Applied Intelligence
https://doi.org/10.1007/s10489-021-03102-x

Embedding temporal networks inductively via mining neighborhood and community influences

Meng Liu[1] · Zi-Wei Quan[1] · Jia-Ming Wu[1] · Yong Liu[1] · Meng Han[2]

Accepted: 10 December 2021

Abstract
Network embedding aims to generate an embedding for each node in a network, which facilitates downstream machine learning tasks such as node classification and link prediction. Current work mainly focuses on transductive network embedding, i.e. generating fixed node embeddings, which is not suitable for real-world applications. This paper proposes a novel **con**tinual temporal network embedding method called ConMNCI by **m**ining **n**eighborhood and **c**ommunity **i**nfluences inductively. We propose an aggregator function that integrates neighborhood influence with community influence to generate node embeddings at any time, and introduce the idea from continual learning to enhance inductive learning. We conduct extensive experiments on several real-world datasets and compare ConMNCI with several state-of-the-art baseline methods on various tasks, including node classification and network visualization. The experimental results show that ConMNCI significantly outperforms the state-of-the-art baselines.

Keywords Temporal network · Inductive network embedding · Continual learning · Influence mining

1 Introduction

Social networks, protein networks, and e-commerce networks [54] are broadly existed in the real-world systems. Recently, both industrial and academic are becoming increasingly interested in mining information on large-scale networks [45]. Considering the complex and irregular features of network patterns, learning network data is challenging [18]. As a popular field, network embedding, also known as network representation learning (NRL), aims to model network by mapping nodes to a low-dimensional space [5, 8, 50]. NRL can be used in many real-world domains. The node embeddings generated by NRL could be used for downstream machine learning tasks such as node classification and link prediction [4, 36].

Network Representation Learning has drawn considerable attention due to the wide range of application. Current research works usually focus on transductive learning, which generate fixed node embeddings by directly training the whole network in its final state [44, 48]. However, in the real world, networks change frequently, with new nodes being added and new interactions happening constantly. Therefore, many real-world tasks require node embeddings to be updated alongside network changes. Thus transductive learning will have to retrain the whole network to obtain new node embeddings, which is not feasible for real-world networks, especially large-scale networks.

Unlike transductive learning, inductive learning [48] no longer focuses on the network's final node embeddings but attempts to learn a model that can dynamically generate node embeddings over time even for unseen nodes.

Inductive learning usually learn node embeddings in temporal networks. In a temporal network, edges are annotated by sequential interactive events between nodes. Many real-world networks contain interaction time between nodes, such as the bitcoin trading network, the citation network, etc. Combining this temporal information allows researchers to learn the dynamics of individual nodes in the network more comprehensively. However, how to capture temporal information and mine network changes over time is a challenging problem.

To address the challenge in temporal networks, in this paper, we propose a **con**tinual inductive network embedding

✉ Yong Liu
liuyong123456@hlju.edu.cn

Meng Han
menghan@kennesaw.edu

[1] Heilongjiang University, Harbin, China

[2] Zhejiang University, Hangzhou, Zhejiang, China

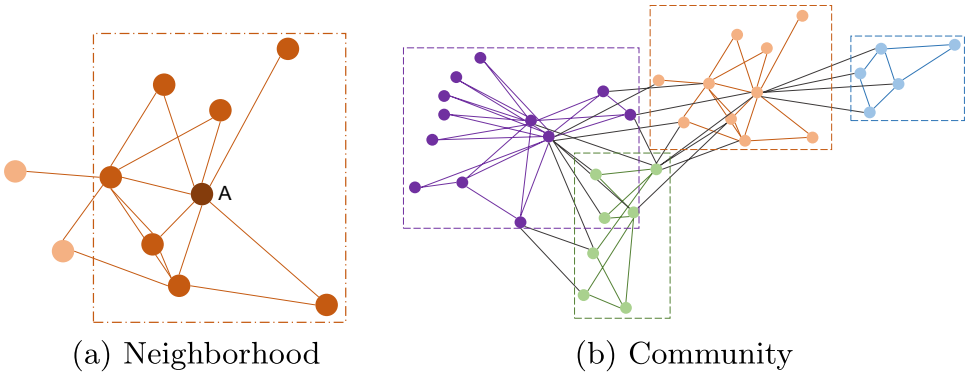


Fig. 1 Neighborhood and community

method called ConMNCI by **m**ining **n**eighborhood and **c**ommunity **i**nfluences for each node.

First, we give a brief introduction to neighborhood and community. As shown in Fig. 1a, the nodes that are directly connected to node *A* constitute its neighborhood, while the two nodes with the lightest color that are not directly connected to node *A* are not in its neighborhood. As shown in Fig. 1b, the nodes with similar behavior patterns or preferences constitute a community. In this figure, we label four different communities with four colors.

For mining ***neighborhood*** influence, it is obvious that the historical neighbors of a node will influence its future interactions. This influence is not only related to neighbor's own characteristics but also related to their interaction time, thus we should combine them to reflect the influence. We calculate the affinity between node and neighbors through their embeddings, and also encode the interaction time into embeddings.

For mining ***community*** influence, we introduce the concepts of community detection [11, 51] and community embedding [6] by defining several communities and learning an embedding for each community. Given a node, it may have different closeness to different communities. The deeper closeness a node is to a community, the more influence this community has on the node. For example, users on Twitter are influenced differently by different topics depending on their interests, and consumers also have different preferences for different products.

After mining neighborhood and community influences, we devise a new aggregator function to obtain node embeddings by modifying the Gated Recurrent Unit (GRU) framework [7, 16]. In this way, we can obtain effective node embeddings at any time.

Finally, we introduce the idea from continual learning [47] used to alleviate catastrophic forgetting problem to enhance inductive learning. Specifically, after the optimization of each training batch, we select experience embeddings from this batch and store them into the experience buffer. During subsequent training, we sample the experience embeddings from the buffer as a new constraint, thus ensuring that ConMNCI consolidaes the existing knowledge learned from historical data.

We evaluate ConMNCI on mutiple real-world datasets with various tasks, including node classification, link prediction, and visualization, etc. The results demonstrate that ConMNCI can achieve better performance than state-of-the-art baseline methods, which illustrates the capacity of ConMNCI in capturing network changes.

Compared with our conference version [26] that has been accepted as a short paper in SIGIR 2021, this paper has been extended as follows. (1) We are the first to introduce the idea from continual learning to enhance inductive learning, and use the self-attention mechanism and Kullback-Leibler divergence for experience retaining. (2) In Community Influence part, we discuss both overlapping and non-overlapping community patterns in a complementary way. (3) In Neighborhood Influence part, we analyze the effect of length of historical neighbor sequence on embedding performance by previous work and experiments. (4) We conduct more experiments than the conference version, including link prediction, parameter sensitivity, and ablation study.

Our main contributions can be summarized as follows.

(1) We propose ConMNCI to inductively generate effective node embeddings at any time in temporal networks by utilizing the positional encoding technology to initialize node embedding, which can speed up the convergence speed of training process.
(2) We incorporate community detection and community embedding to mine community influence in temporal networks with the consideration of both overlapping and non-overlapping community patterns.
(3) We propose a novel GRU-based aggregation function to aggregate neighborhood and community influences. We also introduce the idea from continual learning to further improve the effect of inductive learning.
(4) We conduct extensive experiments on several real-world datasets to demonstrate the effectiveness of our proposed method ConMNCI. The results show that

ConMNCI significantly outperforms the state-of-the-art baselines.

The rest of this paper is organized as follows. In Section 2, we summarize recent research related to our work. In Section 3, we describe our proposed method in detail. In Section 4, we provide the experimental results and analysis. In Section 5, we conclude this work and discuss future work.

The source code and data can be downloaded from https://github.com/MGitHubL/MNCI.

2 Related work

Served as an important role in downstream machine learning tasks, more and more work was carried out in network representation learning (NRL) field. In the meantime, NRL is developing in several directions. ***Based on the network type***, we can divide NRL into static network learning and dynamic network learning. ***Based on the training goal***, we can also divide NRL into transductive learning and inductive learning.

2.1 Static and dynamic network

Static network means that the network is fixed where neither topological structure nor node attribute changes over time. In early stage, researchers usually focus on the topological structure of the network. They first obtain the adjacency matrix of the network, then use random walk or matrix decomposition [37] to learn node embeddings. To name a few, DeepWalk first applies random walks to generate sequences of nodes over the network and then employs the Skip-Gram [33] model to learn node embeddings [40]. LINE focus on the first-order and second-order proximity among nodes in the network [46]. node2vec uses the random walk procedure to balance the breadth-first and depth-first search strategy [13]. SDNE effectively captures highly non-linear network structure to generate node embeddings [53]. GraphSAGE learns a function to generate node embeddings by sampling and aggregating features from nodes' local neighborhood [14]. RGCN applies the idea of ResNet into GCNs and construct RGCN to learn the possibility of linkage between two nodes [41]. SGL supplements the classical supervised task of recommendation with an auxiliary self-supervised task, which reinforces node representation learning via self-discrimination [55].

Unlike static network, **dynamic network** means that a network contains dynamic changes that can help researchers learn the evolution of the network structure and obtain more effective embeddings. In the early stages of dynamic network, researchers usually divide the network into several states based on the timestamps. The network state at each timestamp is called a ***static snapshot*** of the dynamic network. By comparing the differences between multiple static snapshots, researchers can learn the evolutionary patterns of the network over time. To name a few, DySAT computes node embeddings through joint self-attention along two dimensions of structural neighborhood and temporal dynamics [43]. EvolveGCN [38] uses a RNN to estimate the GCN parameters for the future snapshots.

Since there are many interactions in the interval between two static snapshots, it is difficult to accurately represent network changes, researchers began focusing on learning node embeddings in ***temporal network*** with chronological interactive events. To name a few, CTDNE applies a biased or unbiased random walk procedure to combine temporal information into node embeddings [34]. HTNE uses the Hawkes process to capture the influence of historical neighbors on the current node [63]. JODIE applies RNNs to estimate the future embedding of nodes and introduces a novel projection operator which learns to estimate node embeddings at any time in the future [24]. TRRN employs transformer-style self-attention to reason over a set of memories and considers both updated memories and different factors that influence node behaviors [57]. HTGN follows the concise and effective GRNN framework and leverages the power of hyperbolic graph neural network and facilitates hierarchical arrangement to capture the topological dependency [60].

2.2 Transductive and inductive learning

Transductive learning generates fixed node embeddings by directly optimizing the final state of the network. Most of the existing approaches for generating node embeddings are inherently transductive. However, the disadvantage of transductive learning is that when the network changes, these approaches need to retrain the whole network to generate new node embeddings, which involves expensive calculations. Thus, transductive learning is not suitable for generating new node embeddings in dynamic networks.

Different from transductive learning, **inductive learning** no longer generates fixed node embeddings but focuses on learning a model that can generate node embeddings at any time. When a new node is added, the model can directly calculate the new node embedding using the node's features and other information. To name a few, GraphSAGE learns a function to generate node embeddings by sampling and aggregating features from nodes' local neighborhood [14]. DyREP uses RNNs to learn node embeddings while its loss function is built upon temporal point process [48]. TGAT leverages GAT to extract node representations where the nodes' neighbors are sampled from the history and encodes temporal information [59]. ER-GNN stores knowledge from previous tasks as experiences and replays them when

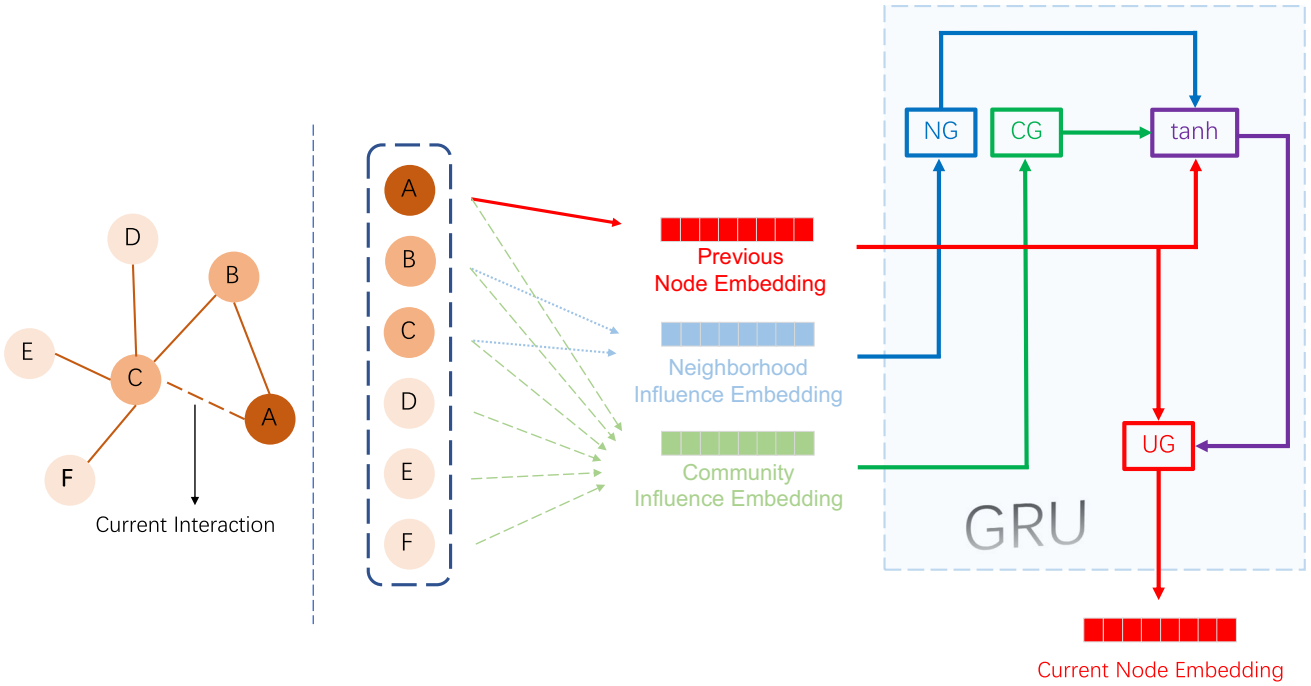


Fig. 2 ConMNCI Framework

learning new tasks to mitigate the catastrophic forgetting issue [61].

According to the above classification, our method ConMNCI belongs to inductive learning in temporal networks. In real-world network datasets, temporal networks can represent network changes accurately, and inductive learning can capture these changes flexibly. Therefore, our method ConMNCI is more suitable for generating effective node embeddings, and we will introduce it in detail below.

3 Method

3.1 Problem definition

First, we introduce the basic framework of ConMNCI in Fig. 2. We use one interaction between two nodes as an example to explain the process of generating node embeddings inductively. Assume that node A interacts with node C at the current moment. In order to update the node embedding of A, we feed A's embedding before interaction, the neighborhood influence embedding and the community influence embedding into the GRU to generate A's node embedding after interaction. In GRU, NG, CG, and UG are called neighborhood reset gate, community reset gate, and update gate, which control the update of neighborhood influence, community influence, and node embedding, respectively.

Then, according to the time information of node interaction, we can formally define the temporal network.

Definition 1 (Temporal Network) When two nodes interact, it will always be accompanied by a clear timestamp. A temporal network can be defined as a graph $G = (V, E, T)$, where V and E denote the set of nodes and edges, and T denotes the set of interactions. Given an edge $e(u, v)$ between node u and v, there is at least one interaction matching $e(u, v)$, i.e., $T(u, v) = \{(u, v, t_1), (u, v, t_2), \cdots, (u, v, t_n)\}$.

In a temporal network, the data is stored in the tuple of (u, v, t), where there is one interaction between node n and v at t. The meaning of interaction varies in different networks. For example, an interaction in a citation network is a literature citation, an interaction in an email network is an email correspondence, and an interaction in a commerce network is a commodity purchase.

In particular, we define each interaction can be considered as an edge constructed in two nodes, i.e., one interaction is one edge. In this case, two nodes may interact multiple times, and these interactions can be ordered by timestamp. When two nodes interact, we call them neighbors. The historical neighbor sequence of a node can be defined as follows.

Definition 2 (Historical Neighbor Sequence) For each node u, there is a historical neighbor sequence H_u, which stores the historical interactions of u up to the current

Table 1 Notation

Notation	Description
$z_u^{t_n}$	embedding of node u at time t_n
H_u	historical neighbor sequence of u
$z_{(u,i)}^{t_i}$	temporal embedding of edge $e = (u, i, t_i)$
$a_{(u,i)}$	affinity weight between two nodes
$a_{(u,c_k)}$	affinity weight between node and community
z_{c_k}	embedding of the k^{th} community c_k
$NE_u^{t_n}$	neighborhood influence embedding of u at t_n
$CO_u^{t_n}$	community influence embedding of u at t_n
$\delta_u^{NE}, \delta_u^{CO}$	learnable parameter of $NE_u^{t_n}$ and $CO_u^{t_n}$
$\mathcal{E}, R, \mathbb{B}$	experience embeddings, number, and buffer
B	batch of training data
β	dynamic weight in loss function $L(u, \mathcal{E})$
$L(u, v)$	loss function of neighbor nodes
$L(c)$	loss function of community detection
$L(\mathcal{E})$	loss function of continual learning

moment, i.e., $H_u = \{(v_1, t_1), (v_2, t_2), \cdots, (v_n, t_n)\}$. Each tuple in the sequence represents an event, i.e., node v_i interacts with u at time t_i.

During the interaction of nodes, their neighbors tend to influence their behavior, which we call neighborhood influence. In addition, nodes in a network may also be influenced by the communities. Graph theory proposes two rules to define the relationship between nodes and communities [10, 11]. According these rules, we can define the community as follows.

Definition 3 (Community) Communities are the subgraphs in a network where (1) nodes in a community are densely connected, and (2) nodes in different communities are sparsely connected. Here, we define K communities $C = \{c_1, \cdots, c_K\}$ divided from a network G, where c_k is the k^{th} community ($k \in \{1, \cdots, K\}$). Each node may may belongs to one or more communities.

Our goal is to capture neighborhood and community influences to generate effective node embeddings for downstream tasks. The notations and descriptions appearing in this paper are shown in Table 1.

3.2 Node Embedding Initialization

For network representation learning (NRL) methods, node embeddings need to be initialized before training. Unlike the random initialization used by common methods, we propose a **time positional encoding** technology to generate node embeddings by using time information, which can speed up the convergence speed of training process. Note that the positional encoding part is only used to generate initial node embeddings and does not participate in the subsequent update process.

To the best of our knowledge, the idea of positional encoding [52] is first proposed in Natural Language Processing (NLP) field. Considering that in many real-world scenarios, most nodes have no clear feature information for researchers to obtain prior knowledge. In this case, the initial time when node u joins a network will be very useful for u, which should be further exploited.

According to the initial time order, we can obtain an ordered node sequence $S_{node} = \{u_1, u_2, \cdots, u_n\}$. Then, we use sine and cosine functions with different frequencies to define the encoding on each dimension in the node embeddings.

$$\begin{aligned} PE_{(u,2i)} &= \sin(u/10000^{2i/d}) \\ PE_{(u,2i+1)} &= \cos(u/10000^{2i/d}) \end{aligned} \tag{1}$$

Where u is the u^{th} node position number in S_{node}, d is the dimension size of node embedding, $2i$ is the $(2i)^{th}$ dimension in node embedding, and $PE_{(u,2i)}$ is the encoding for the $(2i)^{th}$ dimension of the u^{th} node embedding in S_{node}. Here each dimension corresponds to a sinusoid, and the wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$ [52]. We select sine and cosine functions for coding because they have the following properties.

$$\begin{aligned} \sin(u+k) &= \sin u \cos k + \cos u \sin k \\ \cos(u+k) &= \cos u \cos k - \sin u \sin k \end{aligned} \tag{2}$$

Let PE_{u+k} and PE_u be the embeddings for the $(u+k)^{th}$ node and the u^{th} node in S_{node}, respectively. According to this property, for any fixed offset k, PE_{u+k} can be formalized into a linear function of PE_u, which means that the function in (1) can capture the relative time positions of nodes. In this way, we can obtain the initial node embedding $z_u^{t_0}$ of node u at the initial time t_0 as follows.

$$z_u^{t_0} = PE_u = [PE_{(u,0)} \oplus PE_{(u,1)} \oplus \cdots \oplus PE_{(u,d-1)}] \tag{3}$$

Where $\oplus$ denotes concatenation operator, and $PE_{(u,i)}$ represents a position value. After concatenating each position value to obtain the initial node embedding, we can mine neighborhood and community influences. Note that both influences of a node are calculated every time it interacts with other nodes, thus we omit the time superscript by default in the following unless we want to distinguish two variables with different timestamps.

3.3 Neighborhood Influence

We believe that after an interaction occurs between node u and v, node v will influence the future interactions

of node u with other nodes, and u will also influence v. Given a node u, we assume that the influence on u is not only related to neighbor's own characteristics, but also related to their interaction time. Therefore, to mine the neighborhood influence on each node, we will analyze its neighbors' embedding and interaction time, respectively.

Note that in real-world networks, the length of neighbor sequence may vary significantly over all nodes. To keep the computational pattern of each batch fixed and more efficient, we fix the sequence length l and select the latest L neighbors for each node instead of using full neighbors. Referencing to previous works [17, 27, 63] and our experiments, the experience value of sequence length L is 5, we will study the sensitivity of the hyperparameter L in experiments.

Affinity weight We assume that there is an affinity between any two nodes, which reflects the closeness of their relationship. Given a node u and its neighbor sequence H_u, we can calculate u's affinity to different neighbors. After normalizing these affinities, the affinity weight $a_{(u,i)}$ for neighbor i on node u can be calculated as follows.

$$a_{(u,i)} = \frac{\sigma(-\|z_u - z_i\|^2)}{\sum_{i' \in H_u} \sigma(-\|z_u - z_{i'}\|^2)} \tag{4}$$

Where σ is the sigmoid function, H_u is node u's historical neighbor sequence. We use negative squared Euclidean distance to measure the affinity between two embeddings.

Temporal Embedding In temporal networks, network structure and node behavior will evolve over time. Thus, learning temporal information is an important way to capture the evolutionary process of neighborhood influence. In this stage, we learn a temporal embedding for two nodes based on their interactive timestamp. Given an interaction (u, i, t_i), the temporal embedding $z^{t_i}_{(u,i)}$ between two nodes at time t_i can be calculated as follows.

$$z^{t_i}_{(u,i)} = F(t_c - t_i) \tag{5}$$

Where t_c is the current time, $F(t)$ is the encoding function. For $F(t)$, we adopt random Fourier features to encode time [3, 31] which may approach any positive definite kernels according to the Bochner's theorem [54, 58, 59].

$$F(t) = [cos(\omega_1 t), sin(\omega_1 t), \cdots, cos(\omega_{d/2} t), sin(\omega_{d/2} t)] \tag{6}$$

Where $\omega = \{\omega_1, \cdots, \omega_{d/2}\}$ is a set of learnable parameters to ensure that the dimension size of temporal embeddings and node embedding are the same as d.

Neighborhood influence embedding Combining affinity weight and temporal embedding, the neighborhood influence embedding $NE^{t_n}_u$ of u at time t_n can be calculated.

$$NE^{t_n}_u = \delta^{NE}_u \sum_{i \in H_u} a_{(u,i)} z^{t_i}_{(u,i)} \odot z^{t_{n-1}}_i \tag{7}$$

Where δ^{NE}_u is a learnable parameter that regulates u's neighborhood influence embedding, $z^{t_{n-1}}_i$ is the embedding of u's neighbor i at time t_{n-1}, $\odot$ denotes element-wise multiplication. To calculate the influence embedding of the current timestamp, we need to use the node embedding of the previous timestamp, which will be introduced later.

3.4 Community Influence

Here we introduce the concepts of community detection and community embedding to mine community influence. Community detection, or more specifically, clustering nodes based on similar behavior or structure, helps us understand the inherent influences and patterns of networks [11]. In real-world networks, nodes in the same community tend to have similar behavior patterns.

Community detection in temporal networks is more challenging than traditional community detection, because community assignments and embeddings will change as the network evolves. In this paper, we define K communities $C = \{c_1, ..., c_K\}$ and learn an embedding z_{c_k} for each community c_k ($k \in \{1, \cdots, K\}$), where K is a hyperparameter. Given a node u, it may have different affinities to these communities. The deeper affinity u is to a community c_k, the more likely u is to belong to c_k, and the more influence c_k has on u.

For node u, we calculate its affinity with all communities. Then we normalize these affinities to obtain the affinity weights of different communities on u. In this case, a community c_k's affinity weight $\omega_{(u,c_k)}$ on u can be calculated. Here we also use negative squared Euclidean distance to measure the affinity between two embeddings.

$$a_{(u,c_k)} = \frac{\sigma(-\left\|z_u - z_{c_k}\right\||^2)}{\sum_{c_{k'} \in C} \sigma(-\left\|z_u - z_{c_{k'}}\right\|^2)} \tag{8}$$

Unlike general community detection methods that simply assign nodes to communities, our insight for assigning nodes is to have the capability to represent the membership strength of nodes to communities over time. By calculating the affinity weight between each community and node, we are able to obtain the community assignment at any time.

It is worth noting that we have considered two community patterns, i.e., non-overlapping communities and overlapping communities.

Non-overlapping communities It means that a node will only belong to the community with the greatest affinity. In this case, if a community c_k has the the highest affinity weight to node u at time t_n, after updating u's embedding from $z_u^{t_{n-1}}$ to $z_u^{t_n}$, we will dynamically update c_k's embedding, i.e., we consider that u belongs to c_k at time t_n.

$$z_{c_k} := z_{c_k} - z_u^{t_{n-1}} + z_u^{t_n} \tag{9}$$

Based on the non-overlapping pattern, we calculate a distribution $p(c_k|u)$ for each node u [45], where $p(c_k|u) = a_{(u,c_k)}$. Since a node will only belong to one community, the optimization objective for the community part is to maximize the affinity of node u with community c_k it belongs to.

$$L(c) = \sum_{u\in V} \max_{c_k\in C}(\log p(c_k|u)) = \sum_{u\in V} \max_{c_k\in C}(\log a_{(u,c_k)}) \tag{10}$$

Overlapping communities It means that a node will belong to more than one community, and the node has different affinities to different communities. In this case, each community embedding will be updated with a different affinity weight in combination with node u's embedding z_u.

$$\text{for each } c_k \in C, \quad z_{c_k} := z_{c_k} - a_{(u,c_k)} \times (z_u^{t_n-1} - z_u^{t_n}) \tag{11}$$

Based on the overlapping pattern, in addition to calculating the distribution $p(c_k|u)$ for each node u, we also calculate a distribution $p(v|c_k)$ for each community c_k [45].

$$p(v|c_k) = \frac{\sigma(-\left\|z_v - z_{c_k}\right\|^2)}{\sum_{v'\in V}\sigma(-\left\|z_{v'} - z_{c_k}\right\|^2)} \tag{12}$$

Since a node belongs to more than one community, it means that a node will interact with different nodes based on different community contexts. Thus, the process of nodes interacting and becoming neighbors can be formulated in a probabilistic way to optimize the generation of community embeddings.

$$L(u,c) = \sum_{u\in V}\sum_{c_k\in C}\sum_{v\in H_u} \log p(v|c_k)p(c_k|u) \tag{13}$$

Here we choose the pattern of non-overlapping communities by default, which depends on the datasets used for the experiment.

Community influence embedding Finally, the community influence embedding $CO_u^{t_n}$ of node u at time t_n can be calculated, where δ_u^{CO} is a learnable parameter that regulates u's community influence embedding.

$$CO_u^{t_n} = \delta_u^{CO} \sum_{c_k\in C} a_{(u,c_k)} z_{c_k} \tag{14}$$

It is worth noting that the *community influence embedding* represents the community influence of a single node, while the *community embedding* represents a single community, they are not the same.

3.5 Aggregator Function

The GRU network can capture the temporal patterns of sequential data by controlling the aggregation degree of different information and determining the proportion of historical information to be reversed [7, 56]. In this paper, we extend GRU to devise an aggregator function, which combines neighborhood and community influences with the node embeddings at the previous timestamp to generate the node embeddings at the current timestamp. The aggregator function is defined as follows.

$$UG_u^{t_n} = \sigma(W_{UG}[z_u^{t_{n-1}} \oplus NE_u^{t_n} \oplus CO_u^{t_n}] + b_{UG}) \tag{15}$$

$$NG_u^{t_n} = \sigma(W_{NG}[z_u^{t_{n-1}} \oplus NE_u^{t_n} \oplus CO_u^{t_n}] + b_{NG}) \tag{16}$$

$$CG_u^{t_n} = \sigma(W_{CG}[z_u^{t_{n-1}} \oplus NE_u^{t_n} \oplus CO_u^{t_n}] + b_{CG}) \tag{17}$$

$$\tilde{z}_u^{t_n} = \tanh(W_z[z_u^{t_{n-1}} \oplus (NG_u^{t_n} \odot NE_u^{t_n}) \oplus (CG_u^{t_n} \odot CO_u^{t_n})] + b_z) \tag{18}$$

$$z_u^{t_n} = (1 - UG_u^{t_n}) \odot z_u^{t_{n-1}} + UG_u^{t_n} \odot \tilde{z}_u^{t_n} \tag{19}$$

Here σ is the sigmoid function, $\oplus$ denotes concatenation operator, $\odot$ denotes element-wise multiplication. $NE_u^{t_n}$, $CO_u^{t_n}$ and $z_u^{t_n}$ are neighborhood influence embedding, community influence embedding and node u's embedding at time t_n, respectively. $W_{UG}, W_{NG}, W_{CG}, W_z \in \mathbb{R}^{d\times 3d}$, $b_{UG}, b_{NG}, b_{CG}, b_z \in \mathbb{R}^d$ are learnable parameters, $UG_u^{t_n}, NG_u^{t_n}, CG_u^{t_n} \in \mathbb{R}^d$ are called update gate, neighborhood reset gate, and community reset gate, respectively.

Here we divide the reset gate in GRU into two reset gates, i.e., neighborhood reset gate $NG_u^{t_n}$ and community reset gate $CG_u^{t_n}$. We use $NG_u^{t_n}$ and $CG_u^{t_n}$ to control the reservation degree of neighborhood and community influence embeddings, respectively. Then, we aggregate the node embedding at the previous timestamp with reserved neighborhood and community influence embeddings to obtain a new hidden state $\tilde{z}_u^{t_n}$ at the current timestamp. Finally, we use $UG_u^{t_n}$ to control the reservation degree of historical information. Based on the node embedding $z_u^{t_{n-1}}$ at the pervious timestamp and the new hidden state $\tilde{z}_u^{t_n}$ at the current timestamp, we can obtain a node embedding $z_u^{t_n}$ at the current timestamp. In this way, we can calculate node embeddings inductively.

Note that during the training process, we process one batch of data at a time and update all node embeddings in this batch. This has the same effect as updating the

embedding for one node at a time, unless a node interacts multiple times in the same batch. This is because that when there are multiple interactions about the same node in a batch, only last interaction will be used to update the embedding of this node, and other interactions will be discarded. But when the batch size is small (batch size $\leq$ 128), this problem will rarely occur and can be ignored.

On the other hand, when one batch of data is fed into GRU, how to calculate the current embedding based on the previous embedding? Suppose that the interaction sequence of node u is $\{(u, v_1, t_1), (u, v_2, t_2), (u, v_3, t_3), ..., (u, v_n, t_n)\}$. When we process the interaction (u, v_3, t_3), the current timestamp for u is t_3, while the previous timestamp for u is t_2. In the real training process, we only need to save all node embeddings of the previous timestamp. More specially, we create a global tensor to save all node embeddings and write a node embedding back to the tensor only when this node embedding is updated in training. At any time, node embeddings in this tensor can be considered as the embeddings at the previous timestamp.

3.6 Continual Learning

Continual learning can learn over time continually by capturing and transforming new information while retaining previous knowledge or experiences [9, 39, 47]. The main challenge of continual learning is to be catastrophic forgetting in the learninig process [12, 42], i.e., a model will forget old experiences in the process of learning new information [29, 30, 39]. In the worst scenario, the old knowledge learned by the model will be completely overwritten by the new knowledge.

To alleviate the catastrophic forgetting problem, researchers focus on learning multiple tasks sequentially [20, 32]. In particular, they store knowledge from previous tasks as experiences and replays them when learning new tasks [61]. Inductive learning (**IL**) has many similarities to continual learning (**CL**).

(1) CL divides a task into multiple subtasks with the same objective, and each subtask corresponds to a sub-dataset. IL also keeps the same objective during training, and learns the model from the dataset in batches.
(2) CL generalizes the experience from the previous tasks for the subsequent tasks. IL also summarizes node interaction patterns from previous batches of training, and continually adjusts parameters for subsequent training.
(3) CL faces the catastrophic forgetting problem in learning, where old knowledge is overwritten by new knowledge. IL also faces this problem, where parameters are biased towards new batches of data during training.
(4) The core problem with CL is that tasks change before and after. IL built on the temporal network had to face similar changes, i.e., the evolution of the network over time.

Due to the similarity, we introduce the idea from continual learning [47] used to alleviate catastrophic forgetting problem to enhance inductive learning, thus constraining the stable update of parameters and node embeddings during training.

Specifically, we construct the experience buffer $\mathbb{B}$ to hold experience node embeddings for each batch. Note that although conventional CL methods usually select real **nodes**, in a temporal network, the selected experience nodes may undergo new interactions in subsequent training batch, i.e., their embeddings will update over time. Therefore, we directly select experience **node embeddings**.

Here we introduce the self-attention mechanism [1, 2] to select top-R experience embeddings for each batch. An attention function can be considered as the scaled dot-product calculation consisting of queries, keys, and values [52, 62].

$$\text{Att}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^{\mathrm{T}}}{\sqrt{d}}\right)\boldsymbol{V} \tag{20}$$

Where $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{|B| \times d}$ denotes the "queries", "keys", and "values", respectively. Here d is the size of embedding dimension, and $|B|$ is the size of a batch B. The node embeddings in a batch B is denoted as $Z \in \mathbb{R}^{|B| \times d}$, thus we can use three parameter matrices $W^{\boldsymbol{Q}}, W^{\boldsymbol{K}}, W^{\boldsymbol{V}} \in \mathbb{R}^{d \times d}$ to generate $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ respectively. Specially, $\boldsymbol{Q} = ZW^{\boldsymbol{Q}}$, $\boldsymbol{K} = ZW^{\boldsymbol{K}}$, and $\boldsymbol{V} = ZW^{\boldsymbol{V}}$.

Our goal is to find the top-R most representative embeddings from Z as experience embedding set $\mathcal{E}$. Let $\boldsymbol{q}_i$, $\boldsymbol{k}_i$, $\boldsymbol{v}_i$ be the i^{th} row in $\boldsymbol{Q}$, $\boldsymbol{K}$, $\boldsymbol{V}$ respectively. Inspired by [49, 62], we can define the attention function as a probability-based kernel function.

$$\text{Att}(\boldsymbol{q}_i, \boldsymbol{K}, \boldsymbol{V}) = \sum_j \frac{k(\boldsymbol{q}_i, \boldsymbol{k}_j)}{\sum_l k(\boldsymbol{q}_i, \boldsymbol{k}_l)} \boldsymbol{v}_j = \mathbb{E}_{p(\boldsymbol{k}_j|\boldsymbol{q}_i)}[\boldsymbol{v}_j] \tag{21}$$

Where the kernel function $k(\boldsymbol{q}_i, \boldsymbol{k}_j)$ denotes the asymmetric exponential kernel $\exp(\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}} / \sqrt{d})$, and the probability $p(\boldsymbol{k}_j|\boldsymbol{q}_i)$ is equal to $k(\boldsymbol{q}_i, \boldsymbol{k}_j) / \sum_l k(\boldsymbol{q}_i, \boldsymbol{k}_l)$. In this way, the attention function encourages the query's attention probability distribution corresponding to dominant dot-product pairs away from the uniform distribution. Let $q(\boldsymbol{k}_j|\boldsymbol{q}_i) = 1/|B|$ denotes the uniform distribution, we can calculate the difference between $p(\boldsymbol{k}_j|\boldsymbol{q}_i)$ and $q(\boldsymbol{k}_j|\boldsymbol{q}_i)$ to distinguish the representative embeddings. Specially, we use the Kullback-Leibler divergence [21, 62] to measure the representativeness as follows.

$$\text{KL}(q||p) = \ln \sum_{j=1}^{|B|} e^{\frac{\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}}}{\sqrt{d}}} - \frac{1}{|B|} \sum_{j=1}^{|B|} \frac{\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}}}{\sqrt{d}} - \ln |B| \tag{22}$$

When we calculate and compare the Kullback-Leibler divergence for each query, the constant $\ln |B|$ can be omitted. Thus the i^{th} query's representativeness measurement (RM value) can be defined as follows.

$$\mathrm{RM}(\boldsymbol{q}_i, \boldsymbol{K}) = \ln \sum_{j=1}^{|B|} e^{\frac{\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}}}{\sqrt{d}}} - \frac{1}{|B|} \sum_{j=1}^{|B|} \frac{\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}}}{\sqrt{d}} \tag{23}$$

The larger the RM value is, the more representative the query is. Note that we need to calculate all dot-product pairs to obtain the RM value for each query, which means large and complex calculations. To simplify this process, we use an empirical approximation for the calculation. According to [62], for each query $\boldsymbol{q}_i \in \mathbb{R}^d$, we have the bound as follows.

$$\ln |B| \leqslant \mathrm{RM}(\boldsymbol{q}_i, \boldsymbol{K}) \leqslant \max_j \{\frac{\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}}}{\sqrt{d}}\} - \frac{1}{|B|} \sum_{j=1}^{|B|} \frac{\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}}}{\sqrt{d}} + \ln |B| \tag{24}$$

In this way, we only need to calculate the max value of $(\boldsymbol{q}_i \boldsymbol{k}_j^{\mathrm{T}} / \sqrt{d})$ to obtain the representativeness of each query. Then we select the most representative top-R node embeddings as experience embedding set $\mathcal{E}$ and put them into the experience buffer $\mathbb{B}$, i.e., $\mathbb{B} = \mathbb{B} \cup \mathcal{E}$. Embeddings in the experience buffer will be used in the optimization process, thus forcing the average node embeddings generated in subsequent batches to be as similar as possible to the experience embeddings. Specially, we randomly choose R embeddings from the buffer $\mathbb{B}$ into the optimization process, and the selected embeddings may come from different batches.

By adding a new constraint in the original loss function, we attempt to consolidate the old experience as we learns new experience. Assuming that the original loss function is L, the new loss function can be formulated as follows.

$$L := L + \beta L(\mathcal{E}) \tag{25}$$

$$L(\mathcal{E}) = \log \sigma(-\left\| \frac{1}{|B|} \sum_{u \in B} z_u^{t_n} - \frac{1}{R} \sum_{i \in \mathcal{E}} \varepsilon_i \right\|^2) \tag{26}$$

Here $|B|$ is the size of a batch B, R is the number of the selected experience embeddings, β is the dynamic weight which will change with the changes of $|B|$ and R. Since we calculate the final loss for each node while $L(\mathcal{E})$ is for each batch, we want to scale it by a certain percentage (i.e., β) to each node loss.

$$\beta = R/(|B| + R) \tag{27}$$

To the best of our knowledge, we are the first to compare continual learning with inductive representation learning. This work may provide an initial attempt to exploit continual learning for inductive representation learning and open up new research possibilities. Moreover, we also verify the effectiveness of continual learning for network embedding in the experiments.

3.7 Model Optimization

To learn node embeddings in a fully unsupervised setting, we apply a network-based loss function, and optimize it with the Adam method [19].

$$L = \sum_{u \in V} \sum_{v \in H_u} L(u, v) + L(c) + \beta L(\mathcal{E}) \tag{28}$$

This loss function can be divided into three parts, i.e., loss function $L(u, v)$ based on neighbor nodes, loss function $L(c)$ based on community detection, loss function $L(\mathcal{E})$ based on continual learning as shown in Eq. (26).

For the loss function $L(u, v)$ based on neighbor nodes, we define the interaction between node u and v as a positive sample, while all other nodes not in the neighborhood of u are negative samples. Thus the optimization objective is to encourage nearby nodes (positive sample) to have similar embeddings while enforcing that the embeddings of disparate nodes (negative sample) are highly distinct.

However, this will result in a huge amount of computation. Thus we introduce negative sampling, in which only a portion of negative samples are randomly selected for computation. In this way, we construct the loss function $L(u, v)$ and use negative squared Euclidean distance to measure the similarity between two embeddings.

$$L(u, v) = \log \sigma\left(-\left\| z_u^{t_n} - z_v^{t_n} \right\|^2\right) - Q \cdot E_{v_n \sim P_n(v)} \log \sigma\left(-\left\| z_u^{t_n} - z_{v_n}^{t_n} \right\|^2\right) \tag{29}$$

Here $P_n(v)$ is a negative sampling distribution, Q is the number of negative samples.

For the loss function $L(c)$ based on community detection, we encourage each node to have high affinity with the community it belongs to. According to the two patterns of non-overlapping communities and overlapping communities, we select (10) or (13) as $L(c)$, respectively.

$$L(c) = \begin{cases} \sum\limits_{u \in V} \max\limits_{c_k \in C} (\log p(c_k|u)), \text{non-overlapping} \\ \sum\limits_{u \in V} \sum\limits_{c_k \in C} \sum\limits_{v \in H_u} \log p(v|c_k) p(c_k|u), \text{overlapping} \end{cases} \tag{30}$$

When we choose the overlapping pattern, the loss function $L(u, c)$ will face the enormous computation cost. Because in (12), the affinity of each node v' in the network with community c_k needs to be calculated. Thus, we also use negative sampling and the sampled nodes only need to follow (29). This is because we only need to ensure that the negative nodes have not interacted with node u.

3.8 Complexity Analyses

Algorithm 1 ConMNCI procedure

Input: Temporal network $G = (V,\ E,\ T)$.
Output: Node embeddings; Community embeddings.
1: Initialize embeddings for each node based on Eq. (3);
2: Initialize embeddings for each community randomly;
3: Initialize empty experience buffer $\mathbb{B}$;
4: **repeat**
5: **for** each *batch* in network **do**
6: **for** each node u in *batch* **do**
7: Calculate $NE_u^{t_n}$ based on Eq. (7);
8: Calculate $CO_u^{t_n}$ based on Eq. (14);
9: Calculate $z_u^{t_n}$ based on Eq. (19);
10: **end for**
11: Update community embeddings based on Eq. (9);
12: Optimize the loss function based on Eq.(28);
13: Obtain experience embeddings set $\mathcal{E}$;
14: Add $\mathcal{E}$ to experience buffer: $\mathbb{B} = \mathbb{B} \cup \mathcal{E}$;
15: **end for**
16: **until** Convergence

In this part, we analyze the complexity of ConMNCI. The procedure for ConMNCI is shown in Algo. 1.

Suppose that the number of nodes and edges in the graph are $|V|$ and $|E|$, respectively. Let t be the number of epochs, s be the number of batches ($s = |E|/|B|$), $|B|$ be the size of each batch, d be the embedding size, L be the length of the historical neighbor sequence, K be the number of communities, R be the number of experience embeddings in each batch, and Q be the number of negative sample nodes.

According to Algo. 1, we can divide ConMNCI into five parts: Initialization, Batch Training, Updating Community Embedding, Loss Function Optimization, and Selecting Experience Embeddings. We first calculate the time complexity of each part and then accumulate them.

1. Initialization (lines 1-3). In this part, we initialize node embedding, community embedding and experience buffer. For node embedding, according to (1)-(3), there are $|V|$ nodes in the network and we generate a d-dimensional embedding for each node, thus the time complexity is $O(d|V|)$. For community embedding, we randomly generate K community embeddings, which time complexity is $O(Kd)$. For experience buffer, its complexity is a constant. Therefore, the time complexity of this part is $O(d|V| + Kd)$.
2. Batch Training (lines 6-10). In this part, we discuss the complexity of calculating $NE_u^{t_n}$, $CO_u^{t_n}$, and $z_u^{t_n}$ separately. According to (4)-(7), calculating $NE_u^{t_n}$ can be done in time complexity of $O(L(Ld + d)) = O(L^2 d)$. According to (8) and (14), if we select the non-overlapping pattern, calculating $CO_u^{t_n}$ can be done in time complexity of $O(K^2 d)$. According to (15)-(19), calculating $z_u^{t_n}$ can be done in time complexity of $O(3d^2 + d^2) = O(d^2)$. Thus, the time complexity of traing one batch data is $O(|B|(L^2 d + K^2 d + d^2))$.
3. Updating Community Embedding (line 11). In this part, we first assign nodes to communities based on their affinities, and then update the community embeddings based on node embeddings. According to (9) and (10), if we select the non-overlapping pattern, the time complexity of this part is $O(|B|(K + d))$.
4. Loss Function Optimization (line 12). In this part, according to (28), we need to discuss the complexity of three loss functions $L(u, v)$, $L(c)$, and $L(\mathcal{E})$ separately. According to (29), the complexity of calculating $L(u, v)$ is $O(|B|LQd)$, where L is the neighbor sequence length and Q is the number of negative samples. According to (30), the complexity of calculating $L(c)$ is $O(|B|K^2 d)$. According to (26), the complexity of calculating $L(\mathcal{E})$ is $O(|B|d + Rd)$, where R is the number of experience embeddings in each batch. After calculating the above three loss functions, ConMNCI needs to perform backpropagation to optimize the model parameters. The parameters to be optimized are $\{\delta_u^{NE}, \delta_u^{CO}\}$, $\{W_{UG}, W_{NG}, W_{CG}, W_z\}$, $\{b_{UG}, b_{NG}, b_{CG}, b_z\}$, and $\{\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}\}$, and the time complexity of optimizing

Table 2 Description of the datasets

Datasets	DBLP	BITotc	BITalpha	ML1M	AMms	Yelp
Nodes	28,085	5,881	3,783	9,746	74,526	424,450
Edges	236,894	35,592	24,186	1,100,209	89,689	2,610,143
Labels	10	7	7	5	5	5
Timestamps	25	22,115	981	10,850	5,082	70

these parameters is $O(d + 3d^2 + |B|d^2) = O(|B|d^2)$. Thus, the time complexity of this part is $O(|B|LQd + |B|K^2d + Rd + |B|d^2)$.

5. Selecting Experience Embeddings (lines 13 and 14). In this part, we need to discuss the complexity of self-attention value, RM value, and top-R selection separately. According to (20) and (21), the time complexity of calculating self-attention value is $U(|B|d^2 + |B|^2 + |B|^2d)$. According to (22)-(24), the time complexity of calculating RM value is $O(|B|d)$. The time complexity of conducting top-R selection is $O(|B|d)$. Thus, the time complexity of this part is $O(|B|d^2 + |B|^2 + |B|^2d + |B|d + Rd) = O(|B|d^2 + |B|^2d + Rd)$.

In summary, considering the number of epochs t and the number of batches s, the total time complexity of ConMNCI can be calculated as follows.

$$\begin{aligned} &O((d|V| + Kd) + ts(|B|(L^2d + K^2d + d^2) + |B|(K + d)+ \\ &(|B|LQd + |B|K^2d + Rd + |B|d^2) + (|B|d^2 + |B|^2d + Rd))) \\ &= O(d|V| + ts(|B|(L^2d + K^2d + d^2) + |B|LQd + Rd + |B|^2d)) \end{aligned} \tag{31}$$

Considering that L, K, R, Q are small constants, the time complexity of ConMNCI can be simplified as $O(d|V| + ts(|B|d^2 + |B|^2d))$.

4 Experiments

We compare ConMNCI with six state-of-the-art baselines and conduct experiments on six real-world datasets.

4.1 Datasets

We list the statistical information of the following six real-world network datasets in Table 2.

DBLP [63] is a co-authorship dataset of Computer Science domain taken from DBLP which has ten research fields. If more than half of a researcher's last ten papers are published in a particular research field, we assume that this researcher belongs to this field.

BIT otc/alpha [22, 23] are two datasets taken from two bitcoin trading platforms OTC and Alpha, respectively. A member will rate other members in a scale of -10 (total distrust) to +10 (total trust) in steps of 1 after the transaction. We assume every three score steps into one category, thus there are seven categories, e.g., users with scores -10, -9 and -8 are in the same category.

ML1M [25] is a widely used movie dataset (version MovieLens-1M). For each movie, we choose the score that people rated most as its label. Since the score is an integer between 1 and 5, we divide all movies into five categories.

AMms [35] is a magazine subscription dataset from the Amazon website. For each magazine, we choose the score people rated most as its label and also divide all magazines into five categories.

Yelp [63] is a challenge dataset from the Yelp website. In this real-world network, users and businesses are defined as nodes, and commenting behaviors are taken as edges. Each business is assigned either one or more categories. We only retain business in the top-5 categories. If a business has more than one category, we assign the top one category as the business's label.

4.2 Baselines

We compare ConMNCI with six state-of-the-art baselines. Each of these methods represents a category of related work.

DeepWalk [40] first applies random walks to generate sequences of nodes over the network and then employs the Skip-Gram [33] model to learn node embeddings, which is a ***classic*** method in the field of NRL.

node2vec [13] uses the random walk procedure to balance the breadth-first and depth-first search strategy, which is a ***static transductive*** method.

GraphSAGE [14] learns a function to generate node embeddings by sampling and aggregating features from nodes' local neighborhood, which is a ***static inductive*** method.

HTNE [63] uses the Hawkes process to capture influence of historical neighbors on the current node, which is a ***dynamic temporal transductive*** method.

DyREP [48] uses RNNs to learn node embeddings while it loss function is built upon temporal point process, which is a ***dynamic temporal inductive*** method.

EvolveGCN [38] uses a RNN to estimate the GCN parameters for the future snapshots, which is a ***static snapshot transductive*** method.

4.3 Tasks and Evaluation Measures

First, we compare ConMNCI with baselines on three fundamental tasks: node classification, network visualization, and link prediction. Note that Network Embedding for node classification can be considered as supervised learning, because node classifiers are trained based on node labels. Network Embedding for network visualization and link prediction can be considered as unsupervised learning, because node labels are not involved in these tasks.

Node Classification: We train a classifier to predict node labels using node embeddings. In this task, we use both Accuracy and Weighted-F1 as metrics.

Network Visualization: We select some nodes with different labels and project them onto a 2-dimensional space, then observe the selected nodes' ditributions in the 2-dimensional space.

Link Prediction: Based on two node embeddings, we can calculate their dot product to determine whether there is an edge between these two nodes. We use both the AUC score and Accuracy as metrics.

Then we perform ablation study and parameter sensitivity study to further evaluate ConMNCI.

Ablation Study: We evaluate the performance improvements of positional encoding, neighborhood and community influences, and continual learning for ConMNCI.

Parameter Sensitivity Study: We evaluate the effect of several hyperparameters on the performance, such as the embedding dimension size d, the length of neighbor sequence l, and the number of communities K, etc.

4.4 Parameter Settings

For all methods, we set the embedding dimension size d, the learning rate, the batch size b, the number of negative samples Q, the length of neighbor sequence L, and the number of communities K to be 128, 0.001, 128, 10, 5, and 10, respectively. We use default values for other parameters in baselines.

In our datasets, the data is arranged chronologically in (u, v, t) format. For each dataset, we sort all interactions and split the total interaction time range $[t_0, t_n]$ into two intervals: $[t_0, t_{train})$, $[t_{train}, t_n]$. The interactions in these two time intervals are used for training and testing, respectively. Note that We fix $t_{train}/t_n = 80\%$, i.e., we select the top 80% of each dataset as the training set, and the rest 20% as the test set. If the same timestamp interactions are assigned to both training set and test set, we assign all interactions at this timestamp to the training set. This is because in our experiments, we use interactions that occurred in the past to predict possible future interactions. Therefore, the training and test sets should be divided strictly in chronological order.

4.5 Node Classification

Here we train a Logistic Regression function as the classifier to perform 5-fold cross-validation to predict node labels. Then we evaluate the classification results on all datasets by Accuracy and Weighted-F1.

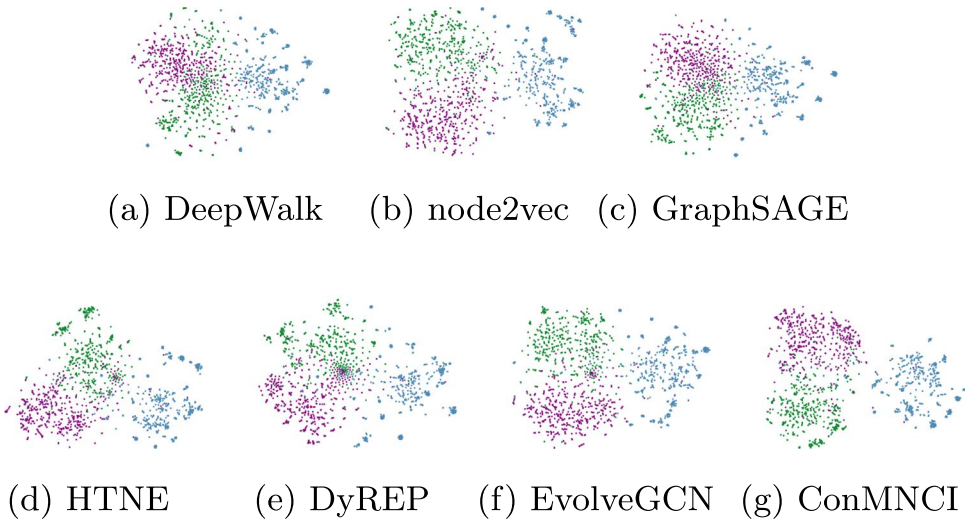
From Table 3, it is observed that ConMNCI achieves the best performance. In addition, HTNE, DyREP and EvolveGCN perform better than Deepwalk, node2vec and GraphSAGE in most cases, which demonstrates that the acquisition of dynamic information is critical for learning effective network representations. Compared with GraphSAGE and HTNE that use neighborhood interactions, ConMNCI focuses on both neighborhood and community influences, leading to further performance improvements.

Note that all methods' results are close on AMms. This is because that the average degree of each node is 2.40 in AMms, i.e., most nodes may interact with only one neighbor

Table 3 Node classification results of all methods on all datasets

Metric(%)	method	DBLP	BITotc	BITalpha	ML1M	AMms	Yelp
Accuracy	DeepWalk	61.40±0.55	59.07±1.33	72.94±2.87	60.29±0.25	57.80±0.33	50.67±0.89
	node2vec	62.49±1.16	59.58±0.44	74.95±0.28	61.96±0.62	57.72±0.02	51.35±0.43
	GraphSAGE	63.31±0.53	60.03±0.68	73.89±0.35	61.24±0.59	57.63±0.16	51.84±1.16
	HTNE	63.47±0.38	59.99±0.67	76.35±0.85	58.90±1.48	57.67±0.01	52.73±0.16
	DyREP	62.59±2.42	61.00±0.69	74.30±1.41	60.23±0.88	57.55±0.34	52.09±0.27
	EvolveGCN	62.64±1.87	59.28±0.62	78.58±0.13	56.64±0.75	58.48±0.01	50.93±1.79
	ConMNCI	64.65±0.37	62.94±0.42	79.43±0.11	62.89±0.24	58.87±0.07	54.17±0.15
Weighted-F1	DeepWalk	61.07±2.78	51.20±0.67	67.61±4.41	58.63±1.88	42.52±0.29	39.81±1.15
	node2vec	62.10±0.54	51.23±0.59	68.32±2.84	58.36±0.83	42.48±0.85	41.84±1.82
	GraphSAGE	62.39±0.55	51.05±0.69	67.50±0.41	57.66±1.02	42.16±1.25	40.65±1.92
	HTNE	63.07±0.54	51.09±1.17	68.06±0.88	54.15±0.22	42.55±0.15	41.80±1.29
	DyREP	62.03±1.23	51.14±1.13	68.43±0.49	57.29±0.71	42.48±0.31	40.93±1.79
	EvolveGCN	61.98±2.34	51.79±0.82	67.83±2.21	59.65±0.24	41.53±0.85	39.94±2.23
	ConMNCI	64.43±0.82	51.72±1.23	68.64±0.57	60.85±0.77	42.68±0.94	43.07±1.02

Fig. 3 Network visualization



in AMms. The topology of AMms looks like a longer chain, which leads to poor performance of all methods on AMms.

4.6 Network Visualization

We employ the t-SNE method [28] to project node embeddings on DBLP to a 2-dimensional space. In particular, we select three fields and 500 researchers in each field. Selected researchers are shown in a scatter plot, in which different fields are marked with different colors, i.e., green for data mining, purple for computer vision, blue for computer network.

As shown in Fig. 3, both DeepWalk, node2vec, and GraphSAGE failed to separate the three fields clearly. HTNE, DyREP and EvolveGCN can only roughly distinguish the field boundaries. ConMNCI separates the three fields clearly, and one of them has a clear border, which indicates that ConMNCI has better performance.

4.7 Link Prediction

For all datasets, we use both Area Under the ROC Curve (AUC) [15] and Accuracy as metrics to compare ConMNCI with baselines for link prediction.

In the training set, we first generate node embeddings by applying ConMNCI and baselines. In the test set, we sample a certain number of node pairs connected by interactions as positive samples and sample the same number of node pairs without interactions as negative samples. Then we calculate the dot product of their embeddings for each pair of nodes and use the sigmoid function to normalize the dot product as the interaction probability.

Table 4 Link prediction results of all methods on all datasets

Metric(%)	method	DBLP	BITotc	BITalpha	ML1M	AMms	Yelp
AUC	DeepWalk	82.53±1.53	51.99±1.44	55.58±1.35	46.35±1.24	54.83±2.11	77.40±0.82
	node2vec	81.73±2.76	57.99±1.42	62.45±1.03	50.12±2.56	52.28±3.93	84.26±2.07
	GraphSAGE	84.52±1.48	59.67±1.62	69.64±2.56	50.55±3.77	55.54±0.94	85.53±0.83
	HTNE	88.68±0.99	71.45±1.83	74.01±1.77	50.21±0.96	57.41±2.54	88.21±0.87
	DyREP	87.63±1.52	71.12±2.12	73.42±0.89	50.69±1.43	58.07±2.42	86.64±3.46
	EvolveGCN	85.54±2.34	71.79±0.11	71.64±1.32	53.87±1.23	51.43±0.75	79.53±4.82
	ConMNCI	89.66±1.03	74.77±0.89	74.32±0.76	54.13±1.31	59.34±0.57	87.29±1.24
Accuracy	DeepWalk	52.25±0.71	53.90±0.66	53.99±1.25	50.04±0.89	50.67±1.33	51.74±0.76
	node2vec	50.09±0.89	50.17±1.12	50.31±0.57	50.08±0.08	50.00±0.07	50.02±0.13
	GraphSAGE	66.62±0.78	55.39±0.64	55.50±0.35	50.49±1.23	53.32±0.82	50.23±1.17
	HTNE	73.57±0.64	59.12±0.88	62.38±1.62	46.39±0.58	56.87±2.03	52.90±0.96
	DyREP	72.03±0.72	60.49±1.63	64.33±0.79	50.23±2.13	54.54±1.22	51.48±0.97
	EvolveGCN	71.48±1.23	61.79±0.64	68.83±0.21	49.53±0.85	53.52±1.43	49.46±3.11
	ConMNCI	78.04±0.53	69.72±0.62	69.36±0.64	50.58±1.42	57.00±0.13	52.99±1.03

For AUC, we sort all interaction probabilities in descending order and assume that there are edges between each node pair of the top-half. By comparing the truth on node pairs, we can obtain the AUC score. For Accuracy, we set a threshold value of 0.5 to evaluate the prediction result. When the probability (normalize dot product) of a node pair is greater than 0.5, we consider that there exists an edge between this node pair.

As shown in Table 4, it can be seen that ConMNCI has the best performance on all datasets, which demonstrates the ability of ConMNCI to capture interactive information. We also find that all methods obtain poor performance on dataset ML1M and AMms. According to the Table 2, we can find that the average degree of each node is 2.40 and 225.77, respectively. Compare with the other datasets whose average degrees are all in the interval (12,17), the network structure of datasets ML1M and AMms are very special. We assume that for such datasets with too large or too small average degrees, researchers need to design targeted methods. This may be a new research direction for further grounding of network representation learning in industry.

In addition, through the experimental results we find that methods such as HTNE which exploits the interaction time are more effective than methods such as Deepwalk which only focuses on the network structure. Combined with the above experiments, we believe that temporal information is very useful for capturing the network evolution process and should be paid more attention and further utilized.

4.8 Ablation Study

Here, we construct several variants of ConMNCI to study the role of positional encoding, community and neighborhood influences, and continual learning, respectively.

4.8.1 Positional Encoding

Different from the existing NRL methods, we use positional encoding instead of random initialization to generate the initial embedding. To compare their differences, we construct two variants based on positional encoding (**PE**) and random initialization (**RI**) to initialize node embeddings.

Although there is no significant difference between the final performance of PE and RI through experiments, PE is much faster than RI in raising the loss function's convergence speed. Since one epoch represents one complete training on the whole dataset, we use ***epoch number*** as a metric to measure the convergence speed.

On DBLP, when the performance is almost the same, RI requires ***30 epochs*** to converge, while PE only needs ***10 epochs***. The convergence speed of the latter is 3 times as fast as the former. On BITotc and BITalpha, the convergence speeds of RI and PE are ***20*** and ***5 epochs*** respectively. The convergence speed of the latter is 4 times as fast as the former. The results demonstrate that positional encoding can accelerate the convergence speed of ConMNCI without reducing performance, which is especially applicable for large-scale datasets.

4.8.2 Neighborhood and Community Influences

In GRU, we aggregate three types of information: node embedding, neighborhood influence and community influence. Here, we evaluate the improvement brought by neighborhood and community, respectively.

Let ConMNCI.z be a variant of ConMNCI which only aggregate node embedding, i.e., $\tilde{z}_u^{t_n} = \tanh[W_z z_u^{t_{n-1}} + b_z]$. ConMNCI.zn denotes a variant that only aggregate neighborhood influence and node embedding, i.e., $\tilde{z}_u^{t_n} = \tanh[W_z(z_u^{t_{n-1}} + NG_u^{t_n} \cdot NE_u^{t_n}) + b_z]$.

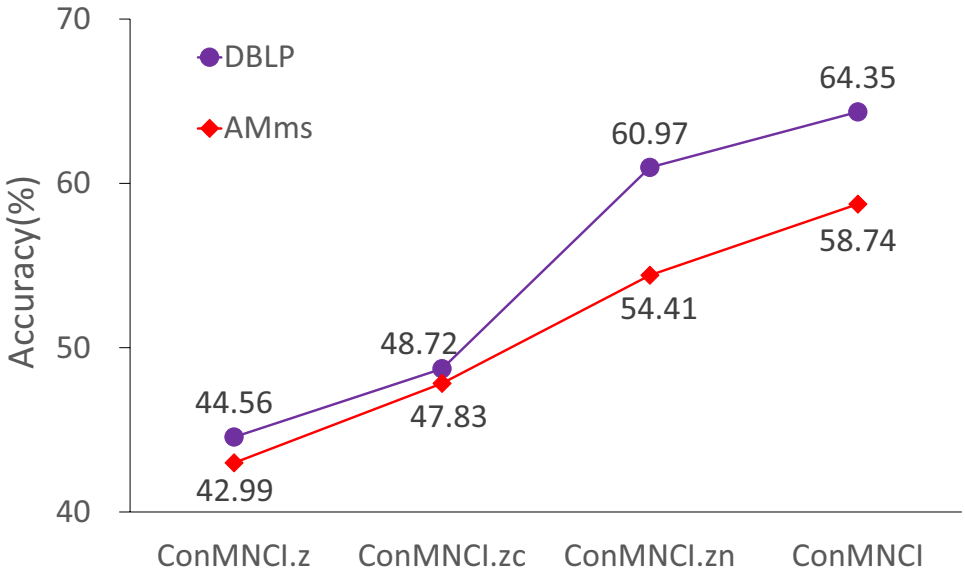
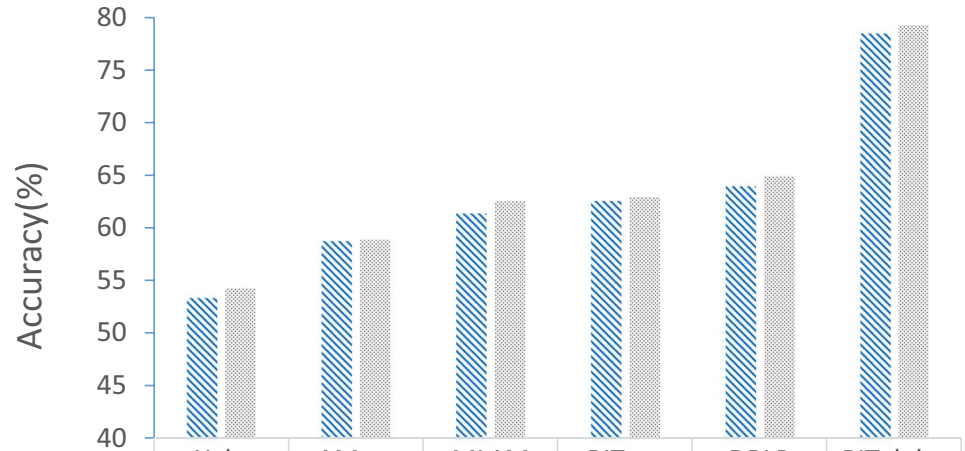


Fig. 4 Ablation study of community and neighborhood influences



	Yelp	AMms	ML1M	BITotc	DBLP	BITalpha
MNCI	53.34	58.74	61.37	62.56	63.95	78.42
ConMNCI	54.29	58.87	62.55	62.94	64.91	79.23

Fig. 5 Ablation study of continual learning

ConMNCI.zc denotes a variant that only aggregate community influence and node embedding, i.e., $\tilde{z}_u^{t_n} = \tanh[W_z(z_u^{t_{n-1}} + CG_u^{t_n} \cdot CO_u^{t_n}) + b_z]$.

We evaluate these variants of ConMNCI via node classification task. As shown in Fig. 4, when neighborhood and community influences are not used, the performance is the worst. The performance is improved when we use community influence or neighborhood influence. Note that ConMNCI.zn is better than ConMNCI.zc, which means that both of two influences are effective, and neighborhood influence is more important than community influence.

Comparing the two datasets' performance, the performance improvement of neighborhood influence on DBLP is greater than that on AMms. Since in Table 2, the average degree of each node in DBLP and AMms is 16.87 and 1.20, respectively. Therefore, the nodes in DBLP are more sensitive to neighborhood influence than nodes in AMms, because nodes in DBLP have more interactions with neighbors. Therefore, when we consider neighborhood influence, the performance improvement of ConMNCI on DBLP is greater than that on AMms.

4.8.3 Continual Learning

In this paper, we introduce the concept of continual learning, which constrain the method to keep sensitive to the old knowledge when training new data. To demonstrate the validity of continual learning, we define the method without continual learning is MNCI, and compare with ConMNCI via node classification task on all datasets.

Table 5 Parameter sensitivity of dimension size d

Accuracy(%)						
dataset	method	d=32	d=64	d=128	d=256	d=512
DBLP	DeepWalk	59.90	60.57	61.40	61.29	60.51
	node2vec	62.12	62.46	62.49	62.07	62.49
	GraphSAGE	62.39	63.03	63.31	63.40	63.05
	HTNE	61.36	62.55	63.47	63.28	63.37
	DyREP	62.09	62.33	62.59	62.03	61.87
	EvolveGCN	62.02	62.57	62.64	62.55	62.43
	ConMNCI	63.92	63.99	64.35	64.07	63.98
AMms	DeepWalk	56.60	57.57	57.80	57.11	56.97
	node2vec	56.12	56.46	57.72	57.07	57.49
	GraphSAGE	56.85	56.87	57.63	57.59	57.03
	HTNE	57.36	57.43	57.67	57.28	57.68
	DyREP	56.80	57.03	57.55	57.44	57.31
	EvolveGCN	58.31	58.02	58.48	57.69	58.17
	ConMNCI	58.02	58.11	58.74	58.09	58.07
BITotc	DeepWalk	59.02	58.97	59.07	59.00	58.94
	node2vec	59.94	60.00	60.01	59.97	59.65
	GraphSAGE	60.02	60.17	60.21	60.14	60.17
	HTNE	61.12	61.39	61.54	61.08	61.48
	DyREP	61.07	61.22	61.23	61.02	61.13
	EvolveGCN	59.02	59.02	59.27	59.19	59.17
	ConMNCI	62.53	62.87	62.94	62.91	62.76

As shown in Fig. 5, even the performance improvement varies across datasets, ConMNCI still outperforms MNCI on all datasets. By comparing the improvements on different datasets, we find that continual learning seems to work better on a large scale datasets. The results demonstrate the effectiveness of continual learning, which is worthy of further research.

4.9 Parameter Sensitivity Study

We evaluate the effect of the embedding dimension size d, the length of neighbor sequence L, and the community number K on the performance of ConMNCI, respectively.

4.9.1 Embedding dimension size

To evaluate the effect of embedding dimension size d on the performance of NRL methods, we conduct node classification experiment on three datasets: DBLP, AMms, and BITotc. Specially, we fix the other parameter settings and vary d from 32 to 512 to test the performance on ConMNCI and baselines. To evaluate these methods via node classification task, we use Accuracy as a metric.

As shown in Table 5, we find that the embedding size has little effect on the performance of ConMNCI and it performs the best overall. Note that GraphSAGE does not achieve the best performance on DBLP dataset when d is 128, and similarly HTNE on AMms. But we can also find that the performance (63.31%) of GraphSAGE with d=128 are very close to the best performance (63.40%), and so is HTNE (57.67% vs. 57.68%). In addition, all other methods achieve the best performance when d is 128. By comparing the experimental results on the three datasets, we speculate that due to random initialization in training process, HTNE and GraphSAGE do not achieve the best performance at d=128.

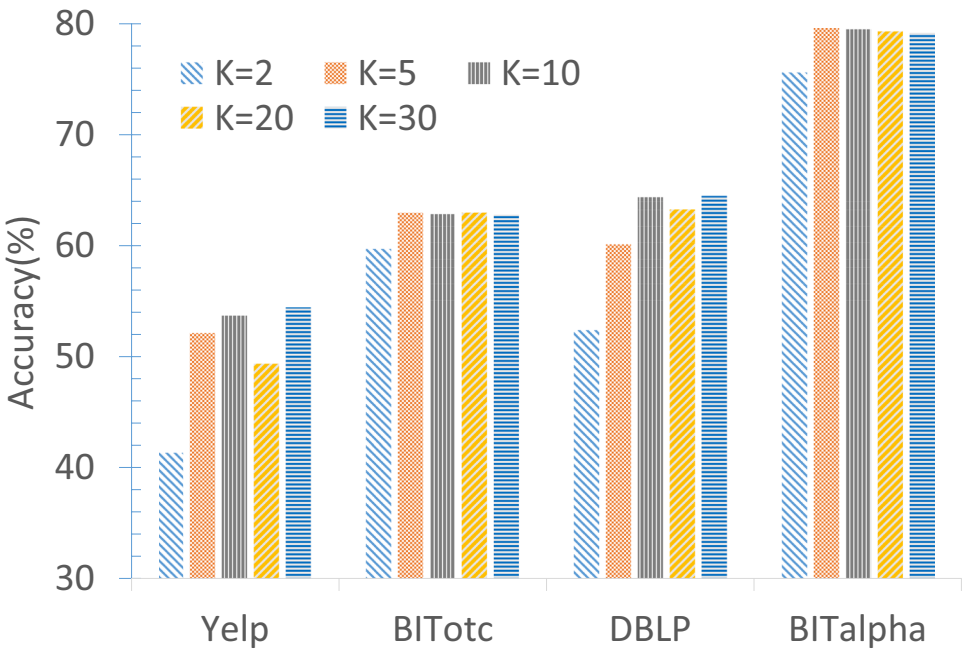
However, another issue has drawn our attention. Why do almost all methods work best when d is 128? In the field of Network Embedding, most of the work [13, 24, 40, 46, 53] sets d=128 by deafult and obtains best performance in experiments. This phenomenon could be explained as follows.

When d is less than 128, with the increase of dimension size, the representation ability of node embedding is enhanced and thus the performance is improved. But when d is greater than 128, the performance begins to decline. We have two speculations about this phenomenon. On the one hand, as the dimension size increases, more noise may be introduced into node embeddings, which interferes with the performance. On the other hand, the embeddings of two similar nodes will show similarity in more dimensions as the dimension size increases, thus causing difficulties in node classification.

Table 6 Parameter sensitivity experiment of sequence length L

Accuracy(%)					
Datasets	L=2	L=3	L=5	L=10	L=20
BITotc	60.82	62.93	63.21	62.57	61.42
BITalpha	73.11	75.63	79.54	78.89	78.03
ML1M	53.62	56.58	60.12	62.23	62.19
AMms	**57.77**	57.68	57.05	56.82	56.53

Fig. 6 Parameter sensitivity of community number K



We, therefore, believe that the embedding dimension size should be balanced between representation ability and experimental performance. Some researchers [6] have started to experiment with dynamically generating embeddings with different dimension sizes for different nodes. However, since the change in dimension size has a little influence on performance in the field of network embedding, this issue has not received much attention from researchers and d is usually set to 128. In the future, we will work on this issue.

In summary, the experimental results prove that the embedding dimension size has little effect on the performance of the method, and the best results tend to arise at d=128. Therefore, we set the embedding dimension size d to be 128 on ConMNCI and use default values of d in baselines. In fact, all baseline method set the default d=128.

4.9.2 Historical Neighbor Sequence Length

When mining neighborhood influence, we use a hyperparameter of the historical neighbor sequence length L, which is designed to truncate a fixed length sequence of node neighbors by the latest interactions.

Reference to previous research [17, 27, 63], neighbors of very early interactions have little effect on the current node interaction. In other words, only the few newly interactive neighbors may play a major role in most cases, while other neighbors may interfere with traninig. Therefore, to verify this intuition, we choose 2, 3, 5, 10, and 20 for the length L, other parameters are the same as above.

Specifically, we evaluate ConMNCI with different L via node classification task. Due to the large difference in node degrees, we select four different datasets, BITotc, BITalpha, ML1M, and AMms.

As shown in Table 6, when L is 5, ConMNCI achieves the best performance on both BITotc and BITalpha. However, ML1M tends to have better performance with more neighbors (L=10), and AMms tends to have better performance with fewer neighbors (L=2). This may be due to different node degree in the dataset. According to data description in Table 2, the average degree of each node is 2.40 in AMms dataset and is 225.77 in ML1M dataset, respectively. The average degree of the other datasets are all in the interval (12,17). It means that most of the nodes in AMms have few edges, while most of the nodes in ML1M have a lot of edges. Thus our method requires a short neighbor sequence length in AMms (L=2) and a long one in ML1M (L=10). In the link prediction experiments (Section 4.7), we also discuss the effect of differnt neighbor sequence length L.

In summary, we should select different neighbor sequence length L for different network data. On the other hand, we find that when L is 5, the results on each dataset are closer to the best results. Therefore, we select L=5 by default for all datasets.

4.9.3 Community Number

When mining community influence, we use a hyperparameter of the community number K. Here we fix the other parameters and choose 2, 5, 10, 20, and 30 for K to observe performance changes. We consider the task of classification and take Accuracy as a metric to evaluate the performance of ConMNCI on four datasets: DBLP, BITotc, BITalpha, and Yelp.

As shown in Fig. 6, on BITotc and BITalpha, ConMNCI achieves the best performance when K is 5, and on DBLP and Yelp, ConMNCI achieves the best performance when K is 10. As shown in Table 2, the numbers of node labels are 5 for BITotc and BITalpha, 7 for Yelp, and 10 for DBLP, respectively. This is generally consistent with the optimal community number K on the different datasets.

Note that the performance on all datasets is very similar to the best performance when K is greater than 10. But when K is large, ConMNCI will divide the whole network into a large number of smaller communities, leading to expensive calculations. Thus we select K=10 for most of the datasets to balance performance and efficiency.

5 Conclusions

We propose an inductive continual network representation learning method ConMNCI that captures both neighborhood and community influences to generate node embeddings at any time. Extensive experiments on several real-world datasets demonstrate that ConMNCI significantly outperforms state-of-the-art baselines. In the future, we will further investigate the influence of node text information on node embeddings.

Acknowledgements This work was supported by the National Natural Science Foundation of China (No. 61972135), the Natural Science Foundation of Heilongjiang Province in China (No. LH2020F043), the Innovation Talents Project of Science and Technology Bureau of Harbin (No. 2017RAQXJ094), and the Foundation of Graduate Innovative Research of Heilongjiang University in China (No.YJSCX2021-076HLJU)

References

1. Bahdanau D, Cho K, Bengio Y (2014) Neural machine translation by jointly learning to align and translate. In: International conference on learning representations
2. Bastings J, Filippova K (2020) The elephant in the interpretability room: Why use attention as explanation when we have saliency methods?. In: Proceedings of the Third BlackboxNLP workshop on analyzing and interpreting neural networks for NLP, pp 149–155
3. Bochner S (1934) A theorem on fourier-stieltjes integrals. Bulletin of The American Mathematical Society, pp 271–277
4. Bruna J, Zaremba W, Szlam A, LeCun Y (2014) Spectral networks and locally connected networks on graphs. ICLR
5. Cao S, Lu W, Xu Q (2015) Grarep: Learning graph representations with global structural information. CIKM
6. Cavallari S, Zheng WV, Cai H, Chang CCK, Cambria E (2017) Learning community embedding with community detection and node embedding on graphs. CIKM, pp 377–386
7. Cho K, Merrienboer vB, Gulcehre C, Bahdanau D, Bougares F, Schwenk H, Bengio Y (2014) Learning phrase representations using rnn encoder-decoder for statistical machine translation. EMNLP pp 1724–1734
8. Cui P, Wang X, Pei J, Zhu W (2019) A survey on network embedding. IEEE Transactions on Knowledge and Data Engineering
9. Ditzler G, Roveri M, Alippi C, Polikar R (2015) Learning in non-stationary environments: A survey. IEEE Computational Intelligence Magazine pp 12–25
10. Erdos P (1961) Graph theory and probability. Canadian Journal of Mathematics
11. Fanzhen L, Shan X, Jia W, Chuan Z, Wenbin H, Cecile P, Surya N, Jian Y, S PY (2020) Deep learning for community detection: Progress, challenges and opportunities. IJCAI pp 4981–4987
12. Grossberg S (1980) How does a brain build a cognitive code? Psychological review pp 1–51
13. Grover A, Leskovec J (2016) node2vec: Scalable feature learning for networks. KDD pp 855–864
14. Hamilton, LW, Ying R, Leskovec J (2017) Inductive representation learning on large graphs. NIPS pp 1024–1034
15. Hanley AJ, McNeil JB (1982) The meaning and use of the area under a receiver operating characteristic (roc) curve. Radiology pp 29–36
16. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neural Computation pp 1735–1780
17. Hu L, Li C, Shi C, Yang C, Shao C (2020) Graph neural news recommendation with long-term and short-term interest modeling. Information Processing and Management
18. Kim D, Oh A (2020) How to find your friendly neighborhood: Graph attention design with self-supervision. ICLR
19. Kingma PD, Ba LJ (2015) Adam: A method for stochastic optimization. ICLR
20. Kirkpatrick J, Pascanu R, Rabinowitz CN, Veness J, Desjardins G, Rusu AA, Milan K, Quan J, Ramalho T, Grabska-Barwinska A, Hassabis D, Clopath C, Kumaran D, Hadsell R (2017) Overcoming catastrophic forgetting in neural networks. In: Proceedings of the national academy of sciences of the United States of America
21. Kullback S, Leibler AR (1951) On information and sufficiency. The Annals of Mathematical Statistics pp 79–86
22. Kumar S, Hooi B, Makhija D, Kumar M, Faloutsos C, Subrahmanian V (2018) Rev2: Fraudulent user prediction in rating platforms. In: WSDM, ACM, pp 333–341
23. Kumar S, Spezzano F, Subrahmanian V, Faloutsos C (2016) Edge weight prediction in weighted signed networks. In: ICDM, IEEE, pp 221–230
24. Kumar S, Zhang X, Leskovec J (2018) Learning dynamic embeddings from temporal interactions. arXiv: Machine Learning
25. Li J, Wang Y, McAuley JJ (2020) Time interval aware self-attention for sequential recommendation. WSDM pp 322–330
26. Liu M, Liu Y (2021) Inductive representation learning in temporal networks via mining neighborhood and community influences.

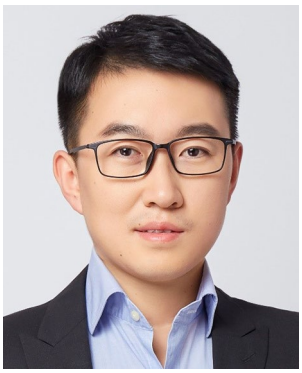
In: SIGIR 2021: 44th international ACM SIGIR conference on research and development in information retrieval
27. Liu M, Quan Z, Liu Y (2020) Network representation learning algorithm based on neighborhood influence sequence. ACML pp 609–624
28. Maaten vdL, Hinton G (2008) Visualizing data using t-sne. Journal of Machine Learning Research
29. McClelland LJ, McNaughton LB, O'Reilly CR (1995) Why there are complementary learning systems in the hippocampus and neocortex: insights from the successes and failures of connectionist models of learning and memory. Psychological review pp 419–457
30. McCloskey M, Cohen JN (1989) Catastrophic interference in connectionist networks: The sequential learning problem. Psychology of Learning and Motivation pp 109–165
31. Mehran SK, Rishab G, Sepehr E, Janahan R, Jaspreet S, Sanjay T, Stella W, Cathal S, Pascal P, Marcus B (2019) Time2vec: Learning a vector representation of time. arXiv: Social and Information Networks
32. Mermillod M, Bugaiska A, Bonin P (2013) The stability-plasticity dilemma: investigating the continuum from catastrophic forgetting to age-limited learning effects. FRONTIERS IN PSYCHOLOGY pp 504–504
33. Mikolov T, Chen K, Corrado G, Dean J (2013) Efficient estimation of word representations in vector space. CoRR
34. Nguyen HG, Lee BJ, Rossi AR, Ahmed KN, Koh E, Kim S (2018) Continuous-time dynamic network embeddings. WWW pp 969–976
35. Ni J, Li J, McAuley J (2019) Justifying recommendations using distantly-labeled reviews and fined-grained aspects. EMNLP/IJCNLP pp 188–197
36. Niepert M, Ahmed MH, Kutzkov K (2016) Learning convolutional neural networks for graphs. In: ICLR, pp 2014–2023
37. Ou M, Cui P, Pei J, Zhu, W (2016) Asymmetric transitivity preserving graph embedding. KDD
38. Pareja A, Domeniconi G, Chen J, Ma T, Suzumura T, Kanezashi H, Kaler T, Schardl TB, Leiserson CE (2020) EvolveGCN: Evolving graph convolutional networks for dynamic graphs. In: Proceedings of the Thirty-Fourth AAAI conference on artificial intelligence
39. Parisi IG, Kemker R, Part LJ, Kanan C, Wermter S (2019) Continual lifelong learning with neural networks: A review. Neural Networks pp 54–71
40. Perozzi B, Al-Rfou' R, Skiena S (2014) Deepwalk: online learning of social representations. KDD pp 701–710
41. Qi C, Zhang J, Jia H, Mao Q, Wang L, Song H (2021) Deep face clustering using residual graph convolutional network. Knowledge Based Systems 211:106561
42. Grossberg S (2013) Adaptive resonance theory: how a brain learns to consciously attend, learn, and recognize a changing world. Neural Networks pp 1–47
43. Sankar A, Wu Y, Gou L, Zhang W, Yang H (2020) Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In: WSDM, pp 519–527
44. Srinivasan B, Ribeiro B (2020) On the equivalence between node embeddings and structural graph representations. ICLR
45. Sun FY, Qu M, Hoffmann J, Huang CW, Tang J (2019) vgraph: A generative model for joint community detection and node representation learning. NIPS pp 512–522
46. Tang J, Qu M, Wang M, Zhang M, Yan J, Mei Q (2015) Line: Large-scale information network embedding. WWW
47. Thrun BS, Mitchell MT (1993) Lifelong robot learning. Lifelong Robot Learning
48. Trivedi R, Farajtabar M, Biswal P, Zha H (2019) Dyrep - learning representations over dynamic graphs. ICLR
49. Tsai HYH, Bai S, Yamada M, Morency LP, Salakhutdinov R (2019) Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. EMNLP/IJCNLP 1:4343–4352
50. Tu C, Liu H, Liu Z, Sun M (2017) Cane: Context-aware network embedding for relation modeling. In: ACL, pp 1722–1731
51. Tu C, Zeng X, Wang H, Zhang Z, Liu Z, Sun M, Zhang B, Lin L (2018) A unified framework for community detection and network representation learning. IEEE Transactions on Knowledge and Data Engineering pp 1–1
52. Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez NA, Kaiser L, Polosukhin I (2017) Attention is all you need. NIPS pp 5998–6008
53. WANG D, Cui P, Zhu W (2016) Structural deep network embedding. KDD
54. Wang Y, Chang YY, Liu Y, Leskovec J, Li P (2021) Inductive representation learning in temporal networks via causal anonymous walks. ICLR
55. Wu J, Wang X, Feng F, He X, Chen L, Lian J, Xie X (2021) Self-supervised graph learning for recommendation. SIGIR pp 726–735
56. Xu D, Cheng W, Luo D, Liu X, Zhang X (2019) Spatio-temporal attentive rnn for node classification in temporal attributed graphs. IJCAI pp 3947–3953
57. Xu D, Liang J, Cheng W, Wei H, Chen H, Zhang X (2021) Transformer-style relational reasoning with dynamic memory updating for temporal network modeling. AAAI pp 4546–4554
58. Xu D, Ruan C, Korpeoglu E, Kumar S, Achan K (2019) Self-attention with functional time representation learning. NIPS pp 15889–15899
59. Xu D, Ruan C, Korpeoglu E, Kumar S, Achan K (2020) Inductive representation learning on temporal graphs. ICLR
60. Yang M, Zhou M, Kalander M, Huang Z, King I (2021) Discrete-time temporal network embedding via implicit hierarchical learning in hyperbolic space. KDD pp 1975–1985
61. Zhou F, Cao C (2021) Overcoming catastrophic forgetting in graph neural networks with experience replay. AAAI
62. Zhou H, Zhang S, Peng J, Zhang S, Li J, Xiong H, Zhang W (2021) Informer: Beyond efficient transformer for long sequence time-series forecasting. In: The Thirty-Fifth AAAI conference on artificial intelligence, AAAI 2021, p. online. AAAI Press
63. Zuo Y, Liu G, Lin H, Guo J, Hu X, Wu J (2008) Embedding temporal network via neighborhood formation. KDD pp 2857–2866

Meng Liu received his BEng degrees in software engineering from the Henan University of Economics and Law, China. He is currently working toward the master's degree with the School of Computer Science and Technology, Heilongjiang University, China. His research interests include graph embedding and social computing.

Zi-Wei Quan received her BEng degrees in software engineering from the Heilongjiang University, China. Her main research interests include data mining and social network.

Jia-Ming Wu received his BEng degrees in information management and information system from Taiyuan University of Science and Technology, China. He is currently working toward the master's degree with School of Computer Science and Technology, Heilongjiang University, China. His research interest include network representation learning.

Dr. Yong Liu received the PhD degree in computer science from the Harbin Institute of Technology, China. He is currently an associative professor in the School of Computer Science and Technology at Heilongjiang University, China. His research interests include graph mining and social network analysis.



Dr. Meng Han is currently the Director of Research Center for Innovation through Data Intelligence, ZJU-BJ 100-Young Professor at Binjiang Institute of Zhejiang University. Dr. Han got his Ph.D. in Computer Science and MBA from Georgia State University and Georgia Institute of Technology. He is currently an IEEE Senior member, an IEEE COMSOC member, and an ACM member. His research interests include data-driven intelligence, data security & privacy, and financial technology, etc.