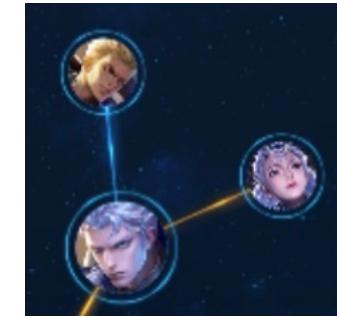
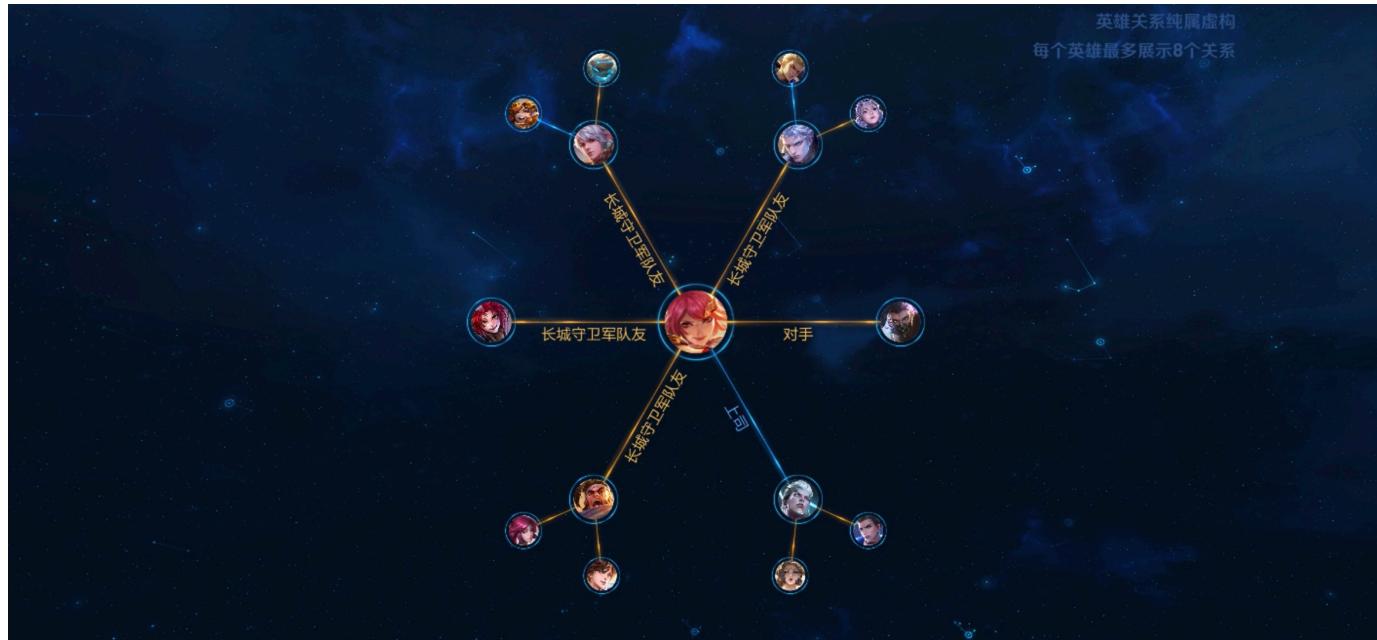
有向图的度：

司空震

出度 : 2

入度 : 3

# 子图 subgraph



subgraph

# 连通图，连通分量

1. 对于一个无向图，如果任意的节点 $i$ 能够通过一些边到达节点 $j$ ，则称之为连通图



连通图



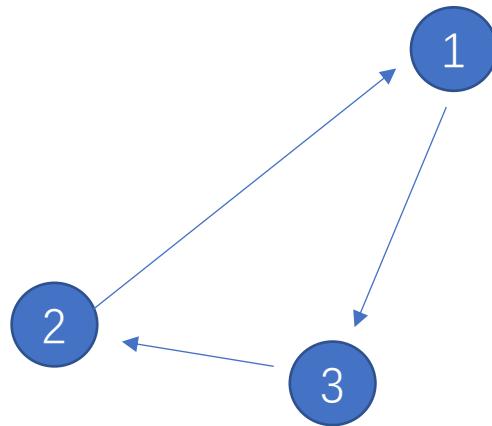
非连通图

**连通分量：**无向图  $G$  的一个极大连通子图称为  $G$  的一个连通分量（或连通分支）。连通图只有一个连通分量，即其自身；非连通的无向图有多个连通分量。

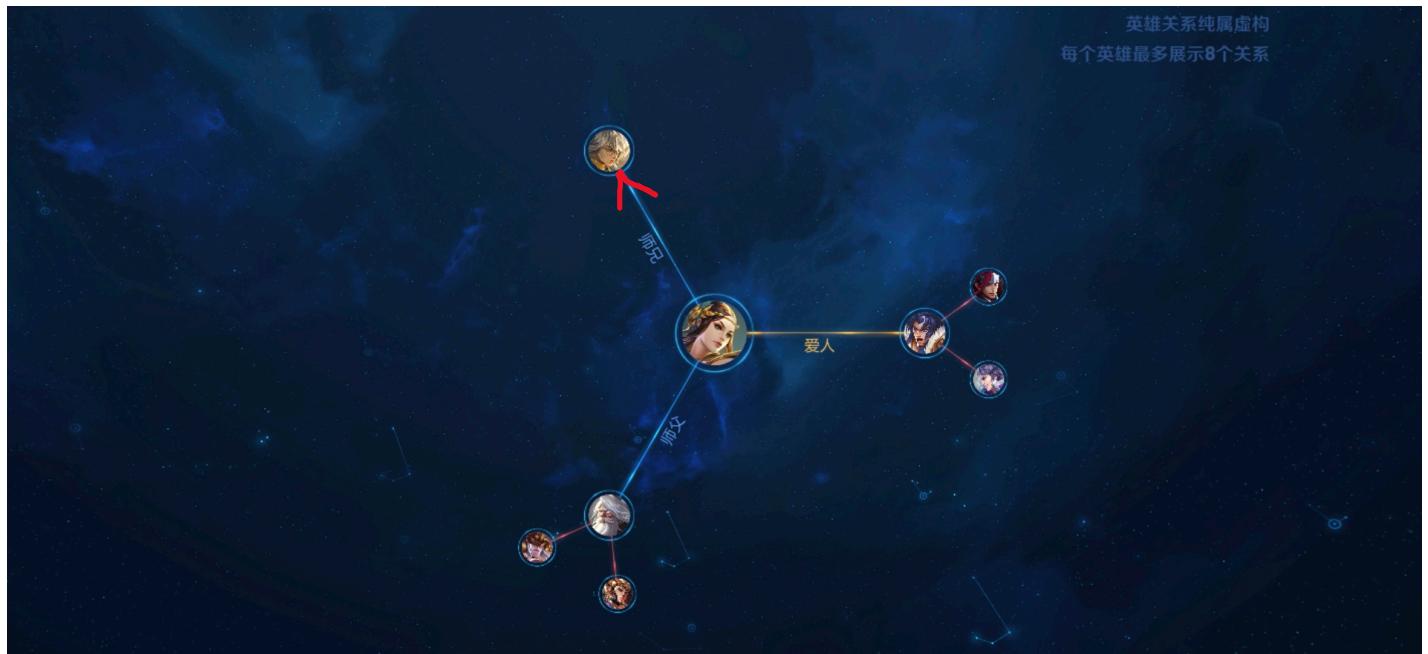
# 有向图连通性

**强连通图：**给定有向图 $G=(VE)$ , 并且给定该图G中的任意两个结点 $u$ 和 $v$ , 如果结点 $u$ 与结点 $v$ 相互可达, 即至少存在一条路径可以由结点 $u$ 开始, 到结点 $v$ 终止, 同时存在至少有一条路径可以由结点 $v$ 开始, 到结点 $u$ 终止, 那么就称该有向图G是强连通图。

**弱连通图:**若至少有一对结点不满足单向连通, 但去掉边的方向后从无向图的观点看是连通图, 则D称为弱连通图.



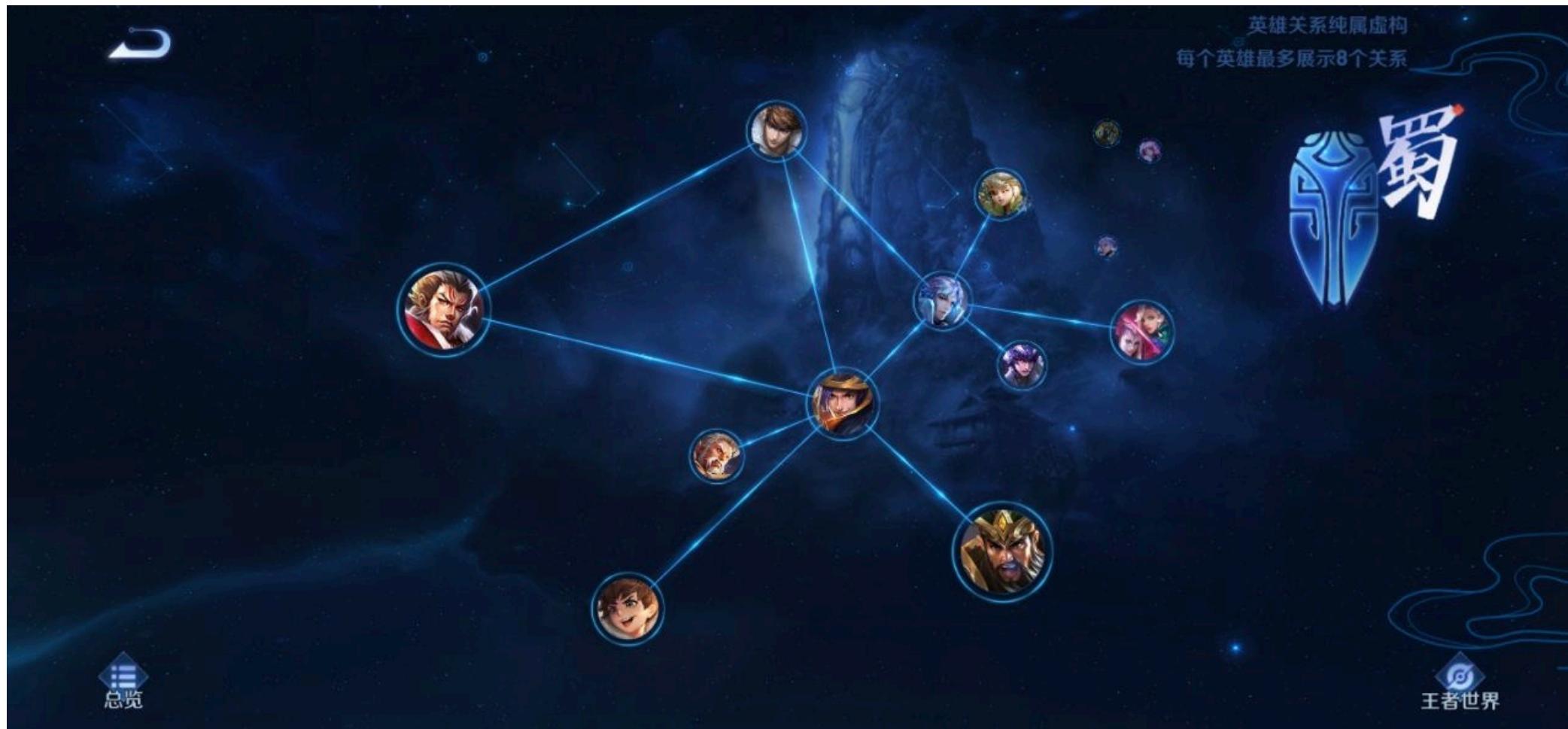
强连通图



弱连通图

# 图中两两节点距离的最大值

最短路径，图直径



度中心性

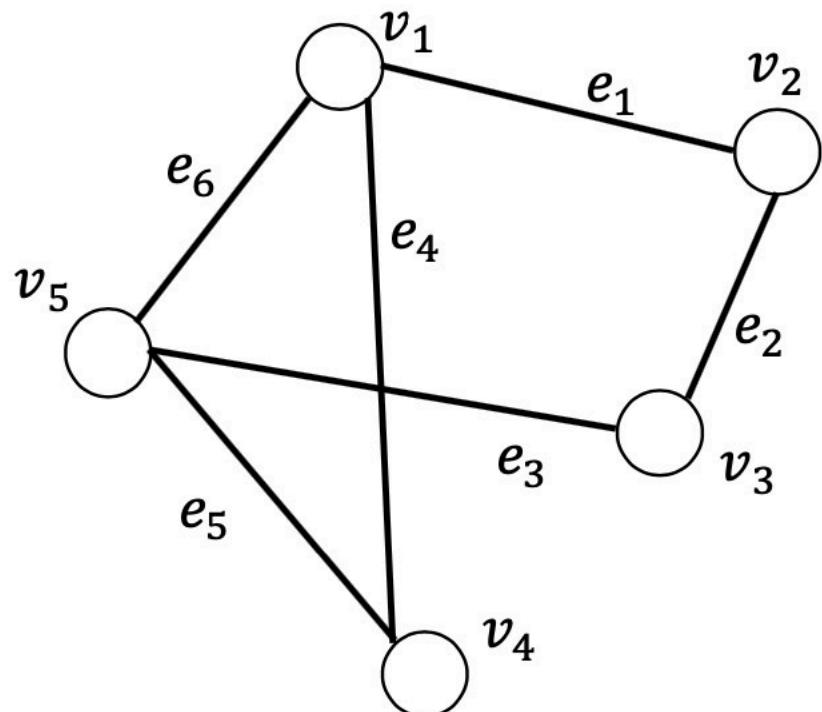


$$\text{度中心性} = \frac{N_{degree}}{n - 1}$$

$$\text{刘备} = \frac{6}{10 - 1}$$

$$\text{赵云} = \frac{3}{10 - 1}$$

## 特征向量中心性 Eigenvector Centrality



$$A * x = \lambda * x$$

特征值:

[ 2.4811943 -2.	-1.17008649 -0.	0.68889218]
-----------------	-----------------	-------------

特征向量:

[ [-0.5298991 -0.5	-0.43248663 0.5	0.1793384 ]
[-0.35775124 0.5	0.19929465 0.5	-0.57645095]
[-0.35775124 -0.5	0.19929465 -0.5	-0.57645095]
[-0.42713229 0.	0.73923874 0.	0.52065737]
[-0.5298991 0.5	-0.43248663 -0.5	0.1793384 ]]

特征向量中心性

{1: 0.53, 2: 0.358, 3: 0.358, 4: 0.427, 5: 0.53}

## 中介中心性 Betweenness Centrality



$$Betweenness = \frac{\text{经过该节点的最短路径}}{\text{其余两两节点的最短路径}}$$

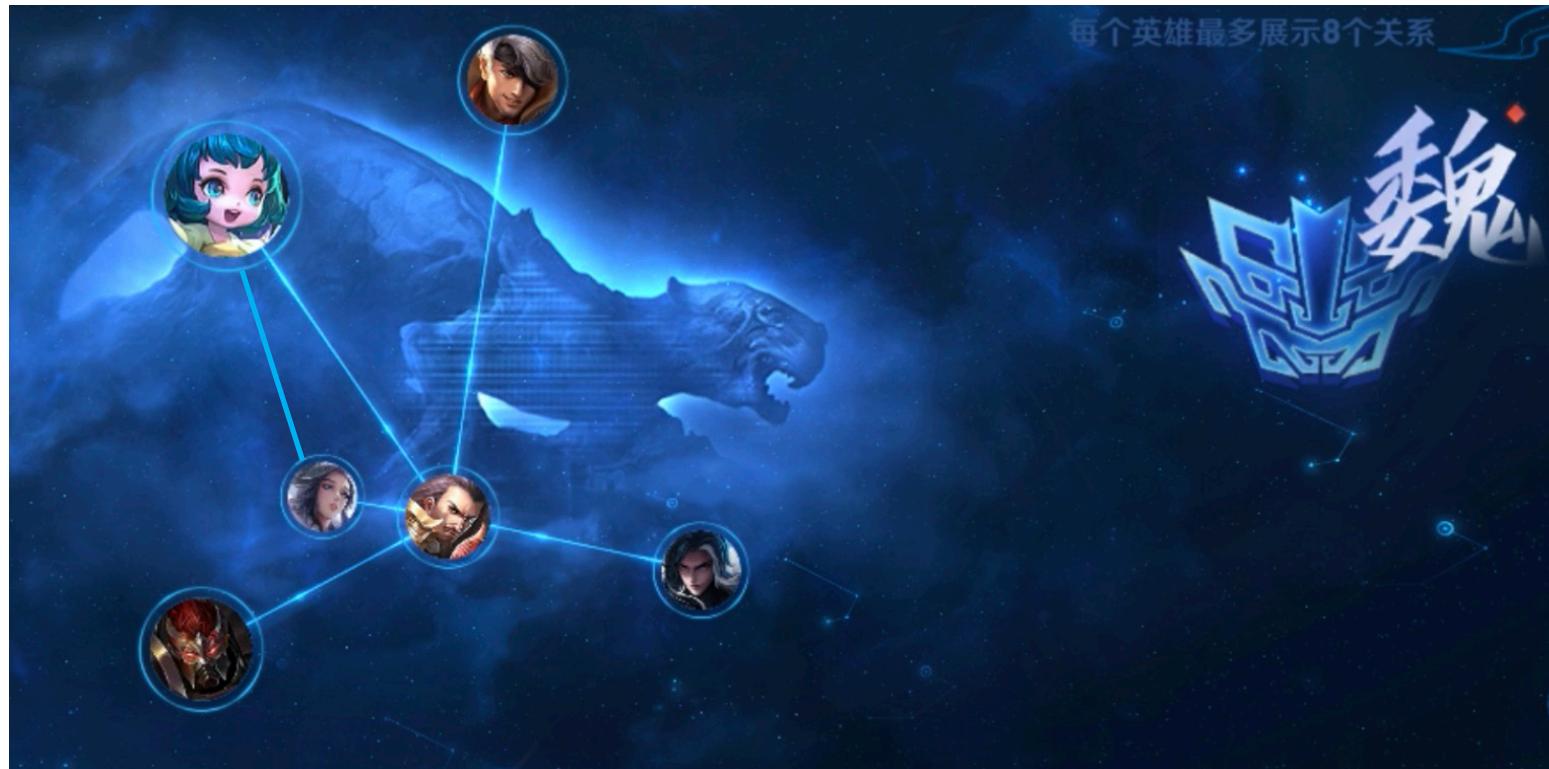
$$\text{曹操}_{betweenness} = \frac{(0 + 1 + 1 + 0.5) + (0 + 0 + 1 + 1) + (0 + 1 + 1 + 0.5) + 4 + 4}{4 * 5} = 15/20$$

$$\text{甄姬}_{betweenness} = \frac{0.5 + 0.5}{20} = 0.05$$

$$\text{蔡文姬 司马懿 ... } betweenness = 0$$

感谢：暖心的爸爸

## 连接中心性 Closeness

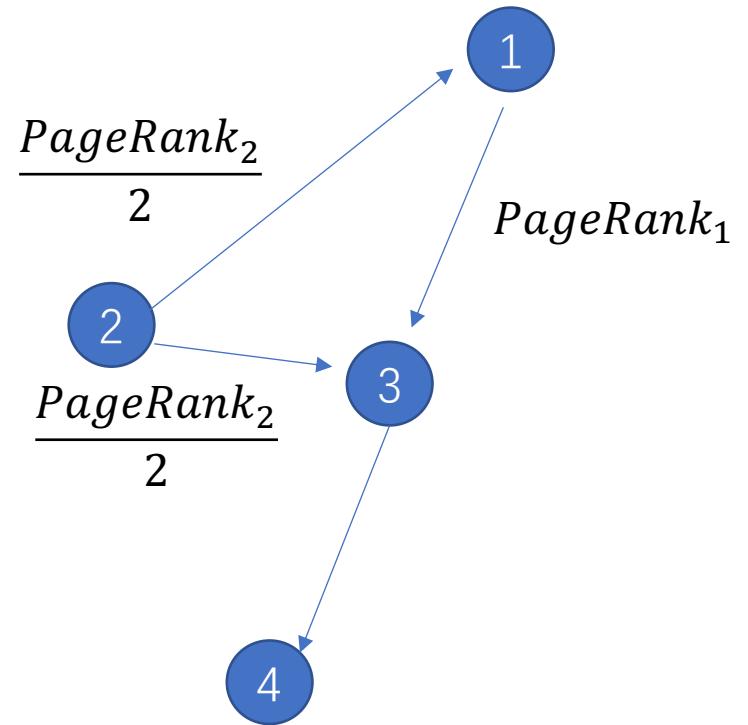


$$Closeness = \frac{n - 1}{\text{节点到其他节点最短路径之和}}$$

$$\text{曹操}_{closeness} = \frac{6 - 1}{5} = 1$$

$$\text{甄姬}_{closeness} = \frac{6 - 1}{1 + 1 + 2 + 2 + 2} = 5/8$$

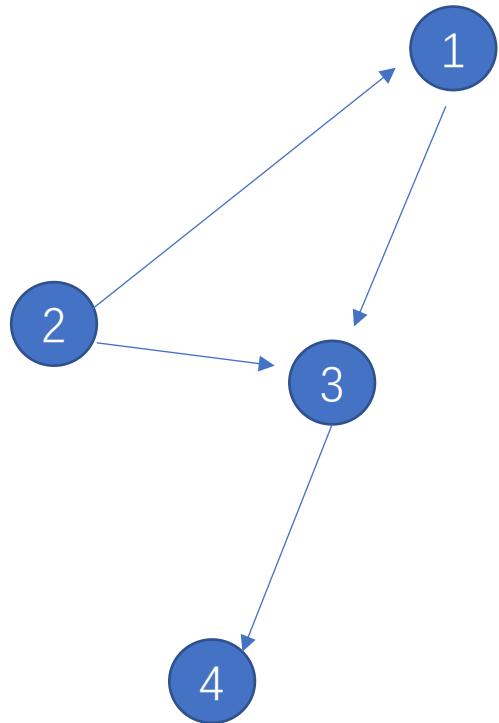
## PageRank



$$PageRank_3 = PageRank_1 + \frac{PageRank_2}{2}$$

阻尼系数=0.85

## HITS



Hub : 门户网站 hao123  
Authority : 百度, 淘宝, 天猫

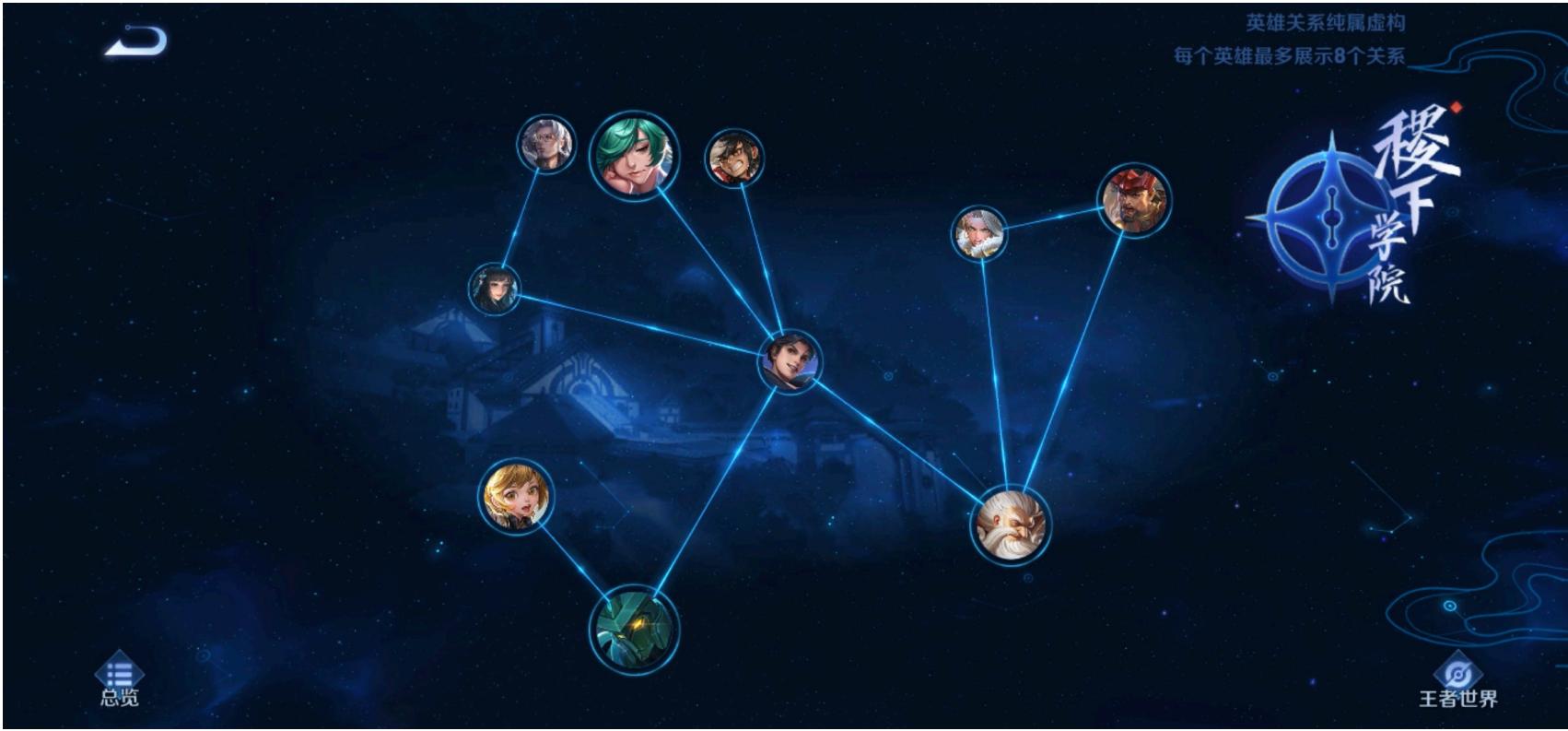
$$V1_{hub} = V3_{authority}$$

$$V1_{authority} = V2_{hub}$$

$$V3_{hub} = V4_{authority}$$

$$V3_{authority} = V2_{hub} + V1_{hub}$$

例子



	<i>degree</i>	<i>Eigenvector</i>	<i>Betweenness</i>	<i>closeness</i>	<i>PageRank</i>	<i>HITS(Hub)</i>	<i>HITS(Authority)</i>
	5	0.574	0.861	0.692	0.237	0.2015	0.2015
	1	0.105	0	0.333	0.060	0.0371	0.0371
	3	0.466	0.388	0.529	0.133	0.1636	0.1636

## 2. Graph Embedding

1. DeepWalk

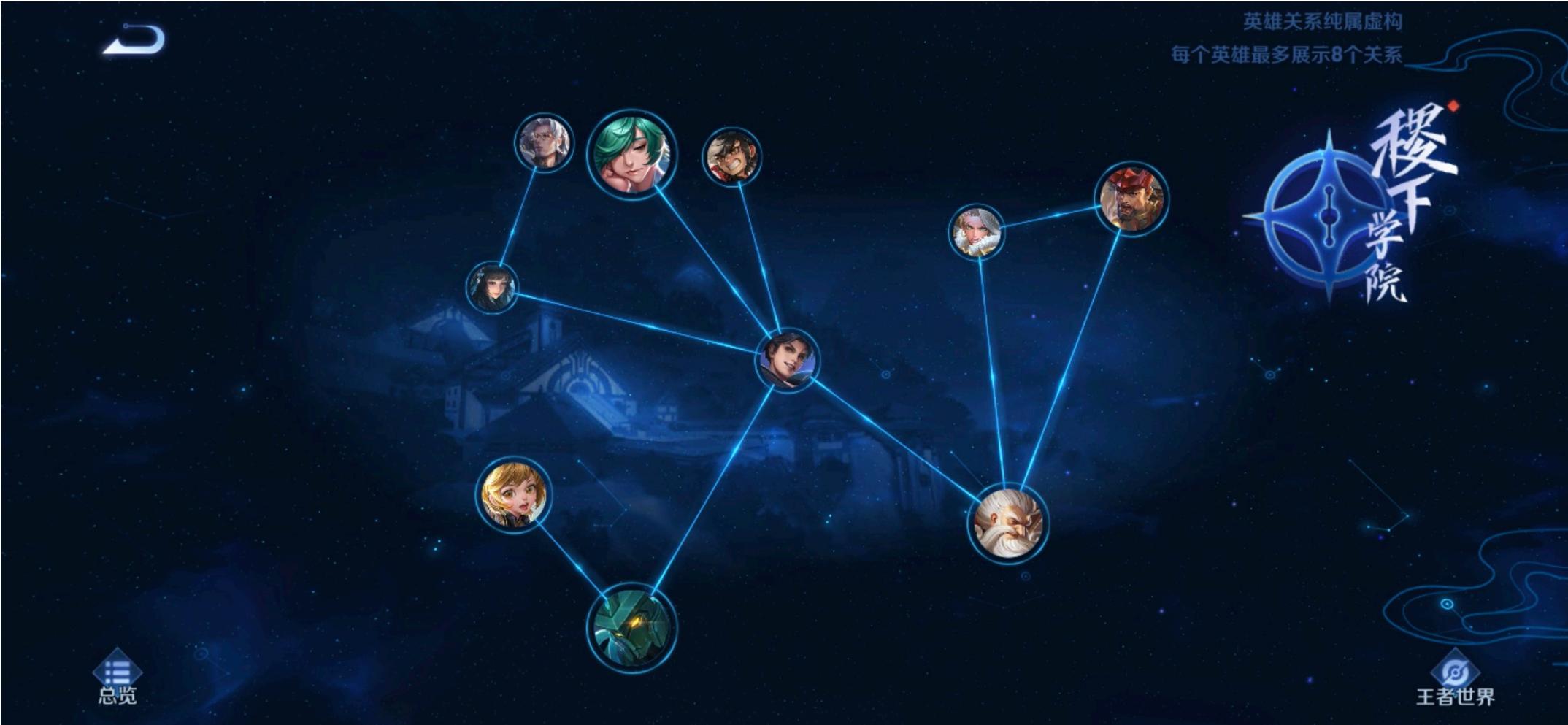
2. LINE

3. SDNE

4. Node2vec

5. Struc2vec

## Graph embedding的作用

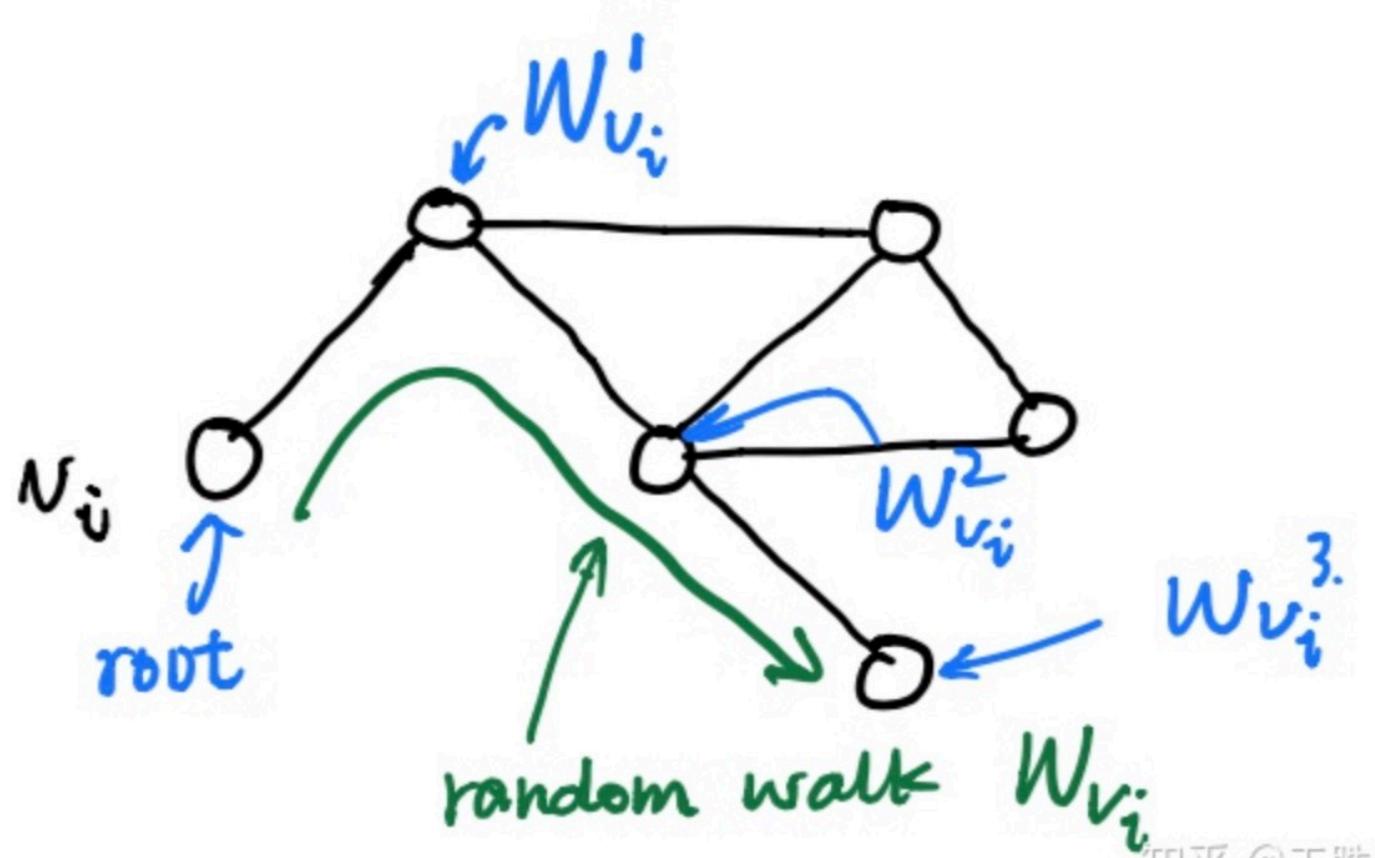


10个节点，如果我要表示这10个节点，需要一个10维度one-hot向量

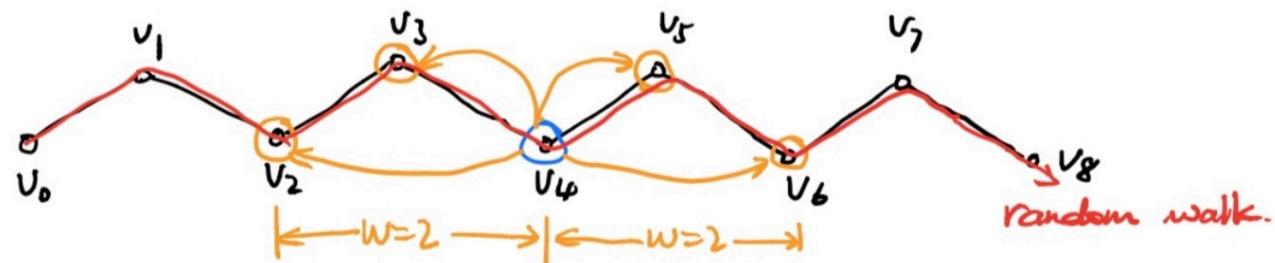
[0. 1. 0. 0. 0. 0. 0. 0. 0.]

[0. 0. 1. 0. 0. 0. 0. 0. 0.]

# 1. DeepWalk



知乎 @王胜



$\Pr \{ \underbrace{\{v_2, v_3, v_5, v_6\}}_{\text{rw 中出现 } v_2, v_3, v_5, v_6} | \underbrace{\Phi(v_4)}_{\text{以 } v_4}\}$

---

**Algorithm 1** DEEPWALK( $G, w, d, \gamma, t$ )

---

**Input:** graph  $G(V, E)$

window size  $w$

embedding size  $d$

walks per vertex  $\gamma$

walk length  $t$

**Output:** matrix of vertex representations  $\Phi \in \mathbb{R}^{|V| \times d}$

1: Initialization: Sample  $\Phi$  from  $\mathcal{U}^{|V| \times d}$

2: Build a binary Tree  $T$  from  $V$

3: **for**  $i = 0$  to  $\gamma$  **do**

4:    $\mathcal{O} = \text{Shuffle}(V)$

5:   **for each**  $v_i \in \mathcal{O}$  **do**

6:      $\mathcal{W}_{v_i} = \text{RandomWalk}(G, v_i, t)$

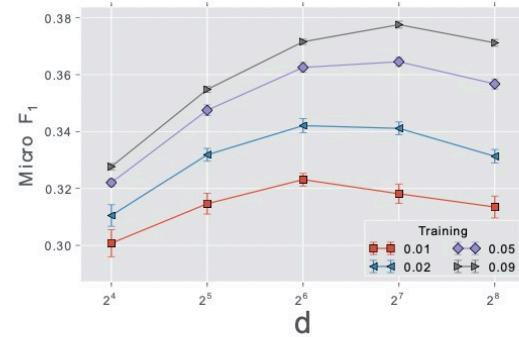
7:     SkipGram( $\Phi, \mathcal{W}_{v_i}, w$ )

8:   **end for**

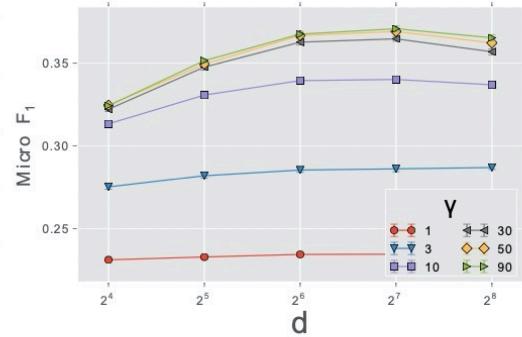
9: **end for**

---

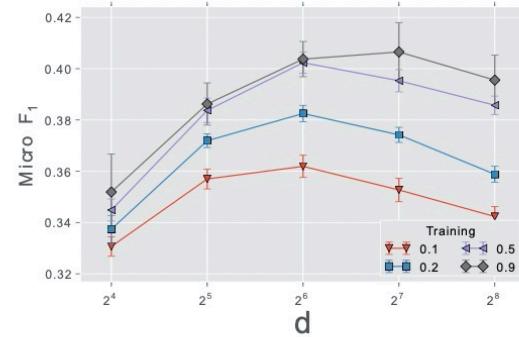
$d$  - embedding维度       $\gamma$  迭代次数



(a1) FLICKR,  $\gamma = 30$

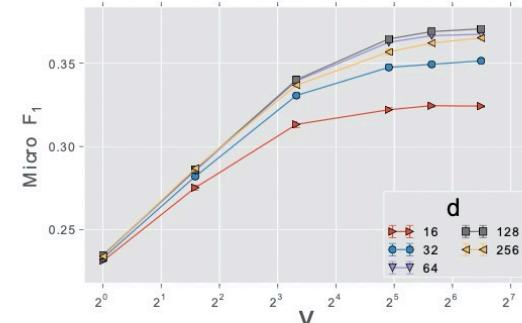


(a2) FLICKR,  $T_R = 0.05$

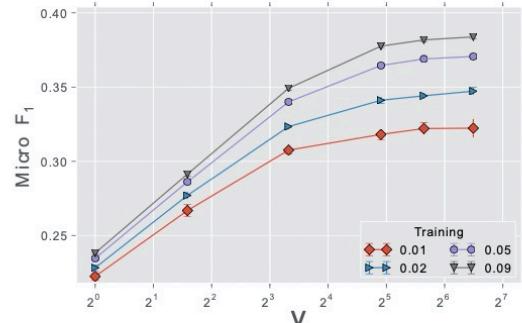


(a3) BLOGCATALOG,  $\gamma = 30$

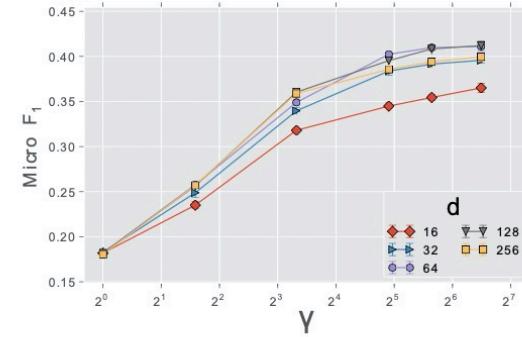
(a) Stability over dimensions,  $d$



(b1) FLICKR,  $T_R = 0.05$

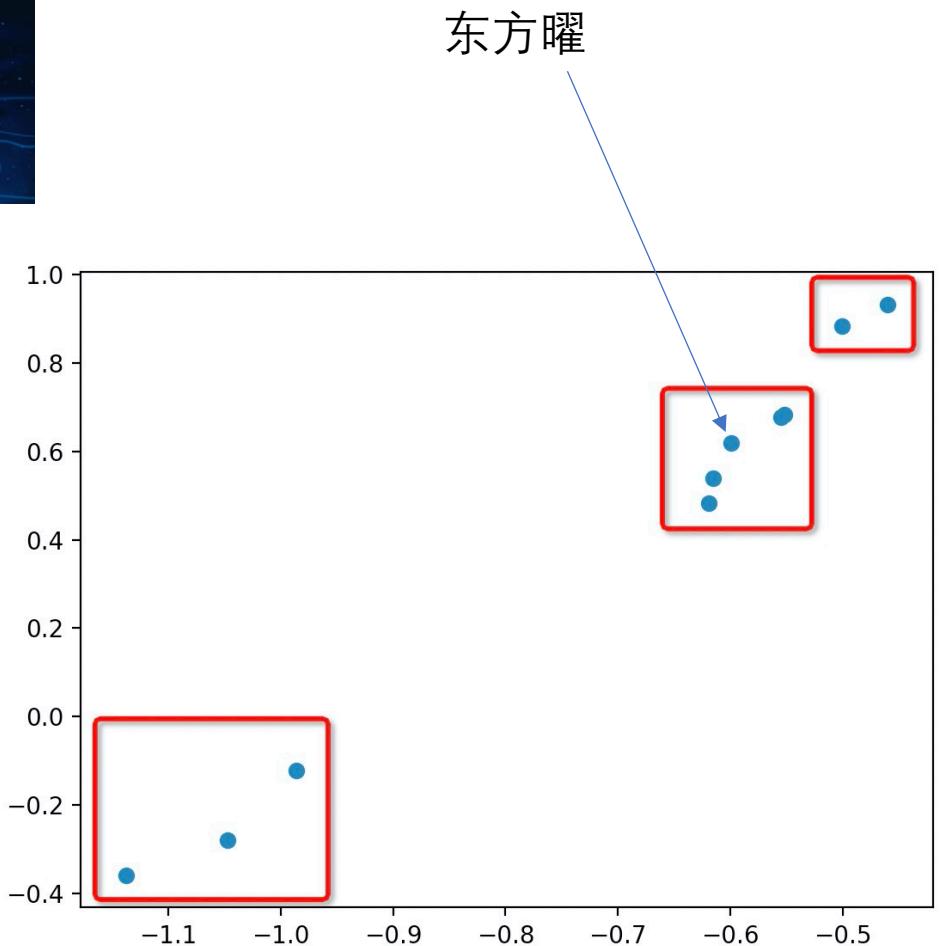
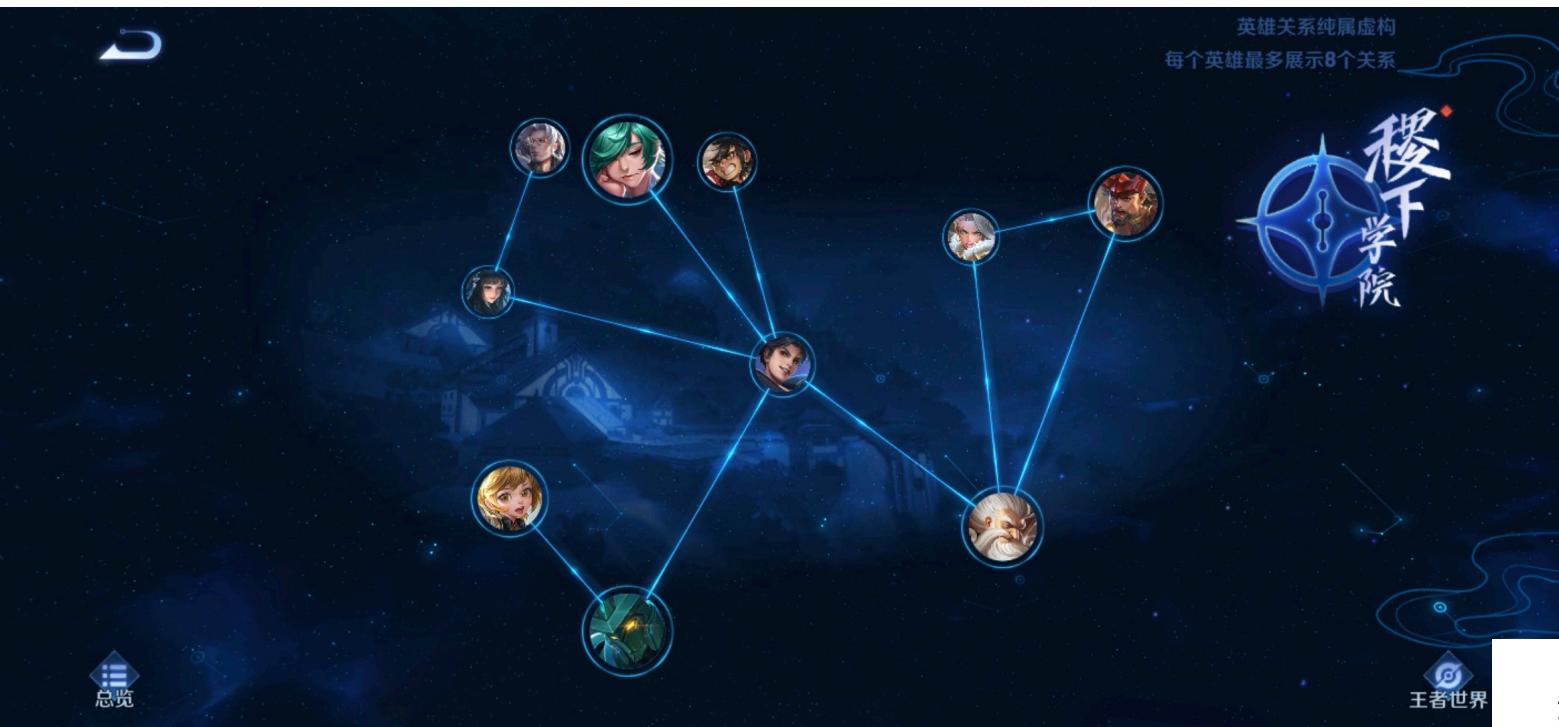


(b2) FLICKR,  $d = 128$

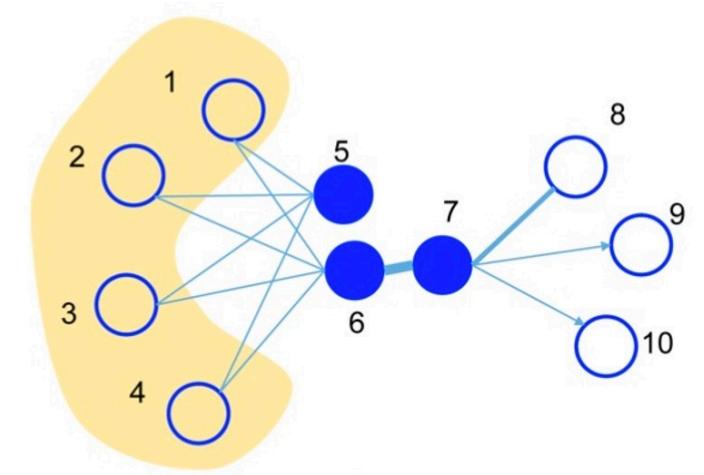


(b3) BLOGCATALOG,  $T_R = 0.5$

(b) Stability over number of walks,  $\gamma$



## 2. LINE: Large-scale Information Network Embedding



LINE – 大规模的图上，表示节点之间的结构信息  
一阶：局部的结构信息  
二阶：节点的邻居。共享邻居的节点可能是相似的

We discuss several practical issues of the LINE model.  
**Low degree vertices.** One practical issue is how to accurately embed vertices with small degrees. As the number

DeepWalk在无向图上, LINE在有向图可以使用

一阶相似性：

*first-order* proximity, for each undirected edge  $(i, j)$ , we define the joint probability between vertex  $v_i$  and  $v_j$  as follows:

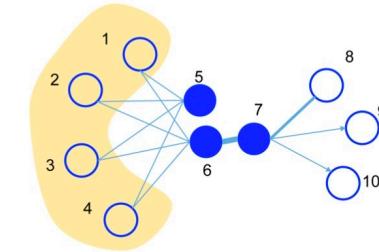
$$p_1(v_i, v_j) = \frac{1}{1 + \exp(-\vec{u}_i^T \cdot \vec{u}_j)}, \quad (1)$$

where  $\vec{u}_i \in R^d$  is the low-dimensional vector representation of vertex  $v_i$ . Eqn. (1) defines a distribution  $p(\cdot, \cdot)$  over the space  $V \times V$ , and its empirical probability can be defined as  $\hat{p}_1(i, j) = \frac{w_{ij}}{W}$ , where  $W = \sum_{(i,j) \in E} w_{ij}$ . To preserve the first-order proximity, a straightforward way is to minimize the following objective function:

$$O_1 = d(\hat{p}_1(\cdot, \cdot), p_1(\cdot, \cdot)), \quad (2)$$

where  $d(\cdot, \cdot)$  is the distance between two distributions. We choose to minimize the KL-divergence of two probability distributions. Replacing  $d(\cdot, \cdot)$  with KL-divergence and omitting some constants, we have:

$$O_1 = - \sum_{(i,j) \in E} w_{ij} \log p_1(v_i, v_j), \quad (3)$$



$$p_1(v_6, v_7) = \frac{1}{1 + \exp(-u_6^T \cdot u_7)} \quad \text{联合概率分布}$$

$$\hat{p}_1(6,7) = \frac{w_{6,7}}{w_{1,5} + w_{2,5} + w_{6,7} + \dots} \quad \text{经验概率分布}$$

$$D_{KL}(p||q) = \sum_{i=1}^n p(x_i) \log\left(\frac{p(x_i)}{q(x_i)}\right)$$

$A(0) = 1/2, A(1) = 1/2$

$$B(0) = 1/4, B(1) = 3/4$$

$$\text{KL}(A \| B) = 1/2 \log\left(\frac{1/2}{1/4}\right) + 1/2 \log\left(\frac{1/2}{3/4}\right) = 1/2 \log(4/3)$$

$$\Sigma \frac{w_{ij}}{W} \log\left(\frac{w_{ij}}{W} / p_1(v_i, v_j)\right)$$

$$= \Sigma \frac{w_{ij}}{W} \left[ \log\left(\frac{w_{ij}}{W}\right) - \log(p_1(v_i, v_j)) \right]$$

$$= \Sigma \frac{w_{ij}}{W} \log\left(\frac{w_{ij}}{W}\right) - \Sigma \frac{w_{ij}}{W} \log(p_1(v_i, v_j))$$

$\vec{u}$ : 顶点本身的表示向量

$\vec{u}'$  : 该点作为其他节点邻居时的表示向量

二阶相似性：

定义给定顶点 $v_i$ 条件下，产生邻居顶点 $v_j$ 的概率

$$p_2(v_j|v_i) = \frac{\exp(\vec{u}'_j^T \cdot \vec{u}_i)}{\sum_{k=1}^{|V|} \exp(\vec{u}'_k^T \cdot \vec{u}_i)}, \quad (4)$$

控制节点重要性的因子

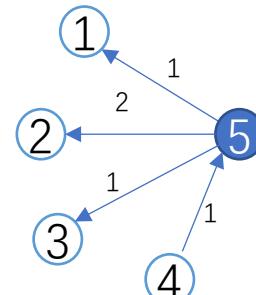
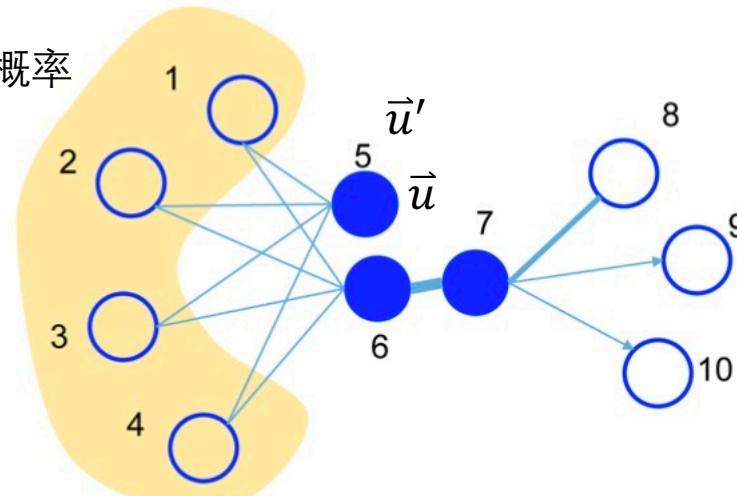
$$O_2 = \sum_{i \in V} \lambda_i d(\hat{p}_2(\cdot|v_i), p_2(\cdot|v_i)), \quad (5)$$

as  $\hat{p}_2(v_j|v_i) = \frac{w_{ij}}{d_i}$ , where  $w_{ij}$  is the weight of the edge  $(i, j)$  and  $d_i$  is the out-degree of vertex  $i$ , i.e.  $d_i = \sum_{k \in N(i)} w_{ik}$ ,

tion. Replacing  $d(\cdot, \cdot)$  with KL-divergence, setting  $\lambda_i = d_i$  and omitting some constants, we have:

$$O_2 = - \sum_{(i,j) \in E} w_{ij} \log p_2(v_j|v_i). \quad (6)$$

By learning  $\{\vec{u}_i\}_{i=1..|V|}$  and  $\{\vec{u}'_i\}_{i=1..|V|}$  that minimize this objective, we are able to represent every vertex  $v_i$  with a d-dimensional vector  $\vec{u}_i$ .



$$p_2(v_1|v_5) = \frac{\exp(\vec{u}'_1 \cdot \vec{u}_5)}{\exp(1,5) + \exp(2,5) + \exp(3,5)}$$

$$\hat{p}_2(v_1|v_5) = 1/(1 + 2 + 1)$$

$$D_{KL}(p||q) = \sum_{i=1}^n p(x_i) \log\left(\frac{p(x_i)}{q(x_i)}\right)$$

$$\frac{w_{ij}}{d_i} \log\left(\frac{w_{ij}}{d_i} / p_2(v_j|v_i)\right)$$

一阶二阶embedding训练完成之后，如果将first-order和second-order组合成一个embedding

#### 4.1.3 *Combining first-order and second-order proximities*

To embed the networks by preserving both the *first-order* and *second-order* proximity, a simple and effective way we find in practice is to train the LINE model which preserves the *first-order* proximity and *second-order* proximity separately and then concatenate the embeddings trained by the two methods for each vertex. A more principled way to

**直接拼接**

LINE效果

GF – 图分解技术

DeepWalk

SkipGram

LINE-SGD : 利用不同的梯度下降方法

Table 3: Results of Wikipedia page classification on WIKIPEDIA data set.

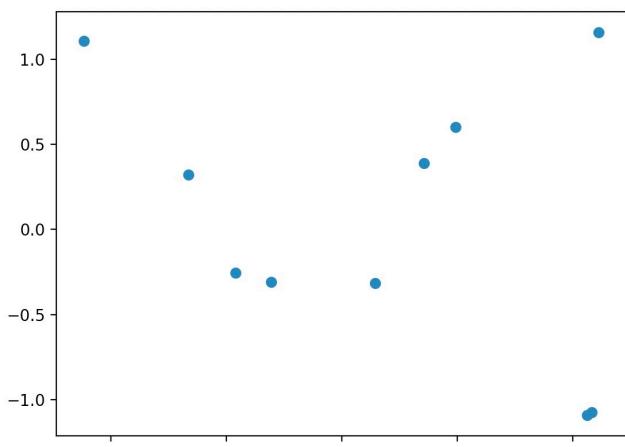
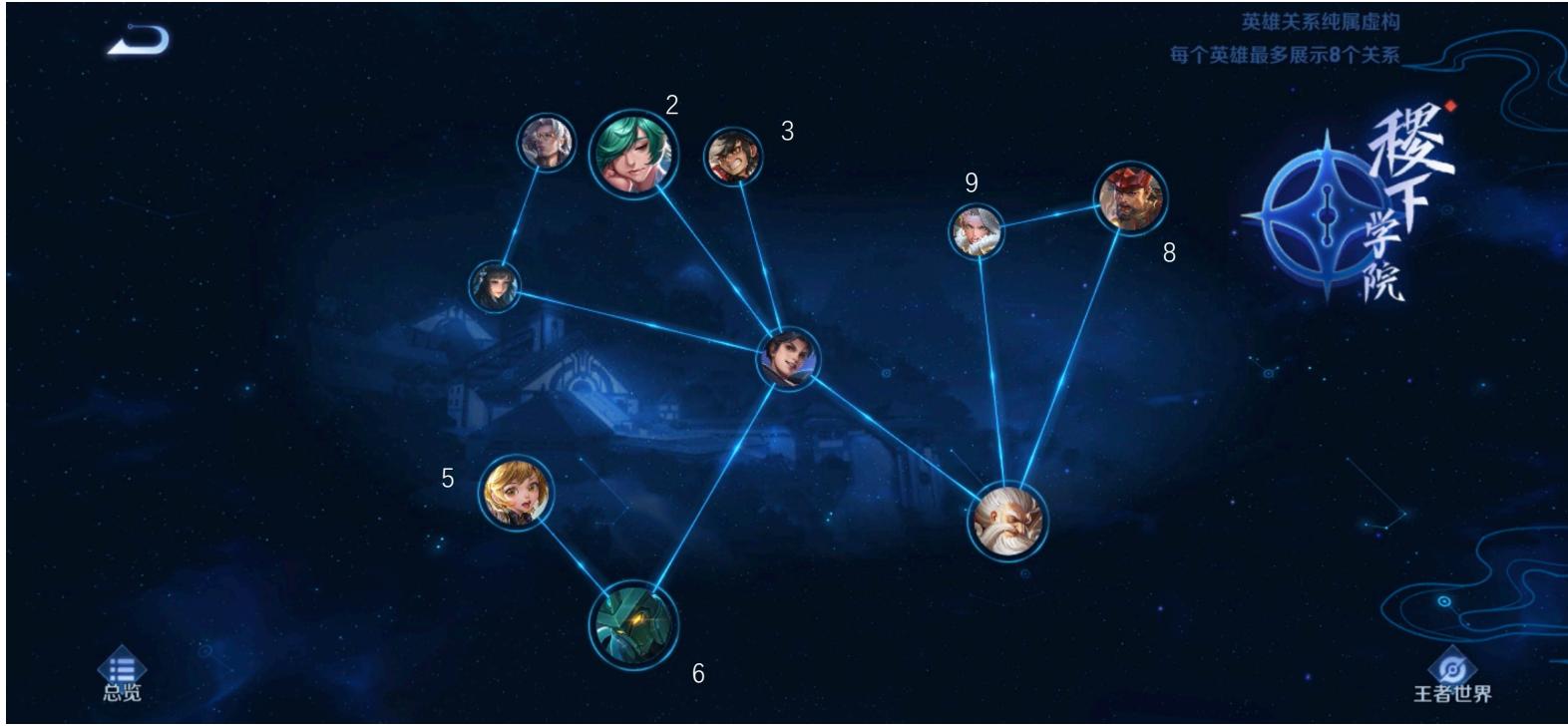
Metric	Algorithm	10%	20%	30%	40%	50%	60%	70%	80%	90%
Micro-F1	GF	79.63	80.51	80.94	81.18	81.38	81.54	81.63	81.71	81.78
	DeepWalk	78.89	79.92	80.41	80.69	80.92	81.08	81.21	81.35	81.42
	SkipGram	79.84	80.82	81.28	81.57	81.71	81.87	81.98	82.05	82.09
	LINE-SGD(1st)	76.03	77.05	77.57	77.85	78.08	78.25	78.39	78.44	78.49
	LINE-SGD(2nd)	74.68	76.53	77.54	78.18	78.63	78.96	79.19	79.40	79.57
	LINE(1st)	79.67	80.55	80.94	81.24	81.40	81.52	81.61	81.69	81.67
	LINE(2nd)	79.93	80.90	81.31	81.63	81.80	81.91	82.00	82.11	82.17
	LINE(1st+2nd)	<b>81.04**</b>	<b>82.08**</b>	<b>82.58**</b>	<b>82.93**</b>	<b>83.16**</b>	<b>83.37**</b>	<b>83.52**</b>	<b>83.63**</b>	<b>83.74**</b>
Macro-F1	GF	79.49	80.39	80.82	81.08	81.26	81.40	81.52	81.61	81.68
	DeepWalk	78.78	79.78	80.30	80.56	80.82	80.97	81.11	81.24	81.32
	SkipGram	79.74	80.71	81.15	81.46	81.63	81.78	81.88	81.98	82.01
	LINE-SGD(1st)	75.85	76.90	77.40	77.71	77.94	78.12	78.24	78.29	78.36
	LINE-SGD(2nd)	74.70	76.45	77.43	78.09	78.53	78.83	79.08	79.29	79.46
	LINE(1st)	79.54	80.44	80.82	81.13	81.29	81.43	81.51	81.60	81.59
	LINE(2nd)	79.82	80.81	81.22	81.52	81.71	81.82	81.92	82.00	82.07
	LINE(1st+2nd)	<b>80.94**</b>	<b>81.99**</b>	<b>82.49**</b>	<b>82.83**</b>	<b>83.07**</b>	<b>83.29**</b>	<b>83.42**</b>	<b>83.55**</b>	<b>83.66**</b>

Significantly outperforms GF at the: \*\* 0.01 and \* 0.05 level, paired t-test.

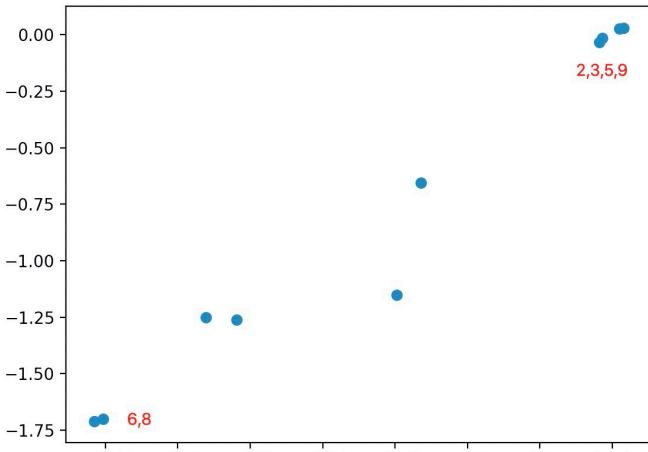
Table 5: Results of multi-label classification on the FLICKR network.

Metric	Algorithm	10%	20%	30%	40%	50%	60%	70%	80%	90%
Micro-F1	GF	53.23	53.68	53.98	54.14	54.32	54.38	54.43	54.50	54.48
	DeepWalk	60.38	60.77	60.90	61.05	61.13	61.18	61.19	61.29	61.22
	DeepWalk(256dim)	60.41	61.09	61.35	61.52	61.69	61.76	61.80	61.91	61.83
	LINE(1st)	63.27	63.69	63.82	63.92	63.96	64.03	64.06	64.17	64.10
	LINE(2nd)	62.83	63.24	63.34	63.44	63.55	63.55	63.59	63.66	63.69
	LINE(1st+2nd)	<b>63.20**</b>	<b>63.97**</b>	<b>64.25**</b>	<b>64.39**</b>	<b>64.53**</b>	<b>64.55**</b>	<b>64.61**</b>	<b>64.75**</b>	<b>64.74**</b>
Macro-F1	GF	48.66	48.73	48.84	48.91	49.03	49.03	49.07	49.08	49.02
	DeepWalk	58.60	58.93	59.04	59.18	59.26	59.29	59.28	59.39	59.30
	DeepWalk(256dim)	59.00	59.59	59.80	59.94	60.09	60.17	60.18	60.27	60.18
	LINE(1st)	62.14	62.53	62.64	62.74	62.78	62.82	62.86	62.96	62.89
	LINE(2nd)	61.46	61.82	61.92	62.02	62.13	62.12	62.17	62.23	62.25
	LINE(1st+2nd)	<b>62.23**</b>	<b>62.95**</b>	<b>63.20**</b>	<b>63.35**</b>	<b>63.48**</b>	<b>63.48**</b>	<b>63.55**</b>	<b>63.69**</b>	<b>63.68**</b>

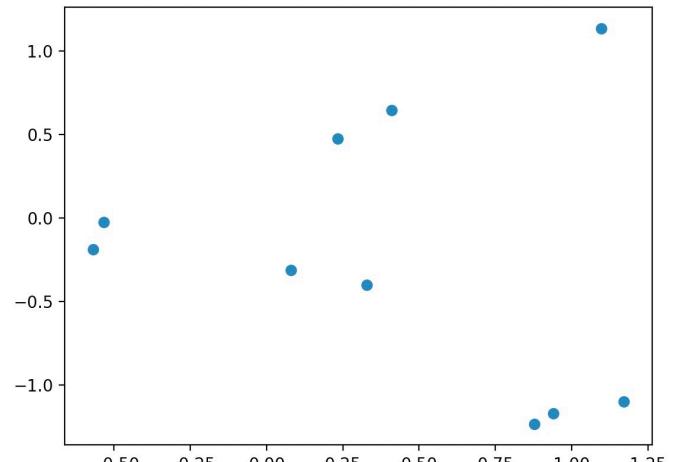
Significantly outperforms DeepWalk at the: \*\* 0.01 and \* 0.05 level, paired t-test.



first



second



all

### 3. node2vec

homophily : 同质性

structural equivalence : 结构等价性

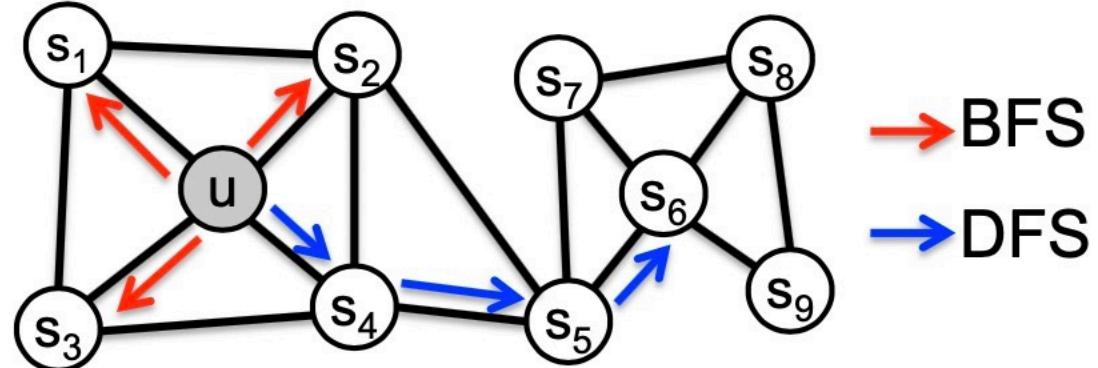
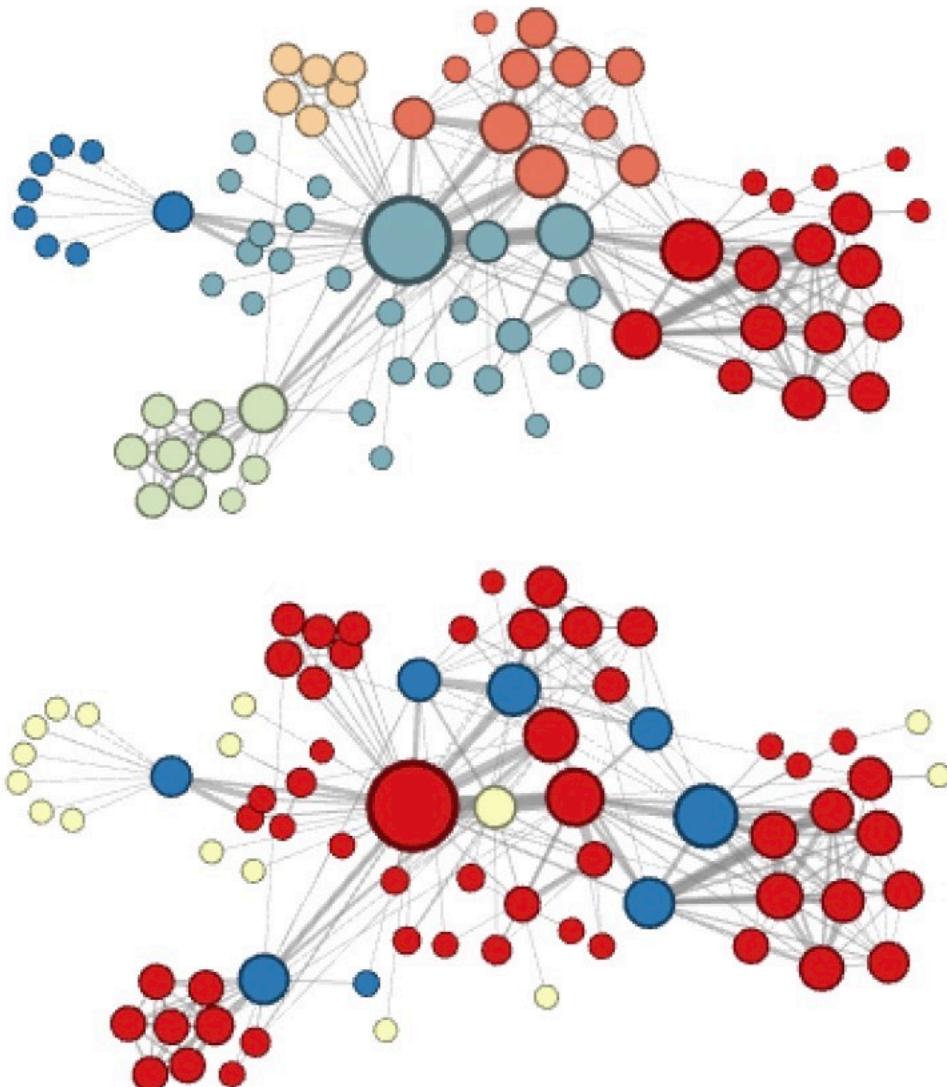


Figure 1: BFS and DFS search strategies from node  $u$  ( $k = 3$ ).



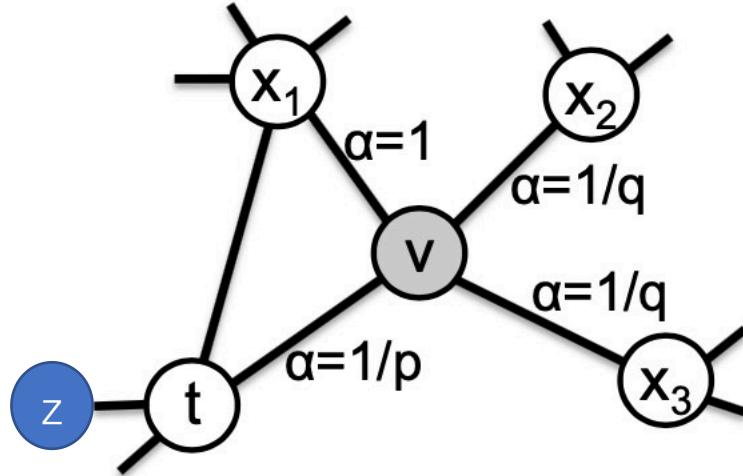


Figure 2: Illustration of the random walk procedure in *node2vec*. The walk just transitioned from  $t$  to  $v$  and is now evaluating its next step out of node  $v$ . Edge labels indicate search biases  $\alpha$ .

abilities  $\pi_{vx}$  on edges  $(v, x)$  leading from  $v$ . We set the unnormalized transition probability to  $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$ , where

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

$$t \rightarrow v \rightarrow x$$

然后有一定的概率走向 $v$ 相连接的下一个顶点  
 $t \rightarrow v \rightarrow \{t, x1, x2, x3\}$

$$t \rightarrow v \rightarrow z: 0$$

$$t \rightarrow v \rightarrow t: w_{v,t} * \frac{1}{p}$$

$$t \rightarrow v \rightarrow x1: w_{v,x1} * 1$$

$$t \rightarrow v \rightarrow x3: w_{v,x3} * \frac{1}{q}$$

---

**Algorithm 1** The node2vec algorithm.

---

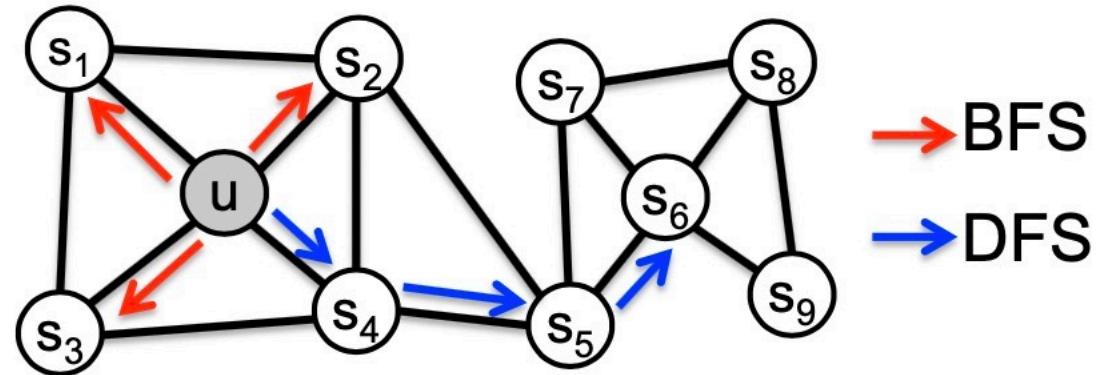
**LearnFeatures** (Graph  $G = (V, E, W)$ , Dimensions  $d$ , Walks per node  $r$ , Walk length  $l$ , Context size  $k$ , Return  $p$ , In-out  $q$ )  
 $\pi = \text{PreprocessModifiedWeights}(G, p, q)$   
 $G' = (V, E, \pi)$   
Initialize  $walks$  to Empty  
**for**  $iter = 1$  **to**  $r$  **do**  
    **for all** nodes  $u \in V$  **do**  
         $walk = \text{node2vecWalk}(G', u, l)$   
        Append  $walk$  to  $walks$   
 $f = \text{StochasticGradientDescent}(k, d, walks)$   
**return**  $f$

---

which maximizes the log-probability of observing a network neighborhood  $N_S(u)$  for a node  $u$  conditioned on its feature representation, given by  $f$ :

$$\max_f \sum_{u \in V} \log Pr(N_S(u) | f(u)). \quad (1)$$

skipGram



random walk  $\{u, s_4, s_5, s_6, s_8, s_9\}$  of length  $l = 6$ , which results in  $N_S(u) = \{s_4, s_5, s_6\}$ ,  $N_S(s_4) = \{s_5, s_6, s_8\}$  and  $N_S(s_5) = \{s_6, s_8, s_9\}$ . Note that sample reuse can introduce some bias in the overall procedure. However, we observe that it greatly improves the efficiency.

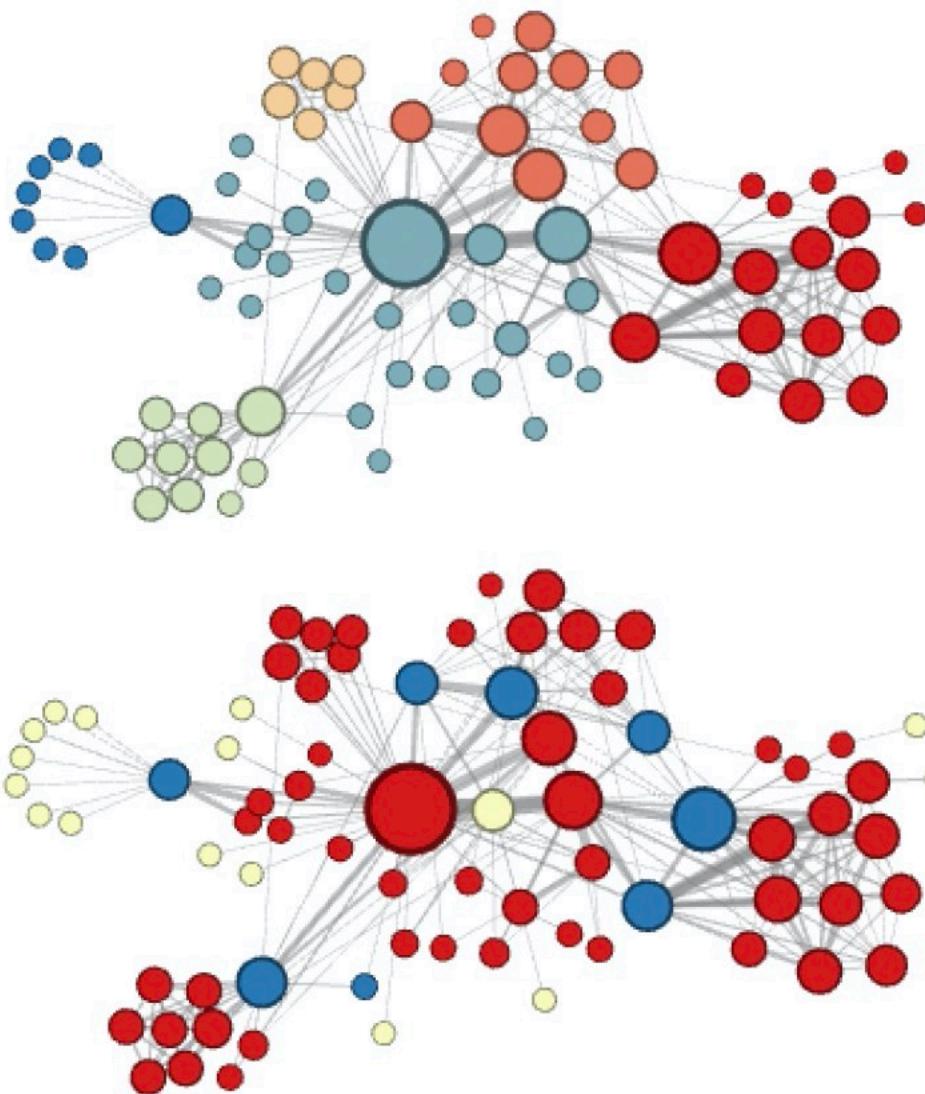


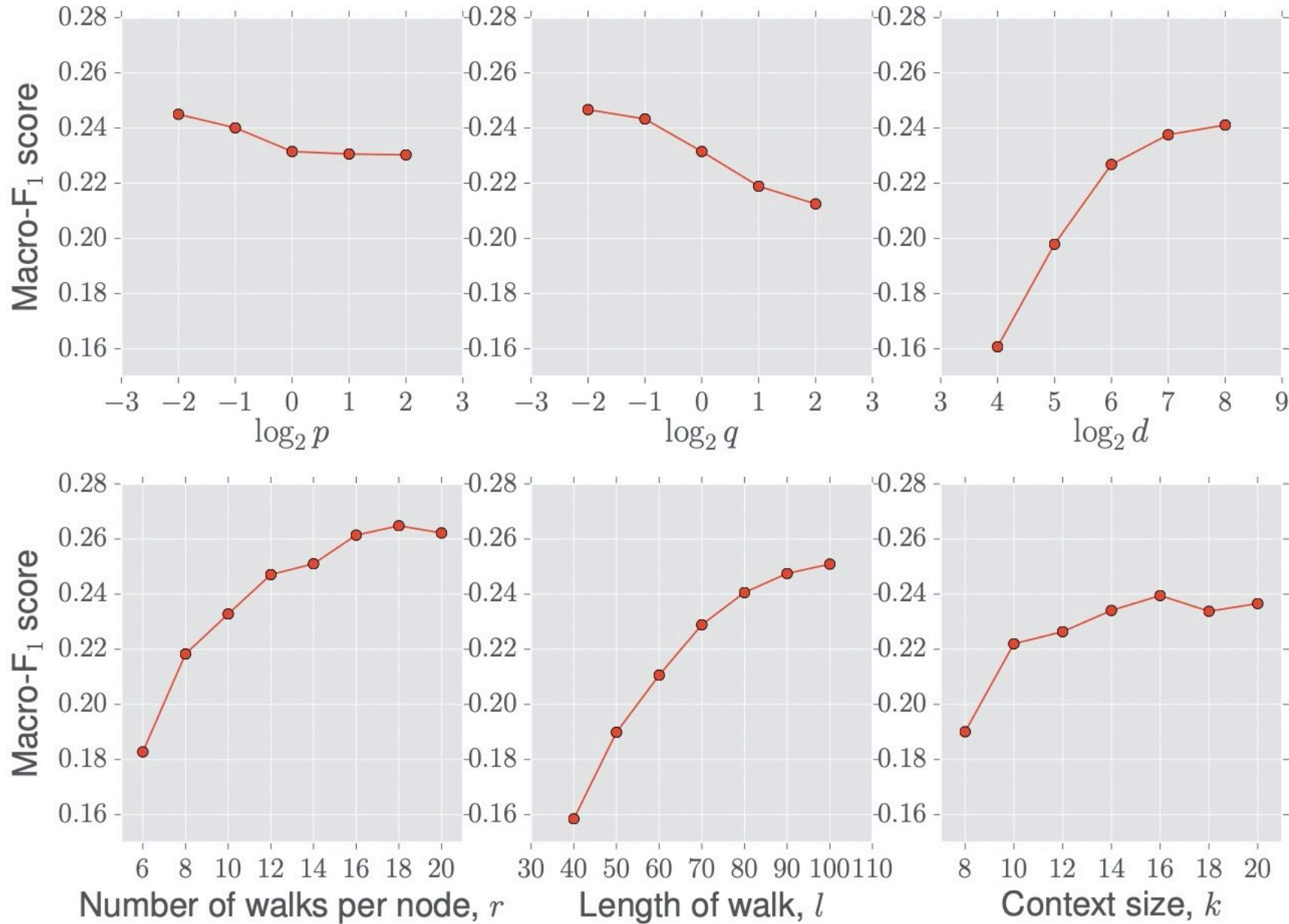
Figure 3(top) shows the example when we set  $p = 1, q = 0.5$ . Notice how regions of the network (*i.e.*, network communities) are colored using the same color. In this setting *node2vec* discov-

DFS, 即 $q$ 值小, 探索强。会捕获homophily同质性节点, 即相邻节点表示类似

BFS, 即 $p$ 值小, 保守周围。会捕获结构性, 即某些节点的图上结构类类似

In order to discover which nodes have the same structural roles we use the same network but set  $p = 1, q = 2$ , use *node2vec* to get node features and then cluster the nodes based on the obtained features. Here *node2vec* obtains a complementary assignment of node

Figure 3: Complementary visualizations of Les Misérables co-appearance network generated by *node2vec* with label colors reflecting homophily (top) and structural equivalence (bottom).

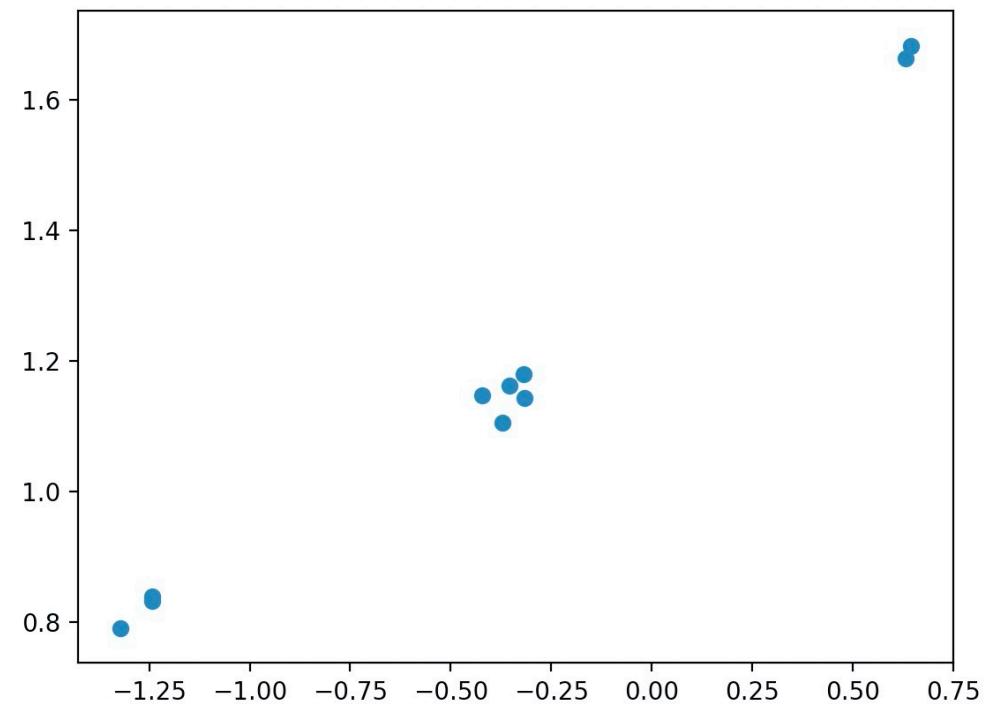
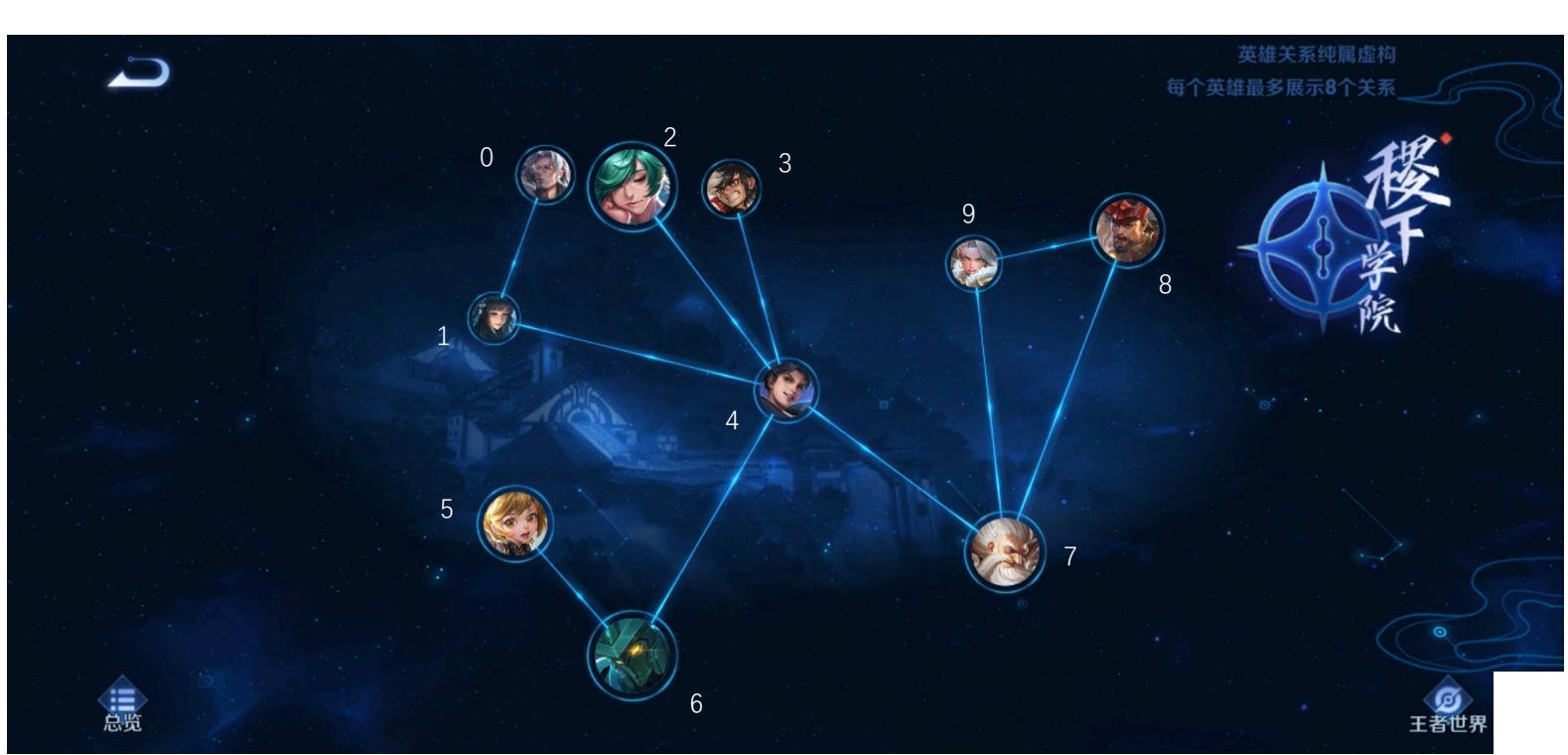


## node和edge的embedding

We also show how feature representations of individual nodes can be extended to pairs of nodes (*i.e.*, edges). In order to generate feature representations of edges, we compose the learned feature representations of the individual nodes using simple binary operators. This compositionality lends *node2vec* to prediction tasks involving nodes as well as edges.

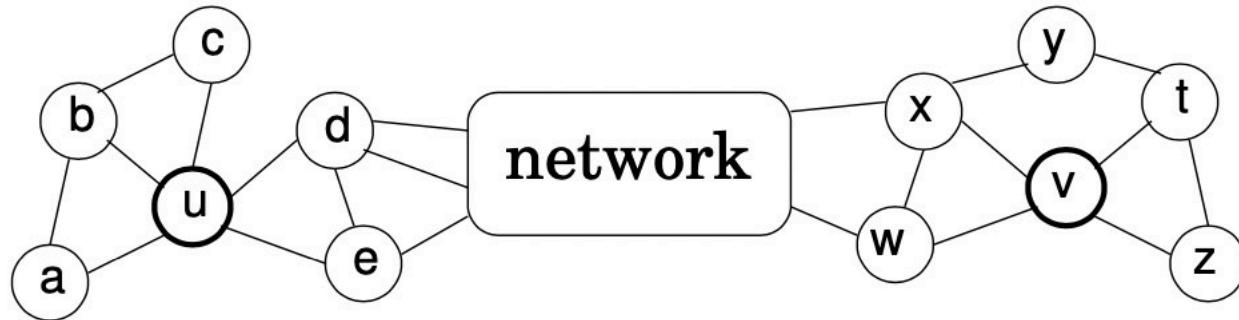
Operator	Symbol	Definition
Average	$\boxplus$	$[f(u) \boxplus f(v)]_i = \frac{f_i(u) + f_i(v)}{2}$
Hadamard	$\boxdot$	$[f(u) \boxdot f(v)]_i = f_i(u) * f_i(v)$
Weighted-L1	$\ \cdot\ _{\bar{1}}$	$\ f(u) \cdot f(v)\ _{\bar{1}i} =  f_i(u) - f_i(v) $
Weighted-L2	$\ \cdot\ _{\bar{2}}$	$\ f(u) \cdot f(v)\ _{\bar{2}i} =  f_i(u) - f_i(v) ^2$

Table 1: Choice of binary operators  $\circ$  for learning edge features. The definitions correspond to the  $i$ th component of  $g(u, v)$ .



#### 4. Struc2vec

之前的node embedding的方式，都是基于近邻关系，但是有些节点没有近邻，但也有相似的结构性



# 1. 定义距离信息

$$f_k(u, v) = f_{k-1}(u, v) + g(s(R_k(u)), s(R_k(v))), \quad k \geq 0 \text{ and } |R_k(u)|, |R_k(v)| > 0$$

(1)

$$f_{-1}(u, v) = 0$$

$$R_0(u) = \{u\}$$

$$R_1(u) = \{A, C, F, D\}$$

$$R_2(u) = \{B, G, E\}$$

$$S(R_1(u)) = S(\{A, C, F, D\}) = \{1, 2, 2, 2\}$$

$$S(R_2(u)) = S(\{B, G, E\}) = \{1, 1, 1\}$$

$$f_{-1}(u, v) = 0$$

$$f_0(u, D) = f_{-1}(u, D) + g(s(R_0(u)), s(R_0(D)))$$

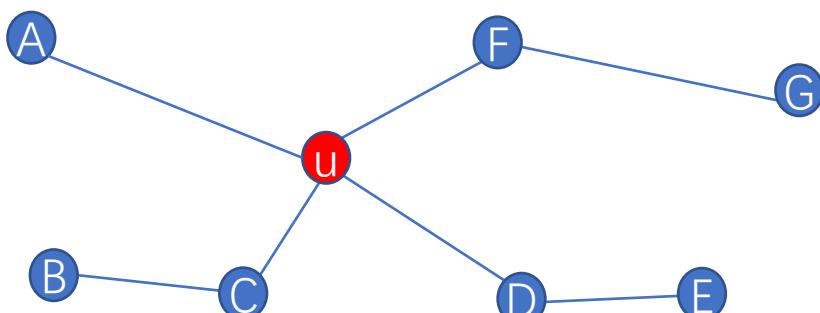
{4}

{2}  $g(\{4\}, \{2\})$

$$f_1(u, D) = f_0(u, D) + g(s(R_1(u)), s(R_1(D)))$$

{1, 2, 2, 2}

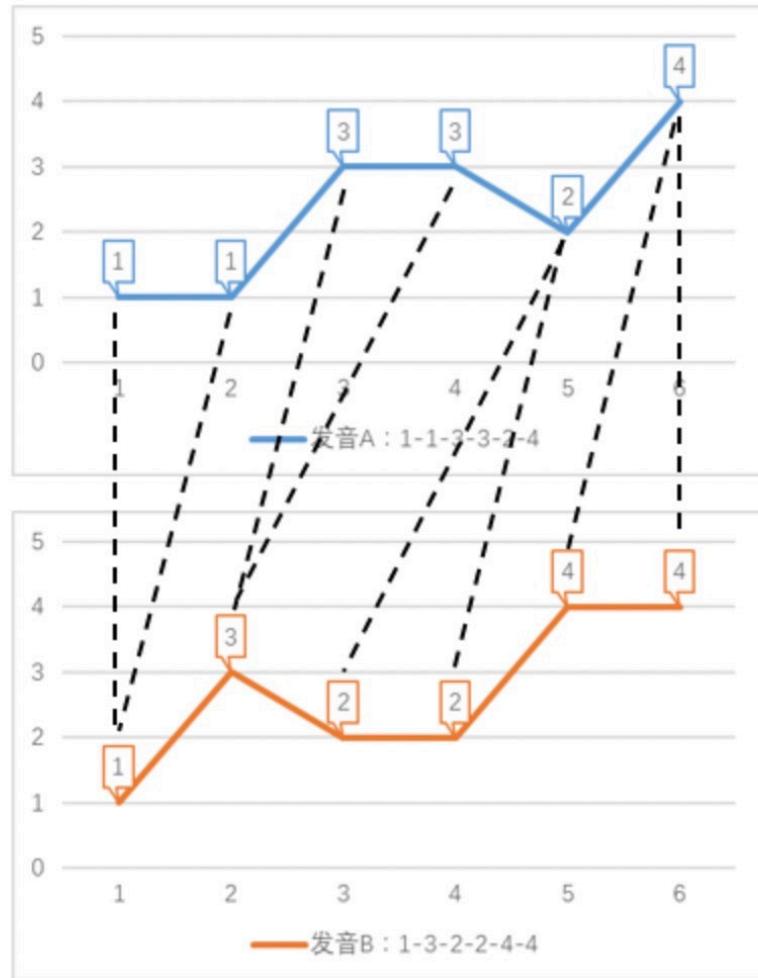
{1, 4}



$g(D_1, D_2)$  – 表示  $D_1, D_2$  之间的距离

$$d(a, b) = \frac{\max(a, b)}{\min(a, b)} - 1 \quad (2) \quad d(u, d) = \frac{4}{2} - 1$$

DTW 动态时间规整：

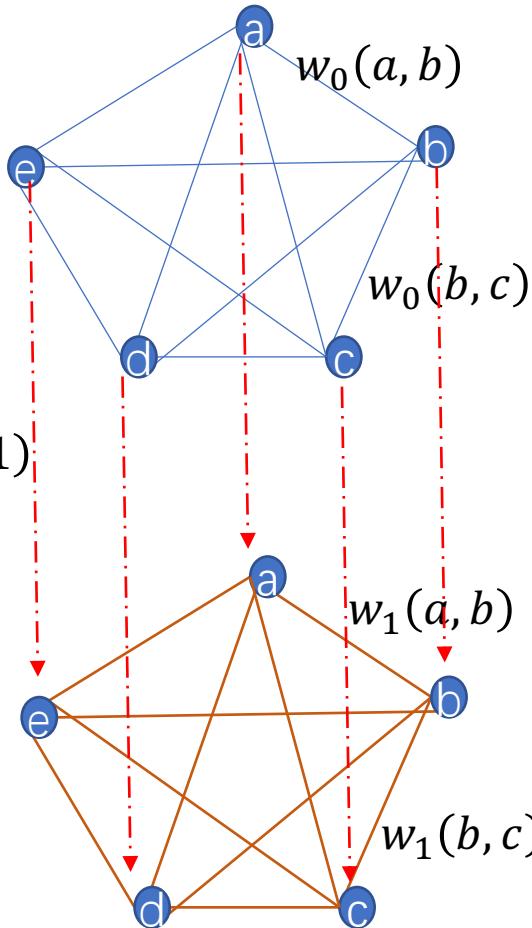


	A(1)=1	A(2) =1	A(3) =3	A(4) =3	A(5) =2	A(6) =4
B(1) =1	0 → 0	2	2	1	3	
B(2) =3	2	2 → 0 → 0	1	1		
B(3) =2	1	1	1	0	2	
B(4) =2	1	1	1	0	2	
B(5) =4	3	3	1	1	2	0
B(6) =4	3	3	1	1	2	0

## 2. 构建多层带权重图

根据上面构建两两节点的  $f_k(u, v)$  的相似度，形成一张完全图

Layer0: 0-hop



$$f_0(a, b)$$

$$f_0(a, c)$$

$$f_0(b, c)$$

$$w_k(a, b) = e^{-f_k(a, b)}$$

$$w(u_k, u_{k+1}) = \log(\Gamma_k(u) + e), \quad k = 0, \dots, k^* - 1$$

$$w(u_k, u_{k-1}) = 1, \quad k = 1, \dots, k^*$$

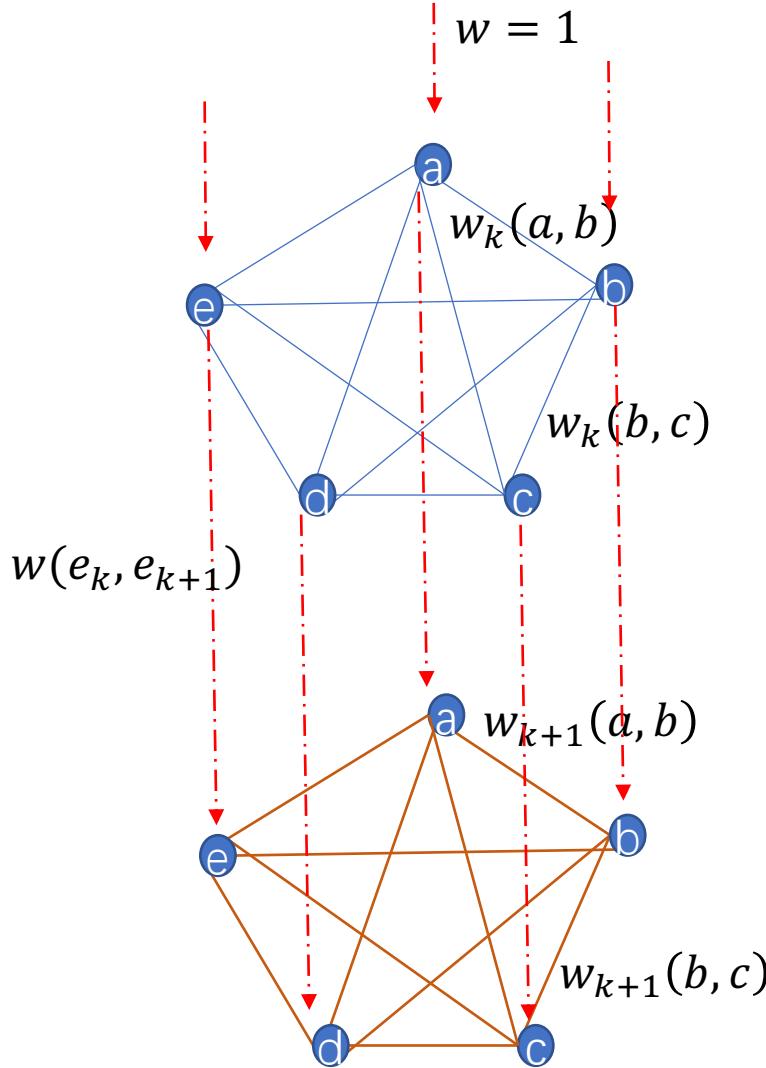
where  $\Gamma_k(u)$  is number of edges incident to  $u$  that have weight larger than the average edge weight of the complete graph in layer  $k$ . In particular:

$$\Gamma_k(u) = \sum_{v \in V} \mathbb{1}(w_k(u, v) > \bar{w}_k) \quad (5)$$

$$\Gamma_0(e) = w_0(e, a), w_0(e, b), w_0(e, c), w_0(e, d) > \bar{w}_k$$

Layer1: 1-hop

### 3. 顶点采样序列



下一次采样时，有 $p$ 的概率在本层游走，有 $1-p$ 的概率进行上下游切换

1. 在本层游走时：

$$p_k(u, v) = \frac{e^{-f_k(u, v)}}{Z_k(u)} \quad (6)$$

where  $Z_k(u)$  is the normalization factor for vertex  $u$  in layer  $k$ , simply given by:

$$Z_k(u) = \sum_{\substack{v \in V \\ v \neq u}} e^{-f_k(u, v)} \quad (7)$$

2. 上下游切换时：

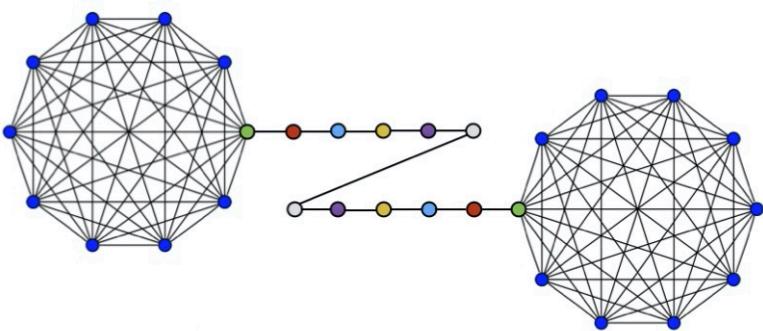
$$p_k(u_k, u_{k+1}) = \frac{w(u_k, u_{k+1})}{w(u_k, u_{k+1}) + w(u_k, u_{k-1})} \quad (8)$$

$$p_k(u_k, u_{k-1}) = 1 - p_k(u_k, u_{k+1})$$

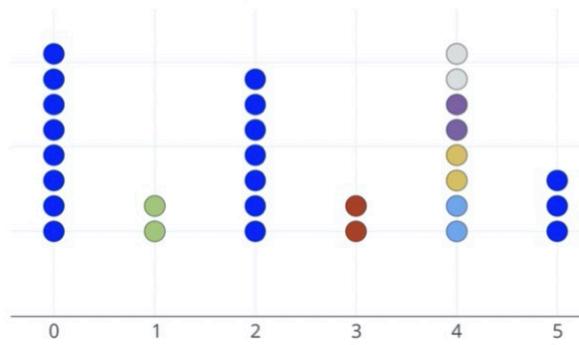
Finally, for each node  $u \in V$ , we start a random walk in its corresponding vertex in layer 0. Random walks have a fixed and relatively short length (number of steps), and the process is repeated a certain number of times, giving rise to multiple independent walks

#### 4. 使用skip-gram生成embedding

We train Skip-Gram according to its optimization problem given by equation (9). Note that while we use Skip-Gram to learn node embeddings, any other technique to learn latent representations for text data could be used in our framework.

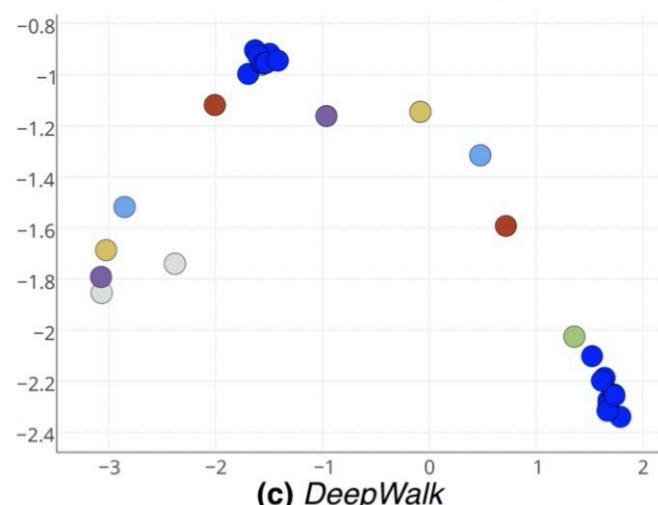


(a) Barbell Graph  $B(10, 10)$

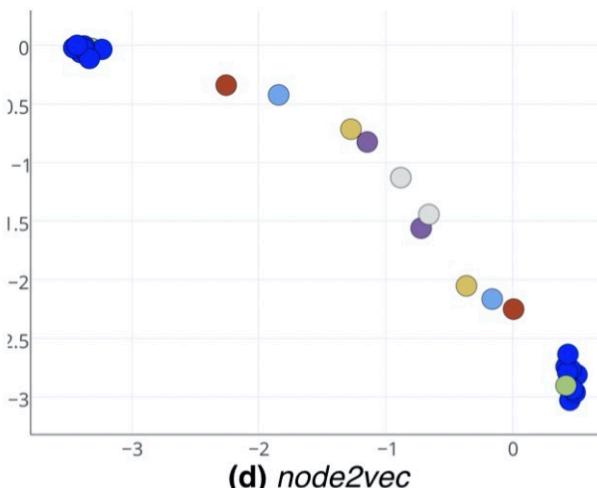


(b) RoIX

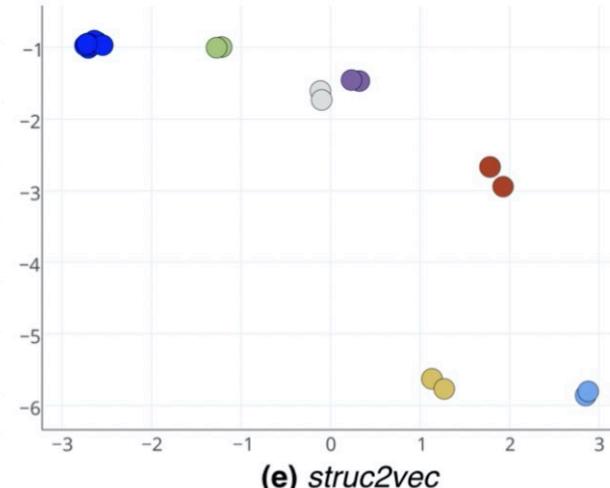
RoIX-捕获某些结构等效性



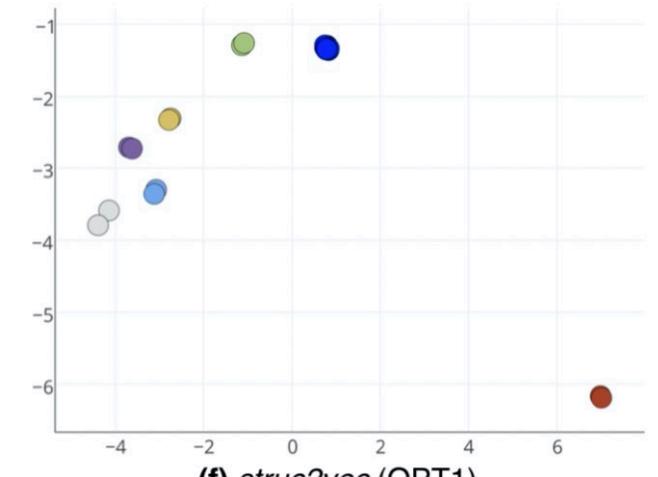
(c) DeepWalk



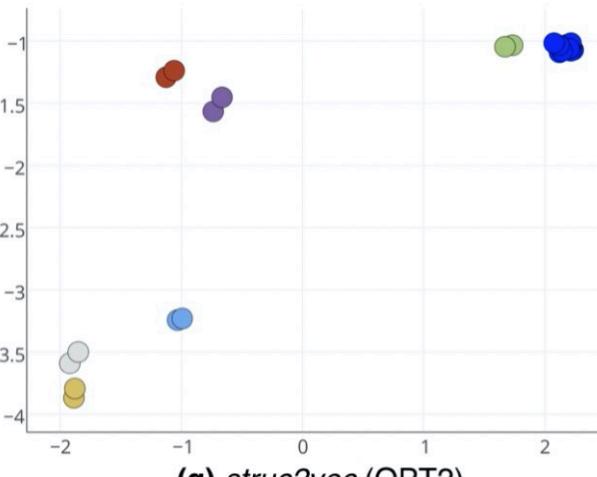
(d) node2vec



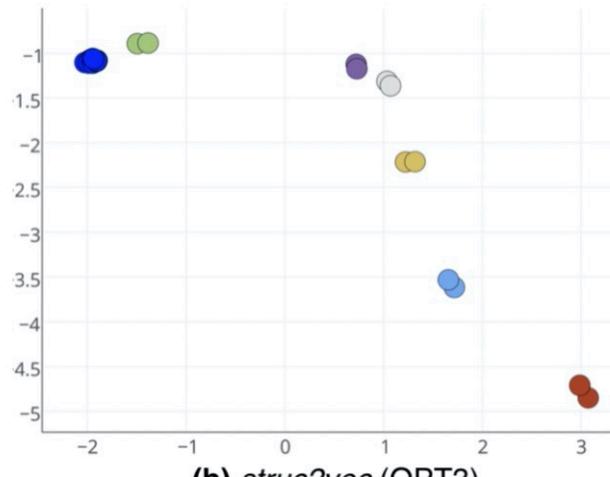
(e) struc2vec



(f) struc2vec (OPT1)



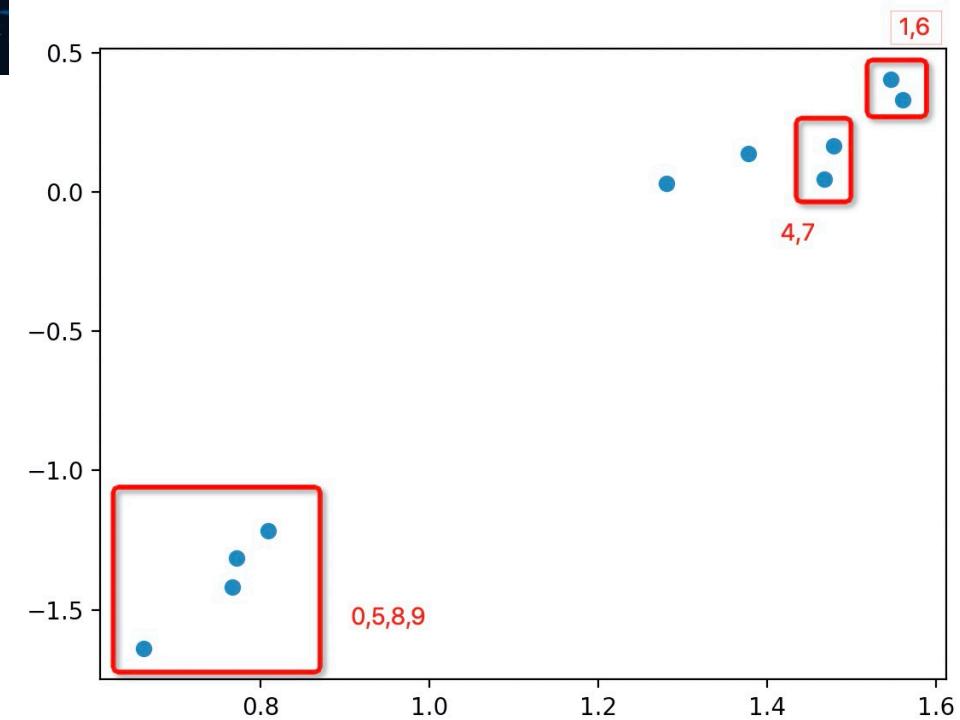
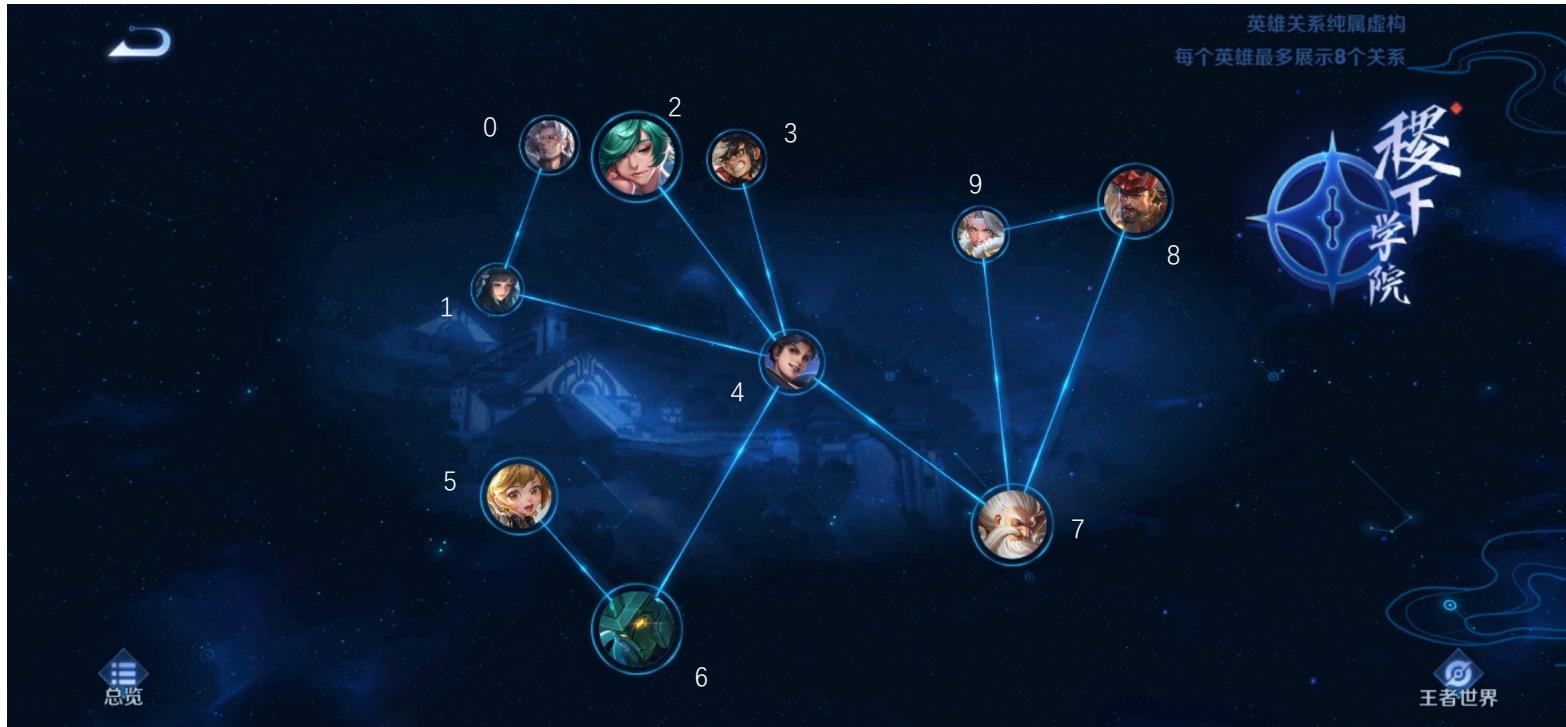
(g) struc2vec (OPT2)



(h) struc2vec (OPT3)

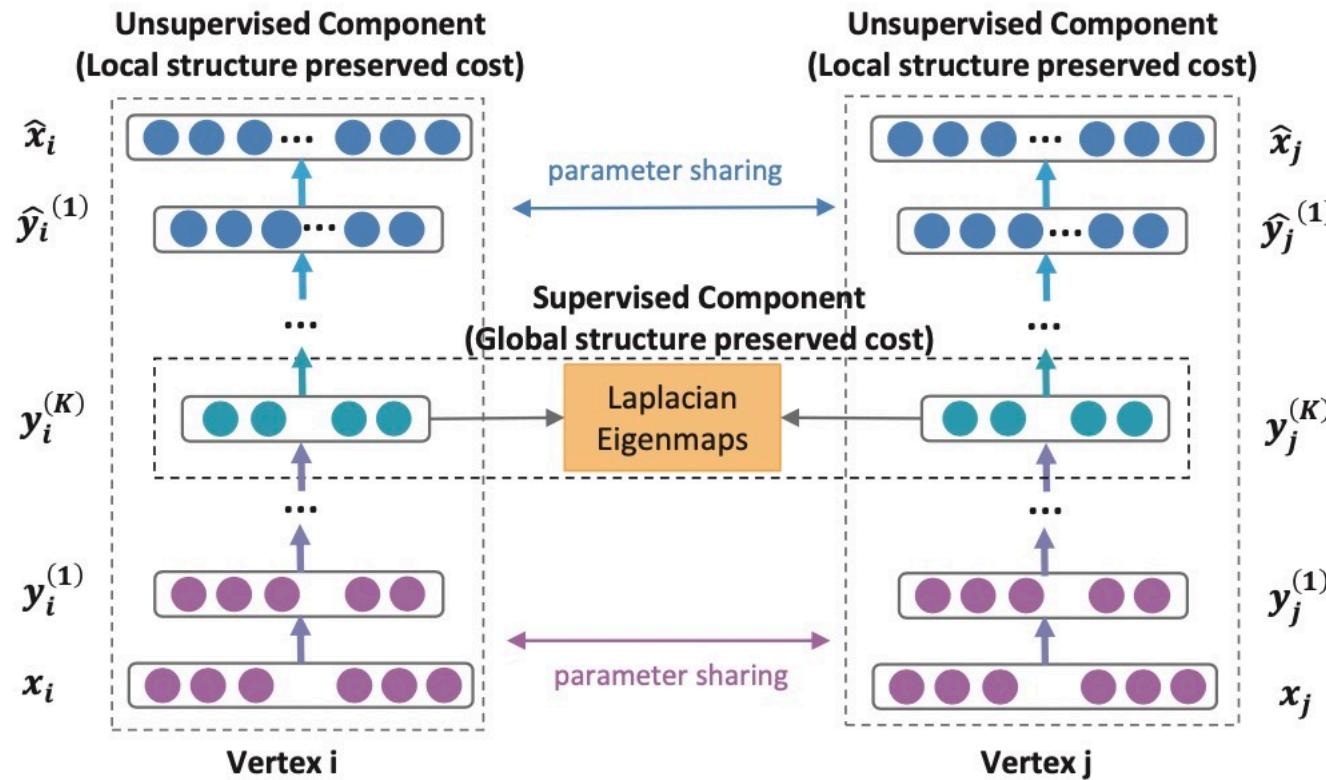
OPT – 优化一些相似性度量

Struc2vec适用于节点分类中，其结构标识比邻居标识更重要。采用Struc2vec效果好



## 5. SDNE : Structural Deep Network Embedding

之前的Deepwalk, LINE, node2vec, struc2vec都使用了浅层的结构,  
浅层模型往往不能捕获高度非线性的网络结构。即产生了SDNE方法,  
使用多个非线性层来捕获node的embedding



$$\begin{aligned}\mathcal{L}_{2nd} &= \sum_{i=1}^n \|(\hat{\mathbf{x}}_i - \mathbf{x}_i) \odot \mathbf{b}_i\|_2^2 \\ &= \|(\hat{\mathbf{X}} - \mathbf{X}) \odot \mathbf{B}\|_F^2\end{aligned}\quad (3)$$

where  $\odot$  means the Hadamard product,  $\mathbf{b}_i = \{b_{i,j}\}_{j=1}^n$ . If  $s_{i,j} = 0$ ,  $b_{i,j} = 1$ , else  $b_{i,j} = \beta > 1$ . Now by using the revised deep au-

$$\begin{aligned}\mathcal{L}_{1st} &= \sum_{i,j=1}^n s_{i,j} \|\mathbf{y}_i^{(K)} - \mathbf{y}_j^{(K)}\|_2^2 \\ &= \sum_{i,j=1}^n s_{i,j} \|\mathbf{y}_i - \mathbf{y}_j\|_2^2\end{aligned}\quad (4)$$

$x_i$  – 所有和  $i$  相连接的节点  $(1, n)$

根据邻居，二阶信息来生成embedding  
再考虑相连接的节点，引入一阶信息

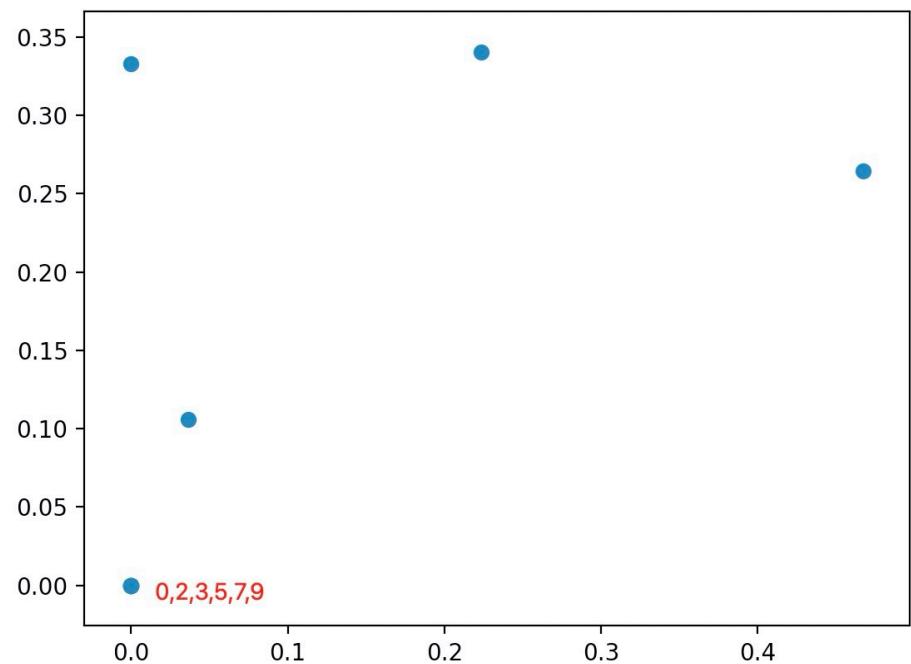
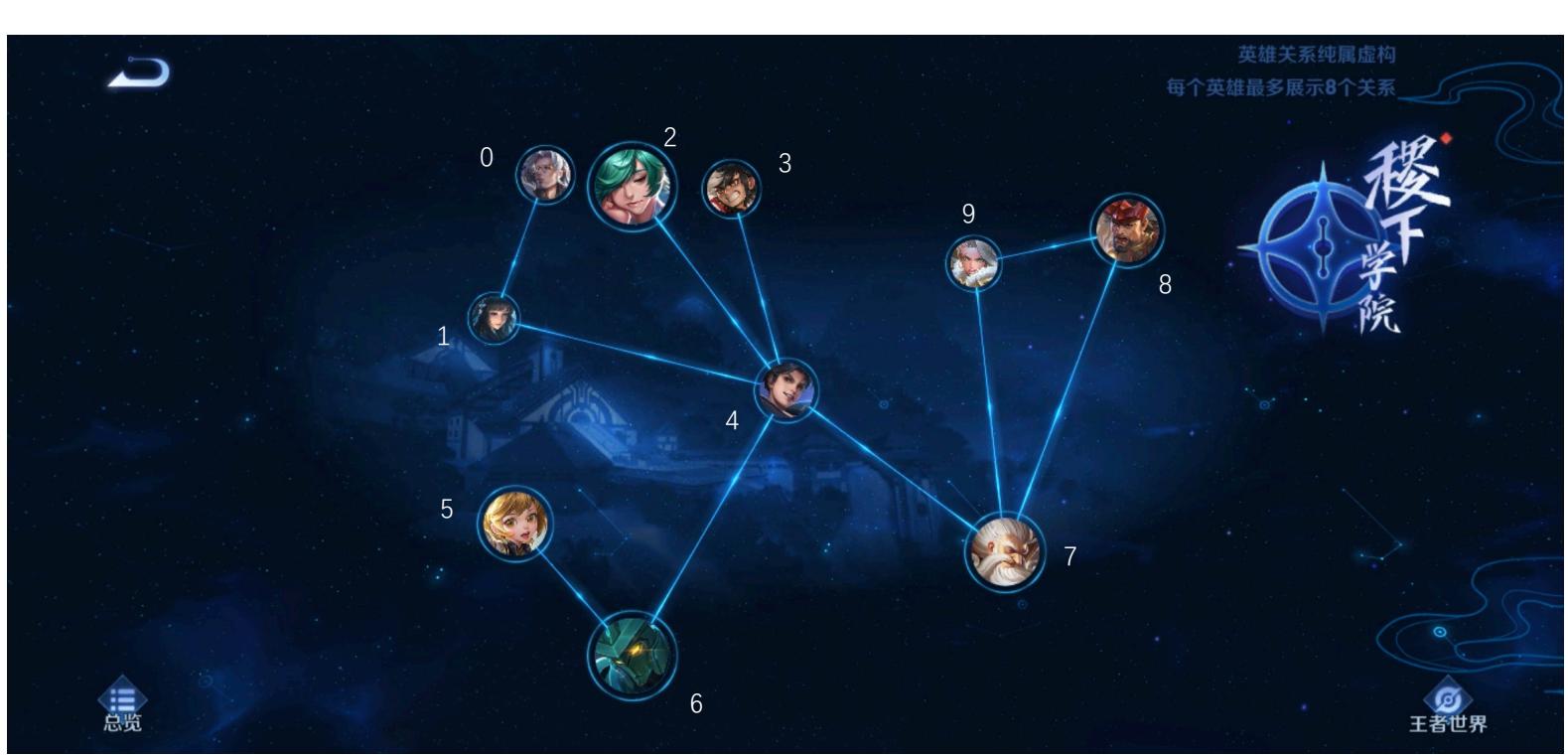
二阶相似性：一对节点邻域有多相似

The second-order proximity refers to how similar the neighborhood structure of a pair of vertexes is. Thus, to model the s

$$\begin{aligned}\mathcal{L}_{mix} &= \mathcal{L}_{2nd} + \alpha \mathcal{L}_{1st} + \nu \mathcal{L}_{reg} \\ &= \|(\hat{X} - X) \odot B\|_F^2 + \alpha \sum_{i,j=1}^n s_{i,j} \|\mathbf{y}_i - \mathbf{y}_j\|_2^2 + \nu \mathcal{L}_{reg} \quad (5)\end{aligned}$$

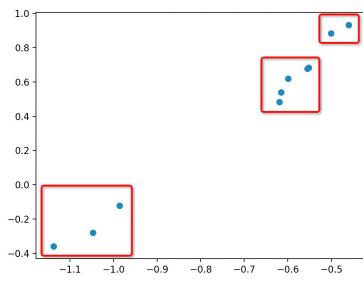
where  $\mathcal{L}_{reg}$  is an L2-norm regularizer term to prevent overfitting, which is defined as follows:

$$\mathcal{L}_{reg} = \frac{1}{2} \sum_{k=1}^K (\|W^{(k)}\|_F^2 + \|\hat{W}^{(k)}\|_F^2) \quad K \text{层的encoder和decoder的参数}$$

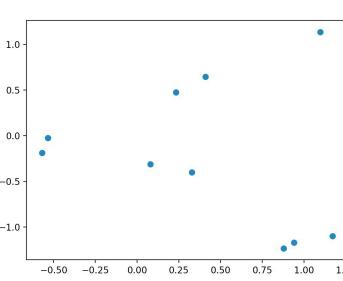


## Graph Embedding:

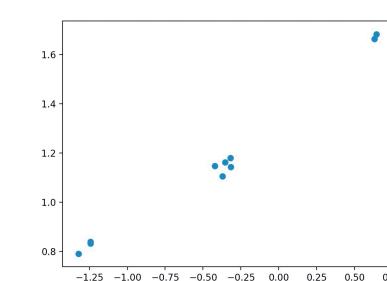
1. DeepWalk : 采用随机游走，形成序列，采用skip-gram方式生成节点embedding。
2. node2vec : 不同的随机游走策略，形成序列，类似skip-gram方式生成节点embedding。
3. LINE : 捕获节点的一阶和二阶相似度，分别求解，再将一阶二阶拼接在一起，作为节点的embedding
4. struc2vec : 对图的结构信息进行捕获，在其结构重要性大于邻居重要性时，有较好的效果。
5. SDNE : 采用了多个非线性层的方式捕获一阶二阶的相似性。



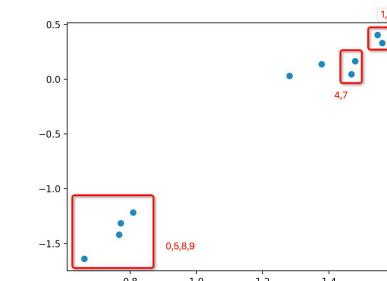
DeepWalk



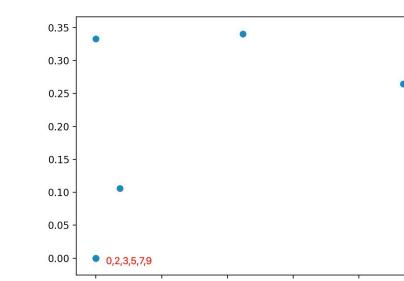
LINE



Node2vec



Struc2vec



SDNE

### 3. Graph Neural Network

GCN

GraphSAGE

GAT

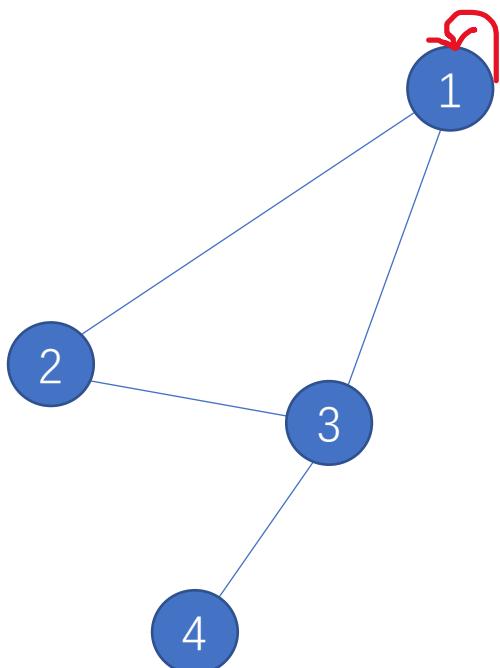
GCN:

In this section, we provide theoretical motivation for a specific graph-based neural network model  $f(X, A)$  that we will use in the rest of this paper. We consider a multi-layer Graph Convolutional Network (GCN) with the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}\right). \quad (2)$$

$\tilde{A} = A + I_N$  is the adjacency matrix of the undirected graph  $\mathcal{G}$  with added self-connections.

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$



$$A : [[0. 1. 1. 0.]  
[1. 0. 1. 0.]  
[1. 1. 0. 1.]  
[0. 0. 1. 0.]]$$

$$\tilde{A} = A + I_N :$$

$$[[1. 1. 1. 0.]  
[1. 1. 1. 0.]  
[1. 1. 1. 1.]  
[0. 0. 1. 1.]]$$

$$\tilde{D} : [[3. 0. 0. 0.]  
[0. 3. 0. 0.]  
[0. 0. 4. 0.]  
[0. 0. 0. 2.]]$$

$$\tilde{D}^{-\frac{1}{2}} :$$

$$[[0.57735027 0. 0. 0.]  
[0. 0.57735027 0. 0.]  
[0. 0. 0.5 0.]  
[0. 0. 0. 0.70710678]]$$

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}\right)$$

$$\tilde{A} = A + I_N :$$

$$\tilde{D}^{-\frac{1}{2}} :$$

$$\begin{array}{c} [[1. 1. 1. 0. ] \\ [1. 1. 1. 0. ] \\ [1. 1. 1. 1. ] \\ [0. 0. 1. 1. ]] \end{array} \quad \begin{array}{c} [[0.57735027 0. 0. 0. ] \\ [0. 0.57735027 0. 0. ] \\ [0. 0. 0.5 0. ] \\ [0. 0. 0. 0.70710678]] \end{array}$$

$$\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} :$$

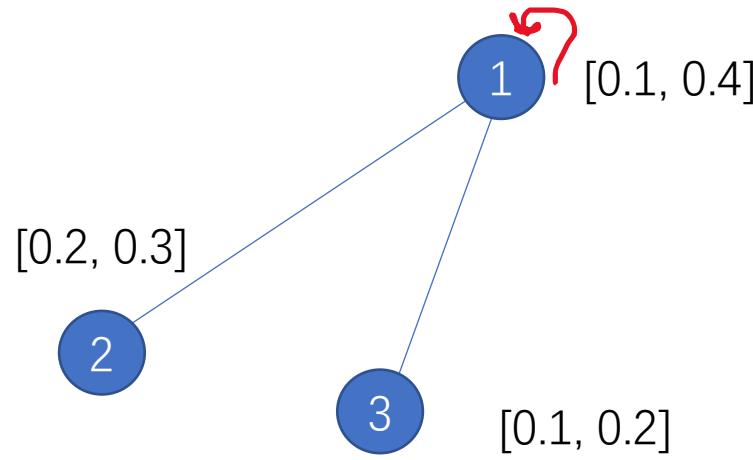
$$\begin{array}{c} [[0.57735027 0. 0. 0. ] \\ [0. 0.57735027 0. 0. ] \\ [0. 0. 0.5 0. ] \\ [0. 0. 0. 0.70710678]] \end{array} \bullet \begin{array}{c} [[1. 1. 1. 0. ] \\ [1. 1. 1. 0. ] \\ [1. 1. 1. 1. ] \\ [0. 0. 1. 1. ]] \end{array} \bullet \begin{array}{c} [[0.57735027 0. 0. 0. ] \\ [0. 0.57735027 0. 0. ] \\ [0. 0. 0.5 0. ] \\ [0. 0. 0. 0.70710678]] \end{array}$$

=

$$\begin{array}{c} [[0.33333333 0.33333333 0.28867513 0. ] \\ [0.33333333 0.33333333 0.28867513 0. ] \\ [0.28867513 0.28867513 0.25 0.35355339] \\ [0. 0. 0.35355339 0.5 ]] \end{array}$$

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}\right)$$

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



$$\begin{aligned} & [[0, 1, 1] \quad [[0.1, 0.4] \quad [[0.2+0.1, 0.3+0.2] \\ & [1, 0, 0] * [0.2, 0.3] = [0.1 \quad , 0.4 \quad ] \\ & [1, 0, 0]] \quad [0.1, 0.2]] \quad [0.1 \quad , 0.4 \quad ]] \end{aligned}$$

$$H^{(l+1)} = \sigma(\tilde{A}H^{(l)}W^{(l)})$$

$$A = [[0, 1, 1] \quad [1, 0, 0] \quad [1, 0, 0]]$$

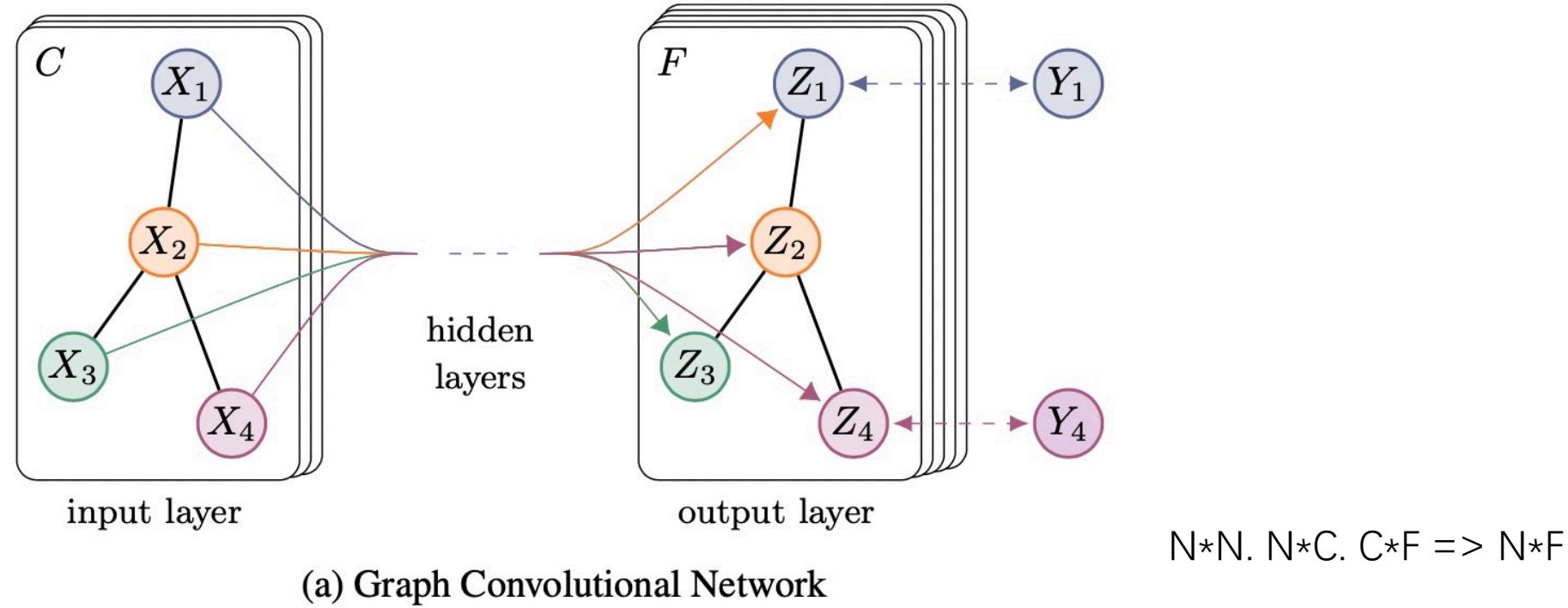
$$\tilde{A} = [[1, 1, 1] \quad [1, 1, 0] \quad [1, 0, 1]]$$

$$\begin{aligned} & [[1, 1, 1] \quad [[0.1, 0.4] \quad [[0.1+0.2+0.1, 0.4+0.3+0.2] \\ & [1, 1, 0] * [0.2, 0.3] = [0.1+0.2 \quad , 0.4+0.3 \quad ] \\ & [1, 0, 1]] \quad [0.1, 0.2]] \quad [0.1+0.1 \quad , 0.4+0.2 \quad ]] \end{aligned}$$

$$X = [[0.1, 0.1] \quad [0.2, 0.3] \quad [0.1, 0.2]]$$

自连接将自身的节点考虑到计算中，将节点本身和邻居的信息都考虑进来  
连接后，对节点进行归一化操作，故左右各乘以  $\tilde{D}^{-1/2}$

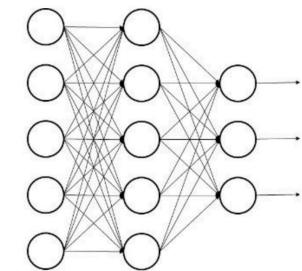
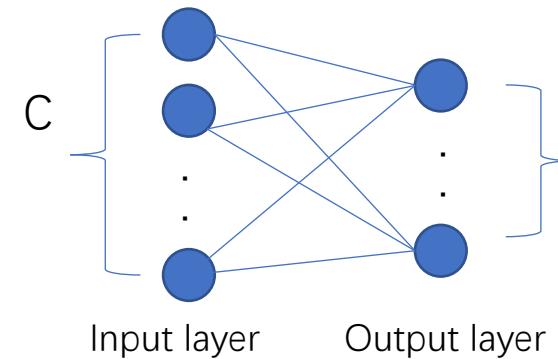
$$Z = f(X, A) = \text{softmax}\left(\hat{A} \text{ ReLU}\left(\hat{A}XW^{(0)}\right) W^{(1)}\right). \quad (9)$$



C 原始特征的维度， 经过训练参数W，  $(N, C) * (C, F) \Rightarrow (N, F)$ 维的特征

如何转换成类别：

1. 最后一层GCN的隐层特征数等于类别数，直接使用softmax输出概率
2. 在输出的F维特征后，在接一个全连接层，训练标签



GraphSAGE : GraphSAGE (SAmple and aggreGatE)

## 采样和聚合

提出归纳式的graph embedding方法，之前的graph embedding方法都是所有节点都在图中，对于没有看到过的节点是不能处理的，这种叫做直推式方法。

而GraphSAGE这种归纳式的方法，可以对于没见过的节点也生成embedding。

GraphSAGE不仅限于构建embedding，也通过聚合周围邻居节点的特征。

归纳式：节点的局部信息和图的全局信息

learning algorithm, we simultaneously learn the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood. While we focus on feature-rich

as well as the distribution of node features in the neighborhood. While we focus on feature-rich graphs (e.g., citation data with text attributes, biological data with functional/molecular markers), our approach can also make use of structural features that are present in all graphs (e.g., node degrees). Thus, our algorithm can also be applied to graphs without node features.

### 3.1 Embedding generation (i.e., forward propagation) algorithm

---

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

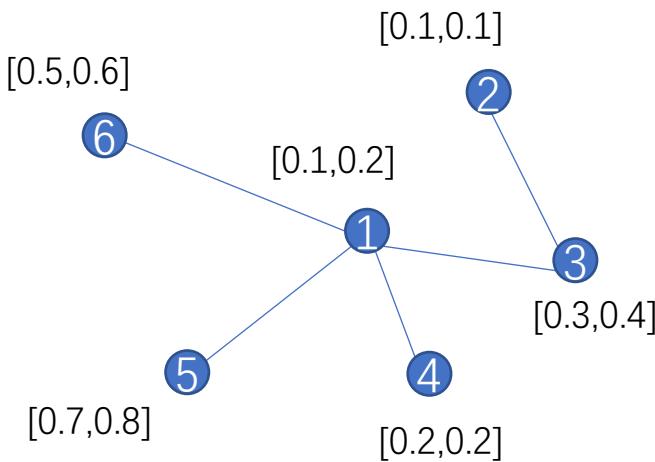
---

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

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

```
1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$ ;  
2 for  $k = 1 \dots K$  do  
3   for  $v \in \mathcal{V}$  do  
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$ ;  
5      $\mathbf{h}_v^k \leftarrow \sigma \left( \mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k) \right)$   
6   end  
7    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$   
8 end  
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 
```

---



节点1, 相连接的节点{3, 4, 5, 6}

$$4. \quad h_{\mathcal{N}(1)}^1 \leftarrow AGGREGATE(\{h_3^0, h_4^0, h_5^0, h_6^0\})$$

$$5. \quad h_1^1 \leftarrow \sigma(W^1 \cdot CONCAT(h_1^0, h_{\mathcal{N}(1)}^1))$$

$$h_{\mathcal{N}(1)}^1 = AGGREGATE(\{h_3^0, h_4^0, h_5^0, h_6^0\}) = Mean([0.3, 0.4], [0.2, 0.2], [0.7, 0.8], [0.5, 0.6])$$

$$h_1^1 = W \cdot CONCAT(h_1^0, h_{\mathcal{N}(1)}^1) = W \cdot [0.1, 0.2, 0.425, 0.5]$$

---

**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  $\mathbf{W}^k, \forall k \in \{1, \dots, K\}$ ; non-linearity  $\sigma$ ; differentiable aggregator functions  $AGGREGATE_k, \forall k \in \{1, \dots, K\}$ ; neighborhood function  $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

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

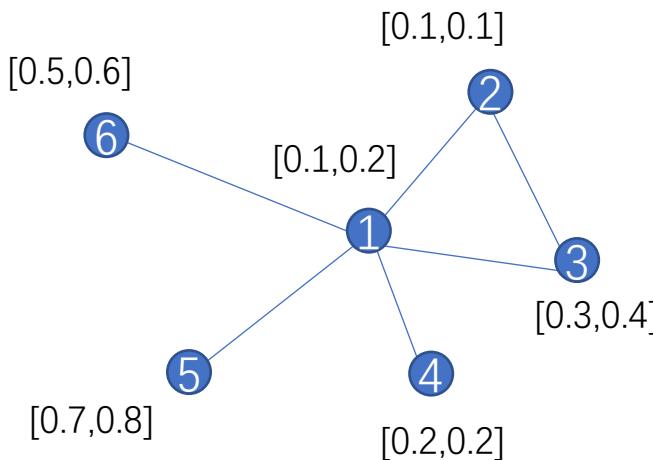
```

1  $\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V};$ 
2 for  $k = 1 \dots K$  do
3   for  $v \in \mathcal{V}$  do
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow AGGREGATE_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\});$ 
5      $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot CONCAT(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$ 
6   end
7    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ 
8 end
9  $\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$ 

```

---

the fact that the random samples are independent across iterations over  $k$ . We use a uniform sampling function in this work and sample with replacement in cases where the sample size is larger than the node's degree.



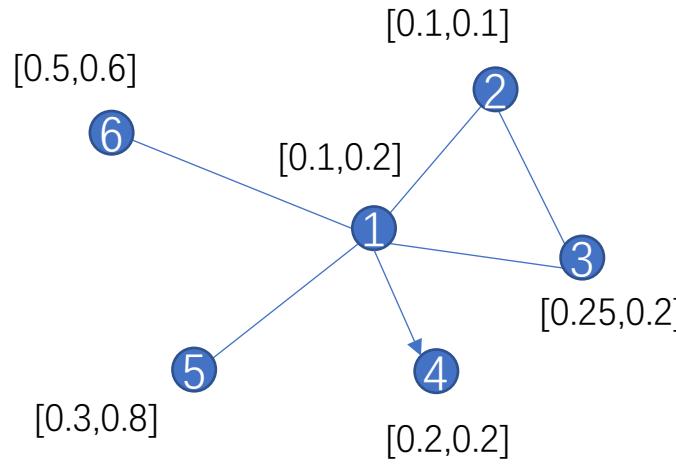
```
if len(neighbors) > self.max_degree:  
    neighbors = np.random.choice(neighbors, self.max_degree, replace=False)  
elif len(neighbors) < self.max_degree:  
    neighbors = np.random.choice(neighbors, self.max_degree, replace=True)
```

## 邻居采样：

the worst case  $O(|\mathcal{V}|)$ . In contrast, the per-batch space and time complexity for GraphSAGE is fixed at  $O(\prod_{i=1}^K S_i)$ , where  $S_i, i \in \{1, \dots, K\}$  and  $K$  are user-specified constants. Practically speaking we found that our approach could achieve high performance with  $K = 2$  and  $S_1 \cdot S_2 \leq 500$  (see Section 4.4 for details).

如果对点的所有邻居进行采样，那么有可能会使得时间复杂度提升。现在是规定采样的大小，即采用又放回的抽样方法，采用固定的抽样大小。作者给出了建议值，即，进行两次传播，第一次采样邻居是S1，第二次S2。

$$S_1 \cdot S_2 < 500$$



对周围节点的3个点采样，求1, 2节点的聚合

$$h_{\mathcal{N}(1)}^1 = \text{Agg}(v6, v5, v2) = [0.3, 0.5]$$

$$h_1^1 = \sigma(W^1 \cdot ([0.1, 0.2, 0.3, 0.5]))$$

$$h_{\mathcal{N}(2)}^1 = \text{Agg}(v1, v3, v3) = [0.2, 0.2]$$

$$h_2^1 = \sigma(W^1 \cdot ([0.1, 0.1, 0.2, 0.2]))$$

```

def construct_adj(self):
    adj = len(self.id2idx)*np.ones((len(self.id2idx)+1, self.max_degree))
    deg = np.zeros((len(self.id2idx),))

    for nodeid in self.G.nodes():
        if self.G.node[nodeid]['test'] or self.G.node[nodeid]['val']:
            continue
        neighbors = np.array([self.id2idx[neighbor]
                             for neighbor in self.G.neighbors(nodeid)
                             if (not self.G[nodeid][neighbor]['train_removed'])])
        deg[self.id2idx[nodeid]] = len(neighbors)
    
```

```

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

```

Aggregator Architectures: 对称的，对于输入排列不变的

Mean aggregator

LSTM aggregator

Pooling aggregator

```
for  $v \in \mathcal{V}$  do
     $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\});$ 
     $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k))$ 
```

Mean:  $\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})).$

LSTM本身是有顺序的，但是通过将输入节点随机排列，使得LSTM可以适用于无序的集合。

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

$$\begin{array}{lll} [0.5, 0.6] & \xrightarrow{} & \sigma(\mathbf{W}_{\text{pool}} \mathbf{h}_{u_i}^k + \mathbf{b}) & [0.6, 0] \\ [0.1, 0.2] & \xrightarrow{} & & [0.2, 1] \\ [0.7, 0.8] & \xrightarrow{} & & [0.8, 0.5] \\ [0.1, 0.1] & \xrightarrow{} & \sigma(\mathbf{W}_{\text{pool}} \mathbf{h}_{u_i}^k + \mathbf{b}) & [0.1, 1.2] \end{array}$$

不仅可以使用 $\max$ , 其他的排列不变性的函数都可以, 作者提到 $\text{mean}$ 和 $\max$ 效果上没有差异

---

**Algorithm 2:** GraphSAGE minibatch forward propagation algorithm

---

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

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

```
1  $\mathcal{B}^K \leftarrow \mathcal{B}$ ;  
2 for  $k = K \dots 1$  do  
3    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k$  ;  
4   for  $u \in \mathcal{B}^k$  do  
5      $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u)$ ;  
6   end  
7 end  
8  $\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0$  ;  
9 for  $k = 1 \dots K$  do  
10  for  $u \in \mathcal{B}^k$  do  
11     $\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\})$ ;  
12     $\mathbf{h}_u^k \leftarrow \sigma(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k))$ ;  
13     $\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2$ ;  
14  end  
15 end  
16  $\mathbf{z}_u \leftarrow \mathbf{h}_u^K, \forall u \in \mathcal{B}$ 
```

Minbatch : GraphSAGE采用聚合邻居，  
和GCN使用全图方式，变成采样。这样  
在minbatch下，可以不使用全图信息，  
这使得在大规模图上训练变得可行。

```

1  $\mathcal{B}^K \leftarrow \mathcal{B};$ 
2 for  $k = K \dots 1$  do
3    $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^k;$ 
4   for  $u \in \mathcal{B}^k$  do
5      $\mathcal{B}^{k-1} \leftarrow \mathcal{B}^{k-1} \cup \mathcal{N}_k(u);$ 
6   end
7 end

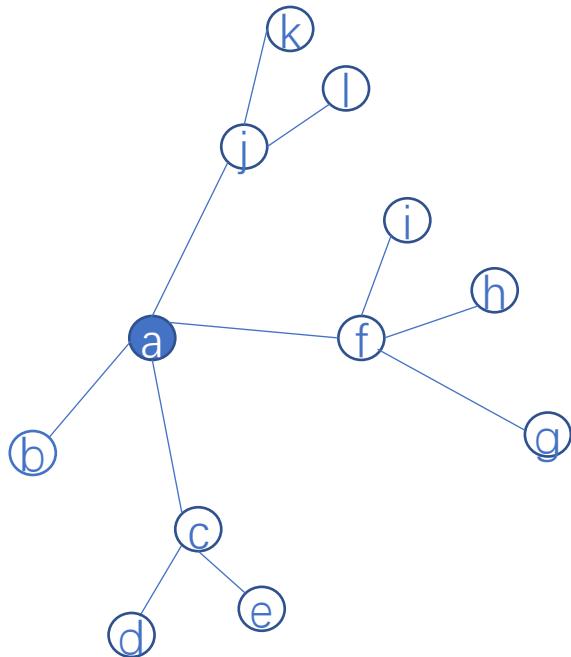
```

$$K = 2, S1 = 2, S2 = 3$$

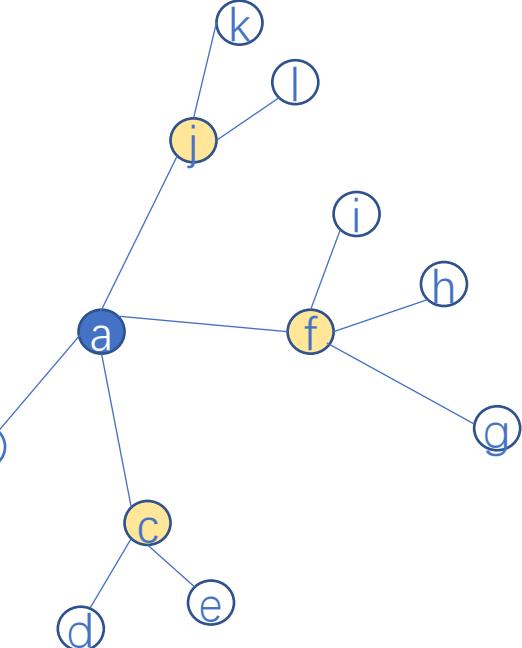
$$\mathcal{B}^2 = \{a\}$$

$$\mathcal{B}^1 = \{a\} \cup \mathcal{N}_2(a) = \{a\} \cup \{c, f, j\}$$

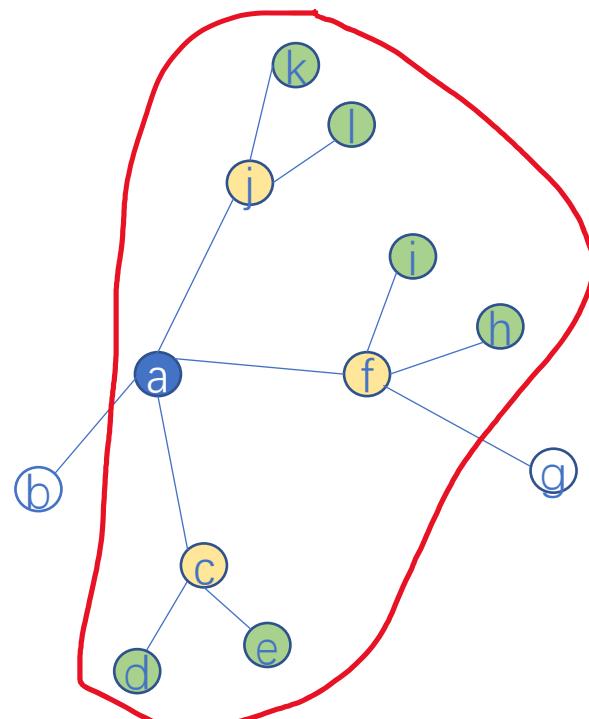
$$\mathcal{B}^0 = \{a\} \cup \{c, f, j\} \cup \mathcal{N}_1(\{c, f, j\}) = \{a\} \cup \{c, f, j\} \cup \{d, e, i, h, k, l\}$$



$\mathcal{B}^2$



$\mathcal{B}^1$



$\mathcal{B}^0$

$$\mathbf{h}_u^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{B}^0 ;$$

**for**  $k = 1 \dots K$  **do**

**for**  $u \in \mathcal{B}^k$  **do**

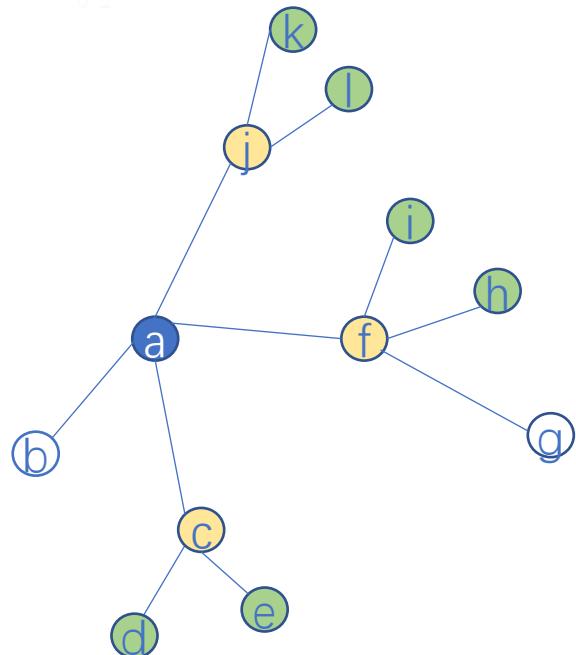
$$~~~~~\mathbf{h}_{\mathcal{N}(u)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_{u'}^{k-1}, \forall u' \in \mathcal{N}_k(u)\});$$

$$~~~~~\mathbf{h}_u^k \leftarrow \sigma \left( \mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_u^{k-1}, \mathbf{h}_{\mathcal{N}(u)}^k) \right);$$

$$~~~~~\mathbf{h}_u^k \leftarrow \mathbf{h}_u^k / \|\mathbf{h}_u^k\|_2;$$

**end**

**end**



$$\mathcal{B}^0$$

$$\mathcal{B}^1 = \{a\} \cup \mathcal{N}_2(a) = \{a\} \cup \{c, f, j\}$$

$$\mathcal{N}_1(c) = \{d, e\}$$

$$h_{\mathcal{N}(c)}^1 \leftarrow \text{AGGREGATE}_1\{h_d^0, h_e^0\}$$

$$h_c^1 \leftarrow \sigma(W^1 \cdot \text{CONCAT}(h_c^0, h_{\mathcal{N}(1)}^1))$$

.

$\bar{\mathbf{W}}^k$ ,  $\forall k \in \{1, \dots, K\}$ , and parameters of the aggregator functions via stochastic gradient descent. The graph-based loss function encourages nearby nodes to have similar representations, while enforcing that the representations of disparate nodes are highly distinct:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log (\sigma(\mathbf{z}_u^\top \mathbf{z}_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log (\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})), \quad (1)$$

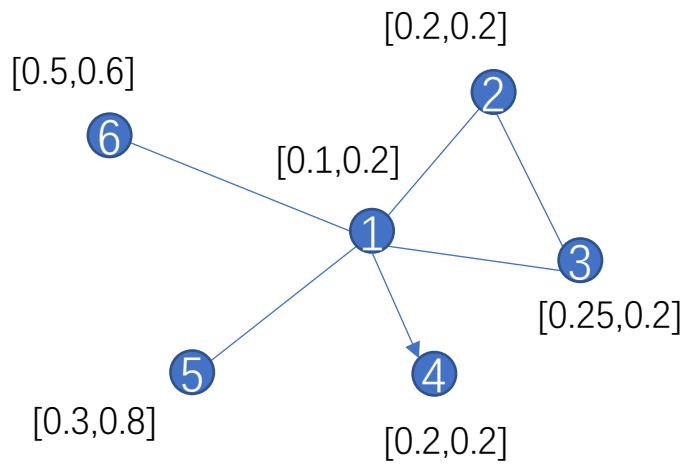
训练的是GraphSAGE的权重函数W, 经过GraphSAGE后, 生成的embedding, Zu, 求loss,  
而不是直接生成节点的embedding

## GAT : GRAPH ATTENTION NETWORKS

$$\alpha_{ij} = \frac{\exp \left( \text{LeakyReLU} \left( \vec{\mathbf{a}}^T [\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_j] \right) \right)}{\sum_{k \in \mathcal{N}_i} \exp \left( \text{LeakyReLU} \left( \vec{\mathbf{a}}^T [\mathbf{W}\vec{h}_i \| \mathbf{W}\vec{h}_k] \right) \right)} \quad (3)$$

where  $\cdot^T$  represents transposition and  $\|$  is the concatenation operation.

$$\vec{h}'_i = \sigma \left( \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W}\vec{h}_j \right). \quad (4)$$



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

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

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

$$\begin{aligned} \alpha &= [1, 1, 1, 1] & w &= [[1, 0], [0, 1]] \\ e_{12} &= \alpha \cdot [0.1, 0.2, 0.2, 0.2] = 0.7 \\ e_{13} &= \alpha \cdot [0.1, 0.2, 0.25, 0.2] = 0.75 \\ e_{14} &= 0 \\ e_{15} &= \alpha \cdot [0.1, 0.2, 0.3, 0.8] = 1.4 \\ e_{16} &= \alpha \cdot [0.1, 0.2, 0.5, 0.6] = 1.4 \end{aligned}$$

$$\alpha_{12} = \frac{\exp(\text{LeakReLU}(e_{12}))}{\exp(\text{LeakReLU}(e_{12})) + \dots + \exp(\text{LeakReLU}(e_{16}))}$$

$$\vec{h}'_1 = \sigma(\alpha_{12} \cdot W \cdot \vec{h}_2 + \alpha_{13} \cdot W \cdot \vec{h}_3 \dots) = \sigma(\alpha_{12} \cdot W \cdot [0.2, 0.2] + \dots)$$

## 多头注意力机制

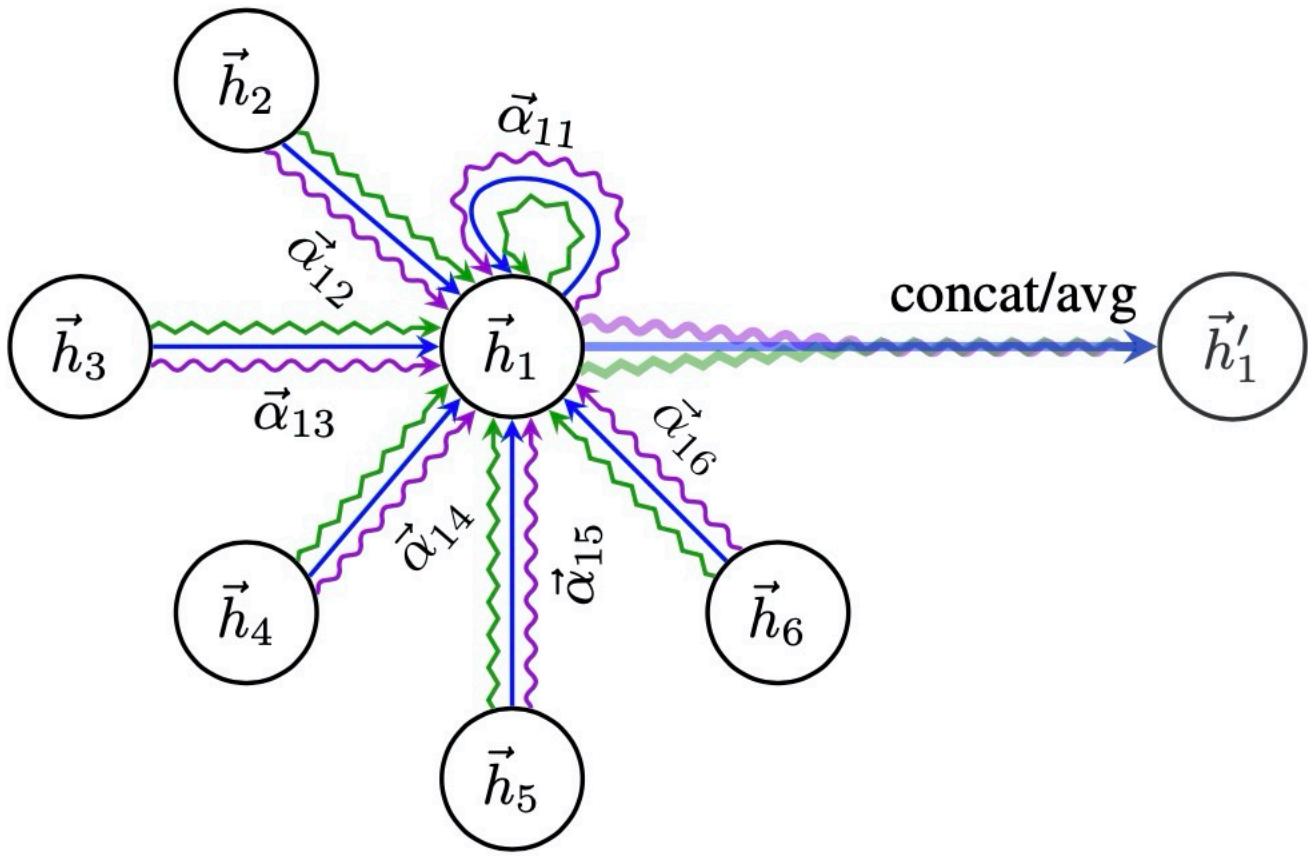
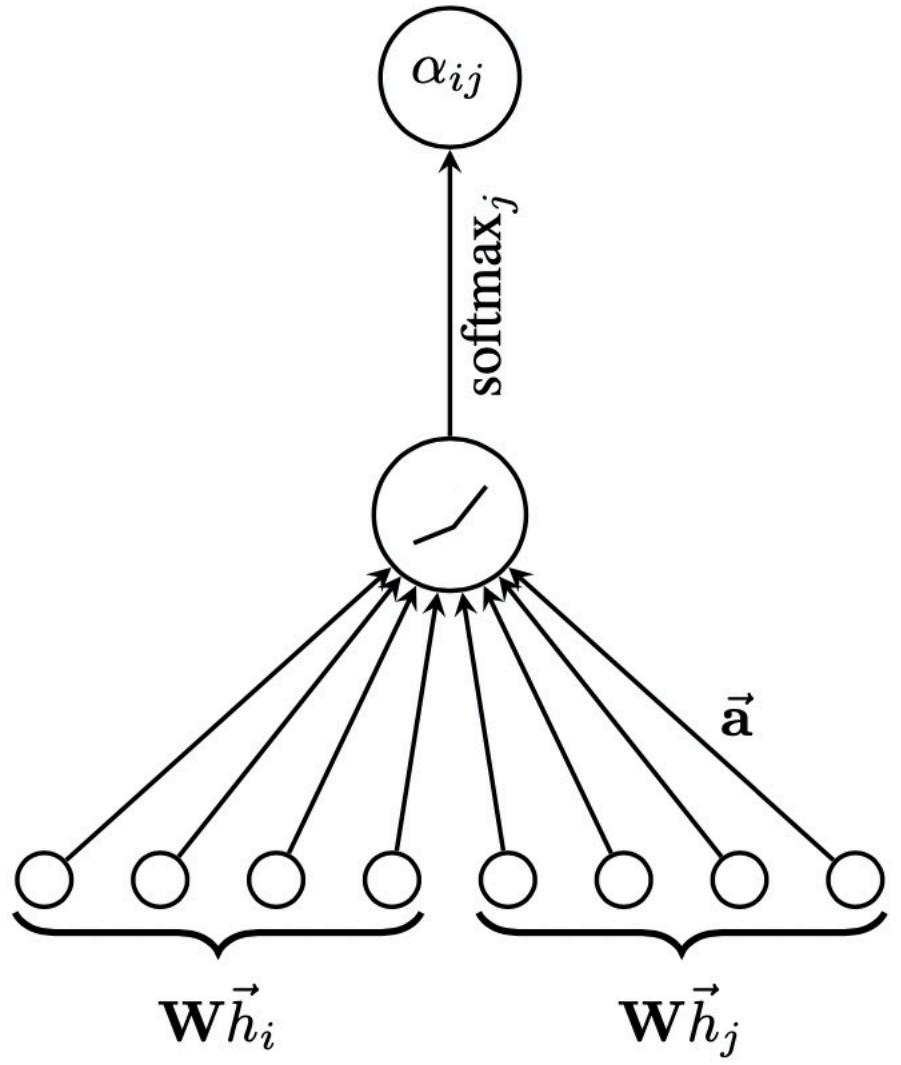
To stabilize the learning process of self-attention, we have found extending our mechanism to employ *multi-head attention* to be beneficial, similarly to Vaswani et al. (2017). Specifically,  $K$  independent attention mechanisms execute the transformation of Equation 4, and then their features are concatenated, resulting in the following output feature representation:

$$\vec{h}'_i = \parallel_{k=1}^K \sigma \left( \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j \right) \quad (5)$$

Specially, if we perform multi-head attention on the final (prediction) layer of the network, concatenation is no longer sensible—instead, we employ *averaging*, and delay applying the final nonlinearity (usually a softmax or logistic sigmoid for classification problems) until then:

$$\vec{h}'_i = \sigma \left( \frac{1}{K} \sum_{k=1}^K \sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \vec{h}_j \right) \quad (6)$$

如果是最后预测层，则这种cocat方式是没有必要的，而是要平均值。这里指的是直接连接softmax的方式，如果接一个全连接层，是无所谓的



*Transductive*

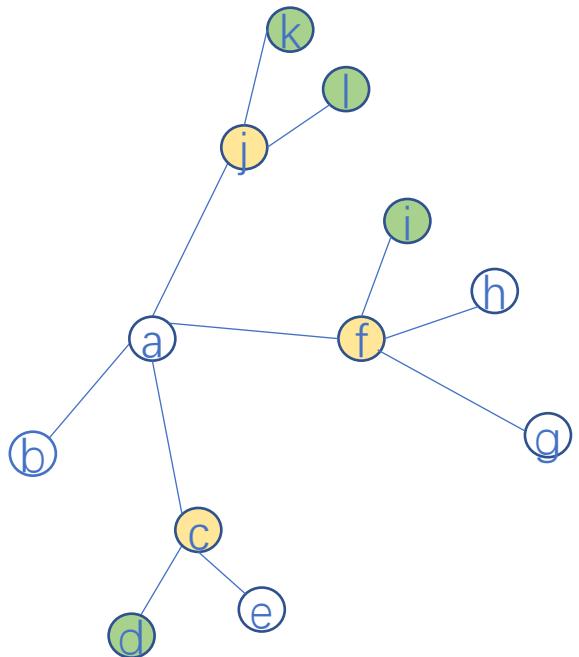
Method	Cora	Citeseer	Pubmed
MLP	55.1%	46.5%	71.4%
ManiReg (Belkin et al., 2006)	59.5%	60.1%	70.7%
SemiEmb (Weston et al., 2012)	59.0%	59.6%	71.7%
LP (Zhu et al., 2003)	68.0%	45.3%	63.0%
DeepWalk (Perozzi et al., 2014)	67.2%	43.2%	65.3%
ICA (Lu & Getoor, 2003)	75.1%	69.1%	73.9%
Planetoid (Yang et al., 2016)	75.7%	64.7%	77.2%
Chebyshev (Defferrard et al., 2016)	81.2%	69.8%	74.4%
GCN (Kipf & Welling, 2017)	81.5%	70.3%	<b>79.0%</b>
MoNet (Monti et al., 2016)	81.7 ± 0.5%	—	78.8 ± 0.3%
GCN-64*	81.4 ± 0.5%	70.9 ± 0.5%	<b>79.0 ± 0.3%</b>
<b>GAT (ours)</b>	<b>83.0 ± 0.7%</b>	<b>72.5 ± 0.7%</b>	<b>79.0 ± 0.3%</b>

*Inductive*

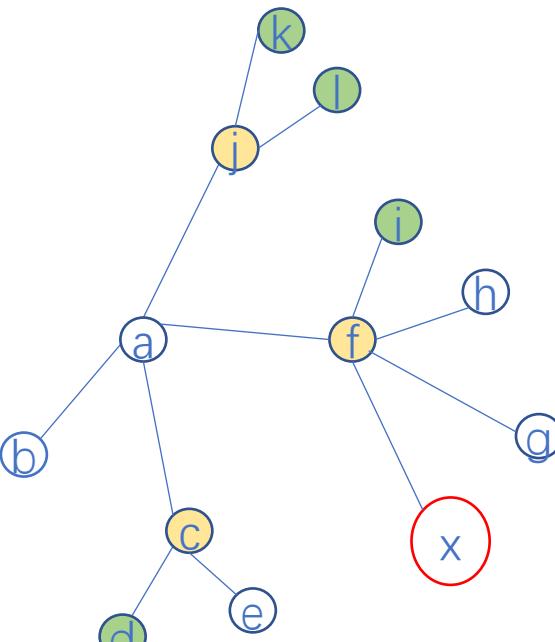
Method	PPI
Random	0.396
MLP	0.422
GraphSAGE-GCN (Hamilton et al., 2017)	0.500
GraphSAGE-mean (Hamilton et al., 2017)	0.598
GraphSAGE-LSTM (Hamilton et al., 2017)	0.612
GraphSAGE-pool (Hamilton et al., 2017)	0.600
GraphSAGE*	0.768
Const-GAT (ours)	$0.934 \pm 0.006$
<b>GAT (ours)</b>	<b><math>0.973 \pm 0.002</math></b>

## 图网络应用

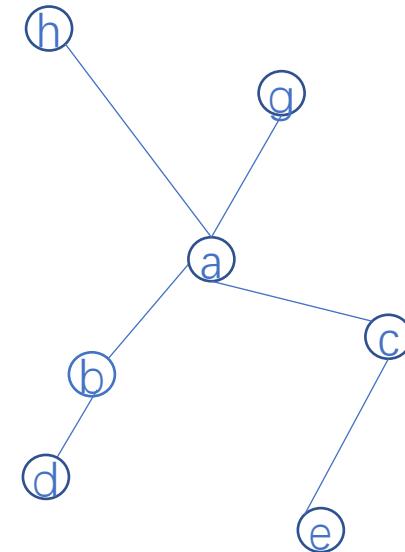
1. GCN, GraphSAGE, GAT, 节点的半监督学习
2. GraphSAGE, GAT的推理学习任务



半监督



推理学习



代码詳解

$$Z = f(X, A) = \text{softmax}\left(\hat{A} \text{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right). \quad (9)$$

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right). \quad (2)$$

$\tilde{A} = A + I_N$  is the adjacency matrix of the undirected graph  $\mathcal{G}$  with added self-connections.

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$$

1. 为何要归一化?

采用加法规则时, 对于度大的节点特征越来越大, 而对于度小的节点却相反, 这可能导致网络训练过程中梯度爆炸或者消失的问题。

2. 如何归一化?

$$\hat{L}_{sym} = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$$

$$\hat{A} = A + I$$

$$\hat{D} = D + I$$

简单的归一化方法:  $\hat{D}^{-1}\hat{A}$

但是这样得到的矩阵是非对称阵

所以使用  $\hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}$

## nn.CrossEntropyLoss() 与 NLLLoss()

NLLLoss 的 输入 是一个对数概率向量和一个目标标签. 它不会为我们计算对数概率. 适合网络的最后一层是log\_softmax. 损失函数 nn.CrossEntropyLoss() 与 NLLLoss() 相同, 唯一的不同是它为我们去做 softmax.

1 | `CrossEntropyLoss()=log_softmax() + NLLLoss()`

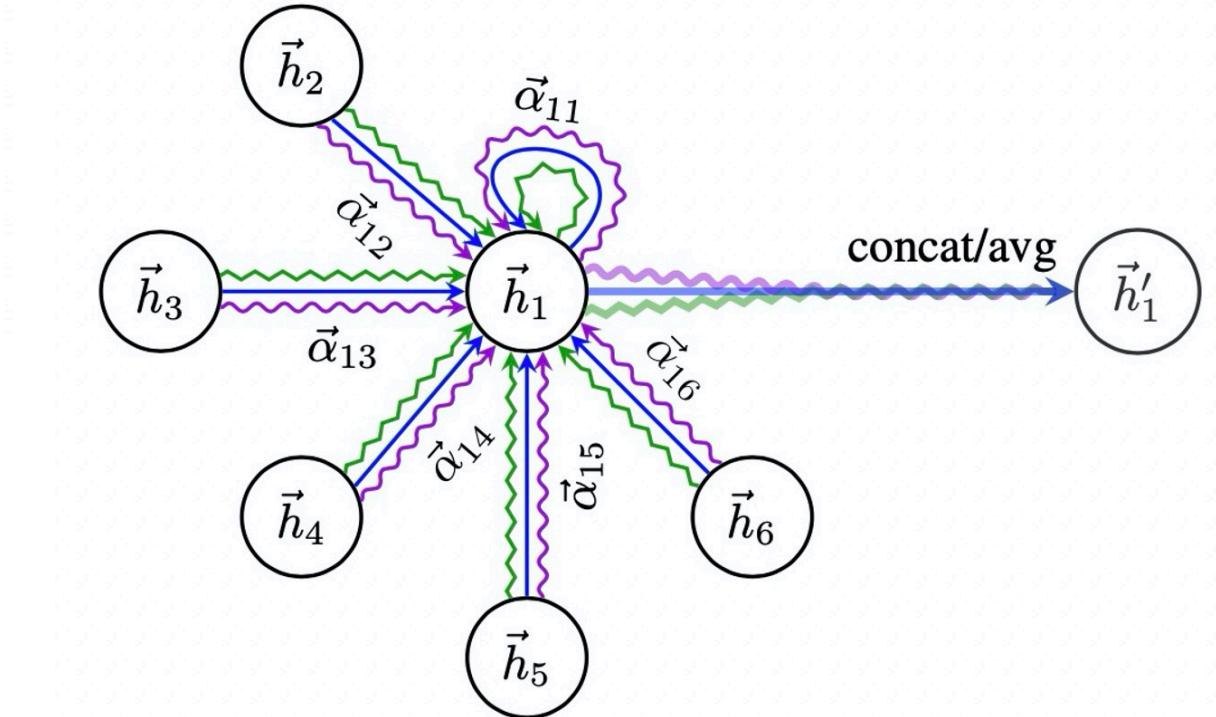
GAT

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

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

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

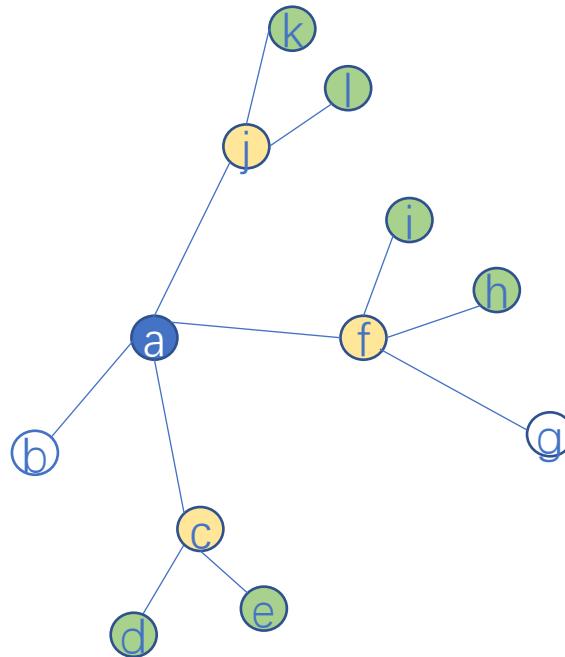
$$\vec{h}'_i = \sigma \left( \sum_{j \in \mathcal{N}_i} \alpha_{ij} \mathbf{W} \vec{h}_j \right).$$



## GraphSAGE

1. 聚合周围邻居节点的信息

```
nodes_batch_layers = [Layer1, Layer0, Layer_center]
```



```
2 for  $k = 1 \dots K$  do  
3   for  $v \in \mathcal{V}$  do  
4      $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}$   
5      $\mathbf{h}_v^k \leftarrow \sigma \left( \mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k) \right)$   
6   end  
7    $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$ 
```

$\mathbf{W} = [128, 1433*2]. \text{layer1}$   
 $[128, 256]. \text{layer2}$

# 图网络的分类

1. Recurrent Graph Neural Networks
2. Convolution Graph Neural Networks
3. Graph Autoencoders
4. Spatial-temporal Graph Neural Networks

— A Comprehensive Survey on Graph Neural Networks

1. Graph Convolution Networks
2. Graph Attention Networks
3. Graph Auto-encoder
4. Graph Generative Networks (图生成网络)
5. Graph Spatial-Temporal Networks (时空网络)

— A Comprehensive Survey on Graph Neural Networks

1. Recurrent Graph Neural Networks
2. Convolution Graph Neural Networks
3. Graph Autoencoders
4. Graph Reinforcement Learning
5. Graph Adversarial Methods

— Deep Learning on Graphs: A Survey

更通用的框架，更一般的框架分类方式：

1. MPNN：图神经网络和图卷积/ *Message Passing Neural Networks*
2. NLNN：统一Attention/ *Non-local Neural Networks*
3. GN：统一以上/ *Graph Networks*

图计算任务框架：

1. 节点问题
2. 边：将两个节点作为输入，输入边的预测
3. 图：图分类/图匹配/图生成

图上任务类别：

1. 半监督任务：Node
2. 有监督：Graph/Node
3. 无监督：Graph embedding

图的类别

1. 异构图
2. 二部图
3. 多维图
4. 符号图
5. 超图
6. 动态图

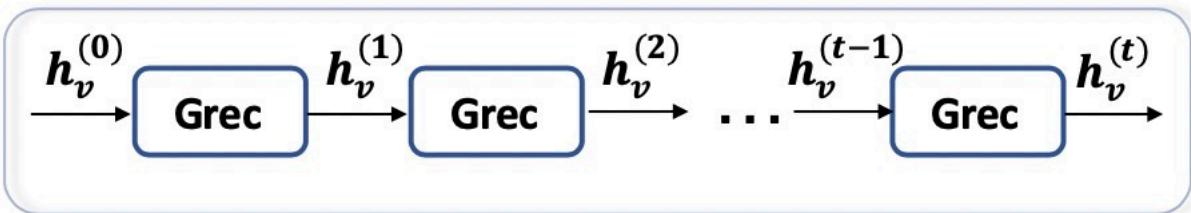
DataSet :

1. 引文网络：Cora/Citesser/Pubmed
2. PPI蛋白质

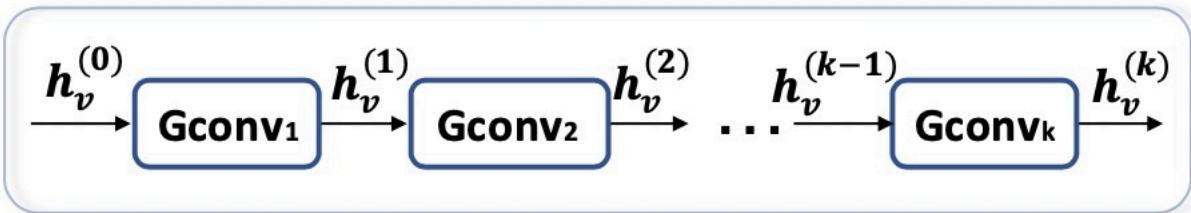
## 1. Recurrent Graph Neural Networks

$$\mathbf{h}_v^{(t)} = GRU(\mathbf{h}_v^{(t-1)}, \sum_{u \in N(v)} \mathbf{W} \mathbf{h}_u^{(t-1)}),$$

Gated Graph Neural Network (GGNN)



(a) Recurrent Graph Neural Networks (RecGNNs). RecGNNs use the same graph recurrent layer (Grec) in updating node representations.



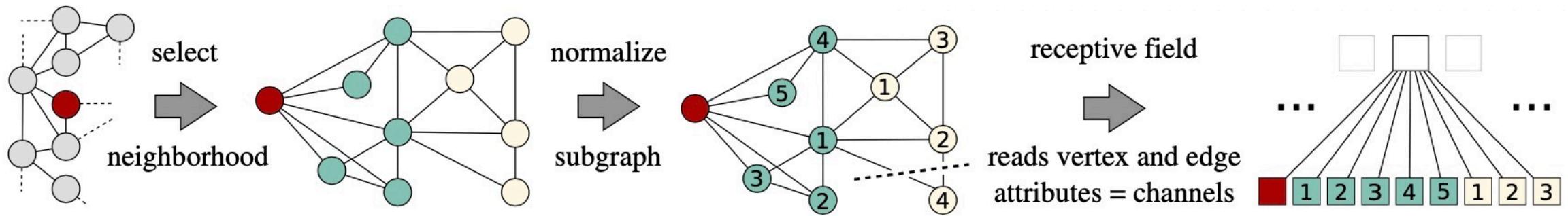
(b) Convolutional Graph Neural Networks (ConvGNNs). ConvGNNs use a different graph convolutional layer (Gconv) in updating node representations.

Fig. 3: RecGNNs v.s. ConvGNNs

## 2. Graph Convolution Networks

GCN , GraphSAGE

PATCH-SAN :



要选择的节点，红色节点

绿色节点表示对红色节点的一阶邻居节点，黄色表示是二阶节点。

每个邻居节点的选择顺序是根据节点的度来定义的，按照一阶度，二阶度的顺序对节点进行排序。

这样变形成了一个有顺序的序列，采用CNN的方式进行训练即可。

---

### Algorithm 1 SELNODESEQ: Select Node Sequence

---

```
1: input: graph labeling procedure  $\ell$ , graph  $G = (V, E)$ , stride  $s$ , width  $w$ , receptive field size  $k$ 
2:  $V_{\text{sort}} = \text{top } w \text{ elements of } V \text{ according to } \ell$ 
3:  $i = 1, j = 1$ 
4: while  $j < w$  do
5:   if  $i \leq |V_{\text{sort}}|$  then
6:      $f = \text{RECEPTIVEFIELD}(V_{\text{sort}}[i])$ 
7:   else
8:      $f = \text{ZERORECEPTIVEFIELD}()$ 
9:   apply  $f$  to each input channel
10:   $i = i + s, j = j + 1$ 
```

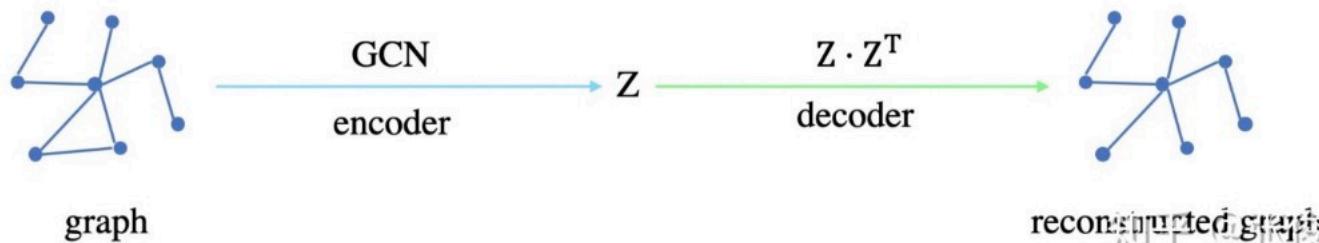
---

### 3. Graph Attention Networks

GAT

## 4. Graph Auto-encoder

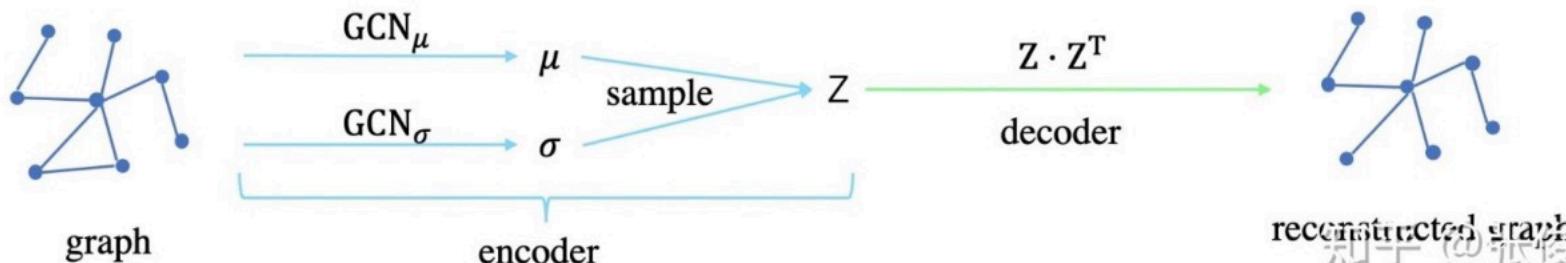
### GAE (Auto-encoder)



**Non-probabilistic graph auto-encoder (GAE) model** For a non-probabilistic variant of the VGAE model, we calculate embeddings  $Z$  and the reconstructed adjacency matrix  $\hat{A}$  as follows:

$$\hat{A} = \sigma(ZZ^T), \text{ with } Z = \text{GCN}(X, A). \quad (4)$$

### VGAE (VAE)



第一层GCN共享参数

$$GCN_0(X, A)$$

第二层GCN学习 $\mu, \sigma$ 参数

$$GCN_\mu(X, A)$$

$$GCN_\sigma(X, A)$$

## 5. Graph Spatial-Temporal Networks（时空网络）

同时考虑图的空间性和时间维度。比如在交通邻域中，速度传感器会随时间变化的时间维度，不同的传感器之间也会形成连接的空间维度的边。

当前的许多方法都应用GCN来捕获图的依赖性，使用一些RNN 或CNN 对时间依赖性建模。

## 6. Graph Generative Networks（图生成网络）

通过RNN或者GAN的方式生成网络。图生成网络的一个有前途的应用领域是化合物合成。在化学图中，原子被视为节点，化学键被视为边。任务是发现具有某些化学和物理性质的新的可合成分子。

## 7. Graph Reinforcement Learning

采用强化学习的方法应用于图网络上。

## 8. Graph Adversarial Methods

GAN的思想，生成器生成样本，分类器去判别样本。

更通用的框架，更一般的框架分类方式：

1. MPNN：图神经网络和图卷积/ *Message Passing Neural Networks*
2. NLNN：统一Attention/ *Non-local Neural Networks*
3. GN：统一以上/ *Graph Networks*

Apart from different variants of graph neural networks, several general frameworks are proposed aiming to integrate different models into one single framework. [27] proposed the message passing neural network (MPNN), which unified various graph neural network and graph convolutional network approaches. [28] proposed the non-local neural network (NLNN). It unifies several “self-attention”-style methods [66], [68], [90]. [30] proposed the graph network (GN) which unified the MPNN and NLNN methods as well as many other variants like Interaction Networks [4], [91],

除了图神经网络的不同变体之外，还提出了一些通用框架，旨在将不同的模型集成到一个框架中。[27]提出了消息传递神经网络（MPNN），它统一了各种图神经网络和图卷积网络方法。[28]提出了非局部神经网络（NLNN）。它统一了几种“自我注意”式的方法[66], [68], [90]。[30]提出了一种图形网络（GN），该网络统一了MPNN和NLNN方法以及许多其他变体，例如交互网络[4], [91]，神经物理引擎[92]，

## MPNN : Message Passing Neural Networks

The forward pass has two phases, a message passing phase and a readout phase

time steps and is defined in terms of message functions  $M_t$  and vertex update functions  $U_t$ . During the message passing phase, hidden states  $h_v^t$  at each node in the graph are updated based on messages  $m_v^{t+1}$  according to

邻域聚合

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \quad (1)$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \quad (2)$$

1. 消息传递阶段
  - 1) 消息函数  $M_t$
  - 2) 更新函数  $U_t$

2. Readout阶段  
保证节点排列不变性

where in the sum,  $N(v)$  denotes the neighbors of  $v$  in graph  $G$ . The readout phase computes a feature vector for the whole graph using some readout function  $R$  according to

$$\hat{y} = R(\{h_v^T \mid v \in G\}). \quad (3)$$

The message functions  $M_t$ , vertex update functions  $U_t$ , and readout function  $R$  are all learned differentiable functions.

Following the non-local mean operation [4], we define a generic non-local operation in deep neural networks as:

$$\mathbf{y}_i = \frac{1}{\mathcal{C}(\mathbf{x})} \sum_{\forall j} f(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_j). \quad (1)$$

i-输出的索引

归一化

i, j节点之间的关系

$\mathbf{x}_j$ 经过 $g(x)$ 函数的变换

is the output signal of the same size as  $\mathbf{x}$ . A pairwise function  $f$  computes a scalar (representing relationship such as affinity) between  $i$  and all  $j$ . The unary function  $g$  computes a representation of the input signal at the position  $j$ . The response is normalized by a factor  $\mathcal{C}(\mathbf{x})$ .

$$f(\mathbf{h}_i, \mathbf{h}_j) = e^{\mathbf{h}_i^T \mathbf{h}_j}$$

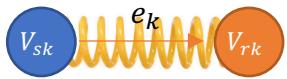
$$f(\mathbf{h}_i, \mathbf{h}_j) = e^{\theta(\mathbf{h}_i)^T \phi(\mathbf{h}_j)}$$

$$f(\mathbf{h}_i, \mathbf{h}_j) = \theta(\mathbf{h}_i)^T \phi(\mathbf{h}_j).$$

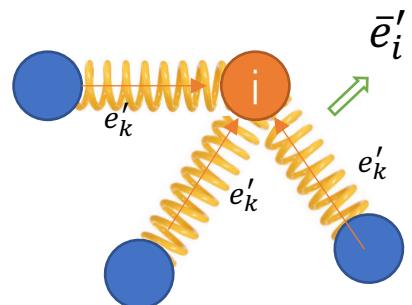
$$f(\mathbf{h}_i, \mathbf{h}_j) = \text{ReLU}(\mathbf{w}_f^T [\theta(\mathbf{h}_i) \| \phi(\mathbf{h}_j)])$$

GN : *Graph Networks*

A GN block contains three “update” functions,  $\phi$ , and three “aggregation” functions,  $\rho$ ,



$e'_k$  = 弹簧之间的力



$E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$

$r_k$  is the index of the receiver node.

$s_k$  is the index of the sender node.

$\mathbf{e}_k$  is the edge's attribute.

$\mathbf{u}$  is a global attribute;

$e'_k$  – 更新边后的特征

$\bar{e}'_i$  – 聚合邻居后特征

$\bar{v}'_i$  – 更新节点后的特征

$\bar{e}'$  – 更新所有边后的特征

$\bar{v}'$  – 更新所有点后的特征

$u'$  – 更新全局后的特征

$$① \quad \mathbf{e}'_k = \phi^e (\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$$

$$③ \quad \mathbf{v}'_i = \phi^v (\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$$

$$⑥ \quad \mathbf{u}' = \phi^u (\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$$

$$② \quad \bar{\mathbf{e}}'_i = \rho^{e \rightarrow v} (E'_i)$$

$$④ \quad \bar{\mathbf{e}}' = \rho^{e \rightarrow u} (E')$$

$$⑤ \quad \bar{\mathbf{v}}' = \rho^{v \rightarrow u} (V')$$

(1)

1.  $\phi^e$  is applied per edge,  $\mathbf{e}'_k = \phi^e (\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$

$$E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}.$$

$E' = \bigcup_i E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$  is the set of all per-edge outputs.

2.  $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$

$$\longrightarrow \bar{\mathbf{e}}'_i = \rho^{e \rightarrow v} (E'_i) \quad \text{作用在} i \text{球上的所有力的和}$$

3.  $\phi^v$  is applied to each node  $i$ , to compute an updated node attribute,  $\mathbf{v}'_i$ .

根据节点本身信息和聚合后的信息, 得出节点新特征。小球的速度, 加速度, 位置等



4. 更新所有的边。计算所有弹簧的势能总和  $\bar{\mathbf{e}}' = \rho^{e \rightarrow u} (E')$

$$V' = \{\mathbf{v}'_i\}_{i=1:N^v}$$

5. 更新所有的点。节点系统的动能  $\bar{\mathbf{v}}' = \rho^{v \rightarrow u} (V')$

6. 更新全局参数u。整个系统的总能值  $\mathbf{u}' = \phi^u (\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$

更新边的信息  $\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$

更新点的信息  $\mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$

更新全局的信息  $\mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$

$\bar{\mathbf{e}}'_i = \rho^{e \rightarrow v}(E'_i)$  某个点聚合周围边的信息

$\bar{\mathbf{e}}' = \rho^{e \rightarrow u}(E')$  所有的边信息

$\bar{\mathbf{v}}' = \rho^{v \rightarrow u}(V')$  所有的节点信息

---

### Algorithm 1 Steps of computation in a full GN block.

---

**function** GRAPHNETWORK( $E, V, \mathbf{u}$ )

**for**  $k \in \{1 \dots N^e\}$  **do**

$\mathbf{e}'_k \leftarrow \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$

**end for**

**for**  $i \in \{1 \dots N^n\}$  **do**

        let  $E'_i = \{(\mathbf{e}'_k, r_k, s_k)\}_{r_k=i, k=1:N^e}$

$\bar{\mathbf{e}}'_i \leftarrow \rho^{e \rightarrow v}(E'_i)$

$\mathbf{v}'_i \leftarrow \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$

**end for**

    let  $V' = \{\mathbf{v}'\}_{i=1:N^v}$

    let  $E' = \{(\mathbf{e}'_k, r_k, s_k)\}_{k=1:N^e}$

$\bar{\mathbf{e}}' \leftarrow \rho^{e \rightarrow u}(E')$

$\bar{\mathbf{v}}' \leftarrow \rho^{v \rightarrow u}(V')$

$\mathbf{u}' \leftarrow \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$

**return**  $(E', V', \mathbf{u}')$

**end function**

    ▷ 1. Compute updated edge attributes

    ▷ 2. Aggregate edge attributes per node

    ▷ 3. Compute updated node attributes

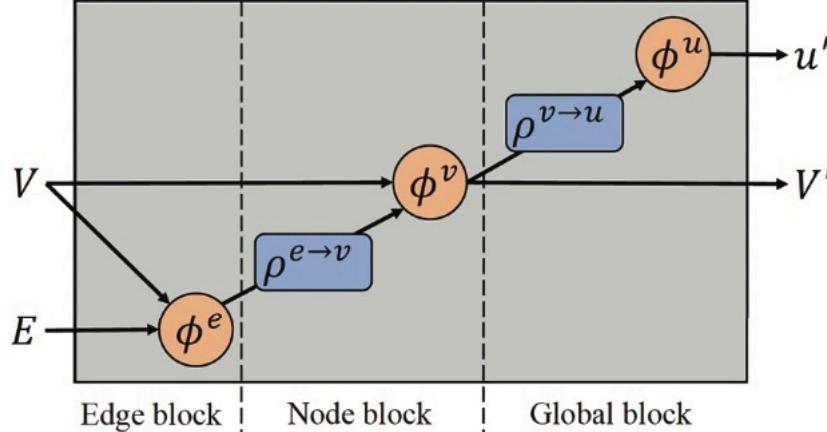
    ▷ 4. Aggregate edge attributes globally

    ▷ 5. Aggregate node attributes globally

    ▷ 6. Compute updated global attribute

MPNN应用于GN：

- the message function,  $M_t$ , plays the role of the GN's  $\phi^e$ , but does not take  $\mathbf{u}$  as input,
- elementwise summation is used for the GN's  $\rho^{e \rightarrow v}$ ,
- the update function,  $U_t$ , plays the role of the GN's  $\phi^v$ ,



(c) Message-Passing Neural Network

$$\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$$

消息传递函数  
采样周围邻居

$$\mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$$

节点更新函数 Ut

$$\mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$$

ReadOut函数

消息函数  $M_t$

$$m_v^{t+1} = \sum_{w \in N(v)} M_t(h_v^t, h_w^t, e_{vw}) \quad (1)$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1}) \quad (2)$$

$$\hat{y} = R(\{h_v^T \mid v \in G\}). \quad (3)$$

Readout

聚合函数

$$\begin{aligned} \bar{\mathbf{e}}'_i &= \rho^{e \rightarrow v}(E'_i) \\ \bar{\mathbf{e}}' &= \rho^{e \rightarrow u}(E') \end{aligned} \quad (1)$$

$$\bar{\mathbf{v}}' = \rho^{v \rightarrow u}(V')$$

聚合所有节点信息

the readout function,  $R$ , plays the role of the GN's  $\phi^u$ , but does not take  $\mathbf{u}$  or  $E'$  as input,

# NLNN应用于GN

1. 聚合邻居attention和更新函数g

$$\phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}) := f^e(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}) = (\alpha^e(\mathbf{v}_{r_k}, \mathbf{v}_{s_k}), \beta^e(\mathbf{v}_{s_k})) = (a'_k, \mathbf{b}'_k) = \mathbf{e}'_k$$

2. 加权求和, 系数C(x)

$$\rho^{e \rightarrow v}(E'_i) := \frac{1}{\sum_{\{k: r_k=i\}} a'_k} \sum_{\{k: r_k=i\}} a'_k \mathbf{b}'_k$$

3. 计算节点v的更新

$$\phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u}) := f^v(\bar{\mathbf{e}}'_i)$$

Multi-heads

聚合邻居attention和本身函数g

$$\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u})$$

聚合节点

$$\mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$$

更新节点

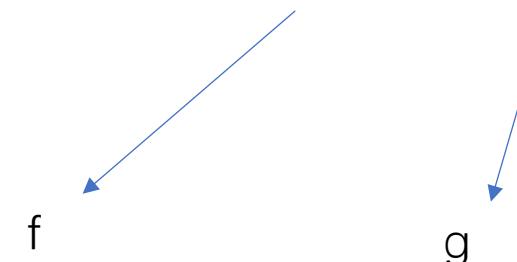
$$\mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$$

加权求和

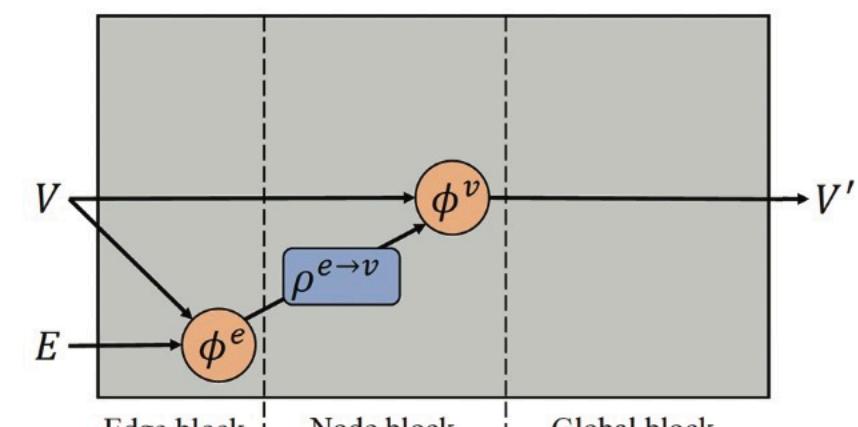
$$\bar{\mathbf{e}}'_i = \rho^{e \rightarrow v}(E'_i)$$

$$\bar{\mathbf{e}}' = \rho^{e \rightarrow u}(E')$$

$$\bar{\mathbf{v}}' = \rho^{v \rightarrow u}(V')$$



$$\mathbf{y}_i = \frac{1}{\mathcal{C}(\mathbf{x})} \sum_{\forall j} f(\mathbf{x}_i, \mathbf{x}_j) g(\mathbf{x}_j). \quad (1)$$



(d) Non-Local Neural Network

我们讨论了简单图的图神经网络模型，这些图是静态的，只有一种节点和一种边。然而，在许多现实世界的应用程序中的图表要复杂得多。它们通常有多种类型的节点、边缘、独特的结构，而且通常是动态的。

### 图的类别

1. 异构图 Heterogeneous Graph Neural Networks
2. 二部图 Bipartite Graph Neural Networks
3. 多维图 Multi-dimensional Graph Neural Networks
4. 符号图 Signed Graph Neural Networks
5. 超图 Hypergraph Neural Networks
6. 动态图 Dynamic Graph Neural Networks

# 1. 异构图 Heterogeneous Graph Neural Networks

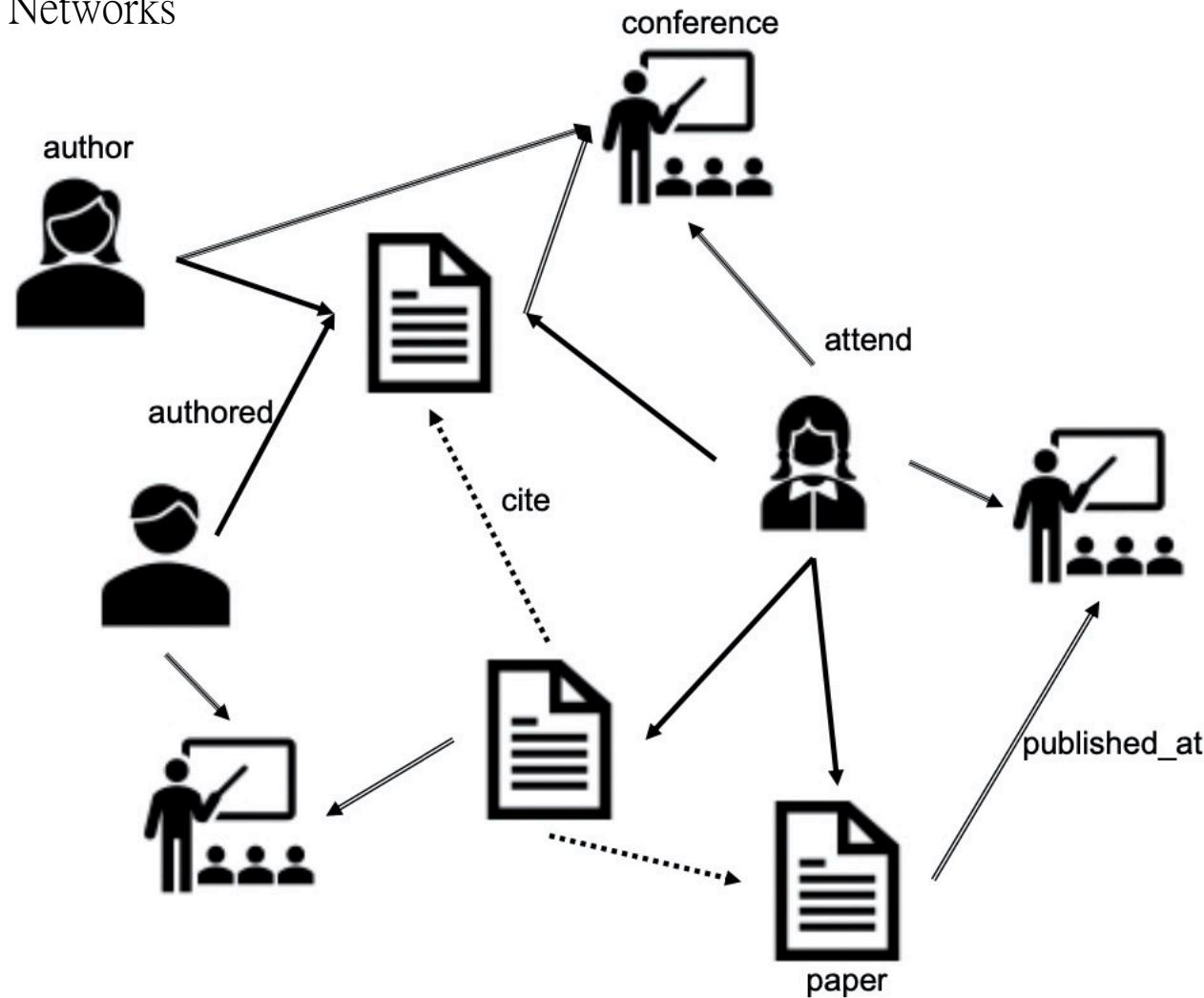
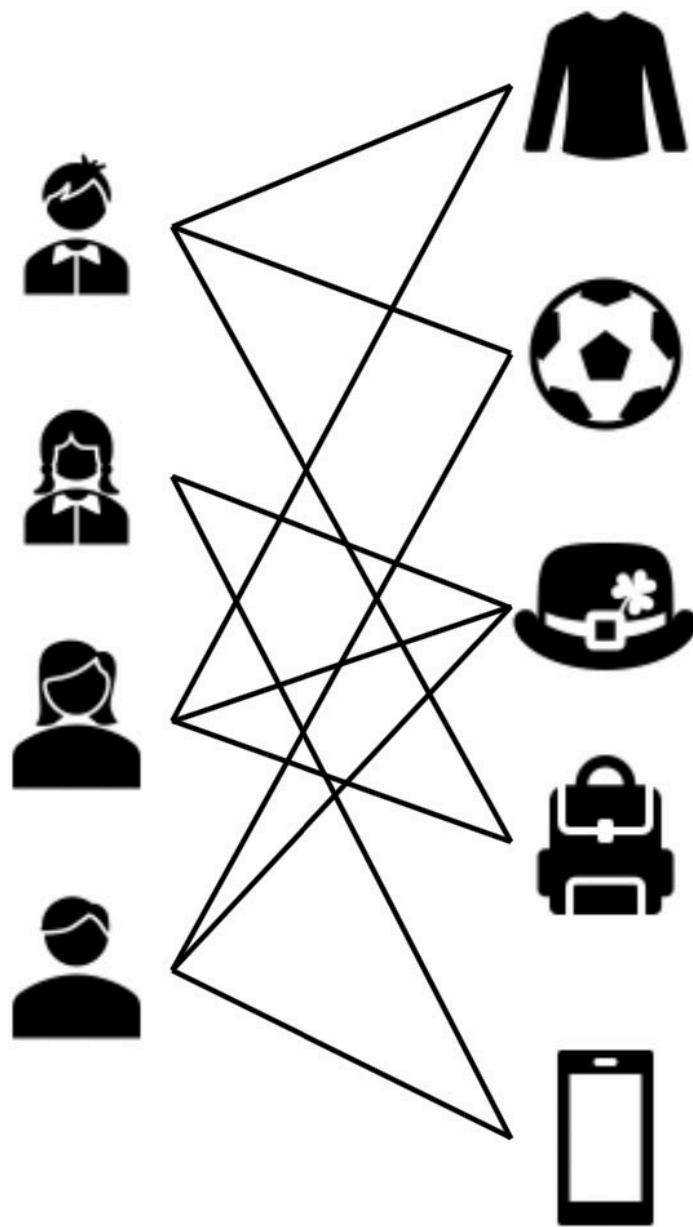


Figure 2.6 A heterogeneous academic graph

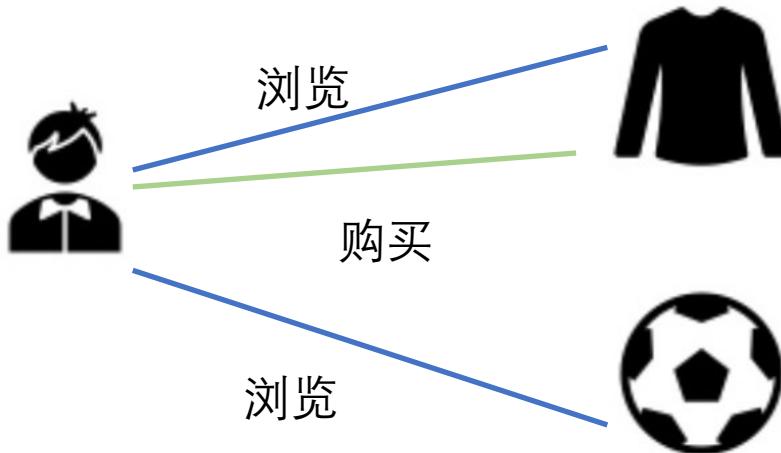


2. 二部图 Bipartite Graph Neural Networks

Figure 2.7 An e-commerce bipartite graph

### 3. 多维图 Multi-dimensional Graph Neural Networks

亚马逊中，用户可以通过“点击”、“购买”、“评论”等多种行为与商品进行交互。这些具有多种关系的图可以通过将每种关系视为一维来自然地建模为多维图



#### 4. 符号图 Signed graph

好友网络中，朋友关系和取消关注朋友关系  
边是积极或者消极的关系中的一种

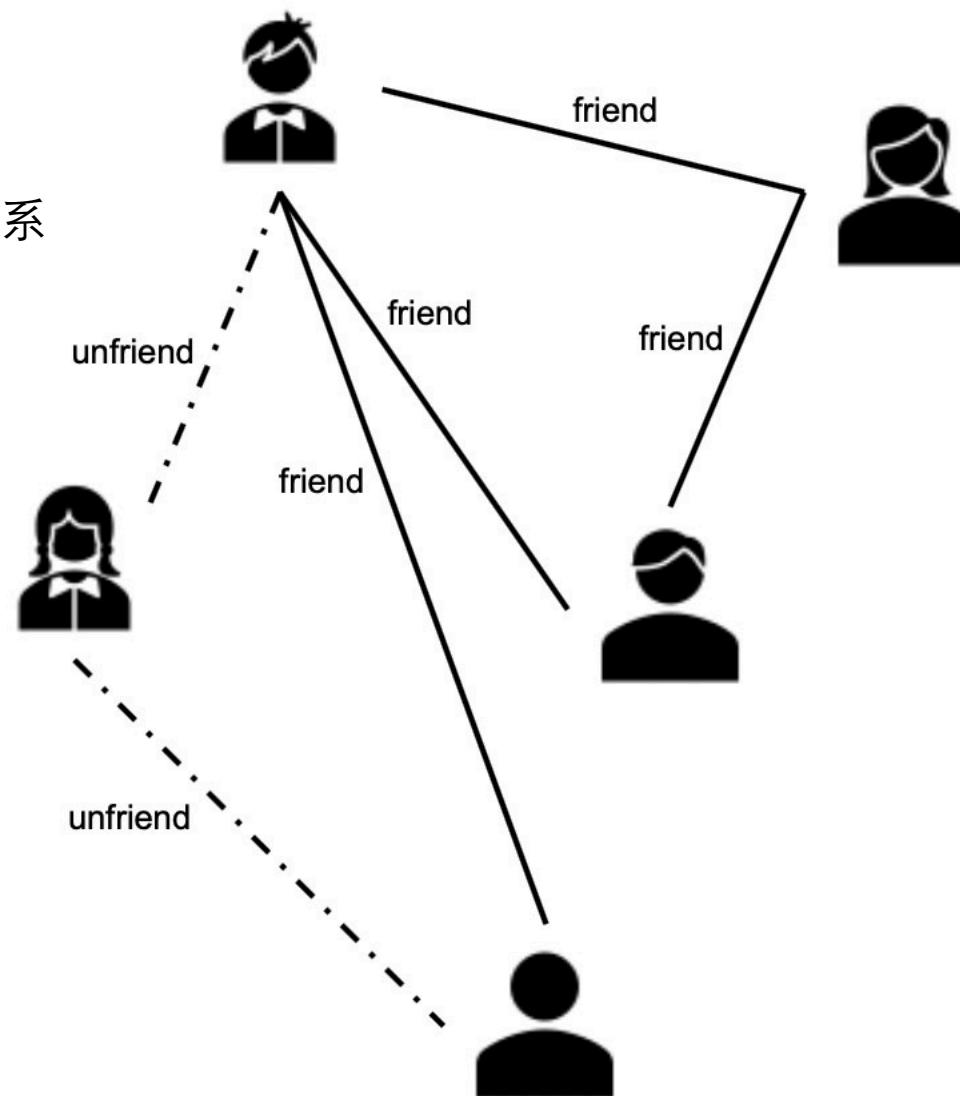
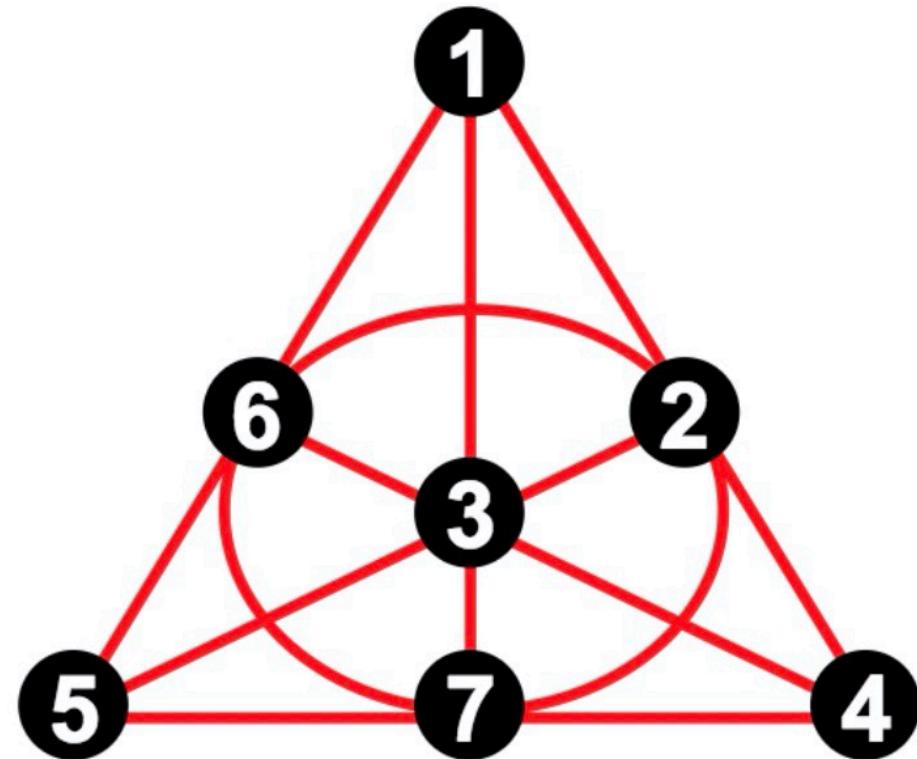


Figure 2.8 An illustrative signed graph

每个边所包含的顶点个数都是相同且为k个的，就可以被称为k阶超图，常见的图就是2阶超图。

5. Hypergraphs 超图：边可能会关联（或者说包含）两个以上的点

**Example of a 3-uniform hypergraph:** The “Fano Plane”,  $V = \{1,2,3,4,5,6,7\}$  and  $E = \{\{1,2,4\}, \{2,3,5\}, \{3,4,6\}, \{4,5,7\}, \{5,6,1\}, \{6,7,2\}, \{7,1,3\}\}$ .



## 6. 动态图 Dynamic Graphs

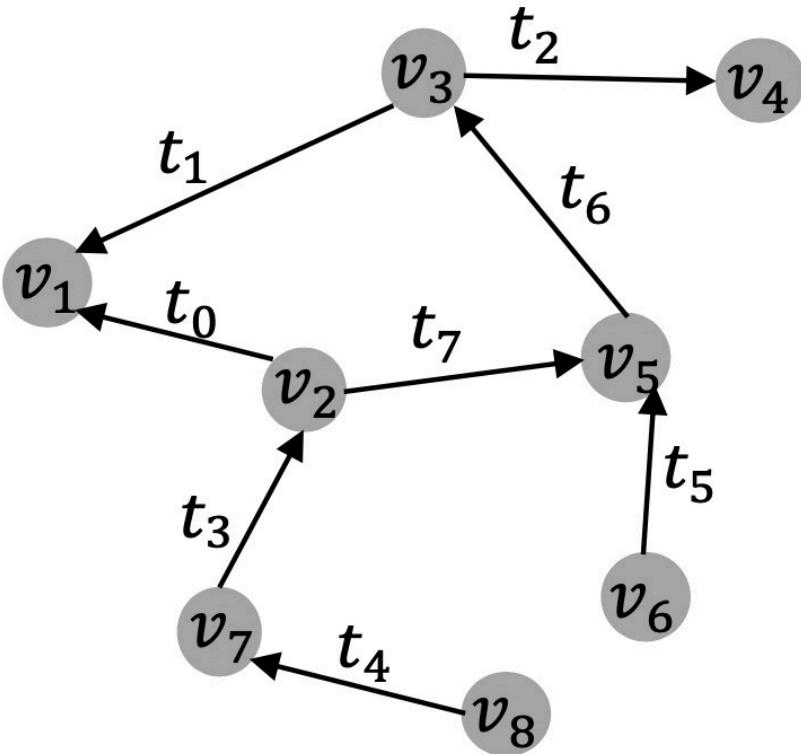


Figure 2.10 An illustrative example of dynamic graphs.

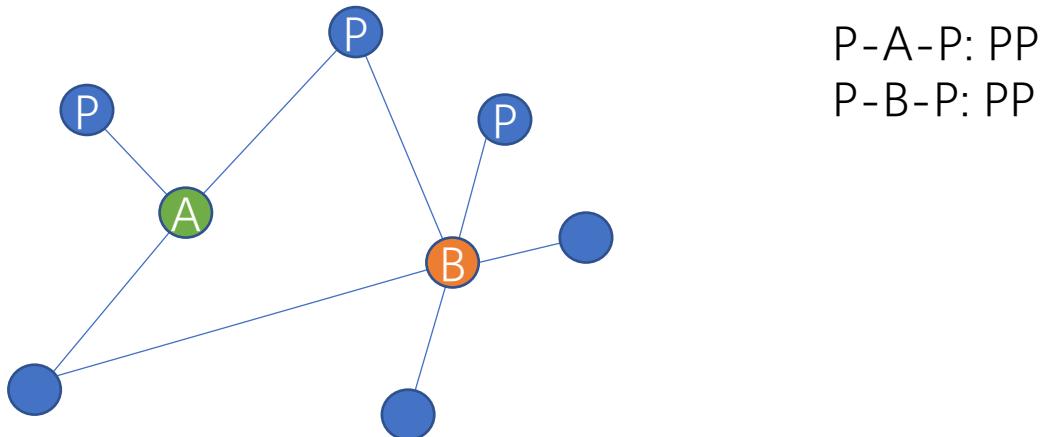
上面提到的图是静态的，观察时节点之间的连接是固定的。但是，在许多实际应用中，随着新节点被添加到图中，图在不断发展，并且新边也在不断出现。例如，在诸如Facebook的在线社交网络中，用户可以不断与他人建立友谊，新用户也可以随时加入Facebook。这些类型的演化图可以表示为动态图，其中每个节点或边都与时间戳关联。

异构图

Heterogeneous Graph Attention Network

包含不同类型节点和链接的异构图

# Heterogeneous Graph Attention Network



### 3 PRELIMINARY

异构图的定义

*Definition 3.1. Heterogeneous Graph* [31]. A heterogeneous graph, denoted as  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ , consists of an object set  $\mathcal{V}$  and a link

set  $\mathcal{E}$ . A heterogeneous graph is also associated with a node type mapping function  $\phi : \mathcal{V} \rightarrow \mathcal{A}$  and a link type mapping function  $\psi : \mathcal{E} \rightarrow \mathcal{R}$ .  $\mathcal{A}$  and  $\mathcal{R}$  denote the sets of predefined object types and link types, where  $|\mathcal{A}| + |\mathcal{R}| > 2$ .

*Definition 3.2. Meta-path* [32]. A meta-path  $\Phi$  is defined as a path in the form of  $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \cdots \xrightarrow{R_l} A_{l+1}$  (abbreviated as  $A_1 A_2 \cdots A_{l+1}$ ), which describes a composite relation  $R = R_1 \circ R_2 \circ \cdots \circ R_l$  between objects  $A_1$  and  $A_{l+1}$ , where  $\circ$  denotes the composition operator on relations.

*Definition 3.3. Meta-path based Neighbors.* Given a node  $i$  and a meta-path  $\Phi$  in a heterogeneous graph, the meta-path based neighbors  $\mathcal{N}_i^\Phi$  of node  $i$  are defined as the set of nodes which connect with node  $i$  via meta-path  $\Phi$ . Note that the node's neighbors includes itself.

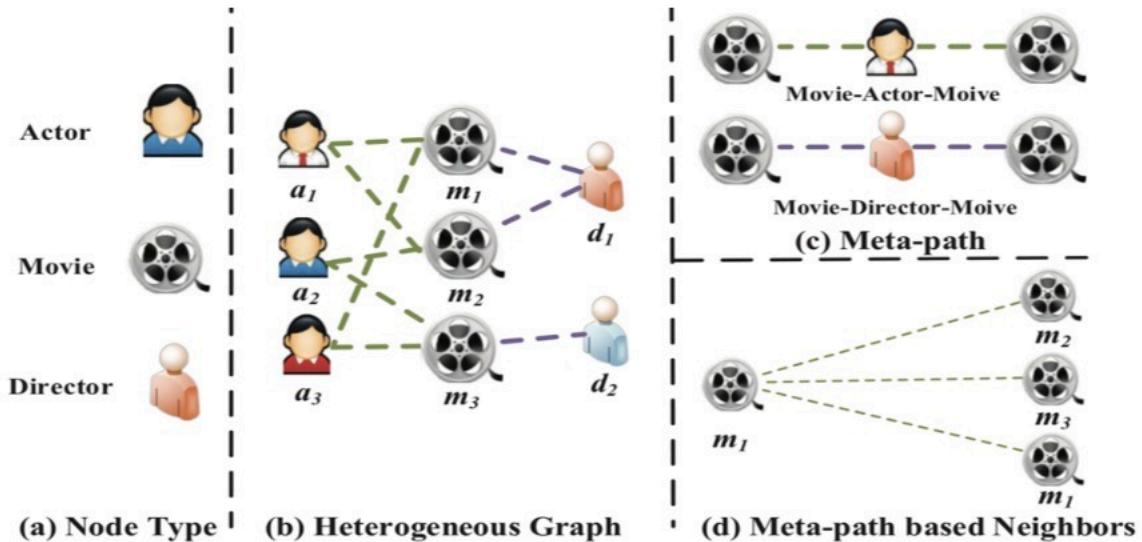
Different meta-paths always reveal different semantics.

译文：我们可以通过一系列邻接矩阵的乘法得到基于元路径的邻居。

## heterogeneous information network (HIN)

Meta-path 元路径

元路径[32]是连接两个对象的复合关系，是一种广泛使用的捕获语义的结构



几种Meta-path。M-A-M; M-D-M  
通过Meta-path找到的邻居信息

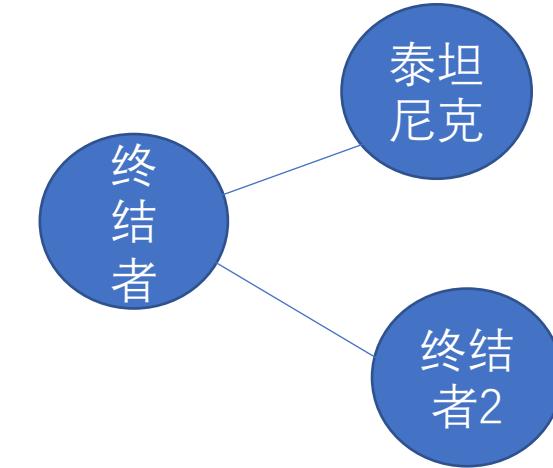
Meta-Path下的邻居节点

**Figure 1: An illustrative example of a heterogeneous graph (IMDB).** (a) Three types of nodes (i.e., actor, movie, director). (b) A heterogeneous graph IMDB consists three types of nodes and two types of connections. (c) Two meta-paths involved in IMDB (i.e., Movie-Actor-Movie and Movie-Director-Movie). (d) Movie  $m_1$  and its meta-path based neighbors (i.e.,  $m_1$ ,  $m_2$  and  $m_3$ ).

## Node-level attention.

在异构图中，节点可以通过各种类型的关系（例如，元路径）连接。给定一个元路径，每个节点都有很多基于元路径的邻居

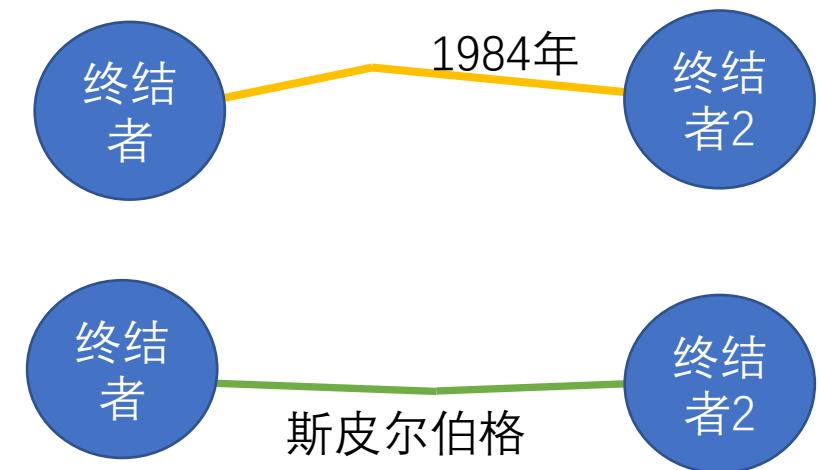
的关注值。仍以IMDB为例，当使用元路径Movie-Director-Movie（电影具有相同的导演）时，The Terminator将通过导演James Cameron连接到Titanic和The Terminator 2。为了更好地将《终结者》的类型识别为科幻电影，该模型应该更多地关注《终结者2》，而不是《泰坦尼克号》。因此，期望如何设计一种能够发现邻居的细微差异并适当学习其权重的模型。



## Semantic-level attention. 语义级别的attention 语义级别的注意力旨在了解每个元路径的重要性，并为其分配适当的权重

配适当的权重。仍以IMDB为例，终结者可以通过Movie-Actor-Movie（均由Schwarzenegger主演）连接到Terminator 2，也可以通过Movie-Year-Movie（均在1984年拍摄）连接到Birdy。但是，在确定电影《终结者》的类型时，MAM通常比MYM扮演更重要的角色。因此，平等对待不同的元路径是不切实际的，并且会削弱某些有用的元路径提供的语义信息。

不同的邻居分配不同的权重



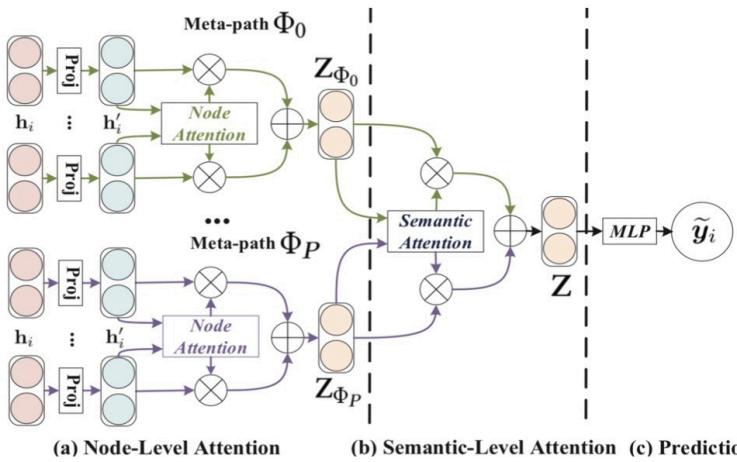
## HAN: Heterogeneous Graph Attention Network

在本文中，我们提出了一个新的异构图注意力网络，叫做HAN，它同时考虑了节点级别和语义级别的注意。特别是在给定节点特征作为输入的情况下，我们使用特定于类型的转换矩阵将不同类型的节点特征投影到同一空间中。然后，节点级别的注意力能够了解节点与其基于元

影到同一空间中。然后，节点级别的注意力能够了解节点与其基于元路径的邻居之间的注意值，而语义级别的目的旨在了解异构图中特定任务的不同元路径的注意值。基于两个级别的学习注意力值，我们的模型可以以分层的方式获得邻居和多个元路径的最佳组合，这使得学习的节点嵌入可以更好地捕获复杂结构和丰富的语义信息。异构图。之

节点级别和语义级别的attention，分层方式学习。同时考虑节点和元路径的重要性

通过对层次注意机制的分析，该算法在异构图分析中具有良好的可解释性。



## 4.1 Node-level Attention

where  $\mathbf{h}_i$  and  $\mathbf{h}'_i$  are the original and projected feature of node  $i$ , respectively. By type-specific projection operation, the node-level

$$e_{ij}^{\Phi} = att_{node}(\mathbf{h}'_i, \mathbf{h}'_j; \Phi). \quad (2)$$

节点*i*和*j*在metapath下的attention

$$\alpha_{ij}^{\Phi} = softmax_j(e_{ij}^{\Phi}) = \frac{\exp(\sigma(\mathbf{a}_{\Phi}^T \cdot [\mathbf{h}'_i \| \mathbf{h}'_j]))}{\sum_{k \in \mathcal{N}_i^{\Phi}} \exp(\sigma(\mathbf{a}_{\Phi}^T \cdot [\mathbf{h}'_i \| \mathbf{h}'_k]))}, \quad (3)$$

GAT

that given meta-path  $\Phi$ , the weight of meta-path based node pair  $(i, j)$  depends on their features. Please note that,  $e_{ij}^{\Phi}$  is asymmetric,

$$\mathbf{z}_i^{\Phi} = \sigma \left( \sum_{j \in \mathcal{N}_i^{\Phi}} \alpha_{ij}^{\Phi} \cdot \mathbf{h}'_j \right). \quad (4)$$

或者多头

Given the meta-path set  $\{\Phi_0, \Phi_1, \dots, \Phi_P\}$ , after feeding node features into node-level attention, we can obtain  $P$  groups of semantic-specific node embeddings, denoted as  $\{\mathbf{Z}_{\Phi_0}, \mathbf{Z}_{\Phi_1}, \dots, \mathbf{Z}_{\Phi_P}\}$ .

## 4.2 Semantic-level Attention

$$meta-path = \Phi_i \quad \{0.1, 0.2, 0.1, 0.5, 0.8\} \quad \{0.3, 0.1, 0.2, 0.4, 0.9\}$$

每一个节点对应多个metapath的值

$$(\beta_{\Phi_0}, \beta_{\Phi_1}, \dots, \beta_{\Phi_P}) = att_{sem}(Z_{\Phi_0}, Z_{\Phi_1}, \dots, Z_{\Phi_P}). \quad (6)$$

$$w_{\Phi_i} = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} q^T \cdot \tanh(W \cdot z_i^\Phi + b), \quad (7)$$

$$\begin{aligned} & W * Z + b \\ & \{0.12, 0.25, 0.8\} \end{aligned}$$

$$q = \{0.12, 0.25, 0.8\}$$

$$W_{\Phi_i} = avg \quad \boxed{0.55 \quad -0.5}$$

$$\beta_{\Phi_i} = \frac{\exp(w_{\Phi_i})}{\sum_{i=1}^P \exp(w_{\Phi_i})}, \quad (8)$$

$w_{\Phi_i}$  节点的  $meta-path$  的值，然后归一化。得到  $meta-path$  的  $attention$

$$Z = \sum_{i=1}^P \beta_{\Phi_i} \cdot Z_{\Phi_i}. \quad (9)$$

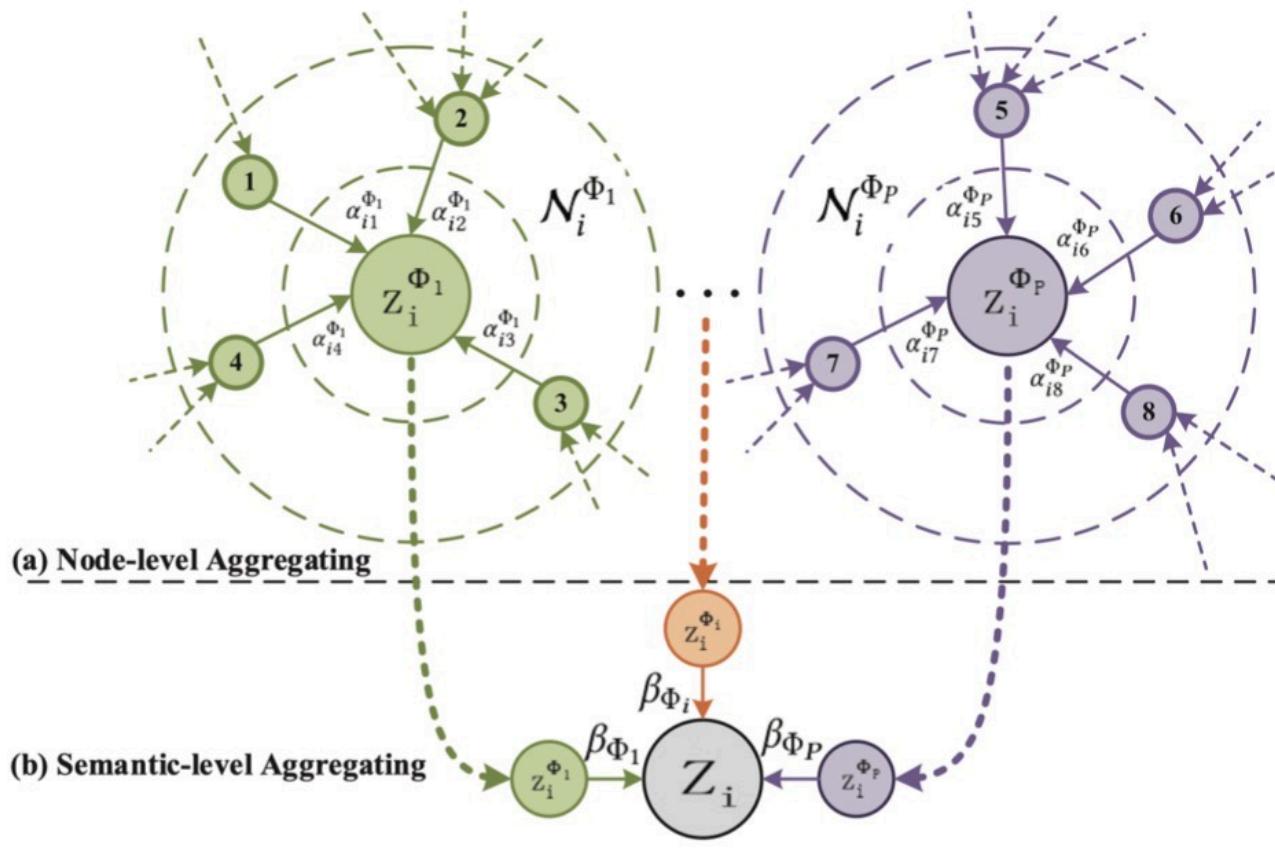
$\{Z_{\Phi_0}, Z_{\Phi_1}, \dots, Z_{\Phi_P}\}$  不同的 meta-path 值求 attention 的和，得到节点最终的表示

每个 metapath 上的表示

---

**Algorithm 1:** The overall process of HAN.

---



**Figure 3: Explanation of aggregating process in both node-level and semantic-level.**

**Input :** The heterogeneous graph  $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ ,  
 The node feature  $\{\mathbf{h}_i, \forall i \in \mathcal{V}\}$ ,  
 The meta-path set  $\{\Phi_0, \Phi_1, \dots, \Phi_P\}$ ,  
 The number of attention head  $K$ ,

**Output:** The final embedding  $Z$  ,  
 The node-level attention weight  $\alpha$  ,  
 The semantic-level attention weight  $\beta$  .

```

1 for  $\Phi_i \in \{\Phi_0, \Phi_1, \dots, \Phi_P\}$  do
2   for  $k = 1 \dots K$  do
3     Type-specific transformation  $\mathbf{h}'_i \leftarrow \mathbf{M}_{\phi_i} \cdot \mathbf{h}_i$  ;
4     for  $i \in \mathcal{V}$  do
5       Find the meta-path based neighbors  $N_i^\Phi$  ;
6       for  $j \in N_i^\Phi$  do
7         Calculate the weight coefficient  $\alpha_{ij}^\Phi$  ;
8       end
9       Calculate the semantic-specific node embedding
10       $\mathbf{z}_i^\Phi \leftarrow \sigma \left( \sum_{j \in N_i^\Phi} \alpha_{ij}^\Phi \cdot \mathbf{h}'_j \right)$ ;
11    end
12    Concatenate the learned embeddings from all attention
13    head  $\mathbf{z}_i^\Phi \leftarrow \parallel_{k=1}^K \sigma \left( \sum_{j \in N_i^\Phi} \alpha_{ij}^\Phi \cdot \mathbf{h}'_j \right)$  ;
14  end
15  Calculate the weight of meta-path  $\beta_{\Phi_i}$  ;
16  Fuse the semantic-specific embedding  $\mathbf{Z} \leftarrow \sum_{i=1}^P \beta_{\Phi_i} \cdot \mathbf{z}_{\Phi_i}$  ;
17  return  $Z, \alpha, \beta$ .

```

---

## 5 EXPERIMENTS

### 5.1 Datasets

- **IMDB.** Here we extract a subset of IMDB which contains 4780 movies (M), 5841 actors (A) and 2269 directors (D). The movies are divided into three classes (*Action*, *Comedy*, *Drama*) according to their genre. Movie features correspond to elements of a bag-of-words represented of plots. We employ the meta-path set {*MAM*, *MDM*} to perform experiments.

- **DBLP<sup>2</sup>.** We extract a subset of DBLP which contains 14328 papers (P), 4057 authors (A), 20 conferences (C), 8789 terms (T). The authors are divided into four areas: *database*, *data mining*, *machine learning*, *information retrieval*. Also, we label each author's research area according to the conferences they submitted. Author features are the elements of a bag-of-words represented of keywords. Here we employ the meta-path set {*APA*, *APCPA*, *APTPA*} to perform experiments.

ACM

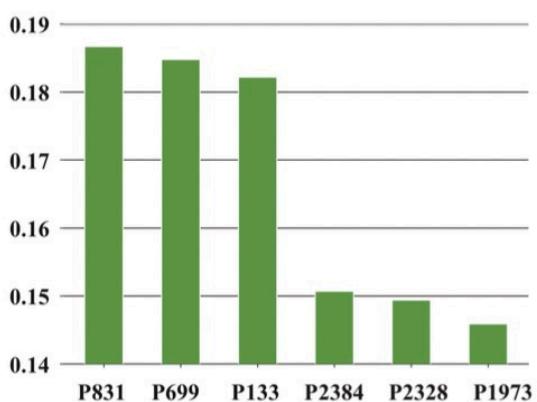
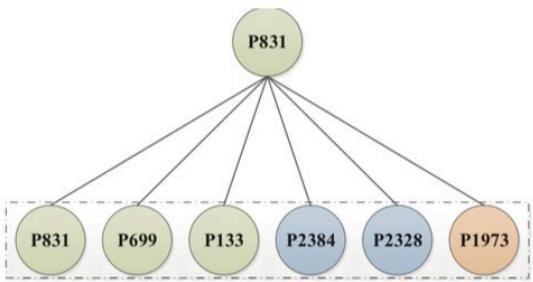
我们提取在KDD, SIGMOD, SIGCOMM, MobiCOMM和VLDB中发表的论文，并将论文分为三类（数据库，无线通信，数据挖掘）。然后，我们构建一个包含3025篇论文 (P)，5835名作者 (A) 和56个主题 (S) 的异构图。纸特征对应于用关键字表示的词袋的元素。我们使用元路径集{PAP, PSP}进行实验。在这里，我们根据发表的论文为论文加标签。

**Table 3: Quantitative results (%) on the node classification task.** $HAN_{nd}$  – 去除节点attention $HAN_{sem}$  – 去除metapath的attention

Datasets	Metrics	Training	DeepWalk	ESim	metapath2vec	HERec	GCN	GAT	$HAN_{nd}$	$HAN_{sem}$	HAN
ACM	Macro-F1	20%	77.25	77.32	65.09	66.17	86.81	86.23	88.15	89.04	<b>89.40</b>
		40%	80.47	80.12	69.93	70.89	87.68	87.04	88.41	89.41	<b>89.79</b>
		60%	82.55	82.44	71.47	72.38	88.10	87.56	87.91	<b>90.00</b>	89.51
		80%	84.17	83.00	73.81	73.92	88.29	87.33	88.48	90.17	<b>90.63</b>
	Micro-F1	20%	76.92	76.89	65.00	66.03	86.77	86.01	87.99	88.85	<b>89.22</b>
		40%	79.99	79.70	69.75	70.73	87.64	86.79	88.31	89.27	<b>89.64</b>
		60%	82.11	82.02	71.29	72.24	88.12	87.40	87.68	<b>89.85</b>	89.33
		80%	83.88	82.89	73.69	73.84	88.35	87.11	88.26	89.95	<b>90.54</b>
DBLP	Macro-F1	20%	77.43	91.64	90.16	91.68	90.79	90.97	91.17	92.03	<b>92.24</b>
		40%	81.02	92.04	90.82	92.16	91.48	91.20	91.46	92.08	<b>92.40</b>
		60%	83.67	92.44	91.32	92.80	91.89	90.80	91.78	92.38	<b>92.80</b>
		80%	84.81	92.53	91.89	92.34	92.38	91.73	91.80	92.53	<b>93.08</b>
	Micro-F1	20%	79.37	92.73	91.53	92.69	91.71	91.96	92.05	92.99	<b>93.11</b>
		40%	82.73	93.07	92.03	93.18	92.31	92.16	92.38	93.00	<b>93.30</b>
		60%	85.27	93.39	92.48	93.70	92.62	91.84	92.69	93.31	<b>93.70</b>
		80%	86.26	93.44	92.80	93.27	93.09	92.55	92.69	93.29	<b>93.99</b>
IMDB	Macro-F1	20%	40.72	32.10	41.16	41.65	45.73	49.44	49.78	<b>50.87</b>	50.00
		40%	45.19	31.94	44.22	43.86	48.01	50.64	52.11	50.85	<b>52.71</b>
		60%	48.13	31.68	45.11	46.27	49.15	51.90	51.73	52.09	<b>54.24</b>
		80%	50.35	32.06	45.15	47.64	51.81	52.99	52.66	51.60	<b>54.38</b>
	Micro-F1	20%	46.38	35.28	45.65	45.81	49.78	55.28	54.17	55.01	<b>55.73</b>
		40%	49.99	35.47	48.24	47.59	51.71	55.91	56.39	55.15	<b>57.97</b>
		60%	52.21	35.64	49.09	49.88	52.29	56.44	56.09	56.66	<b>58.32</b>
		80%	54.33	35.59	48.81	50.99	54.61	56.97	56.38	56.49	<b>58.51</b>

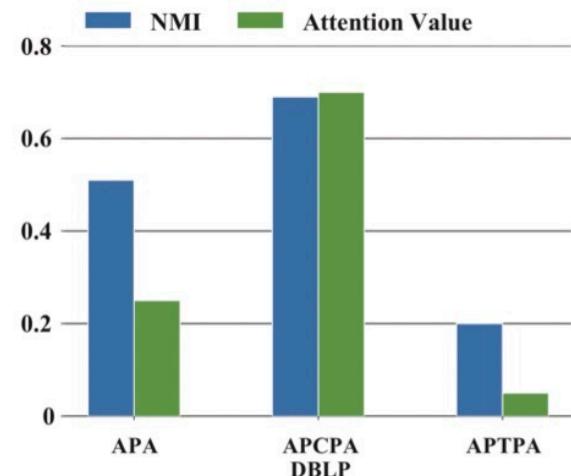
## 5.6 Analysis of Hierarchical Attention Mechanism

HAN的一个显著特性是结合了层次机制，在学习代表性嵌入时考虑了节点邻居和元路径的重要性。

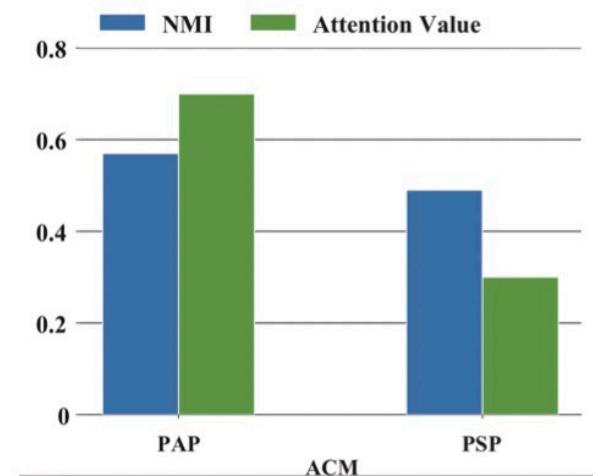


(b) Attention values of P831's neighbors

**Figure 4: Meta-path based neighbors of node P831 and corresponding attention values (Different colors mean different classes, e.g., green means Data Mining, blue means Database, orange means Wireless Communication).**



(a) NMI values on DBLP



(b) NMI values on ACM

**Figure 5: Performance of single meta-path and corresponding attention value.**

代码

ACM数据

- ACM<sup>3</sup>. We extract papers published in KDD, SIGMOD, SIGCOMM, MobiCOMM, and VLDB and divide the papers into three classes (*Database*, *Wireless Communication*, *Data Mining*). Then we construct a heterogeneous graph that comprises 3025 papers (P), 5835 authors (A) and 56 subjects (S). Paper features correspond to elements of a bag-of-words represented of keywords. We employ the meta-path set {PAP, PSP} to perform experiments. Here we label the papers according to the conference they published.

```
>>> data.keys()
dict_keys(['label', 'feature', 'PAP', 'PLP', 'train_idx', 'val_idx', 'test_idx'])
```

Class HAN

HANLayer

Node\_attention:

PAP MetaPath: GATConv(in\_size=1870, out\_size=8, layer\_num\_heads=8)

PLP MetaPath: GATConv (in\_size=1870, out\_size=8, layer\_num\_heads=8)

Semantic\_attention:

SemanticAttention(in\_size=out\_size \* layer\_num\_heads=64)

Predict

nn.Linear(hidden\_size \* num\_heads[-1]=64, out\_size=3)

Dataset : 定义PAP, PLP的对应到的图gs

HAN定义：

每个meta-path对应到的节点级别的attention layer  
语义级别的semantic\_attention

$$w_{\Phi_i} = \frac{1}{|\mathcal{V}|} \sum_{i \in \mathcal{V}} \mathbf{q}^T \cdot \tanh(\mathbf{W} \cdot \mathbf{z}_i^\Phi + \mathbf{b}), \quad (7)$$

```
class SemanticAttention(nn.Module):
    def __init__(self, in_size, hidden_size=128): self: SemanticAttention() in_size: 64 hidden_size: 128
        super(SemanticAttention, self).__init__()

        self.project = nn.Sequential(
            nn.Linear(in_size, hidden_size),
            nn.Tanh(),
            nn.Linear(hidden_size, 1, bias=False)
        )
```

*beta – semanticAttention; z – Node embedding;*

```
return (beta * z).sum(1)
```

$$\mathbf{Z} = \sum_{i=1}^P \beta_{\Phi_i} \cdot \mathbf{Z}_{\Phi_i}. \quad (9)$$



## HAN-Hetero

```
class HAN(nn.Module):
    def __init__(self, meta_paths, in_size, hidden_size, out_size, num_heads, dropout): self: |
        super(HAN, self).__init__()

        self.layers = nn.ModuleList()
        self.layers.append(HANLayer(meta_paths, in_size, hidden_size, num_heads[0], dropout))
        for l in range(1, len(num_heads)):
            self.layers.append(HANLayer(meta_paths, hidden_size * num_heads[l-1],
                                       hidden_size, num_heads[l], dropout))
        self.predict = nn.Linear(hidden_size * num_heads[-1], out_size)
```

```
class HAN(nn.Module):
    def __init__(self, num_meta_paths, in_size, hidden_size, out_size, num_heads, dropout):
        super(HAN, self).__init__()

        self.layers = nn.ModuleList()
        self.layers.append(HANLayer(num_meta_paths, in_size, hidden_size, num_heads[0], dropout))
        for l in range(1, len(num_heads)): # 多层多头, 目前是没有
            self.layers.append(HANLayer(num_meta_paths, hidden_size * num_heads[l-1],
                                       hidden_size, num_heads[l], dropout))
        self.predict = nn.Linear(hidden_size * num_heads[-1], out_size) # hidden*heads, classes;
```



# Graph Transformer Networks

在本文中，我们提出了能够生成新图结构的Graph Transformer Network (GTN)，其中包括识别原始图上未连接节点之间的有用连接，同时在端到端学习新图上的有效节点表示时尚。

最近的一种补救方法是手动设计元路径，这些元路径是与异构边连接的路径，并将异构图转换为由元路径定义的齐次图

表示，而HAN [37]通过将异构图转换为由元路径构造的齐次图来学习图表示学习。但是，这些方法由领域专家手动选择元路径，因此可能无法捕获每个问题的所有有意义的关系。同样，元路径的选择也会显着影响性能。与这些方法不同，我们的Graph Transformer Networks可以在异

性能。与这些方法不同，我们的Graph Transformer Networks可以在异构图上运行并为任务转换图，同时以端到端的方式学习转换图上的节点表示形式。

Preliminaries :

$\mathcal{T}^v$  and  $\mathcal{T}^e$  be the set of node types and edge types

adjacency matrices  $\{A_k\}_{k=1}^K$  where  $K = |\mathcal{T}^e|$ ,

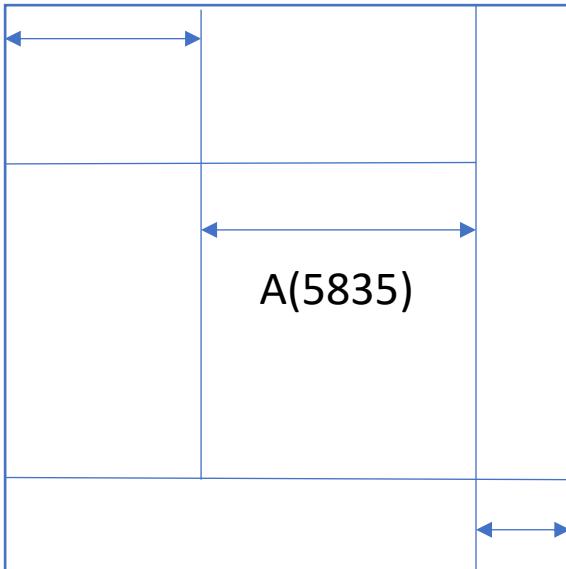
feature matrix  $X \in \mathbf{R}^{N \times D}$  meaning that the D-dimensional

---

ACM数据集: *Paper*(3025), *Author*(5835), *Subject*(56)

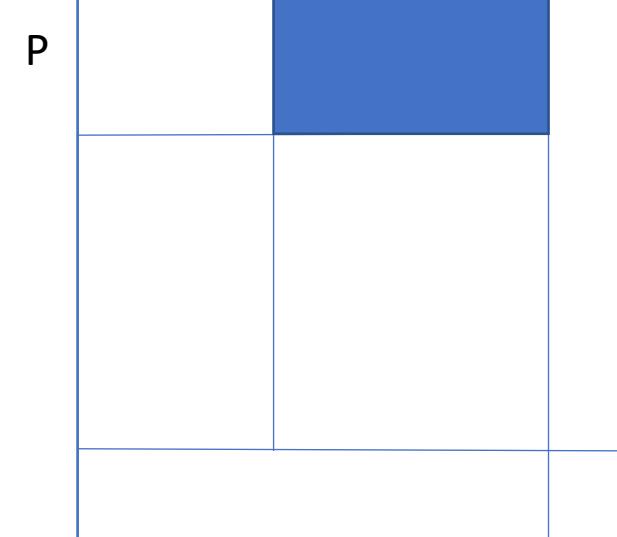
$$\mathcal{T}^e = \{PA, AP, PS, SP\}$$

P(3025)



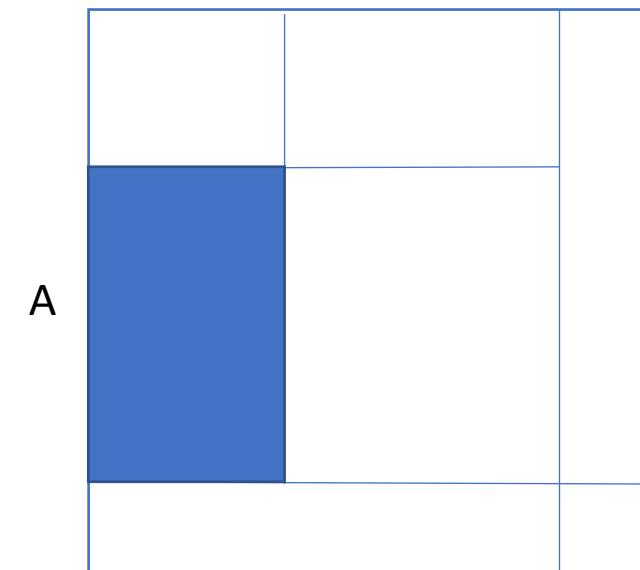
S(56)

Author



$$A_{PA} = \text{Adjacency}$$

Paper



$$A_{AP} = \text{Adjacency}$$

特征如何构建？

NodeType={Paper, Author, Subject}

Paper1 = [1, 0, 0, 0, 1, 1, 0..... 0, 0]

Paper2 = [1, 1, 0, 1, 0, 1, 0..... 0, 0]

Author1 = [Paper1,Paper2] = [1,1,0,1,1,1,0 ..... 0,0]

Subject1 = [Paper1,Paper3] = [1,1,0,1,1,1,0 ..... 0,0]

Paper

Author

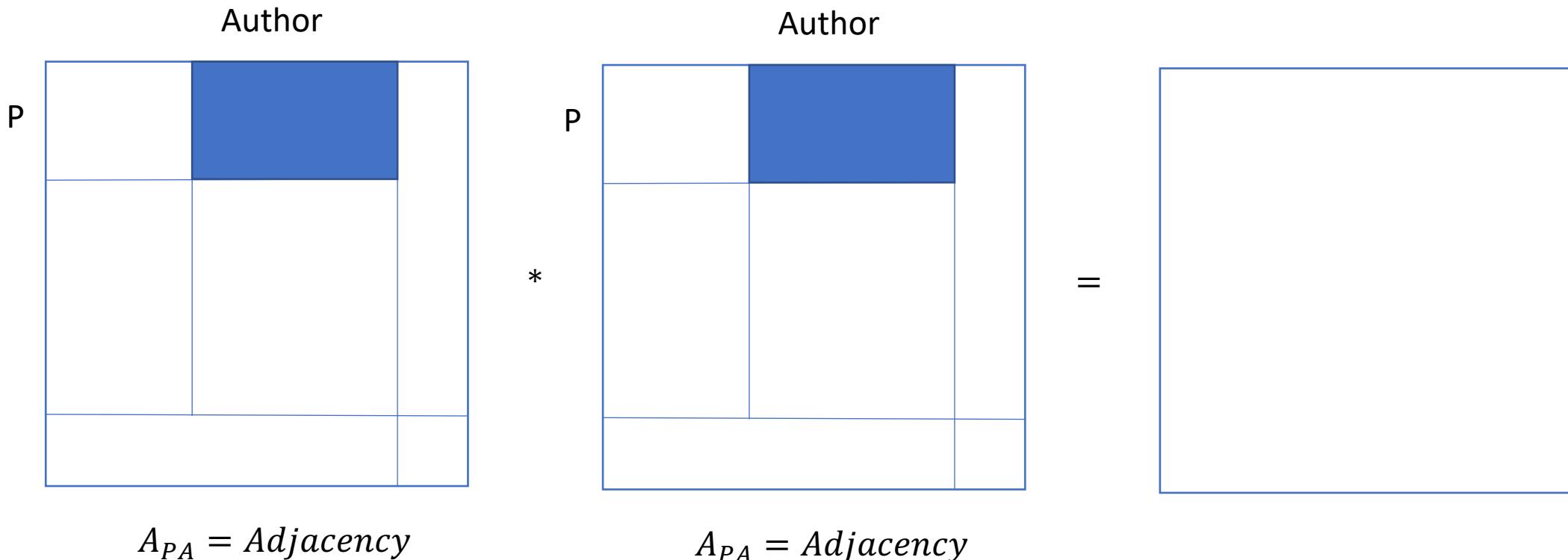
Subject

GTN如何组合多元的meta-path呢？

$$\mathcal{T}^e = \{PA, AP, PS, SP\}$$

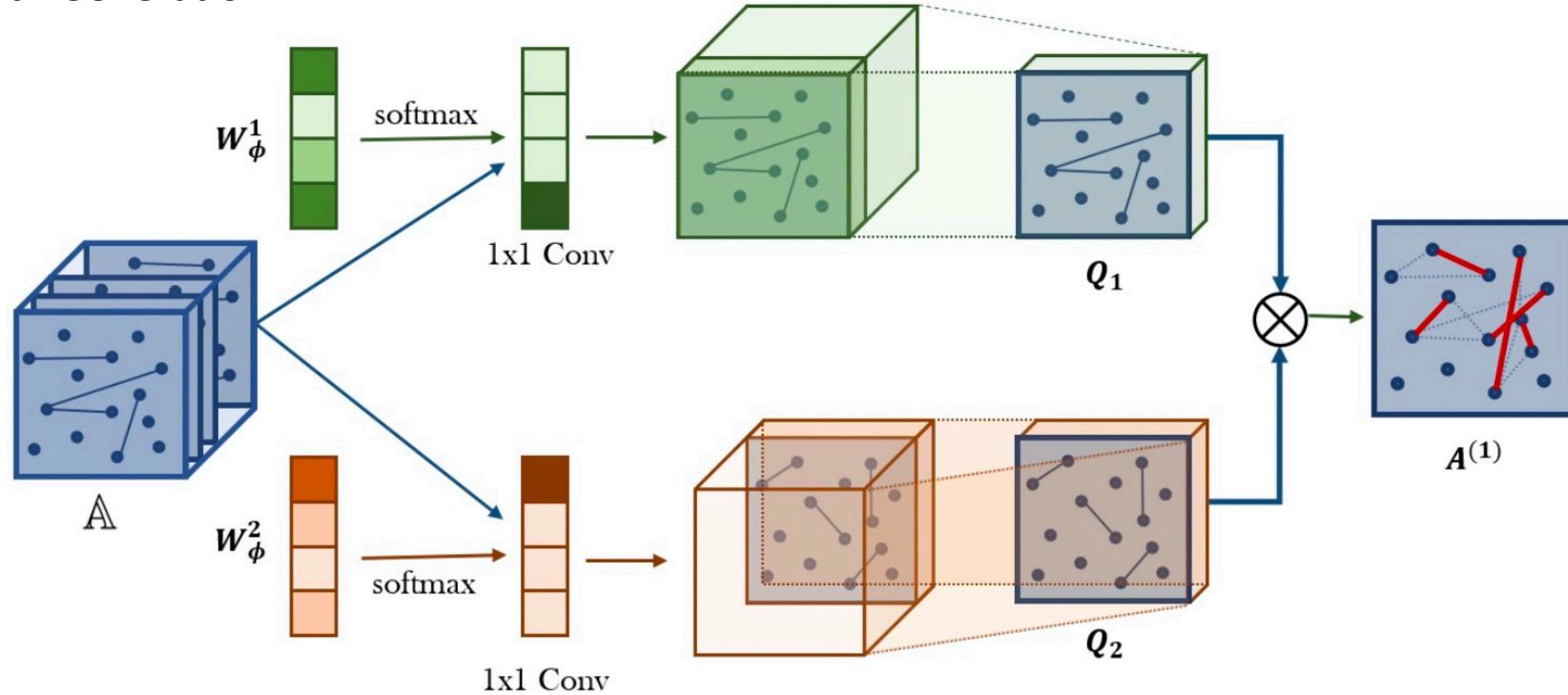
$$meta-path = \{PAP\} = \{PA, AP\} = Adj_{PA} * Adj_{AP}$$

$$meta-path = \{PA, PA\} = Adj_{PA} * Adj_{PA} \quad \text{同理}\{PA, PS\}, \{AP, SP\} = Nan$$



$meta-path = \{AP, PA\} = Adj_{AP} * Adj_{PA} = \{AA\}$ 得到的和Paper无关，对结果没有影响

## Meta-Path Generation



$$\mathcal{T}^e = \{t1, t2, t3, t4\}$$

$$A = \{A1, A2, A3, A4\}$$

$$W_\phi^1 = \{\alpha_1^1, \alpha_2^1, \alpha_3^1, \alpha_4^1\} \quad W_\phi^2 = \{\alpha_1^2, \alpha_2^2, \alpha_3^2, \alpha_4^2\}$$

$$Q_1 = \alpha_1^1 * A_1 + \alpha_2^1 * A_2 + \alpha_3^1 * A_3 + \alpha_4^1 * A_4$$

$$Q_2 = \alpha_1^2 * A_1 + \alpha_2^2 * A_2 + \alpha_3^2 * A_3 + \alpha_4^2 * A_4$$

$$A^1 = Q1 * Q2$$

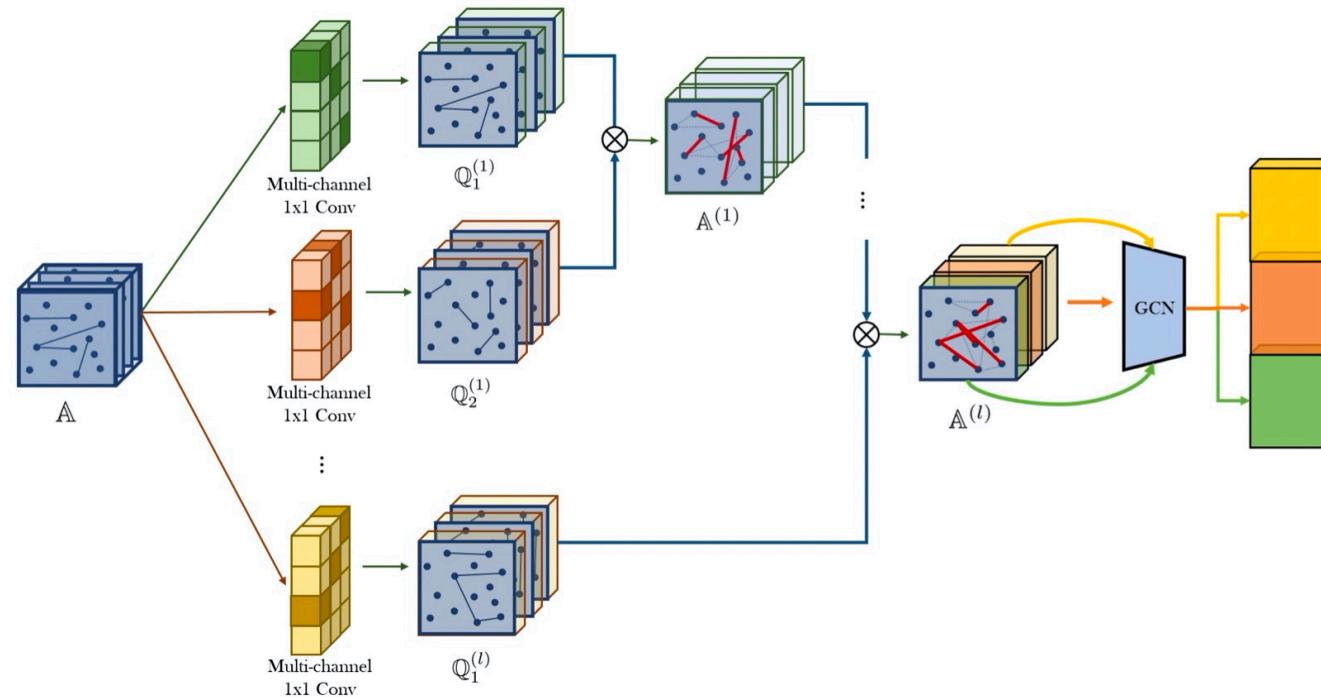
$$\sum_{t_l \in \mathcal{T}^e} \alpha_{tl}^{(l)} A_{tl}$$

$\mathcal{T}^e$  – 边的类型

$\alpha_{tl}$  – 这一层的参数

$A_{tl}$  – 这一层的邻接矩阵

## Graph Transformer Networks



$$\begin{aligned}
 Q_1 &= \alpha_1^1 * A_1 + \alpha_2^1 * A_2 + \alpha_3^1 * A_3 + \alpha_4^1 * A_4 \\
 Q_2 &= \alpha_1^2 * A_1 + \alpha_2^2 * A_2 + \alpha_3^2 * A_3 + \alpha_4^2 * A_4 \\
 Q_3, Q_4 \dots Q_l \\
 A^1 &= Q1 * Q2 \\
 A^l &= Q1 * Q2 * Q3 * \dots * Ql
 \end{aligned}$$

$$A_P = \left( \sum_{t_1 \in \mathcal{T}^e} \alpha_{t_1}^{(1)} A_{t_1} \right) \left( \sum_{t_2 \in \mathcal{T}^e} \alpha_{t_2}^{(2)} A_{t_2} \right) \dots \left( \sum_{t_l \in \mathcal{T}^e} \alpha_{t_l}^{(l)} A_{t_l} \right)$$

通过这种方式，可以学习任意长度的元路径结构  
(4)

当卷积核参数为 $1 \times 1 \times K \times C$ 时候，输出的邻接矩阵为 $N \times N \times C$

$$Q_1 = \alpha_1^1 * A_1 = 1 \times 1 \times K \times C \circ N \times N \times K = N \times N \times C$$

For numerical stability, the matrix is normalized by its degree matrix as  $A^{(l)} = D^{-1}Q_1Q_2$ .

1. 矩阵相乘之后，得到的是一阶meta-path关系的图，但是图本身的性质没有保留?  
为了保留图本身的性质，给矩阵A增加一个单位阵，目的是  $I * A_1 = A_1$

2. 矩阵相乘后，或出现自连接边，应该将其去掉？

$$\begin{array}{c|c} PA & AP \\ \hline \end{array} = \begin{array}{c|c} & PP \\ \hline & \diagdown \end{array}$$

$$P1 - A1 - P1 \Rightarrow P1 \text{---} P1$$

Table 2: Evaluation results on the node classification task (F1 score).

	DeepWalk	metapath2vec	GCN	GAT	HAN	$GTN_{-I}$	<b><math>GTN</math> (proposed)</b>
DBLP	63.18	85.53	87.30	93.71	92.83	93.91	<b>94.18</b>
ACM	67.42	87.61	91.60	92.33	90.96	91.13	<b>92.68</b>
IMDB	32.08	35.21	56.89	58.14	56.77	52.33	<b>60.92</b>

$GTN_{-I} \rightarrow$  不添加单位阵  $I$

我们提取在KDD, SIGMOD, SIGCOMM, MobiCOMM和VLDB中发表的论文，并将论文分为三类（数据库，无线通信，数据挖掘）。然后，我们构建一个包含3025篇论文（P），5835名作者（A）和56个主题（S）的异构图。纸特征对应于用关键字表示的词袋的元素。我们使用元路径集{PAP, PSP}进行实验。在这里，我们根据发表的论文为论文加标签。

## Paper-Author

```
>>> (edges[0].sum(axis=1)!=0).sum()  
3020  
>>> (edges[0].sum(axis=0)!=0).sum()  
5912
```

```
>>> (edges[1].sum(axis=1)!=0).sum()  
5912  
>>> (edges[1].sum(axis=0)!=0).sum()  
3020
```

## Paper-Subject

```
>>> (edges[2].sum(axis=1)!=0).sum()  
3025  
>>> (edges[2].sum(axis=0)!=0).sum()  
57
```

```
>>> (edges[3].sum(axis=1)!=0).sum()  
57  
>>> (edges[3].sum(axis=0)!=0).sum()  
3025
```

## Code

GTN(num\_edge=5, num\_channels=2, w\_in=1902, w\_out=64, num\_class=3, num\_layers=2)

    GTLayer(num\_edge=5, num\_channels=2, first=True)  Conv的参数; =>A

        GTConv(in\_channels, out\_channels)  Q1

        GTConv(in\_channels, out\_channels)  Q2

    GTLayer(num\_edge, num\_channels, first=False)

        GTConv(in\_channels, out\_channels)  Q3

GCN

    nn.Parameter(torch.Tensor(w\_in=1902, w\_out=64))

    nn.Linear

For numerical stability, the matrix is normalized by its degree matrix as  $A^{(l)} = D^{-1}Q_1Q_2$ .

矩阵相乘后，去掉自连接边

## metapath2vec

### 元路径随机游走

- 如果忽略节点类型进行随机游走，结果会是有偏的，数目较多的节点类型出现的概率更大。
- 元路径随机游走：定义好一个游走类型路径，在上图中，路径可以定为APA,APVPA等，然后按照这个路径游走，即下一个节点只采样符合要求的节点类型；元路径通常为对称的。
- 优点：这种元路径随机游走策略可以确保不同类型的节点语义关系被恰当的并入skip-gram模型中

## 异构图的定义

*Definition 2.1. A Heterogeneous Network* is defined as a graph  $G = (V, E, T)$  in which each node  $v$  and each link  $e$  are associated with their mapping functions  $\phi(v) : V \rightarrow T_V$  and  $\varphi(e) : E \rightarrow T_E$ , respectively.  $T_V$  and  $T_E$  denote the sets of object and relation types, where  $|T_V| + |T_E| > 2$ .

**PROBLEM 1. *Heterogeneous Network Representation Learning:*** Given a heterogeneous network  $G$ , the task is to learn the  $d$ -dimensional latent representations  $\mathbf{X} \in \mathbb{R}^{|V| \times d}$ ,  $d \ll |V|$  that are able to capture the structural and semantic relations among them.

### 3.1 Homogeneous Network Embedding

ally, given a network  $G = (V, E)$ , the objective is to maximize the network probability in terms of local structures [8, 18, 22], that is:

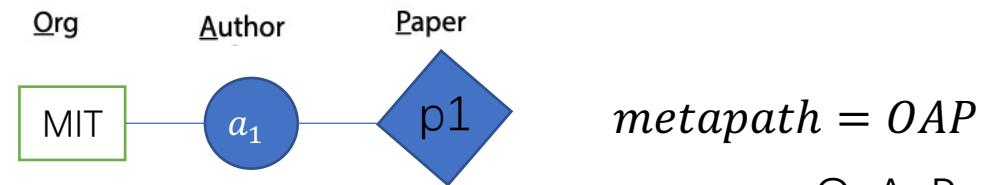
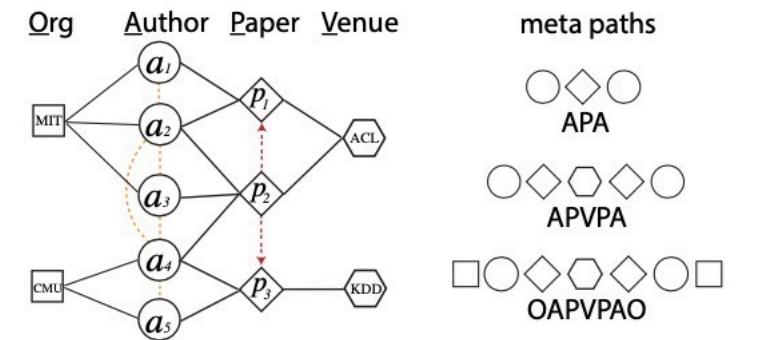
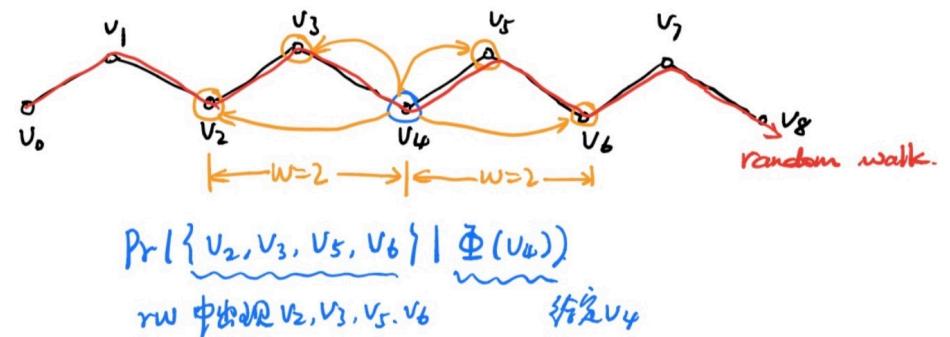
$$\arg \max_{\theta} \prod_{v \in V} \prod_{c \in N(v)} p(c|v; \theta) \quad (1)$$

where  $N(v)$  is the neighborhood of node  $v$  in the network  $G$ , which can be defined in different ways such as  $v$ 's one-hop neighbors, and  $p(c|v; \theta)$  defines the conditional probability of having a context node  $c$  given a node  $v$ .

### 3.2 Heterogeneous Network Embedding: *metapath2vec*

**Heterogeneous Skip-Gram.** In *metapath2vec*, we enable skip-gram to learn effective node representations for a heterogeneous network  $G = (V, E, T)$  with  $|T_V| > 1$  by maximizing the probability of having the heterogeneous context  $N_t(v)$ ,  $t \in T_V$  given a node  $v$ :

$$\arg \max_{\theta} \sum_{v \in V} \sum_{t \in T_V} \sum_{c_t \in N_t(v)} \log p(c_t|v; \theta) \quad (2)$$

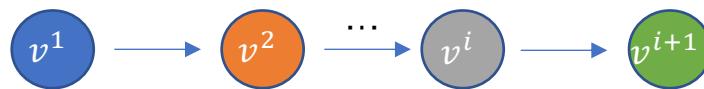


$$p(MIT, P1|a1; \theta)$$

a meta-path scheme  $\mathcal{P}$ :  $V_1 \xrightarrow{R_1} V_2 \xrightarrow{R_2} \cdots V_t \xrightarrow{R_t} V_{t+1} \cdots \xrightarrow{R_{l-1}} V_l$ ,

$$p(v^{i+1}|v_t^i, \mathcal{P}) = \begin{cases} \frac{1}{|N_{t+1}(v_t^i)|} & (v^{i+1}, v_t^i) \in E, \phi(v^{i+1}) = t+1 \\ 0 & (v^{i+1}, v_t^i) \in E, \phi(v^{i+1}) \neq t+1 \\ 0 & (v^{i+1}, v_t^i) \notin E \end{cases} \quad (3)$$

$v^i \rightarrow v^{i+1}$  必须连接到是满足metapath的下一个类别

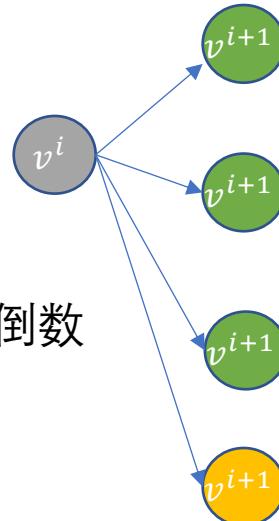


meta-paths are commonly used in a symmetric way,

$A - B - A; A - B - C - B - A$  对称的metapath

个人理解，得到的是 $A - B; B - A$ 的概率，所以要一直游走到采样长度 $l$

概率等于该类型节点数量倒数



# Skip-gram

## 1. 训练语料

"The quick brown fox jumps over the lazy dog." window size = 5.

### Source Text

The quick brown fox jumps over the lazy dog. →

### Training Samples

(the, quick)  
(the, brown)

The quick brown fox jumps over the lazy dog. →

(quick, the)  
(quick, brown)  
(quick, fox)

The quick brown fox jumps over the lazy dog. →

(brown, the)  
(brown, quick)  
(brown, fox)  
(brown, jumps)

The quick brown fox jumps over the lazy dog. →

(fox, quick)  
(fox, brown)  
(fox, jumps)  
(fox, over)

## Subsampling

# Training Samples

(the, quick)  
(the, brown)

(quick, the)  
(quick, brown)  
(quick, fox)

(brown, the)  
(brown, quick)  
(brown, fox)  
(brown, jumps)

(fox, quick)  
(fox, brown)  
(fox, jumps)  
(fox, over)

(quick, the) : 没有学习到quick的任何上下文相关的信息

(the, quick) (the, brown) : 这种语料对于我们来说也没有任何上下文相关性

Subsampling : 有一定的概率删掉这个词，概率和单词出现的频率有关

### Sampling rate

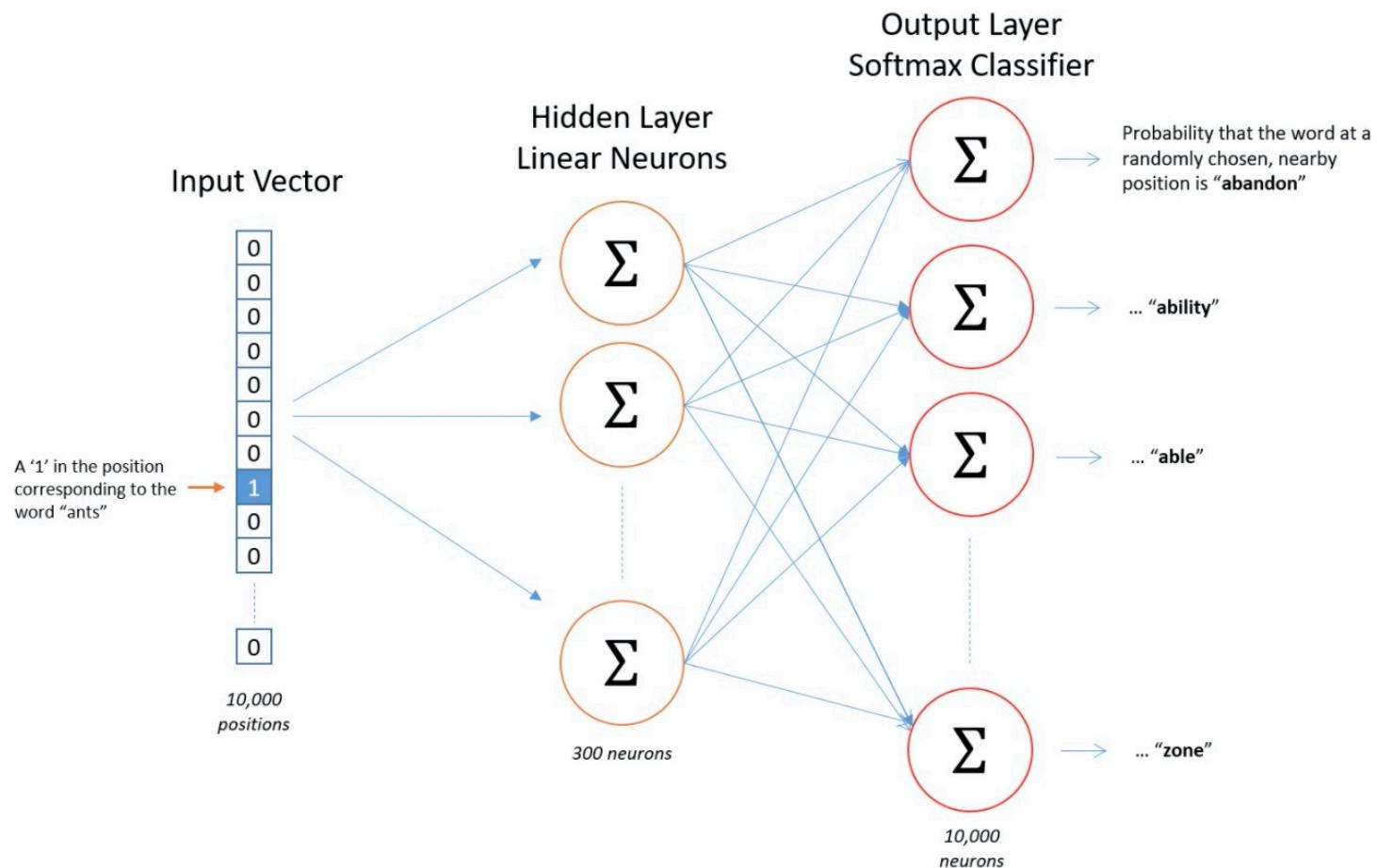
$w_i$  is the word,  $z(w_i)$  is the fraction of the total words in the corpus that are that word. For example, if the word "peanut" occurs 1,000 times in a 1 billion word corpus, then  $z('peanut') = 1E-6$ .

$P(w_i)$  is the probability of *keeping* the word:

$$P(w_i) = \left( \sqrt{\frac{z(w_i)}{0.001}} + 1 \right) \cdot \frac{0.001}{z(w_i)}$$

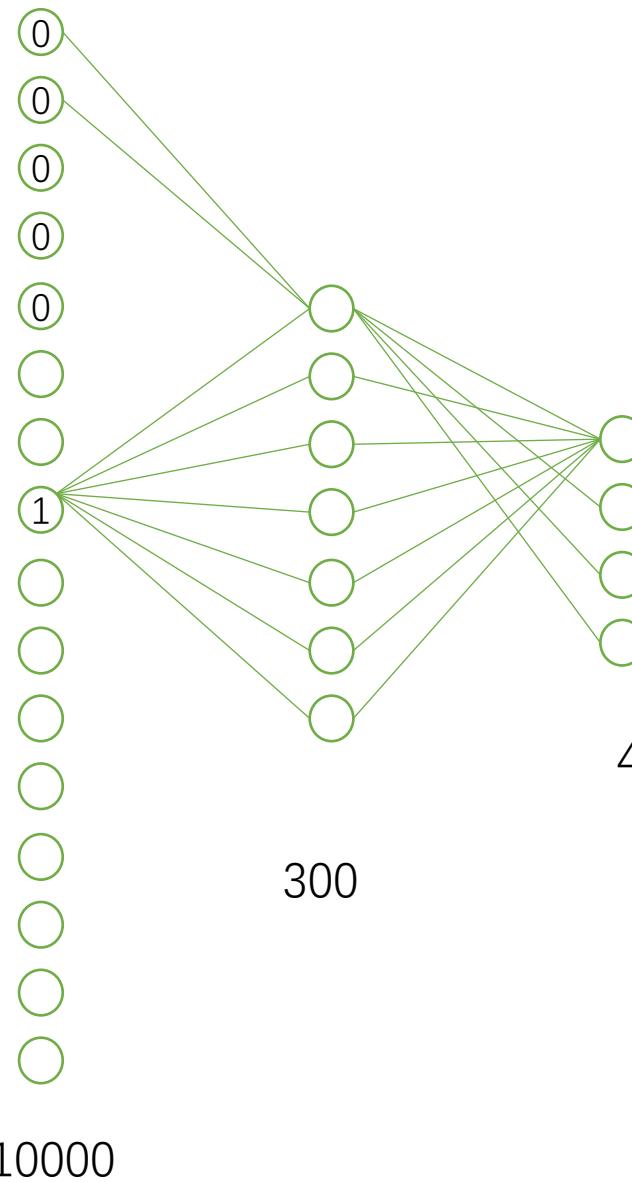
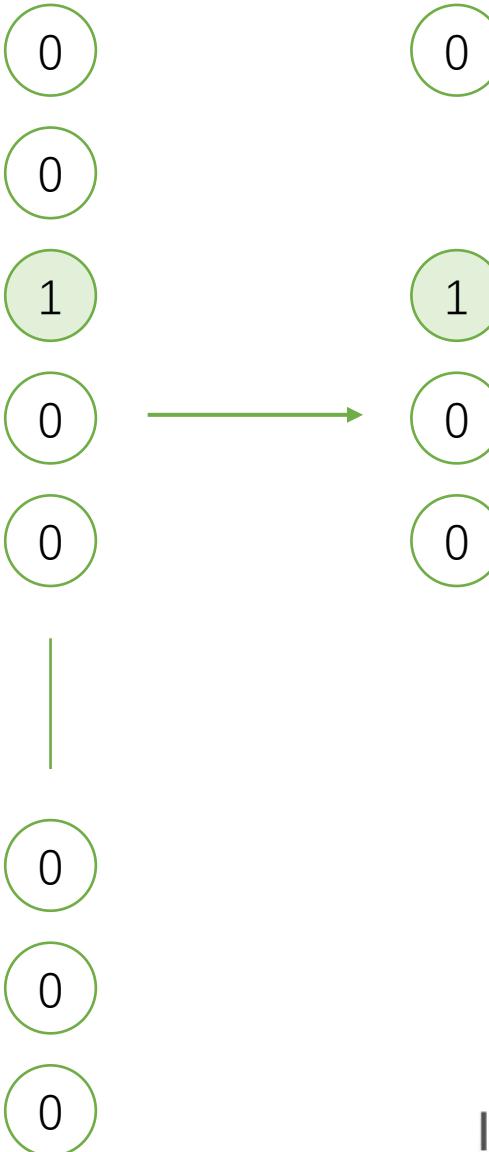
- $P(w_i) = 1.0$  (100% chance of being kept) when  $z(w_i) \leq 0.0026$ .
  - This means that only words which represent more than 0.26% of the total words will be subsampled.
- $P(w_i) = 0.5$  (50% chance of being kept) when  $z(w_i) = 0.00746$ .
- $P(w_i) = 0.033$  (3.3% chance of being kept) when  $z(w_i) = 1.0$ .
  - That is, if the corpus consisted entirely of word  $w_i$ , which of course is ridiculous.

# Negative Sampling



Hidden Layer -> Output Layer:  
300\*10000

Input Layer -> Hidden Layer:  
1\*300



In the hidden layer, only the weights for the input word are updated (this is true whether you're using Negative Sampling or not).

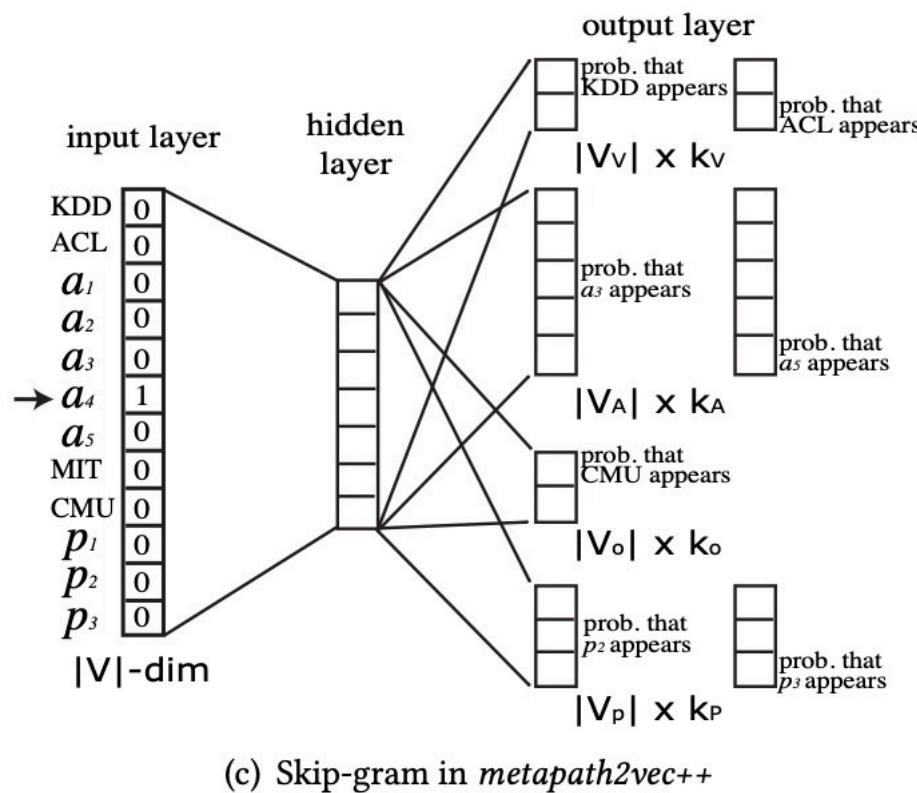
# Selecting Negative Samples

出现频率越高的单词，被选中的几率就越大

$$P(w_i) = \frac{f(w_i)}{\sum_{j=0}^n (f(w_j))}$$

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=0}^n (f(w_j)^{3/4})}$$

## Metapath2vec++



**Heterogeneous negative sampling.** We further propose the *metapath2vec++* framework, in which the softmax function is normalized with respect to the node type of the context  $c_t$ . Specifically,  $p(c_t|v; \theta)$  is adjusted to the specific node type  $t$ , that is,

$$p(c_t|v; \theta) = \frac{e^{X_{c_t} \cdot X_v}}{\sum_{u_t \in V_t} e^{X_{u_t} \cdot X_v}} \quad (5)$$



Code

random\_walk\_txt:语料

Dataset:

self.parse\_random\_walk\_txt(random\_walk\_txt, window\_size) 导入语料, 字典, 频率

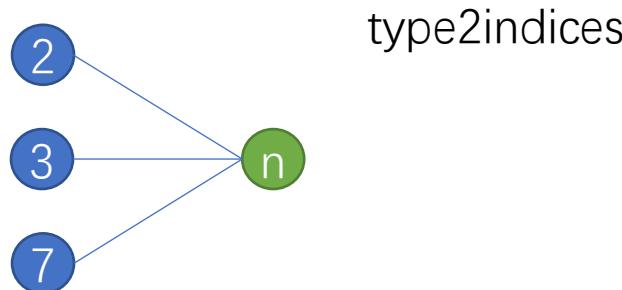
=> index2token, token2index, word\_and\_counts, index2frequency, node\_context\_pairs(Skip-gram语料)

self.parse\_node\_type\_mapping\_txt(node\_type\_mapping\_txt, self.nodeid2index) 节点类型

=> index2type, type2indices 类型边相连接的点

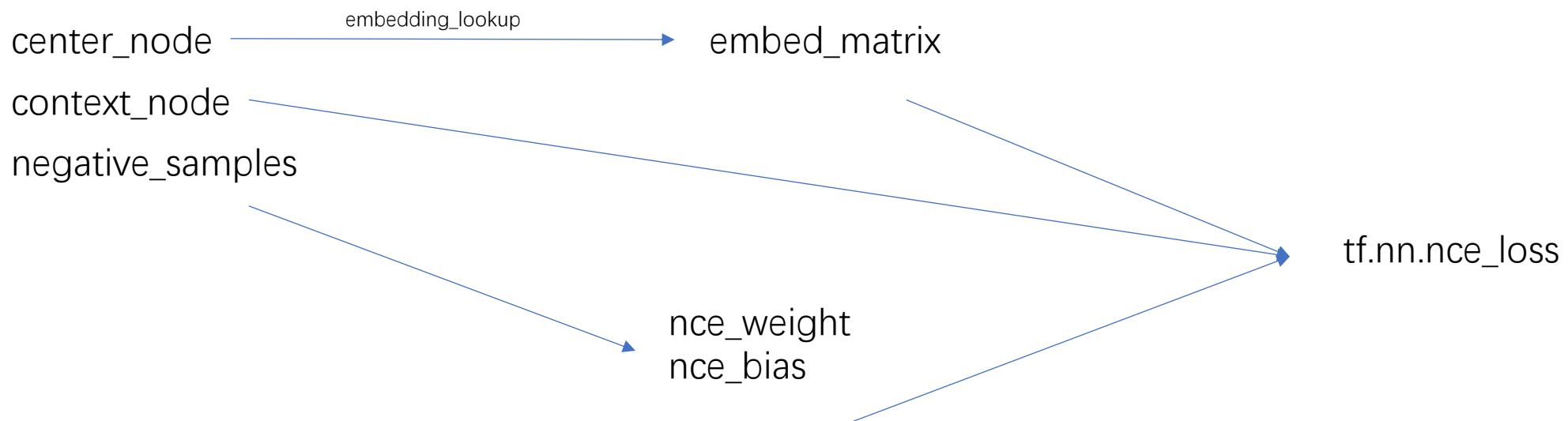
self.prepare\_sampling\_dist(index2frequency, index2type, type2indices) 负采样

=> 均匀采样, 类别采样的概率



build\_model

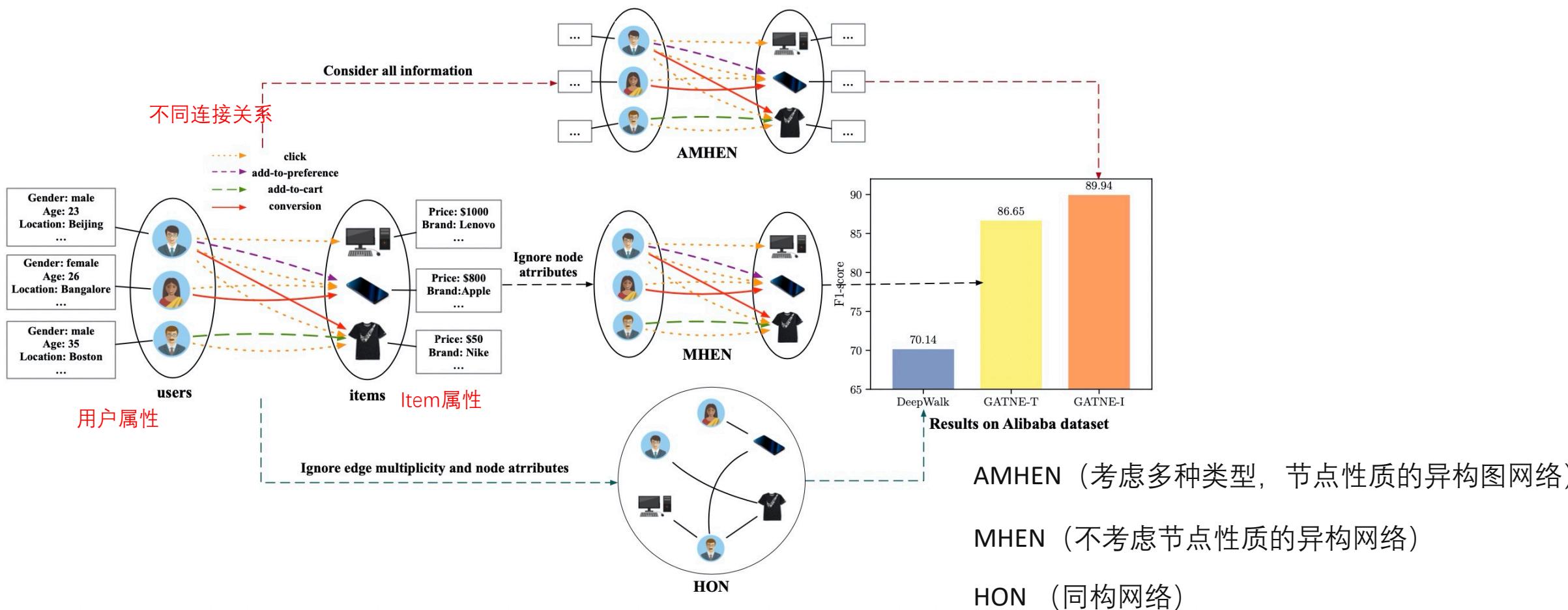
Input:



# Representation Learning for Attributed Multiplex Heterogeneous Network

Attributed：节点的属性，用户，商品

Multiplex：用户和item之间可以浏览、点击、添加到购物车、购买等

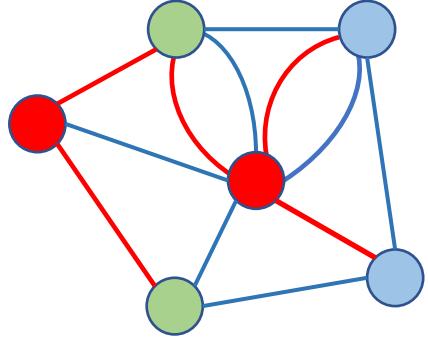


## GATNE : General Attributed Multiplex HeTerogeneous Network Embedding

本文提出General Attributed Multiplex Heterogeneous Network Embedding(GATNE)，希望每个节点在不同类型边中有不同的表示，比如说用户A在点击商品的场景下学习一种向量表示，在购买商品的场景下学习另一种向量表示，而不同场景之间并不完全独立，希望用base embedding来当作不同类型关系传递信息的桥梁，我们综合base embedding与每一类型边的edge embedding来进行建模，在直推式学习(Transductive)背景下，提出GATNE-T模型，在归纳式学习(Inductive)背景下，考虑节点特征，提出GATNE-I模型。

模型结构如下图所示，网络结构利用两部分向量Base Embedding和Edge embedding表示，其中Base Embedding为共享向量，出现在每一种边类型中；Edge embedding在每一种边类型中不同；GATNE-T仅仅利用了网络结构信息，GATNE-I同时考虑了网络结构信息和节点性质。

## 4.1 Transductive Model: GATNE-T



Base embedding

Edge embedding

红色type (评论场景下) 图节点的embedding

$$Edge\_V_{red,i} = embedding$$

蓝色type (购买场景下) 图节点的embedding

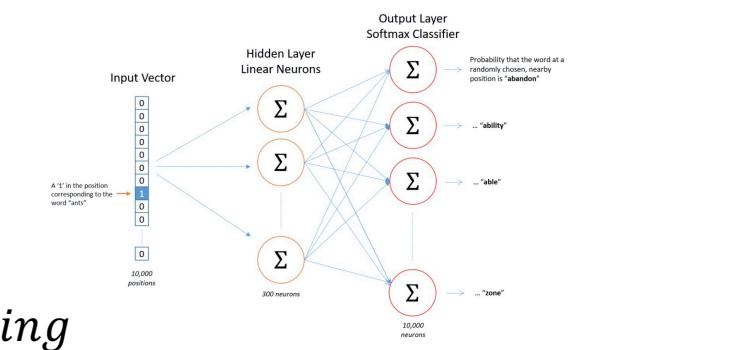
$$Edge\_V_{blue,i} = embedding$$

$$V_{i,blue} = Base\_V_i + Edge\_V_{blue,i}$$

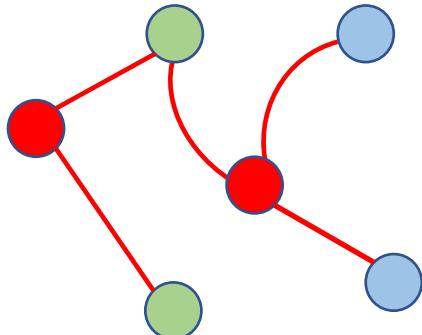
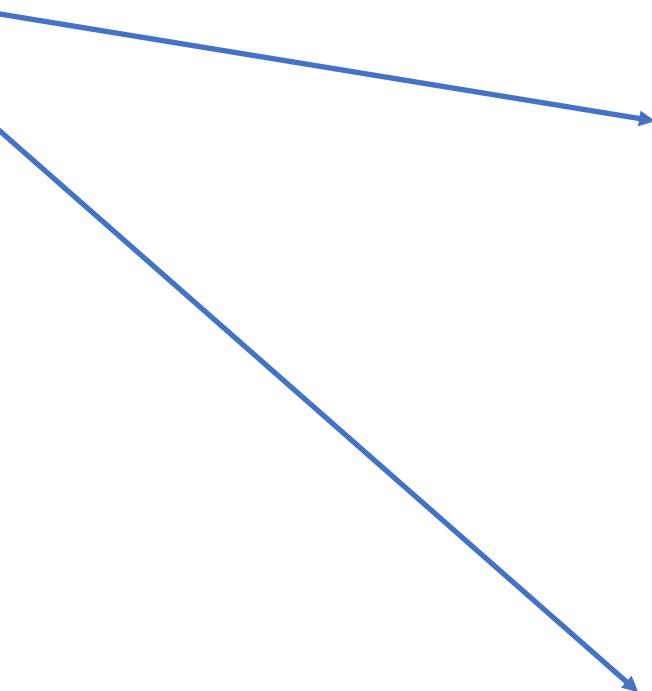
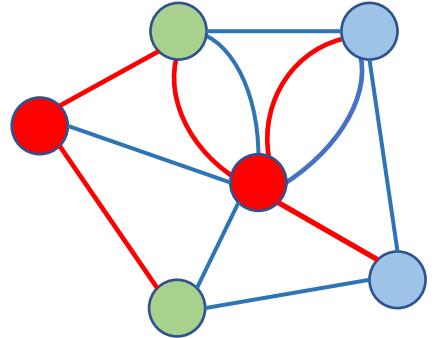
$$V_{i,red} = Base\_V_i + Edge\_V_{red,i}$$

生成节点的embedding  $b_i$

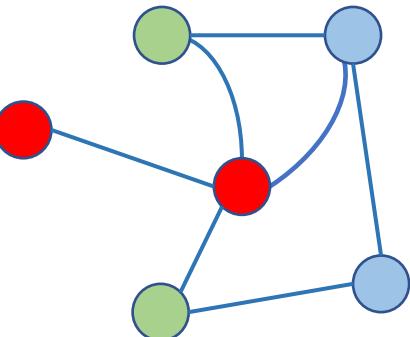
源码：查表方式生成embedding



我要计算红色type (评论场景下) 的节点embedding

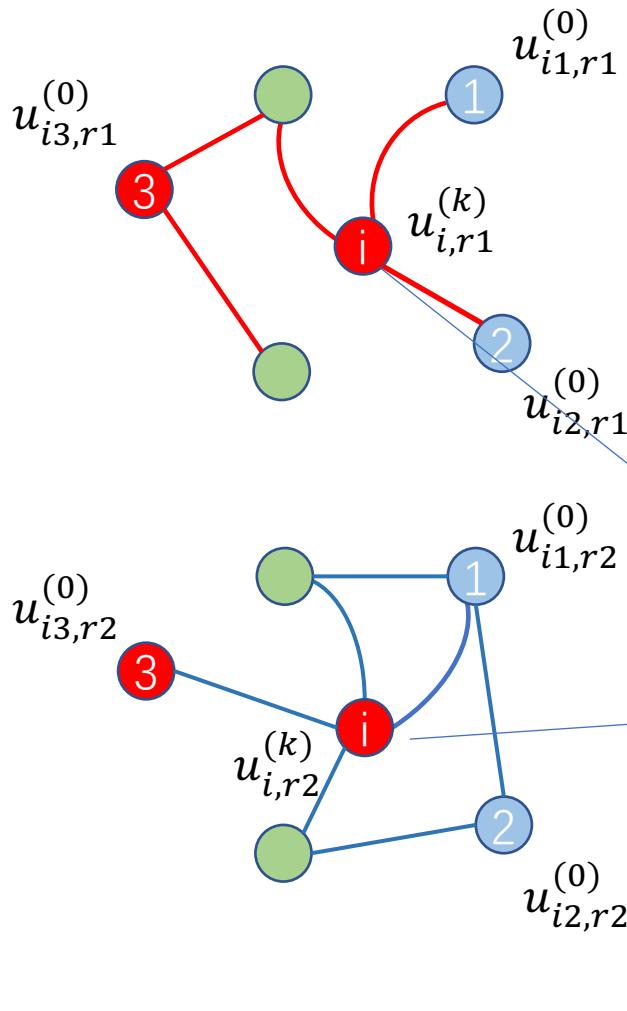


按照红色type生成的图  
(评论)



按照蓝色type生成的图  
(购买)

红色类别下每个节点的embedding

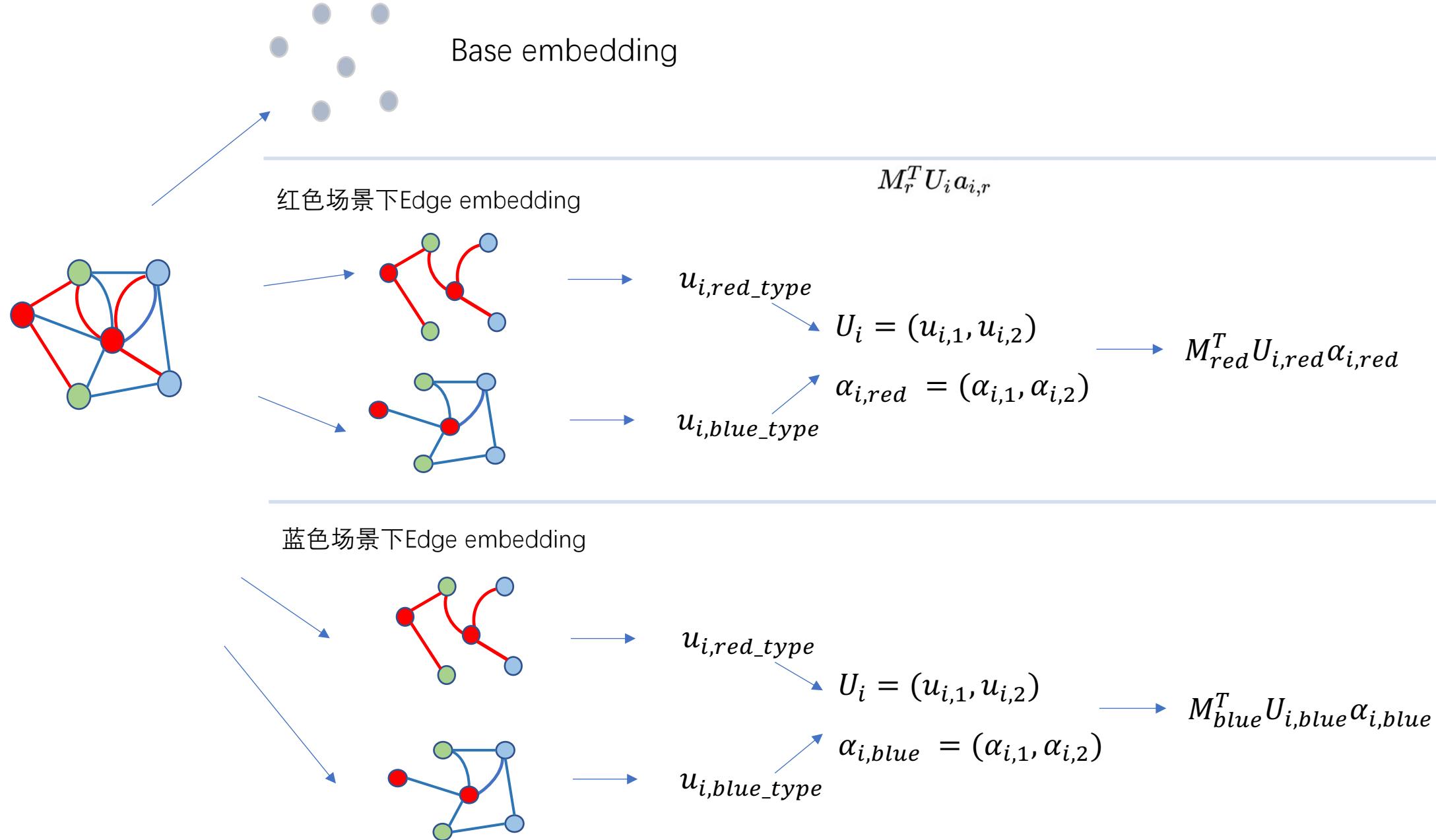


- Step0:  $u_{i1,r1}^{(0)}$  初始化每个类别下节点的embedding
- Step1: 类似于Graphsage对邻居聚合的思想, 节点  $v_i$  对边类型为r的第k阶邻居进行聚合, 得到 edge embedding  $u_{i,r}^{(k)}$  :
$$\mathbf{u}_{i,r}^{(k)} = \text{aggregator}(\{\mathbf{u}_{j,r}^{(k-1)}, \forall v_j \in \mathcal{N}_{i,r}\}), \quad (1)$$
- Step2: 把第k阶邻居, 不同类型的边节点, 对应的edge embedding进行concat聚合:
$$U_i = (u_{i,1}, u_{i,2}, \dots, u_{i,m})$$
- Step3: 考虑到不同类型边的影响不同, 利用注意力机制计算权重:
$$a_{i,r} = \text{softmax}(w_r^T \tanh(W_r U_i))^T, \text{ 其中 } w_r \in R^{d_a}, W_r \in R^{d_a * s} \text{ 为要学习的参数}$$
$$\alpha_{i,r} = (\alpha_{i,1}, \alpha_{i,2})$$
- Step4: 计算Edge embedding  $M_r^T U_i a_{i,r}$

$M_r$  –  $r$ 场景下可学习的参数

· Step5: 综合base embedding和edge embedding得到最终节点  $v_i$  边类型r的节点向量表示:

$v_{i,r} = b_i + \alpha_r M_r^T U_i a_{i,r}$  , 其中  $b_i$  为节点i的base embedding,  $\alpha_r$  为超参数控制edge embedding的重要程度



## 4.3 Model Optimization



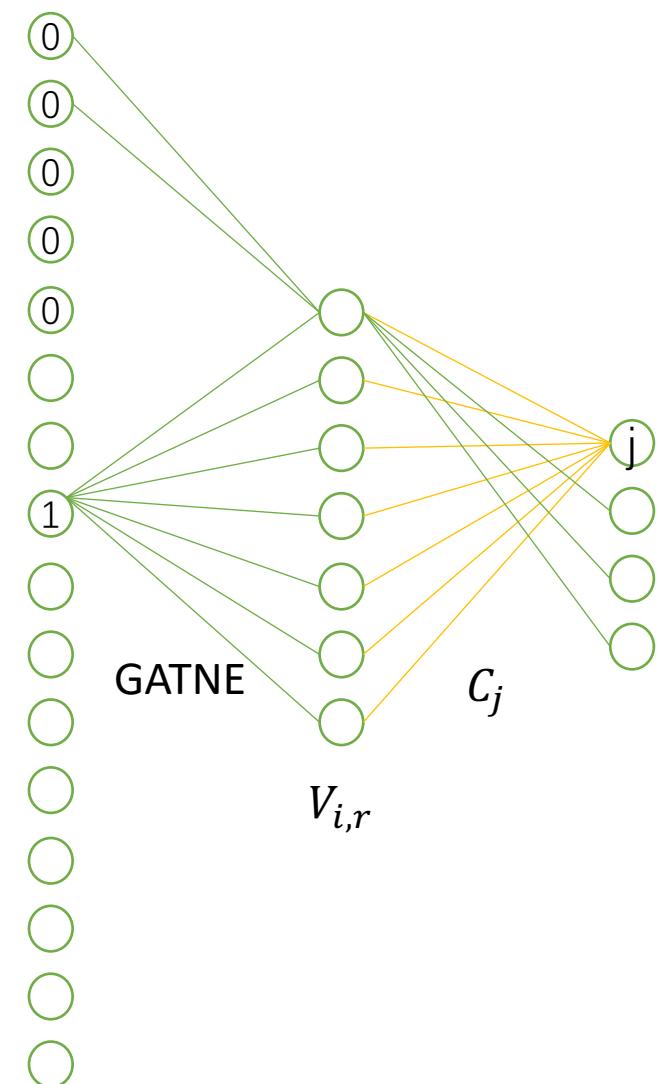
$$p(v_j|v_i, \mathcal{T}) = \begin{cases} \frac{1}{|\mathcal{N}_{i,r} \cap \mathcal{V}_{t+1}|} & (v_i, v_j) \in \mathcal{E}_r, v_j \in \mathcal{V}_{t+1}, \\ 0 & (v_i, v_j) \in \mathcal{E}_r, v_j \notin \mathcal{V}_{t+1}, \\ 0 & (v_i, v_j) \notin \mathcal{E}_r, \end{cases} \quad (14)$$

Metapath方式生成skip-gram上下文

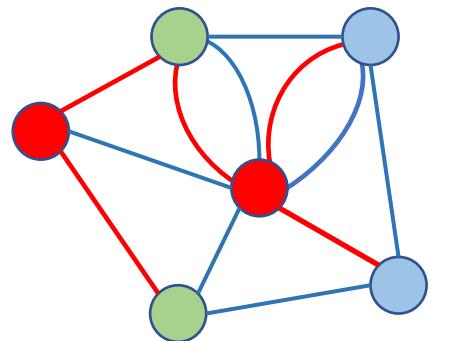
$$E = -\log \sigma(\mathbf{c}_j^T \cdot \mathbf{v}_{i,r}) - \sum_{l=1}^L \mathbb{E}_{v_k \sim P_t(v)} [\log \sigma(-\mathbf{c}_k^T \cdot \mathbf{v}_{i,r})], \quad (17)$$

$\mathbf{c}_j$  – 要学习的参数

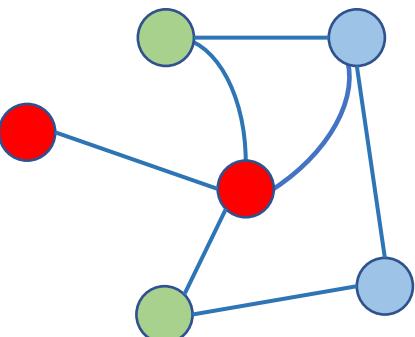
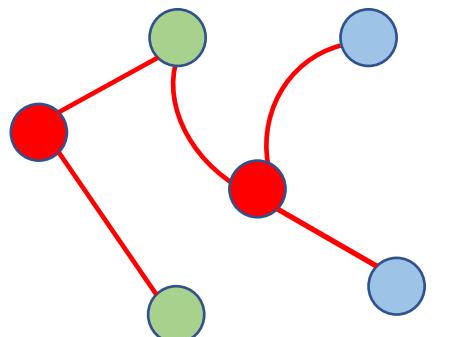
负采样方式构造损失函数



Predict



Base embedding



Edge embedding

## 4.2 Inductive Model: **GATNE-I**

$$\mathbf{v}_{i,r} = \mathbf{b}_i + \alpha_r \mathbf{M}_r^T \mathbf{U}_i \mathbf{a}_{i,r}, \quad (6) \quad : \mathbf{GATNE-T}$$

$$\mathbf{v}_{i,r} = \mathbf{h}_z(\mathbf{x}_i) + \alpha_r \mathbf{M}_r^T \mathbf{U}_i \mathbf{a}_{i,r} + \beta_r \mathbf{D}_z^T \mathbf{x}_i, \quad (13) \quad : \mathbf{GATNE-I}$$

$$\mathbf{b}_i = \mathbf{h}_z(\mathbf{x}_i)$$

$$\mathbf{u}_{i,r}^{(0)} = \mathbf{g}_{z,r}(\mathbf{x}_i)$$

$\beta_r$  is a coefficient and  $\mathbf{D}_z$  is a feature transformation matrix  
on  $v_i$ 's corresponding node type  $z$ .

Features = None

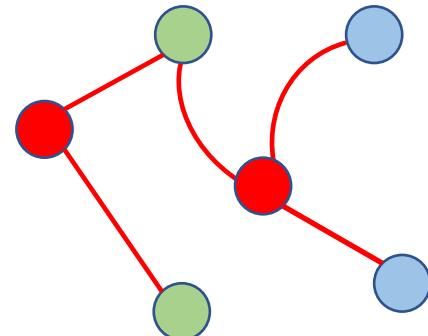
training\_data\_by\_type = load\_training\_data(file\_name + "/train.txt") Training data

train\_model(training\_data\_by\_type, feature\_dic) Main program

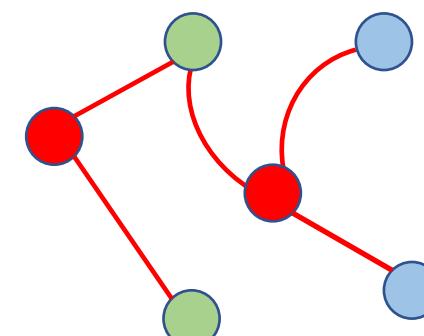
生成游走序列

vocab, index2word, train\_pairs = generate(network\_data, args.num\_walks, args.walk\_length, args.schema, file\_name, args.window\_size, args.num\_workers,

每个节点对应到的信息  
训练样本, skip-gram  
节点



按照类别进行随机游走  
生成上下文节点



按类别进行邻居采样

neighbors = generate\_neighbors(network\_data, vocab, num\_nodes, edge\_types, neighbor\_samples)

## GATNE

$$\mathbf{a}_{i,r} = \text{softmax}(\mathbf{w}_r^T \tanh(\mathbf{W}_r \mathbf{U}_i))^T,$$

```
model = GATNEModel(
```

num_nodes, embedding_size, embedding_u_size, edge_type_count, dim_a, features
-------------------------------------------------------------------------------

```
)
```

$u_{i1,r1}^{(0)}$

10

2

20

None

节点数

Embedding size=200

Base embedding :  $\mathbf{b}_i$

```
self.node_embeddings = Parameter(torch.FloatTensor(num_nodes, embedding_size)) # [511, 200]
```

$u_{i1,r1}^{(0)}$

```
self.node_type_embeddings = Parameter(
    torch.FloatTensor(num_nodes, edge_type_count, embedding_u_size) # [511, 2, 10]
) # [511, 2, 10]
```

$\mathbf{u}_{i,r}^{(k)}$

[i, r, embedding]

```
self.trans_weights = Parameter( # [2, 10, 200]
    torch.FloatTensor(edge_type_count, embedding_u_size, embedding_size)
)
```

可训练参数

```
self.trans_weights_s1 = Parameter( # [2, 10, 20]
    torch.FloatTensor(edge_type_count, embedding_u_size, dim_a)
)
```

```
self.trans_weights_s2 = Parameter(torch.FloatTensor(edge_type_count, dim_a, 1)) # [2, 20, 1]
```

```

class NSLoss(nn.Module):
    def __init__(self, num_nodes, num_sampled, embedding_size):
        self.weights = Parameter(torch.FloatTensor(num_nodes, embedding_size)) # [511, 200]
        self.sample_weights = F.normalize( # [511]; 对节点进行初始化, 求聚合边的信息

```

`def get_batches(pairs, neighbors, batch_size):` 得到训练的batch样本

`yield torch.tensor(x), torch.tensor(y), torch.tensor(t), torch.tensor(neigh)` [center, context, type, neigh]

Forward:

`node_embed = self.node_embeddings[train_inputs]` [511, 200] [64] => [64, 200] 查表对应的node的embedding

`node_embed_neighbors = self.node_type_embeddings[node_neigh]` [511, 2, 10] [节点邻居:64, 2, 10] => [64, 2, 10, 2, 10]

`node_embed_tmp = torch.cat( # [64, 2, 10, 10]` 64 batch, 2种类别, neigh 10个节点, 每个10维特征

`node_type_embed = torch.sum(node_embed_tmp, dim=2) # [64, 2, 10]` 10个邻居求和, 聚合邻居节点

$$\mathbf{U}_i = (\mathbf{u}_{i,1}, \mathbf{u}_{i,2}, \dots, \mathbf{u}_{i,m}). \quad (4)$$

邻居节点信息

```
trans_w = self.trans_weights[train_types] # [64, 10, 200] Train_types 节点类型
trans_w_s1 = self.trans_weights_s1[train_types] # [64, 10, 20]
trans_w_s2 = self.trans_weights_s2[train_types] # [64, 20, 1]

    self.trans_weights = Parameter( # [2, 10, 200]
        torch.FloatTensor(edge_type_count, embedding_u_size, embedding_size)
    self.trans_weights_s1 = Parameter( # [2, 10, 20]
        torch.FloatTensor(edge_type_count, embedding_u_size, dim_a)
    self.trans_weights_s2 = Parameter(torch.FloatTensor(edge_type_count, dim_a, 1)) # [2, 20, 1]
```

```
>>> train_types
tensor([0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
        0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0,
        0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0]) 按照0, 1位置复制trans_weights
>>> trans_w[0]==trans_w[2]
```

```
attention = F.softmax( # [64, 1, 2]
```

```
    torch.matmul(  
        torch.tanh(torch.matmul(node_type_embed, trans_w_s1)), trans_w_s2  
    ).squeeze(2),  
    dim=1,  
) .unsqueeze(1)
```

[64, 20, 1]

[64, 2, 10] [64, 10, 20]

$$\mathbf{U}_i = (\mathbf{u}_{i,1}, \mathbf{u}_{i,2}, \dots, \mathbf{u}_{i,m}). \quad (4)$$

节点邻居的embedding => [64, 2, 20]

[64, 2, 20]\*[64, 20, 1] => [64, 2, 1]

[64, 2] 节点在每个type上的attention值

[64, 1, 2]

$$\mathbf{a}_{i,r} = \text{softmax}(\mathbf{w}_r^T \tanh(\mathbf{W}_r \mathbf{U}_i))^T, \quad (5)$$

$\mathbf{U}_i \mathbf{a}_i$

```
node_type_embed = torch.matmul(attention, node_type_embed) # [64, 1, 2] * [64, 2, 10]
```

[64, 1, 10] 每个类别求attention后的值

[64, 1, 10] [64, 10, 200]

```
node_embed = node_embed + torch.matmul(node_type_embed, trans_w).squeeze(1)
```

$$\mathbf{v}_{i,r} = \mathbf{b}_i + \alpha_r \mathbf{M}_r^T \mathbf{U}_i \mathbf{a}_{i,r}, \quad (6)$$

直接查表

在某个类别下的embedding  
这有m个Mr



Node embedding [64, 200]

这里直接输出一个Mr, 即输出  
是节点的一个embedding

Loss

Center, embedding, context

```
def forward(self, input, embs, label):
```

```
    log_target = torch.log( # torch.mul:对应位置相乘
```

```
        torch.sigmoid(torch.sum(torch.mul(embs, self.weights[label]), 1))
```

```
)| [64, 200] [511, 200]=>[64,200]
```

$$E = -\log \sigma(\mathbf{c}_j^T \cdot \mathbf{v}_{i,r}) - \sum_{l=1}^L \mathbb{E}_{v_k \sim P_t(v)} [\log \sigma(-\mathbf{c}_k^T \cdot \mathbf{v}_{i,r})], \quad (17)$$

```
negs = torch.multinomial( # 抽样函数
```

```
    self.sample_weights, self.num_sampled * n, replacement=True
).view(n, self.num_sampled)
```

每个节点的随机初始值

```
noise = torch.neg(self.weights[negs]) # 所有值 * -1
```

抽出节点对应到的查表值

$C_j$

```
sum_log_sampled = torch.sum(
```

```
    torch.log(torch.sigmoid(torch.bmm(noise, embs.unsqueeze(2)))), 1 # [64, 5, 1]
```

```
).squeeze() [64,5,200] [64,200]=>[64,200,1]
```

```
loss = log_target + sum_log_sampled
```

```
return -loss.sum() / n
```

Loss:

```
log_target = torch.log( # torch.mul:对应位置相乘
    torch.sigmoid(torch.sum(torch.mul(embs, self.weights[label])), 1))
)
```

$$E = -\log \sigma(\mathbf{c}_j^T \cdot \mathbf{v}_{i,r}) - \sum_{l=1}^L \mathbb{E}_{v_k \sim P_t(v)} [\log \sigma(-\mathbf{c}_k^T \cdot \mathbf{v}_{i,r})], \quad (17)$$



```
noise = torch.neg(self.weights[negs]) # 所有值 * -1
```

```
sum_log_sampled = torch.sum(
    torch.log(torch.sigmoid(torch.bmm(noise, embs.unsqueeze(2)))), 1) # [64, 5, 1]
).squeeze()
```

GATNE-I

args.features

return 每个节点特征dictionary

```
features = None features: None
```

```
if feature_dic is not None:
```

```
    feature_dim = len(list(feature_dic.values())[0]) # 特征长度
```

```
if features is not None:
```

```
    self.features = features
```

```
    feature_dim = self.features.shape[-1]
```

```
    self.embed_trans = Parameter(torch.FloatTensor(feature_dim, embedding_size)) # [142, 200]
```

```
    self.u_embed_trans = Parameter(torch.FloatTensor(edge_type_count, feature_dim, embedding_u_size)) # [2, 142, 10]
```

```
else:
```

```
    self.node_embeddings = Parameter(torch.FloatTensor(num_nodes, embedding_size)) # [511, 200]
```

```
    self.node_type_embeddings = Parameter(
```

```
        torch.FloatTensor(num_nodes, edge_type_count, embedding_u_size)
```

```
) # [511, 2, 10]
```

```
self.embed_trans = Parameter(torch.FloatTensor(feature_dim, embedding_size)) # [142, 200] h_z(x_i)
```

```
self.u_embed_trans = Parameter(torch.FloatTensor(edge_type_count, feature_dim, embedding_u_size)) # [2, 142, 10]
```

$$\mathbf{u}_{i,r}^{(0)} = \mathbf{g}_{z,r}(\mathbf{x}_i),$$

```
def forward(self, train_inputs, train_types, node_neigh):    self: GATNEModel()  train_inputs: tensor([224, 142,  
if self.features is None:  
    node_embed = self.node_embeddings[train_inputs]  
    node_embed_neighbors = self.node_type_embeddings[node_neigh]  
else: # self.features:节点特征; self.embed_trans  
    node_embed = torch.mm(self.features[train_inputs], self.embed_trans) # [64, 142]  
    node_embed_neighbors = torch.einsum('bijk,akm->bijam', self.features[node_neigh], self.u_embed_trans)
```

```
node_embed = torch.mm(self.features[train_inputs], self.embed_trans) # [64, 142] h_z(x_i)
```

```
node_embed_neighbors = torch.einsum('bijk,akm->bijam', self.features[node_neigh], self.u_embed_trans)
```

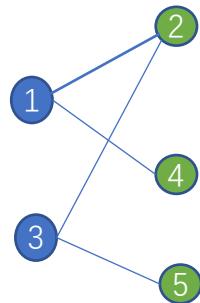
[64, 2, 10, 142] [2, 142, 10] =>[64, 2, 10, 2, 10]

二部图

二部图可以看做是特殊的异构图，可以使用异构图的算法去解决二部图的问题

我们使用异构图方法去解决二部图问题，一个关键限制是它将显式和隐式关系视为同等重要的。

但是实际应用中，应该赋予他们不同的权重，在学习过程中灵活调整。



不同类别之间的连接情况

显式关系： $1 - 2; 1 - 4; 3 - 2; 3 - 5$

隐式关系： $1 - 3; 2 - 4; 2 - 5;$

同类别之间的连接情况

具体而言，显式关系的建模旨在通过关注观察到的链接来构建。  
对于隐式关系的建模，我们旨在捕获二部网络中的高阶相关性。

## 4 BINE: BIPARTITE NETWORK EMBEDDING

通过观察到的显式关系和未观察到的隐式关系，通过优化显式和隐式两个任务来共同学习顶点的embedding向量

参考Graph Embedding : LINE

### 4.1 Modeling Explicit Relations

显式关系建模

$$P(i, j) = \frac{w_{ij}}{\sum_{e_{ij} \in E} w_{ij}}. \quad (1)$$

$$\hat{P}(i, j) = \frac{1}{1 + \exp(-\vec{u}_i^T \vec{v}_j)}. \quad (2)$$

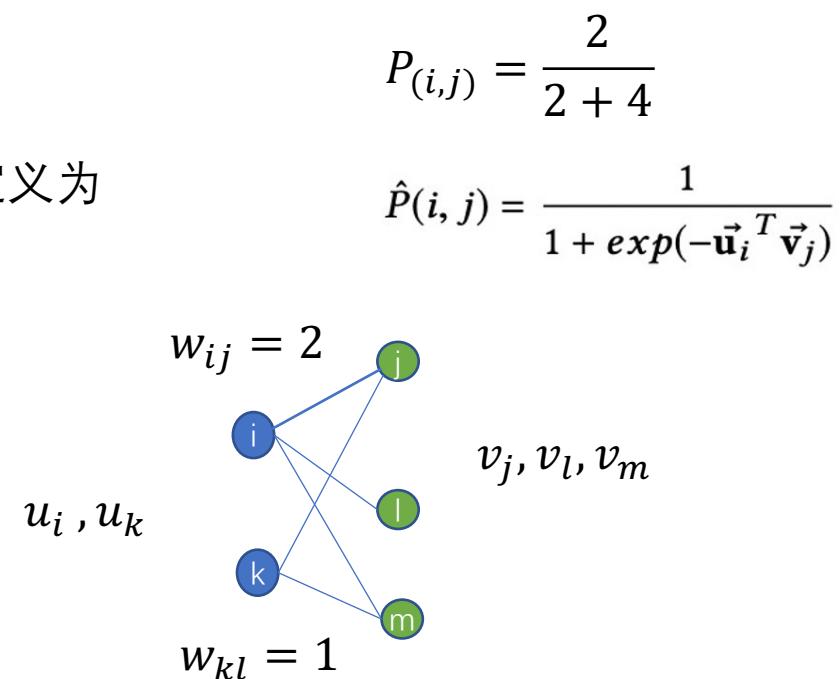
i, j连接的显式关系，定义为

估计embedding中两个顶点之间的局部接近度

Object : 最小化两个分布的距离

which can be defined as:

$$\begin{aligned} \text{minimize} \quad O_1 &= KL(P || \hat{P}) = \sum_{e_{ij} \in E} P(i, j) \log\left(\frac{P(i, j)}{\hat{P}(i, j)}\right) \\ &\propto - \sum_{e_{ij} \in E} w_{ij} \log \hat{P}(i, j). \end{aligned} \quad (3)$$



## 4.2 Modeling Implicit Relations 隐式关系建模

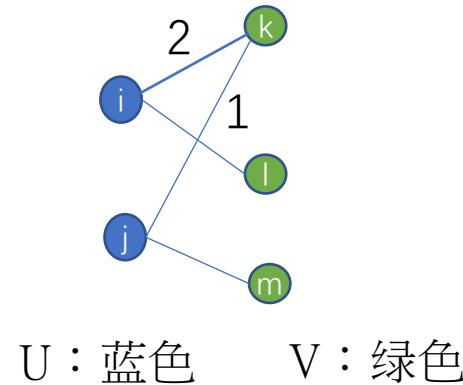
对于两个相同类型的顶点，如果它们之间存在路径，则它们之间应该存在一定的隐式关系。

### 4.2.1 Constructing Corpus of Vertex Sequences.

$$w_{ij}^U = \sum_{k \in V} w_{ik} w_{jk}; \quad w_{ij}^V = \sum_{k \in U} w_{ki} w_{kj}.$$

$$\begin{aligned} i \rightarrow j: & i - k - j \\ &= 2 * 1 \\ &= 2 \end{aligned}$$

$$\begin{array}{cccc} & k & l & m \\ i & 2 & 1 & 0 \\ j & 1 & 0 & 1 \\ \text{adj} & & & \end{array}$$



两个顶点的二阶相似性（隐式关系）定义为  
(4)

$$\begin{array}{ccc} 2 & 1 & 0 \\ 1 & 0 & 1 \\ \hline \end{array} \cdot \begin{array}{ccc} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & \end{array} = \begin{array}{cc} 5 & 2 \\ 2 & 2 \end{array}$$

$$\begin{array}{ccc} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & \end{array} \cdot \begin{array}{ccc} 2 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 2 \end{array} = \begin{array}{ccc} 5 & 2 & 1 \\ 2 & 1 & 0 \\ 1 & 0 & 2 \end{array}$$

我们构造了两个同构网络，在该网络中执行随机游走，就构造了相同类型顶点之间的二阶接近度。

首先，我们将从每个顶点开始的随机游走次数与它的最重要性相关联，这可以通过其中心性来衡量。对于顶点，其中心性越大，随机游走的可能性就越大。结果，可以在一定程度上保留顶点重要性。

我们指定在每个步骤中停止随机游走的概率。与DeepWalk等[14]在随机游走上应用固定长度的方法相反，我们允许生成的顶点序列具有可变长度，以便与自然语言中的可变长度句子非常相似。

---

**Algorithm 1:** WalkGenerator( $W, R, maxT, minT, p$ )

---

**Input** : weight matrix of the bipartite network  $\mathbf{W}$ , vertex set  $R$  (can be  $U$  or  $V$ ), maximal walks per vertex  $maxT$ , minimal walks per vertex  $minT$ , walk stopping probability  $p$

**Output**: a set of vertex sequences  $D^R$

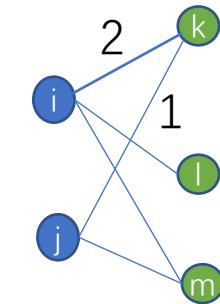
- 1 Calculate vertices' centrality:  $\mathbf{H} = CentralityMeasure(\mathbf{W})$ ;
- 2 Calculate  $W^R$  w.r.t. Equation (4);
- 3 **foreach** vertex  $v_i \in R$  **do**
- 4    $l = \max(\mathbf{H}(v_i) \times maxT, minT)$ ;
- 5   **for**  $i = 0$  **to**  $l$  **do**
- 6      $D_{v_i} = BiasedRandomWalk(W^R, v_i, p)$ ;
- 7     Add  $D_{v_i}$  into  $D^R$ ;
- 8 **return**  $D^R$ ;

## 4.2.2 Implicit Relation Modeling.

分别在两个同构网络上执行有偏随机游走后，我们获得了两个顶点序列语料库。采用skip-gram和负采样计算节点的embedding

$$\text{maximize } O_2 = \prod_{u_i \in S \wedge S \in D^U} \prod_{u_c \in C_S(u_i)} P(u_c | u_i). \quad (5) \quad D^U:$$

$$\text{maximize } O_3 = \prod_{v_j \in S \wedge S \in D^V} \prod_{v_c \in C_S(v_j)} P(v_c | v_j). \quad (6) \quad D^V:$$



U : 蓝色      V : 绿色

$$P(u_c | u_i) = \frac{\exp(\vec{u}_i^T \vec{\theta}_c)}{\sum_{k=1}^{|U|} \exp(\vec{u}_i^T \vec{\theta}_k)}, \quad P(v_c | v_j) = \frac{\exp(\vec{v}_j^T \vec{\vartheta}_c)}{\sum_{k=1}^{|V|} \exp(\vec{v}_j^T \vec{\vartheta}_k)}. \quad (7)$$

$P(u_c | u_i)$      $U_i$  观察到上下文  $U_c$  的可能性

所有的节点概率  $\sum_{k=1}^{|U|} \exp(\vec{u}_i^T \vec{\theta}_k)$

### 4.2.3 Negative Sampling.

probability  $p(u_c|u_i)$  defined in Equation (7) as:

$$p(u_c, N_S^{ns}(u_i)|u_i) = \prod_{z \in \{u_c\} \cup N_S^{ns}(u_i)} P(z|u_i), \quad (8) \quad \text{近似概率}$$

$N_S^{ns}(u_i)$  denote the  $ns$  negative samples for a center vertex

where the probability  $P(z|u_j)$  is defined as:

$$P(z|u_i) = \begin{cases} \sigma(\vec{\mathbf{u}}_i^T \vec{\boldsymbol{\theta}}_z), & \text{if } z \text{ is a context of } u_i \\ 1 - \sigma(\vec{\mathbf{u}}_i^T \vec{\boldsymbol{\theta}}_z), & z \in N_S^{ns}(u_i) \end{cases}$$

## 4.3 Joint Optimization

$$\text{maximize } L = \alpha \log O_2 + \beta \log O_3 - \gamma O_1. \quad (9) \quad \text{超参数}$$

**Step I:** For a stochastic explicit relation, i.e., an edge  $e_{ij} \in E$ , we first update the embedding vectors  $\vec{\mathbf{u}}_i$  and  $\vec{\mathbf{v}}_j$  by utilizing SGA to maximize the last component  $L_1 = -\gamma O_1$ . We give the SGA update rule for  $\vec{\mathbf{u}}_i$  and  $\vec{\mathbf{v}}_j$  as follows:

$$\vec{\mathbf{u}}_i = \vec{\mathbf{u}}_i + \lambda \{ \gamma w_{ij} [1 - \sigma(\vec{\mathbf{u}}_i^T \vec{\mathbf{v}}_j)] \cdot \vec{\mathbf{v}}_j \}, \quad (10)$$

$$\vec{\mathbf{v}}_j = \vec{\mathbf{v}}_j + \lambda \{ \gamma w_{ij} [1 - \sigma(\vec{\mathbf{u}}_i^T \vec{\mathbf{v}}_j)] \cdot \vec{\mathbf{u}}_i \}, \quad (11)$$

**Step II:**

$$\vec{\mathbf{u}}_i = \vec{\mathbf{u}}_i + \lambda \{ \sum_{z \in \{u_c\} \cup N_S^{ns}(u_i)} \alpha [I(z, u_i) - \sigma(\vec{\mathbf{u}}_i^T \vec{\theta}_z)] \cdot \vec{\theta}_z \} \quad (12)$$

$$\vec{\mathbf{v}}_j = \vec{\mathbf{v}}_j + \lambda \{ \sum_{z \in \{v_c\} \cup N_S^{ns}(v_j)} \beta [I(z, v_j) - \sigma(\vec{\mathbf{v}}_j^T \vec{\vartheta}_z)] \cdot \vec{\vartheta}_z \} \quad (13)$$

which can be defined as:

$$\begin{aligned} \text{minimize } O_1 &= KL(P || \hat{P}) = \sum_{e_{ij} \in E} P(i, j) \log \left( \frac{P(i, j)}{\hat{P}(i, j)} \right) \\ &\propto - \sum_{e_{ij} \in E} w_{ij} \log \hat{P}(i, j). \end{aligned} \quad (3)$$

更新显示关系

$$\text{maximize } O_2 = \prod_{u_i \in S \wedge S \in D^U} \prod_{u_c \in C_S(u_i)} P(u_c | u_i). \quad (5)$$

$$\text{maximize } O_3 = \prod_{v_j \in S \wedge S \in D^V} \prod_{v_c \in C_S(v_j)} P(v_c | v_j). \quad (6)$$

更新隐式关系

where  $I(z, u_i)$  is an indicator function that determines whether vertex  $z$  is in the context of  $u_i$  or not; similar meaning applies

## 4.4 Discussions

超参数的选择：

**Table 2: The search range and optimal setting (highlighted in red) of hyper-parameters for our BiNE method.**

Parameter	Meaning	Test values
$ns$	number of negative samples	[1, 2, 4, 6, 8, 10]
$ws$	size of window	[1, 3, 5, 7, 9]
$p$	walk stopping probability	[0.05, 0.1, 0.15, 0.2, 0.3, 0.4, 0.5]
$\beta$	trade-off parameter	[0.0001, 0.001, 0.01, 0.1, 1]
$\gamma$	trade-off parameter	[0.01, 0.05, 0.1, 0.5, 1, 5]

For our BiNE, we fix the loss trade-off parameter  $\alpha$  as 0.01 and tune the other two. The  $minT$  and  $maxT$  are respectively set to 1 and 32,

$$\text{maximize } L = \alpha \log O_2 + \beta \log O_3 - \gamma O_1. \quad (9)$$

RQ1：BiNE和现有方法相比，表现如何？

RQ2：隐式关系的建模是否有助于学习二部网络更好的表示？

RQ3：我们提出的随机漫步生成器能帮助学习更好的顶点表示吗？

RQ4：关键的超参数如何影响BiNE的性能？

RQ1：BiNE和现有方法相比，表现如何？

**Table 3: Link prediction performance on Tencent and Wikipedia.**

Algorithm	Tencent		Wikipedia	
	AUC-ROC	AUC-PR	AUC-ROC	AUC-PR
<b>CN</b>	50.63%	65.66%	86.85%	90.68%
<b>JC</b>	51.49%	66.18%	63.90%	73.04%
<b>AA</b>	50.63%	65.66%	87.37%	91.12%
<b>AL</b>	50.44%	65.70%	90.28%	91.81%
<b>Katz</b>	50.90%	65.06%	90.84%	92.42%
<b>PA</b>	55.60%	68.99%	90.71%	93.37%
<b>DeepWalk</b>	57.62%	71.32%	89.71%	91.20%
<b>LINE</b>	59.68%	73.48%	91.62%	93.28%
<b>Node2vec</b>	59.28%	72.62%	89.93%	91.23%
<b>Metapath2vec++</b>	60.70%	73.69%	89.56%	91.72%
<b>BiNE</b>	<b>60.98%**</b>	<b>73.77%**</b>	<b>92.91%**</b>	<b>94.45%**</b>

\*\* indicates that the improvements are statistically significant for  $p < 0.01$   
judged by paired t-test.

**Table 4: Performance comparison of Top-10 Recommendation on VisualizeUs, DBLP, and MovieLens.**

Algorithm	VisualizeUs				DBLP				Movielens			
	F1@10	NDCG@10	MAP@10	MRR@10	F1@10	NDCG@10	MAP@10	MRR@10	F1@10	NDCG@10	MAP@10	MRR@10
BPR	6.22%	9.52%	5.51%	13.71%	8.95%	18.38%	13.55%	22.25%	8.03%	7.58%	2.23%	40.81%
RankALS	2.72%	3.29%	1.50%	3.81%	7.62%	11.50%	7.52%	14.87%	8.48%	7.95%	2.66%	38.93%
FISMauc	10.25%	15.46%	8.86%	16.67%	9.81%	13.77%	7.38%	14.51%	6.77%	6.13%	1.63%	34.04%
DeepWalk	5.82%	8.83%	4.28%	12.12%	8.50%	24.14%	19.71%	31.53%	3.73%	3.21%	0.90%	15.40%
LINE	9.62%	13.76%	7.81%	14.99%	8.99%	14.41%	9.62%	17.13%	6.91%	6.50%	1.74%	38.12%
Node2vec	6.73%	9.71%	6.25%	13.95%	8.54%	23.89%	19.44%	31.11%	4.16%	3.68%	1.05%	18.33%
Metapath2vec++	5.92%	8.96%	5.35%	13.54%	8.65%	25.14%	19.06%	31.97%	4.65%	4.39%	1.91%	16.60%
BiNE	13.63%**	24.50%**	16.46%**	34.23%**	11.37%**	26.19%**	20.47%**	33.36%**	9.14%**	9.02%**	3.01%**	45.95%**

\*\* indicates that the improvements are statistically significant for  $p < 0.01$  judged by paired t-test.

RQ2：隐式关系的建模是否有助于学习二部网络更好的表示？

**Table 5: BiNE with and without implicit relations.**

	Without Implicit Relations	With Implicit Relations	Link Prediction	
Dataset	AUC-ROC	AUC-PR	AUC-ROC	AUC-PR
Tencent	59.78%	73.05%	<b>60.98%**</b>	<b>73.77%**</b>
WikiPedia	91.47%	93.73%	<b>92.91%**</b>	<b>94.45%**</b>
Recommendation				
Dataset	MAP@10	MRR@10	MAP@10	MRR@10
VisualizeUS	7.91%	15.65%	<b>16.46%**</b>	<b>34.23%**</b>
DBLP	20.20%	32.95%	<b>20.47%**</b>	<b>33.36%**</b>
MovieLens	2.86%	43.98%	<b>3.01%**</b>	<b>45.95%**</b>

\*\* indicates that the improvements are statistically significant for  $p < 0.01$   
judged by paired t-test.

是否存在隐式关系

RQ3：我们提出的随机漫步生成器能帮助学习更好的顶点表示吗？

**Table 6: BiNE with different random walk generators.**

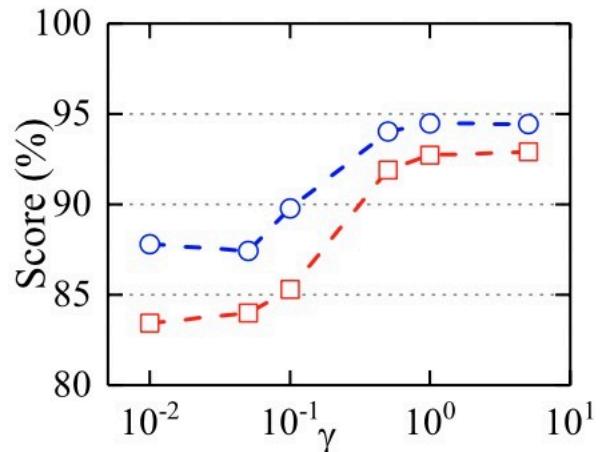
I	Uniform Random Walk Generator	Biased and Self-adaptive Random Walk Generator		
<b>Link Prediction</b>				
Dataset	AUC-ROC	AUC-PR	AUC-ROC	AUC-PR
Tencent	59.75%	73.06%	<b>60.98%**</b>	<b>73.77%**</b>
WikiPedia	88.77%	91.91%	<b>92.91%**</b>	<b>94.45%**</b>
<b>Recommendation</b>				
Dataset	MAP@10	MRR@10	MAP@10	MRR@10
VisualizeUS	15.93%	33.66%	<b>16.46%**</b>	<b>34.23%**</b>
DBLP	11.79%	23.41%	<b>20.47%**</b>	<b>33.66%**</b>
MovieLens	2.91%	46.12%	<b>3.04%**</b>	<b>46.20%**</b>

\*\* indicates that the improvements are statistically significant for  $p < 0.01$   
judged by paired t-test.

这表明有偏和自适应的随机游走发生器有助于改善顶点嵌入。

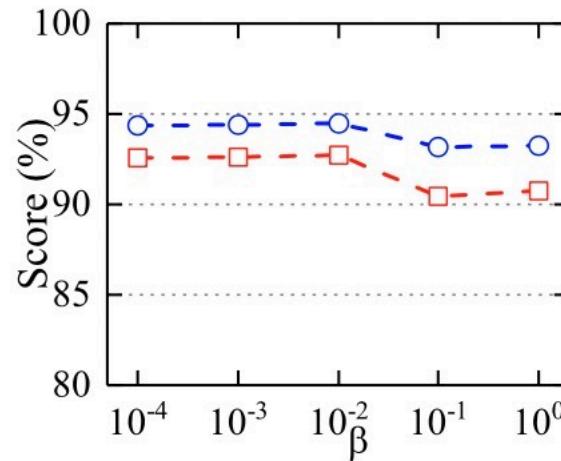
## RQ4：关键的超参数如何影响BiNE的性能？

we fix  $\alpha = 0.01$



(a) Performance VS.  $\gamma$

— □ — AUC-ROC    — ○ — AUC-PR



(b) Performance VS.  $\beta$

$\beta$ 增大，夸大隐式关系

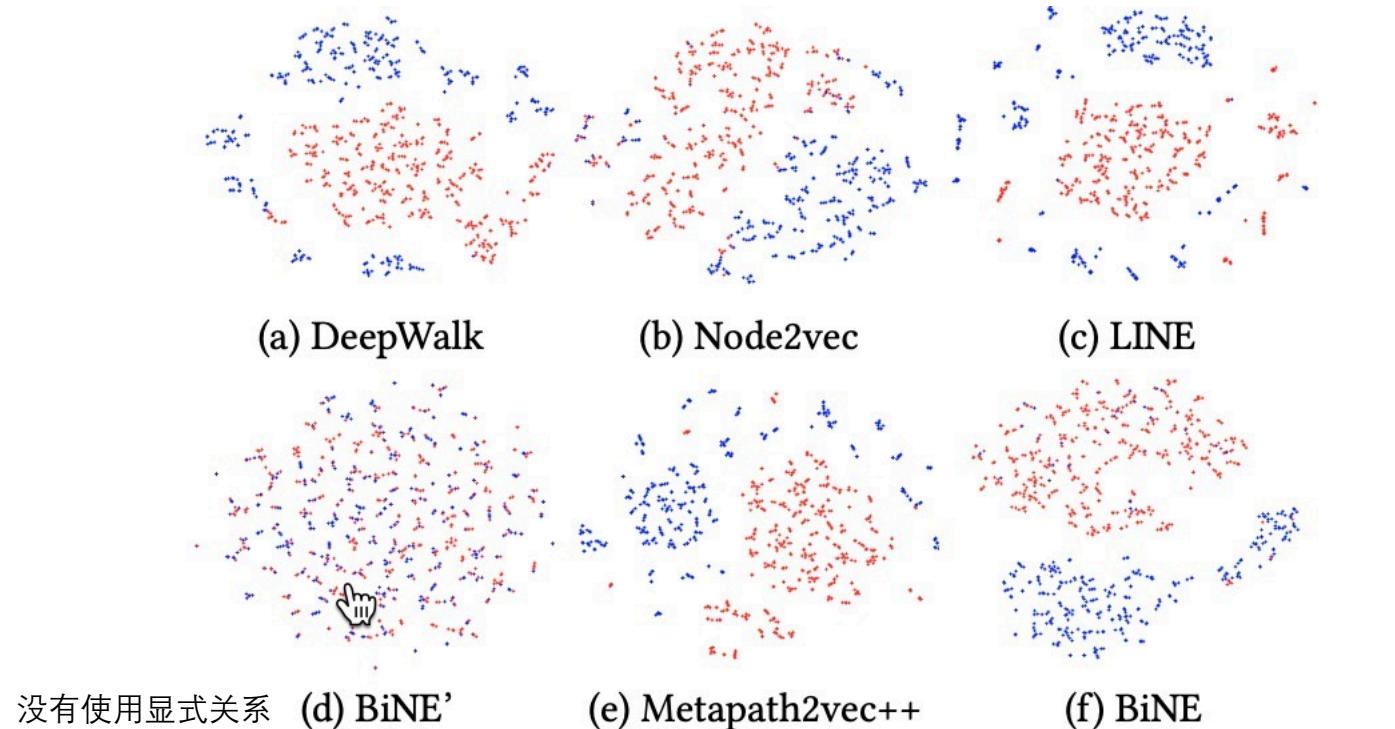
$$\text{maximize } L = \alpha \log O_2 + \beta \log O_3 - \gamma O_1. \quad (9)$$

两个矩阵隐式连接 显式连接

the best value of  $\gamma$  is larger than that of  $\alpha$  and  $\beta$ .

这表明在二部网络嵌入中，显式关系比隐式关系更重要。

## 分类结果比较



**Figure 6: Visualization of authors in DBLP. Color of a vertex indicates the research fields of the authors (red: “computer science theory”, blue: “artificial intelligence”). BiNE’ is the version of BiNE – without implicit relations.**

## Code

```
class GraphUtils(object):
    [ ('u3', 'i6', 1.0), ('u3', 'i7', 1.0), ('u3', 'i8', 1.0), ('u3', 'i11', 1.0),
      self.edge_list = edge_list_u_v 点边信息

    self.edge_dict_u = {} 点边连接信息dictionary
    self.edge_dict_v = {}

    <class 'dict'>: {'u3': {'i6': 1.0, 'i7': 1.0, 'i8': 1.0, 'i11': 1.0, 'i13': 1.0, 'i15': 1.0, 'i16': 1.0, 'i19': 1.0, 'i20': 1.0, 'i22': 1.0, 'i24': 1.0, 'i26': 1.0,
```

---

```
self.G_u, self.walks_u 节点和随机游走序列
self.G_v, self.walks_v
```

```
context_dict_u, neg_dict_u, context_dict_v, neg_dict_v, node_u, node_v = get_context_and_negative_samples(gul, args)
```

节点上下文和负采样节点

biased and self-adaptive random walk generator

```

1 gul.calculate_centrality(args.mode) # HITS计算节点的中心性

2 A = bi.biadjacency_matrix(self.G, self.node_u, self.node_v, dtype=np.float, weight='weight', format='csr')

AT = A.transpose()
self.save_homogenous_graph_to_file(A.dot(AT), self.fw_u, index_row, index_row)
self.save_homogenous_graph_to_file(AT.dot(A), self.fw_v, index_item, index_item)

```

```
4 num_paths = max(int(math.ceil(maxT * hits_dict[node])), minT)
```

- 1 Calculate vertices' centrality:  $\mathbf{H} = CentralityMeasure(\mathbf{W})$ ;
- 2 Calculate  $W^R$  w.r.t. Equation (4);
- 3 **foreach** vertex  $v_i \in R$  **do**
- 4      $l = \max(\mathbf{H}(v_i) \times maxT, minT)$ ;
- 5     **for**  $i = 0$  **to**  $l$  **do**
- 6          $D_{v_i} = BiasedRandomWalk(W^R, v_i, p)$ ;
- 7         Add  $D_{v_i}$  into  $D^R$ ;
- 8 **return**  $D^R$ ;

$$w_{ij}^U = \sum_{k \in V} w_{ik} w_{jk}; \quad w_{ij}^V = \sum_{k \in U} w_{ki} w_{kj}. \quad (4)$$

```
def get_context_and_negative_samples(gul, args):  gul: <graph_utils.GraphUtils object at 0x1339579b0>  args: Namespace(al  
"""  
    get context and negative samples offline  
    :param gul:  
    :param args:  
    :return: context_dict_u, neg_dict_u, context_dict_v, neg_dict_v,gul.node_u,gul.node_v  
"""  
  
    if args.large == 0:  
        neg_dict_u, neg_dict_v = gul.get_negs(args.ns) # 负采样方法 neg_dict_u: <class 'dict'>: {'u3': ['u5624', 'u11623'  
        print("negative samples is ok....")  
        context_dict_u, neg_dict_u = gul.get_context_and_negatives(gul.G_u, gul.walks_u, args.ws, args.ns, neg_dict_u)  
        context_dict_v, neg_dict_v = gul.get_context_and_negatives(gul.G_v, gul.walks_v, args.ws, args.ns, neg_dict_v)  
    else:  
        neg_dict_u, neg_dict_v = gul.get_negs(args.ns)  
        # print len(gul.walks_u),len(gul.walks_u)  
        print("negative samples is ok....")  
        context_dict_u, neg_dict_u = gul.get_context_and_negatives(gul.node_u, gul.walks_u, args.ws, args.ns, neg_dict_u)  
        context_dict_v, neg_dict_v = gul.get_context_and_negatives(gul.node_v, gul.walks_v, args.ws, args.ns, neg_dict_v)  
  
    return context_dict_u, neg_dict_u, context_dict_v, neg_dict_v,gul.node_u,gul.node_v
```

```
node_list_u[i]['embedding_vectors'] = preprocessing.normalize(vectors, norm='l2')
node_list_u[i]['context_vectors'] = preprocessing.normalize(help_vectors, norm='l2')
```

```
V = np.array(node_list[center]['embedding_vectors'])
```

```
Theta = np.array(node_list[u]['context_vectors'])
```

```
X = float(V.dot(Theta.T)) # v*c X: 0.7455320782821748
```

```
sigmod = 1.0 / (1 + (math.exp(-X * 1.0)))
```

```
update += pa * lam * (I_z[u] - sigmod) * Theta
```

```
node_list[u]['context_vectors'] += pa * lam * (I_z[u] - sigmod) * V
```

$$\alpha[I(z, u_i) - \sigma(\vec{u}_i^T \vec{\theta}_z)] \cdot \vec{\theta}_z$$

$$\alpha \log O_2 + \beta \log O_3$$

where  $I(z, u_i)$  is an indicator function that determines whether vertex  $z$  is in the context of  $u_i$  or not; similar meaning applies

$$\vec{\theta}_z = \vec{\theta}_z + \lambda \{ \alpha [I(z, u_i) - \sigma(\vec{u}_i^T \vec{\theta}_z)] \cdot \vec{u}_i \}, \quad (14)$$

$$\vec{\theta}_z = \vec{\theta}_z + \lambda \{ \beta [I(z, v_j) - \sigma(\vec{v}_j^T \vec{\theta}_z)] \cdot \vec{v}_j \}. \quad (15)$$

$$\vec{u}_i = \vec{u}_i + \lambda \{ \sum_{z \in \{u_c\} \cup N_S^{ns}(u_i)} \alpha [I(z, u_i) - \sigma(\vec{u}_i^T \vec{\theta}_z)] \cdot \vec{\theta}_z \} \quad (12)$$

$$\vec{v}_j = \vec{v}_j + \lambda \{ \sum_{z \in \{v_c\} \cup N_S^{ns}(v_j)} \beta [I(z, v_j) - \sigma(\vec{v}_j^T \vec{\theta}_z)] \cdot \vec{\theta}_z \} \quad (13)$$

```
loss += pa * (I_z[u] * math.log(sigmod) + (1 - I_z[u]) * math.log(1 - sigmod))
```

```
tmp_z, tmp_loss = skip_gram(u, z, neg_u, node_list_u, lam, alpha)
```

```
node_list_u[z]['embedding_vectors'] += tmp_z # 更新节点embedding
```

$$L(\hat{y}, y) = -y \log(\hat{y}) - (1 - y) \log(1 - \hat{y})$$

$$\vec{u}_i = \vec{u}_i + \lambda \{ \sum_{z \in \{u_c\} \cup N_S^{ns}(u_i)} \alpha [I(z, u_i) - \sigma(\vec{u}_i^T \vec{\theta}_z)] \cdot \vec{\theta}_z \} \quad (12)$$

```

U = np.array(node_list_u[u]['embedding_vectors'])  U: [[1.08137323e-01 1.31560005e-01 1.
V = np.array(node_list_v[v]['embedding_vectors'])  V: [[0.10291547 0.05468078 0.13052218
X = float(U.dot(V.T))  X: 0.7865115986431102

```

```

sigmod = 1.0 / (1 + (math.exp(-X * 1.0)))  sigmod: 0.6870818128197933

```

```

update_u += gamma * lam * ((e_ij * (1 - sigmod)) * 1.0 / math.log(math.e, math.e)) * V
update_v += gamma * lam * ((e_ij * (1 - sigmod)) * 1.0 / math.log(math.e, math.e)) * U

```

$$L_1 = -\gamma O_1$$

$$\vec{u}_i = \vec{u}_i + \lambda \{\gamma w_{ij} [1 - \sigma(\vec{u}_i^T \vec{v}_j)] \cdot \vec{v}_j\}, \quad (10)$$

$$\vec{v}_j = \vec{v}_j + \lambda \{\gamma w_{ij} [1 - \sigma(\vec{u}_i^T \vec{v}_j)] \cdot \vec{u}_i\}, \quad (11)$$

```

try:
    loss += gamma * e_ij * math.log(sigmod)

```

which can be defined as:

$$\begin{aligned}
& \text{minimize } O_1 = KL(P || \hat{P}) = \sum_{e_{ij} \in E} P(i, j) \log \left( \frac{P(i, j)}{\hat{P}(i, j)} \right) \\
& \propto - \sum_{e_{ij} \in E} w_{ij} \log \hat{P}(i, j).
\end{aligned} \quad (3)$$



SGCN

之前介绍的GNN模型主要集中在无符号的网络（或仅由正链接组成的图）上，符号图带来的挑战，主要集中在于否定链接，与正链接相比，它不仅具有不同的语义，而且其原理本质上是不同的，并且它们与正链接形成了复杂的关系

我们面临的主要挑战是：

- 1) 如何处理负链接，他们的属性和正链接是完全不同的
- 2) 如何将正负链接组合一个模型，以更好的学习节点的表征

为了解决以上问题，作者提出了SGCN，主要贡献：

- 提出一个基于平衡理论构建的签名图卷积网络（SGCN），以在聚合过程中正确地整合负面链接；
- 根据签署的网络社会理论为我们的SGCN构建目标函数，以轻松学习网络中每个节点的有效低维表示；
- 在四个真实世界的签名网络上进行实验，以全面证明我们提议的SGCN框架的有效性。

定义

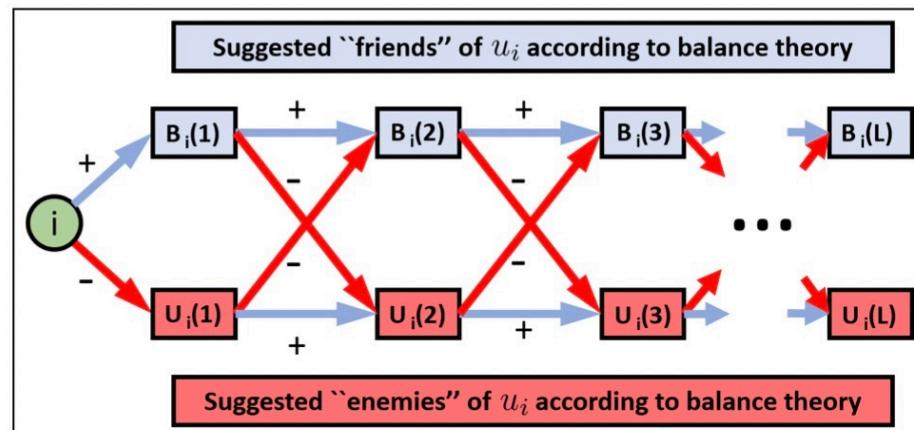
$$\mathcal{G} = (\mathcal{U}, \mathcal{E}^+, \mathcal{E}^-) \quad \mathcal{E}^+ \cap \mathcal{E}^- = \emptyset$$

$$F : \mathbf{A} \rightarrow \mathbf{Z} \quad \text{Low-dimensional representation of signed network } \mathcal{G}$$

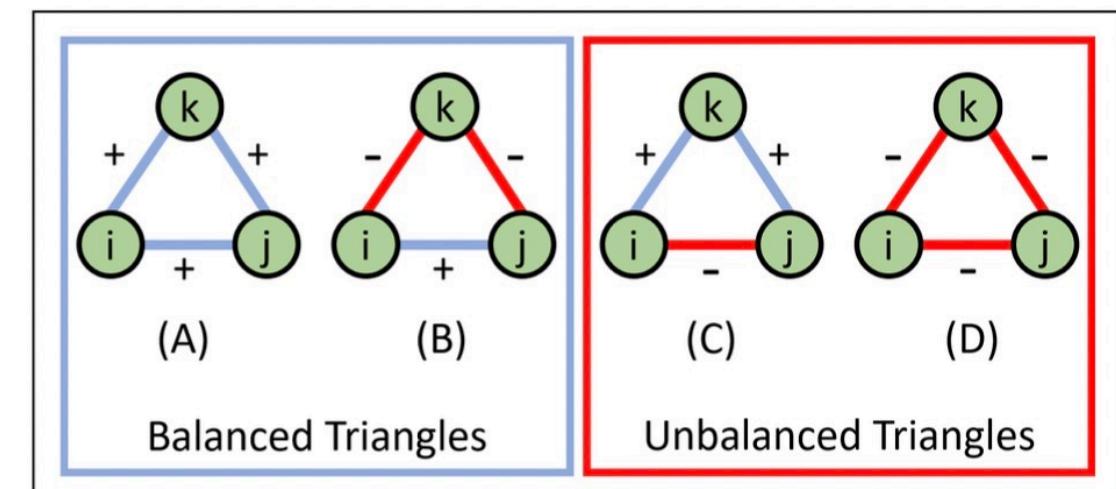
### B. Aggregation paths with positive and negative links

平衡理论就是“我朋友的朋友是我的朋友”，“我朋友的敌人是我的敌人”，“我敌人的敌人是朋友”

平衡路径表示为包含偶数个负连接的路径；不平衡路径也表示为包含奇数个负连接的路径



平衡三角形



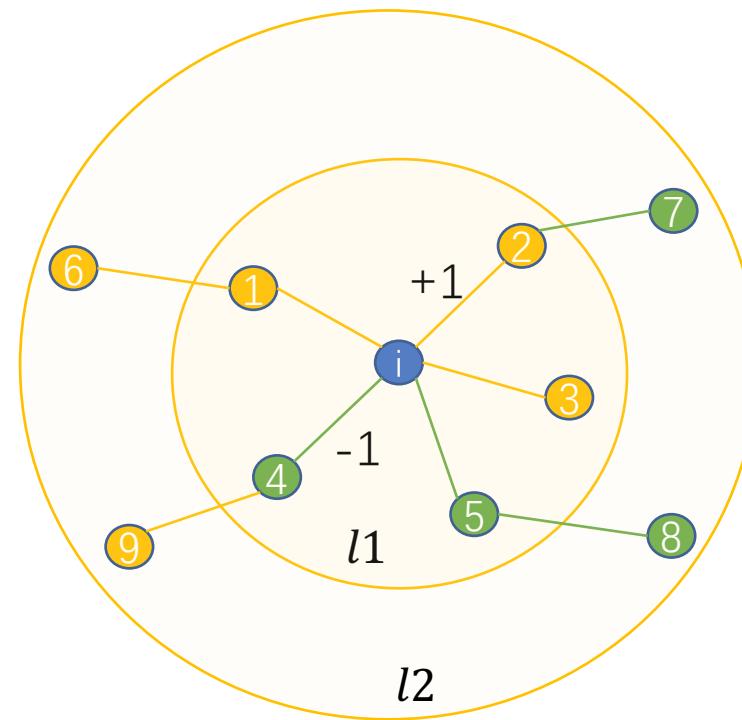
When  $l = 1$

$i$ 节点的positive邻居

$$B_i(1) = \{u_j \mid u_j \in \mathcal{N}_i^+\} \quad \{1,2,3\}$$

$$U_i(1) = \{u_j \mid u_j \in \mathcal{N}_i^-\} \quad \{4,5\}$$

$i$ 节点的negative邻居



平衡理论

For  $l > 1$

$$1,2,3 \rightarrow 6$$

$$\begin{aligned} B_i(l+1) = & \{u_j \mid u_k \in B_i(l) \text{ and } u_j \in \mathcal{N}_k^+\} \\ \cup & \{u_j \mid u_k \in U_i(l) \text{ and } u_j \in \mathcal{N}_k^-\} \end{aligned}$$

$B_i(l)$ 中节点, 下层positive节点

$U_i(l)$ 中节点, 下层negative节点

$$4,5 \rightarrow 8$$

$i$ 节点下层的balance邻居

$$\text{negative} + \text{negative} = \text{positive}$$

$$4,5 \rightarrow 9$$

$$\begin{aligned} U_i(l+1) = & \{u_j \mid u_k \in U_i(l) \text{ and } u_j \in \mathcal{N}_k^+\} \\ \cup & \{u_j \mid u_k \in B_i(l) \text{ and } u_j \in \mathcal{N}_k^-\} \end{aligned}$$

$$1,2,3 \rightarrow 7 \quad (2)$$

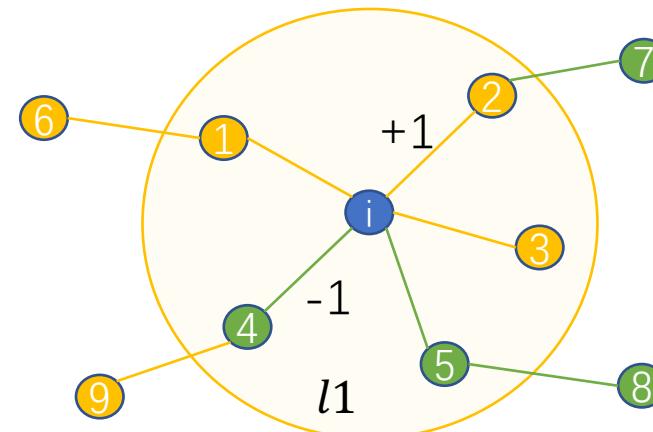
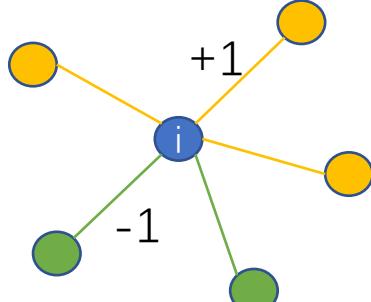
$i$ 节点下层的unbalance邻居

### C. Signed Graph Convolutional Network

set (i.e., suggested “enemies”). Similar to the unsigned GCN, we use  $\mathbf{h}_i^{(0)} \in \mathbb{R}^{d^{in}}$  to represent the initial  $d^{in}$  node features for user  $u_i$ . Thus, for the first aggregation layer (i.e, when

$$\mathbf{h}_i^{B(1)} = \sigma \left( \mathbf{W}^{B(1)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{(0)}}{|\mathcal{N}_i^+|}, \mathbf{h}_i^{(0)} \right] \right) \quad (3)$$

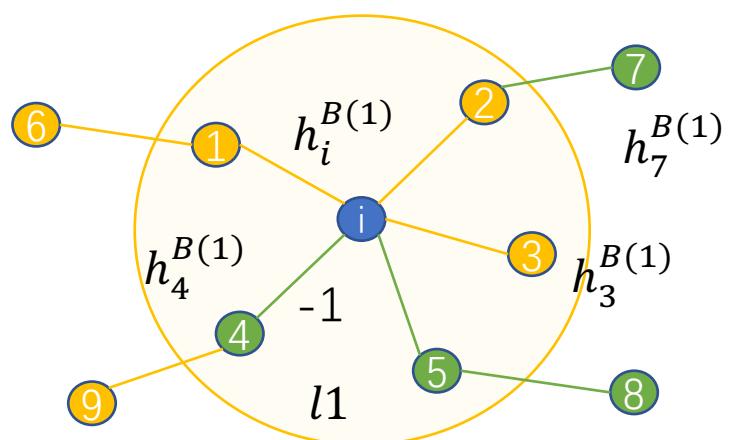
$$\mathbf{h}_i^{U(1)} = \sigma \left( \mathbf{W}^{U(1)} \left[ \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{(0)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{(0)} \right] \right) \quad (4)$$



negative links as seen in Figure 2. The aggregations for  $l > 1$  are defined as follows:

$$\mathbf{h}_i^{B(l)} = \sigma \left( \mathbf{W}^{B(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{B(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{U(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{B(l-1)} \right] \right) \quad (5)$$

$$\mathbf{h}_i^{U(l)} = \sigma \left( \mathbf{W}^{U(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{U(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{B(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{U(l-1)} \right] \right) \quad (6)$$



negative links as seen in Figure 2. The aggregations for  $l > 1$  are defined as follows:

$$\mathbf{h}_i^{B(l)} = \sigma \left( \mathbf{W}^{B(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{B(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{U(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{B(l-1)} \right] \right) \quad (5)$$

$$\frac{h_1^{B(1)} + h_2^{B(1)} + h_3^{B(1)}}{3}$$

$$\frac{h_4^{U(1)} + h_5^{U(1)}}{2}$$

$$h_i^{B(1)}$$

$$\mathbf{h}_i^{U(l)} = \sigma \left( \mathbf{W}^{U(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{U(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{B(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{U(l-1)} \right] \right) \quad (6)$$

---

**Algorithm 2:** Signed GCN Embedding Generation.

---

**Input:**  $\mathcal{G} = (\mathcal{U}, \mathcal{E}^+, \mathcal{E}^-)$ ; an initial seed node representation  $\{\mathbf{x}_i, \forall u_i \in \mathcal{U}\}$ ; number of aggregation layers  $L$ ; weight matrices  $\mathbf{W}^{B(l)}$  and  $\mathbf{W}^{U(l)}$ ,  $\forall l \in \{1, \dots, L\}$ ; non-linear function  $\sigma$

**Output:** Low-dimensional representations  $\mathbf{z}_i, \forall u_i \in \mathcal{U}$

```
1  $\mathbf{h}_i^{(0)} \leftarrow \mathbf{x}_i, \forall u_i \in \mathcal{U}$ 
2 for  $u_i \in \mathcal{U}$  do
3    $\mathbf{h}_i^{B(1)} \leftarrow \sigma \left( \mathbf{W}^{B(1)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{(0)}}{|\mathcal{N}_i^+|}, \mathbf{h}_i^{(0)} \right] \right)$  we set  $\mathbf{h}_i^{(0)}$  equal to  $\mathbf{x}_i$ 
4    $\mathbf{h}_i^{U(1)} \leftarrow \sigma \left( \mathbf{W}^{U(1)} \left[ \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{(0)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{(0)} \right] \right)$ 
5 end
6 if  $L > 1$  then
7   for  $l = 2 \dots L$  do
8     for  $u_i \in \mathcal{U}$  do
9        $\mathbf{h}_i^{B(l)} =$ 
10       $\sigma \left( \mathbf{W}^{B(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{B(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{U(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{B(l-1)} \right] \right)$ 
11       $\mathbf{h}_i^{U(l)} =$ 
12       $\sigma \left( \mathbf{W}^{U(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{U(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{B(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{U(l-1)} \right] \right)$ 
13    end
14  end
15 end
16  $\mathbf{z}_i \leftarrow [\mathbf{h}_i^{B(L)}, \mathbf{h}_i^{U(L)}], \forall u_i \in \mathcal{U}$  最后将两个embedding拼接在一起
```

---

Loss

$\omega_s$  is used for the weight associated with the class  $s$

译文：第二个词的目标是让在嵌入空间中有积极链接的用户比没有链接对的用户更接近，而没有链接对的用户应该比他们之间有消极链接的用户更接近。总体目标如下所示

$$\mathcal{L}(\theta^W, \theta^{MLG}) =$$

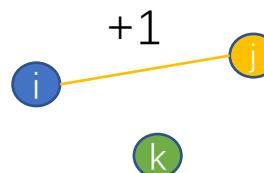
$$-\frac{1}{\mathcal{M}} \sum_{(u_i, u_j, s) \in \mathcal{M}} \omega_s \log \frac{\exp ([\mathbf{z}_i, \mathbf{z}_j] \theta_s^{MLG})}{\sum_{q \in \{+, -, ?\}} \exp ([\mathbf{z}_i, \mathbf{z}_j] \theta_q^{MLG})}$$

$$+ \lambda \left[ \frac{1}{|\mathcal{M}_{(+,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(+,?)}}} \max \left( 0, (\|\mathbf{z}_i - \mathbf{z}_j\|_2^2 - \|\mathbf{z}_i - \mathbf{z}_k\|_2^2) \right) \right]$$

$$+ \frac{1}{|\mathcal{M}_{(-,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(-,?)}}} \max \left( 0, (\|\mathbf{z}_i - \mathbf{z}_k\|_2^2 - \|\mathbf{z}_i - \mathbf{z}_j\|_2^2) \right)$$

$$+ Reg(\theta^W, \theta^{MLG})$$

预测边的类别



$\theta^{MLG}$  denotes the parameters of the MLG classifier

属于这三个类别值的和

$k$ 表示和*i*不相连的节点*u<sub>k</sub>*

正样本，连接节点的loss比未连接的要小

负样本，未连接节点的loss比负连接的要小

(7)

正则项

negatively linked users, respectively, where for every linked pair  $(u_i, u_j)$  we further sample another user  $u_k$  randomly (and different in each epoch) that has no link to  $u_i$ . The term

在本节中，我们将通过实验评估提议的符号图卷积网络（SGCN）在学习节点表示中的有效性。我们试图回答以下问题：（1）SGCN是否能够学习有意义的低维表示？（2）在聚合过程中引入平衡理论以及更长的路径信息是否可以提高学习节点嵌入的性能？

## LINK SIGN PREDICTION RESULTS WITH AUC.

Embedding Method	Bitcoin-Alpha	Bitcoin-OTC	Slashdot	Epinions
SSE	0.764	0.803	0.769	0.822
SiNE	0.778	0.814	0.792	0.849
SIDE	0.630	0.618	0.547	0.571
SGCN-1	0.780	0.818	0.784	0.663
SGCN-1+	0.785	0.817	<b>0.804</b>	0.722
SGCN-2	<b>0.796</b>	<b>0.823</b>	<b>0.804</b>	<b>0.864</b>

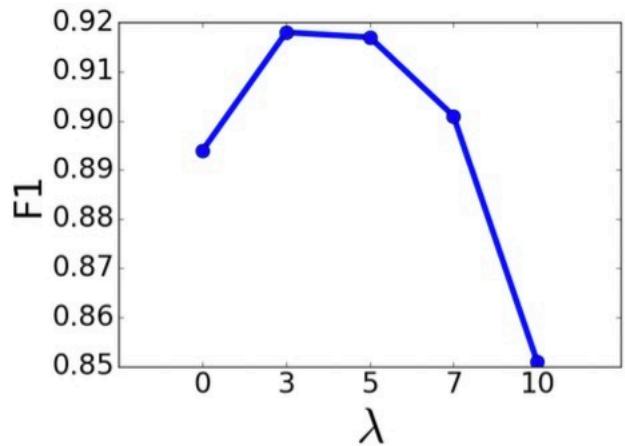
链接预测：预测链接是正或者负

- Signed Spectral Embedding (SSE) [14]: A spectral clustering algorithm based on the proposed signed version of the Laplacian matrix. We utilize the top- $d^{out}$  eigenvectors corresponding to the smallest eigenvalues as the embedding vectors for each node.
- SiNE [23]: This method is a deep learning framework that utilized extended structural balance theory.
- SIDE [24]: A random walk based method, utilizing balance theory, is used to obtain indirect connections for a likelihood formulation.
- SGCN-1: 将正负边分开建模，传播一次
- SGCN-1+: 未使用平衡理论，只是将正负边传播两次

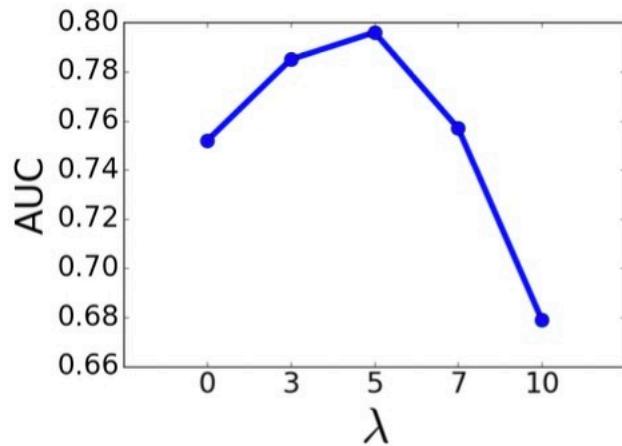
一种基于拉普拉斯矩阵谱聚类算法

一种利用扩展结构平衡理论的深度学习方法

基于随机游动的方法和平衡理论



(a) SGCN-2 (F1)



(b) SGCN-2 (AUC)

$$\begin{aligned}
 \mathcal{L}(\theta^W, \theta^{MLG}) = & \\
 & -\frac{1}{|\mathcal{M}|} \sum_{(u_i, u_j, s) \in \mathcal{M}} \omega_s \log \frac{\exp([\mathbf{z}_i, \mathbf{z}_j] \theta_s^{MLG})}{\sum_{q \in \{+, -, ?\}} \exp([\mathbf{z}_i, \mathbf{z}_j] \theta_q^{MLG})} \\
 & + \lambda \left[ \frac{1}{|\mathcal{M}_{(+,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(+,?)}}} \max \left( 0, (||\mathbf{z}_i - \mathbf{z}_j||_2^2 - ||\mathbf{z}_i - \mathbf{z}_k||_2^2) \right) \right. \\
 & + \frac{1}{|\mathcal{M}_{(-,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(-,?)}}} \max \left( 0, (||\mathbf{z}_i - \mathbf{z}_k||_2^2 - ||\mathbf{z}_i - \mathbf{z}_j||_2^2) \right) \left. \right] \\
 & + Reg(\theta^W, \theta^{MLG})
 \end{aligned} \tag{7}$$

Input :

3580,3643,1.0

3580,3659,1.0

3580,3662,1.0

3580,3726,1.0

3580,3788,-1.0

3580,3981,1.0

3580,5411,1.0

3584,3671,1.0

3584,3693,1.0

3584,3705,1.0

3584,3918,1.0

3585,3593,1.0

3589,3596,-1.0

3590,3645,1.0

3590,3655,1.0

3590,3682,1.0

3590,3727,1.0

---

**Positive edges**

**Negative edges**

Code

```
class SignedGCNTrainer(object):

    self.ecount = len(self.positive_edges + self.negative_edges) # 训练样本数量

    # SVD分解，邻接矩阵，为节点特征；
    self.X = setup_features(self.args,

        # positive样本的点边关系
        self.positive_edges = torch.from_numpy(np.array(self.positive_edges,
            dtype=np.int64).T).type(torch.long).to(self.device)

        # negative样本的点边关系
        self.negative_edges = torch.from_numpy(np.array(self.negative_edges,
            dtype=np.int64).T).type(torch.long).to(self.device)

        # label: [0],[1] 两个部分; [2]是两倍的点边数量-表示不连接的边; !!
        np.array([0]*len(self.positive_edges) + [1]*len(self.negative_edges[0]) + [2]*(self.ecount*2))

    self.X = torch.from_numpy(self.X).float().to(self.device) # [5881,64]的input features
```

构建SGCN

```
self.model = SignedGraphConvolutionalNetwork(self.device, self.args, self.X).to(self.device)
```

构建第一层SGCN

```
self.positive_base_aggregator = SignedSAGEConvolutionBase(self.X.shape[1]*2, # 64*2 -> 32  
self.neurons[0]).to(self.device)
```

构建后几层SGCN

```
for i in range(1, self.layers): i:  
    self.positive_aggregators.append(SignedSAGEConvolutionDeep(3*self.neurons[i-1],  
self.neurons[i]).to(self.device))
```

$$\mathbf{h}_i^{B(l)} = \sigma \left( \mathbf{W}^{B(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{B(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{U(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{B(l-1)} \right] \right) \quad (5)$$

$$\mathbf{h}_i^{U(l)} = \sigma \left( \mathbf{W}^{U(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{U(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{B(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{U(l-1)} \right] \right) \quad (6)$$

```
self.regression_weights = Parameter(torch.Tensor(4*self.neurons[-1], 3)) # 4*32, 128
```

$$\mathbf{z}_i \leftarrow [\mathbf{h}_i^{B(L)}, \mathbf{h}_i^{U(L)}], \forall u_i \in \mathcal{U}$$
$$- \frac{1}{\mathcal{M}} \sum_{(u_i, u_j, s) \in \mathcal{M}} \omega_s \log \frac{\exp([\mathbf{z}_i, \mathbf{z}_j] \theta_s^{MLG})}{\sum_{q \in \{+, -, ?\}} \exp([\mathbf{z}_i, \mathbf{z}_j] \theta_q^{MLG})}$$

```
self.model = SignedGraphConvolutionalNetwork(self.device, self.args, self.X).to(self.device)

>>> self.model
SignedGraphConvolutionalNetwork(
    (positive_base_aggregator): SignedSAGEConvolutionBase(128, 32) 第一层数据 : input=64*2, output=32
    (negative_base_aggregator): SignedSAGEConvolutionBase(128, 32)
    (positive_aggregators): ListModule(
        (0): SignedSAGEConvolutionDeep(96, 32)
    )
    (negative_aggregators): ListModule( 第二层数据 : input=32*3, output=32
        (0): SignedSAGEConvolutionDeep(96, 32)
    )
)
```

```
self.h_pos.append(torch.tanh(self.positive_base_aggregator(self.X, positive_edges)))
```

节点特征

Positive样本

```
>>> positive_edges
tensor([[3623, 230, 1459, ..., 1853, 3649, 2039],
       [4092, 772, 1981, ..., 2077, 4230, 2048]])
```

forward

```
class SignedGraphConvolutionalNetwork(torch.nn.Module):  
    def forward(self, positive_edges, negative_edges, target):  
        self.h_pos.append(torch.tanh(self.positive_base_aggregator(self.X, positive_edges)))  
        self.h_neg.append(torch.tanh(self.negative_base_aggregator(self.X, negative_edges)))
```

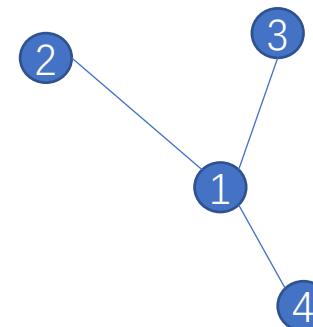
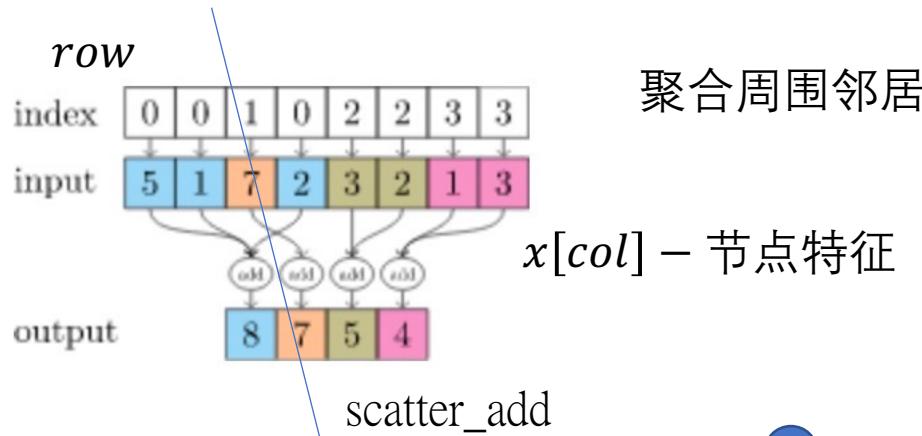
$$\mathbf{h}_i^{B(1)} = \sigma \left( \mathbf{W}^{B(1)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{(0)}}{|\mathcal{N}_i^+|}, \mathbf{h}_i^{(0)} \right] \right) \quad (3)$$

```
class SignedSAGEConvolutionBase(SignedSAGEConvolution):
```

```
    row, col = edge_index
```

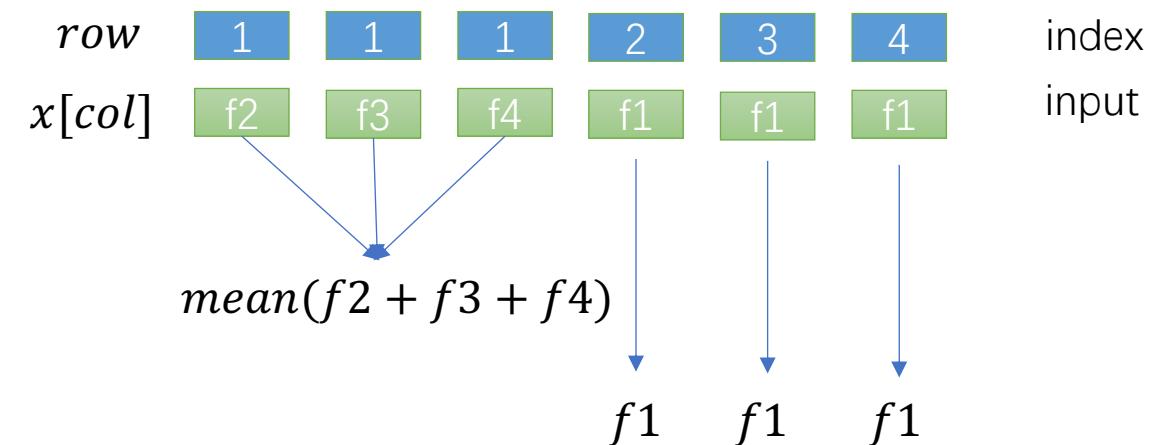
```
    if self.norm:
```

```
        out = scatter_mean(x[col], row, dim=0, dim_size=x.size(0)) # scatter_mean函数表示, row索引位置相加, 返回结果;
```



$$\mathbf{h}_i^{B(1)} = \sigma \left( \mathbf{W}^{B(1)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{(0)}}{|\mathcal{N}_i^+|}, \mathbf{h}_i^{(0)} \right] \right) \quad (3)$$

$row = [1, 1, 1, 2, 3, 4]$   
 $col = [2, 3, 4, 1, 1, 1]$



```
out = torch.cat((out, x), 1) # 将聚合特征和原始特征拼接  
out = torch.matmul(out, self.weight)
```

[128, 32]

$$\mathbf{h}_i^{B(1)} = \sigma \left( \mathbf{W}^{B(1)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{(0)}}{|\mathcal{N}_i^+|}, \mathbf{h}_i^{(0)} \right] \right) \quad (3)$$

第一层节点Balance的特征

```
self.h_pos.append(torch.tanh(self.positive_base_aggregator(self.X, positive_edges)))
```

## 第二层及后面的网络

```

for i in range(1, self.layers):
    self.h_pos.append(torch.tanh(self.positive_aggregators[i-1](self.h_pos[i-1], self.h_neg[i-1], positive_edges,
    self.h_neg.append(torch.tanh(self.negative_aggregators[i-1](self.h_neg[i-1], self.h_pos[i-1], positive_edges,
    self.h_pos.append(torch.tanh(self.positive_aggregators[i-1](self.h_pos[i-1], self.h_neg[i-1], positive_edges, negative_edges)))
    self.h_neg.append(torch.tanh(self.negative_aggregators[i-1](self.h_neg[i-1], self.h_pos[i-1], positive_edges, negative_edges)))

>>> self.positive_aggregators
ListModule(
    (0): SignedSAGEConvolutionDeep(96, 32)
)

```

```

class SignedSAGEConvolutionDeep(SignedSAGEConvolution):

```

```

if self.norm:
    out_1 = scatter_mean(x_1[col_pos], row_pos, dim=0, dim_size=x_1.size(0))
    out_2 = scatter_mean(x_2[col_neg], row_neg, dim=0, dim_size=x_2.size(0))

out = torch.cat((out_1, out_2, x_1), 1)
out = torch.matmul(out, self.weight)

```

[96, 32]

pos特征 neg特征 边

[5881, 32]

$$\mathbf{h}_i^{B(l)} = \sigma \left( \mathbf{W}^{B(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{B(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{U(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{B(l-1)} \right] \right) \quad (5)$$

`self.z = torch.cat((self.h_pos[-1], self.h_neg[-1]), 1)`  
[5881, 32] [5881, 32]

$\mathbf{z}_i \leftarrow [\mathbf{h}_i^{B(L)}, \mathbf{h}_i^{U(L)}], \forall u_i \in \mathcal{U}$

$$\mathbf{h}_i^{U(l)} = \sigma \left( \mathbf{W}^{U(l)} \left[ \sum_{j \in \mathcal{N}_i^+} \frac{\mathbf{h}_j^{U(l-1)}}{|\mathcal{N}_i^+|}, \sum_{k \in \mathcal{N}_i^-} \frac{\mathbf{h}_k^{B(l-1)}}{|\mathcal{N}_i^-|}, \mathbf{h}_i^{U(l-1)} \right] \right) \quad (6)$$

计算loss

```
loss = self.calculate_loss_function(self.z, positive_edges, negative_edges, target)

def calculate_loss_function(self, z, positive_edges, negative_edges, target):
    loss_term_1 = self.calculate_positive_embedding_loss(z, positive_edges)
    loss_term_2 = self.calculate_negative_embedding_loss(z, negative_edges)
    regression_loss, self.predictions = self.calculate_regression_loss(z, target)
    loss_term = regression_loss+self.args.lamb*(loss_term_1+loss_term_2)
    return loss_term
```

$$\begin{aligned} \mathcal{L}(\theta^W, \theta^{MLG}) &= \\ &- \frac{1}{\mathcal{M}} \sum_{(u_i, u_j, s) \in \mathcal{M}} \omega_s \log \frac{\exp ([\mathbf{z}_i, \mathbf{z}_j] \theta_s^{MLG})}{\sum_{q \in \{+, -, ?\}} \exp ([\mathbf{z}_i, \mathbf{z}_j] \theta_q^{MLG})} \\ &+ \lambda \left[ \frac{1}{|\mathcal{M}_{(+,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(+,?)}}} \max \left( 0, (||\mathbf{z}_i - \mathbf{z}_j||_2^2 - ||\mathbf{z}_i - \mathbf{z}_k||_2^2) \right) \right. \\ &\quad \left. + \frac{1}{|\mathcal{M}_{(-,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(-,?)}}} \max \left( 0, (||\mathbf{z}_i - \mathbf{z}_k||_2^2 - ||\mathbf{z}_i - \mathbf{z}_j||_2^2) \right) \right] \\ &+ Reg(\theta^W, \theta^{MLG}) \end{aligned} \tag{7}$$

```

def calculate_positive_embedding_loss(self, z, positive_edges):
    self.positive_z_i = z[positive_edges[:, 0], :] # row的index
    self.positive_z_j = z[positive_edges[:, 1], :] # col的index
    self.positive_z_k = z[self.positive_surrogates, :] # 随机的节点
    norm_i_j = torch.norm(self.positive_z_i-self.positive_z_j, 2, 1, True).pow(2)
    norm_i_k = torch.norm(self.positive_z_i-self.positive_z_k, 2, 1, True).pow(2)
    term = norm_i_j-norm_i_k  term: tensor([0.0292],\n                                         [0.0000],\n                                         term[term < 0] = 0
    loss_term = term.mean()  loss_term: tensor(0.0336, grad_fn=<MeanBackward0>

```

positive相连接的点距离

```

def calculate_negative_embedding_loss(self, z, negative_edges):

```

$$+ \lambda \left[ \frac{1}{|\mathcal{M}_{(+,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(+,?)}}} \max \left( 0, (||\mathbf{z}_i - \mathbf{z}_j||_2^2 - ||\mathbf{z}_i - \mathbf{z}_k||_2^2) \right) \right]$$

```
regression_loss, self.predictions = self.calculate_regression_loss(z, target)
```

```
pos = torch.cat((self.positive_z_i, self.positive_z_j), 1) positive连接的两个节点
```

```
neg = torch.cat((self.negative_z_i, self.negative_z_j), 1)
```

```
surr_neg_i = torch.cat((self.negative_z_i, self.negative_z_k), 1)
```

negative和随机选择的节点k

```
surr_neg_j = torch.cat((self.negative_z_j, self.negative_z_k), 1)
```

```
surr_pos_i = torch.cat((self.positive_z_i, self.positive_z_k), 1)
```

```
surr_pos_j = torch.cat((self.positive_z_j, self.positive_z_k), 1)
```

```
features = torch.cat((pos, neg, surr_neg_i, surr_neg_j, surr_pos_i, surr_pos_j))
```

---

---

---

0      1

2

```
predictions = torch.mm(features, self.regression_weights)
```

```
predictions_soft = F.log_softmax(predictions, dim=1)
```

```
loss_term = F.nll_loss(predictions_soft, target)
```

$$-\frac{1}{\mathcal{M}} \sum_{(u_i, u_j, s) \in \mathcal{M}} \omega_s \log \frac{\exp ([\mathbf{z}_i, \mathbf{z}_j] \theta_s^{MLG})}{\sum_{q \in \{+, -, ?\}} \exp ([\mathbf{z}_i, \mathbf{z}_j] \theta_q^{MLG})}$$

L(Y,P(Y|X))=-logP(Y|X) 对数损失函数

```
# label: [0],[1] 两个部分; [2]是两倍的点边数量-表示不连接的边; !! np.array([0]*len(self.positive_edges) + [1]*  
self.y = np.array([0]*len(self.positive_edges[0]) + [1]*len(self.negative_edges[0]) + [2]*(self.ecount*2))
```

```
loss_term = regression_loss+self.args.lamb*(loss_term_1+loss_term_2)
```

$$\begin{aligned}
\mathcal{L}(\theta^W, \theta^{MLG}) = & \\
& - \frac{1}{|\mathcal{M}|} \sum_{(u_i, u_j, s) \in \mathcal{M}} \omega_s \log \frac{\exp([\mathbf{z}_i, \mathbf{z}_j] \theta_s^{MLG})}{\sum_{q \in \{+, -, ?\}} \exp([\mathbf{z}_i, \mathbf{z}_j] \theta_q^{MLG})} \\
& + \lambda \left[ \frac{1}{|\mathcal{M}_{(+,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(+,?)}}} \max \left( 0, (\|\mathbf{z}_i - \mathbf{z}_j\|_2^2 - \|\mathbf{z}_i - \mathbf{z}_k\|_2^2) \right) \right. \\
& \left. + \frac{1}{|\mathcal{M}_{(-,?)}|} \sum_{\substack{(u_i, u_j, u_k) \\ \in \mathcal{M}_{(-,?)}}} \max \left( 0, (\|\mathbf{z}_i - \mathbf{z}_k\|_2^2 - \|\mathbf{z}_i - \mathbf{z}_j\|_2^2) \right) \right] \\
& + Reg(\theta^W, \theta^{MLG})
\end{aligned} \tag{7}$$

SiGAT

SiGAT :

- 1) 提出一种新颖方法去找到节点的邻居
- 2) 使用GAT的方法去解决聚合邻居的问题

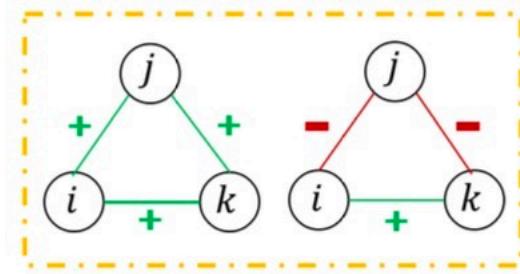


1. 采用社会学中两个重要的理论：
  - 1) 平衡理论
  - 2) 地位理论
2. 在社会学理论基础下，提出Motifs方法来提取节点的邻居。

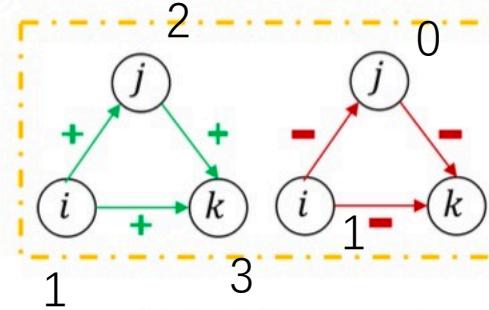
## 平衡理论和地位理论

$$j > i; \ k > i; \ k > j$$

状态顺序



(a) Illustration of structural balance theory



(b) Illustration of status theory

Fig. 1. The balance theory and status theory

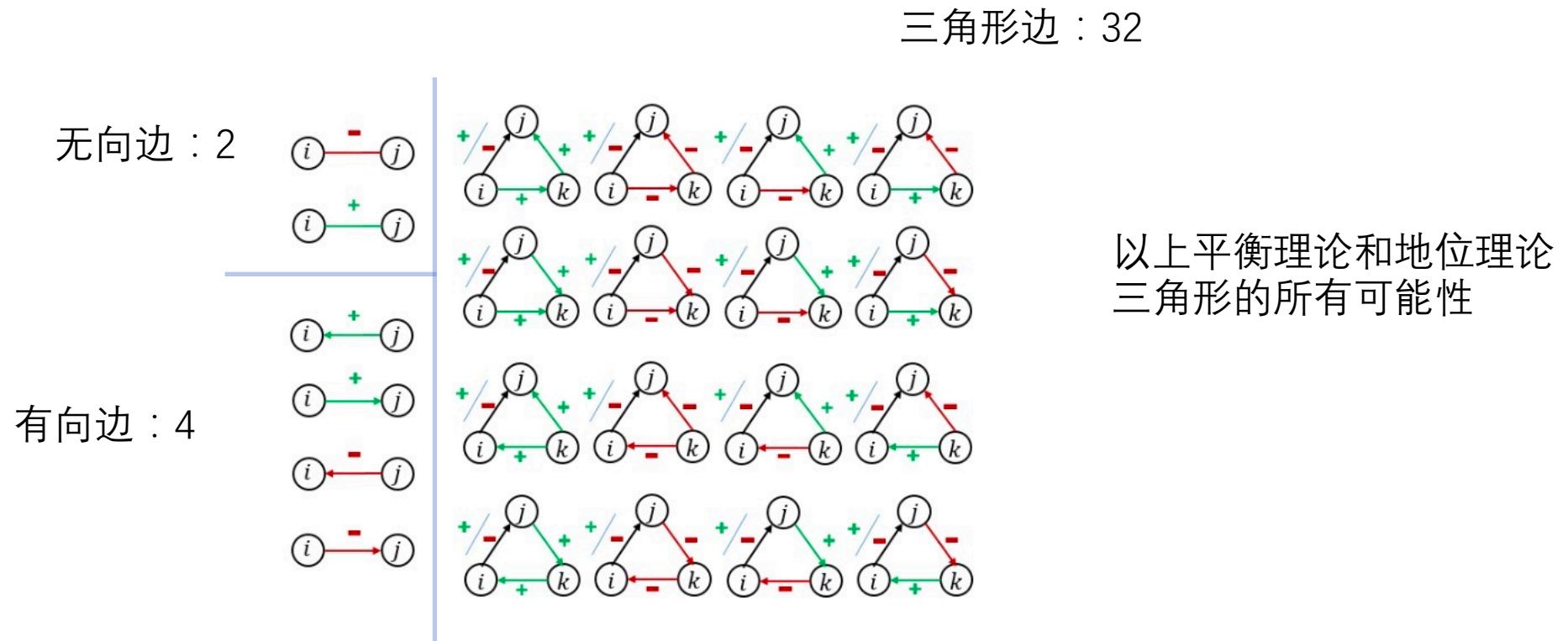
地位理论：

它假定以正号 “+” 或负号 “-” 标记的有向关系表示目标节点的状态高于或低于源节点。

例如，从A到B的积极联系，不仅意味着“B是我的朋友”，还意味着“我认为B的地位比我高”。

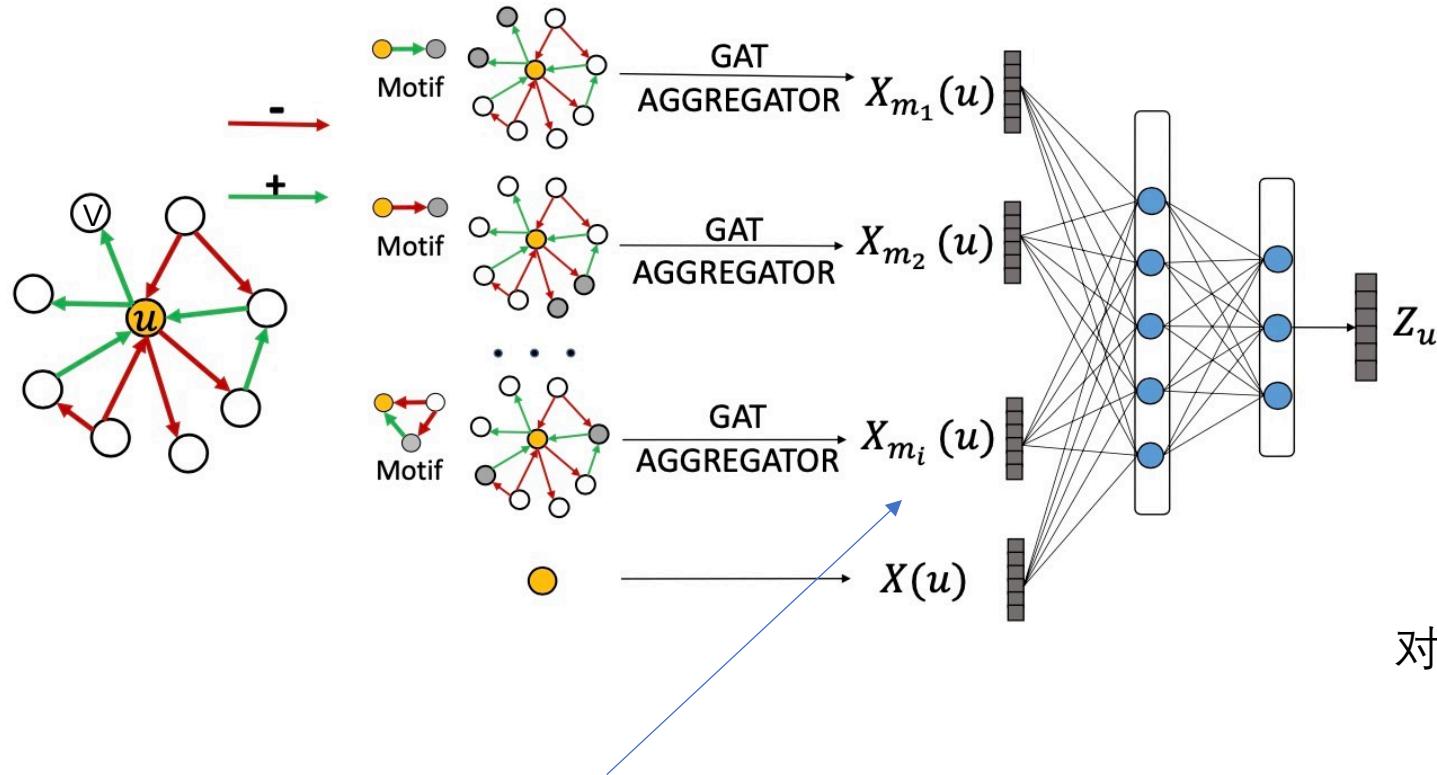
例如，从A到B的正向链接意味着不仅B是A的朋友，而且B的地位比A高。对于图1 (b) 中的三角形，前两个三角形关系层满足地位理论状态顺序，但后两个三角形关系不满足地位理论状态顺序

## 3.2 Signed Motifs



**Fig. 2.** 38 different motifs in SiGAT, mean different influences from node  $j$  to node  $i$

we define positive/negative neighbors (2 motifs), positive /negative with direction neighbors (4 motifs) and 32 different triangle motifs



对每一个motif，都采用GAT方式聚合邻居节点

$$\alpha_{uv}^{m_i} = \frac{\exp \left( \text{LeakyReLU} \left( \mathbf{a}_{m_i}^T [\mathbf{W}_{m_i} \mathbf{X}(u) \| \mathbf{W}_{m_i} \mathbf{X}(v)] \right) \right)}{\sum_{k \in \mathcal{N}_{m_i}(u)} \exp \left( \text{LeakyReLU} \left( \mathbf{a}_{m_i}^T [\mathbf{W}_{m_i} \mathbf{X}(u) \| \mathbf{W}_{m_i} \mathbf{X}(k)] \right) \right)}, \quad (3)$$

$$\mathbf{X}_{m_i}(u) = \sum_{v \in \mathcal{N}_{m_i}(u)} \alpha_{uv}^{m_i} \mathbf{W}_{m_i} \mathbf{X}(v). \quad (4)$$

Loss

$$J_{\mathcal{G}}(\mathbf{Z}_u) = - \sum_{v^+ \in \mathcal{N}(u)^+} \log (\sigma(\mathbf{Z}_u^\top \mathbf{Z}_{v^+})) - Q \sum_{v^- \in \mathcal{N}(u)^-} \log (\sigma(-\mathbf{Z}_u^\top \mathbf{Z}_{v^-})), \quad (5)$$

无监督损失函数，该函数反映了朋友的嵌入是相似的，而敌人的嵌入是不相似的。

---

**Algorithm 1** SiGAT embedding generation (forward) algorithm

---

**Input:** Sigend directed Graph  $G(V, E, s)$ ;

Motifs list  $\mathcal{M}$ ; Epochs  $T$ ; Batch Size  $B$ ;

Motifs graph extract function  $F_{m_i}, \forall m_i \in \mathcal{M}$ ;

Aggregator GAT-AGGREGATOR $_{m_i}$  with the parameter  $\mathbf{W}_{m_i}, \mathbf{a}_{m_i}, \forall m_i \in \mathcal{M}$ ;

Weight matrices  $\mathbf{W}_1, \mathbf{W}_2$  and bias  $\mathbf{b}_1, \mathbf{b}_2$ ;

Non-linearity function Tanh ;

**Output:** Node representation  $\mathbf{Z}_u, \forall u \in V$

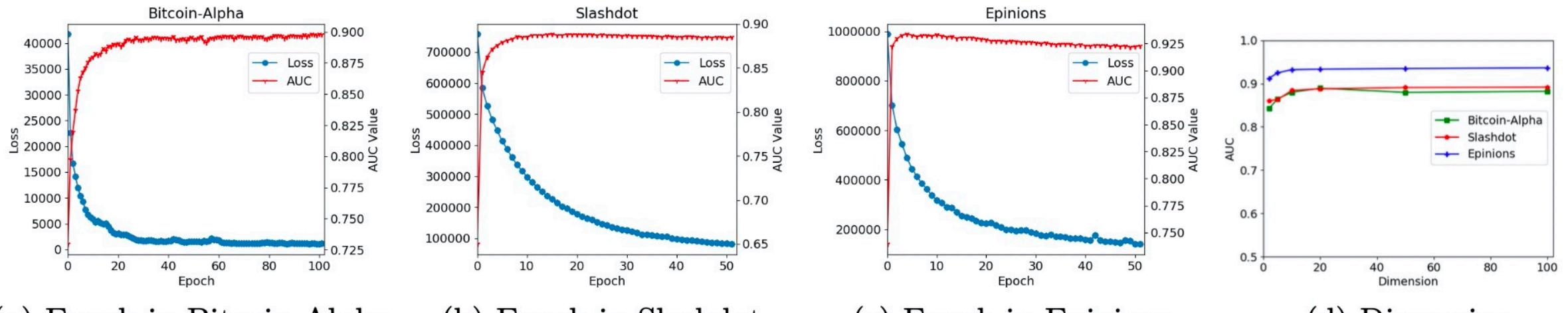
```
1:  $\mathbf{X}(u) \leftarrow \text{random}(0, 1), \forall u \in V$ 
2:  $G_{m_i} \leftarrow F_{m_i}(G), \forall m_i \in \mathcal{M}$ 
3:  $\mathcal{N}_{m_i}(u) \leftarrow \{v | (u, v) \in G_{m_i}\}, \forall m_i \in \mathcal{M}, \forall u \in V$ 
4: for epoch = 1, ..., T do
5:   for batch = 1, ...,  $|V|/B$  do
6:      $\mathcal{B} \leftarrow V_{(batch-1) \times B + 1 : batch \times B}$ 
7:     for  $u \in \mathcal{B}$  do
8:       for  $m_i \in \mathcal{M}$  do
9:          $\mathbf{X}_{m_i}(u) \leftarrow \text{GAT-AGGREGATOR}_{m_i}(\{\mathbf{X}_v, \forall v \in \mathcal{N}_{m_i}(v)\})$ 
10:      end for
11:       $\mathbf{X}'(u) \leftarrow \text{CONCAT}(\mathbf{X}(u), \mathbf{X}_{m_1}(u), \dots, \mathbf{X}_{m_{|\mathcal{M}|}}(u))$ 
12:       $\mathbf{Z}_u \leftarrow \mathbf{W}_2 \cdot \text{Tanh}(\mathbf{W}_1 \cdot \mathbf{X}'(u) + \mathbf{b}_1) + \mathbf{b}_2$ 
13:    end for
14:  end for
15: end for
16: return  $\mathbf{Z}_u$ 
```

---

**Table 2.** The results of Signed Link Prediction on three datasets

		Random	Unsigned Network Embedding			Signed Network Embedding			Feature Engineering	Graph Neural Network		
Dataset	Metric	Random	Deepwalk	Node2vec	LINE	SiNE	SIDE	SIGNet	FExtra	SGCN	SiGAT <sub>+-</sub>	SiGAT
Bitcoin-Alpha	Accuracy	0.9365	0.9365	0.9274	0.9350	0.9424	0.9369	0.9443	0.9477	0.9351	0.9427	<b>0.9480</b>
	F1	0.9672	0.9672	0.9623	0.9662	0.9699	0.9673	0.9706	0.9725	0.9658	0.9700	<b>0.9727</b>
	Macro-F1	0.4836	0.4836	0.5004	0.5431	0.6683	0.5432	0.7099	0.7069	0.6689	0.6570	<b>0.7138</b>
	AUC	0.6395	0.6435	0.7666	0.7878	0.8788	0.7832	<b>0.8972</b>	0.8887	0.8530	0.8699	0.8942
Slashdot	Accuracy	0.7740	0.7740	0.7664	0.7638	0.8269	0.7776	0.8391	0.8457	0.8200	0.8331	<b>0.8482</b>
	F1	0.8726	0.8726	0.8590	0.8655	0.8921	0.8702	0.8984	<b>0.9061</b>	0.8860	0.8959	0.9047
	Macro-F1	0.4363	0.4363	0.5887	0.4463	0.7277	0.5469	0.7559	0.7371	0.7294	0.7380	<b>0.7660</b>
	AUC	0.5415	0.5467	0.7622	0.5343	0.8423	0.7627	0.8749	0.8859	0.8440	0.8639	<b>0.8864</b>
Epinions	Accuracy	0.8530	0.8518	0.8600	0.8262	0.9131	0.9186	0.9116	0.9206	0.9092	0.9124	<b>0.9293</b>
	F1	0.9207	0.9198	0.9212	0.9040	0.9502	0.9533	0.9491	0.9551	0.9472	0.9498	<b>0.9593</b>
	Macro-F1	0.4603	0.4714	0.6476	0.4897	0.8054	0.8184	0.8065	0.8075	0.8102	0.8020	<b>0.8449</b>
	AUC	0.5569	0.6232	0.8033	0.5540	0.8882	0.8893	0.9091	<b>0.9421</b>	0.8818	0.9079	0.9333

只考虑正邻居和负邻居，模型可以验证注意机制是否有效



(a) Epoch in Bitcoin-Alpha    (b) Epoch in Slashdot    (c) Epoch in Epinions    (d) Dimension

**Fig. 4.** Parameter Analysis for the epoch and dimension in three datasets

模型对于超参数上的选择相对稳定

## Code

```
| def run( dataset='bitcoin_alpha', k=2):
```

adj\_lists1 : positive的点边关系dictionary

adj\_lists1\_1 : u->v的点边关系

adj\_lists1\_2 : v->u的点边关系

adj\_lists2 : negative的点边关系dictionary

adj\_lists2\_1 : u->v的点边关系

adj\_lists2\_2 : v->u的点边关系

adj\_lists3 : 所有的点边关系dictionary

1:  $\mathbf{X}(u) \leftarrow \text{random}(0, 1), \forall u \in V$

```
features = nn.Embedding(num_nodes, NODE_FEAT_SIZE) # 建立节点embedding
```

```
| class FeaExtra(object):
```

pos\_out\_edgelists : u->v的点边关系

pos\_in\_edgelists : v->u的点边关系

neg\_out\_edgelists : u->v的点边关系

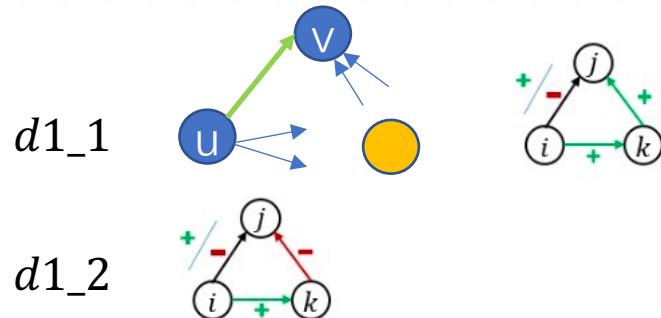
neg\_in\_edgelists : v->u的点边关系

```

def feature_part2(self, u, v): self: <fea_extra.FeaExtra object at 0x14d320828> u: 1 v: 0

d1_1 = len(set(self.pos_out_edgelists[u]).intersection(set(self.pos_in_edgelists[v])))
d1_2 = len(set(self.pos_out_edgelists[u]).intersection(set(self.neg_in_edgelists[v])))
d1_3 = len(set(self.neg_out_edgelists[u]).intersection(set(self.pos_in_edgelists[v])))
d1_4 = len(set(self.neg_out_edgelists[u]).intersection(set(self.neg_in_edgelists[v])))

```



```

d2_1 = len(set(self.pos_out_edgelists[u]).intersection(set(self.pos_out_edgelists[v])))
d2_2 = len(set(self.pos_out_edgelists[u]).intersection(set(self.neg_out_edgelists[v])))
d2_3 = len(set(self.neg_out_edgelists[u]).intersection(set(self.pos_out_edgelists[v])))
d2_4 = len(set(self.neg_out_edgelists[u]).intersection(set(self.neg_out_edgelists[v])))

```

```

d3_1 = len(set(self.pos_in_edgelists[u]).intersection(set(self.pos_out_edgelists[v])))
d3_2 = len(set(self.pos_in_edgelists[u]).intersection(set(self.neg_out_edgelists[v])))
d3_3 = len(set(self.neg_in_edgelists[u]).intersection(set(self.pos_out_edgelists[v])))
d3_4 = len(set(self.neg_in_edgelists[u]).intersection(set(self.neg_out_edgelists[v])))

```

```

d4_1 = len(set(self.pos_in_edgelists[u]).intersection(set(self.pos_in_edgelists[v])))
d4_2 = len(set(self.pos_in_edgelists[u]).intersection(set(self.neg_in_edgelists[v])))
d4_3 = len(set(self.neg_in_edgelists[u]).intersection(set(self.pos_in_edgelists[v])))
d4_4 = len(set(self.neg_in_edgelists[u]).intersection(set(self.neg_in_edgelists[v])))

```

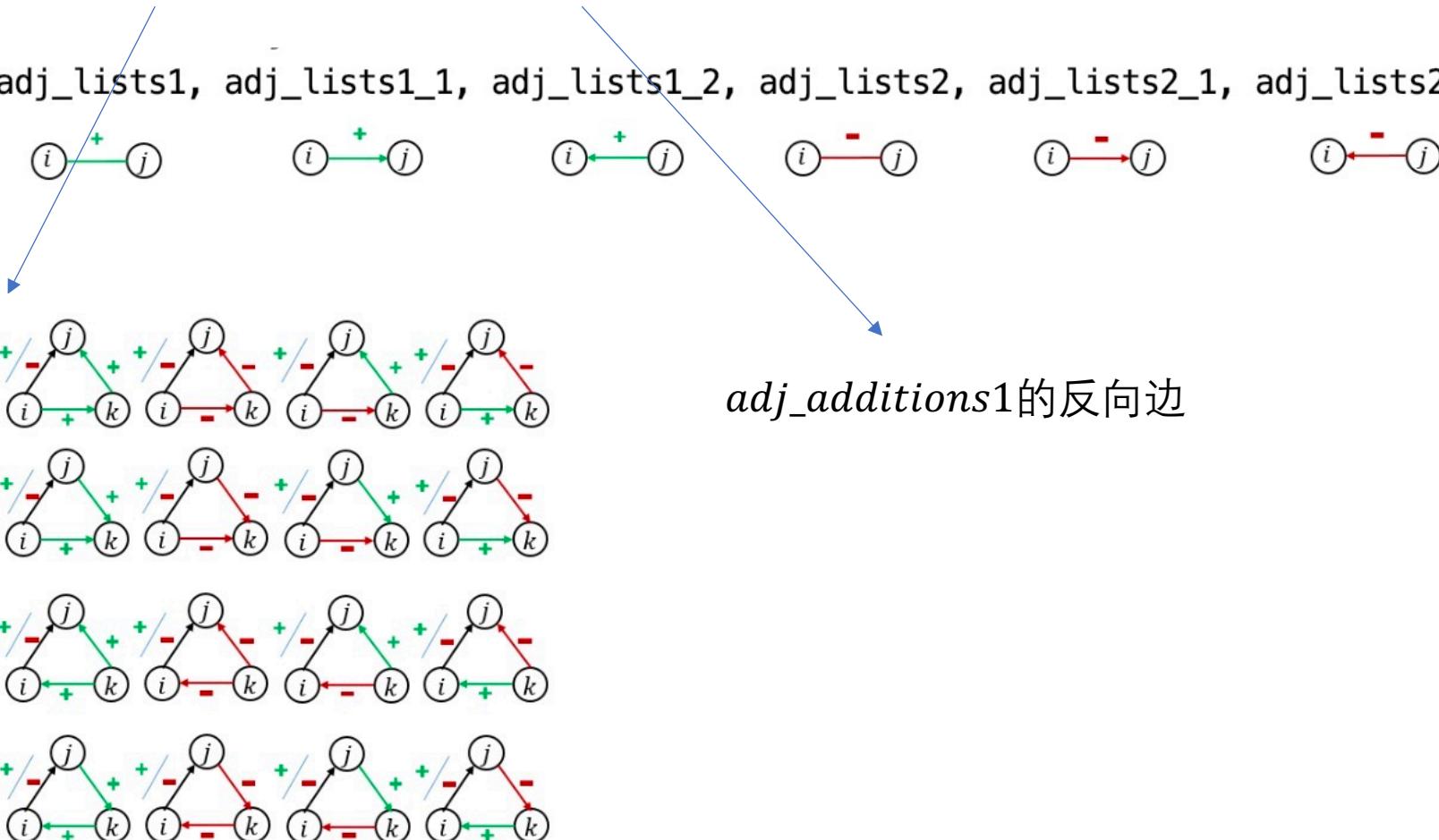
```
adj_additions1 = [defaultdict(set) for _ in range(16)]
adj_additions2 = [defaultdict(set) for _ in range(16)]
adj_additions0 = [defaultdict(set) for _ in range(16)]
```

三角形关系

```
adj_lists = adj_lists + adj_additions1 + adj_additions2
```

邻接矩阵+三角形关系

```
adj_lists = [adj_lists1, adj_lists1_1, adj_lists1_2, adj_lists2, adj_lists2_1, adj_lists2_2]
```



*adj\_additions1*的反向边

## GAT

```
adj_lists = list(map(func, adj_lists)) # 每个motifs的邻接矩阵  
features_lists = [features for _ in range(len(adj_lists))]  
aggs = [AttentionAggregator(features, NODE_FEAT_SIZE, NODE_FEAT_SIZE, num_nodes) for features, adj in  
        zip(features_lists, adj_lists)]
```

38个AttentionAggregator, 每个都包含features和motifs的adj

```
class AttentionAggregator(nn.Module):  
    def __init__(self, features, in_dim, out_dim, node_num, dropout_rate=DROPOUT, slope_ratio=0.1):  
        super(AttentionAggregator, self).__init__()  
  
        self.features = features  
        self.in_dim = in_dim  
        self.out_dim = out_dim  
        self.dropout = nn.Dropout(dropout_rate)  
        self.slope_ratio = slope_ratio  
        self.a = nn.Parameter(torch.FloatTensor(out_dim * 2, 1)) # W1 || W2  
        nn.init.kaiming_normal_(self.a.data)  
        self.special_spmm = SpecialSpmm()  
  
        self.out_linear_layer = nn.Linear(self.in_dim, self.out_dim) # W*h  
        self.unique_nodes_dict = np.zeros(node_num, dtype=np.int32)
```

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



```
enc1 = Encoder(features_lists, NODE_FEAT_SIZE, EMBEDDING_SIZE1, adj_lists, aggs)
```

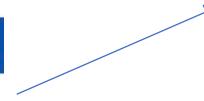
features

38motifs的邻接矩阵

```
↳ class Encoder(nn.Module):
```

```
    self.nonlinear_layer = nn.Sequential(
        nn.Linear(self.feat_dim * (len(adj_lists) + 1), self.feat_dim),
        nn.Tanh(),
        nn.Linear(self.feat_dim, self.embed_dim),
    )
```

11:

$$\mathbf{X}'(u) \leftarrow \text{CONCAT}(\mathbf{X}(u), \mathbf{X}_{m_1}(u), \dots, \mathbf{X}_{m_{|\mathcal{M}|}}(u))$$


```
>>> enc1.agg_37
AttentionAggregator(
    (features): Embedding(3786, 20)
    (dropout): Dropout(p=0.0, inplace=False)
    (speical_spmm): SpecialSpmm()
    (out_linear_layer): Linear(in_features=20, out_features=20, bias=True)
)
```

```
enc1 = Encoder(features_lists, NODE_FEAT_SIZE, EMBEDDING_SIZE1, adj_lists, aggs)
```

```
model = SiGAT(enc1)
```

$$\mathbf{X}'(u) \leftarrow \text{CONCAT}(\mathbf{X}(u), \mathbf{X}_{m_1}(u), \dots, \mathbf{X}_{m_{|\mathcal{M}|}}(u))$$

forward

```
loss = model.criterion(  
    nodes, adj_lists1, adj_lists2  
)  
  
def criterion(self, nodes, pos_neighbors, neg_neighbors): self: SiGAT(\n  (enc): Encoder(\n    (agg_  
    pos_neighbors_list = [set.union(pos_neighbors[i]) for i in nodes] # node的positive节点邻接 pos_ne  
    neg_neighbors_list = [set.union(neg_neighbors[i]) for i in nodes] # node的negative节点邻接 neg_ne  
    unique_nodes_list = list(set.union(*pos_neighbors_list).union(*neg_neighbors_list).union(nodes))  
    unique_nodes_dict = {n: i for i, n in enumerate(unique_nodes_list)} # node: index unique_nodes_  
    nodes_embs = self.enc(unique_nodes_list)
```

→ enc1 = Encoder(features\_lists, NODE\_FEAT\_SIZE, EMBEDDING\_SIZE1, adj\_lists, aggs)

Encoder.ecn1

model = SiGAT(enc1)

```
def forward(self, nodes):  
    neigh_feats = [agg.forward(nodes, adj) for adj, agg in zip(self.adj_lists, self.aggs)] # return 38 motifs GAT
```

AttentionAggregator

```
def forward(self, nodes, adj):  
    new_embeddings = self.out_linear_layer(self.features(n2)) W*h
```

# GAT

```
edge_h_2 = torch.cat((new_embeddings[edges[:, 0], :], new_embeddings[edges[:, 1], :]), dim=1) # Wh||Wh
edges_h = torch.exp(F.leaky_relu(torch.einsum("ij,jl->il", [edge_h_2, self.a]), self.slope_ratio)) # attention系数
```

```
row_sum = self.specical_spmm(edges.t(), edges_h[:, 0], torch.Size((batch_node_num, batch_node_num)), torch.ones(size=(batch_node_num, 1)).to(DEVICES))
```

点边关系      Attention系数

[1.. ..]矩阵

```
def forward(ctx, indices, values, shape, b):    ctx: <torch.autograd.function>
```

```
assert indices.requires_grad == False
```

```
a = torch.sparse_coo_tensor(indices, values, shape, device=DEVICES)
```

```
ctx.save_for_backward(a, b)
```

```
ctx.N = shape[0]
```

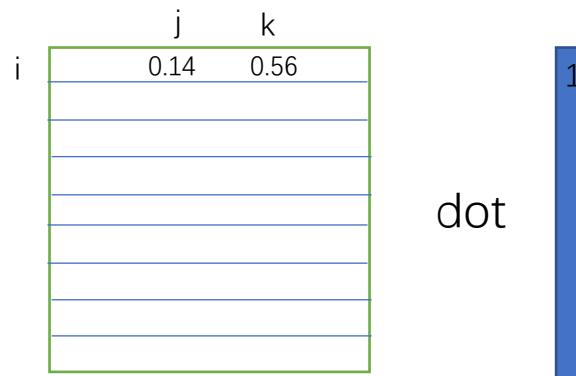
```
return torch.matmul(a, b)
```

点边关系

Attention系数

由edges组成的稀疏矩阵，值为attention

[3435, 3435]



对周围邻居向量求和，  
现在向量为1，则相当于对连接点的attention求和（归一化使用）

$$\alpha_{ij} = \frac{\exp(\text{LeakyReLU}(\mathbf{a}^T [\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_j]))}{\sum_{k \in \mathcal{N}_i} \exp(\text{LeakyReLU}(\mathbf{a}^T [\mathbf{W}\mathbf{h}_i \| \mathbf{W}\mathbf{h}_k])), \quad (1)}$$

```
results = self.specical_spmm(edges.t(), edges_h[:, 0], torch.Size((batch_node_num, batch_node_num)), new_embeddings)
```

```
def forward(ctx, indices, values, shape, b):    ctx: <torch.autograd.function>
    assert indices.requires_grad == False
    a = torch.sparse_coo_tensor(indices, values, shape, device=DEVICES)
    ctx.save_for_backward(a, b)
    ctx.N = shape[0]
    return torch.matmul(a, b)
```

节点embedding

点边关系

Attention系数

[3435, 3435]

对周围邻居向量按照attention系数求和

```
output_emb = results.div(row_sum) # 除法
```

对节点attention进行归一化操作

```
return output_emb[self.unique_nodes_dict[nodes]] # 返回minbatch中涉及到的node
```

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

首先，由于稀疏矩阵参与运算时其中的参数不能自动更新(pytorch中暂时没有其反向传播函数)，所以GAT自己写了稀疏矩阵(计算完注意力后的邻接矩阵)与稠密矩阵(特征)的乘法

```

def criterion(self, nodes, pos_neighbors, neg_neighbors):
    z1 = nodes_embs[unique_nodes_dict[node], :] # 该节点的embedding
    nodes_embs = self.enc(unique_nodes_list) # 输出节点embedding
    loss_total = 0 loss_total: 0
    for index, node in enumerate(nodes): 对每一个节点
        pos_neig_embs = nodes_embs[pos_neigs, :] # positive邻居节点 pos_neig_embs: tensor([[...]])
        loss_pku = -1 * torch.sum(F.logsigmoid(torch.einsum("nj,j->n", [pos_neig_embs, z1])))
        if neg_num > 0:
            neg_neig_embs = nodes_embs[neg_neigs, :]
            loss_pku = -1 * torch.sum(F.logsigmoid(-1 * torch.einsum("nj,j->n", [neg_neig_embs, z1])))
            loss_total += C * NEG_LOSS_RATIO * loss_pku

```

$$J_{\mathcal{G}}(\mathbf{Z}_u) = - \sum_{v^+ \in \mathcal{N}(u)^+} \log (\sigma(\mathbf{Z}_u^\top \mathbf{Z}_{v^+})) - Q \sum_{v^- \in \mathcal{N}(u)^-} \log (\sigma(-\mathbf{Z}_u^\top \mathbf{Z}_{v^-})), \quad (5)$$

得到节点embedding

```
embed = model.forward(values.tolist())
```

```
for  $m_i \in \mathcal{M}$  do
     $\mathbf{X}_{m_i}(u) \leftarrow \text{GAT-AGGREGATOR}_{m_i}(\{\mathbf{X}_v, \forall v \in \mathcal{N}_{m_i}(v)\})$ 
end for
 $\mathbf{X}'(u) \leftarrow \text{CONCAT}(\mathbf{X}(u), \mathbf{X}_{m_1}(u), \dots, \mathbf{X}_{m_{|\mathcal{M}|}}(u))$ 
 $\mathbf{Z}_u \leftarrow \mathbf{W}_2 \cdot \text{Tanh}(\mathbf{W}_1 \cdot \mathbf{X}'(u) + \mathbf{b}_1) + \mathbf{b}_2$ 
```

motifs

forward

[1,2,3,4,5,6 ....]

```
embed = model.forward(values.tolist())
```

SiGAT

AttentionAggregator

```
forward(self, nodes, adj)
```

```
n2 = torch.LongTensor(unique_nodes_list).to(DEVICES) # |E|
```

```
new_embeddings = self.out_linear_layer(self.features(n2))
```

[2745, 20]

```
self.out_linear_layer = nn.Linear(self.in_dim, self.out_dim) # W*h
```

GAT

```
edge_h_2 = torch.cat((new_embeddings[edges[:, 0], :], new_embeddings[edges[:, 1], :]), dim=1) # Wh || Wh
```

```
edges_h = torch.exp(F.leaky_relu(torch.einsum("ij,jl->il", [edge_h_2, self.a]), self.slope_ratio))
```

```
row_sum = self.special_spmm(edges.t(), edges_h[:, 0], torch.Size((batch_node_num, batch_node_num)), torch.ones(size=(batch_node_num, 1)).to(DEVICES))
```

```
a = torch.sparse_coo_tensor(indices, values, shape, device=DEVICES) # 稀疏矩阵
```

由edges组成的稀疏矩阵, 值为attention

[2745, 2745]

```
ctx.save_for_backward(a, b)
```

```
ctx.N = shape[0]
```

```
return torch.matmul(a, b)
```

Attention系数

[2745, 1]

相连接的attention求和

```
results = self.speical_spmm(edges.t(), edges_h[:, 0], torch.Size((batch_node_num, batch_node_num)), new_embeddings)
```

Attention系数

```
def forward(ctx, indices, values, shape, b):    ctx: <torch.autograd.function>
    assert indices.requires_grad == False
    a = torch.sparse_coo_tensor(indices, values, shape, device=DEVICES)
    ctx.save_for_backward(a, b)
    ctx.N = shape[0]
    return torch.matmul(a, b)
```

由edges组成的稀疏矩阵，值为attention  
[2745, 2745]

Attention系数

节点embedding

对周围邻居求和，GAT集合邻居

```
output_emb = results.div(row_sum)
```

对节点attention进行归一化操作

首先，由于稀疏矩阵参与运算时其中的参数不能自动更新(pytorch中暂时没有其反向传播函数)，所以GAT自己写了稀疏矩阵(计算完注意力后的邻接矩阵)与稠密矩阵(特征)的乘法

```
combined = torch.cat([self_feats] + neigh_feats, 1)
```

$$\mathbf{X}'(u) \leftarrow \text{CONCAT}(\mathbf{X}(u), \mathbf{X}_{m_1}(u), \dots, \mathbf{X}_{m_{|\mathcal{M}|}}(u))$$

```
combined = self.nonlinear_layer(combined)
```

[500, 20]

$$\mathbf{Z}_u \leftarrow \mathbf{W}_2 \cdot \text{Tanh}(\mathbf{W}_1 \cdot \mathbf{X}'(u) + \mathbf{b}_1) + \mathbf{b}_2$$

```
loss_pk = -1 * torch.sum(F.logsigmoid(torch.einsum("nj,j->n", [pos_neig_embs, z1])))
```

$$J_{\mathcal{G}}(\mathbf{Z}_u) = - \sum_{v^+ \in \mathcal{N}(u)^+} \log (\sigma(\mathbf{Z}_u^\top \mathbf{Z}_{v^+})) - Q \sum_{v^- \in \mathcal{N}(u)^-} \log (\sigma(-\mathbf{Z}_u^\top \mathbf{Z}_{v^-})), \quad (5)$$



## **SDGNN: Learning Node Representation for Signed Directed Networks**

Signed Directed Graph Neural Networks model (SDGNN).

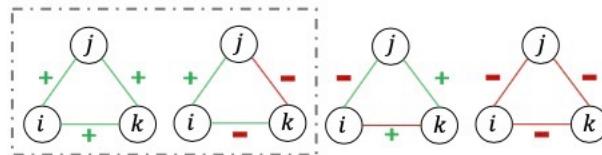
SDGNN : Signed Directed Graph Neural Networks

SGCN : 在无向的符号图中，采用了平衡理论，并使用mean聚合方式

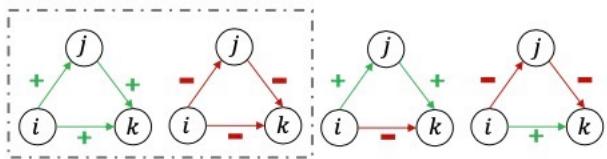
SiGAT : 可以使用在有向的符号图中，使用38个motifs来进行邻居的聚合，计算代价很大，并且在loss函数中没有使用到边的方向和三角形平衡理论

SDGNN :

- 提出一种逐层次的GNN模型应用在符号图上，并且在损失函数上重构符号，方向和三角形三种loss函数



(a) Balance theory



(b) Status theory

平衡理论认为，平衡的三角形关系比不平衡的三角形关系会更有说服力，因此在现实世界的网络中应该更为普遍。

例如，从A到B的正向链接意味着不仅B是A的朋友，而且B的地位比A高。对于图1 (b) 中的三角形，前两个三角形关系层满足地位理论状态顺序，但后两个三角形关系不满足地位理论状态顺序

## Comparison of Balance Theory and Status Theory

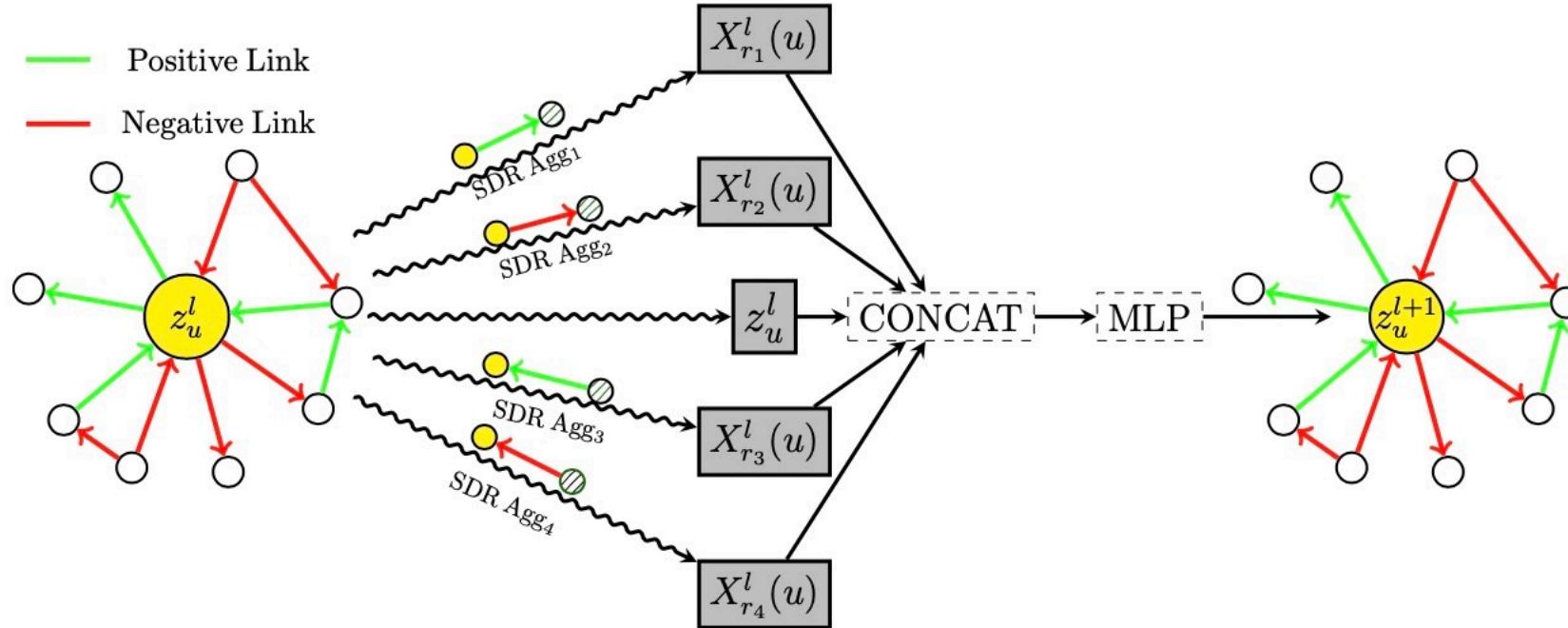
Table 1: The ratio of triads satisfying balance and/or status theory.

Dataset	Both	Only Balance	Only Status	Neither
Bitcoin-Alpha	0.673	0.208	0.094	0.025
Bitcoin-OTC	0.686	0.208	0.083	0.023
Wikirfa	0.686	0.059	0.189	0.066
Slashdot	0.751	0.167	0.066	0.016
Epinions	0.769	0.156	0.066	0.009

只有很小一部分三角形不满足两种理论。大约70%的三角形关系与两种理论都一致。作者认为这些都不满足理论的三角形可能是符号网络嵌入表示的噪声

平衡理论对三个顶点之间的关系进行建模，地位理论则通过传递性来捕获两个顶点之间的关系。两种理论的互补可以成为符号有向网络表示学习的关键。

should be distinguished. We first define 4 different signed directed relation (i.e.,  $u \rightarrow^+ v, u \rightarrow^- v, u \leftarrow^+ v, u \leftarrow^- v$ ).



对于带符号的有向图，节点之间的方向和符号反映了不同的关系和语义。

should be distinguished. We first define 4 different signed directed relation (i.e.,  $u \rightarrow^+ v, u \rightarrow^- v, u \leftarrow^+ v, u \leftarrow^- v$ ). Based on a signed directed relation (SDR)  $r_i$ , we can get

mean aggregators :

$$\begin{aligned} X_{r_i}^l(u) &= \sigma(\mathbf{W}_{r_i}^l \cdot \text{MEAN}(\{z_u^l\} \cup \{z_v^l, \forall v \in \mathcal{N}_{r_i}(u)\})) \\ z_u^{l+1} &= \text{MLP}(\text{CONCAT}(X^l(u), X_{r_1}^l(u), \dots, X_{r_i}^l(u))), \end{aligned} \quad (2)$$

attention aggregators :

$$\alpha_{uv}^{r_i} = \frac{\exp(\text{LeakyReLU}(\vec{\mathbf{a}}_{r_i}^\top [\mathbf{W}_{r_i}^l z_u^l \| \mathbf{W}_{r_i}^l z_v^l]))}{\sum_{k \in \mathcal{N}_{r_i}(u)} \exp(\text{LeakyReLU}(\vec{\mathbf{a}}_{r_i}^\top [\mathbf{W}_{r_i}^l z_u^l \| \mathbf{W}_{r_i}^l z_k^l]))}, \quad (3)$$

$$\begin{aligned} X_{r_i}^l(u) &= \sum_{v \in \mathcal{N}_{r_i}(u)} \alpha_{uv}^{r_i} \mathbf{W}_{r_i} z_v^l, \\ z_u^{l+1} &= \text{MLP}(\text{CONCAT}(X^l(u), X_{r_1}^l(u), \dots, X_{r_i}^l(u))). \end{aligned} \quad (4)$$

## Loss Function

we design three loss functions to reconstruct three critical features of signed networks:  
sign, direction, and triangle.

符号边

$$\begin{aligned} \mathcal{L}_{sign}(u, v) &= -y_{u,v} \log(\sigma(z_u^\top z_v)) - (1 - y_{u,v}) \log(1 - \sigma(z_u^\top z_v)) \\ \mathcal{L}_{sign} &= \sum_{e_{u,v} \in \mathcal{E}} \mathcal{L}_{sign}(u, v), \end{aligned} \quad (5)$$

边符号的loss, 逻辑回归损失函数

地位理论

$$\mathcal{L}_{direction}(u \rightarrow v) = (q_{uv} - (s(z_u) - s(z_v)))^2$$

$$q_{uv} = \begin{cases} \max(s(z_u) - s(z_v), \gamma) & u \rightarrow v : - \\ \min(s(z_u) - s(z_v), \gamma) & u \rightarrow v : + \end{cases}$$

$$\mathcal{L}_{direction} = \sum_{e_{u,v} \in \mathcal{E}} \mathcal{L}_{direction}(u \rightarrow v), \quad (\gamma = 0.5 \text{ in this paper}) \quad (6)$$

$$s(z) = \text{sigmoid}(W \cdot z + b)$$

$$s(z_u) - s(z_v) \quad u, v \text{ 的关系}$$

如果是negative边,  $u, v$ 地位距离应该大于 $\gamma$ , loss = 0  
如果是positive边,  $u, v$ 地位距离应该小于 $-\gamma$ , loss = 0

$u \rightarrow v : -$

说明u的地位高于v, 所以 $s(z_u) - s(z_v) > 0.5$  才没有loss

$u \rightarrow v : +$

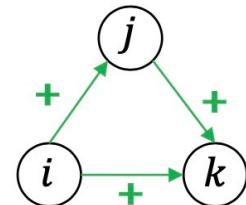
说明v的地位高于u, 所以 $s(z_u) - s(z_v) < -0.5$  才没有loss

## 三角关系理论

现实社会中，绝大多数三角形关系，都满足平衡理论和地位理论，所以求满足理论的三角形各个边之间的损失最小。

$$J_{tri} = \prod_{\Delta \in T} J_{\Delta},$$

$$\mathcal{L}_{triangle} = -\log J_{tri} = \sum_{\Delta \in T} -\log J_{\Delta} = \sum_{\Delta \in T} \mathcal{L}_{\Delta}, \quad (8)$$



如果  $e_{ij} = (+)$ ,  $z_i^T z_j$  越大越好

如果  $e_{ij} = (-)$ ,  $z_i^T z_j$  越小越好

$$\mathcal{L}_{\Delta_{i,j,k}} = \mathcal{L}_{ij} + \mathcal{L}_{ik} + \mathcal{L}_{kj},$$

$$\mathcal{L}_{ij} = -y_{i,j} \log P(+) | e_{ij}) - (1 - y_{i,j}) \log (1 - P(+) | e_{ij}))$$

$$= -y_{i,j} \log (\sigma(z_i^T z_j)) - (1 - y_{i,j}) \log (1 - \sigma(z_i^T z_j)),$$

(9)

注：代码里只用了  $\mathcal{L}_{ij}$  的边

边损失函数的权值是满足三角形条件的数量

最终loss：

$$\mathcal{L}_{loss} = \mathcal{L}_{sign} + \lambda_1 \mathcal{L}_{direction} + \lambda_2 \mathcal{L}_{triangle}, \quad (10)$$

Table 3: The results of link sign prediction on five datasets.

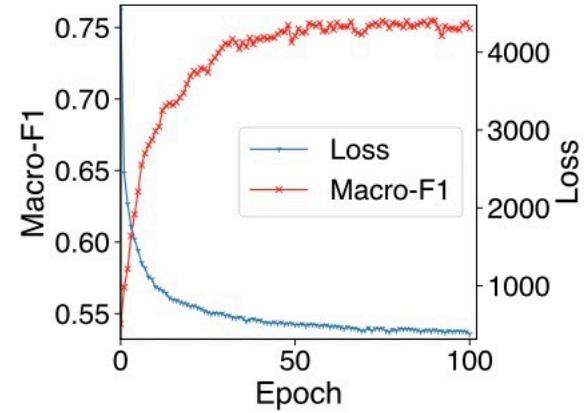
		Random	Unsigned Network Embedding			Signed Network Embedding			Feature Engineering	Graph Neural Network		
Dataset	Metric	Random	Deepwalk	Node2vec	LINE	SiNE	SIGNet	BESIDE	FeExtra	SGCN	SiGAT	SDGNN
Bitcoin-Alpha	Micro-F1	0.9367	0.9367	0.9355	0.9352	0.9458	0.9422	<u>0.9489</u>	0.9486	0.9256	0.9456	<b>0.9491</b>
	Binary-F1	0.9673	0.9673	0.9663	0.9664	0.9716	0.9696	<b>0.9732</b>	<u>0.9730</u>	0.9607	0.9714	0.9729
	Macro-F1	0.4837	0.4848	0.6004	0.5220	0.6869	0.6965	<u>0.7300</u>	0.7167	0.6367	0.7026	<b>0.7390</b>
	AUC	0.6146	0.6409	0.7576	0.7114	0.8728	0.8908	<u>0.8981</u>	0.8882	0.8469	0.8872	<b>0.8988</b>
Bitcoin-OTC	Micro-F1	0.9000	0.8937	0.9089	0.8911	0.9095	0.9229	0.9320	<b>0.9361</b>	0.9078	0.9268	<u>0.9357</u>
	Binary-F1	0.9473	0.9434	0.9507	0.9413	0.9510	0.9581	0.9628	<b>0.9653</b>	0.9491	0.9602	<u>0.9647</u>
	Macro-F1	0.4737	0.5281	0.6793	0.5968	0.6805	0.7386	<u>0.7843</u>	0.7826	0.7306	0.7533	<b>0.8017</b>
	AUC	0.6145	0.6596	0.7643	0.7248	0.8571	0.8935	<b>0.9152</b>	0.9121	0.8755	0.9055	<u>0.9124</u>
Wikirfa	Micro-F1	0.7797	0.7837	0.7814	0.7977	0.8338	0.8384	<u>0.8589</u>	0.8346	0.8489	0.8457	<b>0.8627</b>
	Binary-F1	0.8762	0.8779	0.8719	0.8827	0.8972	0.9001	<u>0.9117</u>	0.8987	0.9069	0.9042	<b>0.9142</b>
	Macro-F1	0.4381	0.4666	0.5626	0.5738	0.7319	0.7384	<u>0.7803</u>	0.7235	0.7527	0.7535	<b>0.7849</b>
	AUC	0.5423	0.5876	0.6930	0.6772	0.8602	0.8682	<b>0.8981</b>	0.8604	0.8563	0.8829	<u>0.8898</u>
Slashdot	Micro-F1	0.7742	0.7738	0.7526	0.7489	0.8265	0.8389	<u>0.8590</u>	0.8472	0.8296	0.8494	<b>0.8616</b>
	Binary-F1	0.8728	0.8724	0.8528	0.8525	0.8918	0.8983	<u>0.9105</u>	0.9070	0.8926	0.9055	<b>0.9128</b>
	Macro-F1	0.4364	0.4384	0.5390	0.5052	0.7273	0.7554	<b>0.7892</b>	0.7399	0.7403	0.7671	<b>0.7892</b>
	AUC	0.5370	0.5408	0.6709	0.6145	0.8409	0.8752	<b>0.9017</b>	0.8880	0.8534	0.8874	<u>0.8977</u>
Epinions	Micro-F1	0.8525	0.8214	0.8563	0.8535	0.9173	0.9113	<u>0.9336</u>	0.9226	0.9112	0.9293	<b>0.9355</b>
	Binary-F1	0.9204	0.9005	0.9170	0.9175	0.9525	0.9489	<u>0.9615</u>	0.9561	0.9486	0.9593	<b>0.9628</b>
	Macro-F1	0.4602	0.5131	0.6862	0.6305	0.8160	0.8060	<u>0.8601</u>	0.8130	0.8105	0.8454	<b>0.8610</b>
	AUC	0.5589	0.6702	0.8081	0.6835	0.8872	0.9095	0.9351	<b>0.9444</b>	0.8745	0.9333	<u>0.9411</u>

Table 4: Ablation study on different aggregators.

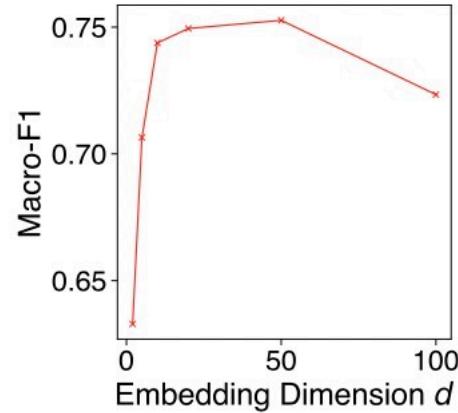
Metric	2-Layer-GAT-AGG	1-Layer-GAT-AGG	2-Layer-MEAN-AGG	1-Layer-MEAN-AGG
Micro-F1	0.9446	0.9442	0.9415	0.9399
Binary-F1	0.9706	0.9703	0.9689	0.9679
Macro-F1	0.7510	0.7516	0.7417	0.7411
AUC	0.9154	0.9095	0.9041	0.9000

Table 5: Ablation study on loss functions.

Metric	$\mathcal{L}_{sign}$	$\mathcal{L}_{sign} + \mathcal{L}_{direction}$	$\mathcal{L}_{sign} + \mathcal{L}_{triangle}$	$\mathcal{L}_{sign} + \mathcal{L}_{direction} + \mathcal{L}_{triangle}$
Micro-F1	0.9386	0.9438	0.9415	0.9475
Binary-F1	0.9677	0.9702	0.9690	0.9721
Macro-F1	0.6738	0.7414	0.7210	0.7585
AUC	0.8883	0.9082	0.9030	0.9109



(a) Epoch



(b)  $d$

Figure 3: Parameter analysis for sign prediction.

## Code

```
def run( dataset='bitcoin_alpha', k=2):
    adj_lists1 : positive的点边关系dictionary
    adj_lists1_1 : u->v的点边关系
    adj_lists1_2 : v->u的点边关系
    adj_lists2 : negative的点边关系dictionary
    adj_lists2_1 : u->v的点边关系
    adj_lists2_2 : v->u的点边关系

    adj_lists3 : 所有的点边关系dictionary

features = nn.Embedding(num_nodes, NODE_FEAT_SIZE) # 建立节点embedding
```

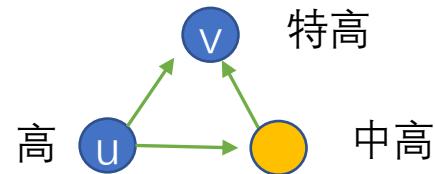
```

v_list1 = fea_model.feature_part2(i, j)  v_list1:
mask = [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1]  mask:
counts1 = np.dot(v_list1, mask) # 构造balance三角形, 在后面计算loss时候使用;  counts1: 0

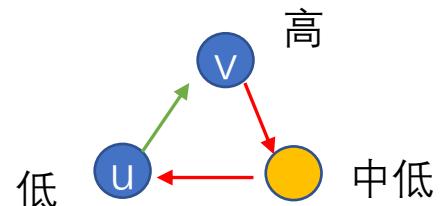
```

构造满足Balance theory 和 Status Theory的三角形

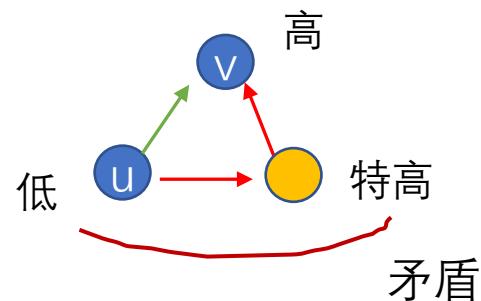
```
d1_1 = len(set(self.pos_out_edgelists[u]).intersection(set(self.pos_in_edgelists[v])))
```



```
d3_4 = len(set(self.neg_in_edgelists[u]).intersection(set(self.neg_out_edgelists[v])))
```



```
d1_4 = len(set(self.neg_out_edgelists[u]).intersection(set(self.neg_in_edgelists[v])))
```



虽然满足Balance theory 但不满足Status Theory

```
aggs = [aggregator(features, NODE_FEAT_SIZE, NODE_FEAT_SIZE, num_nodes) for adj in adj_lists]
```

```
>>> aggs 构建每个motifs下聚合函数
```

```
[AttentionAggregator(  
    (features): Embedding(3786, 20)  
    (dropout): Dropout(p=0.0, inplace=False)  
    (out_linear_layer): Linear(in_features=20, out_features=20, bias=True)]
```

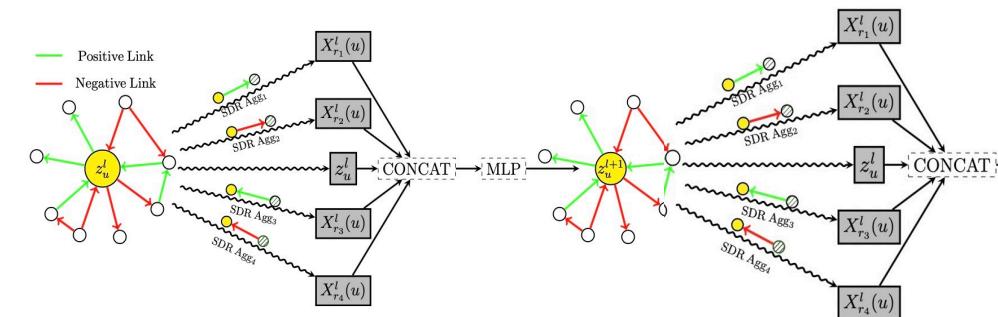
```
enc1 = Encoder(features, NODE_FEAT_SIZE, EMBEDDING_SIZE1, adj_lists, aggs)
```

```
enc1 = enc1.to(DEVICES)
```

```
>>> self.aggs
```

```
[AttentionAggregator(  
    (features): Embedding(3786, 20)  
    (dropout): Dropout(p=0.0, inplace=False)  
    (out_linear_layer): Linear(in_features=20, out_features=20, bias=True)  
, AttentionAggregator(  
    )
```

```
self.nonlinear_layer = nn.Sequential(  
    nn.Linear((len(adj_lists) + 1) * feature_dim, feature_dim),  
    nn.Tanh(),  
    nn.Linear(feature_dim, embed_dim)  
)
```



## 第二层中，聚合上层输出的结果

```
aggs2 = [aggregator(lambda n: enc1(n), EMBEDDING_SIZE1, EMBEDDING_SIZE1, num_nodes) for _ in adj_lists]
```

```
class AttentionAggregator(nn.Module):  
    def __init__(self, features, in_dim, out_dim, node_num, dropout_rate=DROPPOUT, slope_ratio=0.1):
```

4个AttentionAggregator

20

3786

第二层输出的Encoder

```
enc2 = Encoder(lambda n: enc1(n), EMBEDDING_SIZE1, EMBEDDING_SIZE1, adj_lists, aggs2)
```

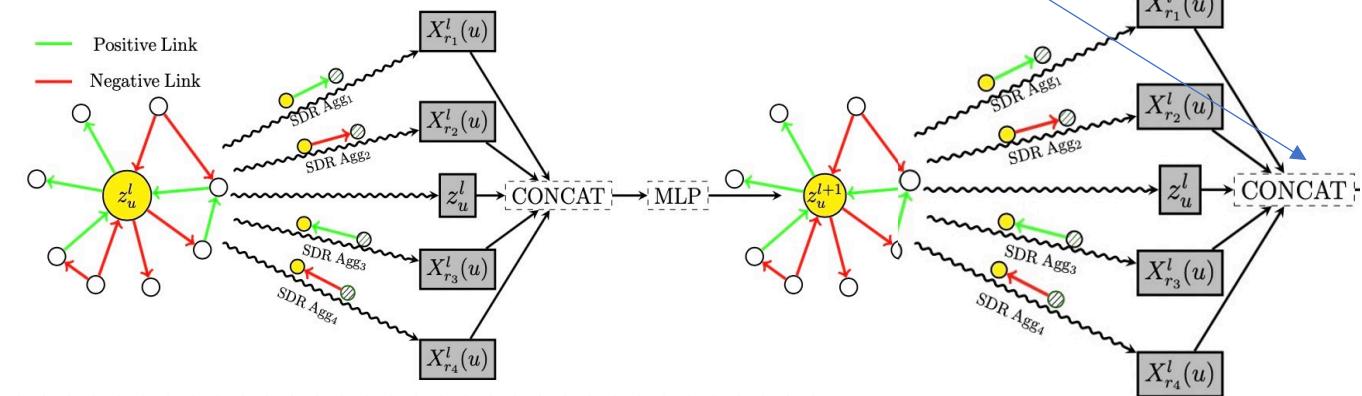
features

```
model = SDGNN(enc2)
```

最终的输出

```
self.enc = enc  
self.score_function1 = nn.Sequential(  
    nn.Linear(EMBEDDING_SIZE1, 1),  
    nn.Sigmoid())
```

```
self.fc = nn.Linear(EMBEDDING_SIZE1 * 2, 1)
```



```

class SDGNN(nn.Module):

    def __init__(self, enc):    self: SDGNN()   enc: Encoder()\n        (agg_0): AttentionAggregator()\n    super(SDGNN, self).__init__()\n    self.enc = enc\n    self.score_function1 = nn.Sequential(\n        nn.Linear(EMBEDDING_SIZE1, 1),\n        nn.Sigmoid()\n    )\n    self.score_function2 = nn.Sequential(\n        nn.Linear(EMBEDDING_SIZE1, 1),\n        nn.Sigmoid()\n    )\n    self.fc = nn.Linear(EMBEDDING_SIZE1 * 2, 1)

```

$$\mathcal{L}_{direction}(u \rightarrow v) = (q_{uv} - (s(z_u) - s(z_v)))^2$$

$$q_{uv} = \begin{cases} \max(s(z_u) - s(z_v), \gamma) & u \rightarrow v : - \\ \min(s(z_u) - s(z_v), \gamma) & u \rightarrow v : + \end{cases}$$

where  $s(z) = \text{sigmoid}(W \cdot z + b)$  is a score function

$$\mathcal{L}_{ij} = -y_{i,j} \log P(+) | e_{ij}) - (1 - y_{i,j}) \log (1 - P(+) | e_{ij}))$$

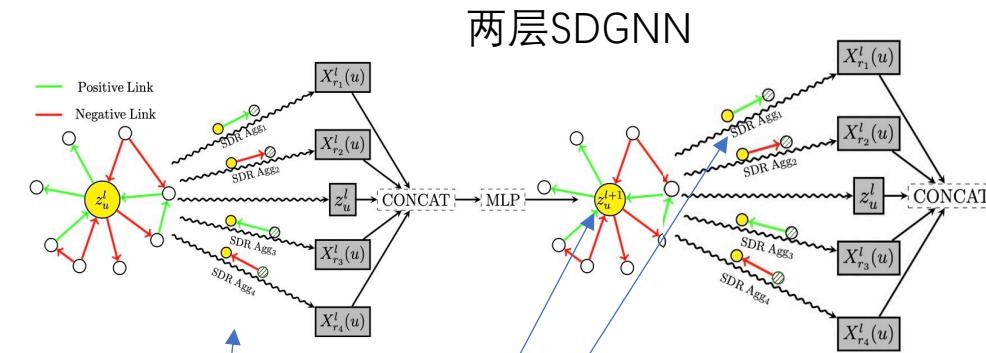
整体架构：

```
nodes_embs = self.enc(unique_nodes_list)
```

节点embedding

enc函数

Encoder



两层SDGNN

```
neigh_feats = [agg(nodes, adj, ind) for adj, agg, ind in zip(self.adj_lists, self.aggs, range(len(self.adj_lists)))]
```

每个motifs

GAT聚合周围邻居

第二层的第一个motifs

```
n2 = torch.LongTensor(unique_nodes_list).to(DEVICES)
```

```
f = self.features(n2)
```

```
>>> self.features
```

```
<function run.<locals>.<listcomp>.<lambda> at 0x143a567b8>
```

```
aggs2 = [aggregator(lambda n: enc1(n), EMBEDDING_SIZE1, EMBEDDING_SIZE1, num_nodes) for _ in adj_lists]
```

Encoder.forward

先经过enc1, n就是节点

需要第一层所有的motifs聚合后的节点信息

```
neigh_feats = [agg(nodes, adj, ind) for adj, agg, ind in zip(self.adj_lists, self.aggs, range(len(self.adj_lists)))]
```

多个motifs, 聚合了邻居节点后的节点embedding

聚合节点邻居信息

```
self_feats = self.features(torch.LongTensor(nodes).to(DEVICES)) #  
combined = torch.cat([self_feats] + neigh_feats, 1) # 邻居+节点本身  
combined = self.nonlinear_layer(combined)
```

第一层输出的信息

return: f = self.features(n2)

```
return combined
```

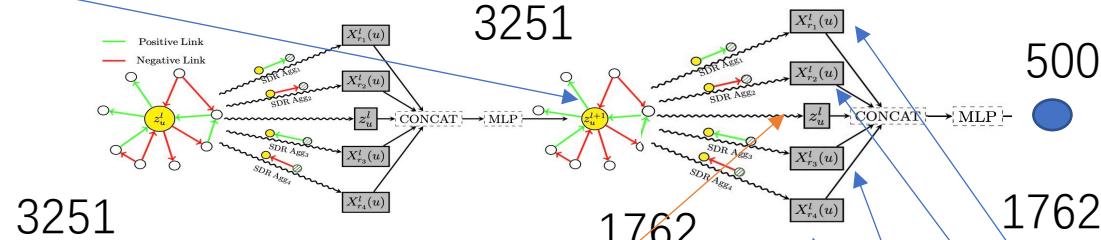
```
f = self.features(n2)
```

[3251,20]

```
new_embeddings = self.out_linear_layer(f)
```

W\*H

```
original_node_edge = np.array([self.unique_nodes_dict[nodes], self.unique_nodes_dict[nodes]]).T  
edges = np.vstack((edges, original_node_edge)) # 加上自连接的边
```



这3251个节点在第二层的一个motifs上做GAT，得到

第二层第一个motifs，聚合4个agg，再concat，聚合输出节点聚合后的embedding，最后聚合输出

第二层第二个motifs，聚合4个agg，再concat，聚合输出节点聚合后的embedding，最后聚合输出

第二层第三个motifs，聚合4个agg，再concat，聚合输出节点聚合后的embedding，最后聚合输出

第二层第四个motifs，聚合4个agg，再concat，聚合输出节点聚合后的embedding，最后聚合输出

node本身节点

Loss

```
pos_neig_embs = nodes_embs[pos_neigs, :] # pos节点embedding pos_neig_embs: tensor([[ 0.082  
loss_pk = F.binary_cross_entropy_with_logits(torch.einsum("nj,j->n", [pos_neig_embs, z1]),  
torch.ones(pos_num).to(DEVICES))
```

$$\mathcal{L}_{sign}(u, v) = -y_{u,v} \log \left( \sigma(z_u^\top z_v) \right) - (1 - y_{u,v}) \log \left( 1 - \sigma(z_u^\top z_v) \right)$$

```
z11 = z1.repeat(len(sta_pos_neighs), 1) # 重复n次 z11: tensor([-0.0819, 1.4090, 1.3168, -0.536  
rs = self.fc(torch.cat([z11, sta_pos_neig_embs], 1)).squeeze(-1) # Z1||pos_emb -> 1 rs: tensor([  
loss_pk += F.binary_cross_entropy_with_logits(rs, torch.ones(len(sta_pos_neighs)).to(DEVICES), \
```

$$\begin{aligned} \text{weight}=\text{pos\_neigs\_weight} \quad \mathcal{L}_{ij} &= -y_{i,j} \log P(+|e_{ij}) - (1 - y_{i,j}) \log(1 - P(+|e_{ij})) \\ ) \text{ 边类别预测} \quad &= -y_{i,j} \log (\sigma(z_i^\top z_j)) - (1 - y_{i,j}) \log(1 - \sigma(z_i^\top z_j)) \end{aligned}$$

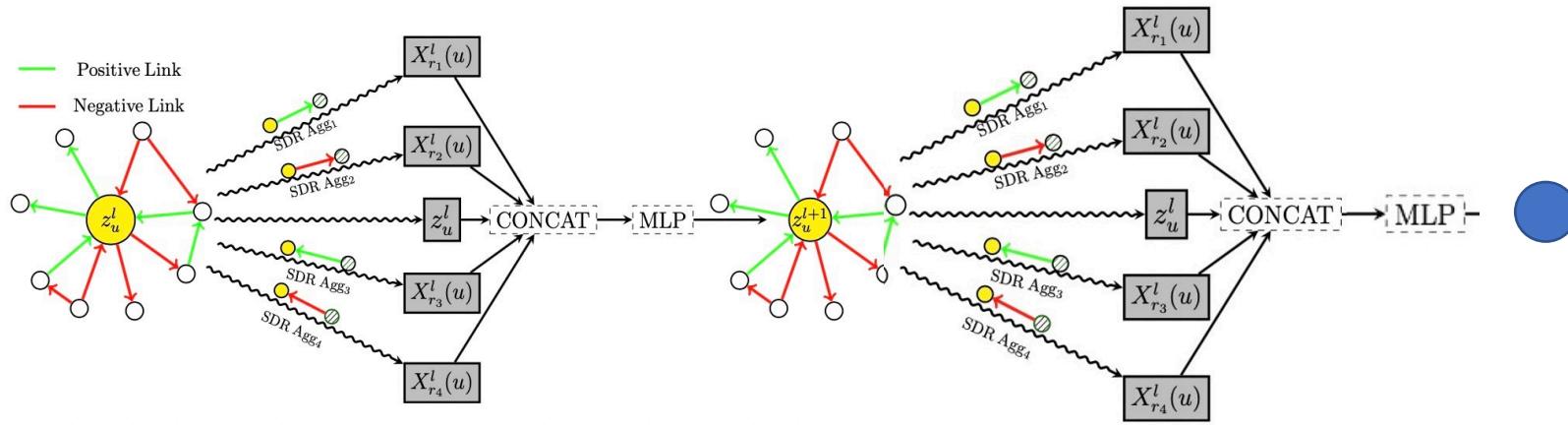
```
s1 = self.score_function1(z1).repeat(len(sta_pos_neighs), 1) # z1重复n次 s1: tensor([0.6899]  
s2 = self.score_function2(sta_pos_neig_embs) s2: tensor([0.4867], \n [0.3916], \n
```

```
q = torch.where((s1 - s2) > -0.5, torch.Tensor([-0.5]).repeat(s1.shape).to(DEVICES), s1 - s2)  
tmp = (q - (s1 - s2))  
loss_pk += 5 * torch.einsum("ij,ij->", [tmp, tmp])
```

$$\mathcal{L}_{direction}(u \rightarrow v) = (q_{uv} - (s(z_u) - s(z_v)))^2$$

$$q_{uv} = \begin{cases} \max(s(z_u) - s(z_v), \gamma) & u \rightarrow v : - \\ \min(s(z_u) - s(z_v), \gamma) & u \rightarrow v : + \end{cases}$$

$\overline{-\gamma}$





# Dynamic Graph

动态网络为网络建模和预测增加了新的维度-时间

这个新的维度从根本上影响网络属性，使网络数据更强大地表示出来，进而提高使用此类数据的方法的预测能力

## A. Dynamic network representations

动态网络表示可以按时间粒度分为四个不同的级别

- (i) Static
- (ii) edge weighted
- (iii) discrete
- (iv) continuous networks

不关注图中的动态性信息，而将其作为一张静态图同等处理  
动态信息只是作为一张静态图中的节点或者边的labels而存在  
离散表示使用一组有序的图(快照)来表示动态图  
使用确切时间信息的表示形式



其中静态网络是最粗糙的，连续表示是最精细的。随着时间粒度的增加，模型的复杂性也随之增加。

## *Discrete Representation:*

离散表示使用一组有序的图(快照)来表示动态图。

$$DG = \{G^1, G^2, \dots, G^T\} \quad T - \text{快照数, 多少个时间间隔}$$

## *Continuous Representation:*

连续的网络是具有确切时间信息的表示形式。可以总结为三个连续的表示形式

### 1. The event-based representation

它是基于链接持续时间的动态网络的表示

$$EB = \{(u_i, v_i, t_i, \Delta_i); i = 1, 2, \dots\} \quad u_i, v_i - \text{表示发生事件的一对节点} \quad t_i - \text{发生时间} \quad \Delta_i - \text{持续时间}$$

### 2. The contact sequence representation

在接触序列中, 链接是瞬时的发生的, 不存在持续时间。比如电子邮件的发送

$$CS = \{(u_i, v_i, t_i); i = 1, 2, \dots\}$$

### 3. The graph stream representation

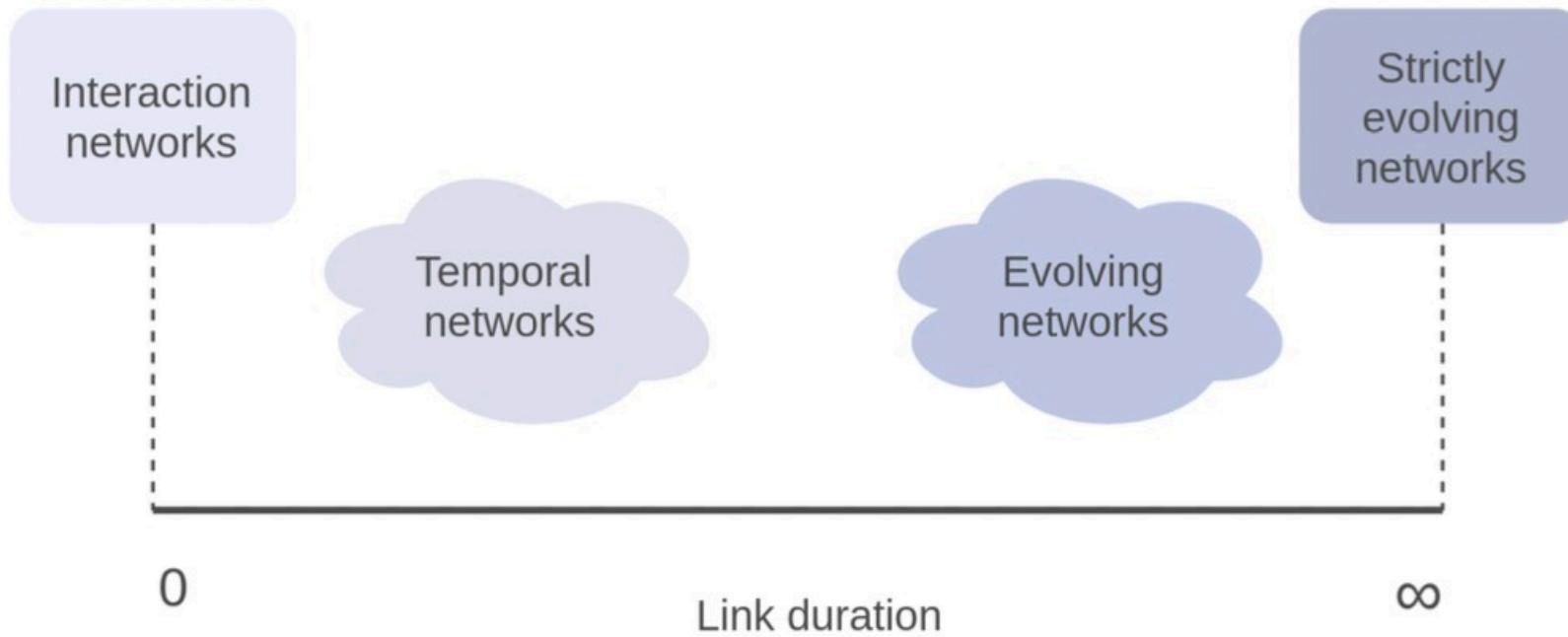
基于事件的表示, 它将链接的出现和链接的消失视为单独的事件。

$$GS = \{e_1, e_2, \dots\}$$

$$e_i = (u_i, v_i, t_i, \delta_i) \quad \delta_i \in \{-1, 1\} \quad \text{表示消失边和出现边}$$

## B. Link duration spectrum

链接持续时间区分不同类型的网络。



Interaction networks : 一种时间网络，其中的链接是瞬时事件。比如电子邮件发送

Temporal networks : 边有一定的持续时间，但比较短。比如社交网络中，人与人的交谈

Evolving networks : 链接持续存在的时间如此长，比如雇佣关系

Strictly evolving networks : 链接出现后就会一直出现，比如引文网络

### C. Node dynamics

动态网络之间的另一个区别因素是节点可能出现还是消失

Static : 节点数量在一段时间内保持不变

Dynamic : 节点可能出现和消失

Growing networks : 只可能出现节点的网络

Dimension	Network types
Temporal granularity	Static, edge-weighted, discrete, continuous
Link duration	Interaction, temporal, evolving, strictly evolving
Node dynamics	Node static, node dynamic, node appearing, node disappearing

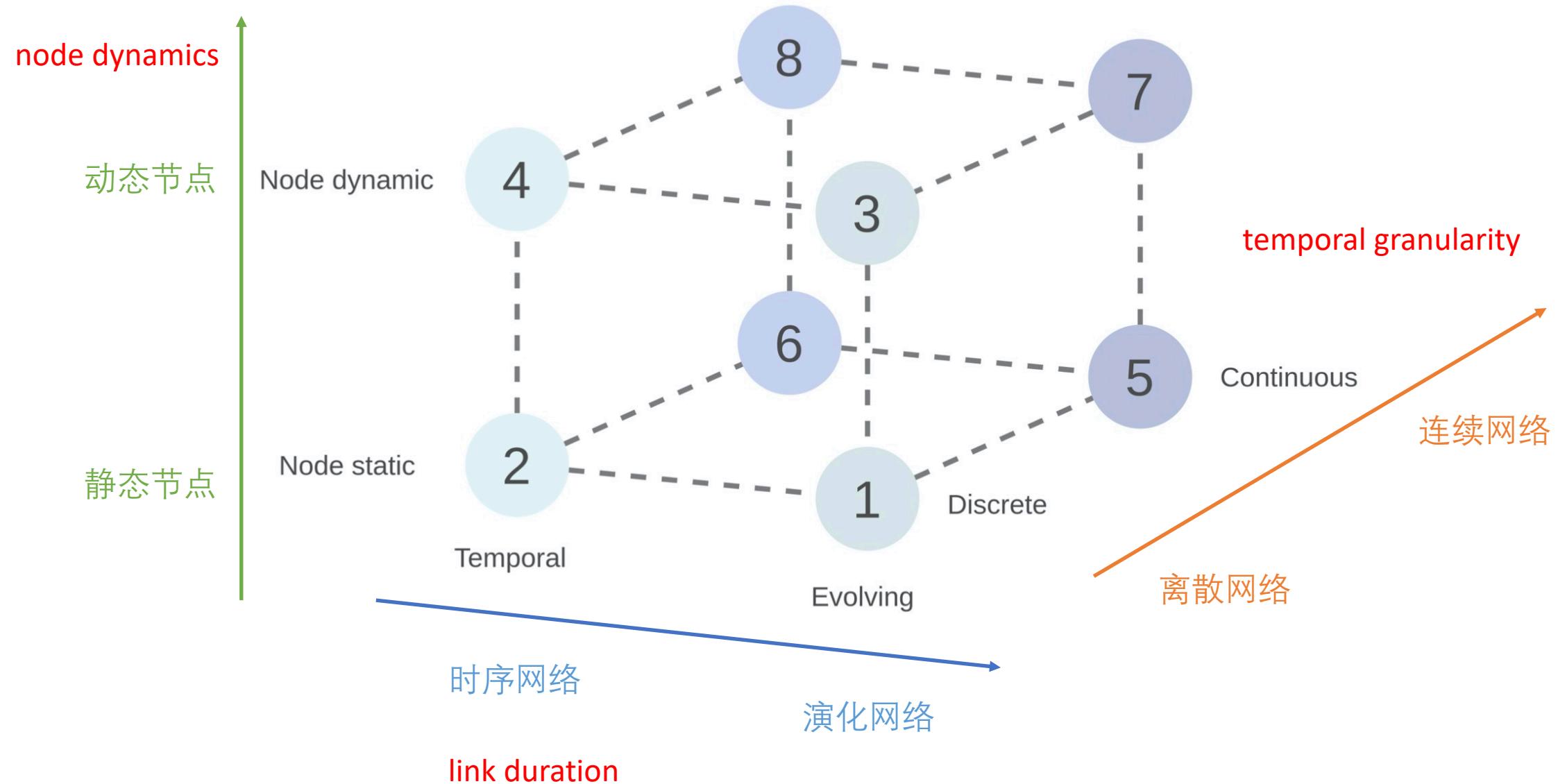
定义了按时间粒度排序的不同动态表示

定义了链路持续时间的网络类型

定义了节点动态的网络类型

## D. The dynamic network cube

排除特殊情况，这些是动态图常研究的方向



# 动态图网络算法

## 离散网络

- 1) Stacked DGNNs (堆叠DGNN) 单独的GNN处理图的每个快照，并将每个GNN输出到RNN模型中
- 2) Integrated DGNNs (集成DGNN) 将gnn和rnn结合在一层，从而结合空间和时间信息的建模。

## 连续网络

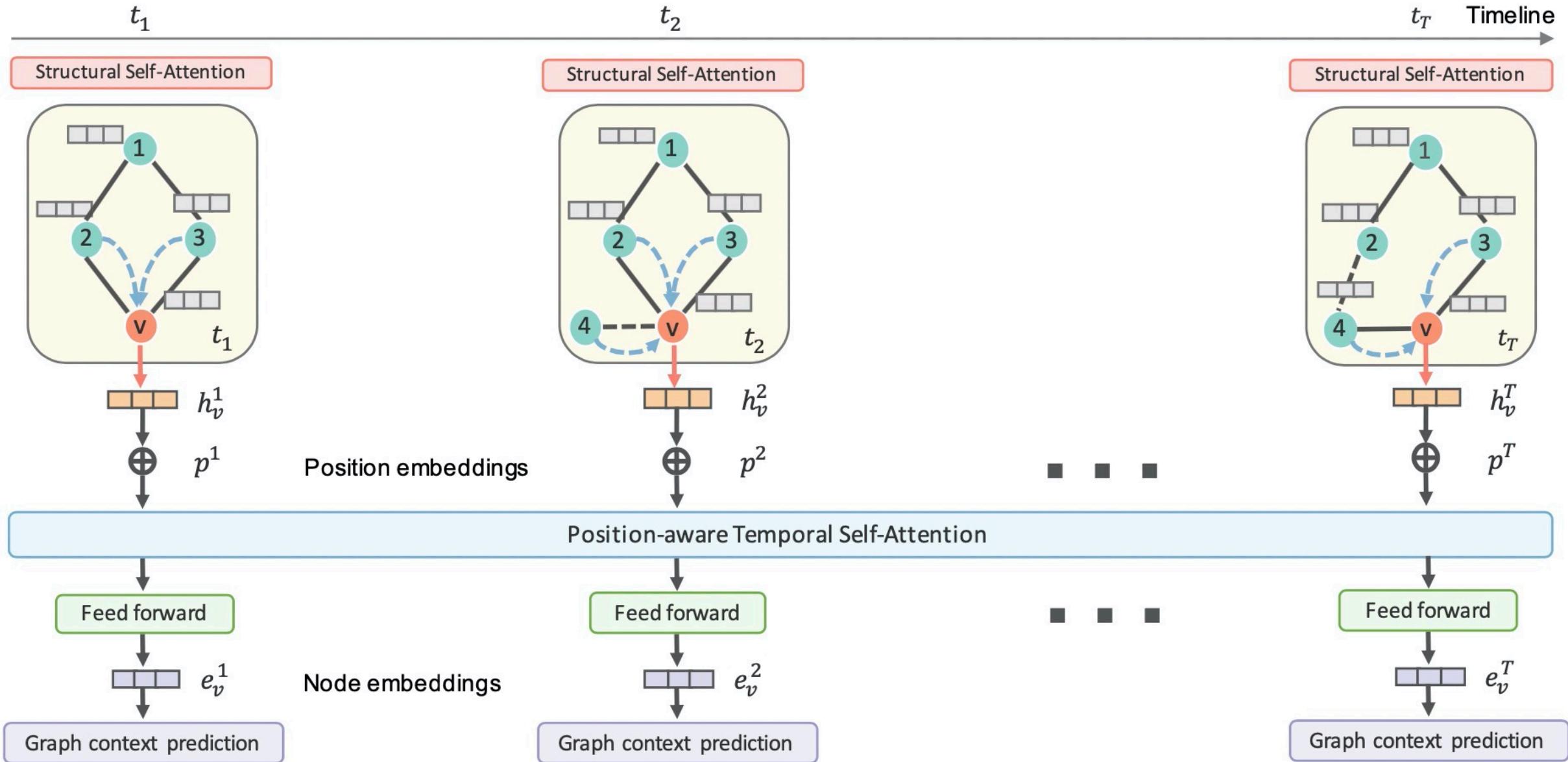
- 1) 基于RNN的方法
- 2) 基于点过程的方法

Model type	Model name	Encoder	Link addition	Link deletion	Node addition	Node deletion	Network type
Discrete networks							
Stacked DGNN	GCRN-M1 [58]	Spectral GCN [59] & LSTM	Yes	Yes	No	No	Any
	WD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	CD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	RgCNN [61]	Spatial GCN [62] & LSTM	Yes	Yes	No	No	Any
	DyGGNN [63]	GGNN [64] & LSTM	Yes	Yes	No	No	Any
	DySAT [67]	GAT [68] & temporal attention from [69]	Yes	Yes	Yes	Yes	Any
Integrated DGNN	GCRN-M2 [58]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	GC-LSTM [72]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	EvolveGCN [71]	LSTM integrated in a GCN [57]	Yes	Yes	Yes	Yes	Any
	LRGCN [73]	R-GCN [75] integrated in an LSTM	Yes	Yes	No	No	Any
	RE-Net [74]	R-GCN [75] integrated in several RNNs	Yes	Yes	No	No	Knowledge network
Continuous networks							
RNN based							
	Streaming GNN [86]	Node embeddings maintained by architecture consisting of T-LSTM [88]	Yes	No	Yes	No	Directed strictly evolving
	JODIE [87]	Node embeddings maintained by an RNN based architecture	Yes	No	No	No	Bipartite and interaction
TPP based							
	Know-Evolve [89]	TPP parameterised by an RNN	Yes	No	No	No	Interaction, knowledge network
	DyREP [36]	TPP parameterised by an RNN aided by structural attention	Yes	No	Yes	No	Strictly evolving
	LDG [90]	TPP, RNN and self-attention	Yes	No	Yes	No	Strictly evolving
	GHN [92]	TPP parameterised by a continuous time LSTM [93]	Yes	No	No	No	Interaction, knowledge network



# DySAT

Dynamic Self-Attention Network

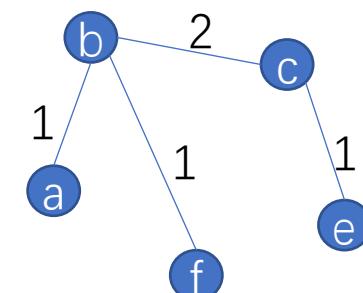


## 1. 创建数据

根据离散型数据构建，也就是快照的方式。Time step 按月进行分割

时间	graph	time step
2021.5.1	a-b; b-c;	1
2021.5.2	c-e; b-c;	1
2021.5.10	b-f;	1
2021.6.1	a-b; b-d;	2
2021.6.2	c-e;	2
2021.7.5	f-g;	3
2021.7.9	g-a;	3

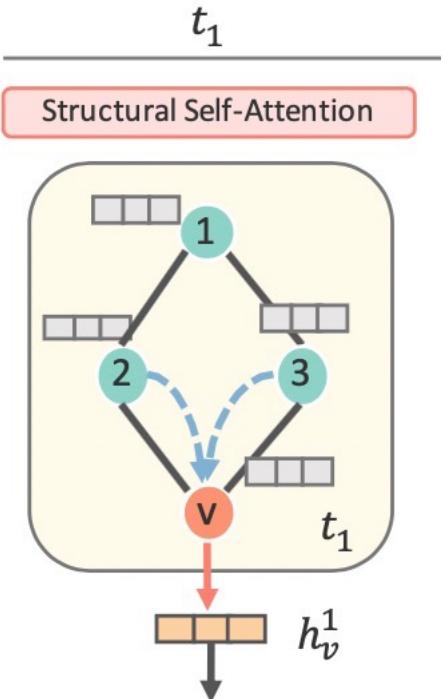
按照time-step构建图



time-step 1

## 4.1 STRUCTURAL SELF-ATTENTION

1. 表示节点的特征，如果没有特征，则用one-hot表示

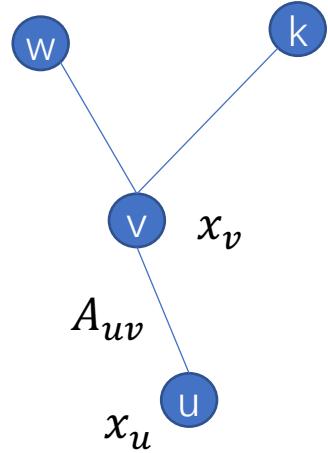


2. 计算attention系数

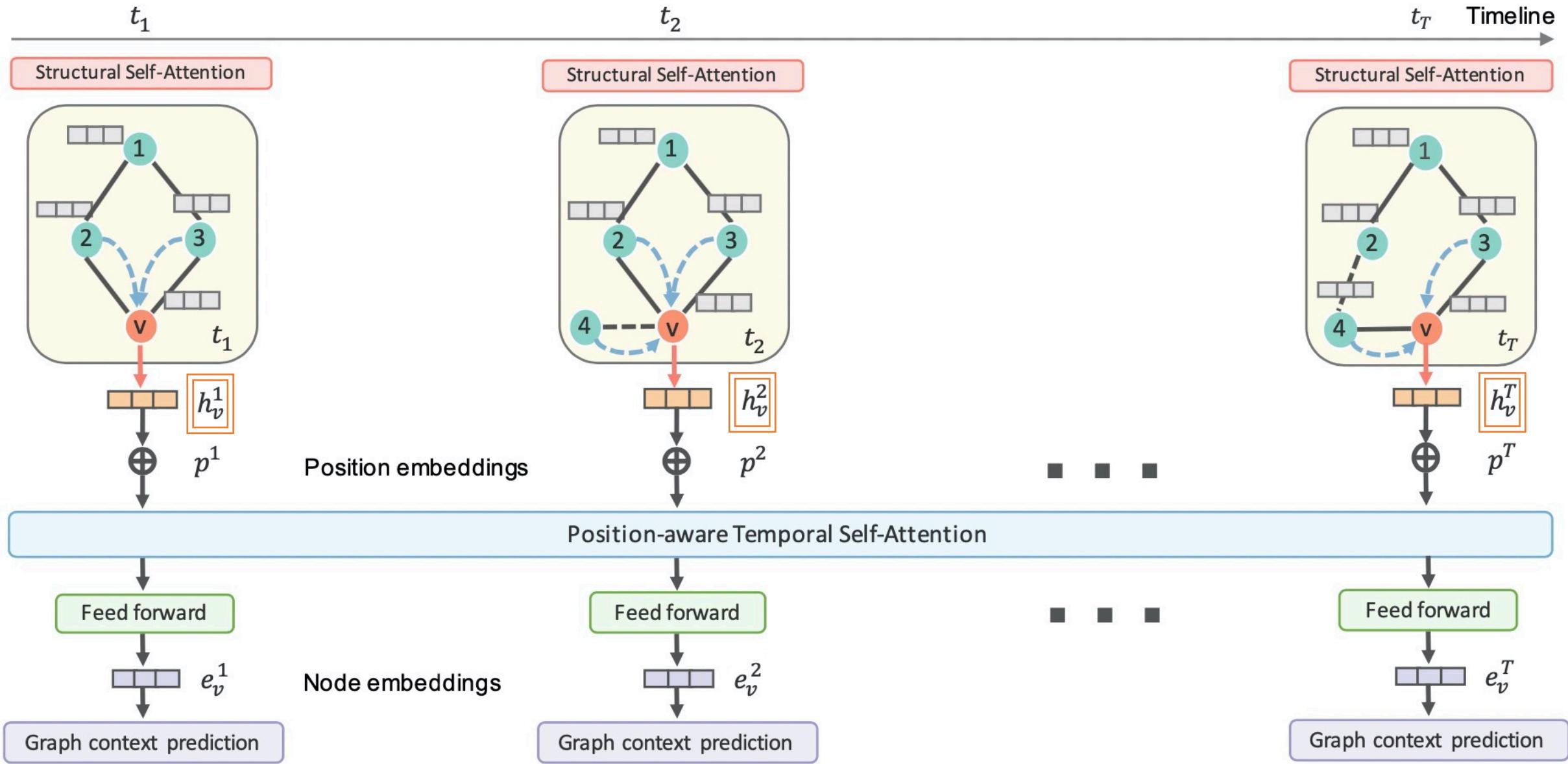
$$\alpha_{uv} = \frac{\exp\left(\sigma\left(A_{uv} \cdot \mathbf{a}^T [\mathbf{W}^s \mathbf{x}_u || \mathbf{W}^s \mathbf{x}_v]\right)\right)}{\sum_{w \in \mathcal{N}_v} \exp\left(\sigma\left(A_{wv} \cdot \mathbf{a}^T [\mathbf{W}^s \mathbf{x}_w || \mathbf{W}^s \mathbf{x}_v]\right)\right)}$$

3. 计算聚合邻居节点后的embedding

$$\mathbf{z}_v = \sigma\left(\sum_{u \in \mathcal{N}_v} \alpha_{uv} \mathbf{W}^s \mathbf{x}_u\right)$$

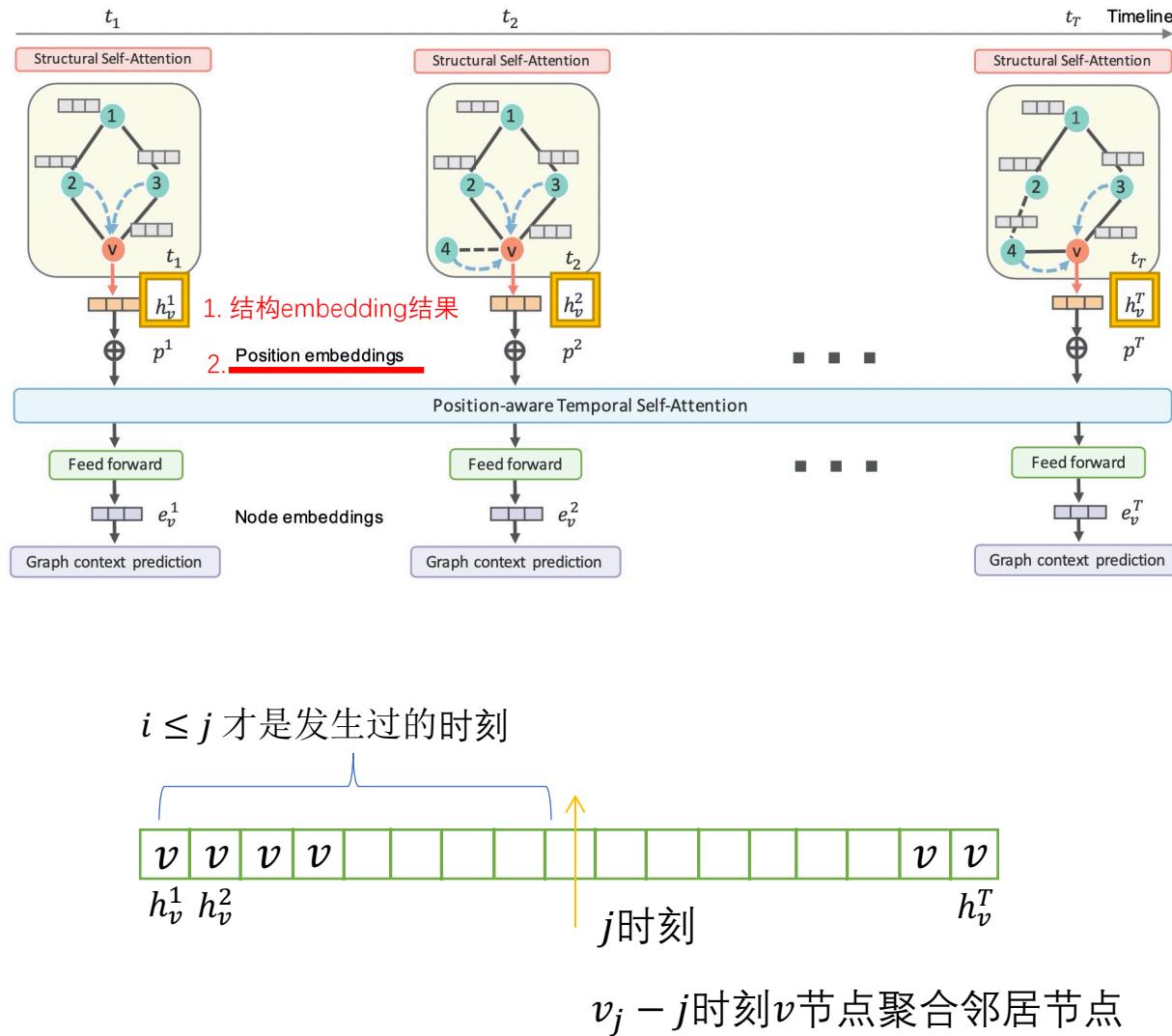


这里和GAT不同的地方在，多了一个边的权重



## 4.2 TEMPORAL SELF-ATTENTION

时间自注意层的关键目标是捕获图结构性在多个时间步长的变化



1. 结构attention的输出结果  $h_v^1, h_v^2, \dots, h_v^T$
2. 加入位置embedding(Position embeddings)
3. 采用self-attention计算每个节点的embedding

$W_q, W_k, W_v$  – selfattention参数

$X_v$  – 节点embedding

$$e_v^{ij} = \left( \frac{((\mathbf{X}_v \mathbf{W}_q)(\mathbf{X}_v \mathbf{W}_k)^T)_{ij}}{\sqrt{F'}} + M_{ij} \right)$$

Query      Key  
缩放因子

$$M_{ij} = \begin{cases} 0, & i \leq j \\ -\infty, & \text{otherwise} \end{cases}$$

$$\beta_v^{ij} = \frac{\exp(e_v^{ij})}{\sum_{k=1}^T \exp(e_v^{ik})}$$

$$\mathbf{Z}_v = \boldsymbol{\beta}_v(\mathbf{X}_v \mathbf{W}_v)$$

Value

## 4.4 DYSAT ARCHITECTURE

在每个时间步，采用随机游走的方式，获得节点 $u$ 的上下文节点 $v$ ，构建正样本，负样本采用负采样方式获得

$$L_v = \sum_{t=1}^T \sum_{u \in \mathcal{N}_{walk}^t(v)} -\log(\sigma(\langle \mathbf{e}_u^t, \mathbf{e}_v^t \rangle)) - w_n \cdot \sum_{u' \in P_n^t(v)} \log(1 - \sigma(\langle \mathbf{e}_{u'}^t, \mathbf{e}_v^t \rangle))$$

*positive* 节点的内积                                           *negative* 节点的内积

negative sampling ratio

针对每一个时刻内

Method	Enron		UCI		Yelp		ML-10M	
	Micro-AUC	Macro-AUC	Micro-AUC	Macro-AUC	Micro-AUC	Macro-AUC	Micro-AUC	Macro-AUC
node2vec	83.72 $\pm$ 0.7	83.05 $\pm$ 1.2	79.99 $\pm$ 0.4	80.49 $\pm$ 0.6	67.86 $\pm$ 0.2	65.34 $\pm$ 0.2	87.74 $\pm$ 0.2	87.52 $\pm$ 0.3
G-SAGE	82.48* $\pm$ 0.6	81.88* $\pm$ 0.5	79.15* $\pm$ 0.4	82.89* $\pm$ 0.2	60.95 $^{\dagger}$ $\pm$ 0.1	58.56 $^{\dagger}$ $\pm$ 0.2	86.19 $^{\ddagger}$ $\pm$ 0.3	89.92 $^{\ddagger}$ $\pm$ 0.1
G-SAGE + GAT	72.52 $\pm$ 0.4	73.34 $\pm$ 0.6	74.03 $\pm$ 0.4	79.83 $\pm$ 0.2	66.15 $\pm$ 0.1	65.09 $\pm$ 0.2	83.97 $\pm$ 0.3	84.93 $\pm$ 0.1
GCN-AE	81.55 $\pm$ 1.5	81.71 $\pm$ 1.5	80.53 $\pm$ 0.3	83.50 $\pm$ 0.5	66.71 $\pm$ 0.2	65.82 $\pm$ 0.2	85.49 $\pm$ 0.1	85.74 $\pm$ 0.1
GAT-AE	75.71 $\pm$ 1.1	75.97 $\pm$ 1.4	79.98 $\pm$ 0.2	81.86 $\pm$ 0.3	65.92 $\pm$ 0.1	65.37 $\pm$ 0.1	87.01 $\pm$ 0.2	86.75 $\pm$ 0.2
DynamicTriad	80.26 $\pm$ 0.8	78.98 $\pm$ 0.9	77.59 $\pm$ 0.6	80.28 $\pm$ 0.5	63.53 $\pm$ 0.3	62.69 $\pm$ 0.3	88.71 $\pm$ 0.2	88.43 $\pm$ 0.1
DynGEM	67.83 $\pm$ 0.6	69.72 $\pm$ 1.3	77.49 $\pm$ 0.3	79.82 $\pm$ 0.5	66.02 $\pm$ 0.2	65.94 $\pm$ 0.2	73.69 $\pm$ 1.2	85.96 $\pm$ 0.3
DynAERNN	72.02 $\pm$ 0.7	72.01 $\pm$ 0.7	79.95 $\pm$ 0.4	83.52 $\pm$ 0.4	69.54 $\pm$ 0.2	68.91 $\pm$ 0.2	87.73 $\pm$ 0.2	89.47 $\pm$ 0.1
<b>DySAT</b>	<b>85.71</b> $\pm$ 0.3	<b>86.60</b> $\pm$ 0.2	<b>81.03</b> $\pm$ 0.2	<b>85.81</b> $\pm$ 0.1	<b>70.15</b> $\pm$ 0.1	<b>69.87</b> $\pm$ 0.1	<b>90.82</b> $\pm$ 0.3	<b>93.68</b> $\pm$ 0.1

Table 2: Experiment results on single-step link prediction (micro and macro averaged AUC with standard deviation). We show GraphSAGE (denoted by G-SAGE) results with the best performing aggregators for each dataset (\* represents GCN,  $\dagger$  represents LSTM, and  $\ddagger$  represents max-pooling).

*G – SAGE + GAT:* GraphSAGE中实现了一个图关注层作为附加聚合器

*GCN – AE:* GCN方式训练AE

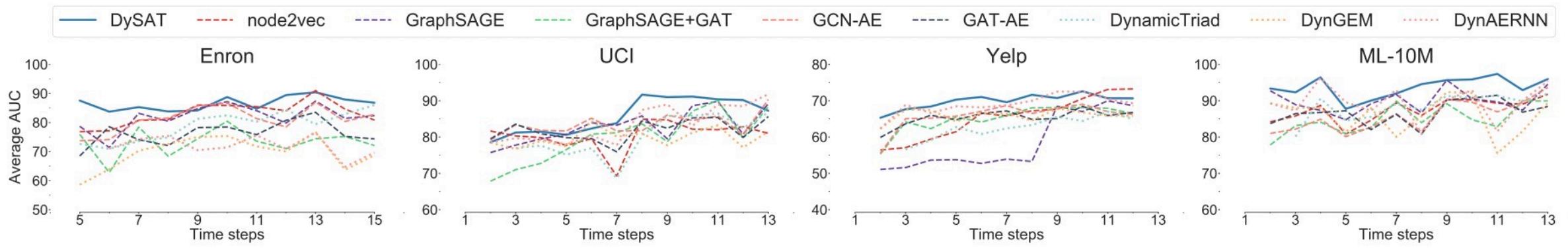


Figure 2: Performance comparison of DySAT with different models across multiple time steps: the solid line represents DySAT; dashed lines represent static graph embedding models; and dotted lines represent dynamic graph embedding models. We truncate the y-axis to avoid visual clutter.

DySAT和其他模型在多个时间步的性能比较

对比中，我们观察到在某些时间步长处静态嵌入方法的性能急剧下降。



pytorch

## 节点特征

```
process.py vis.digraph.allEdges.json vis.graph nodeList.json
The file size (3.88 MB) exceeds configured limit (2.56 MB). Code insight features are not available.

81 {  
82     "_id" : ObjectId("55110f618022bf7180ac70e9"),  
83     "name" : "rod.hayslett@enron.com_stanley.horton@enron.com",  
84     "time" : ISODate("2000-11-30T06:49:00.000+0000")  
85 }  
86 {  
87     "_id" : ObjectId("55110f618022bf7180ac70ea"),  
88     "name" : "tracy.geaccone@enron.com_rod.hayslett@enron.com",  
89     "time" : ISODate("2000-12-04T14:25:00.000+0000")  
90 }  
91 {  
92     "_id" : ObjectId("55110f618022bf7180ac70eb"),  
93     "name" : "rod.hayslett@enron.com_stanley.horton@enron.com",  
94     "time" : ISODate("2000-12-12T07:32:00.000+0000")  
95 }
```

## 边的关系

```
process.py vis.digraph.allEdges.json vis.graph nodeList.json
1 {  
2     "_id" : ObjectId("55098b62251497209062421f"),  
3     "name" : "albert.meyers@enron.com",  
4     "idx" : NumberInt(0)  
5 }
```

```

onehot = np.identity(slice_links[max(slice_links.keys())].number_of_nodes())
graphs = []  graphs:
for id, slice in slice_links.items():
    tmp_feature = []  tmp_feature: <class 'list': [array([1., 0., 0., 0., 0
    for node in slice.nodes():
        tmp_feature.append(onehot[node])
    slice.graph["feature"] = csr_matrix(tmp_feature) # 稀疏矩阵; 添加图中特征
    graphs.append(slice) # 将图保存在list中

```

最终的结果



0时刻所有节点特征

0时刻的所有节点信息

In [41]: graphs[0].nodes

Out [41]: NodeView((0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17))

0时刻的所有边信息，连接的边和时间

In [42]: graphs[0].edges

Out [42]: MultiEdgeView([(0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3), (0, 1, 4), (0, 1, 5)

时间点是采用快照的方式分割

In [39]: graphs[0].graph['feature'].A

Out [39]:

```

array([[1., 0., 0., ..., 0., 0., 0.],
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 1., ..., 0., 0., 0.],
       ...,
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]])
```

```
# Load training context pairs (or compute them if necessary)
context_pairs_train = get_context_pairs(graphs, num_time_steps)
```

导入上下文的随机游走序列

<https://blog.csdn.net/haolexiao/article/details/65157026>

## Alias Sampling Method

问题：比如一个随机事件包含四种情况，每种情况发生的概率分别为：

$1/2, 1/3, 1/12, 1/12$

问怎么用产生符合这个概率的采样方法？

均匀分布生成其他分布的方法，这种方法就是产生0~1之间的一个随机数。

比如：

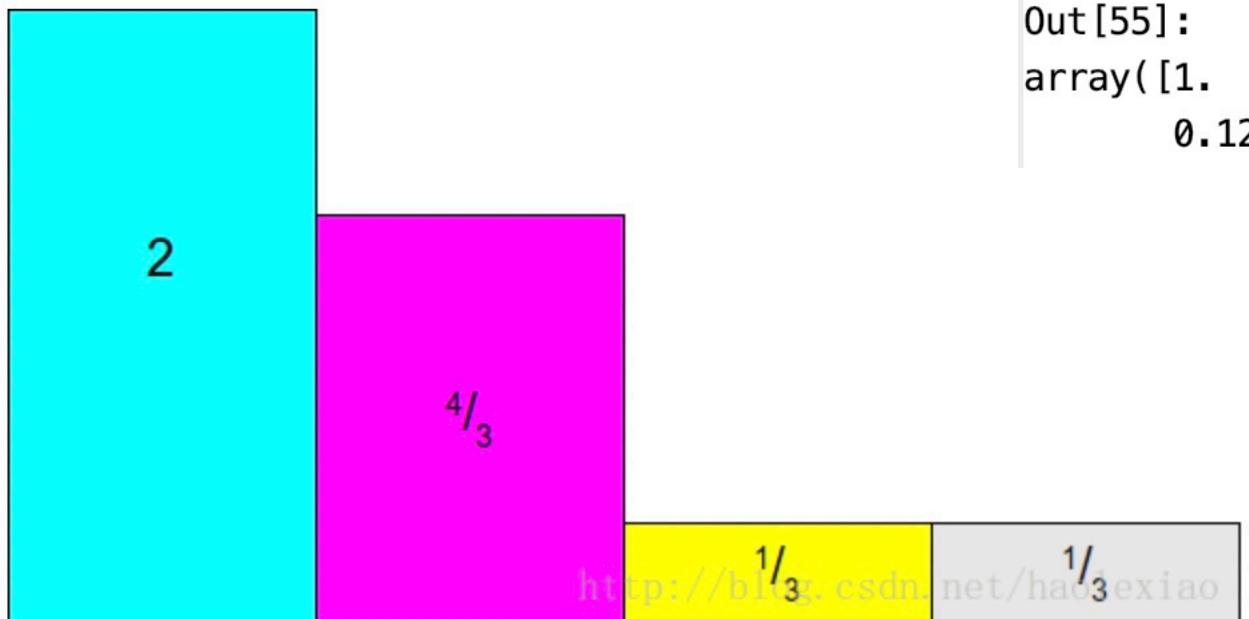
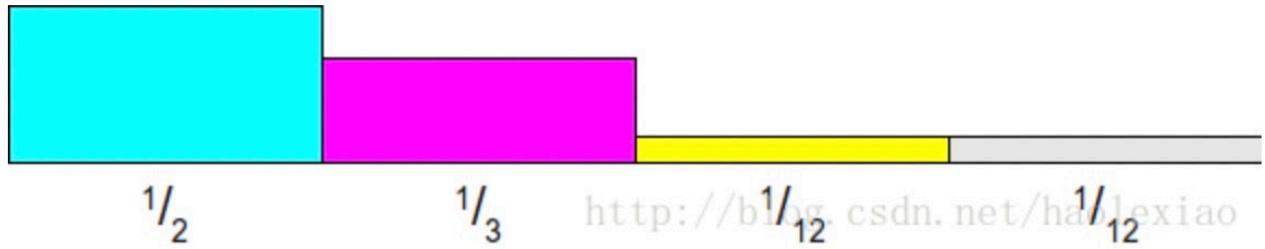
落在 $0 \sim 1/2$ 之间就是事件A

落在 $1/2 \sim 5/6$ 之间就是事件B

落在 $5/6 \sim 11/12$ 之间就是事件C

落在 $11/12 \sim 1$ 之间就是事件D

## Alias Method



```
In[58]: normalized_probs  
Out[58]:  
[0.34615384615384615,  
 0.01282051282051282,  
 0.0641025641025641,  
 0.02564102564102564,  
 0.05128205128205128,  
 0.01282051282051282,  
 0.2564102564102564,  
 0.02564102564102564,  
 0.01282051282051282,  
 0.19230769230769232]
```

```
In[56]: sorted(G.neighbors(node))
```

```
Out[56]: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

对这10个节点采样

```
In[54]: J
```

```
Out[54]: array([0, 0, 0, 0, 0, 6, 0, 9, 9, 6])
```

```
In[55]: q
```

```
Out[55]:  
array([1.          , 0.12820513, 0.64102564, 0.25641026, 0.51282051,  
      0.12820513, 1.          , 0.25641026, 0.12820513, 0.30769231])
```

1. 产生0~9之间的随机数 (2)
2. 产生0~1之间的随机数 (0.8)
3. 决策 ( $0.8 > 0.64$ , 则选择  $J[2]=0$ )  
(否则, 则选择3)

## Node2vec采样

一定游走到的是dst节点的邻居节点

```

for dst_nbr in sorted(G.neighbors(dst)): # dst节点的邻居节点 dst_nbr: 3
    if dst_nbr == src:
        unnormalized_probs.append(G[dst][dst_nbr]['weight']/p) # node2vec向回走概率
    elif G.has_edge(dst_nbr, src):
        unnormalized_probs.append(G[dst][dst_nbr]['weight'])
    else:
        unnormalized_probs.append(G[dst][dst_nbr]['weight']/q)

```

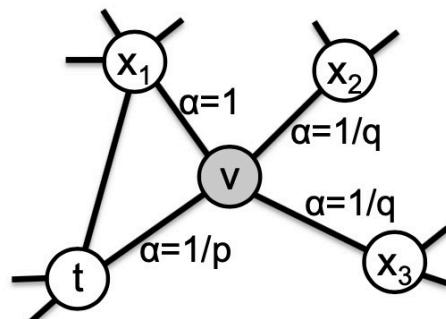


Figure 2: Illustration of the random walk procedure in node2vec. The walk just transitioned from  $t$  to  $v$  and is now evaluating its next step out of node  $v$ . Edge labels indicate search biases  $\alpha$ .

abilities  $\pi_{vx}$  on edges  $(v, x)$  leading from  $v$ . We set the unnormalized transition probability to  $\pi_{vx} = \alpha_{pq}(t, x) \cdot w_{vx}$ , where

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

按照sorted( $G.neighbors(dst)$ )

```
return alias_setup(normalized_probs)
```

In [41]: alias\_setup(normalized\_probs)

Out [41]:

```
(array([0, 0, 0, 0, 0]),
 array([1.          , 0.57142857, 0.14285714, 0.28571429, 0.14285714]))
```

产生两个随机数，第一个产生1~N之间的整数i，决定落在哪一列。扔第二次骰子，0~1之间的任意数，判断其与Prab[i]大小，如果小于Prab[i]，则采样i，如果大于Prab[i]，则采样Alias[i]

```
def preprocess_transition_probs(self):  
    self.alias_nodes = alias_nodes # 节点到下个节点的概率（第一次游走使用）  
    self.alias_edges = alias_edges # node2vec采样概率
```

## 划分测试集

```
def get_evaluation_data(graphs):
    """ Load train/val/test examples to evaluate link prediction performance"""
    eval_idx = len(graphs) - 2
    eval_graph = graphs[eval_idx] # 测试集图
    next_graph = graphs[eval_idx+1] # 测试集下一个图
    print("Generating eval data ....")
    train_edges, train_edges_false, val_edges, val_edges_false, test_edges, test_edges_false = \
        create_data_splits(eval_graph, next_graph, val_mask_fraction=0.2,
                           test_mask_fraction=0.6)

    return train_edges, train_edges_false, val_edges, val_edges_false, test_edges, test_edges_false
```

上一个时刻包含这些节点，才能考虑下一时刻是否有连接

```
def create_data_splits(graph, next_graph, val_mask_fraction=0.2, test_mask_fraction=0.6): graph: M
    edges_next = np.array(list(nx.Graph(next_graph).edges())) # 只保留一次边 edges_next: [[ 1  3]
    edges_positive = [] # Constraint to restrict new links to existing nodes. edges_positive: <c
    for e in edges_next: e: [1 3]                                上一时刻图中是否有该节点
        if graph.has_node(e[0]) and graph.has_node(e[1]): # 上张图中是否有该节点
            edges_positive.append(e) # positive的边
    edges_positive = np.array(edges_positive) # [E, 2]
    edges_negative = negative_sample(edges_positive, graph.number_of_nodes(), next_graph) # 负采样
```

## 负采样

```
def negative_sample(edges_pos, nodes_num, next_graph):  edges_pos: [[ 1  3]\n [ 1 101]\n [\nedges_neg = []\n    while len(edges_neg) < len(edges_pos): # 采样和positive同等数量的边\n        idx_i = np.random.randint(0, nodes_num) # 随机选择i,j节点\n        idx_j = np.random.randint(0, nodes_num)\n        if idx_i == idx_j: # 自连接\n            continue\n        if next_graph.has_edge(idx_i, idx_j) or next_graph.has_edge(idx_j, idx_i): # pos的边\n            continue\n        if edges_neg:\n            if [idx_i, idx_j] in edges_neg or [idx_j, idx_i] in edges_neg: # 存在之前的数据\n                continue\n        edges_neg.append([idx_i, idx_j])\n    return edges_neg
```

```
# 划分训练集，测试集，验证集
train_edges_pos, test_pos, train_edges_neg, test_neg = train_test_split(edges_positive,
    edges_negative, test_size=val_mask_fraction+test_mask_fraction)
val_edges_pos, test_edges_pos, val_edges_neg, test_edges_neg = train_test_split(test_pos,
    test_neg, test_size=test_mask_fraction/(test_mask_fraction+val_mask_fraction))

return train_edges_pos, train_edges_neg, val_edges_pos, val_edges_neg, test_edges_pos, test_edges_neg
```

训练，验证，测试都是预测最后一张图

只是对最后一张图中边进行了处理：如果上张图中没有该节点，则不计算这条边的label

```
dataset = MyDataset(args, graphs, feats, adjs, context_pairs_train)
```

构建dataset

```
class MyDataset(Dataset):
    def __init__(self, args, graphs, features, adjs, context_pairs): self: <utils.minibatch.MyDataset object>
        super(MyDataset, self).__init__()
        self.args = args
        self.graphs = graphs # 所有时刻的图
        self.features = [self._preprocess_features(feat) for feat in features] # 16个图和16个图中的特征
        self.adjs = [self._normalize_graph_gcn(a) for a in adjs] # 邻居矩阵归一化操作
        self.time_steps = args.time_steps
        self.context_pairs = context_pairs # 随机游走序列
        self.max_positive = args.neg_sample_size # 负采样数量
        self.train_nodes = list(self.graphs[self.time_steps-1].nodes()) # all nodes in the graph. 所有的节点
        self.min_t = max(self.time_steps - self.args.window - 1, 0) if args.window > 0 else 0 # 最小时间步
        self.degs = self.construct_degs() # 计算每个时间步图中节点的度
    self.pyg_graphs = self._build_pyg_graphs()
    self.__createitems__()
```



遍历所有节点在每个时间步中的上下文训练语料

For node in all node: 针对单个节点

```
node_1_list = [torch.LongTensor(node) for node in node_1_all_time]
node_2_list = [torch.LongTensor(node) for node in node_2_all_time]
```

节点对应上下文的节点

```
node_2_negative = [] node_2_negative: <class 'list'>: [[array([14, 3, 0, 7, 13, 8, 9, 10, 11, 4])
```

```
for t in range(len(node_2_list)): # 节点负采样
```

```
degree = self.degs[t] # 该时刻每个节点的degree degree: <class 'list'>: [156, 70, 2, 36, 10, 32, 7, 6]
```

```
node_positive = node_2_list[t][:, None] # pos的节点 node_positive: tensor([[ 1], [ 7], [ 8], [ 9], [ 5], [ 4], [ 3], [ 2], [ 6], [ 0]])
```

```
node_negative = fixed_unigram_candidate_sampler(true_clasees=node_positive, node_negative: <class '
```

`num_true=1,`

```
num_sampled=self.args.neg_sample_size, # 负采样数量
```

`unique=False,`

`distortion=0.75,`

unigrams=degree)

```
node_2_negative.append(node_negative)
```

```
feed_dict['node_1']=node_1_list # 该节点  
feed_dict['node_2']=node_2_list # 该节点的上下文正样本节点  
feed_dict['node_2_neg']=node_2_neg_list # 该节点负采样节点  
feed_dict["graphs"] = self.pyg_graphs # 图的信息
```

```
self.data_items[node] = feed_dict # 节点对应到的所有信息
```

16个时间步

In [96]: node\_1\_all\_time

In [98]: node\_2\_all\_time

```
Out[98]: [[10, 1, 10, 7, 3, 7, 7, 5, 16, 7],  
          [3, 1, 5, 1, 8, 3, 1, 1, 19, 1],  
          [7, 7, 3, 10, 7, 3, 7, 3, 7, 1],  
          [31, 5, 10, 7, 1, 7, 31, 1, 10, 3],  
          [10, 7, 7, 7, 1, 10, 47, 10, 12],  
          [10, 7, 1, 7, 18, 1, 10, 7, 35, 1],  
          [76, 7, 1, 1, 10, 1, 7, 7, 1, 1],  
          [26, 7, 5, 18, 7, 1, 7, 1, 15, 10],  
          [1, 1, 1, 7, 78, 8, 1, 1, 7, 1],  
          [7, 72, 36, 1, 2, 85, 101, 7, 1],  
          [1, 7, 101, 18, 7, 130, 101, 9, 9],  
          [8, 85, 29, 72, 39, 7, 1, 18, 1],  
          [127, 54, 54, 20, 7, 7, 20, 127],  
          [],  
          []],  
          []]
```

```
In[109]: node_2_neg_list
Out[109]:
[tensor([[ 8,  0,  5,  1,  7, 16, 11, 13,  3, 14],
       [ 0,  4, 10,  7,  5, 13, 11,  3, 14, 12],
       [ 0,  7,  8,  5, 13,  1, 14,  4,  9, 11],
       [ 0, 13,  3,  1,  6, 10, 11,  4,  5, 12],
       [10, 11,  0, 14,  7,  5,  1,  8, 13, 16],
       [ 8,  6,  3,  0, 10,  5,  1, 13, 14, 15],
       [10,  3,  0,  1, 17, 13,  4,  5, 12,  2],
       [ 0,  6, 10, 13,  4,  7, 16,  3,  1, 14],
       [ 1,  0, 10,  5, 13,  4,  8, 11,  7, 21],
       [ 1, 10,  0,  8,  9,  6,  5, 14, 14, 11]]),
 tensor([[ 0,  7, 13,  6,  9,  1, 14, 21, 10,  5],
       [ 3,  0,  7,  6, 10,  5,  8, 19, 11, 13],
       [ 0,  9, 10,  1,  3,  7,  6, 11,  8, 13],
       [ 0,  7,  5,  3, 18, 11, 10,  8,  6, 13],
       [ 3, 11, 10,  5,  6,  0,  1,  7, 13, 19],
       [ 1,  0,  7,  5, 11, 16, 19, 14,  8],
       [10,  0,  7,  3,  8,  5, 11, 19, 6, 21],
       [ 7, 18,  0, 19, 11,  5,  3,  6,  9, 10],
       [10,  1,  0,  5,  3,  6, 18,  7, 22, 11],
       [ 7, 11,  0,  5, 10,  3,  6, 19,  8, 20]]),
 tensor([[13,  4,  0,  1,  3,  8,  9, 10,  5, 11],
       [ 0,  3,  1, 23, 10,  8, 13,  5,  9, 18],
```

## 模型定义

```
model = DySAT(args, feats[0].shape[1], args.time_steps).to(device)          结构层信息

self.structural_head_config = list(map(int, args.structural_head_config.split(","))) # [16, 8, 8] 结构多头信息
self.structural_layer_config = list(map(int, args.structural_layer_config.split(",,"))) # 结构layer层信息
self.temporal_head_config = list(map(int, args.temporal_head_config.split(",,"))) # [16] 时序多头信息
self.temporal_layer_config = list(map(int, args.temporal_layer_config.split(",,"))) # [128] 时序layer层信息
self.spatial_drop = args.spatial_drop # 定义dropout
self.temporal_drop = args.temporal_drop                                     时序层信息

self.structural_attn, self.temporal_attn = self.build_model()

self.bceloss = BCEWithLogitsLoss() # 定义loss函数; sigmoid和crossentropy节点在一起
```

```

def build_model(self): self: DySAT()
    input_dim = self.num_features  input_dim: 143

# 1: Structural Attention Layers
structural_attention_layers = nn.Sequential()
for i in range(len(self.structural_layer_config)): # [128]
    layer = StructuralAttentionLayer(input_dim=input_dim, # features
                                      output_dim=self.structural_layer_config[i], # output维度
                                      n_heads=self.structural_head_config[i], # 多头参数
                                      attn_drop=self.spatial_drop, # drop参数
                                      ffd_drop=self.spatial_drop,
                                      residual=self.args.residual) # 残差连接
    structural_attention_layers.add_module(name="structural_layer_{}".format(i), module=layer)
    input_dim = self.structural_layer_config[i] # 下一层, StructuralAttention输入维度

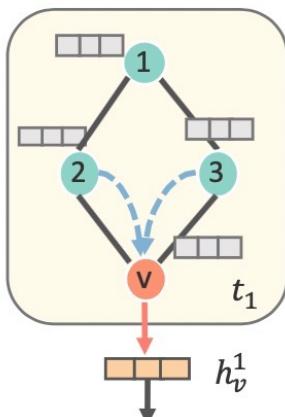
layer = StructuralAttentionLayer(input_dim=input_dim, # features
                                 output_dim=self.structural_layer_config[i], # output维度
                                 n_heads=self.structural_head_config[i], # 多头参数
                                 attn_drop=self.spatial_drop, # drop参数
                                 ffd_drop=self.spatial_drop,
                                 residual=self.args.residual) # 残差连接

```

结构attention, 就是GAT

$t_1$

Structural Self-Attention



## # 2: Temporal Attention Layers

```
input_dim = self.structural_layer_config[-1] # 时序attention的输入维度
temporal_attention_layers = nn.Sequential() temporal_attention_layers: Sequential()
for i in range(len(self.temporal_layer_config)): # [128]
    layer = TemporalAttentionLayer(input_dim=input_dim, # 输入维度
                                    n_heads=self.temporal_head_config[i], # 多头数量
                                    num_time_steps=self.num_time_steps, # 时间维度
                                    attn_drop=self.temporal_drop, # dropout
                                    residual=self.args.residual) # 残差连接
    temporal_attention_layers.add_module(name="temporal_layer_{}".format(i), module=layer)
    input_dim = self.temporal_layer_config[i]
```

# define weights

```
self.position_embeddings = nn.Parameter(torch.Tensor(num_time_steps, input_dim)) # 位置embedding信息[16, 128]
self.Q_embedding_weights = nn.Parameter(torch.Tensor(input_dim, input_dim)) # [128, 128]
self.K_embedding_weights = nn.Parameter(torch.Tensor(input_dim, input_dim))
self.V_embedding_weights = nn.Parameter(torch.Tensor(input_dim, input_dim))

# ff
self.lin = nn.Linear(input_dim, input_dim, bias=True)

# dropout
self.attn_dp = nn.Dropout(attn_drop)
self.xavier_init()
```

## @staticmethod

```
def collate_fn(samples): # 节点对应的所有信息; [143]: {"node_1", "node_2", "node_2_neg"} samples:  
    batch_dict = {} batch_dict:  
        for key in ["node_1", "node_2", "node_2_neg"]:  
            # node_1:节点本身; node_2:节点对应到的10个pos节点; node_2_neg:  
            data_list = [] data_list: <class 'list'>: []  
                for sample in samples: # 每一个节点的所有信息  
                    data_list.append(sample[key]) # 每个节点有样本数量; 每个节点在这一时刻的采样数量  
                concate = []  
                for t in range(len(data_list[0])):  
                    # 遍历每个时间步  
                    concate.append(torch.cat([data[t] for data in data_list])) # 第t个时间步中, key涉及到的所有节点  
                batch_dict[key] = concate  
        batch_dict["graphs"] = samples[0]["graphs"] # graph  
    return batch_dict # 每个类别下, 所有时间步中涉及到的节点 (16个时间步, 每个时间步中的节点flatten)
```

In[122]: samples[0]['node\_1']  
Out[122]:  
[tensor([], dtype=torch.int64),  
 tensor([], dtype=torch.int64),  
 tensor([137, 137, 137, 137, 137, 137, 137, 137, 137, 137]),  
 tensor([137, 137, 137, 137, 137, 137, 137, 137, 137, 137])]

16个时间步节点的信息

每个时间点涉及到的节点

```
In[139]: concate[0]  
Out[139]:  
tensor([ 1,  1,  1,  1,  1,  1,  1,  1,  1,  1,  6,  6,  6,  6,  6,  6,  6,  
       6,  6,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  8,  11, 11, 11, 11, 11, 11,  
      11, 11, 11, 11, 2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  2,  16, 16, 16, 16,  
     16, 16, 16, 16, 16, 16, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 9,  9,  
      9,  9,  9,  9,  9,  9,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  4,  
     10, 10, 10, 10, 10, 10, 10, 10, 10, 10, 3,  3,  3,  3,  3,  3,  3,  3,  3,  3,  
      3,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  12, 12, 12, 12, 12, 12, 12,  
     12, 12, 12, 12, 13, 13, 13, 13, 13, 13, 13, 13, 13, 17, 17, 17, 17, 17,  
    17, 17, 17, 17, 17, 5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  5,  15, 15,  
   15, 15, 15, 15, 15, 15, 7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7,  7])  
In[140]: len(concat)  
Out[140]: 16
```

每个samples都存储同一张图

```
In[141]: batch_dict.keys()  
Out[141]: dict_keys(['node_1', 'node_2', 'node_2_neg', 'graphs'])
```

# GNN

```
def get_loss(self, feed_dict):
    node_1, node_2, node_2_negative, graphs = feed_dict.values()
    # run gnn
    final_emb = self.forward(graphs) # [N, T, F]

class DySAT(nn.Module):
    def forward(self, graphs):
        # Structural Attention forward
        structural_out = [] structural_out: <class 'list'>: []
        for t in range(0, self.num_time_steps): t: 0
            structural_out.append(self.structural_attn(graphs[t]))
        structural_outputs = [g.x[:,None,:] for g in structural_out]
```

```
class StructuralAttentionLayer(nn.Module)
```

```
In[26]: self.structural_attn
Out[3]: Sequential(
    (structural_layer_0): StructuralAttentionLayer(
        (act): ELU(alpha=1.0)
        (lin): Linear(in_features=143, out_features=128, bias=False)
        (leaky_relu): LeakyReLU(negative_slope=0.2)
        (attn_drop): Dropout(p=0.1, inplace=False)
        (ffd_drop): Dropout(p=0.1, inplace=False)
        (lin_residual): Linear(in_features=143, out_features=128, bias=False)
    )
)
```

```
In[145]: graphs[0]
Out[113]: Data(edge_index=[2, 66], edge_weight=[66], x=[18, 143])
```

```
↳ class StructuralAttentionLayer(nn.Module)
```

```
    def forward(self, graph): self: StructuralAttentionLayer\n        (act):\n            graph = copy.deepcopy(graph)\n            edge_index = graph.edge_index # 点边关系 edge_index: tensor([[ 0\n            edge_weight = graph.edge_weight.reshape(-1, 1) edge_weight: tens\n            H, C = self.n_heads, self.out_dim H: 16 C: 8\n            x = self.lin(graph.x).view(-1, H, C) # [N, heads, out_dim]; [18,\n            # attention [18, 16, 8] [18, 16] [18, 143]*[143, 128] => [18,128] => [18,16,8] # 多头attention
```

```
                alpha_l = (x * self.att_l).sum(dim=-1).squeeze() # [N, heads]\n                alpha_r = (x * self.att_r).sum(dim=-1).squeeze()\n                alpha_l = alpha_l[edge_index[0]] # [num_edges, heads]\n                alpha_r = alpha_r[edge_index[1]]\n                alpha = alpha_r + alpha_l\n                alpha = edge_weight * alpha\n                alpha = self.leaky_relu(alpha)\n\n            # output [66, 16, 8] [66, 16, 1]
```

[18, 16, 8]

```
out = self.act(scatter(x_j * coefficients[:, :, None], edge_index[1], dim=0, reduce="sum"))
```

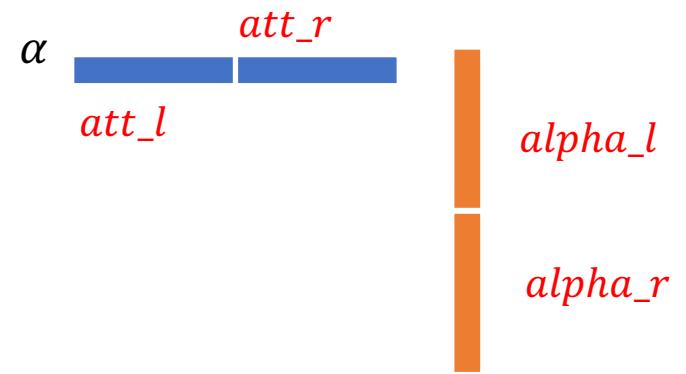
```
out = out.reshape(-1, self.n_heads*self.out_dim) #[num_nodes, output_dim]
```

```
if self.residual:
```

```
    out = out + self.lin_residual(graph.x)
```

```
graph.x = out
```

```
return graph
```



In [43]: `x.shape`  
Out [20]: `torch.Size([18, 16, 8])`

In [42]: `self.att_l.shape`  
Out [19]: `torch.Size([1, 16, 8])`

求 $edge\_index[1]$ 周围邻居和

untitled2 ~/PycharmProjects/untitled2

```

4 alpha = torch.rand((10,5)) # 10个节点, 每个节点有5个head的attention
5 print(alpha)
6 edge_index = torch.tensor([0,1,1,2,2,2,3,3,3,3])
7 print(edge_index)
8
9 print(softmax(alpha, edge_index))

```



```

tensor([[0.3620, 0.6795, 0.4662, 0.7349, 0.9914],
       A [0.5452, 0.5178, 0.7889, 0.0757, 0.9854],
       B [0.8811, 0.5943, 0.8514, 0.0385, 0.3697],
       [0.7825, 0.3775, 0.9791, 0.5111, 0.1358],
       [0.4978, 0.4130, 0.2163, 0.6876, 0.2109],
       [0.6792, 0.3634, 0.2454, 0.1711, 0.8779],
       [0.2810, 0.9360, 0.6311, 0.5542, 0.2882],
       [0.7734, 0.3906, 0.1659, 0.7208, 0.6619],
       [0.9378, 0.9902, 0.1037, 0.2065, 0.6852],
       [0.8492, 0.3223, 0.9007, 0.8533, 0.9663]])
tensor([0, 1, 1, 2, 2, 2, 3, 3, 3, 3])
tensor([[1.0000, 1.0000, 1.0000, 1.0000, 1.0000],
       [0.4168, 0.4809, 0.4844, 0.5093, 0.6492],
       [0.5832, 0.5191, 0.5156, 0.4907, 0.3508],
       [0.3768, 0.3309, 0.5137, 0.3443, 0.2393],
       [0.2834, 0.3429, 0.2396, 0.4107, 0.2580],
       [0.3398, 0.3262, 0.2467, 0.2450, 0.5027],
       [0.1580, 0.3148, 0.2835, 0.2361, 0.1692],
       [0.2585, 0.1825, 0.1780, 0.2789, 0.2458],
       [0.3047, 0.3323, 0.1673, 0.1667, 0.2516],
       [0.2788, 0.1704, 0.3712, 0.3184, 0.3333]])
grad_fn=<DifferentiableGraphBackward>

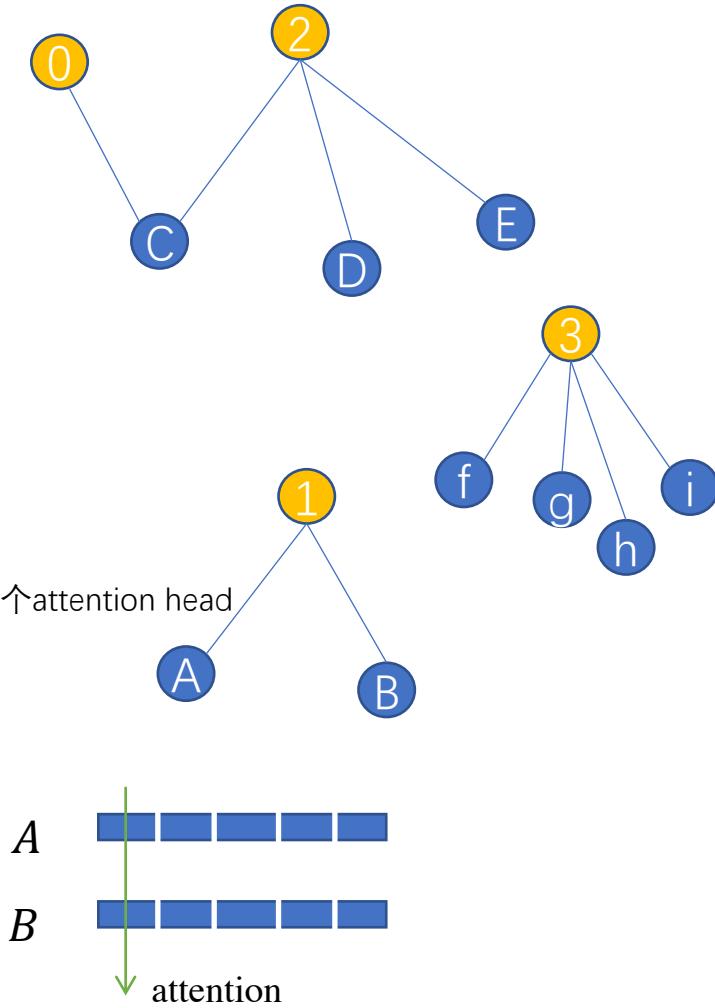
```

节点索引

10个节点和{0,1,2,3}节点连接  
每个连接有5个attention head

该节点每个attention的特征

对应到的节点的attention



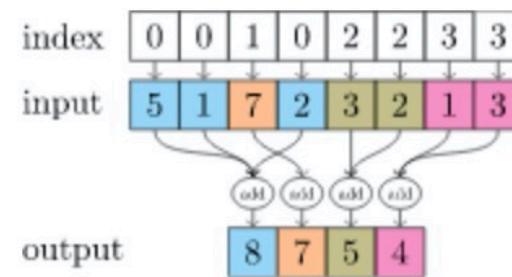
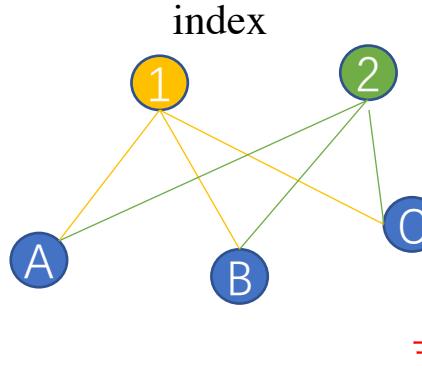
$[c, a, b, c, d, e, f, g, h, i]$

$[0, 1, 1, 2, 2, 2, 3, 3, 3, 3]$

scatter方法通过src和index两个张量来获得一个新的张量。

```
1 | torch_scatter.scatter(src: torch.Tensor, index: torch.Tensor, dim: int = -1, out: Optional[torch.Tensor])
```

原理如图，根据index，将index相同值对应的src元素进行对应定义的计算，dim为在第几维进行相应的运算。e.g.scatter\_sum即进行sum运算，scatter\_mean即进行mean运算。



16个head，每个head包含8个特征  
[18, 16, 8]

```
out = self.act(scatter(x_j * coefficients[:, :, None], edge_index[1], dim=0, reduce="sum"))
```

节点按照attention系数求和

```
out = out.reshape(-1, self.n_heads*self.out_dim) #[num_nodes, output_dim]
```

index

$$\mathbf{z}_v = \sigma \left( \sum_{u \in \mathcal{N}_v} \alpha_{uv} \mathbf{W}^s \mathbf{x}_u \right)$$

Attention后输出的特征维度[18, 128]

## 时序注意力层

将结构注意力结果作为输入

```
# Temporal Attention forward
```

```
temporal_out = self.temporal_attn(structural_outputs_padded)
```

```
class TemporalAttentionLayer(nn.Module)
```

```
# 1: Add position embeddings to input
```

```
position_inputs = torch.arange(0, self.num_time_steps).reshape(1, -1).repeat(inputs.shape[0], 1).long().to(inputs.device)
```

```
temporal_inputs = inputs + self.position_embeddings[position_inputs] # [N, T, F]
```

[16, 128]

16个时间，每个时间的embedding

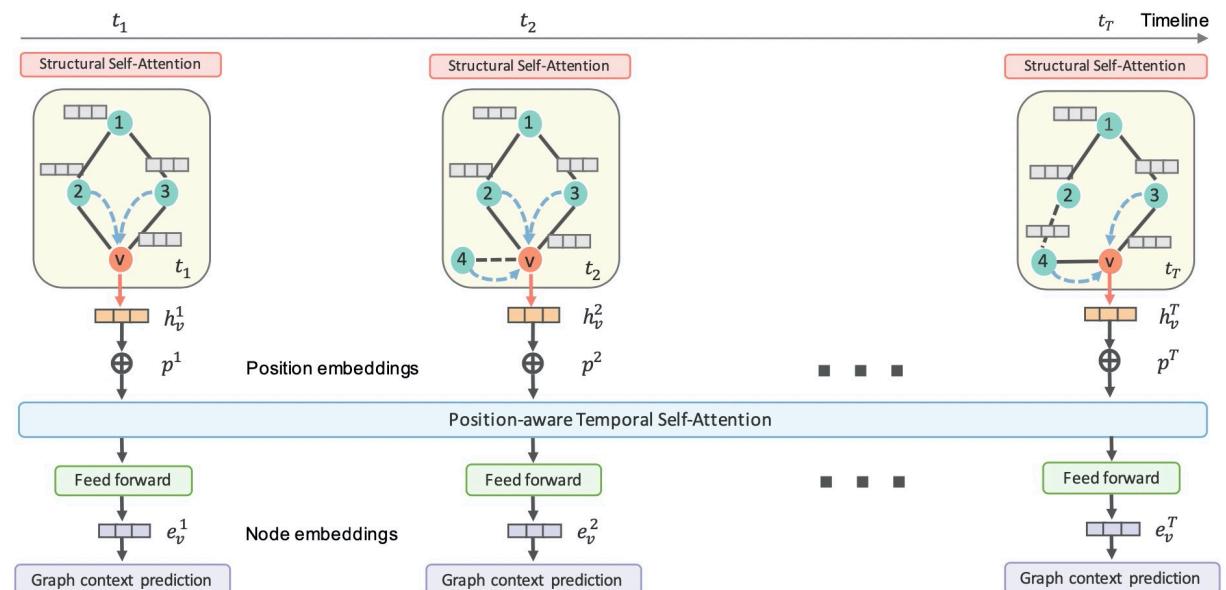
[143, 16, 128]: 143个节点，在每个时间对应到的embedding(128维)

每个节点有16个时间步

[143, 16]

每个时间步对应到的embedding

结构注意力层输出的结果，每个节点在16个时刻对应到embedding(128维)



In[153]: position\_inputs

```
Out[130]:
tensor([[ 0,  1,  2, ..., 13, 14, 15],
        [ 0,  1,  2, ..., 13, 14, 15],
        [ 0,  1,  2, ..., 13, 14, 15],
        ...,
        [ 0,  1,  2, ..., 13, 14, 15],
        [ 0,  1,  2, ..., 13, 14, 15],
        [ 0,  1,  2, ..., 13, 14, 15]])
```

$$X \times W^Q = Q$$

[143, 16, 128]

# 2: Query, Key based multi-head self attention. [128, 128]

```
q = torch.tensordot(temporal_inputs, self.Q_embedding_weights, dims=([2], [0])) # [N, T, F]
k = torch.tensordot(temporal_inputs, self.K_embedding_weights, dims=([2], [0])) # [N, T, F]
v = torch.tensordot(temporal_inputs, self.V_embedding_weights, dims=([2], [0])) # [N, T, F]
```

# 3: Split, concat and scale. [143, 16, 128]

16个 [143, 16, 8]

```
split_size = int(q.shape[-1]/self.n_heads) # 最后一维的维度 split_size: 8
q_ = torch.cat(torch.split(q, split_size_or_sections=split_size, dim=2), dim=0) # [hN, T, F/h]
k_ = torch.cat(torch.split(k, split_size_or_sections=split_size, dim=2), dim=0) # [hN, T, F/h]
v_ = torch.cat(torch.split(v, split_size_or_sections=split_size, dim=2), dim=0) # [hN, T, F/h]
```

[2288, 16, 8]

Q\*K

```
outputs = torch.matmul(q_, k_.permute(0,2,1)) # [hN, T, T]
outputs = outputs / (self.num_time_steps ** 0.5)
```

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

16个时间步，每一步和16个的attention系数

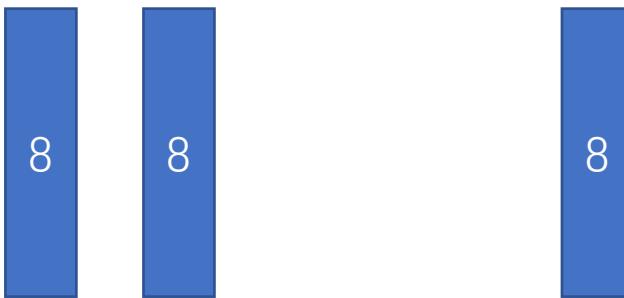
$$\text{softmax}\left(\frac{\text{Q} \times \text{K}^T}{\sqrt{d_k}}\right) \text{V} = \text{Z}$$

$Q = [143, 16, 128]$

split

time feature  
[2288, 16, 8]

16个头，每个头有8个特征



Q K V



8

$$2288 = 143 * 16$$

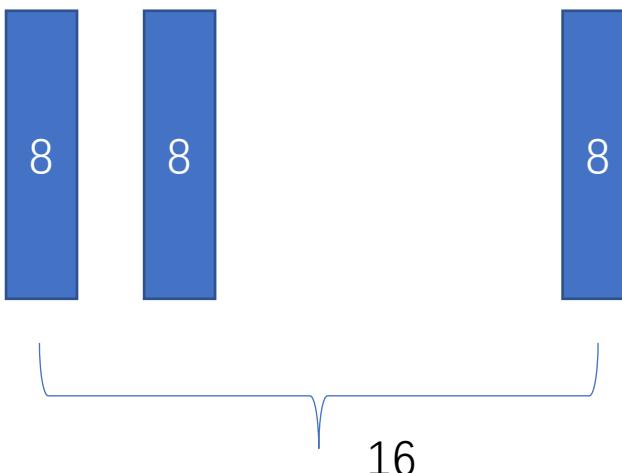
```

# 4: Masked (causal) softmax to compute attention weights.
diag_val = torch.ones_like(outputs[0]) # [16,16]的全1向量 diag_val: Loading timed out
tril = torch.tril(diag_val) # 下三角阵 tril: Loading timed out
masks = tril[None, :, :].repeat(outputs.shape[0], 1, 1) # [h*N, T, T] 重复N次 (2288) ; [2288, 16, 16]
padding = torch.ones_like(masks) * (-2**32+1) # 负无穷 padding: Loading timed out [2288, 16, 16]
outputs = torch.where(masks==0, padding, outputs) # outputs中mask为0的地方, 填充padding中负无穷的数值
outputs = F.softmax(outputs, dim=2)
self.attn_wts_all = outputs # [h*N, T, T]

```

每个节点有16个时间步

目的是将之前没有出现的时间步, 设置为0;  
只计算看到过的时间



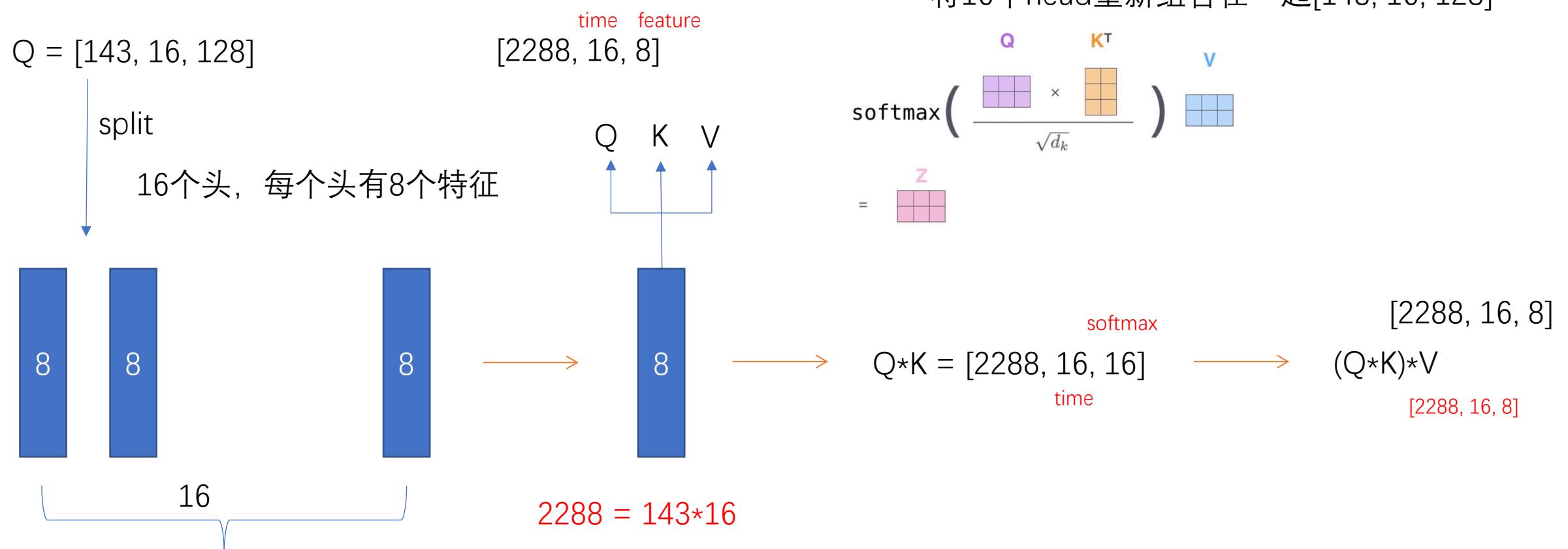
$$e_v^{ij} = \left( \frac{((\mathbf{X}_v \mathbf{W}_q)(\mathbf{X}_v \mathbf{W}_k)^T)_{ij}}{\sqrt{F'}} + M_{ij} \right)$$

Query      Key  
缩放因子

$$M_{ij} = \begin{cases} 0, & i \leq j \\ -\infty, & \text{otherwise} \end{cases}$$

```
# 5: Dropout on attention weights.
```

```
if self.training:  
    outputs = self.attn_dp(outputs) # dropout [2288, 16, 16]  
outputs = torch.matmul(outputs, v_) # [hN, T, F/h] # (K*Q)*V; output-经过归一化后的attention系数[2288, 16, 16]  
outputs = torch.cat(torch.split(outputs, split_size_or_sections=int(outputs.shape[0]/self.n_heads), dim=0), dim=2) # [N, T, F]
```



# 6: Feedforward and residual

```
outputs = self.feedforward(outputs)
```

$$L_v = \sum_{t=1}^T \sum_{u \in \mathcal{N}_{walk}^t(v)} -\log(\sigma(\langle \mathbf{e}_u^t, \mathbf{e}_v^t \rangle)) - w_n \cdot \sum_{u' \in P_n^t(v)} \log(1 - \sigma(\langle \mathbf{e}_{u'}^t, \mathbf{e}_v^t \rangle)) \quad (5)$$

$\langle . \rangle$  denotes the inner product operation.



# **EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs**

DySAT，是基于GNN和RNN的组合：

在每个快照中使用GNN作为特征提取器聚合节点特征，然后在每个快照中使用RNN聚合时间特征来进一步聚合节点的特征。

EvolveGCN：

也是离散型动态GNN，是集成型的DGNN，将GNN和RNN结合在同一层，从而结合空间和时间信息的建模

# 动态图网络算法

## 离散网络

- 1) Stacked DGNNs (堆叠DGNN) 单独的GNN处理图的每个快照，并将每个GNN输出到RNN模型中
- 2) Integrated DGNNs (集成DGNN) 将GNN和RNN结合在同一层，从而结合空间和时间信息的建模

## 连续网络

- 1) 基于RNN的方法
- 2) 基于点过程的方法

Model type	Model name	Encoder	Link addition	Link deletion	Node addition	Node deletion	Network type
Discrete networks							
Stacked DGNN	GCRN-M1 [58]	Spectral GCN [59] & LSTM	Yes	Yes	No	No	Any
	WD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	CD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	RgCNN [61]	Spatial GCN [62] & LSTM	Yes	Yes	No	No	Any
	DyGGNN [63]	GGNN [64] & LSTM	Yes	Yes	No	No	Any
	DySAT [67]	GAT [68] & temporal attention from [69]	Yes	Yes	Yes	Yes	Any
Integrated DGNN	GCRN-M2 [58]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	GC-LSTM [72]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	EvolveGCN [71]	LSTM integrated in a GCN [57]	Yes	Yes	Yes	Yes	Any
	LRGCN [73]	R-GCN [75] integrated in an LSTM	Yes	Yes	No	No	Any
	RE-Net [74]	R-GCN [75] integrated in several RNNs	Yes	Yes	No	No	Knowledge network
Continuous networks							
RNN based							
	Streaming GNN [86]	Node embeddings maintained by architecture consisting of T-LSTM [88]	Yes	No	Yes	No	Directed strictly evolving
	JODIE [87]	Node embeddings maintained by an RNN based architecture	Yes	No	No	No	Bipartite and interaction
TPP based							
	Know-Evolve [89]	TPP parameterised by an RNN	Yes	No	No	No	Interaction, knowledge network
	DyREP [36]	TPP parameterised by an RNN aided by structural attention	Yes	No	Yes	No	Strictly evolving
	LDG [90]	TPP, RNN and self-attention	Yes	No	Yes	No	Strictly evolving
	GHN [92]	TPP parameterised by a continuous time LSTM [93]	Yes	No	No	No	Interaction, knowledge network

# 1. 构图

时间	graph	time step	
2021.5.1	a-b; b-c;	1	
	c-e; b-c;	1	
	b-f;	1	
2021.6.1	a-b; b-d;	2	
	c-e;	2	
2021.7.5	f-g;	3	
	g-a;	3	

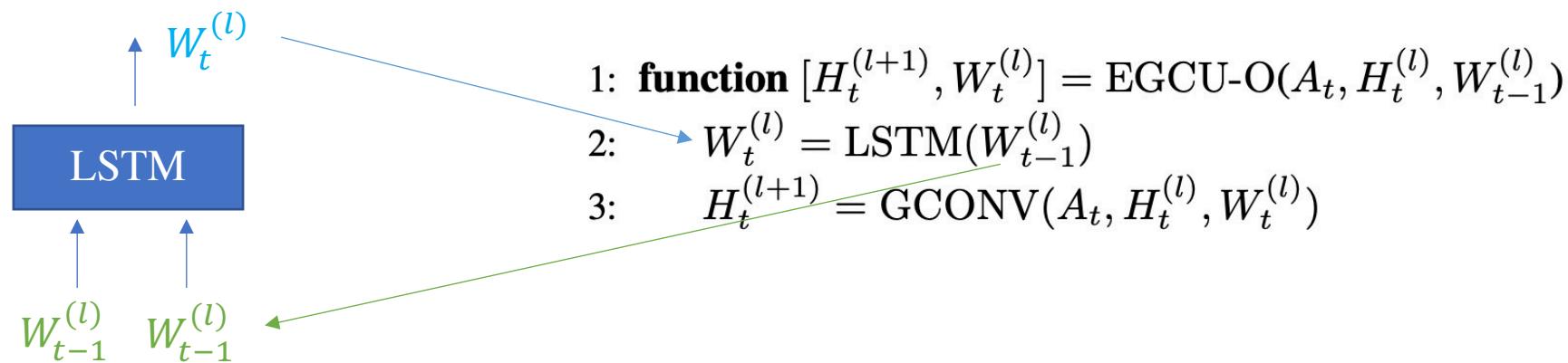
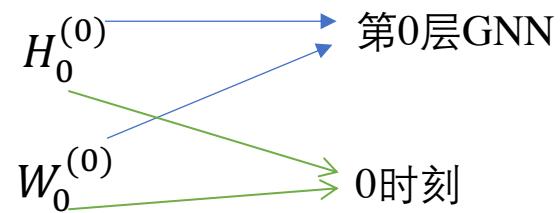
GCN

GCN中W是可学习的参数  
在EvolveGCN中， W是通过RNN学习

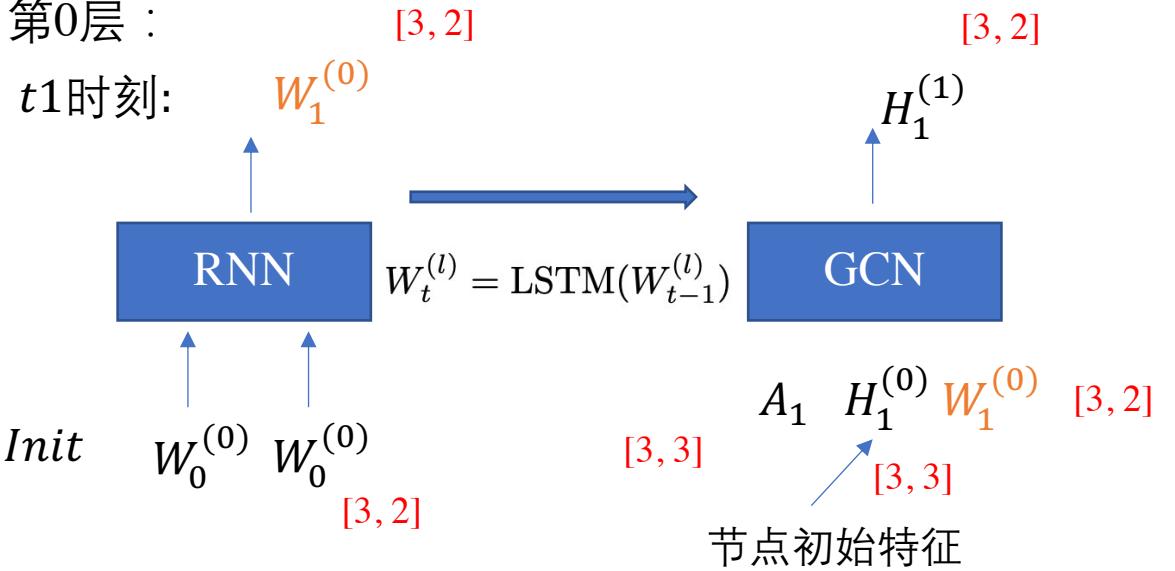
$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} H^{(l)} W^{(l)}\right)$$

$$H^{(l+1)} = \sigma(\hat{A} H^{(l)} W^l)$$

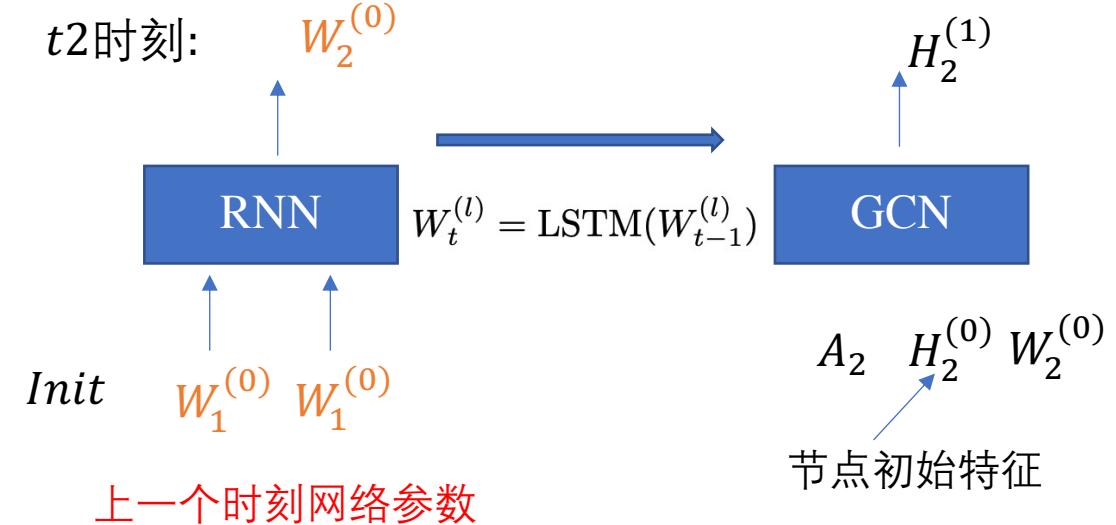
$$\hat{A}_t, H_t^{(l)}, W_{t-1}^{(l)} \Rightarrow H_t^{(l+1)}$$



第0层：



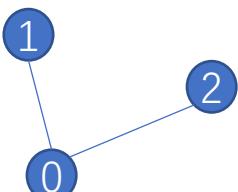
第0层：



0	[0.1, 0.1, 0.2]	t1
1	[0.3, 0.3, 0.1]	t1
2	[0.2, 0.2, 0.2]	t1

初始化

$$W_0^{(0)} = \begin{matrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{matrix}$$



t1

$$H_1^0 = \begin{matrix} 0.1 & 0.1 & 0.2 \\ 0.3 & 0.3 & 0.1 \\ 0.2 & 0.2 & 0.2 \end{matrix}$$

$$A_1 = \begin{matrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 1 & 1 & 0 \end{matrix}$$

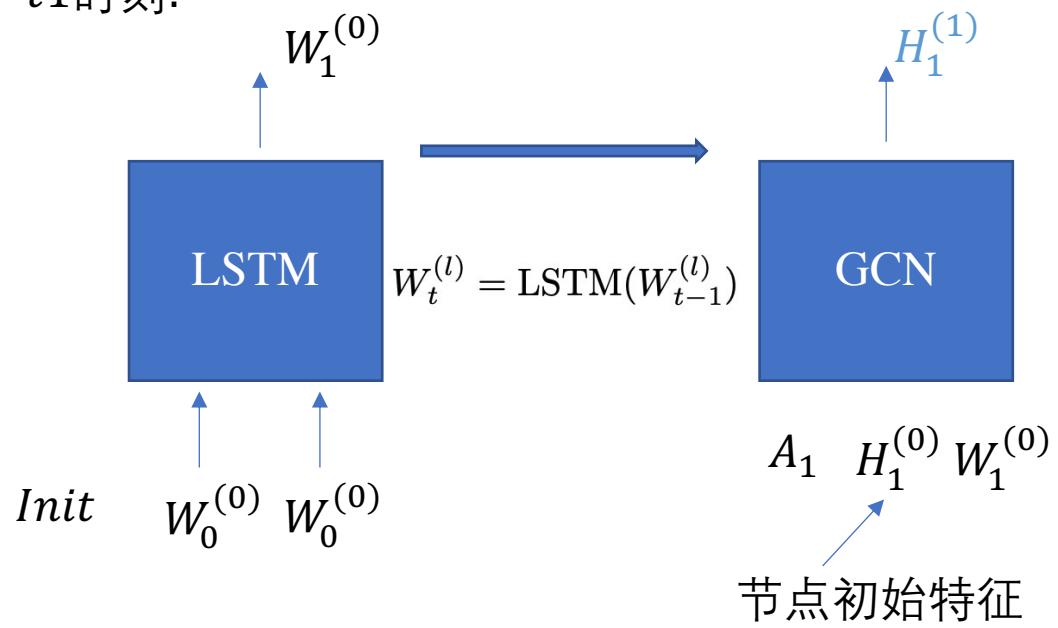
- 1: **function**  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-O}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$
- 2:  $W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$
- 3:  $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$

$$W_0^{(0)} \xrightarrow{\text{RNN}} W_1^{(0)} = \begin{matrix} 0 & 1 \\ 0 & 0 \\ 1 & 2 \end{matrix}$$

$$H_0^{(1)} = A_1 H_1^0 W_1^{(0)} = \begin{matrix} x & x \\ x & x \\ x & x \end{matrix}$$

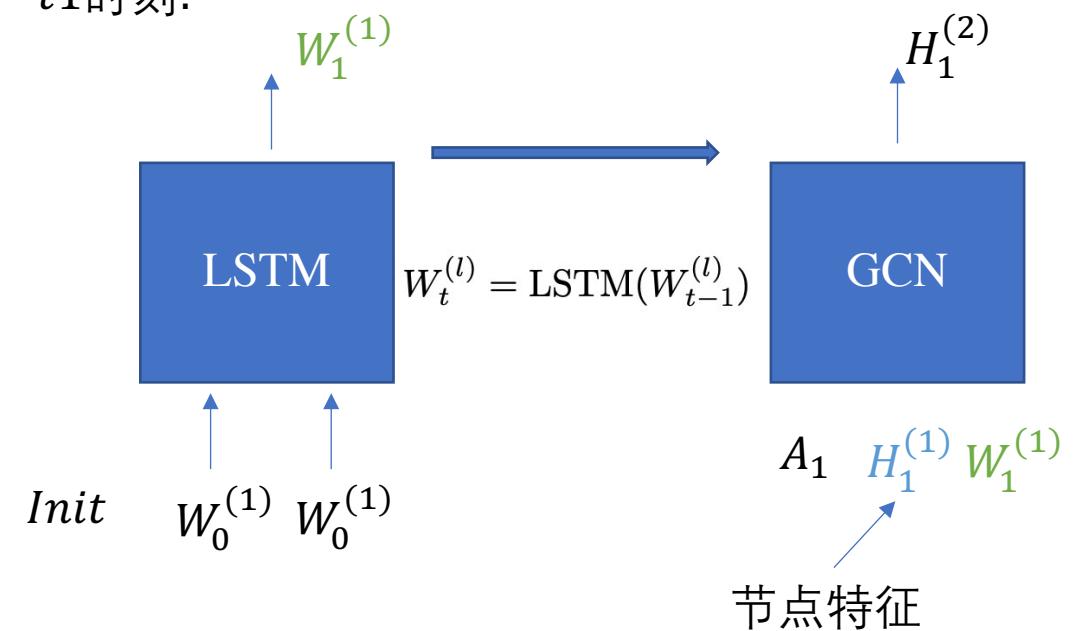
第0层：

$t1$ 时刻:

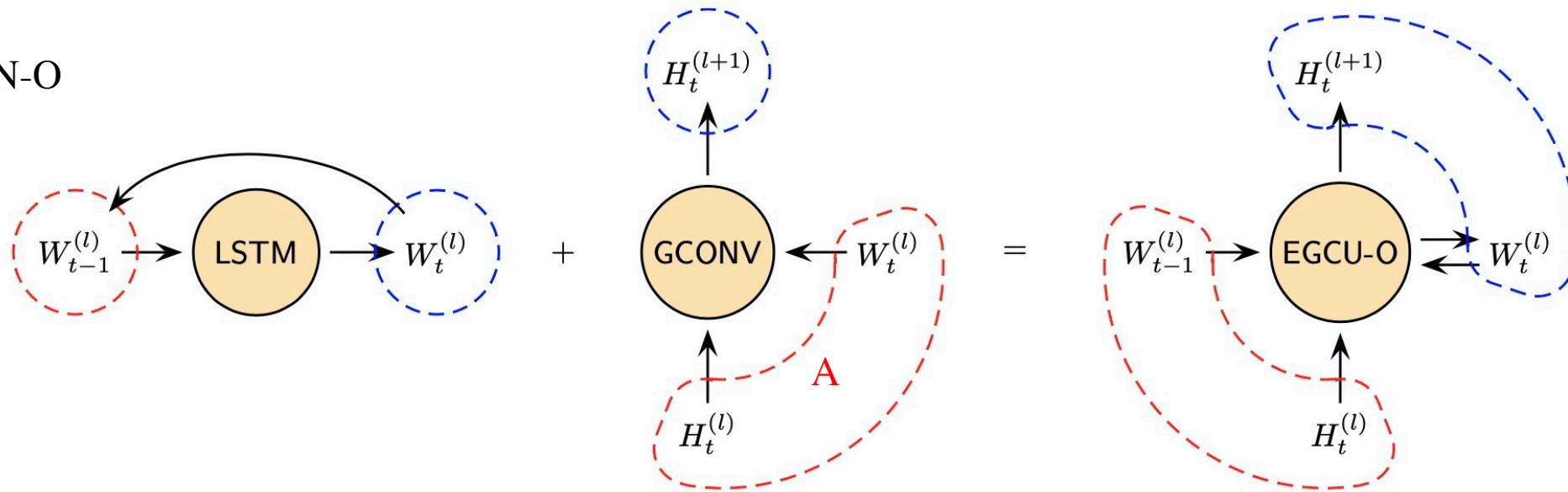


第1层：

$t1$ 时刻:

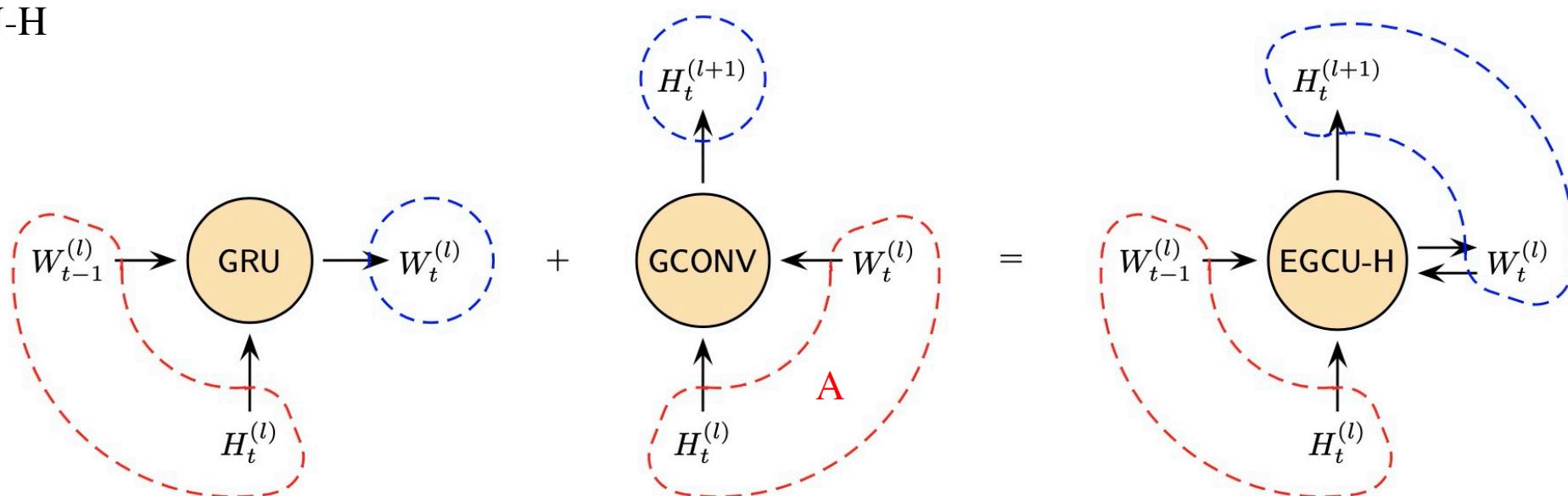


## EvolveGCN-O



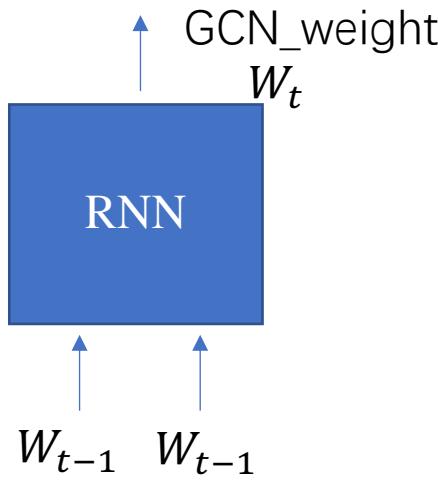
(b) EvolveGCN-O, where the GCN parameters are input/outputs of a recurrent architecture.

## EvolveGCN-H



(a) EvolveGCN-H, where the GCN parameters are hidden states of a recurrent architecture that takes node embeddings as input.

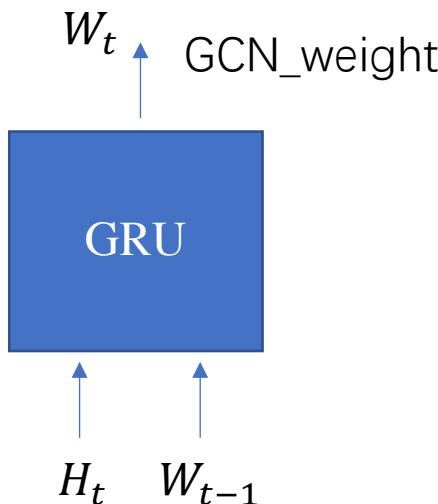
-O



```
1: function  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-O}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$ 
2:  $W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$ 
3:  $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$ 
```

$$W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$$

-H

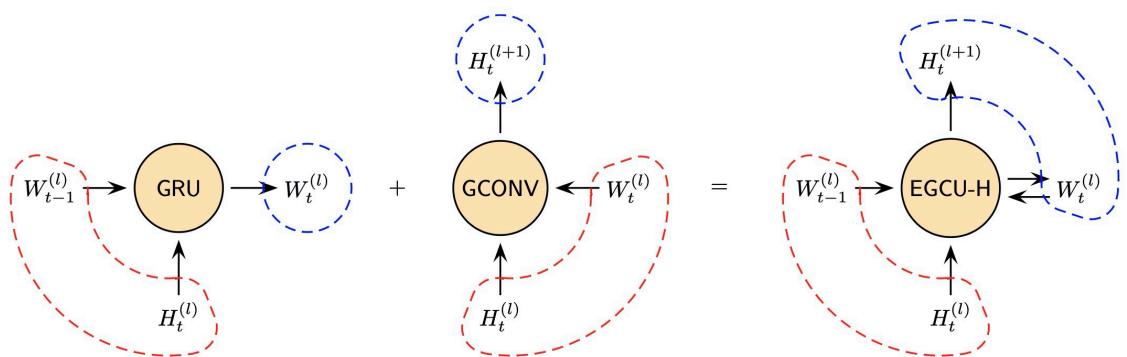


```
1: function  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-H}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$ 
2:  $W_t^{(l)} = \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)})$ 
3:  $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$ 
```

在计算GCN参数W时候， 加上了节点的特征

节点的特征

$$W_t^{(l)} = \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)})$$



(a) EvolveGCN-H, where the GCN parameters are hidden states of a recurrent architecture that takes node embeddings as input.

```

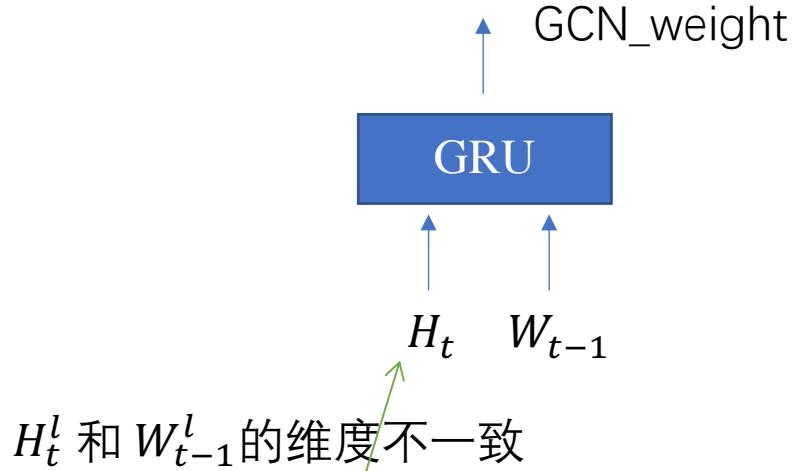
1: function  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-H}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$ 
2:    $W_t^{(l)} = \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)})$ 
3:    $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$ 
4: end function

```

```

1: function  $Z_t = \text{summarize}(X_t, k)$ 
2:    $y_t = X_t p / \|p\|$  对  $H_t^l$  进行一个降维
3:    $i_t = \text{top-indices}(y_t, k)$ 
4:    $Z_t = [X_t \circ \tanh(y_t)]_{i_t}$ 
5: end function

```

$$\begin{array}{c} X_t = [1000, 162] \\ \downarrow \text{降维} \\ Z_t = [100, 162] \\ \downarrow \text{reverse} \\ Z_t = [162, 100] \end{array}$$


$$\begin{aligned} H_t^l &= [1000, 162] \\ W_{t-1}^l &= [162, 100] \end{aligned}$$

两种EvolveGCN该如何选择：

如果节点有信息， -H效果要好， 因为考虑了节点的特征变化

如果节点信息比较少， -O效果好， 因为他更关系图结构的变化

Loss

在每个时间步，有连接的边构建正样本，负样本采用负采样方式获得

$$L_v = \sum_{t=1}^T \sum_{u \in \mathcal{N}_{walk}^t(v)} -\log(\sigma(\langle \mathbf{e}_u^t, \mathbf{e}_v^t \rangle)) - w_n \cdot \sum_{u' \in P_n^t(v)} \log(1 - \sigma(\langle \mathbf{e}_{u'}^t, \mathbf{e}_v^t \rangle))$$

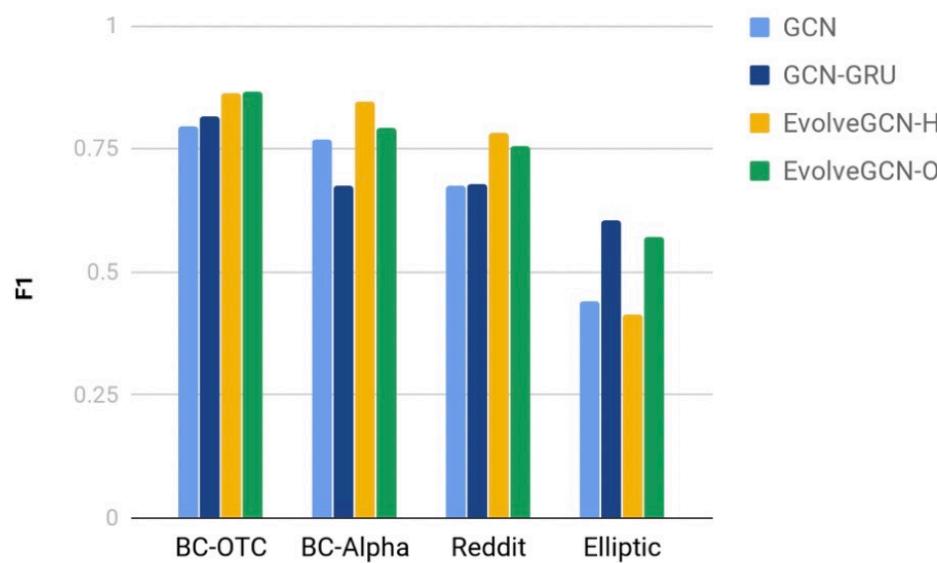
*positive*节点的内积                                    *negative*节点的内积

negative sampling ratio

针对每一个时刻内

Table 2: Performance of link prediction. Each column is one data set.

	mean average precision					mean reciprocal rank				
	SBM	BC-OTC	BC-Alpha	UCI	AS	SBM	BC-OTC	BC-Alpha	UCI	AS
GCN	0.1987	0.0003	0.0003	0.0251	0.0003	0.0138	0.0025	0.0031	0.1141	0.0555
GCN-GRU	0.1898	0.0001	0.0001	0.0114	0.0713	0.0119	0.0003	0.0004	0.0985	0.3388
DynGEM	0.1680	0.0134	0.0525	0.0209	0.0529	0.0139	0.0921	0.1287	0.1055	0.1028
dyngraph2vecAE	0.0983	0.0090	0.0507	0.0044	0.0331	0.0079	0.0916	0.1478	0.0540	0.0698
dyngraph2vecAERNN	0.1593	<b>0.0220</b>	<b>0.1100</b>	0.0205	0.0711	0.0120	<b>0.1268</b>	<b>0.1945</b>	0.0713	0.0493
EvolveGCN-H	0.1947	0.0026	0.0049	0.0126	<b>0.1534</b>	<b>0.0141</b>	0.0690	0.1104	0.0899	<b>0.3632</b>
EvolveGCN-O	<b>0.1989</b>	0.0028	0.0036	<b>0.0270</b>	0.1139	0.0138	0.0968	0.1185	<b>0.1379</b>	0.2746



Code

```

# Assign the requested random hyper parameters; 设置args
args = build_random_hyper_params(args)

#build the dataset
dataset = build_dataset(args) # 构建数据集

#build the tasker
tasker = build_tasker(args,dataset) # 预测任务

#build the splitter
splitter = sp.splitter(args,tasker) # 训练, 测试, 验证集

#build the models
gcn = build_gcn(args, tasker)
classifier = build_classifier(args,tasker) # 输出结果分类器

#build a loss
cross_entropy = ce.Cross_Entropy(args,dataset).to(args.device)

#trainer
trainer = tr.Trainer(args,
                      splitter = splitter, # 训练, 测试, 验证集
                      gcn = gcn,
                      classifier = classifier,
                      comp_loss = cross_entropy,
                      dataset = dataset,
                      num_classes = tasker.num_classes)

trainer.train()

```

设置参数

```

In[31]: dataset.edges
Out[31]:
{'idx': tensor([[ 0,  2,  0],
               [ 0,  3,  0],
               [ 0,  8,  0],
               ....,
               [999, 959, 49],
               [999, 970, 49],
               [999, 991, 49]]), 'vals': tensor([1, 1, 1, ..., 1, 1, 1])}

In[33]: dataset.nodes_feats
Out[33]:
tensor([[0.0290, 0.4019, 0.2598],
       [0.3666, 0.0583, 0.7006],
       [0.0518, 0.4681, 0.6738],
       ...])

```

```

>>> classifier
Classifier(
    (mlp): Sequential(
        (0): Linear(in_features=200, out_features=100, bias=True)
        (1): ReLU()
        (2): Linear(in_features=100, out_features=2, bias=True)
    )
)

```

```
tasker = build_tasker(args,dataset) # 预测任务  
  
if args.task == 'link_pred':  
    return lpt.Link_Pred_Tasker(args,dataset)  
  
def __init__(self,args,dataset):  
  
    self.get_node_feats = self.build_get_node_feats(args,dataset) # 构建节点特征  
  
    return get_node_feats  
  
    def get_node_feats(adj):  
        return tu.get_1_hot_deg_feats(adj,  
                                      max_deg,  
                                      dataset.num_nodes)  
  
self.prepare_node_feats = self.build_prepare_node_feats(args,dataset)  
  
return prepare_node_feats  
  
def prepare_node_feats(node_feats):  
    return u.sparse_prepare_tensor(node_feats,  
                                   torch_size= [dataset.num_nodes,  
                                               self.feats_per_node])
```

= 162 节点max out degree

self.feats\_per\_node = 162 节点特征数量

>>> tasker.feats\_per\_node 162 邻接矩阵，计算节点度

>>> tasker.get\_node\_feats(adj)

>>> tasker.prepare\_node\_feats(node\_feats) 节点特征，预处理

```
splitter = sp.splitter(args,tasker) # 训练, 测试, 验证集
```

切分训练测试集

```
>>> len(splitter.train)  
29  
>>> len(splitter.test)  
10  
>>> len(splitter.dev)  
5
```



```
gcn = build_gcn(args, tasker)
```

```
egcn_o.EGCN(gcn_args, activation=torch.nn.RReLU(), device=args.device)
```

```
gcn_args = u.Namespace(args.gcn_parameters)  
gcn_args.feats_per_node = tasker.feats_per_node # 162
```

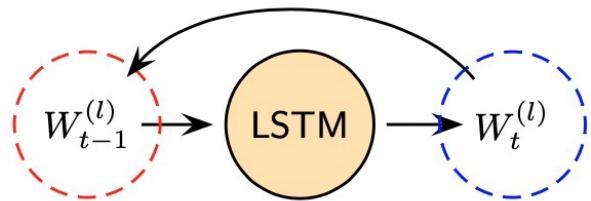
egcn\_o

```
class EGCN(torch.nn.Module):  
    def __init__(self, args, activation, device='cpu', skipfeats=False): self: EGCN()  
        super().__init__()  
        GRCU_args = u.Namespace({}) GRCU_args: <utils.Namespace object at 0x15b1074a8>  
  
        feats = [args.feats_per_node,  feats: <class 'list'>: [162, 100, 100]  
                 args.layer_1_feats,  
                 args.layer_2_feats]  
        self.device = device  
        self.skipfeats = skipfeats  
        self.GRCU_layers = []  
        self._parameters = nn.ParameterList()  
        for i in range(1, len(feats)):  
            GRCU_args = u.Namespace({'in_feats': feats[i-1],  
                                    'out_feats': feats[i],  
                                    'activation': activation})  
            grcu_i = GRCU(GRCU_args) → GRU  
            #print (i, 'grcu_i', grcu_i)  
            self.GRCU_layers.append(grcu_i.to(self.device))  
            self._parameters.extend(list(self.GRCU_layers[-1].parameters()))
```

两层GRCU

```
class GRCU(torch.nn.Module):
```

```
    self.evolve_weights = mat_GRU_cell(cell_args)
```



EvolveGCN

$$Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$$

$$R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$$

$$\tilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$$

```
class mat_GRU_cell(torch.nn.Module):  
    def __init__(self,args):  
        super().__init__()  
        self.args = args  
        self.update = mat_GRU_gate(args.rows,  
                                   args.cols,  
                                   torch.nn.Sigmoid())
```

```
    self.reset = mat_GRU_gate(args.rows,  
                             args.cols,  
                             torch.nn.Sigmoid())
```

```
    self.htilde = mat_GRU_gate(args.rows,  
                               args.cols,  
                               torch.nn.Tanh())
```

```
    self.choose_topk = TopK(feats = args.rows,  
                           k = args.cols)
```

```
class mat_GRU_gate(torch.nn.Module):
```

```
    def __init__(self,rows,cols,activation):  
        super().__init__()  
        self.activation = activation  
        #the k here should be in_feats which is actually the rows  
        self.W = Parameter(torch.Tensor(rows,rows)) # 162*162  
        self.reset_param(self.W)
```

```
    self.U = Parameter(torch.Tensor(rows,rows)) # 162*162  
    self.reset_param(self.U)
```

```
    self.bias = Parameter(torch.zeros(rows,cols)) # 162*100
```

```
    def forward(self,x,hidden):
```

```
        out = self.activation(self.W.matmul(x) + \  
                             self.U.matmul(hidden) + \  
                             self.bias)
```

```
        return out
```

$$W * X + U * \text{hidden} + b$$

这里 -O 方法也是使用了 GRU 函数

$$Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$$

$$R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$$

$$\tilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$$

$$Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$$

$$R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$$

$$\tilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$$

```

self.evolve_weights = mat_GRU_cell(cell_args)

def forward(self, prev_Q):#, prev_Z, mask):
    # z_topk = self.choose_topk(prev_Z, mask)
    z_topk = prev_Q # 上一层的; Xt

    update = self.update(z_topk, prev_Q) # Xt, Ht-1 => Zt Zt = sigmoid(WZ X_t + UZ H_{t-1} + B_Z)
    reset = self.reset(z_topk, prev_Q) # Xt, Ht-1 => Rt Rt = sigmoid(WR X_t + UR H_{t-1} + BR)

    h_cap = reset * prev_Q # Rt * Ht-1 R_t \circ H_{t-1}
    h_cap = self.htilde(z_topk, h_cap) # Ht_hat \tilde{H}_t = tanh(WH X_t + UH (R_t \circ H_{t-1}) + BH)

    new_Q = (1 - update) * prev_Q + update * h_cap # (1 - Zt) * Ht-1 + Zt * Ht_hat
                                                    H_t = (1 - Z_t) \circ H_{t-1} + Z_t \circ \tilde{H}_t

    return new_Q

```

$$\begin{aligned}
Z_t &= \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z) \\
R_t &= \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R) \\
\tilde{H}_t &= \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H) \\
H_t &= (1 - Z_t) \circ H_{t-1} + Z_t \circ \tilde{H}_t
\end{aligned}$$

## 数据集

```
classifier = build_classifier(args,tasker) # 输出结果分类器

# trainer
trainer = tr.Trainer(args,
    splitter = splitter, # 训练, 测试, 验证集
    gcn = gcn, # 模型
    classifier = classifier, # link_pred 分类器
    comp_loss = cross_entropy, # loss
    dataset = dataset, # dataset
    num_classes = tasker.num_classes) # 2分类

trainer.train()
```

**trainer.train()**

## forward

## Training data

```
eval_train, nodes_embs = self.run_epoch(self.splitter.train, e, 'TRAIN', grad = True)
```

## 训练集

```
for s in split:  
    if self.tasker.is_static:  
        s = self.prepare_static_sample(s)  
    else:  
        s = self.prepare_sample(s)
```

```
class data_split(Dataset):
    def __getitem__(self, idx):    self: <splitter.data_split object at 0x10725c3c8>  idx: 7
        idx = self.start + idx
        t = self.tasker.get_sample(idx, test = self.test, **self.kwargs)
        return t
```

link\_pred\_tasker

对应到tasker

★  `def get_sample(self, idx, test, **kwargs):`

```
>>> tasker.feats_per_node  
162  
>>> tasker.get_node_feats(adj)  
>>> tasker.prepare_node_feats(node_feats)
```



```
cur_adj = tu.normalize_adj(adj = cur_adj, num_nodes = self.data.num_nodes) # 节点邻接矩阵和节点度乘积**-0.5的值
```

$$\sqrt{degree_{in} * degree_{out}}$$

```
>>> cur_adj
{'idx': tensor([[ 0,  0],
               [ 0,  2],
               [ 0,  3],
               ...,
               [999, 974],
               [999, 991],
               [999, 999]]), 'vals': tensor([0.0086, 0.0083, 0.0084, ..., 0.0094, 0.0100, 0.0100])}
```

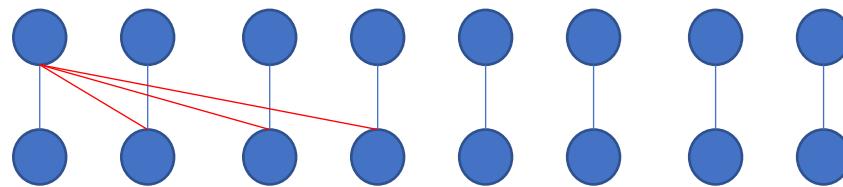
采样数量

False

```
def get_non_existing_edges(adj, number, tot_nodes, smart_sampling, existing_nodes=None):  
    num_edges = min(number, idx.shape[1] * (idx.shape[1]-1) - len(true_ids))
```

约定好的负采样数量

已经连接的数量



和其他节点相连接的数量

## 总结采样函数

```
def get_sample(self, idx, test, **kwargs):
```

```
return {'idx': idx, # 索引  
       'hist_adj_list': hist_adj_list, # 前6个时间图, 点边关系  
       'hist_ndFeats_list': hist_ndFeats_list, # 前6个时间图, 点的出度  
       'label_sp': label_adj, # 边的正负关系  
       'node_mask_list': hist_mask_list} # 节点mask值
```

```
>>> label_adj
{'idx': tensor([[ 0,    2],
               [ 0,    3],
               [ 0,    8],
               ...,
               [533, 981],
               [383, 738],
               [715, 156]]), 'vals': tensor([1, 1, 1, ..., 0, 0, 0])}
```

```
>>> hist_adj_list
[{'idx': tensor([[ 0,  0],
   [ 0,  2],
   [ 0,  3],
   ...,
   [999, 974],
   [999, 991],
   [999, 999]]), 'vals': tensor([0.0088, 0.0086, 0.0086, ..., 0.0087, 0.0092, 0.0093])},
```

```
>>> hist_ndFeats_list
[{'idx': tensor([[ 0, 113],
   [ 1, 104],
   [ 2, 118],
   ...,
   [997, 109],
   [998, 126],
   [999, 107]]), 'vals':
```

```
def prepare_node_feats(node_feats):    node_feats: <class 'dict'>: {'idx':  
    return u.sparse_prepare_tensor(node_feats,  
                                    torch_size= [dataset.num_nodes,  
                                                 self.feats_per_node])
```

节点度去第0个值，  
对值进行数值化  
并转换成稀疏矩阵

```
for s in split:  s: <utils.Namespace object at 0x102ceefd0>  
    if self.tasker.is_static:  
        s = self.prepare_static_sample(s)  
    else:  
        s = self.prepare_sample(s)
```

数据清洗

开始训练

```
predictions, nodes_embs = self.predict(s.hist_adj_list,  
                                         s.hist_ndFeats_list,  
                                         s.label_sp['idx'],  
                                         s.node_mask_list)
```

点边关系，入度出度\*\*-0.5

节点特征：度

label

mask

```
def predict(self,hist_adj_list,hist_ndFeats_list,node_indices,mask_list):  
    nodes_embs = self.gcn(hist_adj_list, # graph  
                           hist_ndFeats_list, # features  
                           mask_list) # mask
```

根据不同时间图中  
图信息（邻接矩阵）  
节点特征（度）

```
egcn_o.EGCN(gcn_args, activation= torch.nn.RReLU(), device= args.device)
```

forward

```
def forward(self,A_list, Nodes_list,nodes_mask_list):
```

```
    node_feats= Nodes_list[-1]
```

```
    for unit in self.GRCU_layers:
```

节点度特征

```
        Nodes_list = unit(A_list,Nodes_list),nodes_mask_list)
```

邻接矩阵

```
    out = Nodes_list[-1]
```

```
    if self.skipfeats:
```

```
        out = torch.cat((out,node_feats), dim=1) # use node_feats.to_dense() if 2hot encoded input
```

```
    return out
```

```
for i in range(1,len(feats)):
```

```
    GRCU_args = u.Namespace({'in_feats': feats[i-1],  
                            'out_feats': feats[i],  
                            'activation': activation})
```

GRU

```
    grcu_i = GRCU(GRCU_args)
```

```
    #print (i,'grcu_i', grcu_i)
```

```
    self.GRCU_layers.append(grcu_i.to(self.device))
```

GRCU

```
def forward(self,A_list,node_embs_list):#,mask_list):
```

```
    GCN_weights = self.GCN_init_weights
```

```
    out_seq = []
```

```
    for t,Ahat in enumerate(A_list): # 遍历每个时间步长矩阵
```

```
        node_embs = node_embs_list[t] # 节点特征; degree
```

#first evolve the weights from the initial and use the new weights with the node\_embs

```
        GCN_weights = self.evolve_weights(GCN_weights),node_embs,mask_list[t])
```

```
        node_embs = self.activation(Ahat.matmul(node_embs.matmul(GCN_weights)))
```

```
        out_seq.append(node_embs)
```

```
    return out_seq
```

```

for unit in self.GRCU_layers:    unit: GRCU(\n  (evolve_weights): mat_GRU_cell(\n
Nodes_list = unit(A_list,Nodes_list)##,nodes_mask_list) # 邻接矩阵; 节点度矩阵

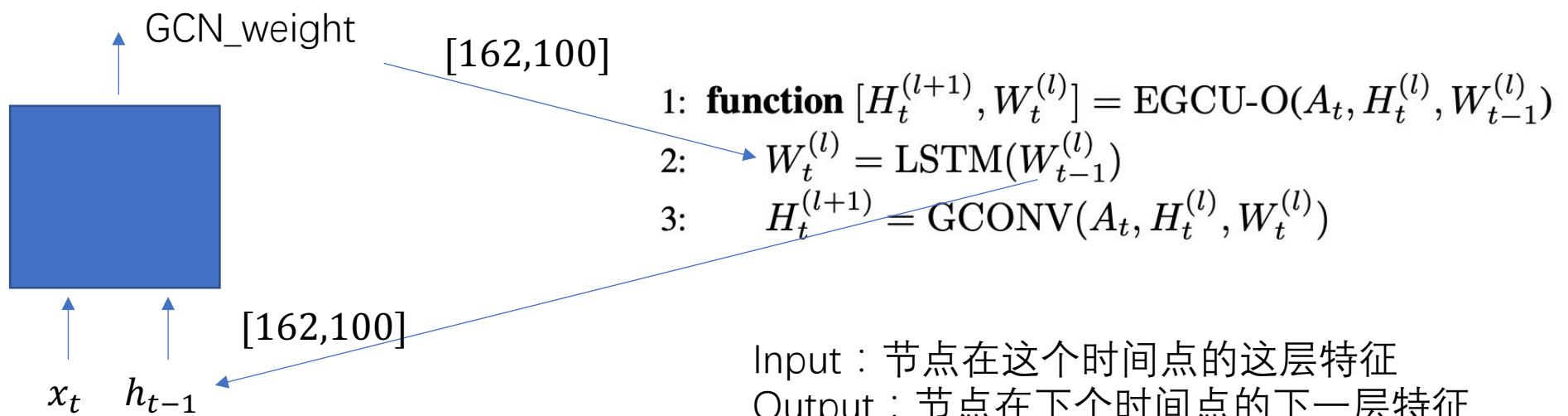
```

6个时间邻接矩阵 6个时间节点特征 (度)

```

def forward(self,A_list,node_embs_list):##,mask_list): self: GRCU(\n  (evolve_weights): mat_GRU_cell(\n    (up
GCN_weights = self.GCN_init_weights # 初始化的GCN_weight GCN_weights: Parameter containing:\ntensor([[[-0.
out_seq = [] out_seq: <class 'list'>: []
for t,Ahat in enumerate(A_list): # 遍历每个时间步长邻接矩阵 t: 0 Ahat: tensor(indices=tensor([[ 0,  0,
node_embs = node_embs_list[t] # 每个时间的节点特征; degree node_embs: tensor(indices=tensor([[ 0,  1
#first evolve the weights from the initial and use the new weights with the node_embs
GCN_weights = self.evolve_weights(GCN_weights) # -0, 不涉及节点特征; ,node_embs,mask_list[t])
node_embs = self.activation(Ahat.matmul(node_embs.matmul(GCN_weights))) # 学习到的下一层节点embedding

```



第0层：

该层下一时刻GCN\_weight

$W_1^0$  [162,100]



$$W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$$

[1000, 100]  $H_1^1$   
 $node\_out\_ebmdding_{t1}$

学习的是如何生成GCN的邻接矩阵

- 1: **function**  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-O}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$
- 2:  $W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$
- 3:  $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$

Init

$W_0^0$   $W_0^0$   
GCN\_weight

t1时刻:

$W_1^0$  [162,100]



[162,100]

$A_1$   $H_1^0$   $W_1^0$   
t1时刻节点初始特征

GCN\_weight

$W_0^0$  第0层  
0时刻

Init

$W_1^0$   $W_1^0$

$A_2$   $H_2^0$   $W_2^0$   
t2时刻节点初始特征

$H_2^1$   
 $node\_out\_ebmdding_{t2}$

## GRU

```

class mat_GRU_cell(torch.nn.Module):
    H_{t-1}
    def forward(self, prev_Q):#, prev_Z, mask):
        # z_topk = self.choose_topk(prev_Z, mask)
        z_topk = prev_Q # 上一层的; Xt

```

$$Z_t \text{ update} = \text{self.update}(z_{\text{topk}}, \text{prev\_Q}) \quad \# \text{ Xt}, H_{t-1} \Rightarrow Z_t \quad Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$$

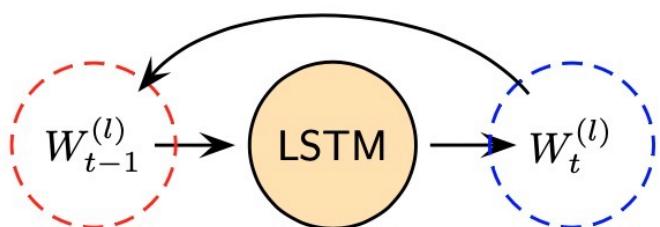
$$R_t \text{ reset} = \text{self.reset}(z_{\text{topk}}, \text{prev\_Q}) \quad \# \text{ Xt}, H_{t-1} \Rightarrow R_t \quad R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$$

$$h_{\text{cap}} = \text{reset} * \text{prev\_Q} \quad \# R_t * H_{t-1} \quad R_t \circ H_{t-1}$$

$$\tilde{H}_t \quad h_{\text{cap}} = \text{self.htilde}(z_{\text{topk}}, h_{\text{cap}}) \quad \# H_{t\_hat} \quad \tilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$$

$$H_t \quad \text{new\_Q} = (1 - \text{update}) * \text{prev\_Q} + \text{update} * h_{\text{cap}} \quad \# (1 - Z_t) * H_{t-1} + Z_t * H_{t\_hat}$$

$$H_t = (1 - Z_t) \circ H_{t-1} + Z_t \circ \tilde{H}_t$$



$$Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$$

$$R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$$

$$\tilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$$

$$H_t = (1 - Z_t) \circ H_{t-1} + Z_t \circ \tilde{H}_t$$

$$Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$$

$$R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$$

$$\tilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$$

```
class mat_GRU_gate(torch.nn.Module):
    def __init__(self, rows, cols, activation):
        super().__init__()
        self.activation = activation
        #the k here should be in_feats which is actually the rows
        self.W = Parameter(torch.Tensor(rows, rows)) # 162*162
        self.reset_param(self.W)

        self.U = Parameter(torch.Tensor(rows, rows)) # 162*162
        self.reset_param(self.U)

        self.bias = Parameter(torch.zeros(rows, cols)) # 162*100

    def forward(self, x, hidden):
        out = self.activation(self.W.matmul(x) + \
                             self.U.matmul(hidden) + \
                             self.bias)
        return out
```

$$W * X + U * \text{hidden} + b$$

该层下一时刻GCN\_weight

```
GCN_weights = self.evolve_weights(GCN_weights) #,node_embs,mask_list[t])  
node_embs = self.activation(Ahat.matmul(node_embs.matmul(GCN_weights)))
```

[1000, 100]

该时刻邻接矩阵 $A_t$

$$H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$$

[162, 100]

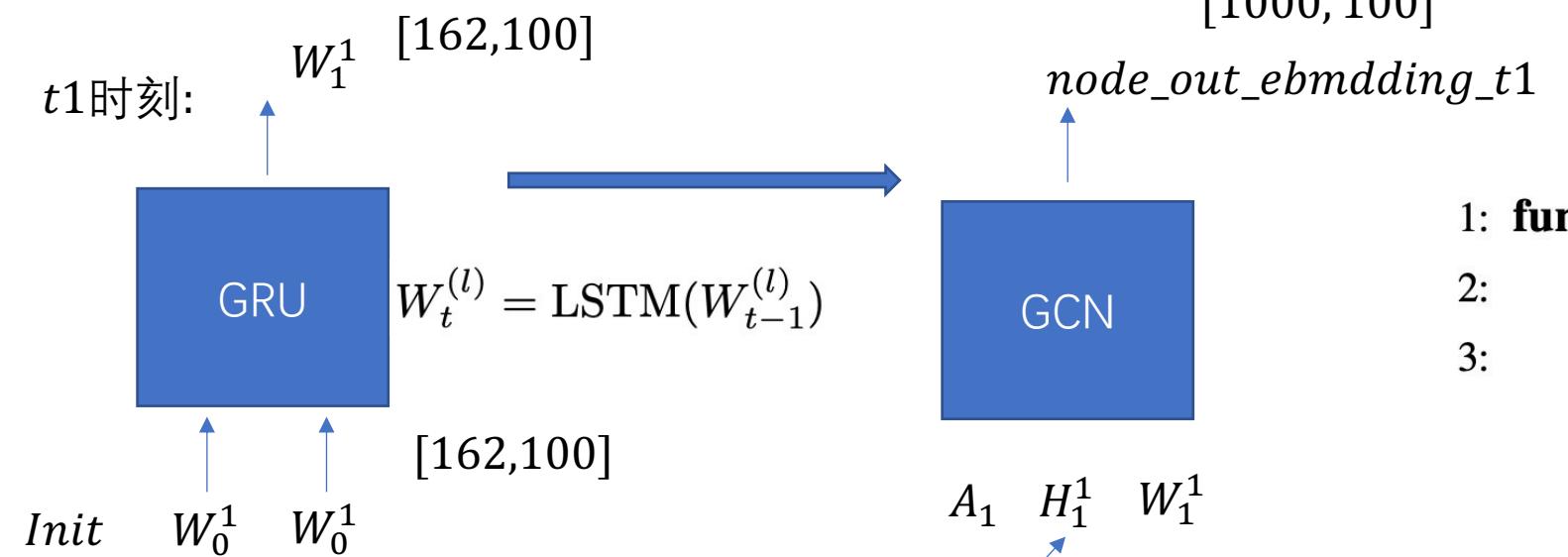
直接应用GCN公式

Weight: 根据GRU学习得到的

```
for unit in self.GRCU_layers:  
    Nodes_list = unit(A_list,Nodes_list) #,nodes_mask_list) # 邻接矩阵; 节点度矩阵
```

两层EvolveGCN ;  
节点degree特征转换成embedding

第1层：



1: **function**  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-O}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$   
2:       $W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$   
3:       $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$

$A_1$      $H_1^1$      $W_1^1$

节点上一层t1时刻的输出特征

在这6个时间步中，最终输出所有节点的embedding

Loss: 预测节点embedding和下个时刻的label

节点上一层t6时刻输出的特征

```
def forward(self,A_list, Nodes_list,nodes_mask_list):
    node_feats= Nodes_list[-1]  node_feats: tensor(in

    for unit in self.GRCU_layers:  unit: GRCU(\n      (ev
        Nodes_list = unit(A_list,Nodes_list)#,nodes_m

    out = Nodes_list[-1]
    if self.skipfeats:
        out = torch.cat((out,node_feats), dim=1)  #
    return out
```



```
nodes_embs = self.gcn(hist_adj_list, # graph  nodes_embs
                      hist_ndFeats_list, # features
                      mask_list) # mask
```

EvolveGCN输出结果

```

for i in range(1 + node_indices.size(1)//predict_batch_size)): i: 0
    cls_input = self.gather_node_embs(nodes_embs, node_indices[:, i*predict_batch_size:(i+1)*predict_batch_size])
    predictions = self.classifier(cls_input)           节点特征
    gather_predictions.append(predictions)
gather_predictions=torch.cat(gather_predictions, dim=0)          有label的点边数据
return gather_predictions, nodes_embs

```

Loss:

```

labels = labels.view(-1,1)
alpha = self.weights(labels)[labels].view(-1,1) # 正负样本的系数 alpha
loss = alpha * (- logits.gather(-1, labels) + self.logsumexp(logits))
return loss.mean()

```

In[31]: logits

Out[31]:

```

tensor([[ -0.0837,  0.0527],
       [ -0.0837,  0.0527],
       [ -0.0837,  0.0527],
       ...,
       [ -0.0837,  0.0527],
       [ -0.0837,  0.0527],
       [ -0.0837,  0.0527]])

```

In[33]: labels

Out[33]:

```

tensor([[1],
       [1],
       [1],
       ...,
       [0],
       [0],
       [0]])

```

tensor([[ 0.0527],

[ 0.0527],

[ 0.0527],

...,

[-0.0837],

[-0.0837],

[-0.0837]])

EGCN-H



## gcn 定义model

```
class EGCRN(torch.nn.Module):
    for i in range(1, len(feats)): i: 1
        GRCU_args = u.Namespace({'in_feats': feats[i-1],
                                'out_feats': feats[i],
                                'activation': activation})
        grcu_i = GRCU(GRCU_args)
        #print (i,'grcu_i', grcu_i)
        self.GRCU_layers.append(grcu_i.to(self.device))
        self._parameters.extend(list(self.GRCU_layers[-1].parameters()))
        self.evolve_weights = mat_GRU_cell(cell_args)
        self.update = mat_GRU_gate(args.rows,
                                  args.cols,
                                  torch.nn.Sigmoid())
```

EGCRN-H

GRU

GRU参数定义

```

1: function  $Z_t = summarize(X_t, k)$  节点embedding [1000, 162]
2:    $y_t = X_t p / \|p\|$   $\rightarrow$  scores = node_embs.matmul(self.scorer) / self.scorer.norm()
3:    $i_t = \text{top-indices}(y_t, k)$  vals, topk_indices = scores.view(-1).topk(self.k)
4:    $Z_t = [X_t \circ \tanh(y_t)]_{i_t}$  node_embs[topk_indices]
5: end function out = node_embs[topk_indices] * tanh(scores[topk_indices].view(-1, 1)) [100, 162]

```

---

$$X_t = [1000, 162] \longrightarrow Z_t = [100, 162]$$

```

1: function  $Z_t = summarize(X_t, k)$ 
2:    $y_t = X_t p / \|p\|$   $X_t = [1000, 162]$ 
3:    $i_t = \text{top-indices}(y_t, k)$  ↓ 降维
4:    $Z_t = [X_t \circ \tanh(y_t)]_{i_t}$   $Z_t = [100, 162]$ 
5: end function ↓ reverse

```

$$Z_t = [162, 100]$$

节点embedding [1000, 162]

```
def forward(self,prev_Q,prev_Z,mask): self: mat_GRU_cell(\n    z_topk = self.choose_topk(prev_Z,mask) z_topk: tensor[[l\n\n        update = self.update(z_topk,prev_Q)\n        reset = self.reset(z_topk,prev_Q)\n\n        h_cap = reset * prev_Q\n        h_cap = self.htilda(z_topk, h_cap)\n\n        new_Q = (1 - update) * prev_Q + update * h_cap\n\n    return new_Q
```

h进行转换

- 1: **function**  $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-H}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$
- 2:       $W_t^{(l)} = \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)})$
- 3:       $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$
- 4: **end function**



# **Streaming Graph Neural Networks**

DGNN Dynamic Graph Neural Network model

# Streaming GNN : 基于RNN的连续性网络

Model type	Model name	Encoder	Link addition	Link deletion	Node addition	Node deletion	Network type
<b>Discrete networks</b>							
Stacked DGNN	GCRN-M1 [58]	Spectral GCN [59] & LSTM	Yes	Yes	No	No	Any
	WD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	CD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	RgCNN [61]	Spatial GCN [62] & LSTM	Yes	Yes	No	No	Any
	DyGGNN [63]	GGNN [64] & LSTM	Yes	Yes	No	No	Any
Integrated DGNN	DySAT [67]	GAT [68] & temporal attention from [69]	Yes	Yes	Yes	Yes	Any
	GCRN-M2 [58]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	GC-LSTM [72]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	EvolveGCN [71]	LSTM integrated in a GCN [57]	Yes	Yes	Yes	Yes	Any
	LRGCN [73]	R-GCN [75] integrated in an LSTM	Yes	Yes	No	No	Any
Continuous networks	RE-Net [74]	R-GCN [75] integrated in several RNNs	Yes	Yes	No	No	Knowledge network
<b>RNN based</b>							
TPP based	Streaming GNN [86]	Node embeddings maintained by architecture consisting of T-LSTM [88]	Yes	No	Yes	No	Directed strictly evolving
	JODIE [87]	Node embeddings maintained by an RNN based architecture	Yes	No	No	No	Bipartite and interaction
	Know-Evolve [89]	TPP parameterised by an RNN	Yes	No	No	No	Interaction, knowledge network
	DyREP [36]	TPP parameterised by an RNN aided by structural attention	Yes	No	Yes	No	Strictly evolving
	LDG [90]	TPP, RNN and self-attention	Yes	No	Yes	No	Strictly evolving
	GHN [92]	TPP parameterised by a continuous time LSTM [93]	Yes	No	No	No	Interaction, knowledge network

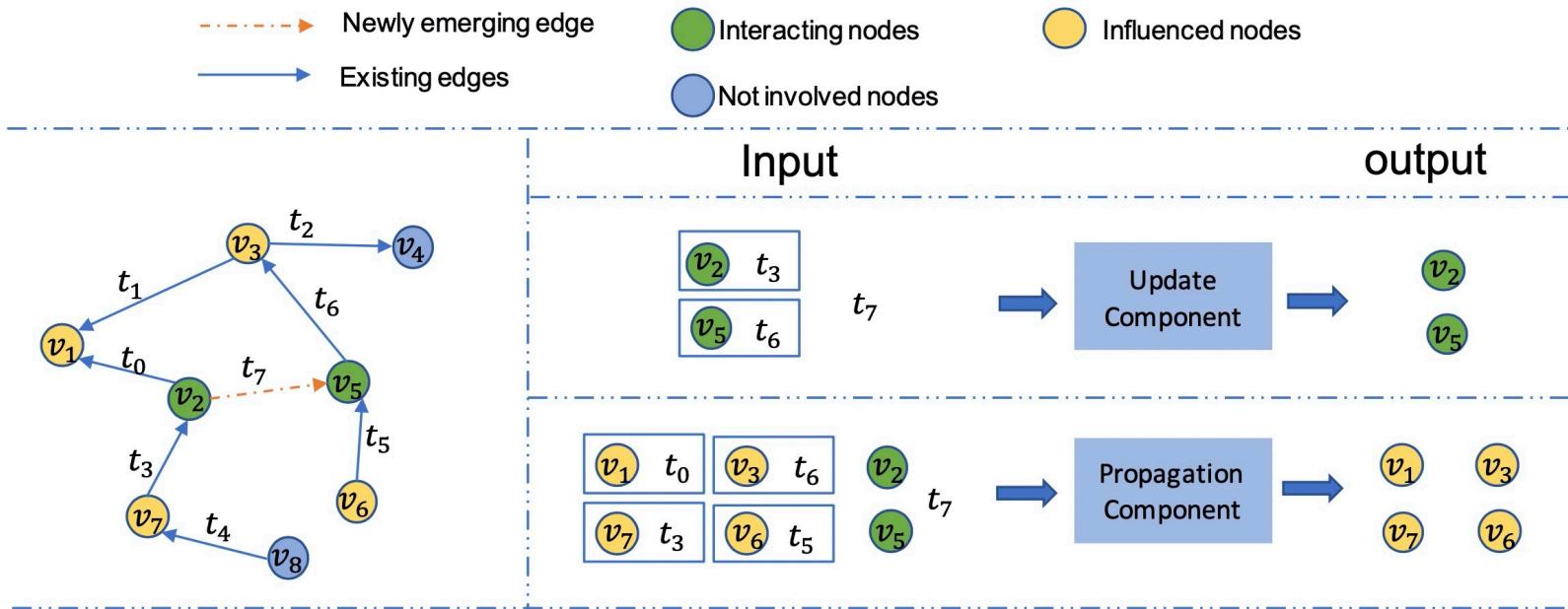
# DGNN

我们要解决的问题是：新的交互发生时，如何不断的更新图网络的信息

一个新的交互不仅可以影响两个交互节点，还可以影响“靠近”交互节点的其他节点，我们称之为“受影响节点”。因此，我们需要将此新交互的信息更新到两个交互节点，并将此信息传播到“受影响节点”。

Streaming Graph Neural Networks

WOODSTOCK'97, July 1997, El Paso, Texas USA



DGNN两个主要部分：

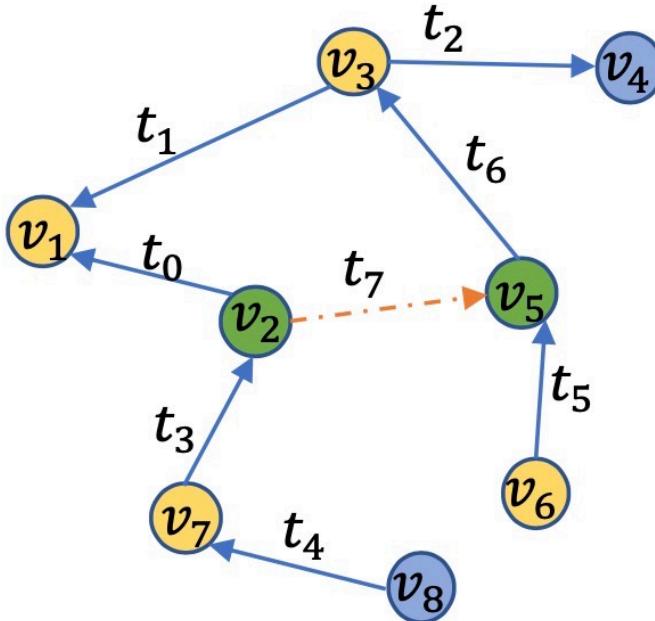
更新组件

传播组件

Figure 1: An overview of DGNN when a new interaction happened at time  $t_7$  from  $v_2$  to  $v_5$ . The two interacting nodes are  $v_2$  and  $v_5$ . The nodes  $\{v_1, v_3, v_6, v_7\}$  are assumed to be the influenced nodes.

## 2.1 The update component

符号介绍



$C_{v2}^s$ :  $v_2$ 作为source节点的cell特征

cell  $C_{v2}$

$C_{v2}^g$ :  $v_2$ 作为target节点的cell特征



hidden  $h_{v2}$

$h_{v2}^s$ :  $v_2$ 作为source节点的hidden特征

$h_{v2}^g$ :  $v_2$ 作为target节点的hidden特征

node  $u_{v2}$ :  $v_2$ 节点的特征

时间相关的概念

$C_{v2}^s(t_0)$  source节点在 $t_0$ 时刻的Cell特征

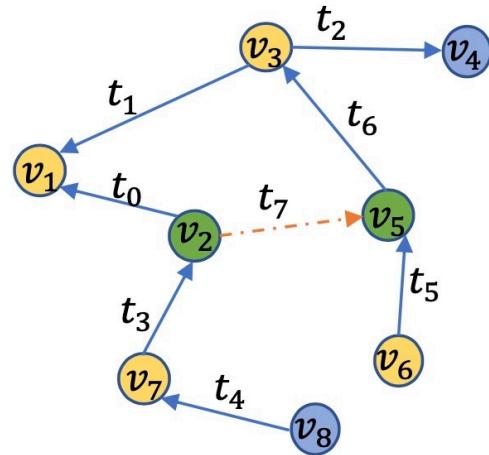
$C_{v2}^s(t_7-)$  节点在 $t_7$ 时刻之前source节点cell特征

$C_{v2}^s(t_3)$

$C_{v2}^g(t_7-)$  节点在 $t_7$ 时刻之前最新cell特征

$C_{v2}^g(t_3)$

## 2.1 The update component



节点在 $t_7$ 时刻之前的状态

$h_{v2}^g$ :  $v_2$ 作为target节点的hidden特征

$C_{v2}^g$ :  $v_2$ 作为target节点的cell特征

$h_{v2}^s$ :  $v_2$ 作为source节点的hidden特征

$C_{v2}^s$ :  $v_2$ 作为source节点的cell特征

$u_{v2}$ :  $v_2$ 节点的特征

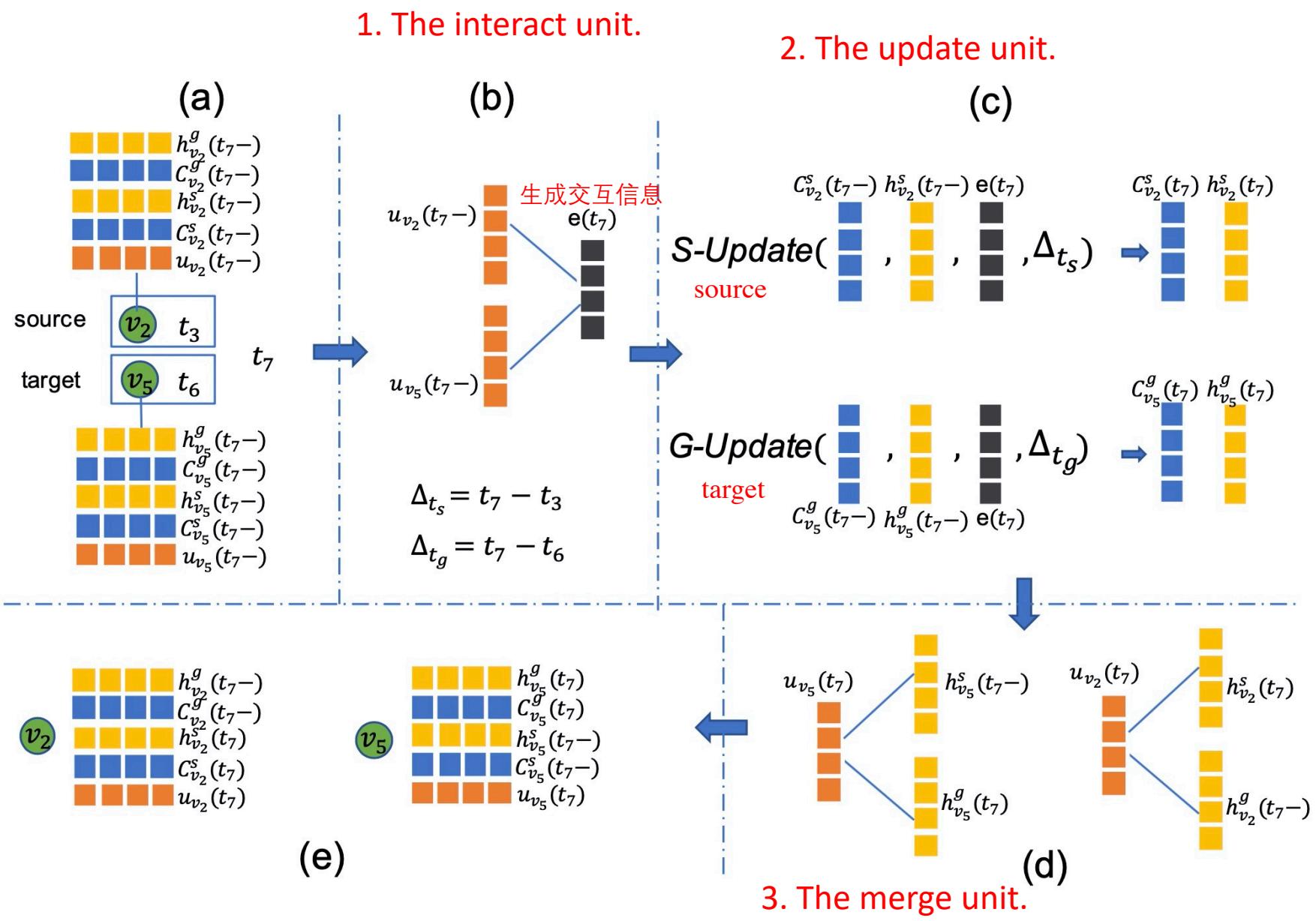
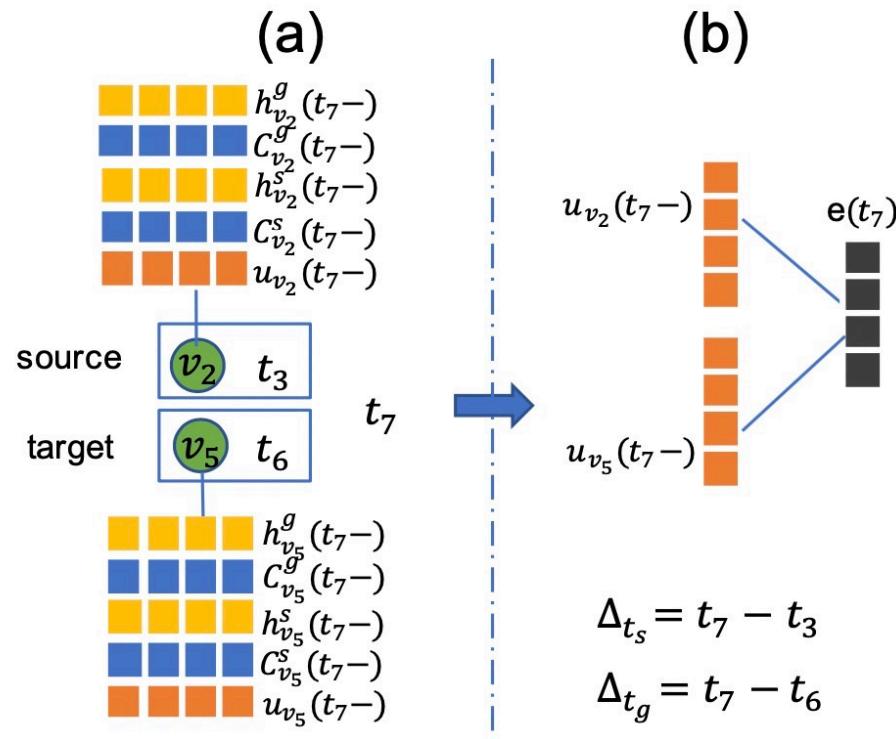


Figure 3: An example to illustrate an overview about the update components when an interaction  $v_2, v_5, t_7$  happened.

## 2.1.1 The interact unit.

$u_{v_s}(t-)$  and  $u_{v_g}(t-)$  are the general features of the nodes  $v_s$

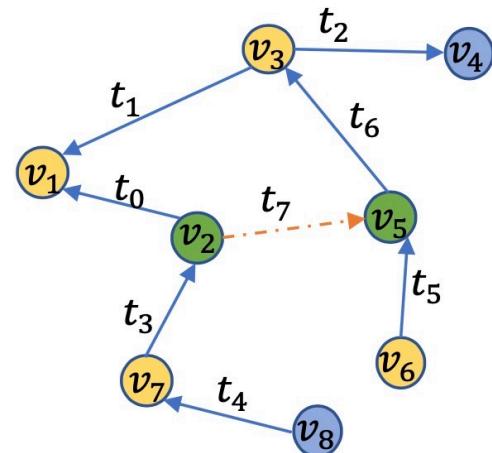
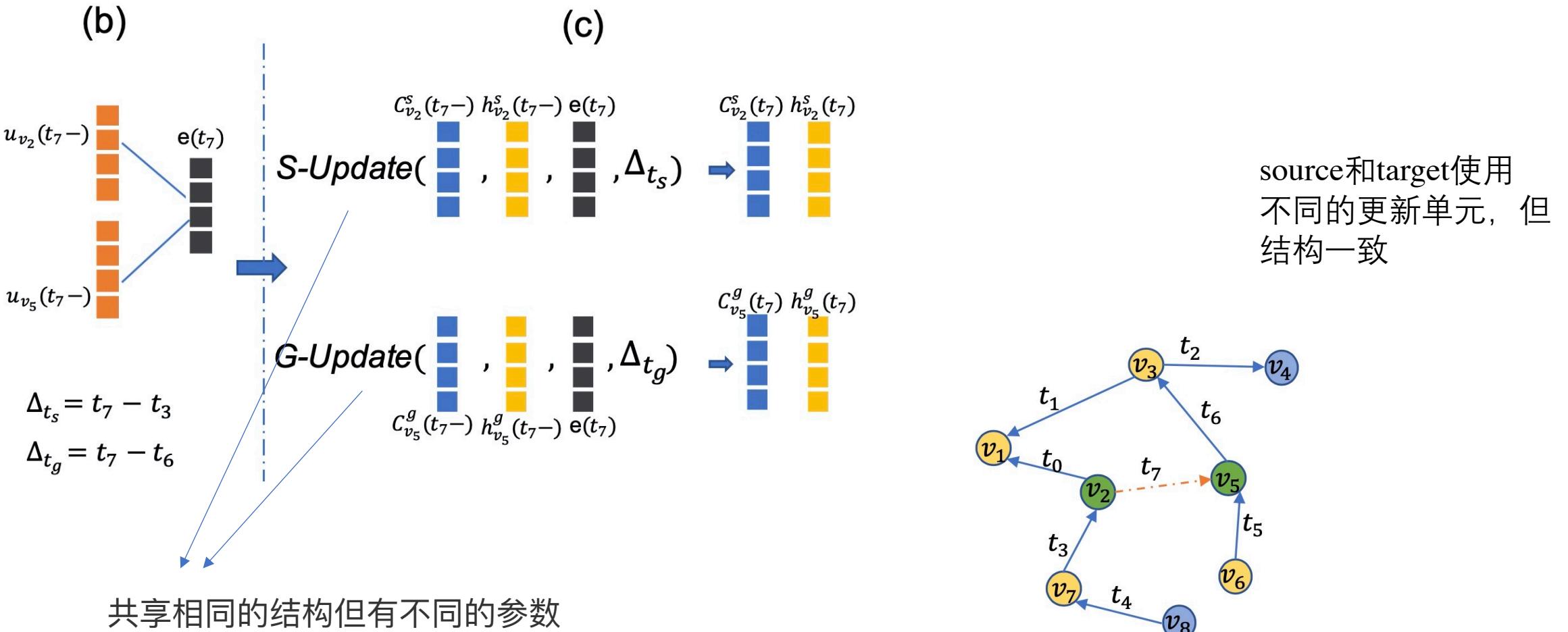


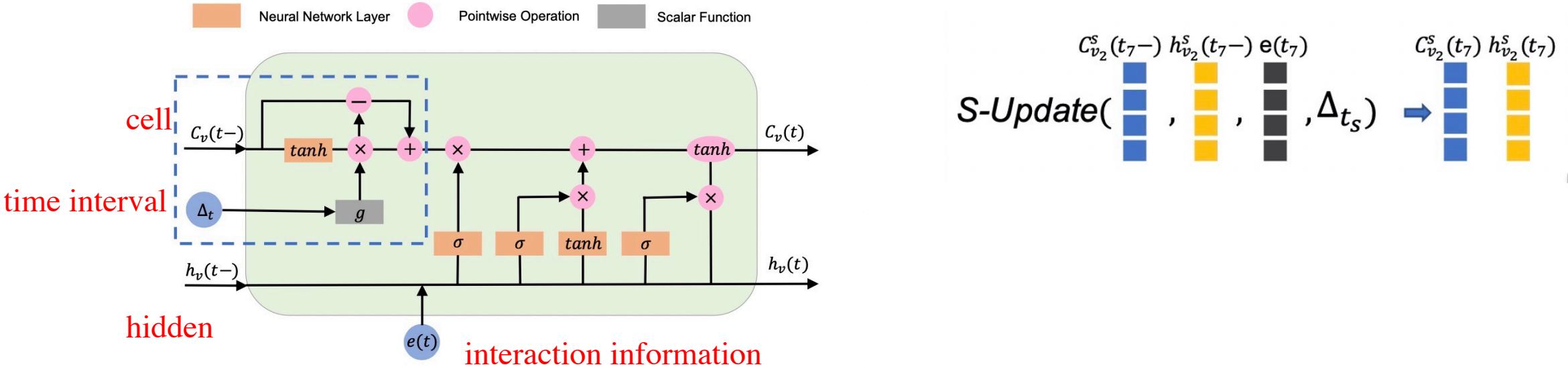
$$e(t) = act(W_1 \cdot u_{v_s}(t-) + W_2 \cdot u_{v_g}(t-) + b_e) \quad (1)$$

Source 节点特征  
包含  $\{v_s, v_g, t\}$  交互信息

Target 节点特征

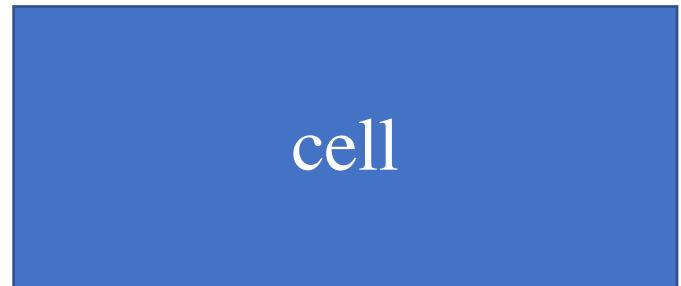
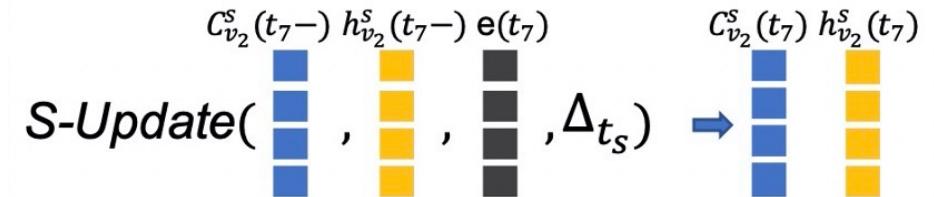
## 2.1.2 The update unit.





**Figure 4: An illustration of the update unit**

和标准LSTM不同点就在于蓝色虚线部分





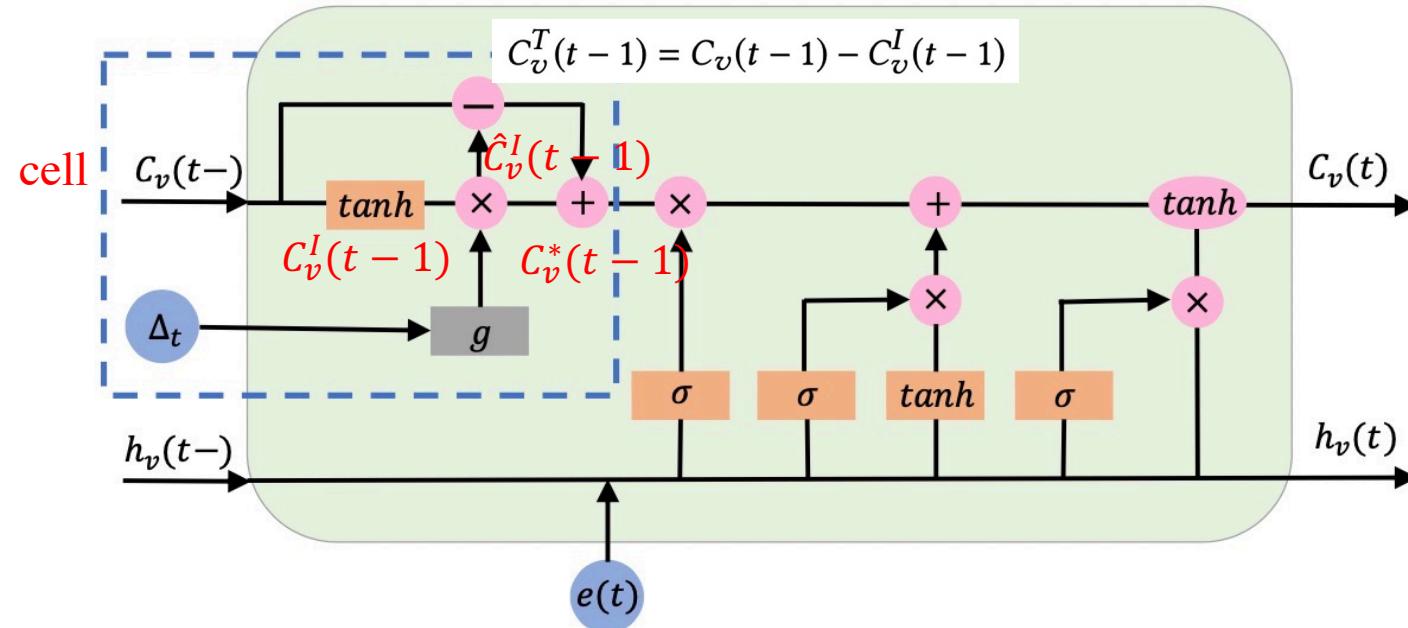
Neural Network Layer



Pointwise Operation



Scalar Function

 $C_v(t-1)$  根据时间调整为  $C_v^*(t-1)$ Input :  $C_v(t-1)$   
 $\Delta_t$ Output :  $C_v^*(t-1)$ 

捕获时间信息，选择遗忘：  
时间间隔会影响应该如何忘记旧信息。直观地  
知道，在过去发生的交互对节点的当前信息的  
影响应该较小，因此应该“大量”地忘记它们

**Figure 4: An illustration of the update unit**

现在节点cell信息，经过网络，生成短时记忆

 $g(\Delta_t)$ 

递减函数，时间间隔越大，短时记忆越少

short term memory

$$C_v^I(t-1) = \tanh(W_d \cdot C_v(t-1) + b_d) \quad (2)$$

$$\hat{C}_v^I(t-1) = C_v^I(t-1) * g(\Delta_t) \quad g(\Delta_t) \text{ 遗忘} \quad (3)$$

$$\text{long term memory } C_v^T(t-1) = C_v(t-1) - C_v^I(t-1) \quad (4)$$

当短时记忆被遗忘时，长时记忆没有影响

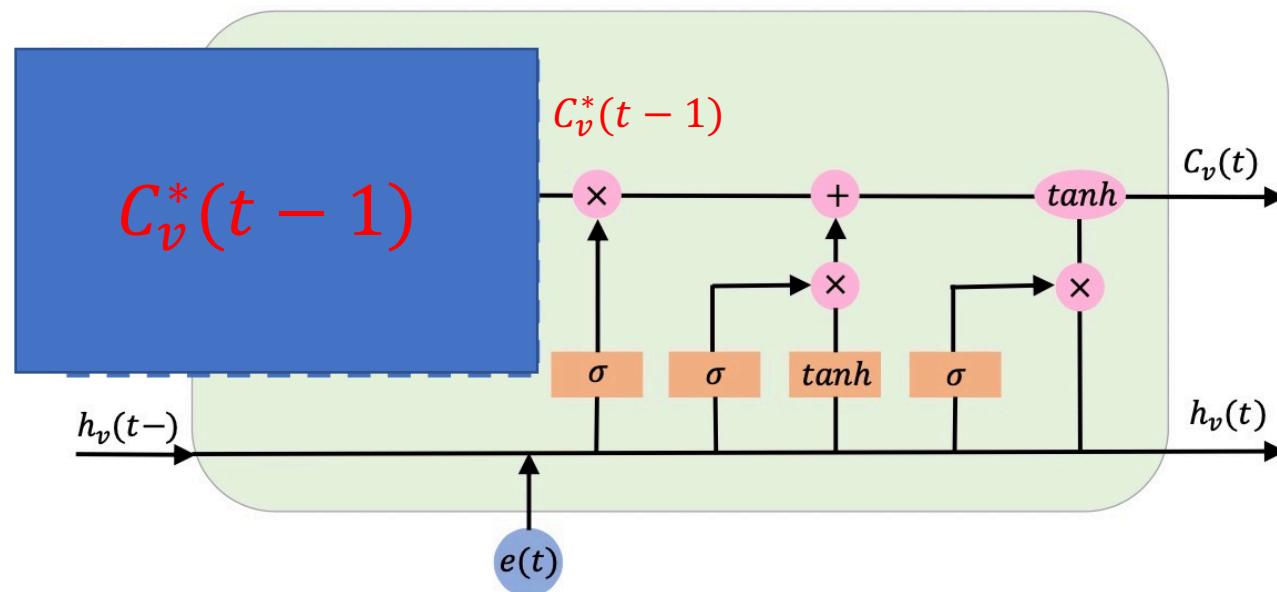
$$C_v^*(t-1) = C_v^T(t-1) + \hat{C}_v^I(t-1) \quad (5)$$

discounted short term memory

Neural Network Layer

Pointwise Operation

Scalar Function



**Figure 4: An illustration of the update unit**

标准的LSTM结构

The formulations of the rest part of the update unit, which are the same as a standard LSTM unit, are as follows

$$f_t = \sigma(W_f \cdot e(t) + U_f \cdot h_v(t-1) + b_f) \quad (6)$$

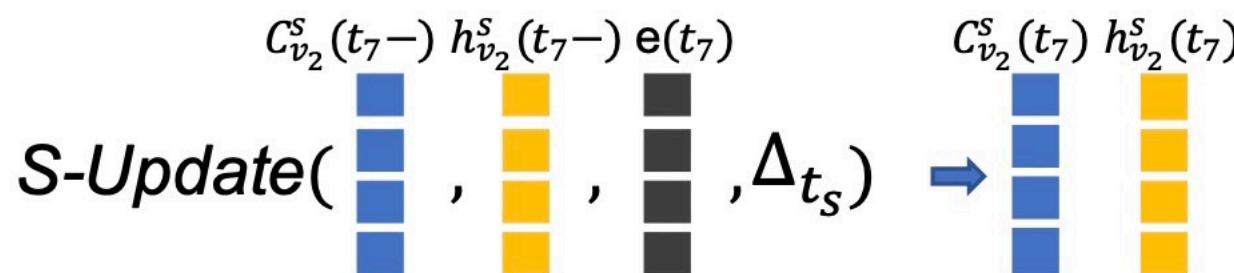
$$i_t = \sigma(W_i \cdot e(t) + U_i \cdot h_v(t-1) + b_i) \quad (7)$$

$$o_t = \sigma(W_o \cdot e(t) + U_o \cdot h_v(t-1) + b_o) \quad (8)$$

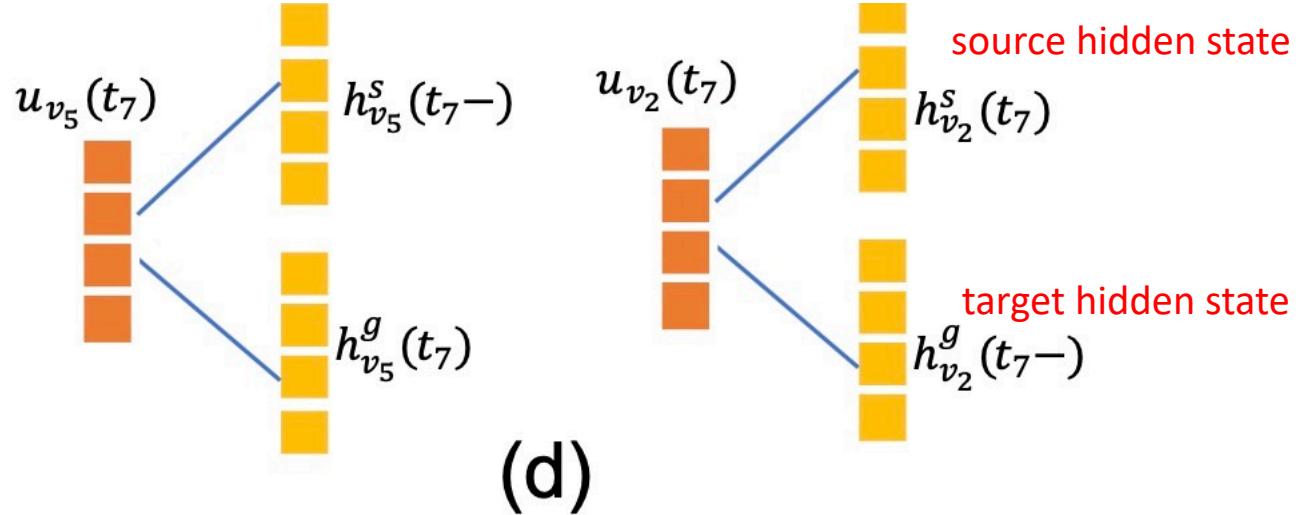
$$\tilde{C}_v(t) = \tanh(W_c \cdot e(t) + U_c \cdot h_v(t-1) + b_c) \quad (9)$$

$$C_v(t) = f_t * C_v^*(t-1) + i_t * \tilde{C}_v(t) \quad (10)$$

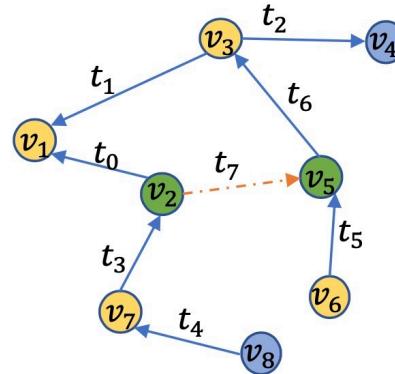
$$h_v(t) = o_t * \tanh(C_v(t)). \quad (11)$$



### 2.1.3 The merge unit.



$v_5$  为 target 节点, 所以更新了  $h_{v_5}^g(t)$



$v_2$  为 source 节点, 所以更新了  $h_{v_2}^s(t)$

$v_2$  的  $h_{v_2}^g(t)$  状态没有更新

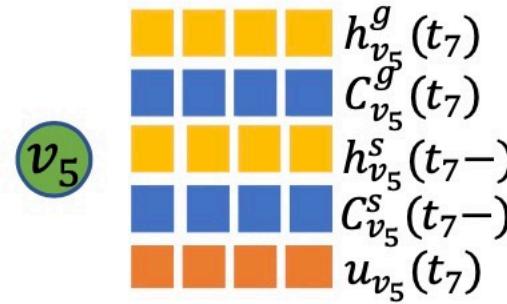
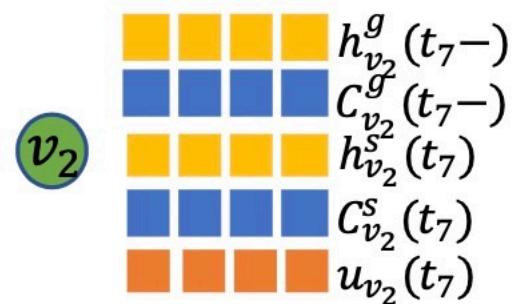
$$u_{v_s}(t) = W^s \cdot h_{v_s}^s(t) + W^g \cdot h_{v_s}^g(t-) + b_u$$

$$u_{v_g}(t) = W^s \cdot h_{v_g}^s(t-) + W^g \cdot h_{v_g}^g(t) + b_u$$



更新后, 生成了  $v_2$  和  $v_5$  的新特征

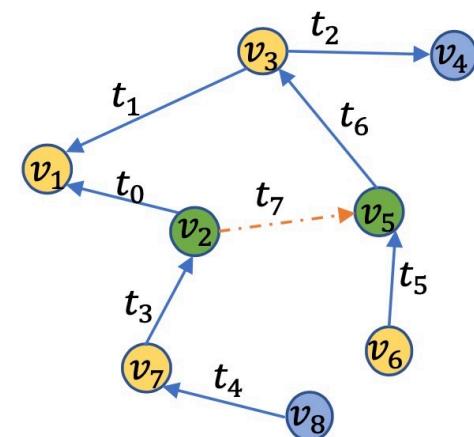
更新交互节点信息



$v_5$  – target节点, 没更新source的信息

(e)

$v_2$  – source节点, 没更新target的信息



## 2.2 The propagation component

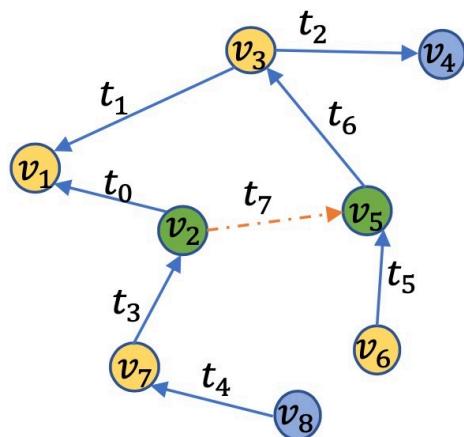
在这项工作中，我们选择两个“交互节点”的当前邻居作为“受影响的节点”。

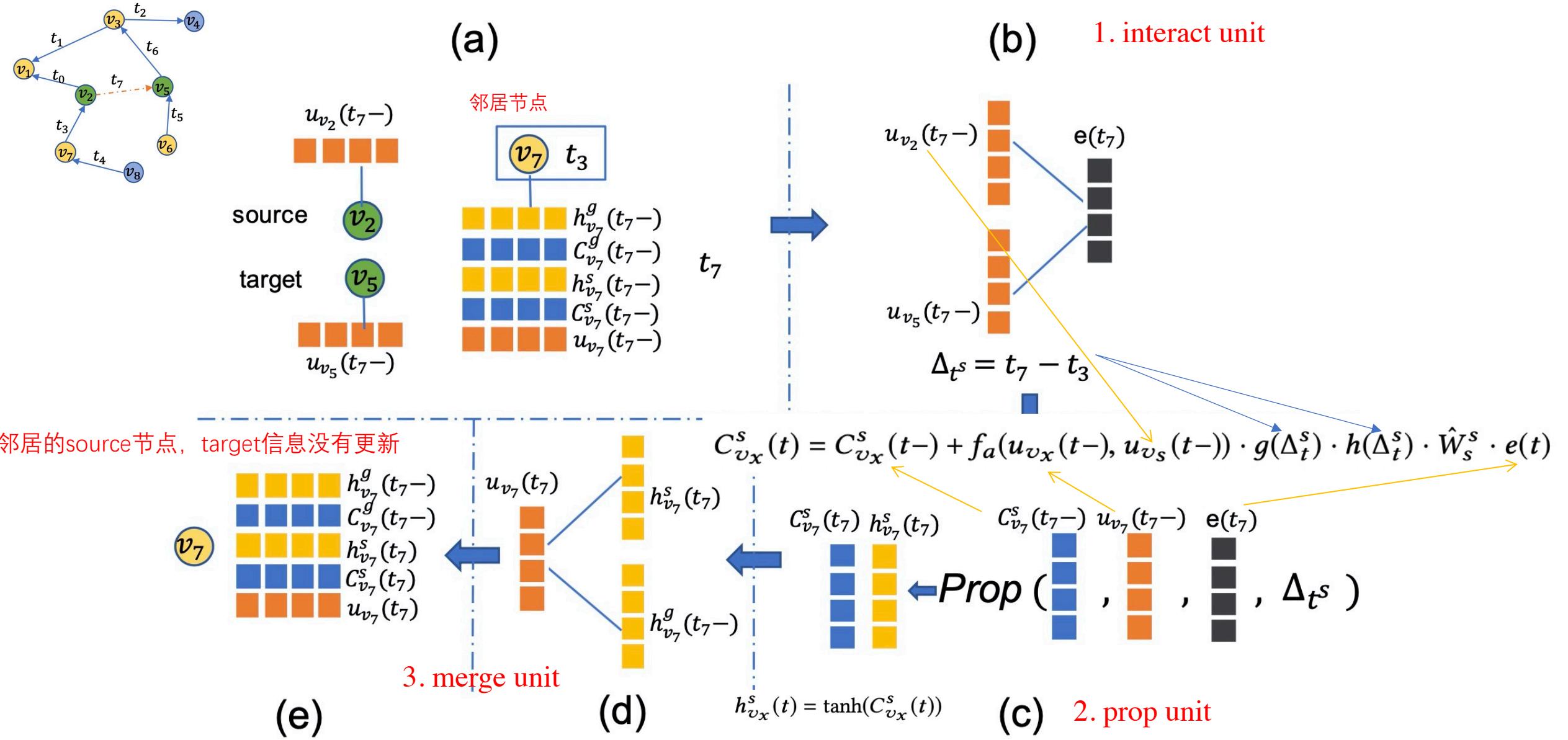
主要原因是三方面的：

首先，正如在挖掘流图时所了解的那样，新边缘对整个图的影响通常是局部的

其次，在我们将信息传播到邻居之后，一旦受影响的节点与其他节点发生交互，信息将进一步传播

第三，我们凭经验发现，当传播更多的受影响点时，性能不会显着提高，甚至会降低，因为我们也可能在传播过程中引入噪声。





从source node  $v_s$  到source neighbors  $N^s(v_s)$

Figure 5: The propagation to the source neighbor  $v_7$  of the source node  $v_2$  when a new interaction  $\{v_2, v_5, t_7\}$  happened.

## 符号概念

$\{v_s, v_g, t\}$  neighbors of these two nodes until time  $t$

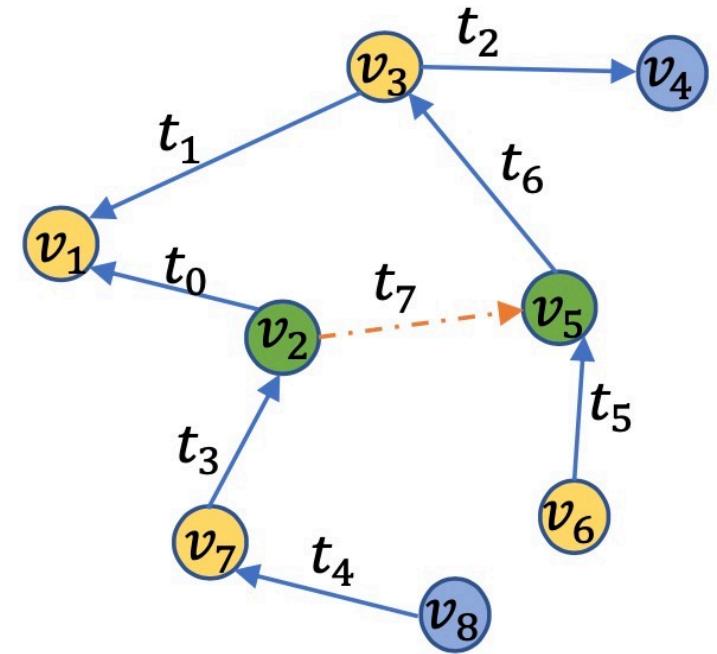
为受影响节点  $N(v_s)$  and  $N(v_g)$ .

$$\{v_1, v_7\} \quad \{v_3, v_6\}$$

有向图中，可以进一步分解为

$$N(v_s) = N^s(v_s) \cup N^g(v_s)$$

$$N(v_g) = N^s(v_g) \cup N^g(v_g)$$

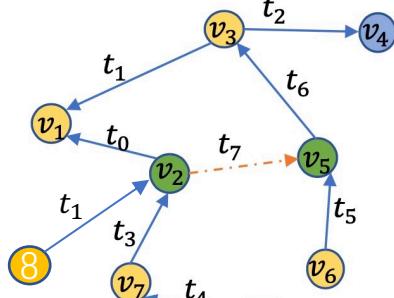


prop unit, 受影响节点

- 1) 从source node  $v_s$  到source neighbors  $N^s(v_s)$   $V_7$
- 2) 从source node  $v_s$  到target neighbors  $N^g(v_s)$   $V_1$
- 3) 从target node  $v_g$  到source neighbors  $N^s(v_g)$   $V_6$
- 4) 从target node  $v_g$  到target neighbors  $N^g(v_g)$   $V_3$

## 2. prop unit

计算 $v_2$ 的source邻居 $\{v_7, v_8\}$



$$C_{v_x}^s(t) = C_{v_x}^s(t-) + f_a(u_{v_x}(t-), u_{v_s}(t-)) \cdot g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t) \quad (15)$$

$$h_{v_x}^s(t) = \tanh(C_{v_x}^s(t)) \quad (16)$$

$$u_{v_7}(t_3) \quad u_{v_2}(t_3)$$

直观地说，将交互信息传播给“非常老的邻居”可能会引入噪声。

$$h(\Delta_t^s) = \begin{cases} 1, & \Delta_t^s \leq \tau, \\ 0, & \text{otherwise.} \end{cases}$$

过滤一些影响节点，如果时间间隔大于 $\tau$ ，我们将停止向此类邻居传播信息，此操作的一个优点是可以进行传播步骤更有效率。

$$f_a(u_{v_x}(t-), u_{v_s}(t-)) = \frac{\exp(u_{v_x}(t-)^T u_{v_s}(t-))}{\sum_{v \in N^s(v_s)} \exp(u_v(t-)^T u_{v_s}(t-))} \quad (17)$$

$v_7$   
邻居节点

$v_2$   
本身节点

source邻居 $\{v_7, v_8\}$

$u_{v_x}(t-)$   $u_{v_s}(t-)$  的连接强度

有多个连接的话，会将影响均分到各节点上

## 2.3 Parameter learning

### 2.3.1 Parameter learning for link prediction.

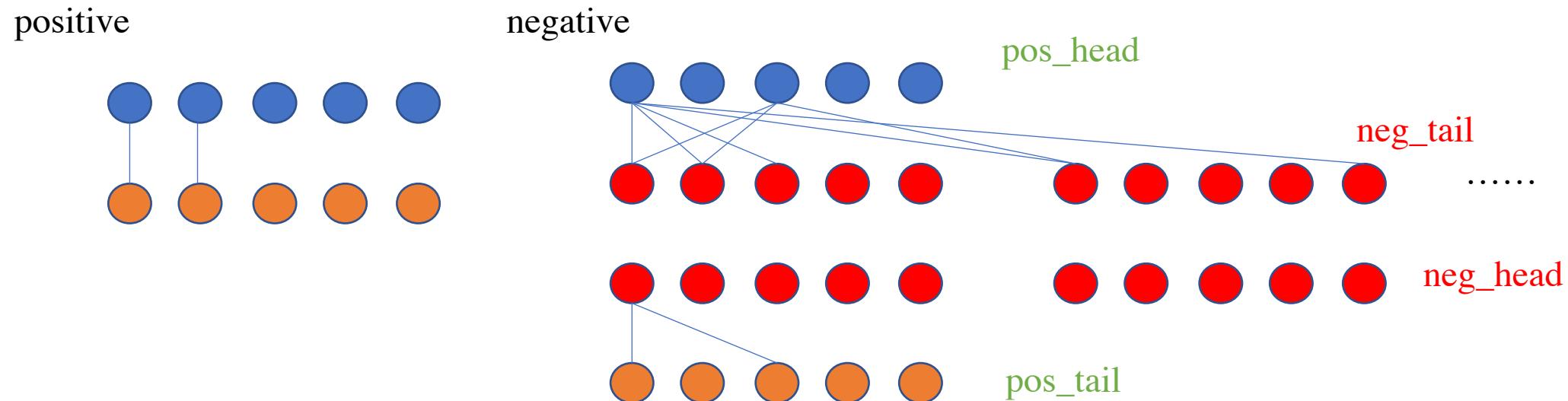
Link prediction任务

$$u_{v_s}^s(t-) = P^s \cdot u_{v_s}(t-) \quad \text{节点特征}$$

$$u_{v_g}^g(t-) = P^g \cdot u_{v_g}(t-)$$

$$\begin{aligned} J((v_s, v_g, t)) = & -\log(\sigma(u_{v_s}^s(t-)^T u_{v_g}^g(t-))) \\ & - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(u_{v_s}^s(t-)^T u_{v_n}^g(t-))) \quad (18) \end{aligned}$$

1 -



### 2.3.2 Learning parameters for node classification.

we first project  $u_v(t)$  to  $u_v^c(t) \in \mathcal{R}^{N_c \times 1}$ .

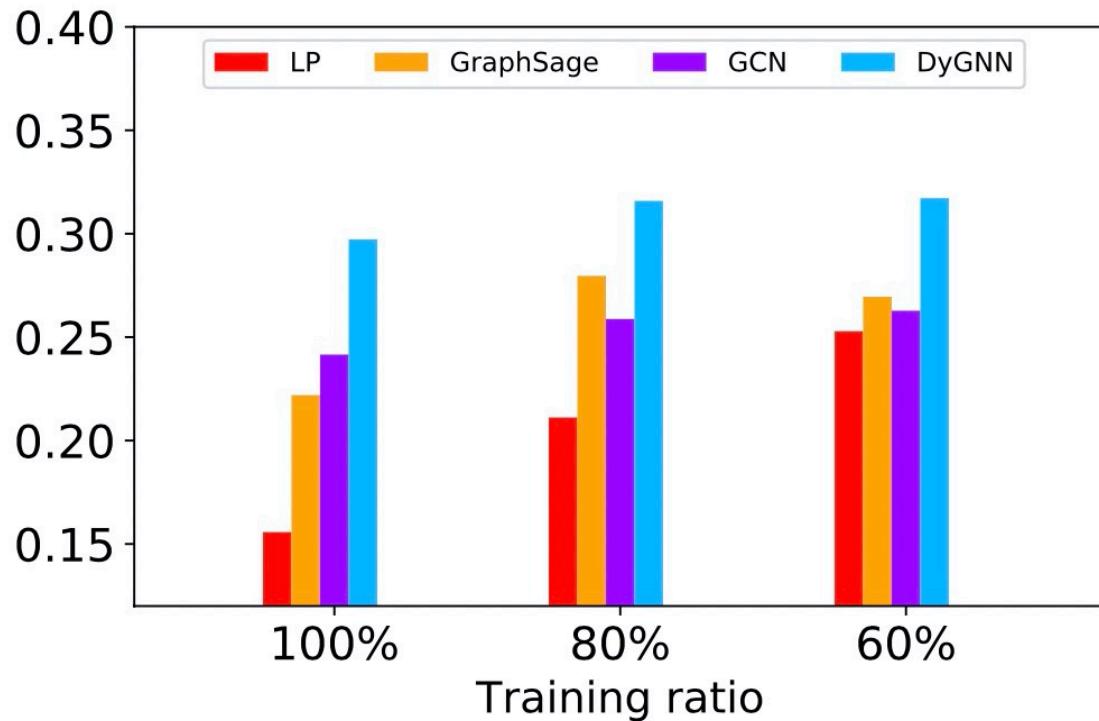
属于哪个类别的概率

$$J(v, t) = - \sum_{i=0}^{N_c-1} y[i] \log \left( \frac{\exp(u_v^c(t))[i]}{\sum_{j=0}^{N_c-1} \exp(u_v^c(t))[j]} \right);$$

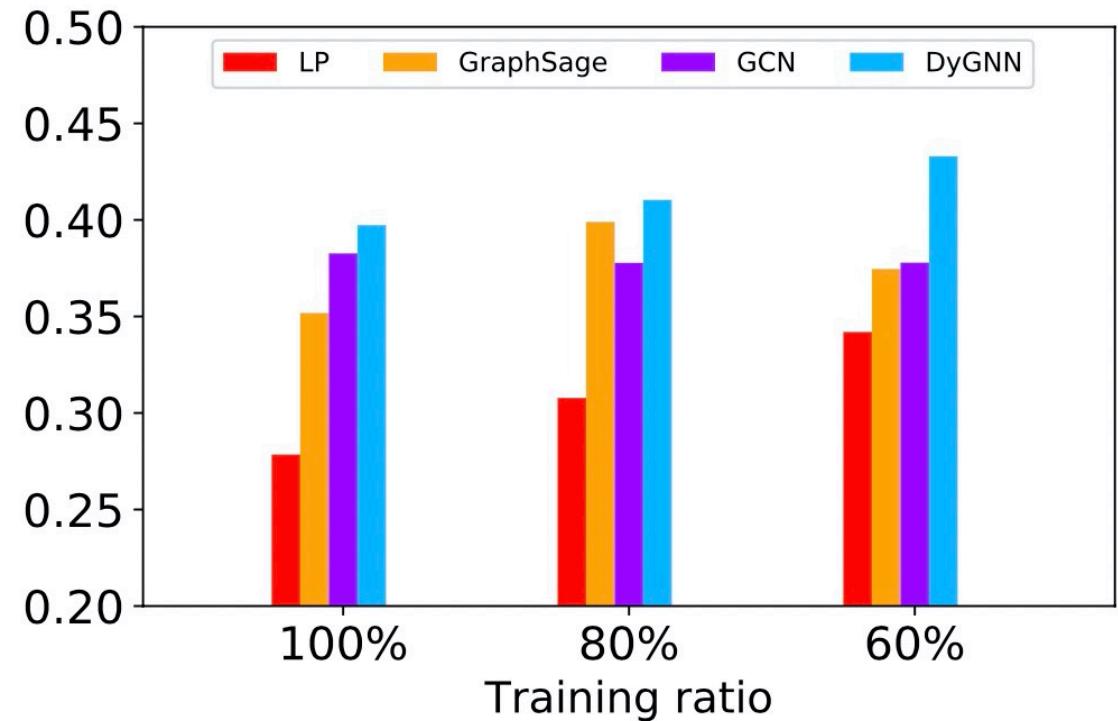
在时间t的节点loss

**Table 2: Performance comparison of link prediction.**

Baselines	UCI			DNC			Epinions		
	MRR	Recall@20	Recall@50	MRR	Recall@20	Recall@50	MRR	Recall@20	Recall@50
DGNN	<b>0.0342</b>	<b>0.1284</b>	<b>0.2547</b>	<b>0.0536</b>	0.1852	<b>0.3884</b>	<b>0.0204</b>	<b>0.0848</b>	<b>0.1894</b>
GCN	0.0138	0.0632	0.1176	0.0447	<b>0.2032</b>	0.3291	0.0045	0.0071	0.0119
GraphSage	0.0060	0.0161	0.0578	0.0167	0.0576	0.1781	0.0035	0.0072	0.0108
node2vec	0.0056	0.0184	0.0309	0.0202	0.0719	0.178	0.0135	0.0571	0.1240
DynGEM	0.0146	0.0773	0.1455	0.0271	0.0971	0.2356	0.0150	0.0657	0.1233
CPTM	0.0138	0.0921	0.1082	0.0109	0.0072	0.0108	0.0036	0.0060	0.0125
DANE	0.0040	0.0110	0.0233	0.0128	0.0270	0.0432	0.0040	0.0100	0.0120
DynamicTriad	0.0150	0.0610	0.1236	0.0146	0.0414	0.0665	0.0170	0.0729	0.1629



(a) Epinions:  $F_1$ -macro



(b) Epinions:  $F_1$ -micro

**Figure 6: Performance Comparison of Node classification on Epinions dataset**

## 消融实验

**Table 3: Comparison of variants on the link prediction task.**

Baselines	UCI			DNC			Epinions		
	MRR	Recall@20	Recall@50	MRR	Recall@20	Recall@50	MRR	Recall@20	Recall@50
DGNN	0.0342	0.1284	0.2547	0.0536	0.185	0.3884	0.0204	0.0848	0.1894
DGNN-prop	0.0103	0.0444	0.1087	0.0046	0	0	0.0171	0.0633	0.1514
DGNN-ti	0.0174	0.0918	0.2118	0.0050	0	0.0054	0.0157	0.0591	0.1589
DGNN-att	0.0200	0.0844	0.2235	0.0562	0.1547	0.3219	0.0177	0.0651	0.1655

通过删除模型中的某些组件来形成模型的以下变体

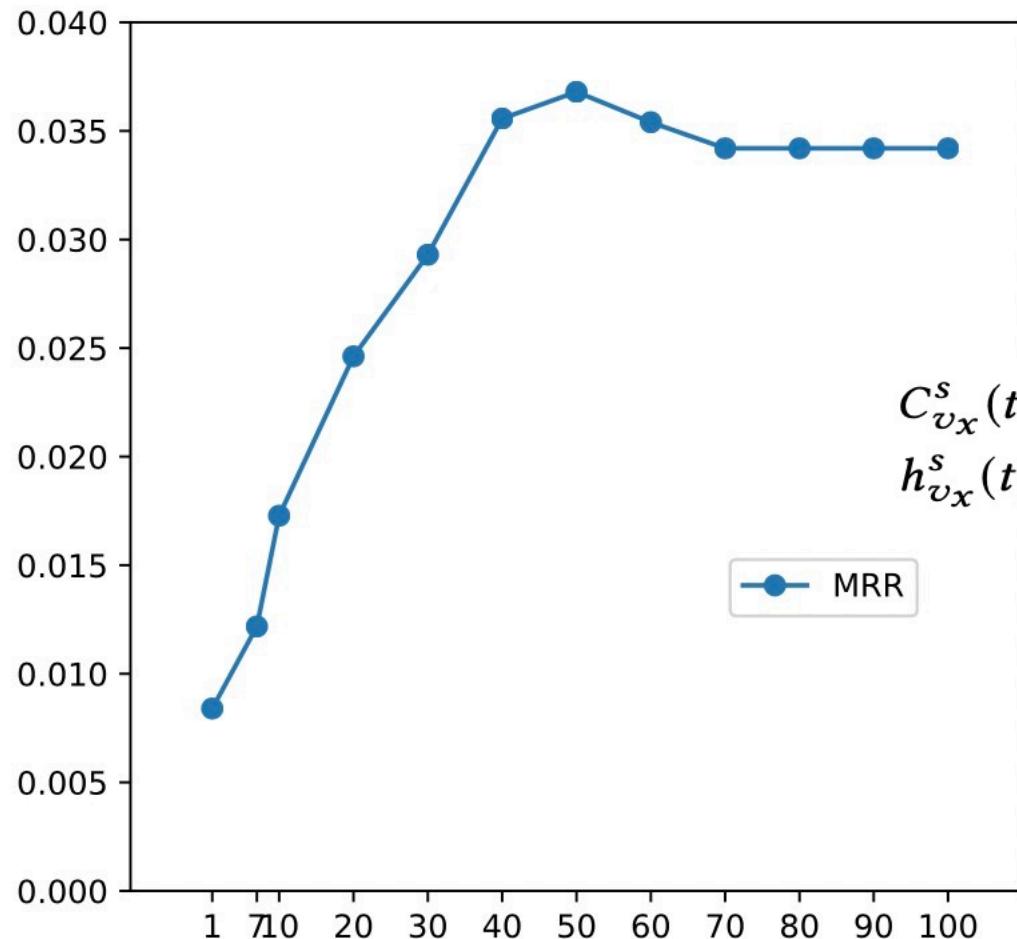
DGNN-prop: 删除了传播组件

DGNN-ti: 更新组件和传播组件中均未使用时间间隔信息。

DGNN-att: 删除了传播组件中的注意力机制

## 参数分析

此数据集的持续时间为194天，  
我们将阈值 $\tau$ 设置为1、7天和10-100天，步长为10



$$C_{v_x}^s(t) = C_{v_x}^s(t-) + f_a(u_{v_x}(t-), u_{v_s}(t-)) \cdot g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t) \quad (15)$$

$$h_{v_x}^s(t) = \tanh(C_{v_x}^s(t)) \quad (16)$$

$$h(\Delta_t^s) = \begin{cases} 1, & \Delta_t^s \leq \tau, \\ 0, & \text{otherwise.} \end{cases}$$

**Figure 7: Impact of  $\tau$**

传播组件中引入了一个参数 $\tau$ 来过滤一些“受影响的节点”

Code

## Code

```
data = Temporal_Dataset('Dataset/UCI_email_1899_59835/opsahl-ucsocial/out.opsahl-ucsocial',1,2)
num_nodes = 1899
model_save_dir = model_save_dir + 'UCI/'
print('Train on UCI_message dataset')
train(args, data, num_nodes, model_save_dir)

dyGnn = DyGNN(num_nodes,64,64,device, w, is_att, transfer, nor, if_no_time, threshold, second_order, if_updated, drop_p, num_negative, act, if_propagation, decay_method)

self.combiner = Combiner(embedding_dims, embedding_dims, act).to(device) # [64, 64]

class Combiner(nn.Module):
    def __init__(self, input_size, output_size, act, bias = True):
        super(Combiner, self).__init__()
        self.h2o = nn.Linear(input_size, output_size, bias)
        self.l2o = nn.Linear(input_size, output_size, bias)
    if act == 'tanh':
        self.act = nn.Tanh()
    elif act == 'sigmoid':
        self.act = nn.Sigmoid()
    else:
        self.act = nn.ReLU()


$$u_{v_s}(t) = W^s \cdot h_{v_s}^s(t) + W^g \cdot h_{v_s}^g(t-) + b_u \quad (13)$$

```

$$\frac{1}{1 + \log(e + 2 * \text{delta})}$$

递减函数

```
self.decayer = Decayer(device, w, decay_method)
```

In[19]: self.linear

Out[19]: Linear(in\_features=1, out\_features=1, bias=False)

## 1. interact unit

```
self.edge_updater_head = Edge_updater_nn(embedding_dims, edge_output_size, act, relation_size).to(device)
self.edge_updater_tail = Edge_updater_nn(embedding_dims, edge_output_size, act, relation_size).to(device)
```

```
class Edge_updater_nn(nn.Module):
    def __init__(self, node_input_size, output_size, act = 'tanh', relation_input_size = None, bias = True):
        super(Edge_updater_nn, self).__init__()
        self.h2o = nn.Linear(node_input_size, output_size, bias)
        self.l2o = nn.Linear(node_input_size, output_size, bias)
        if relation_input_size is not None:
            self.r2o = nn.Linear(relation_input_size, output_size, bias)
        if act == 'tanh':
            self.act = nn.Tanh()
        elif act == 'sigmoid':
            self.act = nn.Sigmoid()
        else:
            self.act = nn.ReLU()
```

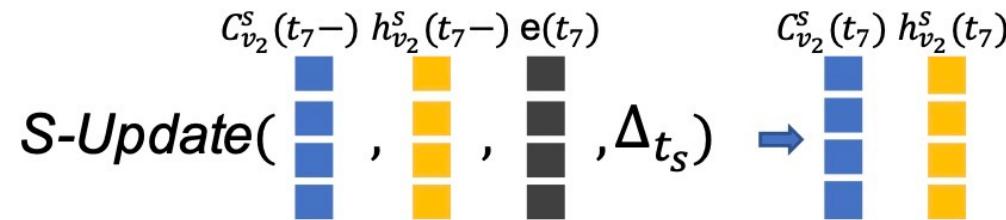
$$e(t) = \text{act}(W_1 \cdot u_{v_s}(t-) + W_2 \cdot u_{v_g}(t-) + b_e) \quad (1)$$

```

self.node_updater_head = TLSTM(edge_output_size, embedding_dims).to(device)
self.node_updater_tail = TLSTM(edge_output_size, embedding_dims).to(device)

class TLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, bias = True): self: TLSTM(\n    (i2h):
        super(TLSTM, self).__init__()
        self.i2h = nn.Linear(input_size, 4*hidden_size, bias)
        self.h2h = nn.Linear(hidden_size, 4*hidden_size, bias)
        self.c2s = nn.Sequential(nn.Linear(hidden_size, hidden_size, bias), nn.Tanh())
        self.sigmoid = nn.Sigmoid()
        self.tanh = nn.Tanh()

```



```

if self.is_att:
    self.attention = Attention(embedding_dims).to(device)

```

```

class Attention(nn.Module):
    def __init__(self, embedding_dims):
        super(Attention, self).__init__()
        self.bilinear = nn.Bilinear(embedding_dims, embedding_dims, 1) # [64, 64 ,1]
        self.softmax = nn.Softmax(0)

```

```

m = nn.Bilinear(20, 30, 40)
input1 = torch.randn(128, 20)
input2 = torch.randn(128, 30)
output = m(input1, input2)

```

```

for k in range(weight.shape[0]): # 40
    buff = np.dot(x1, weight[k]) # [128,20] * [20,30] => [128,30]
    buff = buff * x2 # [128,30] * [128,30] 相乘后, 相加
    buff = np.sum(buff, axis=1) # [128,]
    y[:,k] = buff

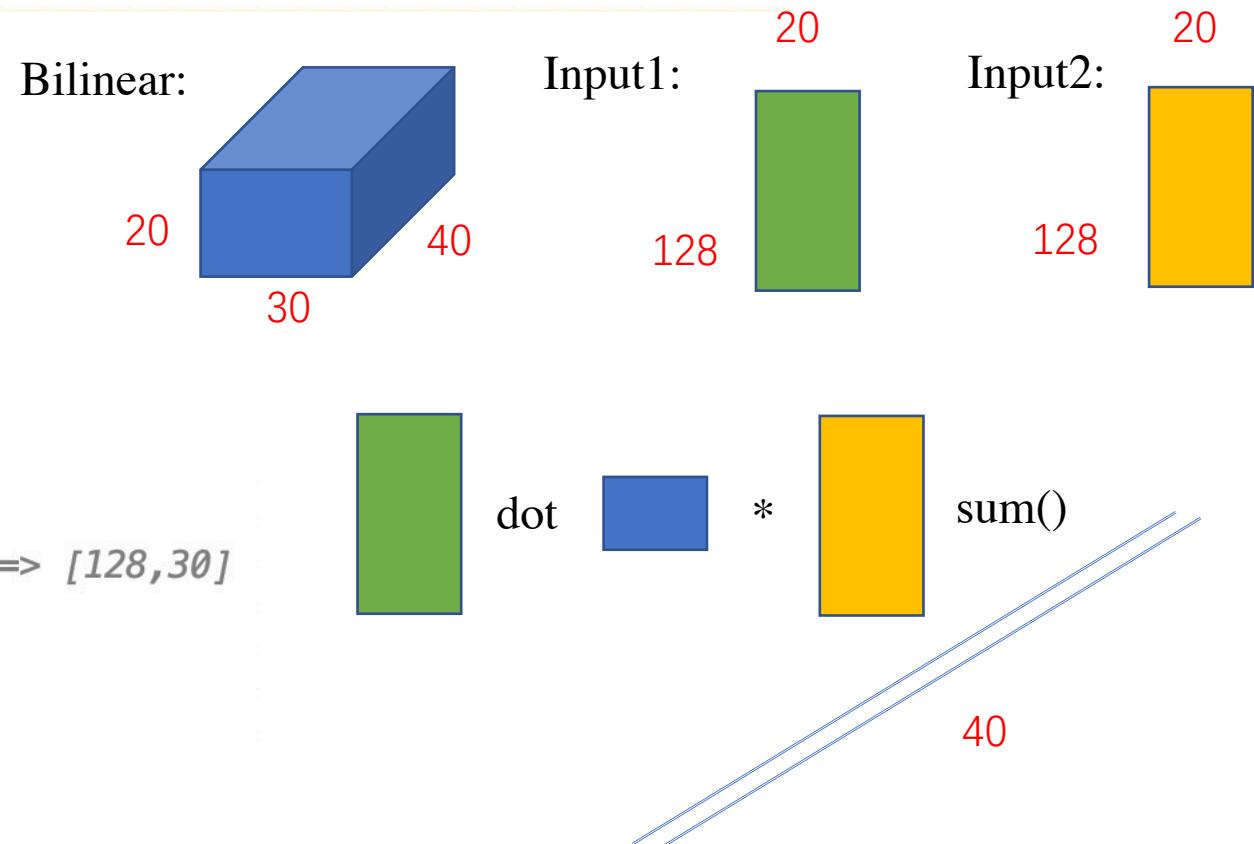
```

$$y = x_1^T A x_2 + b$$

```

>>> m = nn.Bilinear(20, 30, 40)
>>> input1 = torch.randn(128, 20)
>>> input2 = torch.randn(128, 30)
>>> output = m(input1, input2)
>>> print(output.size())
torch.Size([128, 40])

```



```
self.cell_head = nn.Embedding(num_embeddings, embedding_dims, weight).to(device)
self.cell_head.weight.requires_grad = False
self.cell_tail = nn.Embedding(num_embeddings, embedding_dims, weight).to(device)
self.cell_tail.weight.requires_grad = False

self.hidden_head = nn.Embedding(num_embeddings, embedding_dims, weight).to(device)
self.hidden_head.weight.requires_grad = False
self.hidden_tail = nn.Embedding(num_embeddings, embedding_dims, weight).to(device)
self.hidden_tail.weight.requires_grad = False

self.node_representations = nn.Embedding(num_embeddings, embedding_dims, weight).to(device)
self.node_representations.weight.requires_grad = False
```

## 几个embedding

```
self.cell_head = nn.Embedding.from_pretrained(self.cell_head_copy.weight.clone()).to(self.device)
self.cell_tail = nn.Embedding.from_pretrained(self.cell_tail_copy.weight.clone()).to(self.device)
self.hidden_head = nn.Embedding.from_pretrained(self.hidden_head_copy.weight.clone()).to(self.device)
self.hidden_tail = nn.Embedding.from_pretrained(self.hidden_tail_copy.weight.clone()).to(self.device)
self.node_representations = nn.Embedding.from_pretrained(self.node_representations_copy.weight.clone()).to(self.device)
```

forward

```
class DyGNN(nn.Module):  
  
    for i, interactions in enumerate(data_loader):  
        # interactions: [src, dst, time_diff]  
        # Compute and print loss.  
        loss = dyGnn.loss(interactions)  
  
    def loss(self, interactions):  
  
        output_rep_head_tensor, output_rep_tail_tensor, head_neg_tensors, tail_neg_tensors = self.forward(interactions)
```

def forward(self, interactions):

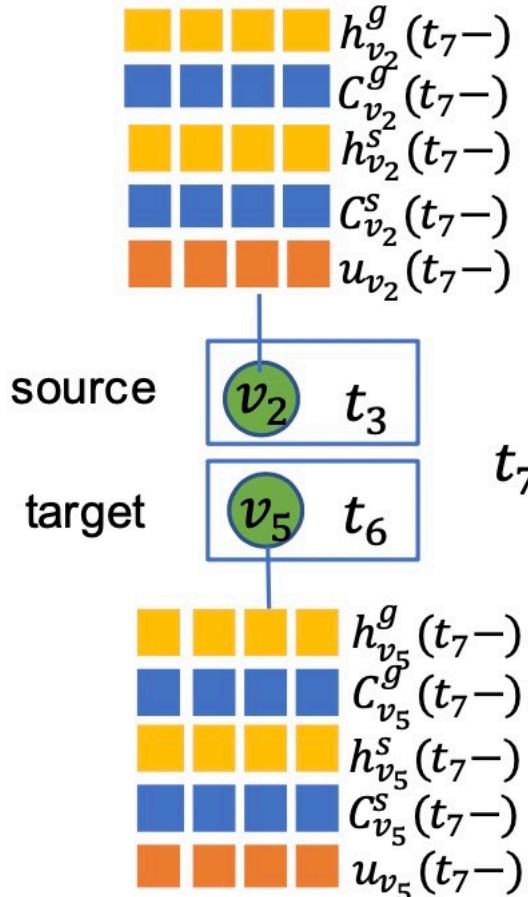
head\_node\_embedding

定义了几个embedding

```
head_node_rep = self.node_representations(head_inx_lt) # head的embedding  
head_node_cell_head = self.cell_head(head_inx_lt) # cell_head的embedding  
head_node_hidden_head = self.hidden_head(head_inx_lt) # hidden的embedding  
head_node_hidden_tail = self.hidden_tail(head_inx_lt) # hidden_tail的embedding
```

```
tail_node_rep = node2rep[tail_index] # tail的embedding  
tail_node_cell_tail = self.cell_tail(tail_inx_lt) # cell_tail的embedding  
tail_node_hidden_tail = self.hidden_tail(tail_inx_lt) # hidden_tail的embedding  
tail_node_hidden_head = self.hidden_head(tail_inx_lt) # hidden_head的embedding
```

(a)



```
head_node_hidden_tail = self.hidden_tail(head_inx_lt) # hidden_tail的embedding  
head_node_cell_tail
```

```
head_node_hidden_head = self.hidden_head(head_inx_lt) # hidden_head的embedding
```

```
head_node_cell_head = self.cell_head(head_inx_lt) # cell_head的embedding
```

```
head_node_rep = self.node_representations(head_inx_lt) # head的embedding
```

```
tail_node_hidden_tail = self.hidden_tail(tail_inx_lt) # hidden_tail的embedding
```

```
tail_node_cell_tail = self.cell_tail(tail_inx_lt) # cell_tail的embedding
```

```
tail_node_hidden_head = self.hidden_head(tail_inx_lt) # hidden_head的embedding
```

```
tail_node_cell_head
```

```
tail_node_rep = self.node_representations(tail_inx_lt) # tail的embedding
```

```

head_delta_t = current_t - head_prev_t # 时间差, 计算delta Time; 本次发生时间-上次发生时间
tail_delta_t = current_t - tail_prev_t

transed_head_delta_t = self.decayer(head_delta_t)
transed_tail_delta_t = self.decayer(tail_delta_t)

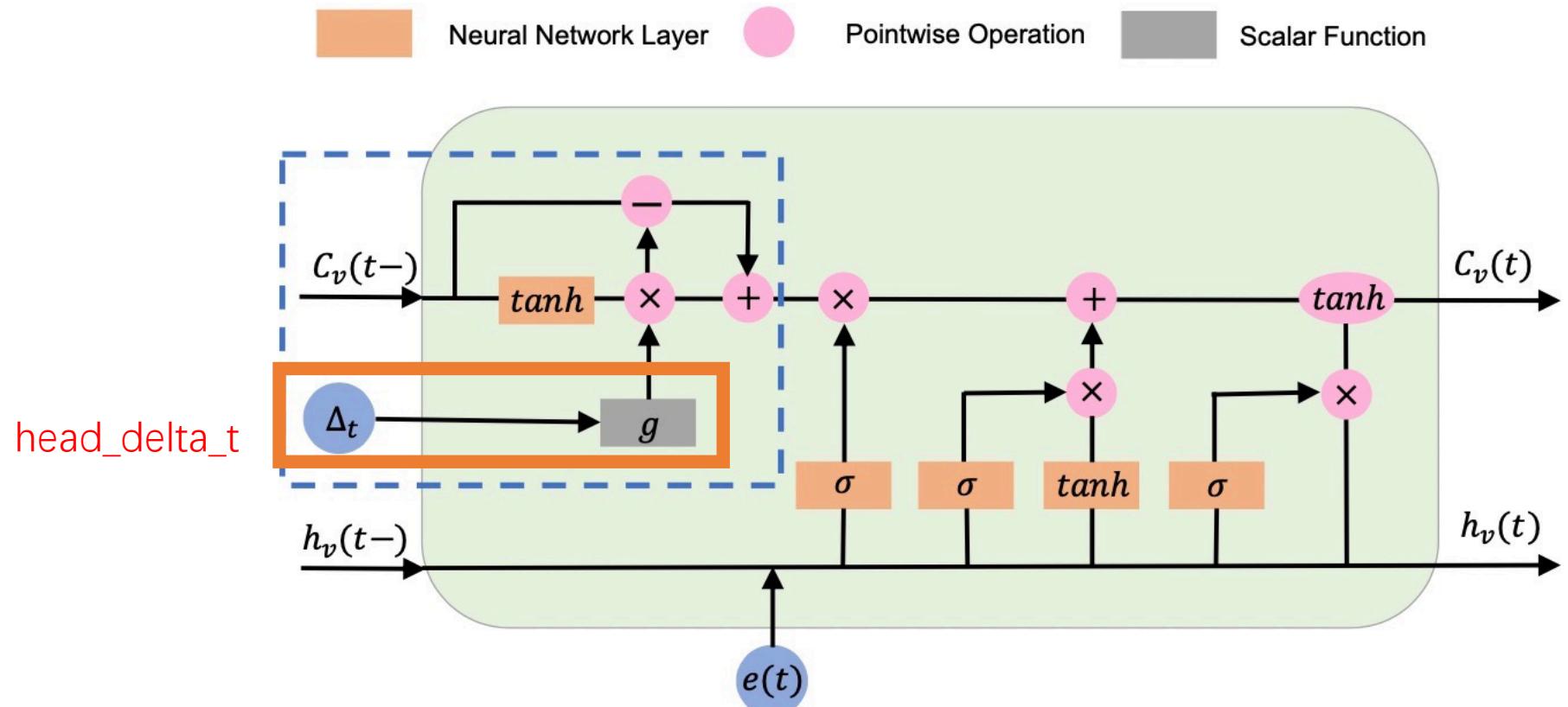
```

$$g(\Delta t)$$

```

def log_decay(self, delta_t):
    return 1/torch.log(2.7183 + self.w*delta_t)

```



## 2.1.1 The interact unit.

针对每个节点

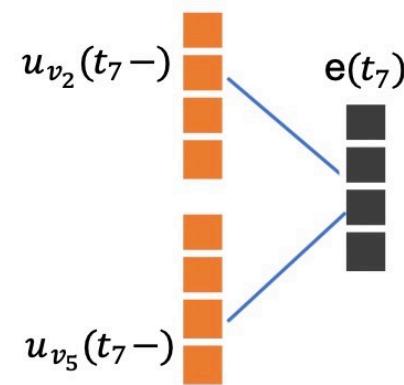
[1, 64]

```
edge_info_head = self.edge_updater_head(head_node_rep, tail_node_rep)  
edge_info_tail = self.edge_updater_tail(head_node_rep, tail_node_rep)
```

$e(t)$

$$e(t) = act(W_1 \cdot u_{v_s}(t-) + W_2 \cdot u_{v_g}(t-) + b_e) \quad (1)$$

(b)



$$\Delta_{t_s} = t_7 - t_3$$

$$\Delta_{t_g} = t_7 - t_6$$

## 2.1.2 The update unit.

`updated_head_node_cell_head, updated_head_node_hidden_head =`

`self.node_updater_head(edge_info_head, head_node_cell_head, head_node_hidden_head, transed_head_delta_t)`

S-update

上一步结果

(c)

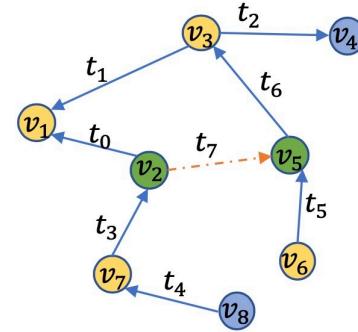
$C_{v_2}^s(t_7-), h_{v_2}^s(t_7-), e(t_7)$

**S-Update(**  ,  ,  **,**  $\Delta t_s$ )

$C_{v_2}^s(t_7), h_{v_2}^s(t_7)$

`updated_head_node_hidden_head`

`updated_head_node_cell_head`



$g(\Delta t)$

## 2.1.2 The update unit.

short term memory

`cell_short = self.c2s(cell)`

`cell_new = cell - cell_short + cell_short* transed_delta_t`

$$C_v^I(t-1) = \tanh(W_d \cdot C_v(t-1) + b_d) \quad (2)$$

$$\hat{C}_v^I(t-1) = C_v^I(t-1) * g(\Delta_t) \quad (3)$$

$$C_v^T(t-1) = C_v(t-1) - C_v^I(t-1) \quad (4)$$

$$C_v^*(t-1) = C_v^T(t-1) + \hat{C}_v^I(t-1) \quad (5)$$

`cell_new`

$$f_t = \sigma(W_f \cdot e(t) + U_f \cdot h_v(t-1) + b_f) \quad (6)$$

$$i_t = \sigma(W_i \cdot e(t) + U_i \cdot h_v(t-1) + b_i) \quad (7)$$

$$o_t = \sigma(W_o \cdot e(t) + U_o \cdot h_v(t-1) + b_o) \quad (8)$$

$$\tilde{C}_v(t) = \tanh(W_c \cdot e(t) + U_c \cdot h_v(t-1) + b_c) \quad (9)$$

$$C_v(t) = f_t * C_v^*(t-1) + i_t * \tilde{C}_v(t) \quad (10)$$

$$h_v(t) = o_t * \tanh(C_v(t)) \quad (11)$$

```
self.i2h = nn.Linear(input_size, 4*hidden_size, bias) # update中LSTM参数
self.h2h = nn.Linear(hidden_size, 4*hidden_size, bias)
```

参数合并在一起，所以为4倍参数

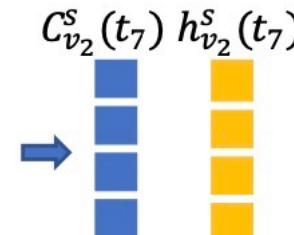
LSTM:

$e(t)$

`gates = self.i2h(input) + self.h2h(hidden)`  
`ingate, forgate, cellgate, outgate = gates.chunk(4, 1)`

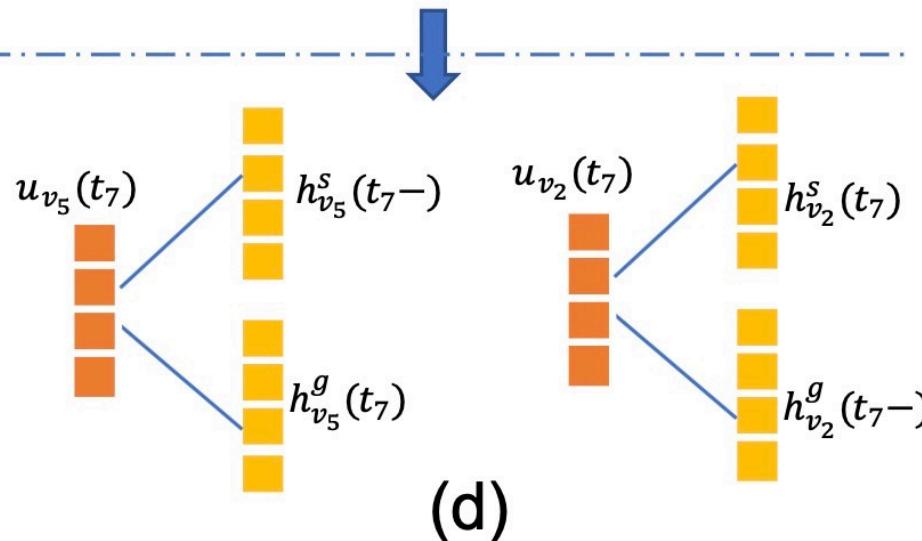
`ingate = self.sigmoid(ingate)`  
`forgate = self.sigmoid(forgate)`  
`cellgate = self.tanh(cellgate)`  
`outgate = self.sigmoid(outgate)`

`cell_output = forgate*cell_new + ingate*cellgate`  
`hidden_output = outgate*self.tanh(cell_output)`



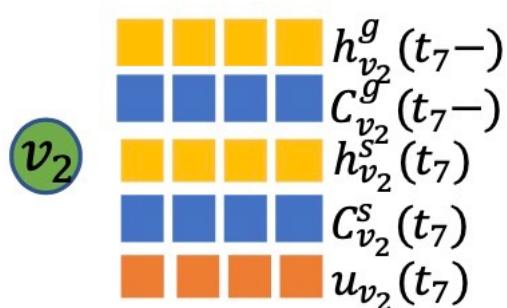
### 2.1.3 The merge unit.

```
updated_head_node_rep = self.combiner(updated_head_node_hidden_head, head_node_hidden_tail)
```



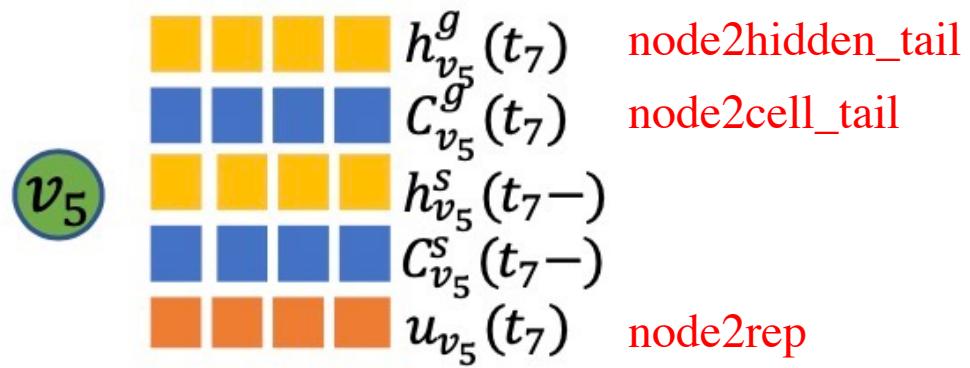
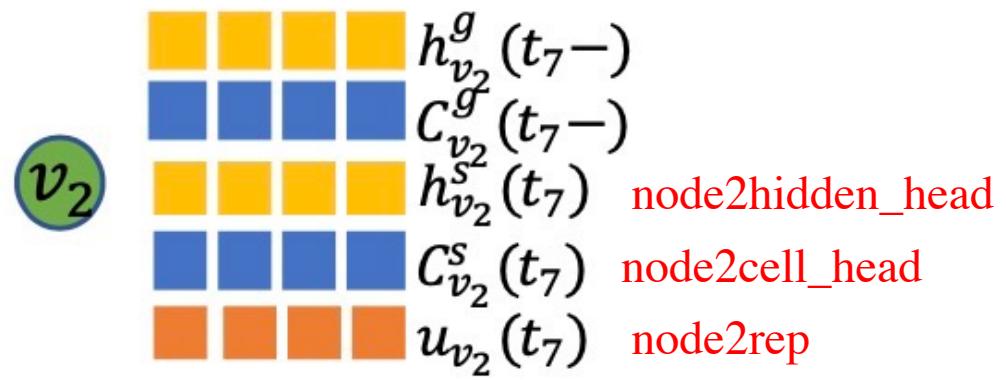
```
def forward(self, head_info, tail_info): self: Combiner(\n    node_output = self.h2o(head_info) + self.l2o(tail_info)\n    node_output_tanh = self.act(node_output)\n    return node_output_tanh
```

$$u_{v_s}(t) = W^s \cdot h_{v_s}^s(t) + W^g \cdot h_{v_s}^g(t-) + b_u \quad (13)$$



# 更新节点的embedding

```
node2cell_head[head_index] = updated_head_node_cell_head\nnode2hidden_head[head_index] = updated_head_node_hidden_head\nnode2rep[head_index] = updated_head_node_rep
```



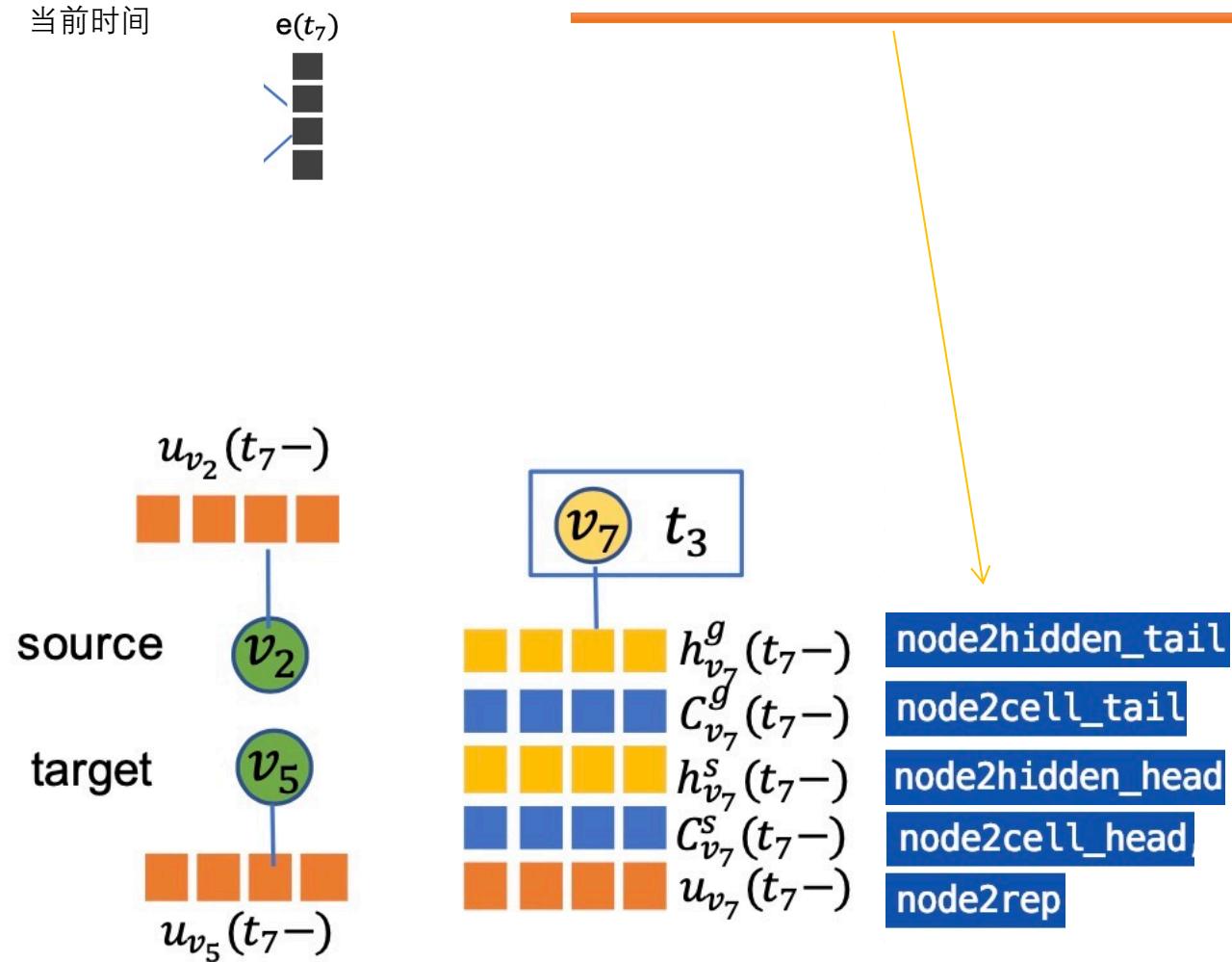
## **2.2 The propagation component**

```
head_node_head_neighbors, head_node_tail_neighbors =
```

```
    self.propagation(
```

```
(head_index, current_t, edge_info_head, 'head', node2cell_head, node2hidden_head, node2cell_tail, node2hidden_tail, node2rep, self.threshold, self.second_order)
```

节点索引 当前时间

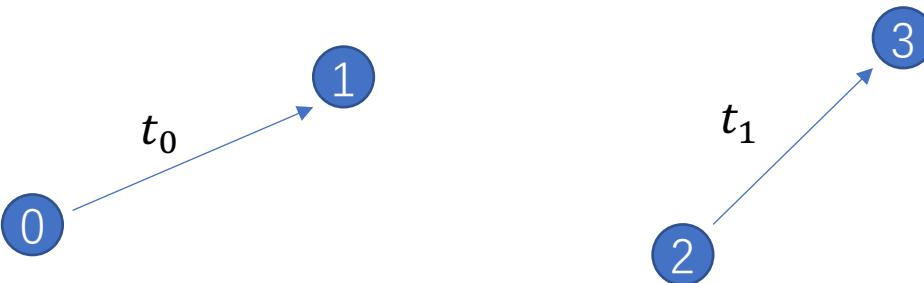


```
if threshold is not None: # 满足时间的 $\tau$ 
```

```
head_inx_th = (current_t.item() - head_timestamps) <= threshold  
head_neighbors = head_neighbors[head_inx_th]  
head_timestamps = head_timestamps[head_inx_th]
```

```
tail_inx_th = (current_t.item() - tail_timestamps) <= threshold  
tail_timestamps = tail_timestamps[tail_inx_th]  
tail_neighbors = tail_neighbors[tail_inx_th]
```

$$h(\Delta_t^s) = \begin{cases} 1, & \Delta_t^s \leq \tau, \\ 0, & \text{otherwise.} \end{cases}$$

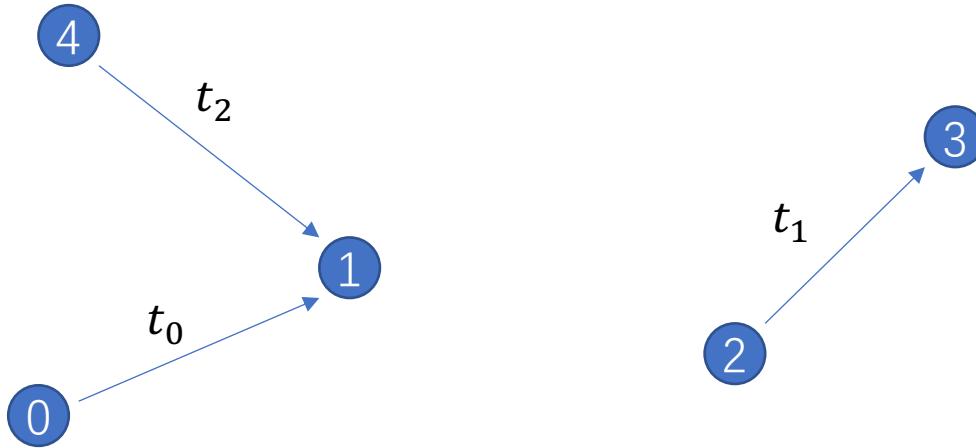
$t_1$ 
 $\text{tail\_candidates} = \text{all\_tail\_nodes} - \{\text{head\_index}, \text{tail\_index}\} - \text{head\_node\_tail\_neighbors}$ 

### Propagation components

 $\text{head\_candidates} = \text{all\_head\_nodes} - \{\text{tail\_index}, \text{head\_index}\} - \text{tail\_node\_head\_neighbors}$ 

time	head_index	tail_index	all_head_nodes	all_tail_nodes	timestamp	tail_candidates	head_candidates
$t_0$	0	1	{0}	{1}	0	{0}	{1}
$t_1$	2	3	{0,2}	{1,3}	31.91	{1}	{0}
$t_2$	4	1					
$t_3$	5	6					
$t_4$	7	6					

$t_1$       head\_neg\_samples = [0, 0, 0, 0, 0]  
               tail\_neg\_samples = [1, 1, 1, 1, 1]

$t_2$ 

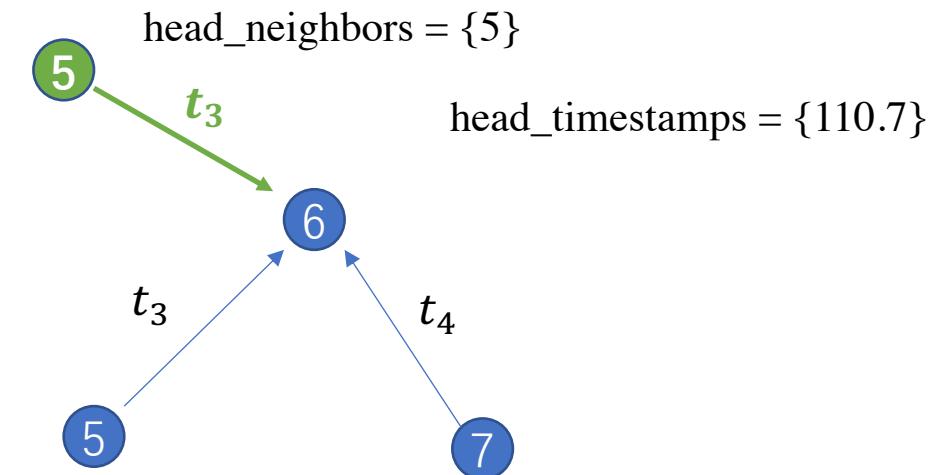
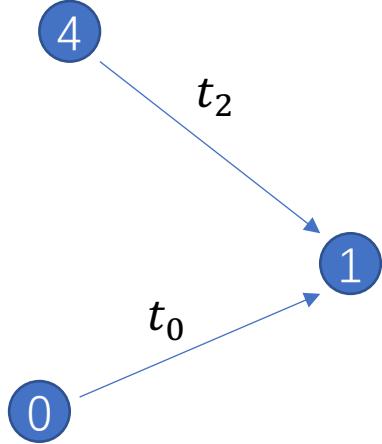
`tail_candidates = all_tail_nodes - {head_index,tail_index} - head_node_tail_neighbors`

**Propagation components**

`head_candidates = all_head_nodes - {tail_index,head_index} - tail_node_head_neighbors`

time	head_index	tail_index	all_head_nodes	all_tail_nodes	timestamp	tail_candidates	head_candidates
$t_0$	0	1	{0}	{1}	0	{0}	{1}
$t_1$	2	3	{0,2}	{1,3}	31.91	{1}	{0}
$t_2$	4	1	{0,2,4}	{1,3}	103.7	{3}	{0,2}
$t_3$	5	6					
$t_4$	7	6					

$t_1$       `head_neg_samples = {0,2}`  
 $t_1$       `tail_neg_samples = {3}`

$t_4$ 

tail\_candidates = all\_tail\_nodes - {head\_index, tail\_index} - head\_node\_tail\_neighbors

**Propagation components**

head\_candidates = all\_head\_nodes - {tail\_index, head\_index} - tail\_node\_head\_neighbors

time	head_index	tail_index	all_head_nodes	all_tail_nodes	timestamp	tail_candidates	head_candidates
$t_0$	0	1	{0}	{1}	0	{0}	{1}
$t_1$	2	3	{0,2}	{1,3}	31.91	{1}	{0}
$t_2$	4	1	{0,2,4}	{1,3}	103.7	{3}	{0,2}
$t_3$	5	6	{0,2,4,5}	{1,3,6}	110.73	{1,3}	{0,2,4}
$t_4$	7	6			110.77		

 $t_4$ 

head\_neg\_samples = {0,2,4}  
tail\_neg\_samples = {1,3}

## Propagation components

$$\hat{W}_s^s \cdot e(t)$$

```
head_nei_edge_info = self.tran_tail_edge_head(edge_info)
```

# 计算时间差; 当前时间和这个节点最近出现的时间;

$$\Delta_t^s = t_4 - t_3$$

```
head_delta_ts = current_t.repeat(len(head_timestamps), 1) - torch.FloatTensor(head_timestamps).to(self.device).view(-1, 1)
```

```
transed_head_delta_ts = self.decayer(head_delta_ts)
```

$$g(\Delta_t^s)$$

```
head_nei_cell = self.get_rep(head_neighbors, 'cell_head', node2cell_head) # 获取节点embedding
```

```
tran_head_nei_edge_info = head_nei_edge_info.repeat(len(head_neighbors), 1) * transed_head_delta_ts
```

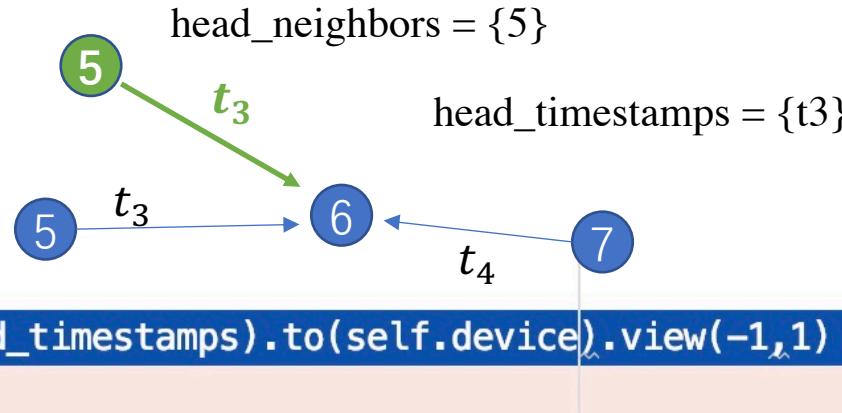
$$g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t)$$

$$\hat{W}_s^s \cdot e(t)$$

$$h(\Delta_t^s)$$

$$g(\Delta_t^s)$$

在处理邻居节点时, 已经选择了



$$C_{v_x}^s(t) = C_{v_x}^s(t-) + f_a(u_{v_x}(t-), u_{v_s}(t-)) \cdot g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t) \quad (15)$$

$$h_{v_x}^s(t) = \tanh(C_{v_x}^s(t)) \quad (16)$$

```
att_score_head = self.get_att_score(node, head_neighbors, node2rep) # attention系数
```

In[60]: `self.bilinear(node1, node2)`

Out[60]: `tensor([[-1.9639]], grad_fn=<AddBackward0>)`  
[64,64]

In[61]: `(torch.mul(torch.mm(node1, self.bilinear.weight[0]), node2)).sum(axis=1) + self.bilinear.bias`  
Out[61]: `tensor([-1.9639], grad_fn=<AddBackward0>)`

`(torch.mul(torch.mm(node1, self.bilinear.weight[0]), node2)).sum(axis=1) + self.bilinear.bias`

$$f_a(u_{v_x}(t-), u_{v_s}(t-)) = \frac{\exp(u_{v_x}(t-)^T u_{v_s}(t-))}{\sum_{v \in N^s(v_s)} \exp(u_v(t-)^T u_{v_s}(t-))} \quad (17)$$

$$f_a(u_{v_x}(t-), u_{v_s}(t-)) \cdot g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t)$$

In[64]: `torch.mul(node1, node2).sum(axis=1)`

Out[64]: `tensor([1.2072], grad_fn=<SumBackward1>)`

```
m = nn.Bilinear(20, 30, 40)
input1 = torch.randn(128, 20)
input2 = torch.randn(128, 30)
output = m(input1, input2) [128, 40]
```

```
weight = m.weight.data.cpu().numpy() # [40, 20, 30]
bias = m.bias.data.cpu().numpy() # [40, ]
x1 = input1.data.cpu().numpy() # [128, 20]
x2 = input2.data.cpu().numpy() # [128, 30]
```

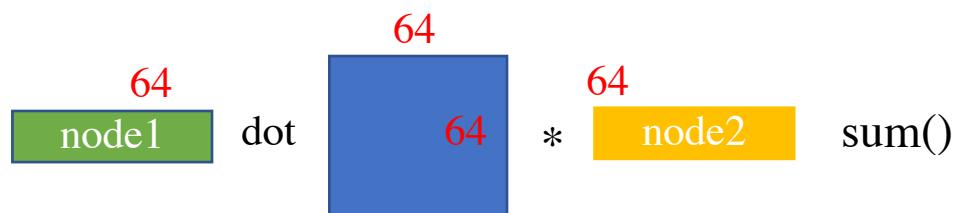
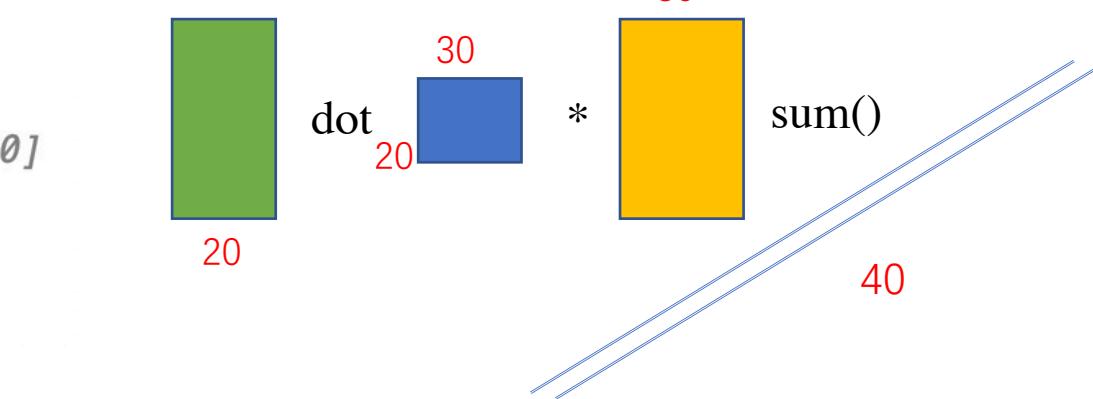
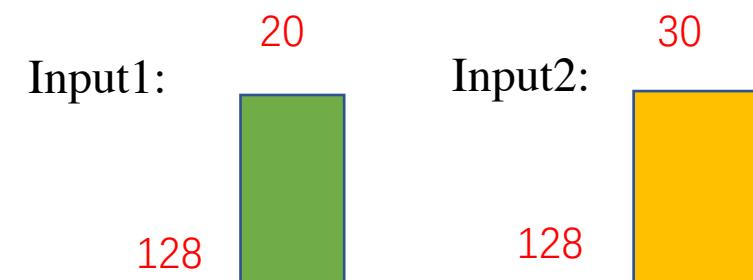
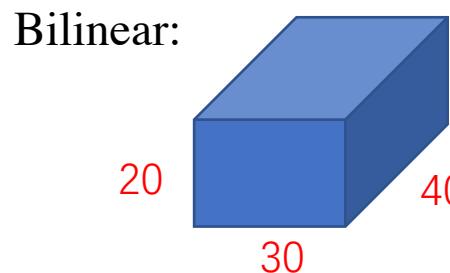
```
for k in range(weight.shape[0]): # 40
    buff = np.dot(x1, weight[k]) # [128,20] * [20,30] => [128,30]
    buff = buff * x2 # [128,30] * [128,30]
    buff = np.sum(buff, axis=1) # [128, ]
    y[:,k] = buff
[128, 40]
```

```
self.bilinear = nn.Bilinear(embedding_dims, embedding_dims, 1) [64, 64, 1]
```

```
self.softmax(self.bilinear(node1, node2).view(-1,1))
[1, 64] [1, 64]
```

$$(node_1 W) * node_2$$

节点和节点相乘后求和



$$f_a(u_{v_x}(t-), u_{v_s}(t-)) = \frac{\exp(u_{v_x}(t-)^T u_{v_s}(t-))}{\sum_{v \in N^s(v_s)} \exp(u_v(t-)^T u_{v_s}(t-))} \quad (17)$$

```
tran_head_nei_edge_info = tran_head_nei_edge_info*att_score_head
```

[1, 64]

$$f_a(u_{v_x}(t-), u_{v_s}(t-)) \cdot g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t)$$

```
head_nei_cell = head_nei_cell + tran_head_nei_edge_info # 更新总的节点特征
```

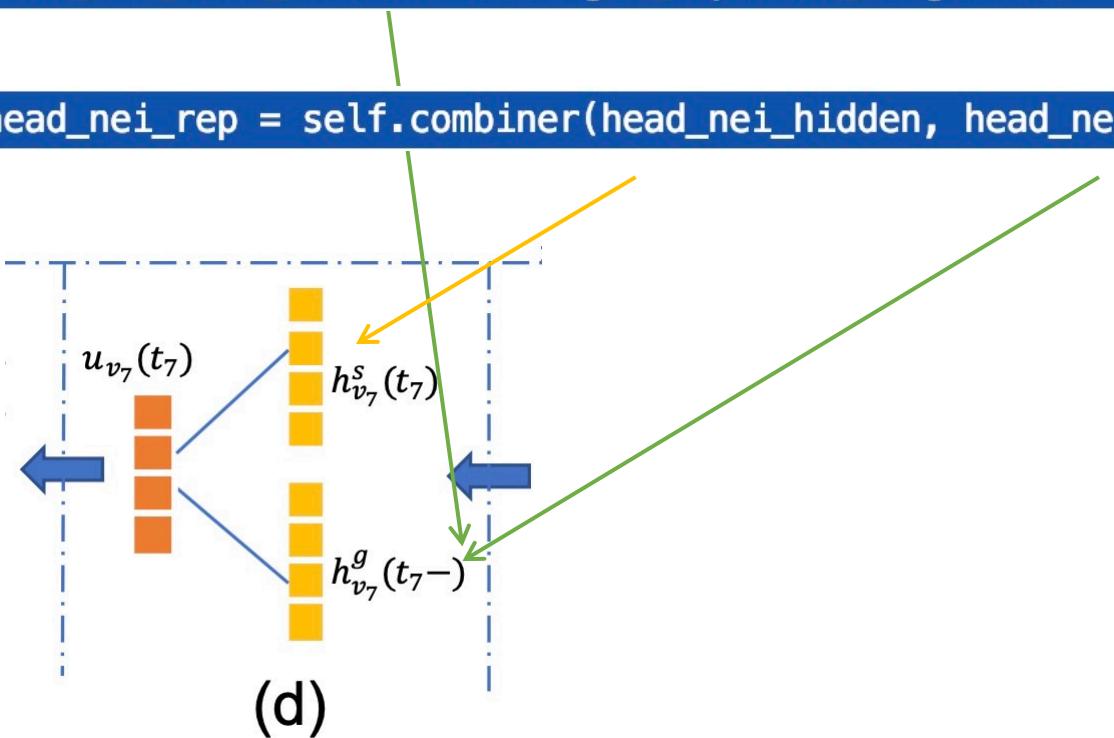
$$C_{v_x}^s(t) = C_{v_x}^s(t-) + f_a(u_{v_x}(t-), u_{v_s}(t-)) \cdot g(\Delta_t^s) \cdot h(\Delta_t^s) \cdot \hat{W}_s^s \cdot e(t) \quad (15)$$

$$h_{v_x}^s(t) = \tanh(C_{v_x}^s(t)) \quad (16)$$

# neighbors的hidden更新

```
head_nei_tail_hidden = self.get_rep(head_neighbors, 'hidden_tail', node2hidden_tail)
```

```
head_nei_rep = self.combiner(head_nei_hidden, head_nei_tail_hidden)
```

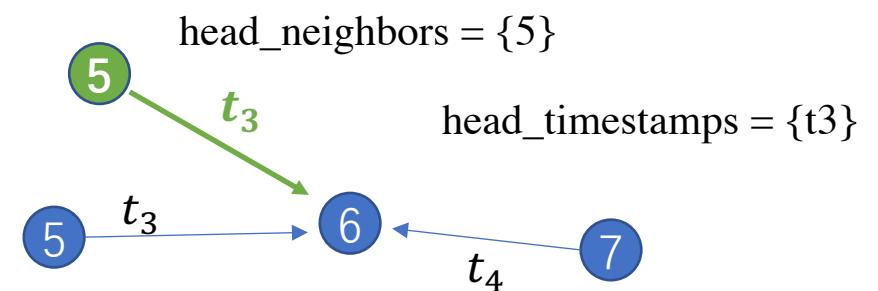
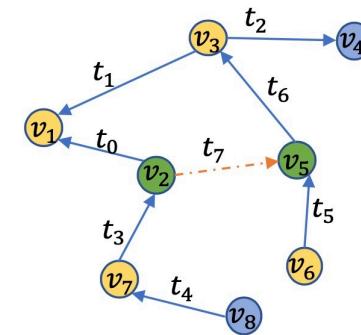
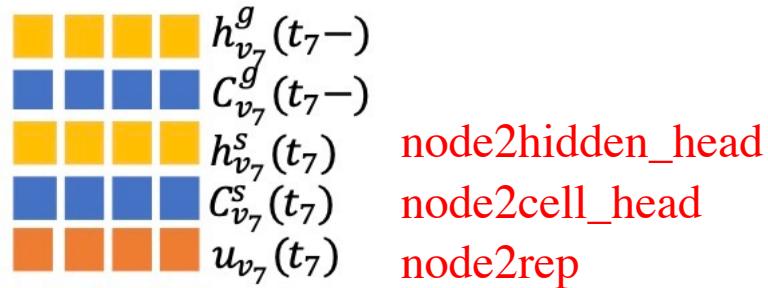


$$u_{v_s}(t) = W^s \cdot h_{v_s}^s(t) + W^g \cdot h_{v_s}^g(t-) + b_u$$

```

# 更新节点embedding
for i, nei in enumerate(head_neighbors): i: 0 nei: 5
    node2cell_head[nei] = head_nei_cell[i].view(-1, self.embedding_dims)
    node2hidden_head[nei] = head_nei_hidden[i].view(-1, self.embedding_dims)
    node2rep[nei] = head_nei_rep[i].view(-1, self.embedding_dims)

```



$t_4$ 

{0, 2, 4, 5, 7}

{ } head节点的head节点 {5}

tail节点的head节点

```
all_head_nodes = all_head_nodes | head_node_head_neighbors | tail_node_head_neighbors
```

```
all_tail_nodes = all_tail_nodes | head_node_tail_neighbors | tail_node_tail_neighbors
```

{1, 3, 6}

{ }

{ }

作为tail的候选

本次连接节点

指向tail的head节点

```
tail_candidates = all_tail_nodes - {head_index, tail_index} - head_node_tail_neighbors
```

{1, 3}

{1, 3, 6}

{6, 7}

作为head的候选

指向head的tail节点

```
head_candidates = all_head_nodes - {tail_index, head_index} - tail_node_head_neighbors
```

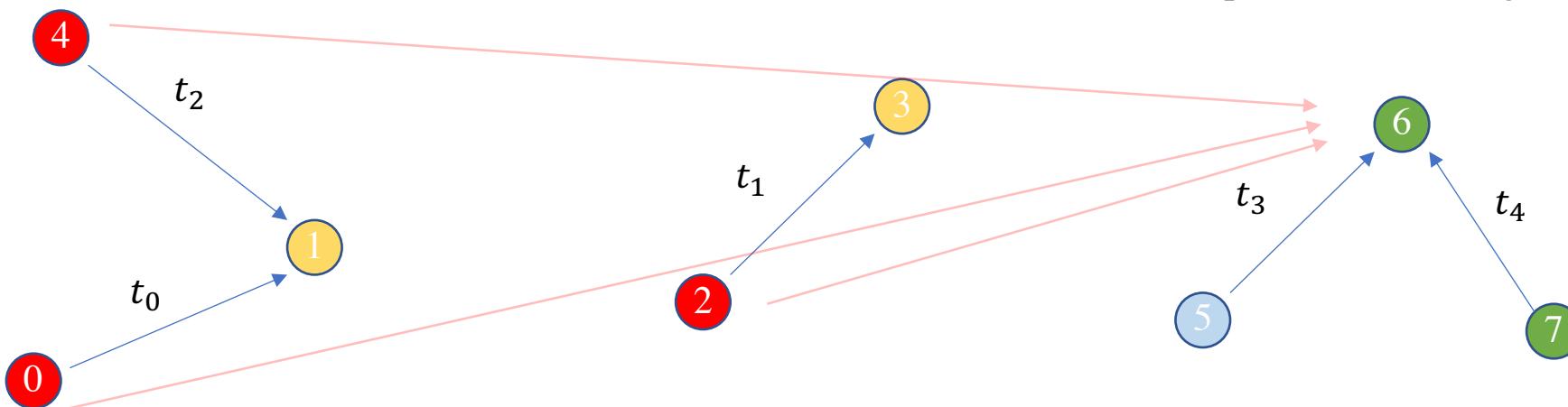
{0, 2, 4}

{0, 2, 4, 5, 7}

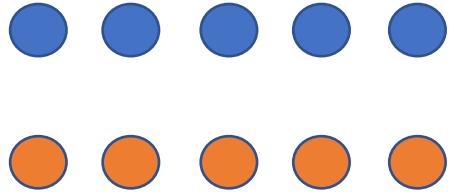
{6, 7}

{5}

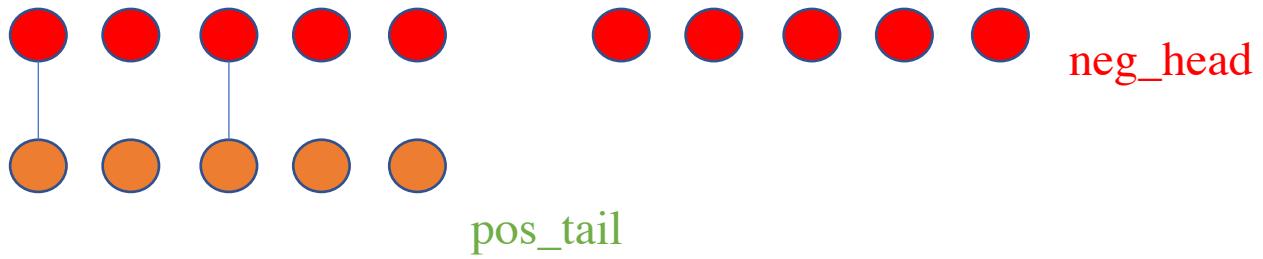
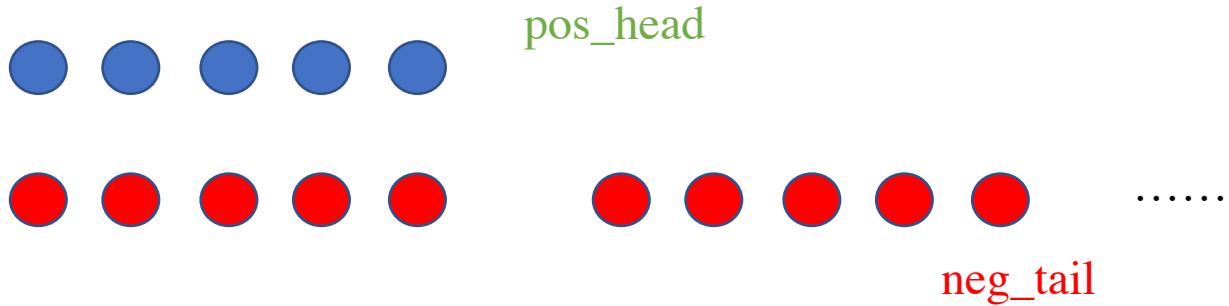
**负采样方式：**要计算所有的neg\_head {0,2,4}和pos\_tail{6}的连接  
 要计算所有的pos\_head{7}和neg\_tail{1,3}的连接



positive



negative



## 获取节点的各种属性值

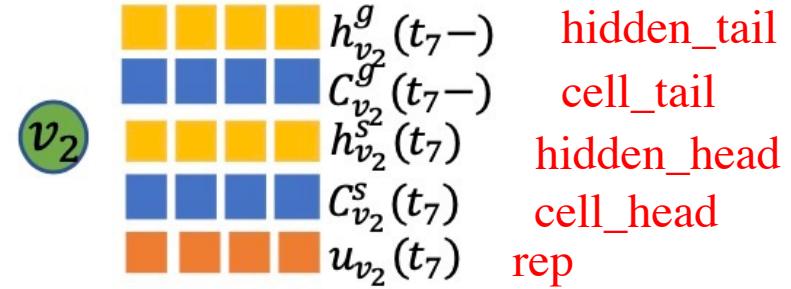
```
cell_head_inx = list(node2cell_head.keys())
output_cell_head = list(node2cell_head.values())

cell_tail_inx = list(node2cell_tail.keys())
output_cell_tail = list(node2cell_tail.values())

hidden_head_inx = list(node2hidden_head.keys())
output_hidden_head = list(node2hidden_head.values())

hidden_tail_inx = list(node2hidden_tail.keys())
output_hidden_tail = list(node2hidden_tail.values())

rep_inx = list(node2rep.keys())
output_rep = list(node2rep.values())
```



```
In[119]: cell_head_inx
Out[119]: [0, 2, 4, 5, 7]
In[120]: len(output_cell_head)
Out[120]: 5
In[121]: output_cell_head[0].shape
Out[121]: torch.Size([1, 64])
```

```
In[122]: cell_tail_inx
Out[122]: [1, 3, 6]
In[123]: len(output_cell_tail)
Out[123]: 3
In[124]: output_cell_tail[0].shape
Out[124]: torch.Size([1, 64])
```

## 样本特征投影

```
if self.transfer:  
    output_rep_head_tensor = self.dropout(self.transfer2head(output_rep_head_tensor))  
    output_rep_tail_tensor = self.dropout(self.transfer2tail(output_rep_tail_tensor))  
  
    head_neg_tensors = self.dropout(self.transfer2head(head_neg_tensors))  
    tail_neg_tensors = self.dropout(self.transfer2tail(tail_neg_tensors))  
  
return output_rep_head_tensor, output_rep_tail_tensor, head_neg_tensors, tail_neg_tensors
```

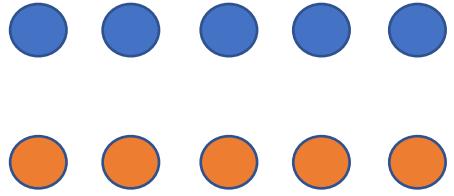
$$u_{v_s}^s(t-) = P^s \cdot u_{v_s}(t-)$$
$$u_{v_g}^g(t-) = P^g \cdot u_{v_g}(t-)$$

In[13]: `self.transfer2head`

Out[13]: `Linear(in_features=64, out_features=64, bias=False)`

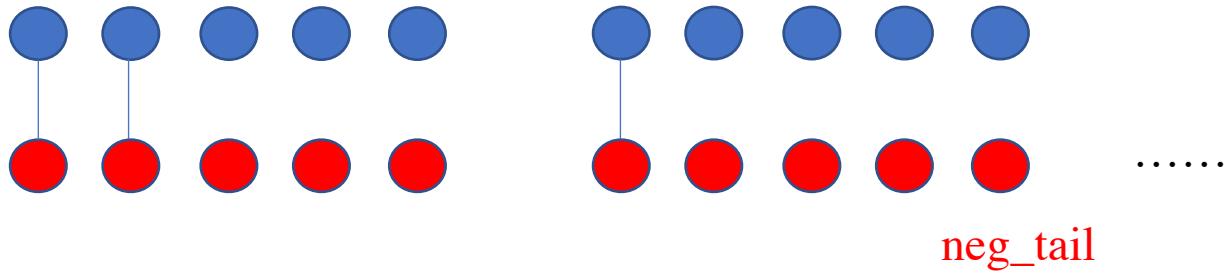
训练样本的特征

positive



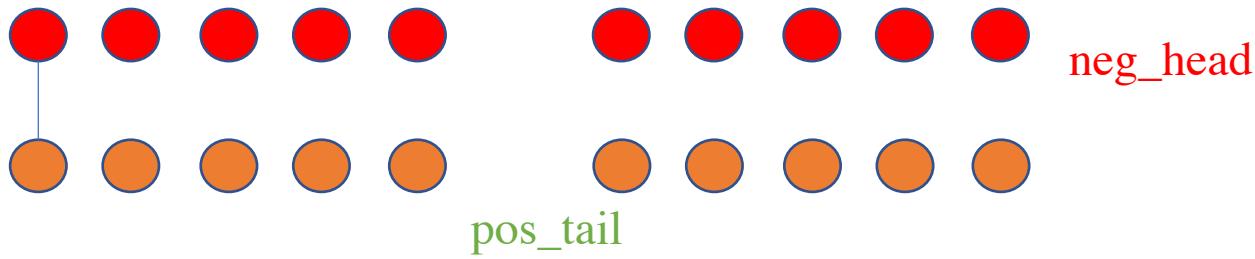
negative

pos\_head



neg\_tail

neg\_head



pos\_tail

Loss

# [5, 1, 64]

# [5, 64, 1]

```
scores_p = torch.bmm(output_rep_head_tensor.view(num_pp, 1, self.embedding_dims), output_rep_tail_tensor.view(num_pp, self.embedding_dims, 1))
```

$$J((v_s, v_g, t)) = -\log(\sigma(u_{v_s}^s(t-)^T u_{v_g}^g(t-))) \\ - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log(\sigma(u_{v_s}^s(t-)^T u_{v_n}^g(t-))) \quad (18)$$

1-

负采样节点

neg的head节点

pos的tail节点

```
scores_n_1 = torch.bmm(head_neg_tensors.view(num_pp * self.num_negative, 1, self.embedding_dims), tail_pos_tensors.view(num_pp * self.num_negative, self.embedding_dims, 1))
```

[25, 1, 64]

[25, 64, 1]

pos的head节点

neg的tail节点

```
scores_n_2 = torch.bmm(head_pos_tensors.view(num_pp * self.num_negative, 1, self.embedding_dims), tail_neg_tensors.view(num_pp * self.num_negative, self.embedding_dims, 1))
```

```
bce_with_logits_loss = nn.BCEWithLogitsLoss() bce_with_logits_loss: BCEWithLogitsLoss()
```

```
loss = bce_with_logits_loss(scores, labels) loss: tensor(1.1135, grad_fn=<BinaryCrossEntropyWithLogitsBackward>)
```

```
In[59]: (torch.log(torch.sigmoid(scores[:5])).sum() + torch.log(1-torch.sigmoid(scores[5:])).sum())/55
```

```
Out[59]: tensor(-1.1135, grad_fn=<DivBackward0>)
```

```
In[60]: (torch.log(torch.sigmoid(scores[:5])).sum() + torch.log(torch.sigmoid(scores[5:])).sum())/55
```

```
Out[60]: tensor(-0.9280, grad_fn=<DivBackward0>)
```

```
(torch.log(torch.sigmoid(scores[:5])).sum() + torch.log(1-torch.sigmoid(scores[5:])).sum())/55
```



# INDUCTIVE REPRESENTATION LEARNING ON TEMPORAL GRAPHS

## TGAT

temporal graph attention (TGAT)

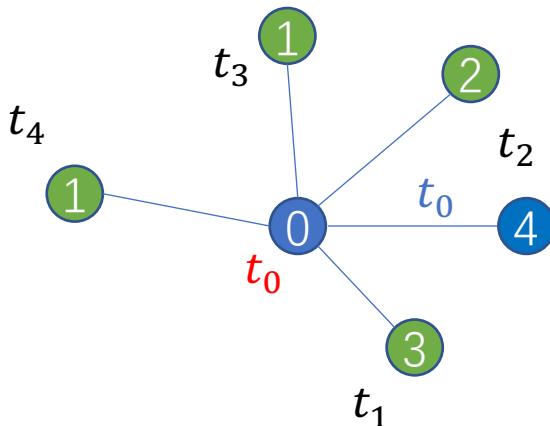
# TGAT : 基于RNN的连续性网络

Model type	Model name	Encoder	Link addition	Link deletion	Node addition	Node deletion	Network type
Discrete networks							
Stacked DGNN	GCRN-M1 [58]	Spectral GCN [59] & LSTM	Yes	Yes	No	No	Any
	WD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	CD-GCN [65]	Spectral GCN [57] & LSTM	Yes	Yes	No	No	Any
	RgCNN [61]	Spatial GCN [62] & LSTM	Yes	Yes	No	No	Any
	DyGGNN [63]	GGNN [64] & LSTM	Yes	Yes	No	No	Any
	DySAT [67]	GAT [68] & temporal attention from [69]	Yes	Yes	Yes	Yes	Any
Integrated DGNN	GCRN-M2 [58]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	GC-LSTM [72]	GCN [59] integrated in an LSTM	Yes	Yes	No	No	Any
	EvolveGCN [71]	LSTM integrated in a GCN [57]	Yes	Yes	Yes	Yes	Any
	LRGCN [73]	R-GCN [75] integrated in an LSTM	Yes	Yes	No	No	Any
	RE-Net [74]	R-GCN [75] integrated in several RNNs	Yes	Yes	No	No	Knowledge network
Continuous networks							
RNN based							
	Streaming GNN [86]	Node embeddings maintained by architecture consisting of T-LSTM [88]	Yes	No	Yes	No	Directed strictly evolving
	JODIE [87]	Node embeddings maintained by an RNN based architecture	Yes	No	No	No	Bipartite and interaction
TPP based							
	Know-Evolve [89]	TPP parameterised by an RNN	Yes	No	No	No	Interaction, knowledge network
	DyREP [36]	TPP parameterised by an RNN aided by structural attention	Yes	No	Yes	No	Strictly evolving
	LDG [90]	TPP, RNN and self-attention	Yes	No	Yes	No	Strictly evolving
	GHN [92]	TPP parameterised by a continuous time LSTM [93]	Yes	No	No	No	Interaction, knowledge network

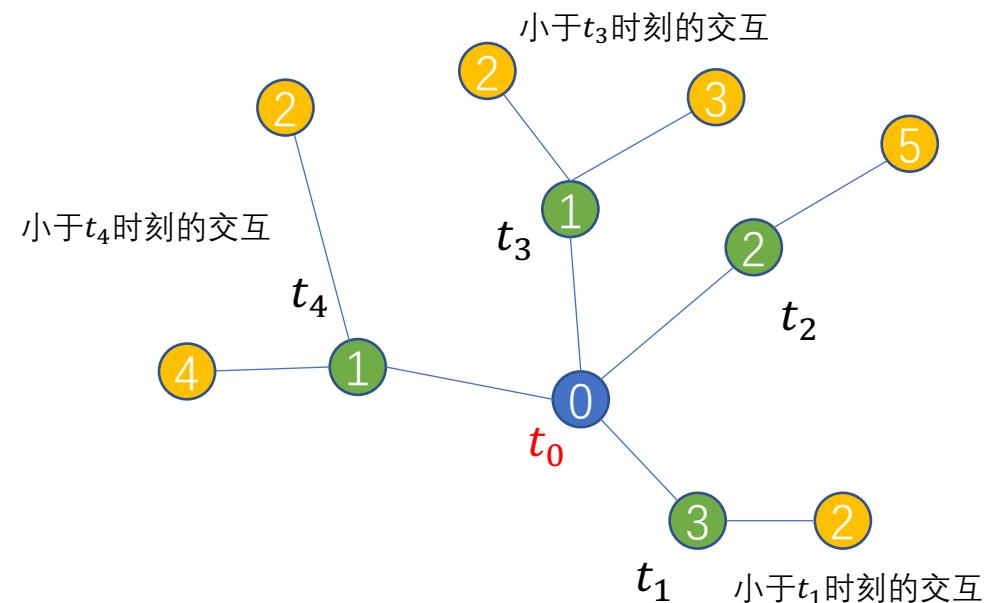
time	edge
$t_1$	0-3
$t_2$	0-2
$t_3$	0-1
$t_4$	0-1
$t_0$	0-4

$$t_0 > t_4 > t_3 > t_2 > t_1$$

$t_0$ 时刻，0-4发生交互  
求 $t_0$ 时刻节点0的邻居信息

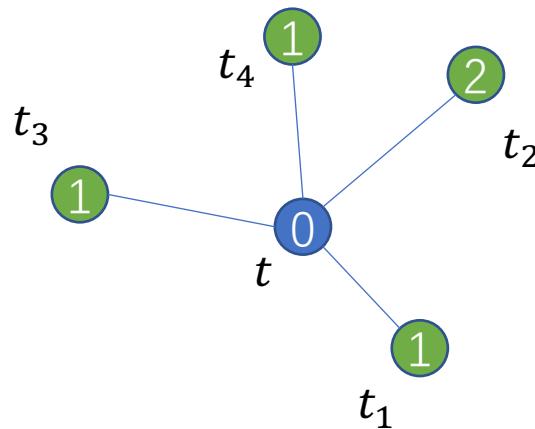


## 两层TGAT



第1层： $\{4,2\} \rightarrow 1$   $\{2,3\} \rightarrow 1$  .....  $\{1,1,2,3\} \rightarrow 0$   
第2层： $\{1,1,2,3\} \rightarrow 0$

$$t > t_4 > t_3 > t_2 > t_1$$



TGAT聚合不仅仅考虑邻居节点特征,  
同时也考虑了时间特征

节点0的上一层的特征

$\mathbf{Z}(t) = [\tilde{\mathbf{h}}_0^{(l-1)}(t) \parallel \Phi_{d_T}(0), \tilde{\mathbf{h}}_1^{(l-1)}(t_1) \parallel \Phi_{d_T}(t - t_1), \dots, \tilde{\mathbf{h}}_N^{(l-1)}(t_N) \parallel \Phi_{d_T}(t - t_N)]^\top$

最近的一次时间交互

目标节点

时间差特征

$t - t = 0$ ; 时间差为0

邻居节点

$\Phi_d(t) = [cos(\omega_1 t + \theta_1), cos(\omega_2 t + \theta_2), \dots, cos(\omega_d t + \theta_d)]$

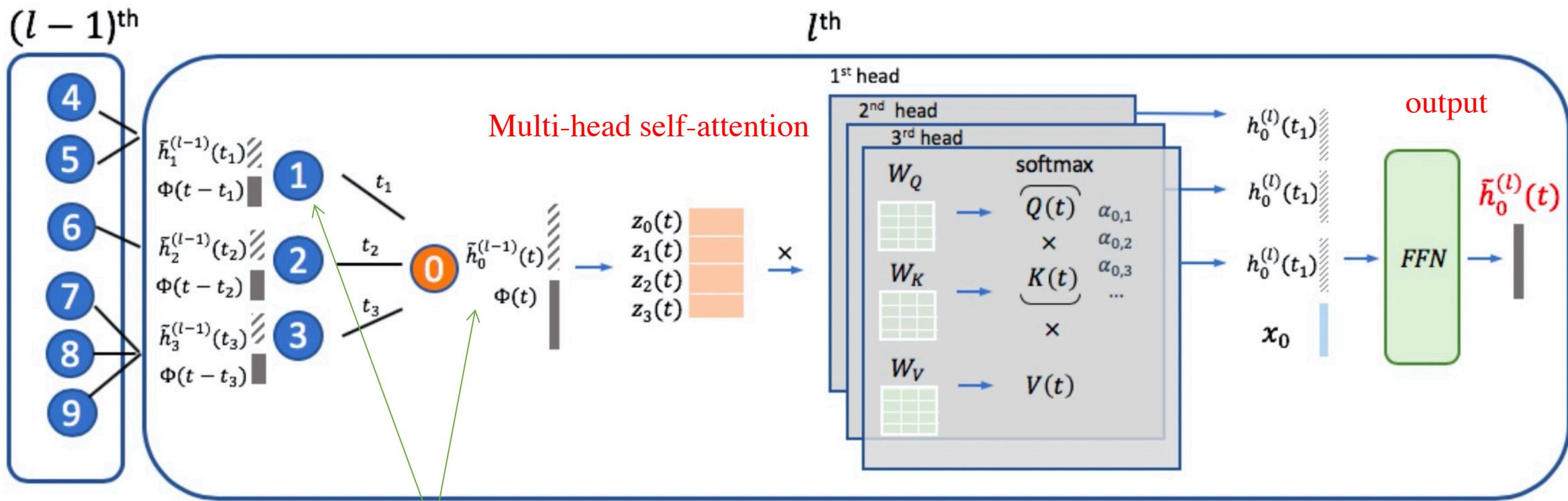


Figure 2: The architecture of the  $l^{th}$  TGAT layer with  $k = 3$  attention heads for node  $v_0$  at time  $t$ .

节点0的上一层的特征

$$\mathbf{Z}(t) = [\tilde{\mathbf{h}}_0^{(l-1)}(t) || \Phi_{d_T}(0), \tilde{\mathbf{h}}_1^{(l-1)}(t_1) || \Phi_{d_T}(t - t_1), \dots, \tilde{\mathbf{h}}_N^{(l-1)}(t_N) || \Phi_{d_T}(t - t_N)]^T$$

时间差特征

时间差为0

$$\Phi_d(t) = [cos(\omega_1 t + \theta_1), cos(\omega_2 t + \theta_2), \dots, cos(\omega_d t + \theta_d)]$$

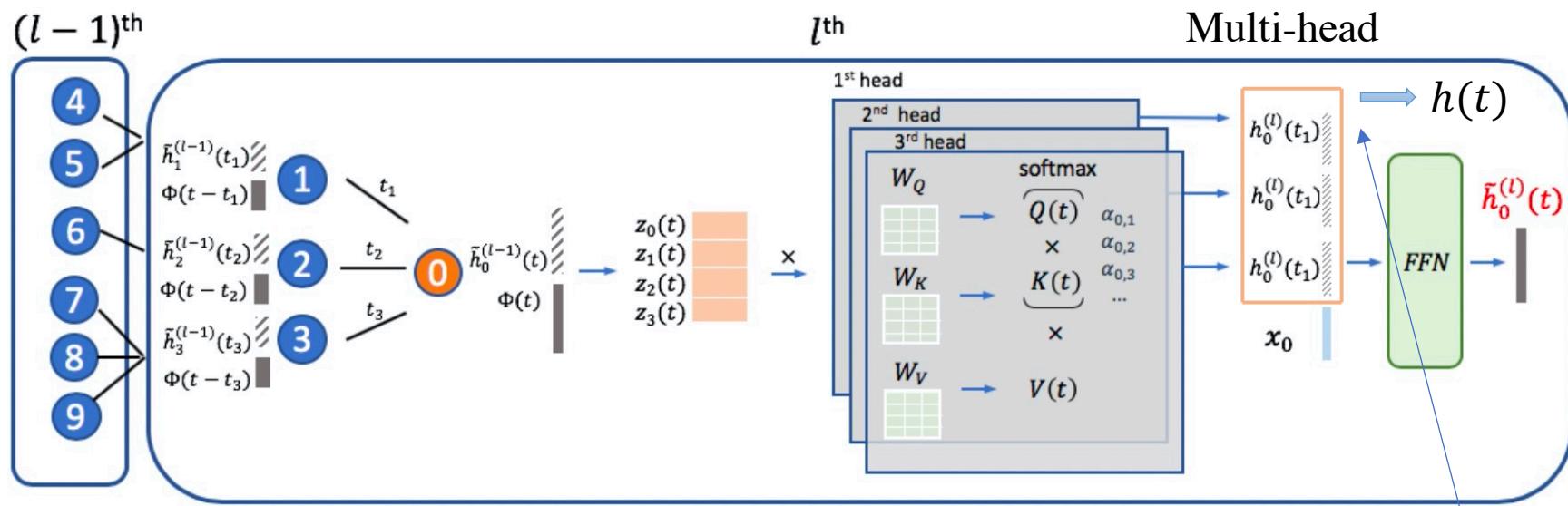
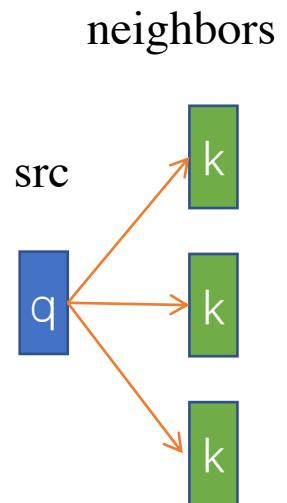


Figure 2: The architecture of the  $l^{th}$  TGAT layer with  $k = 3$  attention heads for node  $v_0$  at time  $t$ .

$$\mathbf{Z}(t) = \left[ \tilde{\mathbf{h}}_0^{(l-1)}(t) || \Phi_{d_T}(0), \tilde{\mathbf{h}}_1^{(l-1)}(t_1) || \Phi_{d_T}(t - t_1), \dots, \tilde{\mathbf{h}}_N^{(l-1)}(t_N) || \Phi_{d_T}(t - t_N) \right]^\top$$



$$\mathbf{q}(t) = [\mathbf{Z}(t)]_0 \mathbf{W}_Q, \mathbf{K}(t) = [\mathbf{Z}(t)]_{1:N} \mathbf{W}_K, \mathbf{V}(t) = [\mathbf{Z}(t)]_{1:N} \mathbf{W}_V$$

src 节点和neighbors 聚合后结果 :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\text{temperature}}\right)V$$

$$\tilde{\mathbf{h}}_0^{(l)}(t) = \text{FFN}\left(\mathbf{h}(t) || \mathbf{x}_0\right) \equiv \text{ReLU}\left([\mathbf{h}(t) || \mathbf{x}_0]\mathbf{W}_0^{(l)} + \mathbf{b}_0^{(l)}\right)\mathbf{W}_1^{(l)} + \mathbf{b}_1^{(l)}$$

节点0的上一层的特征

$$\mathbf{Z}(t) = \left[ \tilde{\mathbf{h}}_0^{(l-1)}(t) || \Phi_{d_T}(0), \tilde{\mathbf{h}}_1^{(l-1)}(t_1) || \Phi_{d_T}(t - t_1), \dots, \tilde{\mathbf{h}}_N^{(l-1)}(t_N) || \Phi_{d_T}(t - t_N) \right]^\top$$

时间差特征

考虑边的信息进行聚合

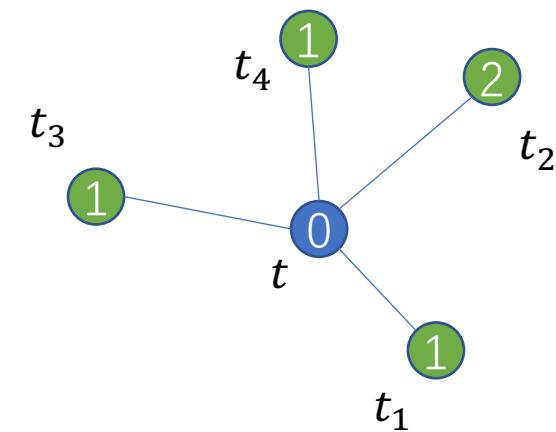
节点0的上一层的特征                          时间差特征

$$\mathbf{Z}(t) = \left[ \dots, \tilde{\mathbf{h}}_i^{(l-1)}(t_i) || \mathbf{x}_{0,i}(t_i) || \Phi_{d_T}(t - t_i), \dots \right] \text{ (or use summation)}, \quad (7)$$

边的特征

neighbors节点边特征为src和neighbor连接的边特征

src节点的边的特征为0



crossentropy

$$\ell = \sum_{(v_i, v_j, t_{ij}) \in \mathcal{E}} -\log \left( \sigma(\textcolor{red}{+} \tilde{\mathbf{h}}_i^l(t_{ij})^\top \tilde{\mathbf{h}}_j^l(t_{ij})) \right) - Q \cdot \mathbb{E}_{v_q \sim P_n(v)} \log \left( \sigma(\textcolor{red}{-} \tilde{\mathbf{h}}_i^l(t_{ij})^\top \tilde{\mathbf{h}}_q^l(t_{ij})) \right), \quad (8)$$

正常连接的边

负采样的边

Dataset	Reddit		Wikipedia		Industrial	
	Metric	Accuracy	AP	Accuracy	AP	Accuracy
GAE	74.31 (0.5)	93.23 (0.3)	72.85 (0.7)	91.44 (0.1)	68.92 (0.3)	81.15 (0.2)
VAGE	74.19 (0.4)	92.92 (0.2)	78.01 (0.3)	91.34 (0.3)	67.81 (0.4)	80.87 (0.3)
DeepWalk	71.43 (0.6)	83.10 (0.5)	76.67 (0.5)	90.71 (0.6)	65.87 (0.3)	80.93 (0.2)
Node2vec	72.53 (0.4)	84.58 (0.5)	78.09 (0.4)	91.48 (0.3)	66.64 (0.3)	81.39 (0.3)
CTDNE	73.76 (0.5)	91.41 (0.3)	79.42 (0.4)	92.17 (0.5)	67.81 (0.3)	80.95 (0.5)
GAT	92.14 (0.2)	97.33 (0.2)	87.34 (0.3)	94.73 (0.2)	69.58 (0.4)	81.51 (0.2)
GAT+T	92.47 (0.2)	97.62 (0.2)	87.57 (0.2)	95.14 (0.4)	70.15 (0.3)	82.66 (0.4)
GraphSAGE	92.31(0.2)	97.65 (0.2)	85.93 (0.3)	93.56 (0.3)	70.19 (0.2)	83.27 (0.3)
GraphSAGE+T	<u>92.58</u> (0.2)	<u>97.89</u> (0.3)	86.31 (0.3)	93.72 (0.3)	<u>71.84</u> (0.3)	<u>84.95</u> (0.)
Const-TGAT	91.39 (0.2)	97.86 (0.2)	86.03 (0.4)	93.50 (0.3)	68.52 (0.2)	81.91 (0.3)
TGAT	<b>92.92</b> (0.3)	<b>98.12</b> (0.2)	<b>88.14</b> (0.2)	<b>95.34</b> (0.1)	<b>73.28</b> (0.2)	<b>86.32</b> (0.1)

直推式任务

Dataset Metric	Reddit		Wikipedia		Industrial	
	Accuracy	AP	Accuracy	AP	Accuracy	AP
GAT	89.86 (0.2)	95.37 (0.3)	82.36 (0.3)	91.27 (0.4)	68.28 (0.2)	79.93 (0.3)
GAT+T	<u>90.44</u> (0.3)	<u>96.31</u> (0.3)	<u>84.82</u> (0.3)	<u>93.57</u> (0.3)	<u>69.51</u> (0.3)	<u>81.68</u> (0.3)
GraphSAGE	89.43 (0.1)	96.27 (0.2)	82.43 (0.3)	91.09 (0.3)	67.49 (0.2)	80.54 (0.3)
GraphSAGE+T	90.07 (0.2)	95.83 (0.2)	84.03 (0.4)	92.37 (0.5)	69.66 (0.3)	82.74 (0.3)
Const-TGAT	88.28 (0.3)	94.12 (0.2)	83.60 (0.4)	91.93 (0.3)	65.87 (0.3)	77.03 (0.4)
TGAT	<b>90.73</b> (0.2)	<b>96.62</b> (0.3)	<b>85.35</b> (0.2)	<b>93.99</b> (0.3)	<b>72.08</b> (0.3)	<b>84.99</b> (0.2)

Table 2: Inductive learning task results for predicting future edges of unseen nodes.

未知节点上的推理任务



Code

导入数据：

```
### Load data and train val test split  
[157474, 6] g_df = pd.read_csv('./processed/ml_{}.csv'.format(DATA)) g_df:  
[157475, 172] e_feat = np.load('./processed/ml_{}.npy'.format(DATA)) e_feat: [[ 0.  
[9228, 172] n_feat = np.load('./processed/ml_{}_node.npy'.format(DATA)) n_feat:
```

In[7]: g\_df.head()

```
Out[7]:      src    dst    time  
           Unnamed: 0   u     i      ts  label  idx  
0             0   1  8228    0.0      0   0   1  
1             1   2  8229   36.0      0   0   2  
2             2   2  8229   77.0      0   0   3  
3             3   3  8230  131.0      0   0   4  
4             4   2  8229  150.0      0   0   5
```

```
src_l = g_df.u.values # src  src_l: [ 1  2  2  3  2  3  4  
dst_l = g_df.i.values # dst  dst_l: [8228 8229 8229 8230 8  
e_idx_l = g_df.idx.values e_idx_l: [ 1  2  3  4  5  
label_l = g_df.label.values label_l: [0 0 0 0 0 0 0 0 0  
ts_l = g_df.ts.values  ts_l: [ 0.  36.  77. 131. 150.
```

0.7      0.85



训练集

时间范围

```
valid_train_flag = (ts_l <= val_time) * (none_node_flag > 0) # 训练集时间 & 不在测试集中的节点
```

验证集

```
valid_val_flag = (ts_l <= test_time) * (ts_l > val_time) # 验证集时间
```

测试集

```
valid_test_flag = ts_l > test_time # 测试集时间
```

至少有一个节点不在训练集中

new\_node\_set : 不在训练集中节点

```
is_new_node_edge = np.array([(a in new_node_set or b in new_node_set) for a, b in zip(src_l, dst_l)])
nn_val_flag = valid_val_flag * is_new_node_edge # val涉及到的边
nn_test_flag = valid_test_flag * is_new_node_edge # test涉及到的边
```

## 训练集数据

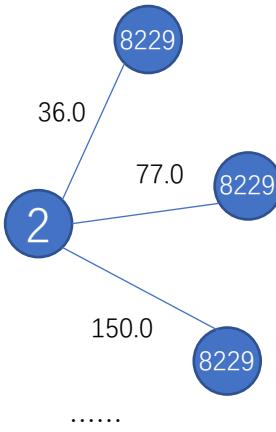
## 无向边

In[46]: adj\_list[2]

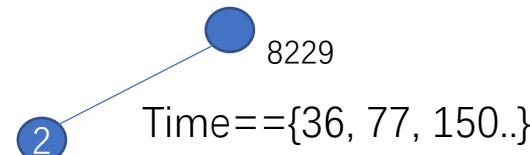
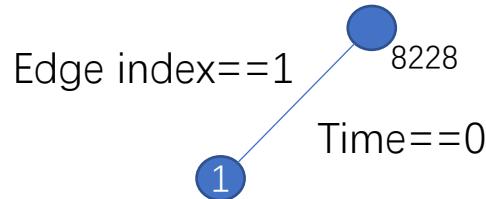
Out[46]:

[(8229, 2, 36.0),  
(8229, 3, 77.0),  
(8229, 5, 150.0),  
(8229, 8, 217.0),  
(8229, 12, 300.0),  
(8229, 13, 376.0),  
(8229, 19, 402.0),  
(8229, 23, 465.0),  
(8229, 26, 623.0),  
(8229, 29, 692.0),  
(8229, 36, 854.0),  
(8229, 41, 989.0),  
(8229, 98, 2313.0)]

## src节点所连接的节点信息



```
train_ngh_finder = NeighborFinder(adj_list, uniform=UNIFORM) # 训练集的数据统计
```



点边连接关系

```
def init_off_set(self, adj_list)
```

```
for i in range(len(adj_list)): # 遍历邻接矩阵 [dst, id, time] i: 1
    curr = adj_list[i] # 当前节点连接的邻居节点 curr: <class 'list'>: [(8228, 1, 0.0)]
    curr = sorted(curr, key=lambda x: x[1]) # 按idx排序, 其实也就是time排序
    n_idx_l.extend([x[0] for x in curr]) # dst
    e_idx_l.extend([x[1] for x in curr]) # id
    n_ts_l.extend([x[2] for x in curr]) # time
    off_set_l.append(len(n_idx_l)) # 连接节点情况, 递增
```

邻居节点      边索引      时间      初始0 第0个节点

第1个节点邻接情况 : [8228]      [1]      [0.0]      [0, 0, 1] 第1个节点

第2个节点邻接情况 : [8228, 8229, 8229, 8229, 8229, 8229, 8229, 8229, 8229, 8229..]

[1, 2, 3, 5, 8, 12, 13, 19, 23, 26 ..] Length = 285

[0.0, 36.0, 77.0, 150.0, 217.0, 300.0, 376.0, 402.0, 465.0, 623.0..]

1节点

off\_set\_l [0, 0, 1, 286]  
初始 0节点 2节点

```
full_ngh_finder = NeighborFinder(full_adj_list, uniform=UNIFORM) # 全图的数据统计
```

In[6]: adj\_list[:2]

Out[6]: [[], [(8228, 1, 0.0)]]

In[11]: adj\_list[2]

Out[11]: 285

(8229, 2, 36.0),

(8229, 3, 77.0),

(8229, 5, 150.0),

(8229, 8, 217.0),

(8229, 12, 300.0),

(8229, 13, 376.0),

(8229, 19, 402.0),

(8229, 23, 465.0),

(8229, 26, 623.0),

(8229, 29, 692.0),

train\_ngh\_finder

full\_ngh\_finder

# Model

```
tgan = TGAN(train_ngh_finder, n_feat, e_feat, # 训练集统计数据, 点特征, 边特征
             num_layers=NUM_LAYER, use_time=USE_TIME, agg_method=AGG_METHOD, attn_mode=ATTN_MODE,
             seq_len=SEQ_LEN, n_head=NUM_HEADS, drop_out=DROP_OUT, node_dim=NODE_DIM, time_dim=TIME_DIM)

[172] [172] [172] 344 -> 172

self.merge_layer = MergeLayer(self.feat_dim, self.feat_dim, self.feat_dim, self.feat_dim)
[172]

[dim1 + dim2] => dim4

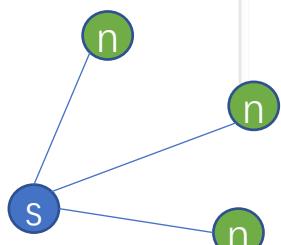
def __init__(self, dim1, dim2, dim3, dim4): self: MergeLayer
super().__init__()
#self.layer_norm = torch.nn.LayerNorm(dim1 + dim2)
self.fc1 = torch.nn.Linear(dim1 + dim2, dim3)
self.fc2 = torch.nn.Linear(dim3, dim4)

def forward(self, x1, x2):
    x = torch.cat([x1, x2], dim=1)
    #x = self.layer_norm(x)
    h = self.act(self.fc1(x))
    return self.fc2(h)
```

```
class AttnModel(torch.nn.Module)
```

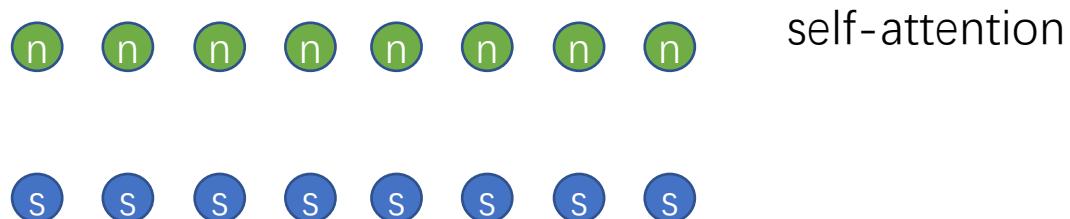
```
    self.attn_model_list = torch.nn.ModuleList([AttnModel(self.feat_dim,
```

AttentionModel



```
self.feat_dim,
self.feat_dim,
attn_mode=attn_mode,
n_head=n_head,
drop_out=drop_out) for _ in range(num_layers)])
```

```
src_e_ph = torch.zeros_like(src_ext)
q = torch.cat([src_ext, src_e_ph, src_t], dim=2)
k = torch.cat([seq, seq_e, seq_t], dim=2) # [B,
v = k
```



```
if attn_mode == 'prod':
```

```
    self.multi_head_target = MultiHeadAttention(n_head,
```

多头attention

```
        d_model=self.model_dim, # feat_dim + edge_dim + time_dim
        d_k=self.model_dim // n_head,
        d_v=self.model_dim // n_head,
        dropout=drop_out)
```

## TimeEncode

```
self.time_encoder = TimeEncode(expand_dim=self.n_feat_th.shape[1]) # 172
```

$$\cos(basis_{freq} * time + phase)$$

$$\Phi_d(t) = [\cos(\omega_1 t + \theta_1), \cos(\omega_2 t + \theta_2), \dots, \cos(\omega_d t + \theta_d)]$$

---

用时间维度去代替位置维度

$$\mathbf{Z}_e = [\mathbf{z}_{e_1} + \mathbf{p}_1, \dots, \mathbf{z}_{e_1} + \mathbf{p}_l]^\top \in \mathbb{R}^{l \times d}, \text{ or } \mathbf{Z}_e = [\mathbf{z}_{e_1} \parallel \mathbf{p}_1, \dots, \mathbf{z}_{e_1} \parallel \mathbf{p}_l]^\top \in \mathbb{R}^{l \times (d+d_{pos})}. \quad (1)$$

假设时域可以用从原点开始的间隔来表示

考虑两个时间点t1和t2以及它们函数编码之间的内积  $\langle \Phi(t_1), \Phi(t_2) \rangle$

## 训练集数据

```
src_l_cut, dst_l_cut = train_src_l[s_idx:e_idx], train_dst_l[s_idx:e_idx]
ts_l_cut = train_ts_l[s_idx:e_idx] # time
label_l_cut = train_label_l[s_idx:e_idx] # label
```

```
src_embed = self.tem_conv(src_idx_l, cut_time_l, self.num_layers, num_neighbors)
target_embed = self.tem_conv(target_idx_l, cut_time_l, self.num_layers, num_neighbors)
background_embed = self.tem_conv(background_idx_l, cut_time_l, self.num_layers, num_neighbors)
```

```
In[22]: src_idx_l[:10]
Out[22]: array([1, 2, 2, 3, 2, 3, 2, 5, 5, 6])
In[23]: target_idx_l[:10]
Out[23]: array([8228, 8229, 8229, 8230, 8229, 8230, 8229, 8232, 8232, 8233])
In[24]: background_idx_l[:10]
Out[24]: array([8650, 8810, 8878, 9082, 8327, 9140, 9006, 9068, 8801, 8265])
In[25]: cut_time_l[:10]
Out[25]: array([ 0., 36., 77., 131., 150., 153., 217., 218., 242., 295.])
```

tem\_conv  
Layer = 2

A

time encode  
node encoder

B

tem\_conv  
Layer = 1  
Layer=2

C

A

D

B

获取邻居节点  
4000

self-attention

节点聚合一阶邻居后结果

[200, 172]

Layer=1

[200, 172]

[4000, 172]

tem\_conv  
Layer = 0

A

B

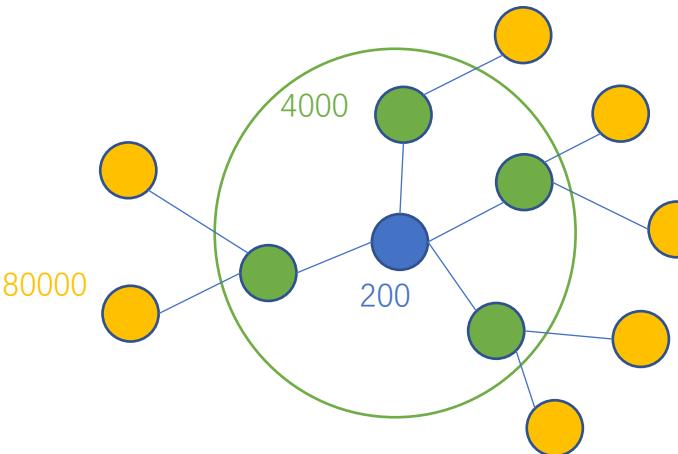
Layer=0

return src\_node\_feat

src节点embedding

tem\_conv  
Layer = 0

src的邻居节点embedding



第一层TGAT  
目标节点

4000个邻居节点

time encode

全0时间

```
src_node_t_embed = self.time_encoder(torch.zeros_like(cut_time_l_th))
```

$$\Phi_d(t) = [\cos(\omega_1 t + \theta_1), \cos(\omega_2 t + \theta_2), \dots, \cos(\omega_d t + \theta_d)]$$

```
def forward(self, ts):    self: TimeEncode()  ts: 1
    # ts: [N, L]
    batch_size = ts.size(0)  batch_size: 200
    seq_len = ts.size(1)  seq_len: 1

    ts = ts.view(batch_size, seq_len, 1) # [N, L,
    map_ts = ts * self.basis_freq.view(1, 1, -1)
    map_ts += self.phase.view(1, 1, -1)

    harmonic = torch.cos(map_ts)  # cos函数;
```

初始节点，所以时间都为0

$$\mathbf{Z}(t) = \left[ \tilde{\mathbf{h}}_0^{(l-1)}(t) | \boxed{\Phi_{d_T}(0)}, \tilde{\mathbf{h}}_1^{(l-1)}(t_1) | \Phi_{d_T}(t - t_1), \dots, \tilde{\mathbf{h}}_N^{(l-1)}(t_N) | \Phi_{d_T}(t - t_N) \right]^\top \text{(or use sum)} \quad (6)$$

node encoder

```
src_node_feat = self.node_raw_embed(src_node_batch_th)
```

节点对应到的embedding

In[40]: self.node\_raw\_embed  
Out[40]: Embedding(9228, 172, padding\_idx=0)

```
self.node_raw_embed = torch.nn.Embedding.from_pretrained(self.n_feat_th, padding_idx=0, freeze=True)
```

## 邻居采样

```
self.ngh_finder = ngh_finder # 训练集数据
```

```
src_ngh_node_batch, src_ngh_eidx_batch, src_ngh_t_batch = self.ngh_finder.get_temporal_neighbor(  
    src_idx_l, # 节点id  
    cut_time_l, # 时间  
    num_neighbors=num_neighbors) # neighbors
```

节点id                   时间

```
for i, (src_idx, cut_time) in enumerate(zip(src_idx_l, cut_time_l)): # src, time
```

```
ngh_idx, ngh_eidx, ngh_ts = self.find_before(src_idx, cut_time)
```

[1]               0.0

找节点在该时间前的邻居节点

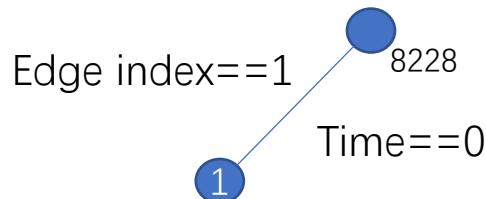
```
In[34]: src_idx_l[:10]
```

```
Out[34]: array([1, 2, 2, 3, 2, 3, 2, 5, 5, 6])
```

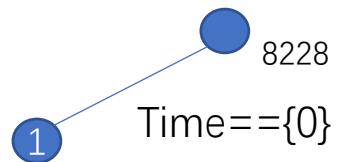
```
In[35]: cut_time_l[:10]
```

```
Out[35]: array([ 0., 36., 77., 131., 150., 153., 217., 218., 242., 295.])
```

节点1在0时刻之前的邻居节点

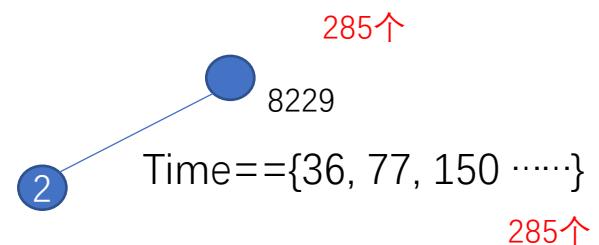


src\_node = 1  
cut\_time = 0.0



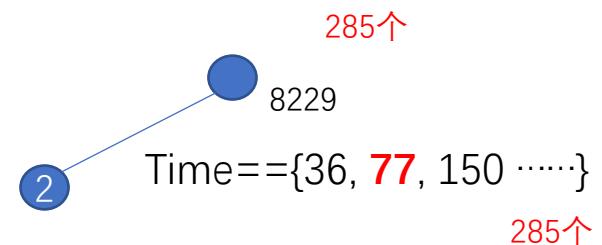
下一个时间

src\_node = 2  
cut\_time = 36.0



下一个时间

src\_node = 2  
cut\_time = 77.0

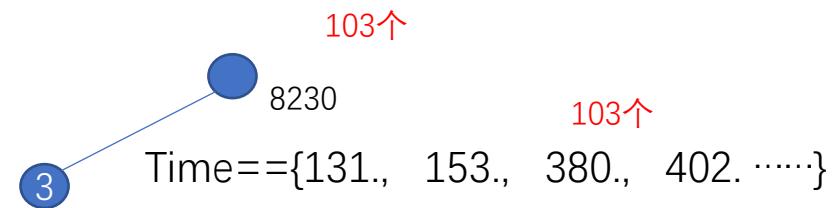


```
ngh_idx, ngh_eidx, ngh_ts = self.find_before(src_idx, cut_time) # 找到小于时间的邻居节点
```

8229      2      36

下一个时间

src\_node = 2  
cut\_time = 131.0



src\_ngh\_node\_batch

节点的邻居节点连接情况

[200, 20]

In [13]: out\_ngh\_node\_batch

Out[13]:

```
array([[ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0, 8229],
       ...,
       [ 0,  0,  0, ...,  0, 8297, 8297],
       [8229, 8229, 8229, ..., 8229, 8229, 8229],
       [ 0,  0,  0, ..., 8262, 8262, 8262]], dtype=int32)
```

节点和邻居连接的edge索引

In [14]: out\_ngh\_eidx\_batch

Out[14]:

```
array([[ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  0],
       [ 0,  0,  0, ...,  0,  0,  2],
       ...,
       [ 0,  0,  0, ...,  0, 307, 310],
       [ 2,  3,  5, ..., 160, 184, 188],
       [ 0,  0,  0, ..., 119, 167, 313]], dtype=int32)
```

节点连接到邻居的time

In [15]: out\_ngh\_t\_batch

Out[15]:

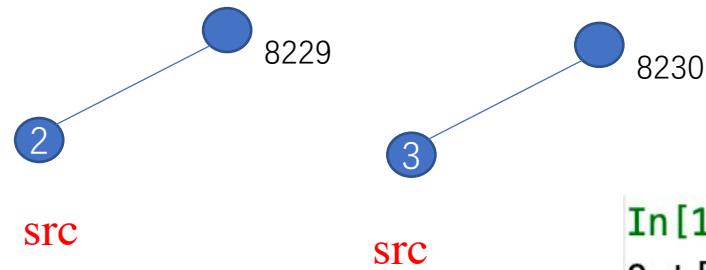
```
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  0., ...,  0.,  0., 36.],
       ...,
       [ 0.,  0.,  0., ...,  0., 7101., 7198.],
       [ 36., 77., 150., ..., 3331., 4188., 4243.],
       [ 0.,  0.,  0., ..., 2595., 3591., 7234.]], dtype=float32)
```

In [96]: src\_idx\_l

Out[96]:

```
array([ 1,  2,  2,  3,  2,  3,  2,  5,  5,  6,  2,  2,  3,  7,  8,  2,  3,
       3,  2,  7,  5,  2,  3,  2,  3, 11, 12, 13,  2,  5, 14, 15, 13,  2,
       15, 15, 16, 13, 18,  5,  5, 19, 12, 21, 22, 23, 18, 12,  5, 24,  5,
       25, 13, 26,  5,  3,  5,  5, 12, 13, 29, 19, 30, 12,  3,  5,  5, 31,
```

原始数据：200个按时间排序的src节点



200个src节点的邻居节点

In[111]: src\_ngh\_t\_batch\_flat

Out[111]: array([ 0., 0., 0., ..., 2595., 3591., 7234.], dtype=float32)

4000

tem\_conv  
Layer = 2



time encode  
node encoder



tem\_conv  
Layer = 2



[200, 172]

获取邻居节点  
4000

self-attention

节点聚合一阶邻居后结果

Layer=1

[200, 172]

[4000, 172]

tem\_conv  
Layer = 0



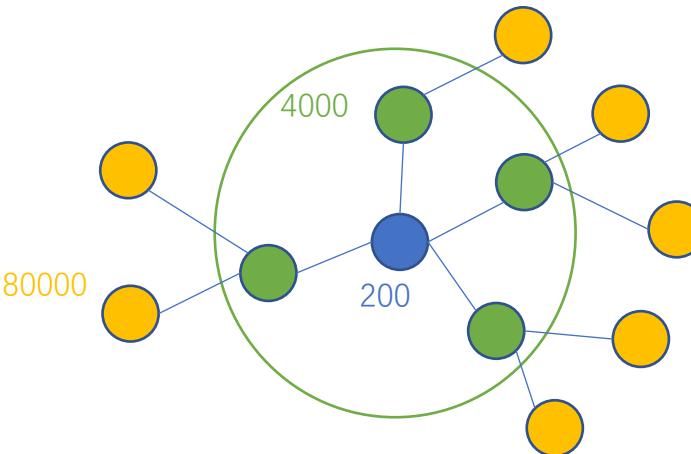
Layer=0

**return src\_node\_feat**

src节点embedding

tem\_conv  
Layer = 0

src的邻居节点embedding



4000个邻居节点

```

# get edge time features and node features
src_ngh_t_embed = self.time_encoder(src_ngh_t_batch_th) # 邻居时间差 src_ngh_
src_ngn_edge_feat = self.edge_raw_embed(src_ngh_eidx_batch) # 边的embedding;
mask = src_ngh_node_batch_th == 0 # 时间差mask
attn_m = self.attn_model_list[curr_layers - 1]

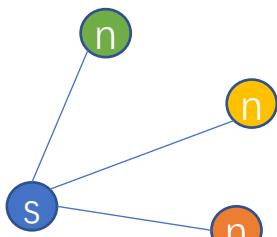
```

```

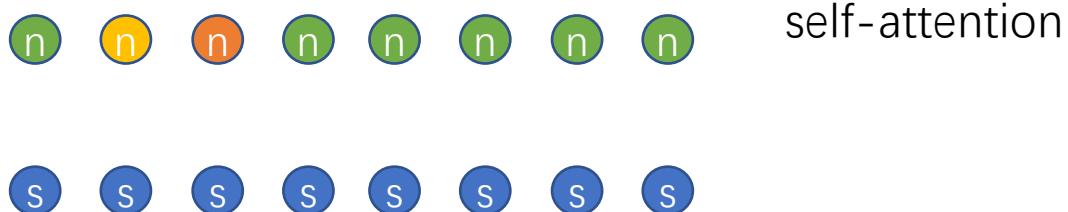
self.attn_model_list = torch.nn.ModuleList([AttnModel(self.feat_dim,
                                                    self.feat_dim,
                                                    self.feat_dim,
                                                    attn_mode=attn_mode,
                                                    n_head=n_head,
                                                    drop_out=drop_out) for _ in range(num_layers)])

```

AttentionModel



$\text{src\_e\_ph} = \text{torch.zeros\_like}(\text{src\_ext})$   
 $\mathbf{q} = \text{torch.cat}([\text{src\_ext}, \text{src\_e\_ph}, \text{src\_t}], \text{dim}=2)$        $\mathbf{Z}(t) = [\dots, \tilde{\mathbf{h}}_i^{(l-1)}(t_i) || \mathbf{x}_{0,i}(t_i) || \Phi_{d_T}(t - t_i), \dots]$   
 $\mathbf{k} = \text{torch.cat}([\text{seq}, \text{seq\_e}, \text{seq\_t}], \text{dim}=2)$  #     $\mathbf{q}(t) = [\mathbf{Z}(t)]_0 \mathbf{W}_Q, \mathbf{K}(t) = [\mathbf{Z}(t)]_{1:N} \mathbf{W}_K, \mathbf{V}(t) = [\mathbf{Z}(t)]_{1:N} \mathbf{W}_V,$   
 $\mathbf{v} = \mathbf{k}$



```
# # target-attention
```

```
output, attn = self.multi_head_target(q=q, k=k, v=v, mask=mask),
```

Multi-head

```
q = self.w_qs(q).view(sz_b, len_q, n_head, d_k) # Q*W; [200, 1, 2, 258]  
k = self.w_ks(k).view(sz_b, len_k, n_head, d_k) # K*W; [200, 20, 2, 258]  
v = self.w_vs(v).view(sz_b, len_v, n_head, d_v) # V*W; [200, 20, 2, 258]  
[200, 20, 2, 258]
```

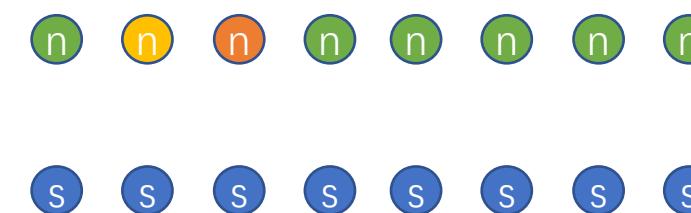
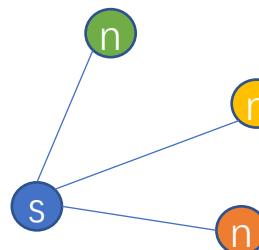
将多头attention放到第一维

```
q = q.permute(2, 0, 1, 3).contiguous().view(-1, len_q, d_k)  
k = k.permute(2, 0, 1, 3).contiguous().view(-1, len_k, d_k)  
v = v.permute(2, 0, 1, 3).contiguous().view(-1, len_v, d_v)
```

[400, 20, 258]

```
mask = mask.repeat(n_head, 1, 1) # (n*b) x .. x ..; [400, 1, 20]
```

```
output, attn = self.attention(q, k, v, mask=mask) # 输出self-attention聚合后的结果;
```



self-attention

```
output = self.merger(output, src) # output, 节点特征; FFN
```

$$\tilde{\mathbf{h}}_0^{(l)}(t) = \text{FFN}\left(\mathbf{h}(t)||\mathbf{x}_0\right) \equiv \text{ReLU}\left([\mathbf{h}(t)||\mathbf{x}_0]\mathbf{W}_0^{(l)} + \mathbf{b}_0^{(l)}\right)\mathbf{W}_1^{(l)} + \mathbf{b}_1^{(l)},$$

tem\_conv  
Layer = 2



time encode  
node encoder



tem\_conv  
Layer = 1  
Layer=2



[200, 172]

获取邻居节点  
4000

self-attention

节点聚合一阶邻居后结果

Layer=1

[200, 172]

[4000, 172]

tem\_conv  
Layer = 0



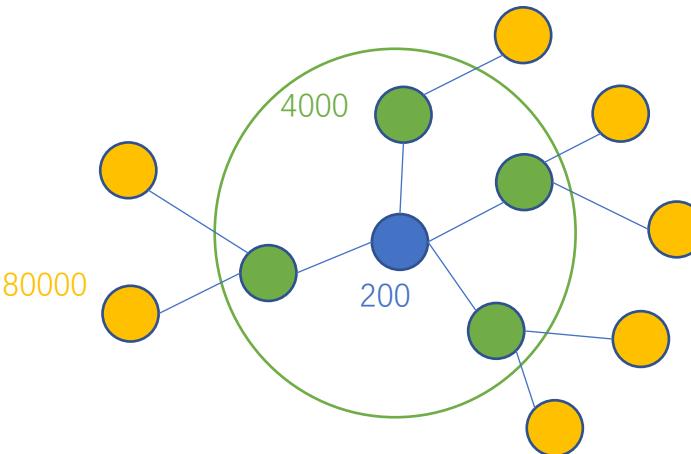
Layer=0

**return src\_node\_feat**

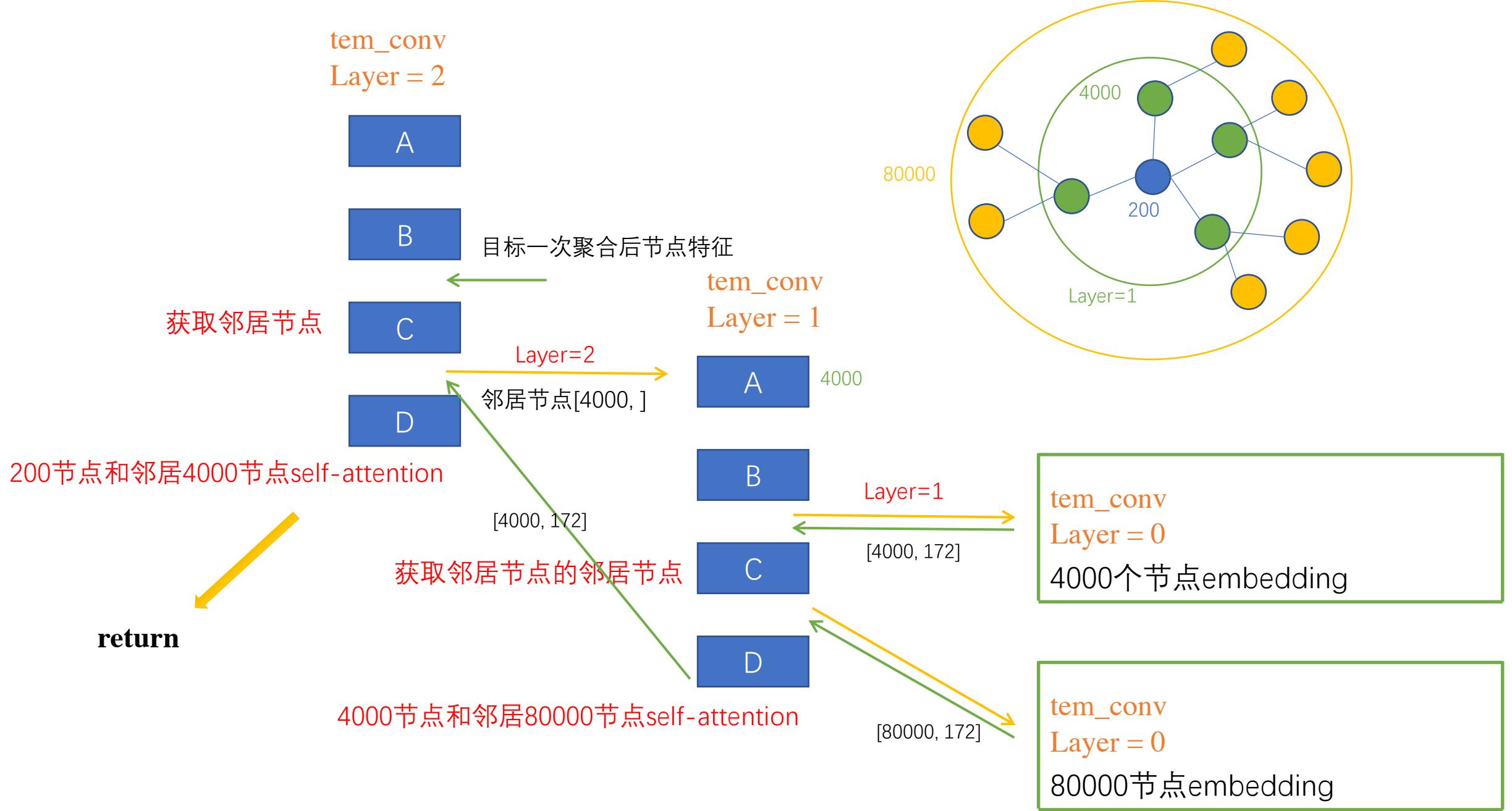
src节点embedding

tem\_conv  
Layer = 0

src的邻居节点embedding



4000个邻居节点



超图

# Hypergraph Neural Networks

HGNN

传统的图网络数据，两个节点之间有一条边连接，但是在现实世界中，在某些领域，一条边可能连接的不止两个节点，那么超图的引入就十分必要了。

比如：在同一个班级中，一个班级可以连接很多个学生。一个ip下面，有多个用户。

超图其实是一种广义的关系结构，一个超边可以拥有任意数量的节点 ( $>1$ )，如果所有超边都连接两个节点，那么超图就退化为简单图，所以说简单图是超图的一个子集。

### 面对多模态数据的场景

多模态场景中，例如论文中的博客例子，其中包括视觉连接、文本连接和社交连接。所以将这些很容易放在超图的构图中去解决，而在简单图中，处理则变得困难的多。

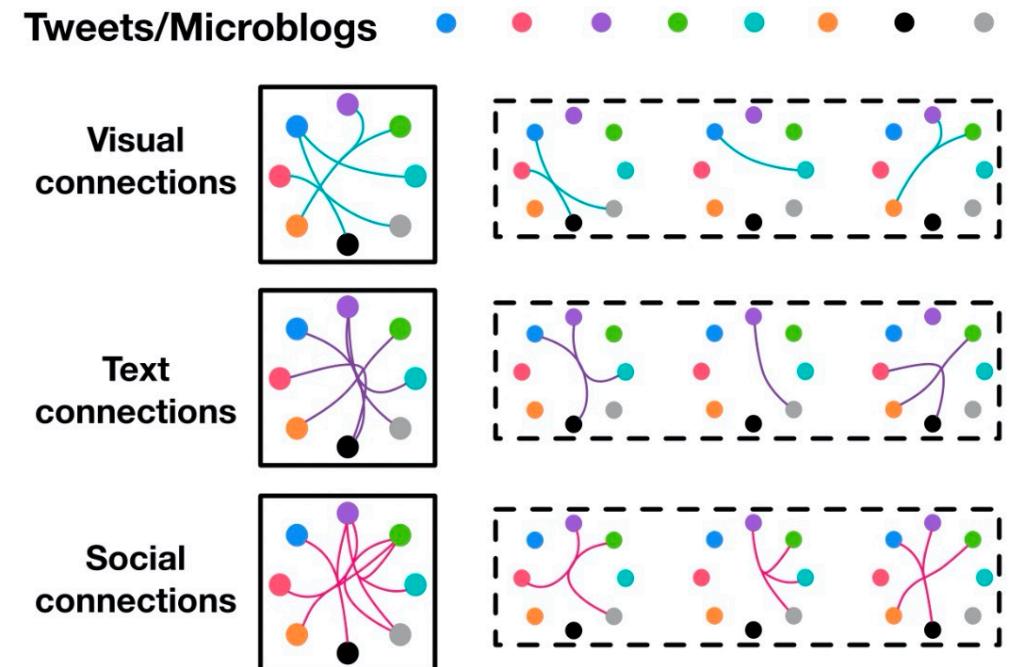


Figure 1: Examples of complex connections on social media data. Each color point represents a tweet or microblog, and there could be visual connections, text connections and social connections among them.

# Hypergraph Neural Networks

超图定义

超边的集合

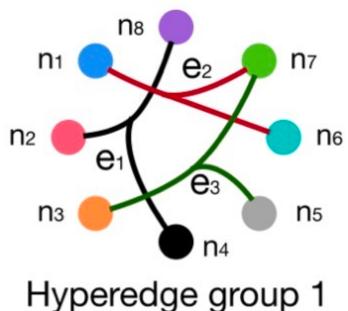
$$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{W})$$

节点

每个超边被赋予的权值

初始化W为一个单位阵，意味着所有超边的权重都是相同的  
W可以根据模型一起去训练，也可以固定为单位阵

如何构建超边



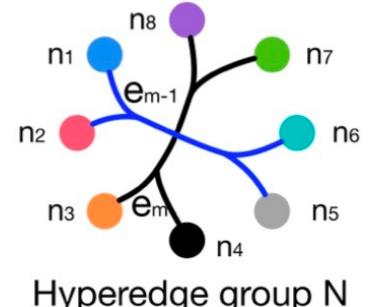
$$e^1 = \{n_2, n_4, n_8\}$$

$$e^2 = \{n_1, n_7, n_6\}$$

$$e^3 = \{n_3, n_5, n_7\}$$

$$e^{m-1} = \{n_1, n_2, n_5, n_6\}$$

$$e^m = \{n_3, n_4, n_7, n_8\}$$



	e <sub>1</sub>	e <sub>2</sub>	e <sub>3</sub>	e <sub>m-1</sub>	e <sub>m</sub>
n <sub>1</sub>	0	1	0	1	0
n <sub>2</sub>	1	0	0	1	0
n <sub>3</sub>	0	0	1	0	1
n <sub>4</sub>	1	0	0	0	1
n <sub>5</sub>	0	0	1	1	0
n <sub>6</sub>	0	1	0	1	0
n <sub>7</sub>	0	1	1	0	1
n <sub>8</sub>	1	0	0	0	1

$$W = \begin{matrix} & e^1 & e^2 & e^3 & e^{m-1} & e^m \\ 1 & & & & & \\ & 1 & & & & \\ & & 1 & & & \\ & & & 0.9 & & \\ & & & & 0.9 & \end{matrix}$$

$$h(v, e) = \begin{cases} 1, & \text{if } v \in e \\ 0, & \text{if } v \notin e, \end{cases} \quad (1)$$



$$H = \begin{matrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{matrix} \quad X = \begin{matrix} x_1 \\ x_2 \\ x_3 \end{matrix} \quad W = \begin{matrix} 0.9 & 0 \\ 0 & 1 \end{matrix}$$

节点Degree定义为：

$$d(v) = \sum_{e \in \mathcal{E}} \omega(e) h(v, e) \quad \text{对每一个边度进行加权求和 } d(v) = 0.9, 1.9, 1.9$$

	$e_1$	$e_2$		
$v_1$	1	0	$\longrightarrow$	$0.9 * 1 + 1 * 0$
$v_2$	1	1	$\longrightarrow$	$0.9 * 1 + 1 * 1$
$v_3$	1	1	$\longrightarrow$	$0.9 * 1 + 1 * 1$

$$DV = \begin{matrix} 0.9 & 0 & 0 \\ 0 & 1.9 & 0 \\ 0 & 0 & 1.9 \end{matrix}$$

边Degree定义为：

$$\delta(e) = \sum_{v \in \mathcal{V}} h(v, e) \quad \text{对每一个节点度进行加权求和 } d(e) = 3, 2$$

	$e_1$	$e_2$		
$v_1$	1	0		
$v_2$	1	1		
$v_3$	1	1		

$$DE = \begin{matrix} 3 & 0 \\ 0 & 2 \end{matrix}$$

↓      ↓

节点Degree定义为：

$$d(v) = \sum_{e \in \mathcal{E}} \omega(e) h(v, e)$$

$\mathbf{D}_e$  and  $\mathbf{D}_v$

边缘度和顶点度的对角矩阵

边Degree定义为：

$$\delta(e) = \sum_{v \in \mathcal{V}} h(v, e)$$

超图更新公式：

$$\mathbf{X}^{(l+1)} = \sigma(\mathbf{D}_v^{-1/2} \mathbf{H} \mathbf{W} \mathbf{D}_e^{-1} \mathbf{H}^\top \mathbf{D}_v^{-1/2} \mathbf{X}^{(l)} \Theta^{(l)})$$

超图的邻域矩阵  
L层的特征  
归一化作用

GCN更新公式：

$$Z = \tilde{\mathbf{D}}^{-\frac{1}{2}} \tilde{\mathbf{A}} \tilde{\mathbf{D}}^{-\frac{1}{2}} X \Theta$$



$$H = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix}$$

$$X = \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix}$$

$$DV = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

$$DE = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$$

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\mathbf{Y} = \mathbf{D}_v^{-1/2} \mathbf{H} \mathbf{W} \mathbf{D}_e^{-1} \mathbf{H}^\top \mathbf{D}_v^{-1/2} \mathbf{X} \Theta, \quad (10) \quad X \Theta : \text{全连接操作, } X=(140,256) \xrightarrow{[256,128]} (140,128)$$

根据点的特征，构建超边特征（将边连接到的节点求和）

$$H^T \cdot X = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x1 \\ x2 \\ x3 \end{pmatrix} = \begin{pmatrix} x1 + x2 + x3 \\ x2 + x3 \end{pmatrix}$$

点1 点2 点3

边1	1	1	1	边1连接着{1,2,3}三个节点
边2	0	1	1	边2连接着{2,3}两个节点

汇聚超边的特征后，完成点特征更新

$$H \cdot H^T \cdot X = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x1 + x2 + x3 \\ x2 + x3 \end{pmatrix} = \begin{pmatrix} x1 + x2 + x3 \\ x1 + 2x2 + 2x3 \\ x1 + 2x2 + 2x3 \end{pmatrix}$$

边1 边2

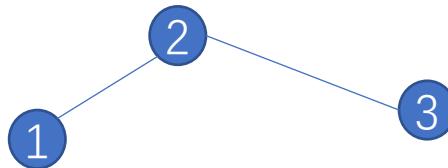
节点1	1	0	节点1连接的一条边
节点2	1	1	节点2连接的两条条边
节点3	1	1	节点2连接的两条条边

归一化后结果

$$0.33 * x1 + 0.23 * x2 + 0.23 * x3$$

$$0.23 * x1 + 0.41 * x2 + 0.41 * x3$$

$$0.23 * x1 + 0.41 * x2 + 0.41 * x3$$



HGNN

$$\mathbf{Y} = \mathbf{D}_v^{-1/2} \mathbf{H} \mathbf{W} \mathbf{D}_e^{-1} \mathbf{H}^\top \mathbf{D}_v^{-1/2} \mathbf{X} \Theta, \quad (10)$$

根据点的特征，构建超边特征

$$H^T \cdot X = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ x_2 + x_3 \end{pmatrix}$$

汇聚超边的特征完成点特征更新

$$H \cdot H^T \cdot X = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x_1 + x_2 \\ x_2 + x_3 \end{pmatrix} = \begin{pmatrix} x_1 + x_2 \\ x_1 + 2x_2 + x_3 \\ x_2 + x_3 \end{pmatrix}$$

归一化后结果

$$\begin{aligned} & 0.5 * x_1 + 0.35 * x_2 \\ & 0.35 * x_1 + 0.5 * x_2 + 0.35 * x_3 \\ & 0.35 * x_2 + 0.5 * x_3 \end{aligned}$$

$$H = \begin{pmatrix} 1 & 0 \\ 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$X = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}$$

$$DV = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

$$DE = \begin{pmatrix} 3 & 0 \\ 0 & 2 \end{pmatrix}$$

$$W = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

GCN :

$$Z = \tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} X \Theta$$

$$\tilde{A} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{pmatrix}$$

$$\tilde{A} \cdot X = \begin{pmatrix} x_1 + x_2 \\ x_1 + x_2 + x_3 \\ x_2 + x_3 \end{pmatrix}$$

归一化后结果

$$\begin{aligned} & 0.5 * x_1 + 0.4 * x_2 \\ & 0.4 * x_1 + 0.3 * x_2 + 0.4 * x_3 \\ & 0.4 * x_2 + 0.5 * x_3 \end{aligned}$$

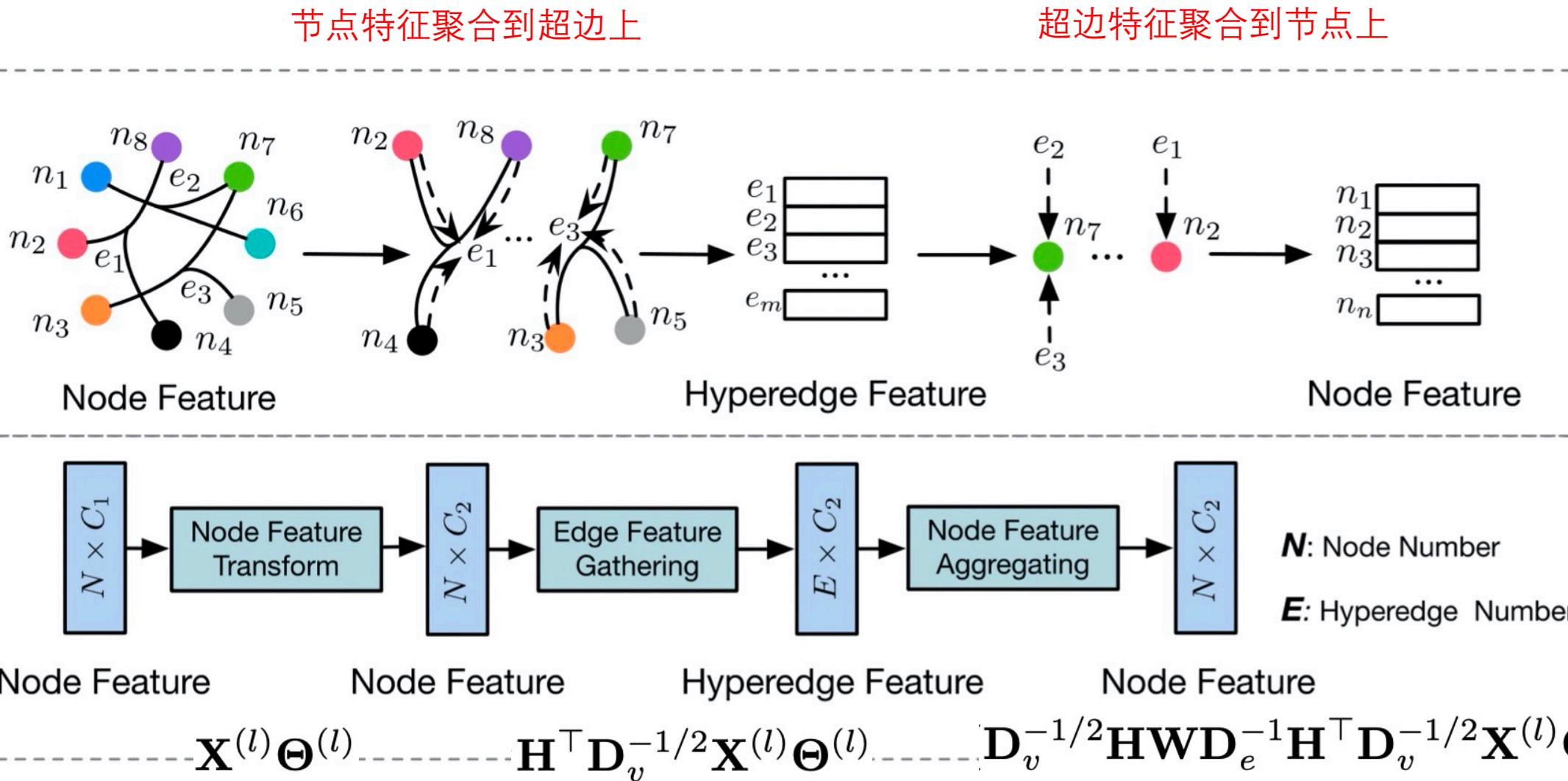


Figure 4: The illustration of the hyperedge convolution layer.

Cora数据集构建超图数据：

每次选择一个节点，超边就是和这个节点相连接的所有节点

因为Cora这种引文数据集中，  
超图结构和原始图结构很类似，  
并且没有加入更多的信息，  
所以效果提升比较少

Method	Cora	Pubmed
DeepWalk (Perozzi, Al-Rfou, and Skiena 2014)	67.2%	65.3%
ICA (Lu and Getoor 2003)	75.1%	73.9%
Planetoid (Yang, Cohen, and Salakhutdinov 2016)	75.7%	77.2%
Chebyshev (Defferrard, Bresson, and Vandergheynst 2016)	81.2%	74.4%
GCN (Kipf and Welling 2017)	81.5%	79.0%
HGNN	<b>81.6%</b>	<b>80.1%</b>

Table 2: Classification results on the Cora and Pubmed datasets.

## 视觉分类任务

每次选择数据集中的一个节点，利用其在所选特征空间中的 10 个最近邻生成一个包括改节点本身的超边

### NTU dataset

Feature	Features for Structure					
	GVCNN		MVCNN		GVCNN+MVCNN	
	GCN	HGNN	GCN	HGNN	GCN	HGNN
GVCNN (Feng et al. 2018)	91.8%	<b>92.6%</b>	91.5%	<b>91.8%</b>	92.8%	<b>96.6%</b>
MVCNN (Su et al. 2015)	92.5%	<b>92.9%</b>	86.7%	<b>91.0%</b>	92.3%	<b>96.6%</b>
GVCNN+MVCNN	-	-	-	-	94.4%	<b>96.7%</b>

Table 4: Comparison between GCN and HGNN on the ModelNet40 dataset.

Feature	Features for Structure					
	GVCNN		MVCNN		GVCNN+MVCNN	
	GCN	HGNN	GCN	HGNN	GCN	HGNN
GVCNN ((Feng et al. 2018))	78.8%	<b>82.5%</b>	78.8%	<b>79.1%</b>	75.9%	<b>84.2%</b>
MVCNN ((Su et al. 2015))	74.0%	<b>77.2%</b>	71.3%	<b>75.6%</b>	73.2%	<b>83.6%</b>
GVCNN+MVCNN	-	-	-	-	76.1%	<b>84.2%</b>

Table 5: Comparison between GCN and HGNN on the NTU dataset.

## ModelNet40

Method	Classification Accuracy
PointNet (Qi et al. 2017a)	89.2%
PointNet++ (Qi et al. 2017b)	90.7%
PointCNN (Li et al. 2018)	91.8%
SO-Net (Li, Chen, and Lee 2018)	93.4%
HGNN	<b>96.7%</b>

Table 6: Experimental comparison among recent classification methods on ModelNet40 dataset.

超图结构能够表达复杂的、高阶的数据之间的相关性，与图结构或无图结构的方法相比，能更好地表示底层数据之间的关系。此外，当多模态数据/特征可用时，HGNN具有通过其灵活的超边将这些多模态信息结合在同一结构中的优势。

Code

```

if is_probH:
    H[node_idx, center_idx] = np.exp(-dis_vec[0, node_idx] ** 2 / (m_prob * avg_dis) ** 2) #

```

距离矩阵计算公式

$$A_{ij} = \exp\left(-\frac{2Dij^2}{\Delta}\right) \quad (12)$$

$$\sum_{e \in \mathcal{E}} \omega(e) h(v, e)$$

```
DV = np.sum(H * W, axis=1) # 度矩阵(但值是距离, 不是1);
```

```
DE = np.sum(H, axis=0) # 超边的度, 入度;  $\delta(e) = \sum_{v \in \mathcal{V}} h(v, e)$ 
```

```
invDE = np.mat(np.diag(np.power(DE, -1))) # DE^-1; 建立对角阵
DV2 = np.mat(np.diag(np.power(DV, -0.5))) # DV^-1/2
```

else:

```
G = DV2 * H * W * invDE * HT * DV2
return G
```

式中( $W + I$ )可视为超边的权值。 $W$ 被初始化为一个单位矩阵, 这意味着所有超边的权值相等。

出度矩阵

入度矩阵

(12311, 2048)

$$\mathbf{Y} = \mathbf{D}_v^{-1/2} \mathbf{H} \mathbf{W} \mathbf{D}_e^{-1} \mathbf{H}^\top \mathbf{D}_v^{-1/2} \mathbf{X} \boldsymbol{\Theta}, \quad (10)$$

超边矩阵  
(12311, 24622)

超边的权重矩阵

初始化 $W$ 为单位阵,  $W$ 中的参数为需要学习的参数

```
def forward(self, x: torch.Tensor, G: torch.Tensor):
    x = x.matmul(self.weight)                                X*Weight
    if self.bias is not None:
        x = x + self.bias
    x = G.matmul(x)                                         G*X*Weight
```

$$\mathbf{X}^{(l+1)} = \sigma(\mathbf{D}_v^{-1/2} \mathbf{H} \mathbf{W} \mathbf{D}_e^{-1} \mathbf{H}^\top \mathbf{D}_v^{-1/2} \mathbf{X}^{(l)} \boldsymbol{\Theta}^{(l)}), \quad (11)$$



# Dynamic Hypergraph Neural Networks

dynamic hypergraph neural networks framework (DHGNN)

我们的之前介绍的图算法，都是基于一个固定的图，也就是在隐层中，使用同样的图结构。  
这样方式忽略了隐层特征其实是经过了转换，转换后的特征应该对结构进行动态修改。

如何解决隐层关系对于图结构的影响，作者提出了DHGNN：  
这个算法主要贡献是在动态的调整隐层中的超图结构

所以DHGNN由两个部分组成：

- 1) dynamic hypergraph construction (DHC) 动态构建超图
- 2) hypergrph convolution (HGC) 超图卷积

vertex convolution and hyperedge convolution

动态构建超图：  
基于特征，使用KNN和Kmeans  
构建超边

### **3 Dynamic Hypergraph Neural Networks**

DHGNN layer consists of two major part:

1. dynamic hypergraph construction (DHG)
2. hypergraph convolution (HGC).

#### **3.1 Dynamic Hypergraph Construction**

超图的构建

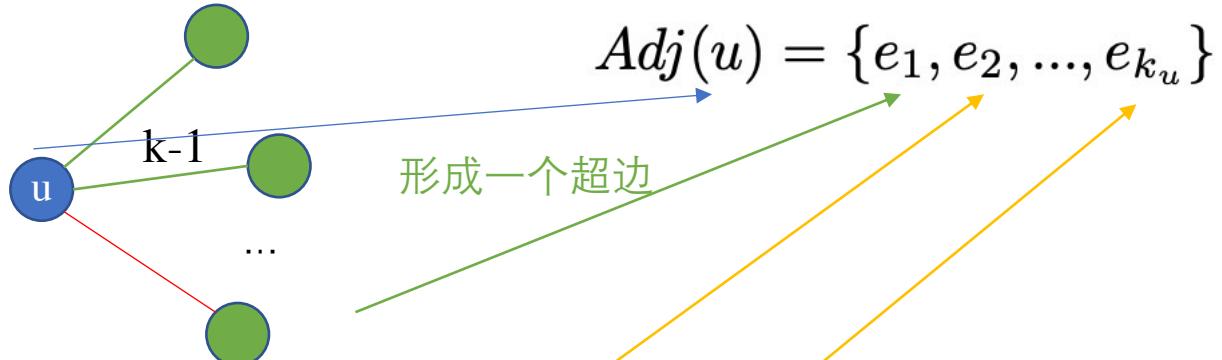
超边的构建

主要包括两种方法

1. KMeans
2. KNN

## KNN

1. 我们计算了每个顶点  $u$  的  $k-1$  个最近邻。这些邻域顶点与顶点  $u$  一起形成  $\text{Adj}(u)$  中的超边



## KMeans

2. 我们根据欧氏距离对每一层的整个特征图进行了k-means算法。对于每个顶点，取最近的  $S-1$  个簇中心，则这些簇节点将被指定为该顶点的超边。



## 超边的构建

### Algorithm 1 Hypergraph Construction

**Input:** Input embedding  $\mathbf{X}$ ; hyperedge size  $k$ ; adjacent hyperedge set size  $S$

**Output:** Hyperedge set  $\mathcal{G}$

**Function:** k-Means clustering  $kMeans$ ; k-nearest neighborhood selection  $knn$ ; distance function  $dis$ ;  $S - 1$  smallest distance index selection  $topK$

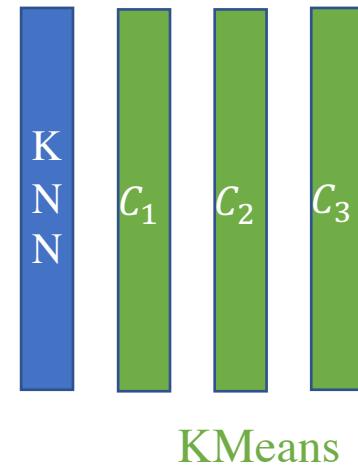
```
1:  $C = kMeans(\mathbf{X})$  聚类C个类别
2: for  $u$  in  $range(len(\mathbf{X}))$  do
3:    $e_b = knn(X[u], X, k)$  K个最新的节点
4:    $\mathcal{G}[u].insert(e_b)$  加入超边
5:    $D = dis(C.center, u)$  节点和C个簇的中心节点距离
6:    $D = sort(D)$ 
7:    $ind = topK(D, S - 1)$  选择S-1个簇
8:   for  $i$  in  $ind$  do
9:      $\mathcal{G}[u].insert(C[i])$  将这S-1个节点加入超边
10:  end for
11: end for
```

主要包括两种方法

1. KMeans
2. KNN

表示包含节点 $u$ 的所有超边集合

$$Adj(u) = \{e_1, e_2, \dots, e_{k_u}\}$$



## 3.2 Hypergraph Convolution

超图卷积

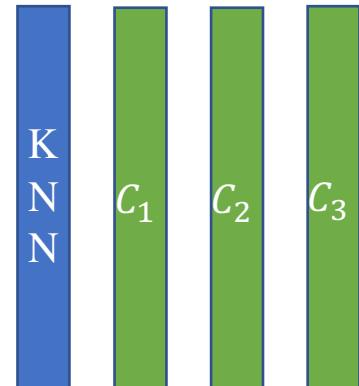
$$Adj(u) = \{e_1, e_2, \dots, e_{k_u}\}$$

vertex convolution

顶点卷积将顶点特征聚集到超边缘

hyperedge convolution

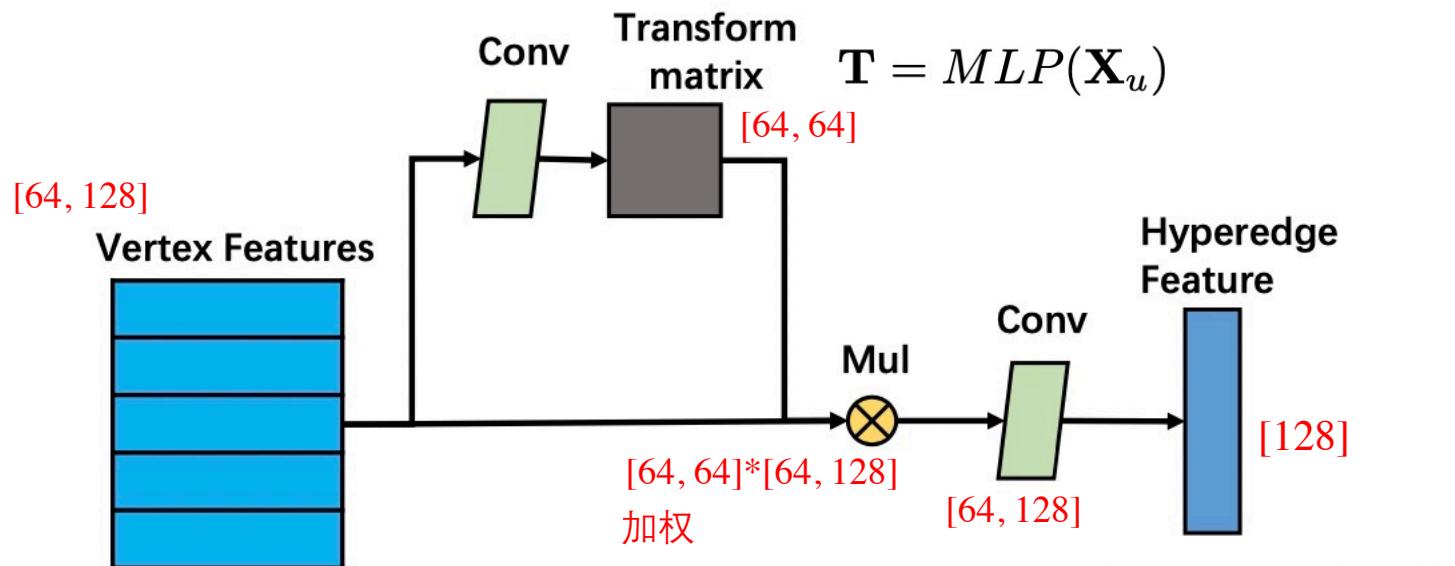
然后超边卷积通过超边卷积将相邻的超边特征聚合到质心顶点。



vertex convolution最简单的方法是pooling方式，之前HGNN其实就是求和方式。

这里作者采用了更好的，一个可学习Transform matrix方法，对超边节点进行加权聚合。

KMeans



一个超边对应到的节点

**Vertex Convolution**

$$\mathbf{T} = MLP(\mathbf{X}_u)$$

$$\mathbf{x}_e = conv(\mathbf{T} \cdot MLP(\mathbf{X}_u))$$

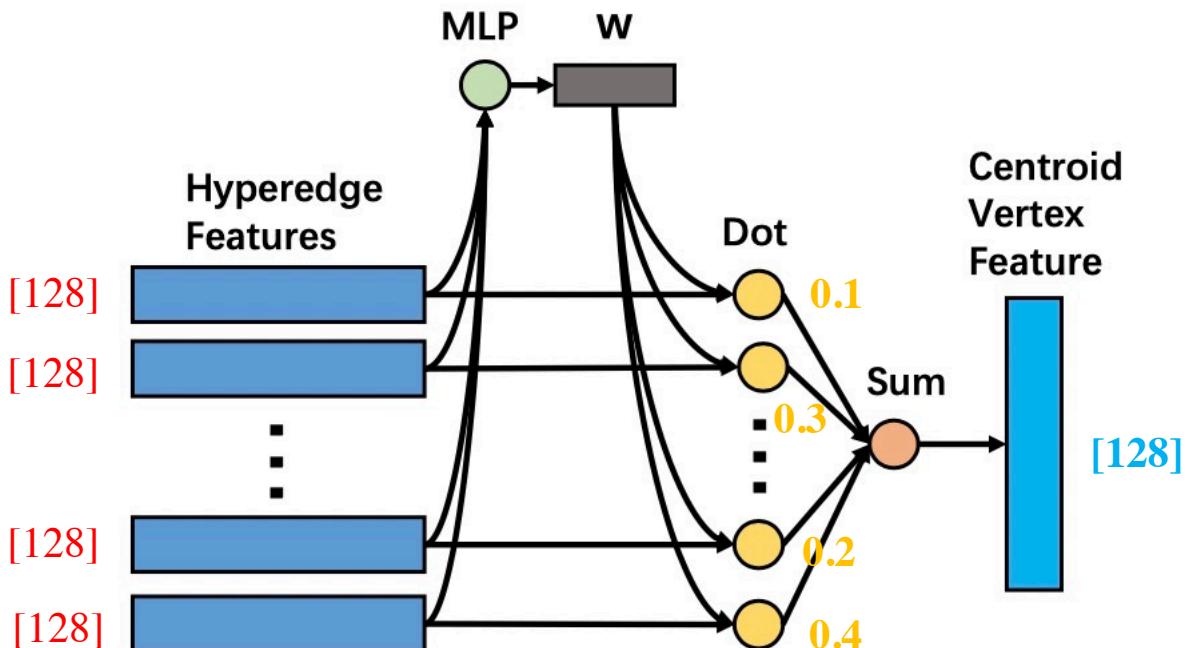
(3a)

(3b)

# Hyperedge Convolution

$$w = \text{softmax}(\mathbf{x}_e \mathbf{W} + \mathbf{b})$$

计算每个超边的权重



$$\mathbf{x}_u = \sum_{i=0}^{|Adj(u)|} w^i \mathbf{x}_e^i$$

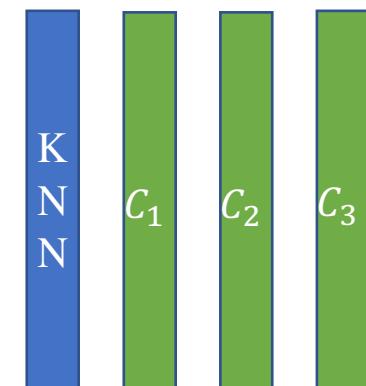
权重加权求和

$$w = \text{softmax}(\mathbf{x}_e \mathbf{W} + \mathbf{b}) \quad (4a)$$

$$\mathbf{x}_u = \sum_{i=0}^{|Adj(u)|} w^i \mathbf{x}_e^i \quad (4b)$$

表示包含节点u的所有超边集合

$$Adj(u) = \{e_1, e_2, \dots, e_{k_u}\}$$



### 3.3 Dynamic Hypergraph Neural Networks

---

**Algorithm 2** Hypergraph Convolution

---

**Input:** Input sample  $\mathbf{x}_u$ ; hypergraph structure  $\mathcal{G}$

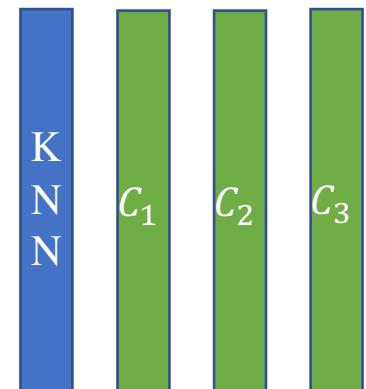
**Output:** Output sample  $\mathbf{y}_u$

```
1:  $\mathbf{xlist} = \Phi$ 
2: for  $e$  in  $Adj(u)$  do
3:    $\mathbf{X}_v = VertexSample(\mathbf{X}, \mathcal{G})$ 
4:    $\mathbf{x}_e = VertexConv(\mathbf{X}_v)$  节点特征到超边特征
5:    $\mathbf{xlist}.insert(\mathbf{x}_e)$ 
6: end for
7:  $\mathbf{X}_e = stack(\mathbf{xlist})$  组合超边特征
8:  $\mathbf{x}_u = EdgeConv(\mathbf{X}_e)$  聚合超边卷积
9:  $\mathbf{y}_u = \sigma(\mathbf{x}_u \mathbf{W} + \mathbf{b})$ 
```

---

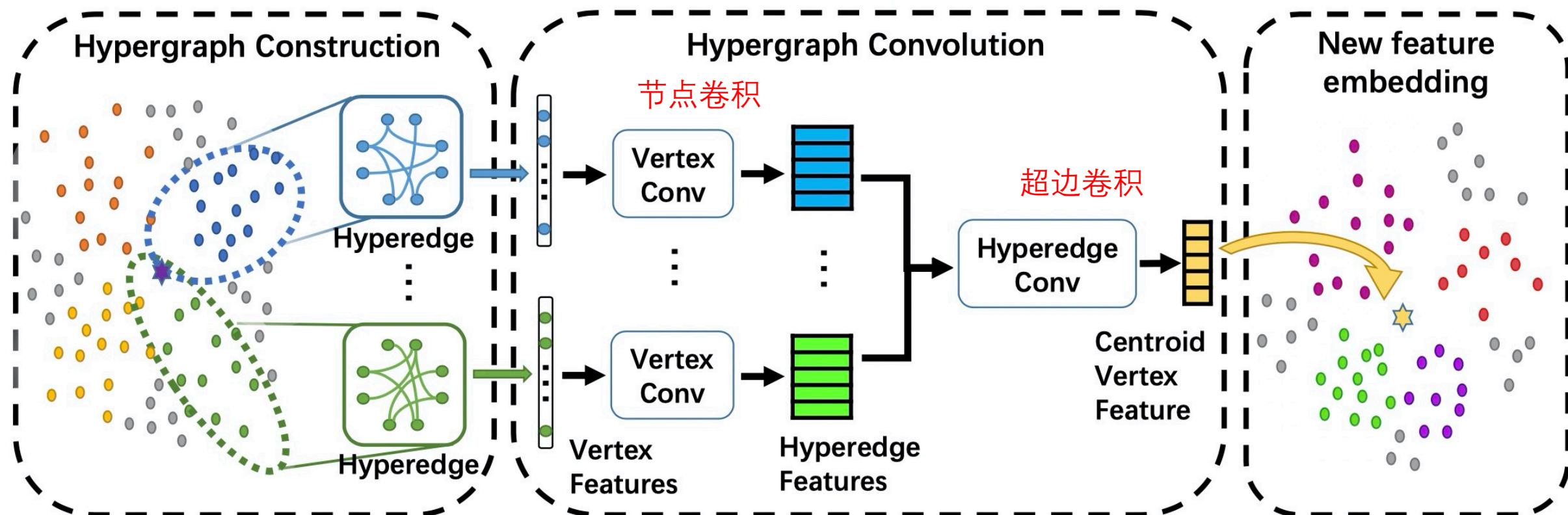
表示包含节点 $u$ 的所有超边集合

$$Adj(u) = \{e_1, e_2, \dots, e_{k_u}\}$$



KMeans

## 构建超图 (构建超边)



lr	#train	GCN	HGNN	GAT	DHGNN	Cora
std	140	81.5%	81.6%	<b>83.0%</b>	82.5%	
2%	54	<b>69.6%</b>	75.4%	74.8%	<b>76.9%</b>	
5.2%	140	77.8%	79.7%	79.4%	<b>80.2%</b>	
10%	270	79.9%	80.0%	81.5%	<b>81.6%</b>	
20%	540	81.4%	80.1%	83.5%	<b>83.6%</b>	
30%	812	81.9%	82.0%	84.5%	<b>85.0%</b>	
44%	1200	82.0%	81.9%	85.2%	<b>85.6%</b>	

准确率高    训练时间短

Method	Acc	Train Time
CBM-NB [Wang <i>et al.</i> , 2014]	71.6%	-
CBM-LR [Wang <i>et al.</i> , 2014]	79.9%	-
CBM-SVM [Wang <i>et al.</i> , 2014]	81.6%	-
HGNN [Feng <i>et al.</i> , 2018]	86.8%	2m11s
MHG_noW [Chen <i>et al.</i> , 2015]	87.3%	-
MHG [Chen <i>et al.</i> , 2015]	88.6%	-
MMHG [Chen <i>et al.</i> , 2015]	88.7%	-
Bi-MHG [Ji <i>et al.</i> , 2019]	90.0%	58.5h
DHGNN (our method)	<b>91.8%</b>	1m32s

微博数据

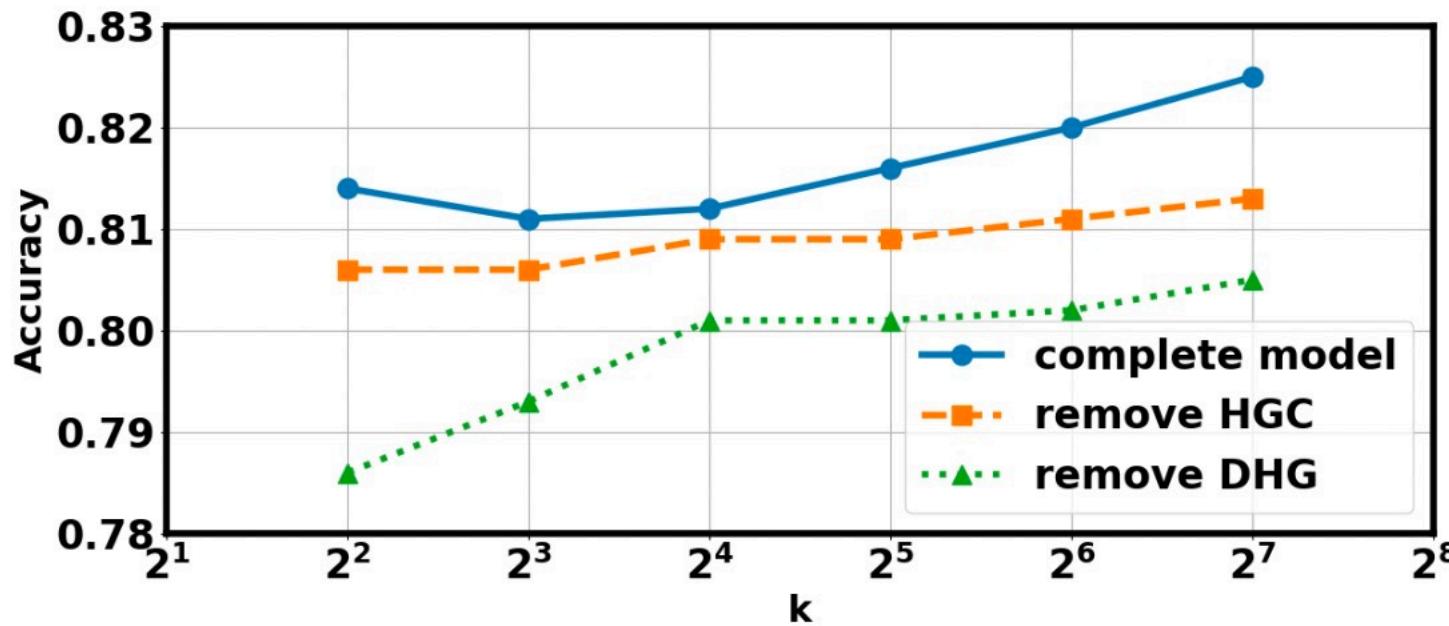
Table 2: Performance comparisons on the Microblog dataset

## Ablation experiments 消融实验

对之前提到的DHGNN的两个模块：

dynamic hypergraph construction (DHG)

hypergraph convolution (HGC)



hyperparameter  $k$  超参数k的影响 (节点邻居的个数)

随着 $k$ 的提升，能够更好地聚合邻居特征



Code

```
In[4]: G
```

```
Out[4]:
```

```
tensor([[0.2375, 0.0000, 0.0000, ..., 0.0000, 0.0000, 0.0000],  
       [0.0000, 0.3125, 0.0851, ..., 0.0000, 0.0000, 0.0000],  
       [0.0000, 0.0851, 0.2069, ..., 0.0000, 0.0000, 0.0000],  
       ...,  
       [0.0000, 0.0000, 0.0000, ..., 0.5000, 0.0000, 0.0000],  
       [0.0000, 0.0000, 0.0000, ..., 0.0000, 0.2267, 0.1600],  
       [0.0000, 0.0000, 0.0000, ..., 0.0000, 0.1600, 0.165]]
```

```
In[2]: H
```

```
Out[2]:
```

```
array([[1., 0., 0., ..., 0., 0., 0.],  
      [0., 1., 1., ..., 0., 0., 0.],  
      [0., 1., 1., ..., 0., 0., 0.],  
      ...,  
      [0., 0., 0., ..., 1., 0., 0.],  
      [0., 0., 0., ..., 0., 1., 1.],  
      [0., 0., 0., ..., 0., 1., 1.]])
```

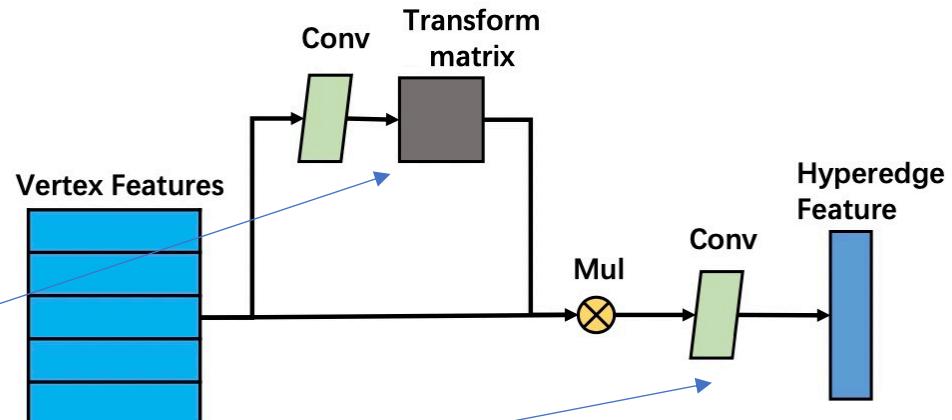
```
In[13]: edge_dict[0]  
Out[13]: [633, 1862, 2582, 0]  
In[14]: edge_dict[10]  
Out[14]: [476, 2545, 10]  
In[15]: edge_dict[1000]  
Out[15]: [281, 2543, 1325, 972, 1000]  
In[16]: edge_dict[2700]  
In[3]: H.sum  
Out[16]: [1151, 2700]  
Out[3]: 13264.0  
In[4]: H.sum(axis=1)  
Out[4]: array([4., 4., 6., ..., 2., 5., 5.])  
In[5]: H.sum(axis=0)  
Out[5]: array([4., 4., 6., ..., 2., 5., 5.])
```

## vertex convolution

```
class VertexConv(nn.Module):
    """
    A Vertex Convolution layer
    Transform  $(N, k, d)$  feature to  $(N, d)$  feature by transform matrix
    """

    def __init__(self, dim_in, k):
        super().__init__()

        self.trans = Transform(dim_in, k) #  $(N, k, d) \rightarrow (N, k, d)$ 
        self.convK1 = nn.Conv1d(k, 1, 1) #  $(N, k, d) \rightarrow (N, 1, d)$ 
```



```
class Transform(nn.Module):
    """
    A Vertex Transformation module
    Permutation invariant transformation:  $(N, k, d) \rightarrow (N, k, d)$ 
    """

    def __init__(self, dim_in, k):
        super().__init__()

        self.convKK = nn.Conv1d(k, k * k, dim_in, groups=k)
        self.activation = nn.Softmax(dim=-1)
        self.dp = nn.Dropout()
```

```

class EdgeConv(nn.Module):
    """
    A Hyperedge Convolution layer
    Using self-attention to aggregate hyperedges
    """

    def __init__(self, dim_ft, hidden):
        """
        :param dim_ft: feature dimension
        :param hidden: number of hidden layer neurons
        """

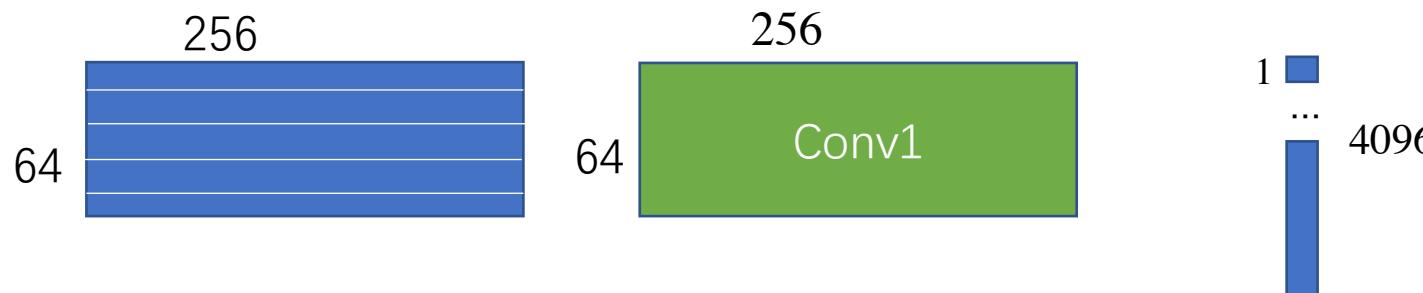
        super().__init__()
        self.fc = nn.Sequential(nn.Linear(dim_ft, hidden), nn.ReLU(), nn.Linear(hidden, 1))
    
```

$$w = \text{softmax}(\mathbf{x}_e \mathbf{W} + \mathbf{b}) \quad (4a)$$

$$\mathbf{x}_u = \sum_{i=0}^{|Adj(u)|} w^i \mathbf{x}_e^i \quad (4b)$$

In[83]: self.convKK

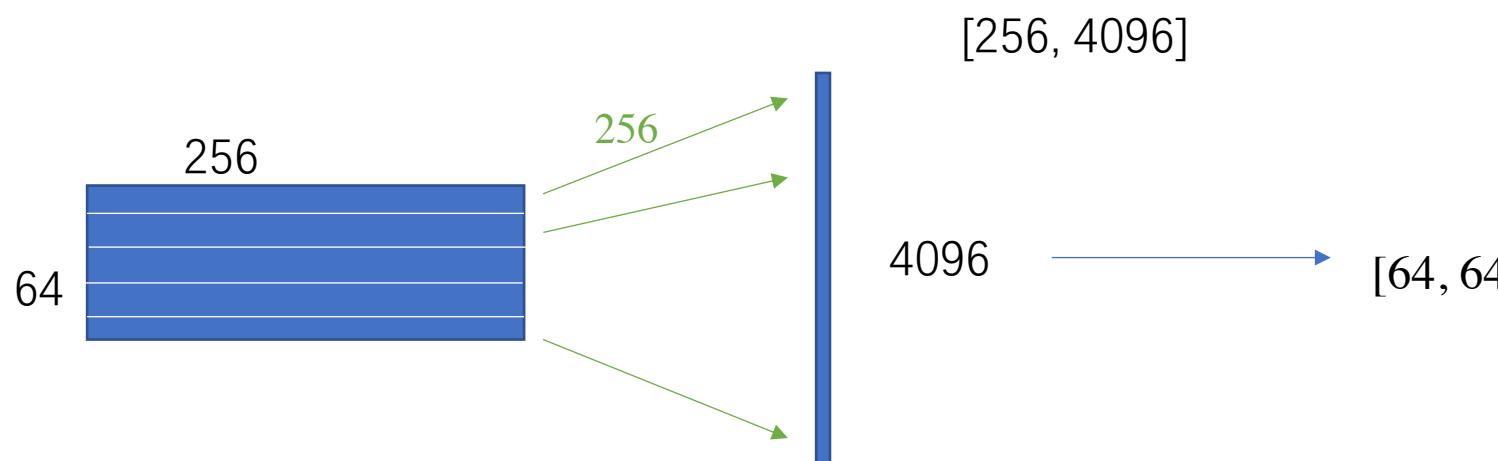
Out[83]: Conv1d(64, 4096, kernel\_size=(256,), stride=(1,),



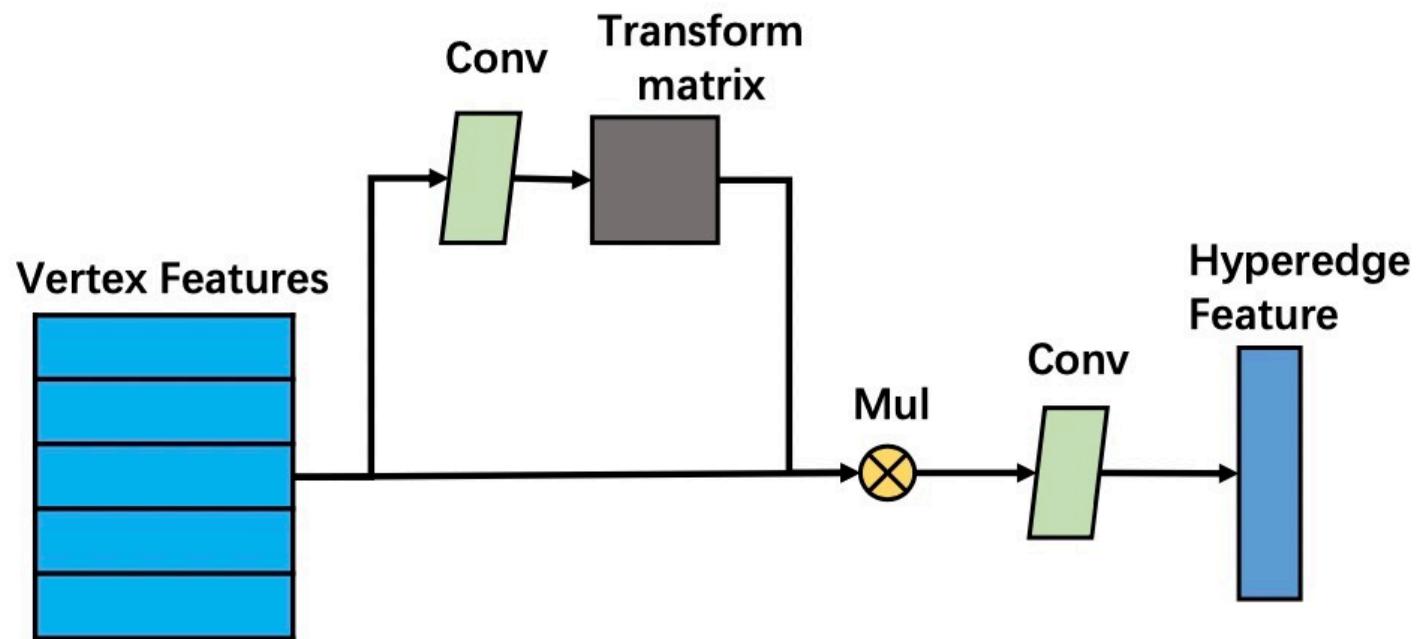
In[83]: self.convKK

Out[83]: Conv1d(64, 4096, kernel\_size=(256,), stride=(1,), groups=64)

At `groups= in_channels`, each input channel is convolved with its own set of filters (of size  $\frac{\text{out\_channels}}{\text{in\_channels}}$  ).



```
multiplier = convd.view(N, k, k) # [140, 64, 64] multiplier
multiplier = self.activation(multiplier) # 最后一个维度softmax 64个邻居进行归一化
transformed_feats = torch.matmul(multiplier, region_feats) # 64个邻居加权求和
[140, 64, 64] [140, 64, 256]
```



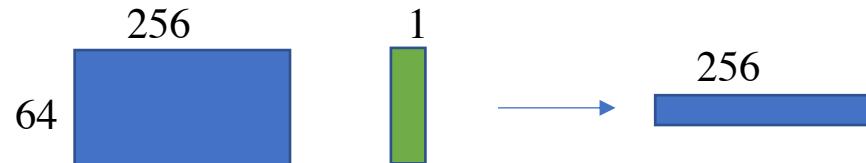
[140, 64, 256]

```
pooled_feats = self.convK1(transformed_feats)
```

[140, 1, 256]

In[111]: self.convK1

Out[111]: Conv1d(64, 1, kernel\_size=(1,), stride=(1,))



**self.convK1**

对64个邻居进行权重求和

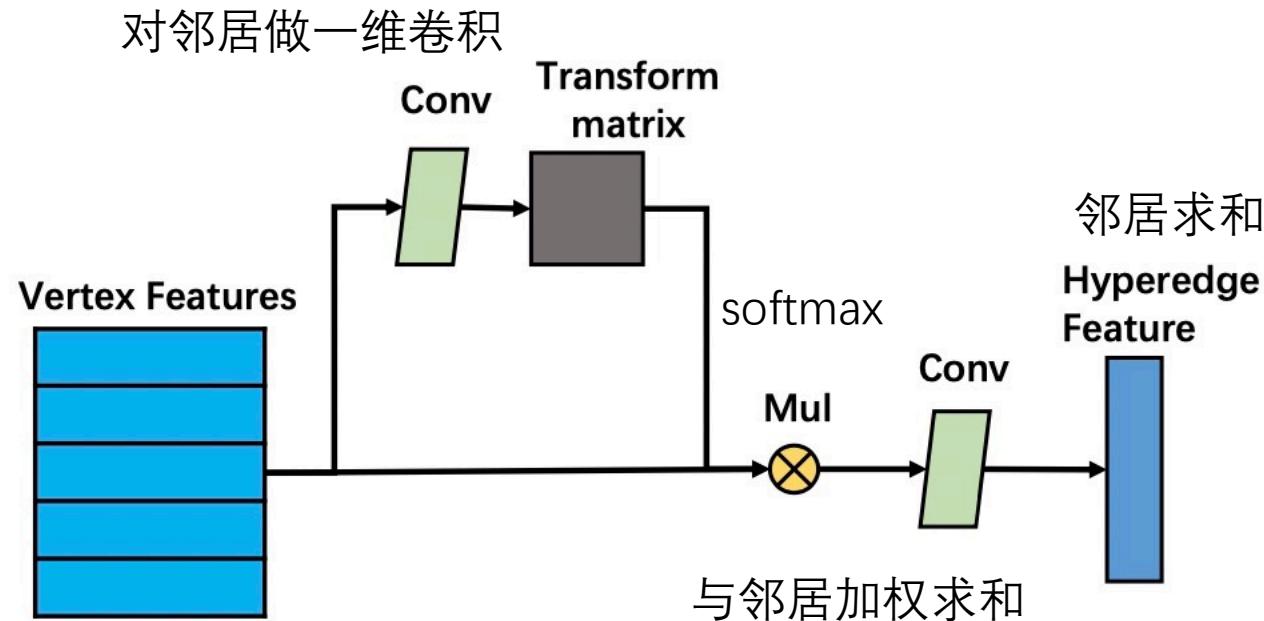
[140, 64, 256]

↓ 对64个邻居进行权重求和

[140, 1, 256]

节点的特征

## VertexConv



节点和邻居节点特征

## self.trans

[140, 64, 256]  
对每个邻居特征做1维卷积

[140, 64, 64]  
softmax

[140, 64, 64]  
按照softmax, 进行加权求和  
[140, 64, 256]

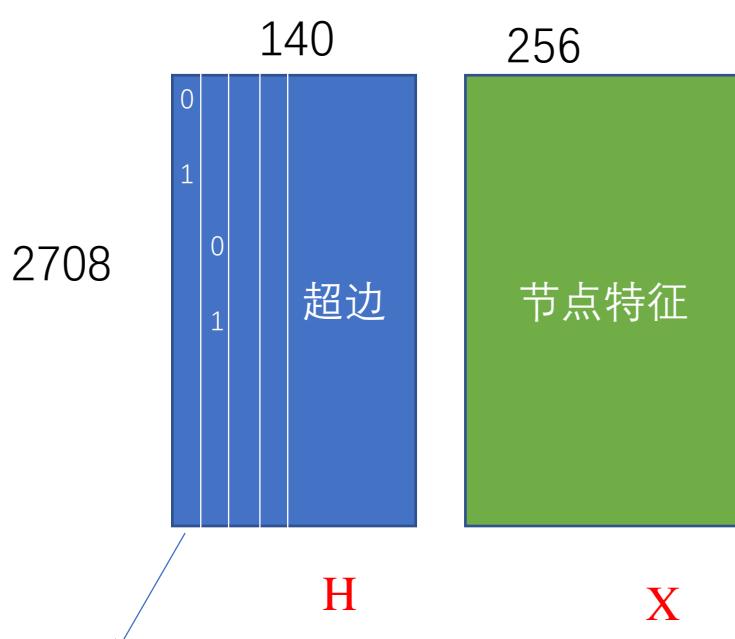
## self.convK1

[140, 64, 256]  
对64个邻居进行权重求和  
[140, 1, 256]

节点的特征

[140, 64, 256]: 140个节点的64个邻居节点，每个邻居节点的特征为256维  
140个节点的64个邻居  
一共有2708个节点，如果按照邻居构建超边

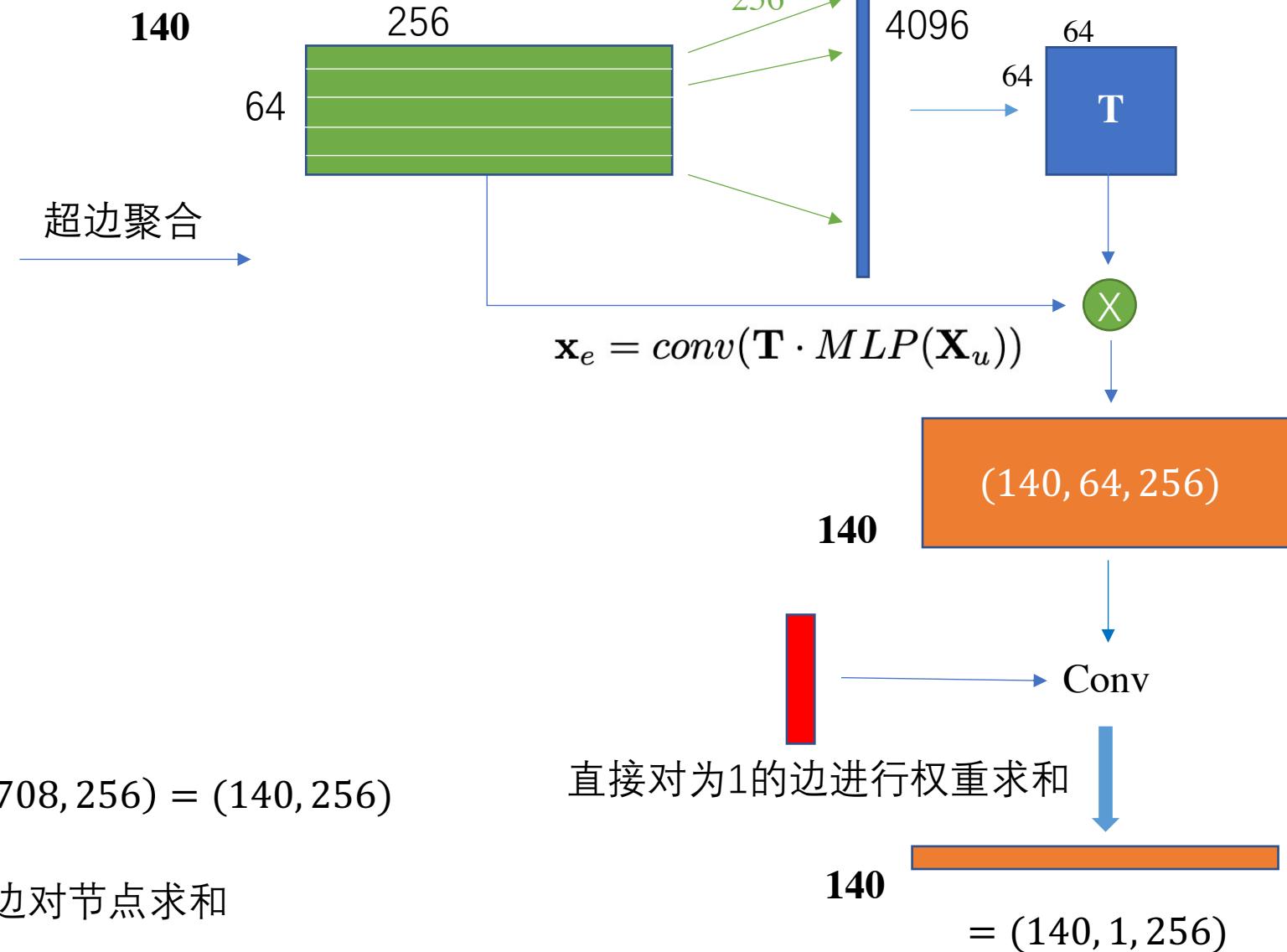
$$\mathbf{T} = MLP(\mathbf{X}_u) \quad (64, 256)$$

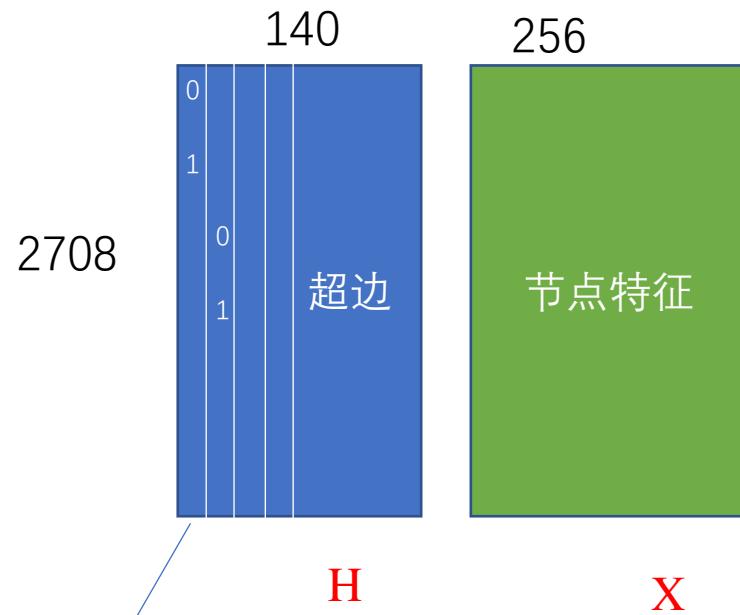


$$H^T X = (140, 2708)(2708, 256) = (140, 256)$$

HGNN

按超边对节点求和



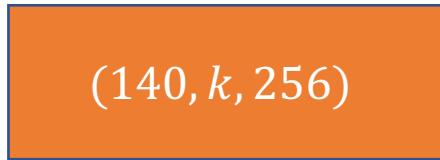


其中有64个节点连接

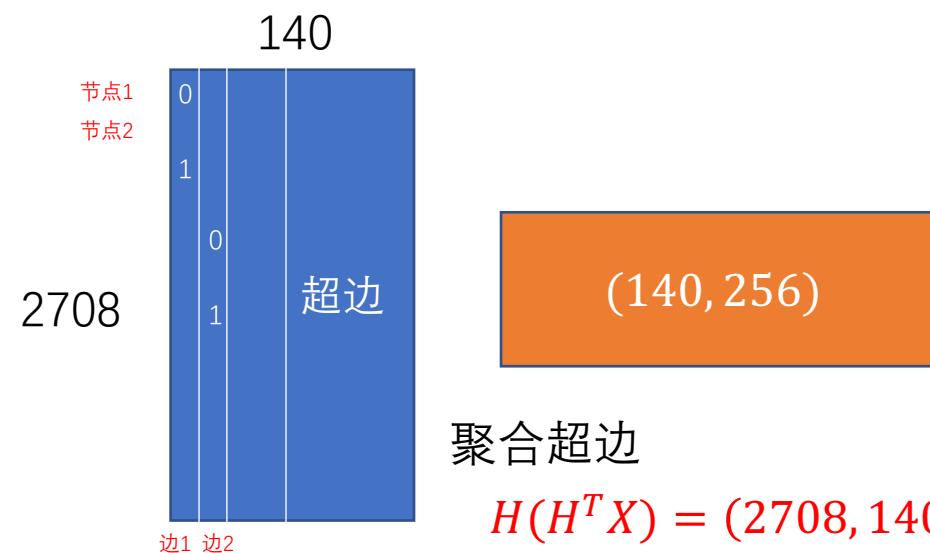
$$H^T X = (140, 2708)(2708, 256) = (140, 256)$$

**HGNN** 按超边对节点求和

k个超边构成的超图



**DHGNN** 计算超边缘特征的加权平均作为节点的特征。



$$H(H^T X) = (2708, 140)(140, 256) = (2708, 256)$$

