Towards Ideal Temporal Graph Neural Networks: Evaluations and Conclusions after 10,000 GPU Hours

Yuxin Yang
University of Southern California
Los Angeles, California
yyang393@usc.edu

Hongkuan Zhou
University of Southern California
Los Angeles, California
hongkuaz@usc.edu

Rajgopal Kannan
DEVCOM Army Research Office
Los Angeles, California
rajgopal.kannan.civ@army.mil

Viktor Prasanna
University of Southern California
Los Angeles, California
prasanna@usc.edu

ABSTRACT

Temporal Graph Neural Networks (TGNNs) have emerged as powerful tools for modeling dynamic interactions across various domains. The design space of TGNNs is notably complex, given the unique challenges in runtime efficiency and scalability raised by the evolving nature of temporal graphs. We contend that many of the existing works on TGNN modeling inadequately explore the design space, leading to suboptimal designs. Viewing TGNN models through a performance-focused lens often obstructs a deeper understanding of the advantages and disadvantages of each technique. Specifically, benchmarking efforts inherently evaluate models in their original designs and implementations, resulting in unclear accuracy comparisons and misleading runtime. To address these shortcomings, we propose a practical comparative evaluation framework that performs a design space search across well-known TGNN modules based on a unified, optimized code implementation. Using our framework, we make the first efforts towards addressing three critical questions in TGNN design, spending over 10,000 GPU hours: (1) investigating the efficiency of TGNN module designs, (2) analyzing how the effectiveness of these modules correlates with dataset patterns, and (3) exploring the interplay between multiple modules. Key outcomes of this directed investigative approach include demonstrating that the most recent neighbor sampling and attention aggregator outperform uniform neighbor sampling and MLP-Mixer aggregator; Assessing static node memory as an effective node memory alternative, and showing that the choice between static or dynamic node memory should be based on the repetition patterns in the dataset. Our in-depth analysis of the interplay between TGNN modules and dataset patterns should provide a deeper insight into TGNN performance along with potential research directions for designing more general and effective TGNNs.

PVLDB Reference Format:
Yuxin Yang, Hongkuan Zhou, Rajgopal Kannan, and Viktor Prasanna.
Towards Ideal Temporal Graph Neural Networks:
Evaluations and Conclusions after 10,000 GPU Hours . PVLDB, 18(4): 956 - 969, 2024.
doi:10.14778/3717755.3717758

Proceedings of the VLDB Endowment, Vol. 18, No. 4 ISSN 2150-8097.
doi:10.14778/3717755.3717758

PVLDB Artifact Availability:
The source code, dat a, and/or other artifacts have been made available at https://github.com/Yang-yuxin/BenchTGNN.

1 INTRODUCTION

Temporal graphs are a prevalent data type modeling numerous real-world systems of interactions and relations spanning from social networks [2, 34], economics [27, 46] to traffic networks [4, 28, 48]. The complexity of temporal graphs stems from the evolving contextual and structural information, as nodes and edges change over time. Dynamic Graph Representation Learning (DGRL) aims at mapping temporal graphs into low-dimensional spaces, that facilitates many downstream applications in forecasting [6, 25, 28], recommendations [2, 7, 58], and anomaly detection [5, 9, 56].

Among DGRL methods, Temporal Graph Neural Networks (TGNNs) stand out as an effective solution. Temporal graphs are divided by the continuity of the timestamps into two categories: Discrete Time Dynamic Graphs (DTDGs) and Continuous Time Dynmic Graphs (CTDGs) [24]. DTDG methods integrate information from a set of static graph snapshots [14, 38, 52, 54], while CTDG methods aggregate information from temporal neighborhoods often obtained by sampling [12, 22, 26, 36, 41, 42, 44, 47, 49, 51]. We focus our discussion on CTDGs since DTDGs are a specific case of the more general CTDGs [59]. On CTDGs, TGNNs generate dynamic node embeddings by performing time-aware message passing, arising from the well-established message-passing paradigms of static GNNs. The interleaved structural and temporal information poses challenges. At the same time, runtime problems arise from the temporal nature of the graphs, such as the proportionally growing size of the temporal neighborhood with time and the time-dependent sequentially aggregated node features.

Initial research efforts in TGNNs centered around the development of effective and efficient models [13, 21, 22, 26, 30, 41, 51, 57, 59, 60]. Recently, a new stream of research has emerged, focusing on the critical evaluation and fair comparison of TGNN models [19, 20, 39]. We identify two critical issues in this emerging research. **First**, the

Table 1: Summary of module performance according to our experimental results. We denote the modules with consistently better performance regardless of varying combinations of other modules as *more effective*, and their opposite as *less effective*. *Dataset dependent* represents the modules whose performance depends on datasets.

Performance	Modules
More effective	Attention aggregator, most recent neighbor sampling, non-learnable time encoding
Dataset dependent	RNN-based node memory, embedding table-based node memory
Less effective	MLP-Mixer aggregator, uniform neighbor sampling, learnable time encoding

evaluation of TGNNs often lacks a comprehensive exploration of the design space. Newer TGNN models typically share similar sets of design choices with previous works, encompassing several *common modules*, with multiple options for each. However, existing works on model advancement pay little attention to searching the design space and are satisfied with incomplete success in limited spaces. Consequently, module advancements may not bring out the full extent of model performance when other parts of the architecture are carelessly designed. **Second**, model comparisons are not performed in a unified framework with reasonable optimization. For example, some works [8, 20, 41] carry out sequential neighbor sampling, while other works use parallel sampling [13]; TGN [41] uses linear probing to identify the last message in a batch, while TGL [59] leverages GPU for parallel processing. The unoptimized code frameworks obscure accurate performance evaluation and create a distorted view of the comparative efficiency of different modules. As a result, there is a limited understanding of TGNN design space, hindering the development of truly optimized models.

In addressing these gaps, our paper proposes a practical approach to TGNN model comparison that compares models at the modular level with a standardized and optimized implementation framework. Unlike existing TGNN frameworks that highlight a complete pipeline [20, 55] or distributed acceleration [59], our methodology is developed and optimized for evaluating individual modules to discern their contribution to the overall model efficacy. Our comparison work focuses on achieving optimality where high predictive accuracy and resource efficiency are balanced. Spending over 10,000 GPU hours (NVIDIA RTX 6000 Ada) on seven datasets, we explore how module choices, sampling strategies, and dataset patterns influence optimality. Specifically, we make the first efforts to address three key *interrelated* research questions in this work:

RQ1. Efficiency and Cost-effectiveness of Module Designs. What module designs, such as the choice between lightweight and more complex neighbor aggregators, strike the optimal balance between effectiveness and resource efficiency?

RQ2. Universality of Module Effectiveness across Datasets. Do the best-performing modules on some datasets, like specific types of node memory, maintain their effectiveness universally across all datasets, or is module performance dependent on dataset characteristics?

RQ3. Interplay between Different Modules in the Model. Does the integration of various modules enhance or undermine performance? For example, does the combination of node memory and a deeper neighbor sampling strategy integrate the improvements offered by each module?

Through extensive experiments based on a modular comparison framework, we obtain some initial answers to the **RQs**. We highlight some of the more important findings below with more detailed findings discussed subsequently. 1. From the cost-effectiveness perspective, *most recent* neighbor samplers are better than *uniform* neighbor samplers and *attention-based* neighbor aggregators are better than the recently proposed *MLP-Mixer* neighbor aggregators. 2. We propose static node memory as a powerful node memory. Further comparison with RNN-based node memory shows that the effectiveness of node memory is largely influenced by dataset repetition patterns, which we also verify using synthetic datasets. 3. Combining node memory with a deeper neighbor sampling strategy proves ineffective due to their overlapping effects. We summarize results about TGNN module effectiveness in Table 1. Our experimental results align with previous theoretical works, while also providing an enlightening perspective on datasets. Meaningful conclusions about TGNN modules are drawn from the results, leading to numerous research directions.

The paper is structured as follows. In Section 2, we introduce several widely-used TGNN models, examine their common architectures, and develop a unified modular framework for evaluating such TGNN models. In Section 3, we detail the implementation of our modular comparison framework and experimental settings. In Section 4, we analyze how various modules and dataset characteristics impact model performance and optimality. In Section 5, we answer the research questions above based on experimental results. Finally, in Section 7, we summarize our findings and reveal their implications for future research.

2 BACKGROUND

Before delving into our experimental design and findings, we review popular TGNN models and existing comparative analyses. We reveal the extensive shared components across different models and identify two crucial issues arising from the lack of modular comparison frameworks: (1) TGNN models are making modular advancements on suboptimal model frameworks, leading to possible flaws in the advancements claimed. (2) Many current models use unoptimized code, which heavily hinders a fair comparison.

2.1 Temporal Graph Neural Networks

Temporal graphs can be interpreted as a set of timestamped interactions among nodes. Continuous time temporal graphs are usually represented as $\mathcal{G}(\mathcal{V}, \mathcal{E})$ with events $\{(u, v, \boldsymbol{e}_{uvt}, t)\}$, where each quadruplet represents an interaction at time t from node u to node v with feature $\boldsymbol{e}_{uvt}$. TGNNs leverage structural and temporal information in a temporal graph to generate informative

low-dimensional node embeddings for prediction tasks. Like static GNNs, TGNNs utilize a message-passing framework to iteratively aggregate information from every node's temporal neighborhood.

In this work, we mainly compare TGNN models that leverage an update-sampling-aggregation scheme as they form the major stream of TGNN models. These models share a general pipeline that aggregates information with an evolving timestamp: Given the timestamp for prediction, the model first updates the temporal neighborhoods and memories of the nodes. Next, past node-node interactions are sampled as neighbors within a node's temporal neighborhood. Finally, an aggregator combines the neighbors, node features, and node memories into an embedding used for prediction.

This entire process can be encapsulated using three key modules: neighbor sampling, neighbor aggregation, and node memory. As illustrated in Table 2, many widely-used TGNN models can be described using this *unified* and *modular* framework.

Table 2: Design choices of different TGNN models. *Neighbor Sampling, Node Memory, Neighbor Aggregator* columns contain the choices of model components. *Add-on* denotes the add-on designs of corresponding models. MR: Most Recent neighbor sampling. RNN: Recurrent Neural Networks (including GRU, LSTM).

Model	*Neighbor Sampling*	*Node Memory*	*Neighbor Aggregator*	*Add-on*
TGN [41]	MR	RNN	Attention	Learnable time encoding [23]
TGAT [51]	Uniform	-	Attention	Learnable time encoding
APAN [47]	MR	RNN	Attention	Learnable time encoding
GraphMixer [12]	MR	-	MLP-Mixer	Fixed time encoding
JODIE [26]	MR	RNN	Attention	Embedding projection one-hot identifier Learnable time encoding

Note that random walk-based models [3, 22, 49] utilize traditional graph analytic algorithms to boost the performance of TGNNs, thus we do not discuss them in this work.

2.2 A modular perspective of TGNN models

We elaborate on our framework by formalizing every module.

Neighbor sampling TGNNs use samplers to sample past interactions of a node for aggregation. For node v at time t, a neighbor sampler uses a strategy to sample within its temporal neighborhood $\mathcal{N}(v, t_0) = \{(v, u, \boldsymbol{e}_{uvt}, t) \in \mathcal{E}, t < t_0\}$. Existing works often employed the most recent sampling [41] or uniform sampling [51]. In most recent sampling, a given number of most recent neighbors are sampled prior to the current timestamp. In uniform sampling, neighbors are uniformly sampled within all past interactions. Previous works also tested inverse time-span sampling, which performs worse than uniform sampling [51]. We do not include the adaptive sampling methods [13, 21] due to the tremendous computation and communication overheads introduced.

Node memory Node memory is a module that stores information from a node's past interactions. They were usually implemented with a fixed-length embedding updated with interactions by sequential models [26, 41]. For an event $(u, v, \boldsymbol{e}_{uvt}, t)$, two messages are generated and sent to the mailboxes of node u and node v:

$$\mathbf{m}_u = \left\{\mathbf{s}_u || \mathbf{s}_v || \Phi(t - t_u^-) || \mathbf{e}_{uvt}\right\} \tag{1}$$

$$\mathbf{m}_v = \left\{\mathbf{s}_v || \mathbf{s}_u || \Phi(t - t_v^-) || \mathbf{e}_{uvt}\right\}, \tag{2}$$

Here $\Phi(\cdot)$ is the time encoding function, t_u^- is the timestamp of the last update, s_u and s_v are the latest node memory for nodes u and v, and $\mathbf{e}_{uvt}$ is the edge feature representing the interaction. Then, we use an update function UPDT, usually a Recurrent Neural Network (RNN) or its variant, to update the node memory of u and v,

$$\mathbf{s}_u = \text{UPDT}\left(\mathbf{s}_u, \mathbf{m}_e^{uv}\right) \qquad \mathbf{s}_v = \text{UPDT}\left(\mathbf{s}_v, \mathbf{m}_e^{vu}\right) \tag{3}$$

In batch training, interactions with node v in the same batch are usually reduced to the mean or the last interaction of the whole batch for efficiency.

To reduce the costs of memory updates, some work [10] leveraged a learnable static node embedding to capture the information of active nodes, bringing more flexibility to training and expediting inference. Some works also used a mixture of static and dynamic node embeddings [10, 60].

Neighbor aggregator Neighbor aggregators are stacked to build a multi-layer TGNN model. It produces the embedding of any node v at time t by aggregating the embedding vectors of the sampled neighbors. Attention aggregator has been a popular choice among neighbor aggregators [26, 41, 47, 51]. Let $\boldsymbol{h}_v^{(l-1)}$ be the $(l-1)$-layer embedding of node v (generated by previous layers), attention aggregator performs attention-based aggregation within its $(l-1)$-layer neighborhood:

$$\boldsymbol{h}_v^{(l)} = \text{Atten}(\boldsymbol{h}_v^{(l-1)}, \boldsymbol{M}_v^{(l)}) \tag{4}$$

where $\boldsymbol{M}_v^{(l)}$ is the l-th layer message matrix of temporal neighborhood $\mathcal{N}(v, t)$. In addition to the commonly used attention aggregator, a recent work, GraphMixer [12], claimed that simply stacking feed-forward and convolution layers yields good node embedding.

$$\boldsymbol{h}_v^{(l)} = \text{Mean}\left(\text{MLP-Mixer}\left(\boldsymbol{M}_u^{(l)}\right)\right). \tag{5}$$

Theoretically, the complexity of MLP-Mixer [43] is linear in the number of inputs, while the attention aggregator has a quadratic complexity. As TGNNs typically do not encode temporal neighborhoods with excessively long sequences, the complexity of using these two modules is comparable in practical experiments.

Other modules There are several powerful add-on modules in TGNN models, including time encoding [12, 41, 51], neighborhood representations [32], time-lapse noise [26], etc. These modules usually produce embeddings as augmentations to existing features at preprocessing, postprocessing, or aggregation stages. We briefly discuss the widely employed time encoding module and leave out the other model-specific modules.

2.3 A unified model framework for comparison and benchmarking

Despite the numerous shared design choices, existing works often use different implementations for the same module, some of which involve unoptimized code. For example, TGN implementation involves a time-consuming memory updating stage with slow linear probing to find the last message in a batch. This occupies almost 50% of total execution time where batch size $bs = 600$. However,

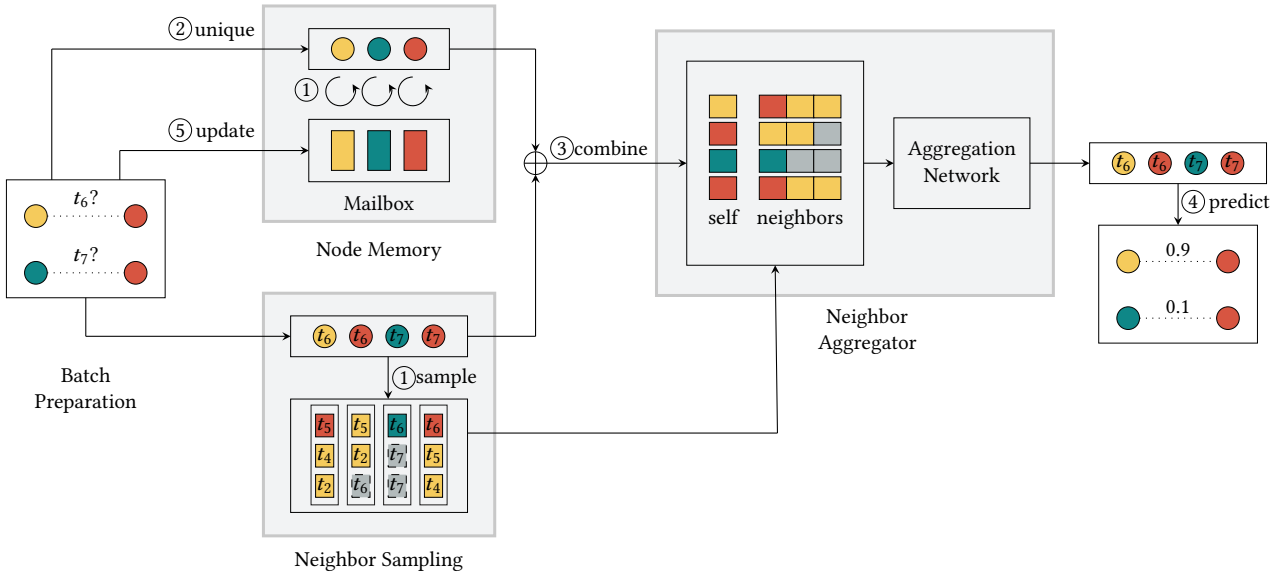

Figure 1: Overview of our generalized update-sampling-aggregation TGNN pipeline. We focus our discussion on the three modules in grey rectangles: node memory, neighbor sampling, and neighbor aggregator. We use small circles to represent nodes and rectangles to represent edge embeddings. Arrows indicate the flow of data, encompassing transformations and rearrangements. In the neighbor sampling module, dotted dark grey rectangles represent dummy nodes where adequate valid neighbors are unavailable. The circled numbers illustrate the sequence of task execution, determined by task dependencies. The same numbers represent that the tasks can be executed parallelly without dependency issues.

this time may be reduced by a factor of ten in our code. These unoptimized implementations hinder a fair comparison of the runtime and efficiency of those modules, consequently obstructing insights into the cost-effectiveness of TGNN modules. We aim to bridge this gap by evaluating the effectiveness of these modules under a unified and optimized framework (See Figure 1). To the best of our knowledge, no works provided a fair comparison of these models at such granularity. We also use an improved metric (Mean Reciprocal Rank, MRR) and include a more challenging inductive setting, following some recent benchmarking works [19, 20].

3 METHODOLOGY

In this section, we describe our experimental framework, including the tasks, metrics, and validation framework for comparison.

3.1 Tasks and Metrics

We evaluate the models with the link prediction tasks on continuous-time dynamic graph datasets [26, 35, 37, 39]. The datasets for node classification tasks are not well-established, with poor and biased label quality in existing datasets like *Wikipedia* and *REDDIT*. Temporal graph classification has been introduced recently [31]. Additionally, the lack of supervision in node classification tasks has led to a challenging two-stage training pipeline that is difficult to optimize or evaluate. We focus our evaluation on the self-supervised link prediction task, which generates informative embeddings with good performance in downstream tasks [41, 51, 59]. We use Mean Reciprocal Rank (MRR) as the performance metric [20, 53], for its enhanced capability to capture the ranking of the positive edge among the mixed positive and negative edges [20].

In the transductive setting, we directly train and test our model on these sets. In the inductive setting, we mask 10% of the nodes from test sets as unseen nodes and remove all edges with these unseen nodes from the training set. The edges containing the unseen nodes are considered inductive samples in the test set. We sample 1:1, 1:9, and 1:49 positive and negative edges on train, validation, and test sets. For negative edge sampling within bipartite graphs, we only select destination nodes from the class different from that of the source node.

3.2 Datasets

We conduct our evaluation using a variety of real-world temporal graph datasets from diverse domains at all scales. The statistics of the datasets are shown in Table 3. The details of the temporal datasets are as follows:

Wikipedia [26] is a bipartite graph of online edits. Editors and pages are nodes, and edges with timestamps represent the editing events. Edge features are extractions of the edited text content.

Table 3: Dataset Statistics. $|d_v|$ and $|d_e|$ denote the dimensions of node features and edge features, respectively. $\bar{k}$ denotes average node degree. Average degree is computed by $\frac{\#Used\ Edges}{\#Nodes}$.

	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	$\lvert d_v\rvert$	$\lvert d_e\rvert$	$\bar{k}$	$\max(t)$
Wikipedia	9,227	157,474	-	172	17.07	2.7e6
Reddit	10,984	672,447	-	172	61.22	2.7e6
UCI	1,900	59,836	-	-	31.49	1.7e7
CollegeMsg	1,900	59,835	-	-	31.49	1.1e9
MOOC	7,144	411,749	172	4	57.64	2.6e6
LastFM	1,980	1,293,103	172	2	505.05	1.3e8
Flights	13,169	1,927,145	100	-	75.94	1.2e2
GDELT	16,682	191,290,882	413	130	59.94	1.6e8
SuperUser	194,086	1,443,339	128	-	7.44	2.4e8

REDDIT [26] is a bipartite graph of user posts. Users and subreddits are nodes, and edges are interactions as users post to the subreddits. Edge features are extracted from the texts of the posts.
MOOC [26] is a bipartite online course content providing interaction network. Students and course units are nodes. Edges represent users interacting with a course unit.
LastFM [26] is a bipartite user-song interaction network. Users and songs are nodes. Edges represent user listening events.
UCI [35] is a user-forum social network. Nodes are students at the University of California, Irvine. Edges are interactions of online messages between users. Features are extracted from the messages.
CollegeMsg [37] is a dataset derived from the social network introduced in dataset UCI.
Flights [39] is a transportation network of airline travels. The nodes are airports, while the edges are flights. Edge features are the number of daily flights between two airports.
GDELT [59]is a global event knowledge graph. Nodes represent actors, and edges represent events. The node and edge features are the CAMEO codes of actors and events.
SuperUser [29] is a temporal network of interactions on the stack exchange website SuperUser. The nodes are users, and the edges represent different types of interactions.

The node features of *LASTFM*, *MOOC*, and *SuperUser* are randomly generated following previous works [29, 59, 60]. For large datasets, we use the last 1 million edges following [13] and split them chronologically into 70%-15%-15% to build train, validation, and test sets.

3.3 Model implementation

In this section, we present our experimental setting by describing the framework first and then the implementation of the neighbor sampler, memory module, and neighbor aggregator. As our framework is modular, it is relatively easy to incorporate and evaluate other methods.

TGNN framework. Our TGNN framework (Figure 1) consists of four stages: neighbor sampling, node memory update, neighbor aggregation, and prediction. After the neighbor sampler samples the edge batch node-wise from the temporal neighborhood, we update the node memory. Then, we use the neighbor aggregator to aggregate the sampled neighborhood for every node. Finally, a task-specific predictor is employed to generate predictions.

Neighbor sampler. Neighbor sampling is a challenging task on dynamic graphs as node neighborhoods constantly evolve with timestamps. We employ the state-of-the-art GPU sampler presented in [13]. At the pre-processing stage, we sort all the edges and store the temporal neighbors of each node with a T-CSR data structure [59], where neighbors of each node are arranged according to their timestamps. We perform a binary search to identify the timestamp-dependent neighborhood at a given timestamp. During the mini-batch fetching stage, newcontentthe edges are sampled in random or chronological order depending on the type of node memory. We use a GPU sampler to parallelize the sampling process: We assign a block of threads to perform the sampling of the neighbors of a node at a given timestamp. By introducing a parallel neighbor sampler, we significantly reduce the sampling overhead in our model. Following [13], we set batch size $bs = 600$ for the train set and $bs = 100$ for validation and test sets.

Memory module. We implement sequentially updated memory modules and static node memory on GPU. For sequentially-updated memory modules, existing works [26, 41, 47] usually maintained an up-to-date node memory on GPU. They use a Gated Recurrent Unit (GRU) as the node memory updater, with interactions encoded as fixed-length messages. It should be noted that RNN, GRU, or LSTM (Long short-term memory) produce comparable outcomes [41], saving the effort of conducting repeated experiments across all node memory updaters. We collectively refer to this type of memory as RNN-based node memory. In RNN-based node memory, training and inference are accelerated using chronological batch training [26, 41, 47, 59]. Our batch processing is as follows: As in the state-of-the-art [47, 59], we maintain a mailbox for every node. Messages sent to a node in the previous batches are stored in the mailbox, and node memory is updated before processing every batch. If a node receives multiple messages within a batch, the messages are usually reduced to the latest or average. We employ the former for simplification and efficiency, considering there is only a minor difference in performance [41]. For further acceleration, we retrieve the last message by parallelized batch operations on GPU, instead of linear probing as in some existing works [19, 20, 41]. As supervision comes from the current batch, we have to prevent information leakage from the current batch by updating the node memory with cached messages from the previous batch [26, 41, 47]. At the end of each batch processing, we detach the updated node memory from the computation graph so that the depth of backward propagation on RNN parameters is controlled.

For static learnable node memory, we use an embedding table with shape $[|\mathcal{V}|, D_{mem}]$ to store the node memory vectors, where D_{mem} is the dimension of the static node memory. The training set is shuffled in each epoch to generate mini-batches, in which the relevant node memory vectors are directly updated by the gradients. Note that this shuffled mini-batching does not apply to RNN-based memory, as it needs to be updated in chronological order. We follow [13, 59, 60] and set the dimensions of all memory modules to 100, a reasonable length to achieve high performance without introducing too much computation.

Neighbor aggregator. We implement attention block [45] and MLP-mixer [43] as neighbor aggregators. Following existing works [41, 59], we combine raw node features, node memories, and time encodings into node embeddings by concatenation, linear transformations, and additions before neighbor aggregation. The aggregators then perform feature aggregation and produce informative low-dimensional node embedding. For the attention aggregator, we set attention heads $h = 2$ as in [13, 41, 59, 60].

Abbreviations. We summarize all the modules and their abbreviations tested in our work.

- *Neighbor Sampler*: **MR** (most-recent neighbor sampler), **uni** (uniform neighbor sampler)
- *Node Memory*: **RNN** (RNN-based node memory), **emb** (embedding table-based node memory)
- *Neighbor Aggregator*: **atten** (attention neighbor aggregator), **mixer** (MLP-Mixer neighbor aggregator)

3.4 Hardware and Software

We implement our code on a machine with dual 96-core AMD EPYC 9654 CPUs with 1.5TB ECC-DDR5 RAM and NVIDIA RTX 6000 Ada GPU with 48GB ECC-GDDR6 VRAM. Our experiments are performed in an environment with Python 3.9, PyTorch 2.1.0, and CUDA 11.8 on Ubuntu 22.04 LTS with Linux kernel version 5.15.0-100-generic.

To use our framework on multiple GPUs, parallel samplers can be launched on every GPU to sample in the temporal neighborhoods of nodes. The edge features, node features (if any), and the global copy of the node memory is stored in the shared memory. The gradients in each iteration are synchronized among the trainer processes through the NCCL backend.

4 RESULTS

After constructing the benchmarking framework, we run extensive experiments using over 10,000 GPU hours and thoroughly analyze the results. We first separately examine the cost-effectiveness and universality of modules, including neighbor sampling (subsection 4.1), memory module (subsection 4.2), neighbor aggregator, and others (subsection 4.4). Following this, we explore the effectiveness of these components when combined to construct the model. We conclude by addressing the research questions posed in the introduction (Section 1) based on our experimental outcomes.

4.1 Neighbor Sampling

We first concentrate on neighbor sampling as sampling directly affects the information aggregated. Our analysis involves comparing two neighbor samplers, deciding the appropriate sampling budget, and designing a distribution strategy among layers given the total budget. Specifically, we compare two popular samplers: **MR** and **uni** (See Figure 2), both with little overhead ($1\% - 2\%$ of training time) when carried out by our GPU sampler. We observe that **MR** samplers often exceed **uni** by a large margin. When using **emb**, the performance gap is smaller, and sometimes **uni** sampler is better than **MR** sampler. This could be caused by the shared nature of static node embedding and uniform sampling of capturing long-term temporal neighborhoods, resulting in good training dynamics. Due to space limits, we only display the results of 1-layer models on two datasets here. We observe similar phenomena in most experimental settings, indicating consistent superiority of **MR** neighbor sampler over **uni** neighbor sampler across various model settings and datasets. This result aligns with the preliminary results in [30]. We observe a larger MRR improvement margin of **MR** over **uni** on *Wikipedia* than *REDDIT*, which may be explained by the recency of important events in the temporal neighborhood (further explained in Section 4.2). As *Wikipedia* is a dataset with short-term repetition patterns, **MR** may capture more relevant events on it, resulting in even better performance on some datasets.

Next, we explore the effect of neighbor sampling budgets on TGNN model performance. We evaluate the performance of 1-layer and 2-layer models with different numbers of sampled neighbors in each layer. As most TGNNs incorporate layer sampling [13, 26, 41, 42, 44, 51, 59, 60], the neighborhood size grows exponentially with layers (a problem known as neighbor explosion [18]). Considering the high computational burden of layerwise sampling and the impact of a larger receptive field from node memory as discussed in [60], we only perform experiments on ≤ 2 layers. For 1-layer models, we sampled 1 to 100 neighbors. For 2-layer models, we test combinations of 5 and 10 neighbors in each layer.

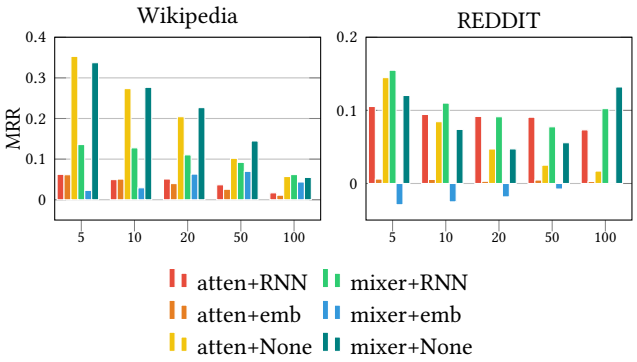

Figure 2: Performance gap between two neighbor samplers across various module combinations on datasets WIKI and *REDDIT*. The values are calculated by subtracting the prediction accuracy of models using uniform sampling from the prediction accuracy of models using the most recent sampling. Results are plotted with varying numbers of sampled neighbors for 1-layer models.

Our results indicate that TGNN models do not need many supporting neighbors or deep model architecture to perform well. Specifically, while some models improve as more neighbors are sampled, models with the highest-performing combinations reach their peaks quickly with less than 10 neighbors in 1-layer models(Figure 3). The result holds for all datasets. Combining past theoretical works [11] and our experimental results, we formulate a *three-fold hypothesis* concerning the impact of neighbor sampling on model performance: 1. **Sampling versus node memory**: The ability of node memory to capture the temporal neighborhood may be compromised due to discarded interactions in mini-batching or defective memory implementations. However, sampling can compensate for insufficient node memory and improve performance, i.e., sampling neighbors directly from a broader neighborhood may compensate for these deficiencies. 2. **Deeper models and generalization gap**: Deeper models induce a generalization gap that contributes to performance saturation. We observe that on small-scale datasets like *UCI* and *CollegeMsg*, sampling more neighbors brings little to no performance improvement. Also, 2-layer models generally achieve better predictions at train time but similar predictions at test time compared to 1-layer models. These results align with the theoretical analyses in [11] on how the generalization error of TGNNs decreases with data volumes and increases with depth (number of layers). 3. **Sampling overcomes suboptimality**: We observe that larger sampling sizes can improve the performance of suboptimal module combinations, alleviating module incompatibility. For instance, pairing **RNN** node memory with uniform sampling is suboptimal since it fails to ensure that the node memory is updated with the latest interactions, potentially compromising the node memory's effectiveness; However increasing the sample size of uniform sampling imitates the reasonable

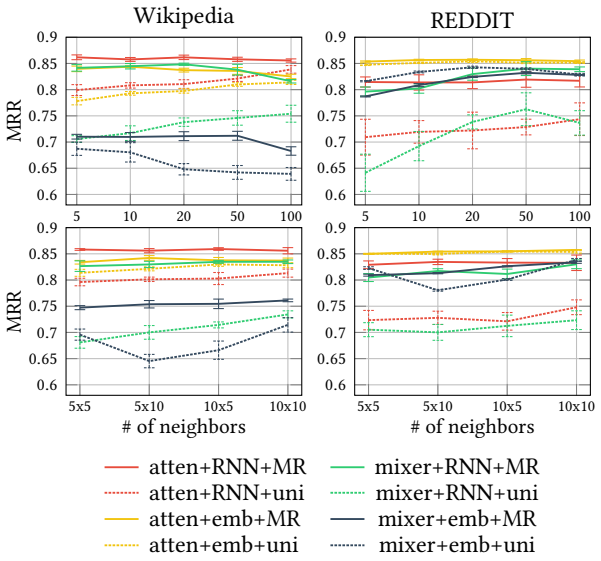

Figure 3: Performance of different combinations of TGNN modules on dataset *Wikipedia* and *REDDIT*. As more neighbors are sampled in models with one layer (first row) and two layers (second row), certain models display enhanced accuracy, while the top-performing model exhibits fast saturation.

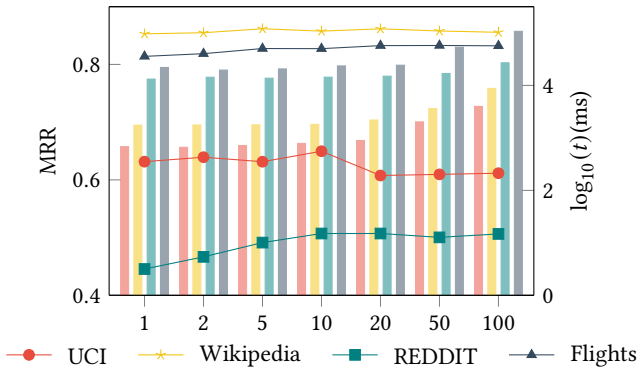

Figure 4: MRRs and runtimes of 1-layer models sampling an increasing number of neighbors on datasets *UCI*, *Wikipedia*, *REDDIT*, and *Flights*. We use atten neighbor aggregator and MR sampling on all datasets. For node memory, we adopt the better-performing ones on each dataset: emb memory on *REDDIT* and *Flights*, and RNN memory on other datasets. MRR is shown by line chart and runtime is shown by bar chart in logarithmic scale. Note that the x-axis uses a scale that has been slightly adjusted to ensure visual tidiness.

MR sampling and memory update process, thus improving model performance, though the combination is suboptimal.

Next, we discuss the cost-effectiveness of neighbor sampling. As the size of the sampled neighborhood grows, model performance follows a convex curve and saturates quickly, whereas the runtime increases linearly. We plot the convex performance curve of MRRs (Figure 4) on all datasets of different scales. While the saturation point differs, 1-layer model with 10 neighbors is close to top-performing on all datasets. Using the best-performing module combination, performance saturates with less than 10 supporting neighbors in 1-layer models. On the other hand, the runtime grows in proportion to the size of supporting neighbors, which is illustrated by a linear trend on the plot with both axes in logarithmic scale.

Note that increasing the depth and width of neighbor sampling still yields large improvements when node memory is absent, as shown in [51]. This corresponds to our first hypothesis. We skip relevant discussions for two reasons: (1). Models without node memory are not the focus of our design space search. (2). We demonstrate in Section 4.2 that without node memory, models with a large sampling budget and longer runtime still produce worse results than very shallow models with node memory.

In summary, the most recent neighbor sampling strategy *performs better* than uniform sampling. In models with node memory, deep or wide neighbor sampling brings minor improvements to prediction accuracy but substantially increases the runtime. We claim that a small neighbor sampling budget (e.g., 1-layer models with 10 neighbors) suffices to aggregate neighborhood information.

4.2 Memory modules

Carefully designed node memory can alleviate computational burdens by capturing the informative temporal neighborhood, thereby reducing the need for extensive neighbor sampling and aggregation. We analyze the performance of **RNN** and **emb** node memory and demonstrate how to select the appropriate node memory. First, we show by empirical results that the choice of node memory significantly affects the model performance. Then, we propose a hypothesis on the connection between node memory and dataset repetition patterns. After that, we introduce several measures to analyze the repetition patterns qualitatively and quantitatively. We then reveal a relationship between the repetition patterns of datasets and the performance of node memories. Finally, we construct synthetic datasets with different repetition patterns and validate our hypothesis with the results on the synthetic datasets.

4.2.1 Empirical results and hypothesis. We run 1-layer models with 5 to 100 neighbors and examine the performance of **emb** and **RNN** memory on all datasets (see Figure 5). All datasets have results lying on one side of the dividing red line, indicating preferences towards one of the node memory. This preference is not shared across datasets and is independent of other designs.

We explain the preference of datasets towards one type of node memory by closely examining the temporal graph datasets and revealing repetition as a fundamental feature of the datasets. Prior works have analyzed the frequency and effects of edge repetition in some widely used graph datasets [39, 40]. While these works show that repetition is common, we further reveal how different repetition patterns affect model performances, especially the performance of node memory.

Based on a simple intuition, we propose a hypothesis and perform experiments to validate it. We first exemplify our hypothesis with two representative datasets: *Wikipedia* and *Flights*.

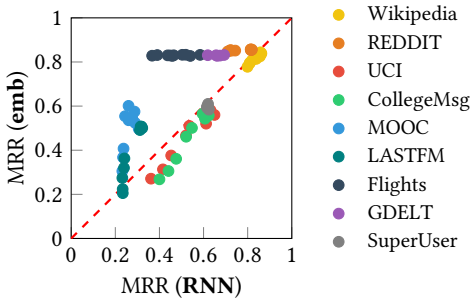

Figure 5: MRR performance of RNN-based and embedding table-based node memory of all datasets. We fix the neighbor sampling to the most recent strategy and the neighbor aggregator to the attention aggregator. 1-layer models are used with 5-100 neighbors on all datasets. The red diagonal dashed line shows equal performance by RNN-based and embedding table-based node memory.

- *Wikipedia* is an interaction graph of user edits, which exhibits strong temporal locality by automatically saving a user's editing events of the same page repeatedly during an editing session.
- *Flights* records the flights between airports and displays a stable repetition pattern throughout a comparatively long period.

From the perspective of a source node, a recurring event is two interactions with the same destination nodes at different timestamps. For Wikipedia, remembering recent interactions will likely help predict future links because recurring events are temporally dense. For dataset *Flights*, a static vector may suffice to encode all possible interactions as recurring events are periodic.

We introduce *short-term repetition patterns* for describing recurring events with a short time gap in between, and *long-term repetition patterns* for describing recurring events with a long time gap in between. With these analyses, our hypothesis is formally stated: Embedding table-based node memory works better on temporal graph datasets with *long-term repetition patterns*, while RNN-based node memory works better on those with *short-term repetition patterns*.

4.2.2 Measuring dataset repetition. To reveal the nature of dataset repetition patterns, we show results from three measures: the recurrence matrix, a quantitative indicator *temporal recency ratio* ϕ_τ, and distribution fitted to session lengths.

Recurrence matrices are widely used tools for measuring recurrences of a trajectory $\vec{x}_i \in \mathbb{R}^d$ bounded by an error [33]. In our case, $\vec{x}_i \in \mathbb{R}^{|\mathcal{V}|}$ represents an interaction with source node i. The time-stamped interactions $\vec{x}_i^{(t)}$ form a trajectory that reflects interactions with source node i. In $\vec{x}_i^{(t)}$, indices corresponding to the source and destination nodes are marked 1 and -1, and the other locations are all 0. Error is set to be an arbitrarily small ϵ so that only interactions with the same destination are considered as recurrences. Algorithm 1 shows the construction of recurrence matrix $\mathbf{M}_R$ given bin number B. The complexity of this algorithm

Algorithm 1: Temporal recurrence matrix generation

Input : T-CSR graph $\mathcal{G}$, bin number B, dataset time range $[t_{\min}, t_{\max}]$
Output: Recurrence Matrix $\mathbf{M}_R$

$\Delta_B \leftarrow \frac{t_{\max} - t_{\min}}{B}$;
$\mathbf{M}_R \leftarrow \mathbf{0}_{B \times B}$;
for $u \in \mathcal{V}$ **do**
  for $(i, j) \leftarrow (1, 2)$ **to** $(|\mathcal{N}_u| - 1, |\mathcal{N}_u|)$ **do**
    $n_i \leftarrow \mathcal{N}_u^{(i)}$, $n_j \leftarrow \mathcal{N}_u^{(j)}$;
    if $\texttt{Dst}(n_i) = \texttt{Dst}(n_j)$ **then**
      $b_i \leftarrow \frac{t_{n_i} - t_{\min}}{\Delta_B}$, $b_j \leftarrow \frac{t_{n_j} - t_{\min}}{\Delta_B}$;
      $\mathbf{M}_R[b_i][b_j] \leftarrow \mathbf{M}_R[b_i][b_j] + 1$;
  end
end

is $O(D^2|\mathcal{V}|)$, where D is the maximum degree. The complexity of this algorithm could be controlled by clipping the maximum degree.

We run Algorithm 1 and show recurrence matrices with heatmaps (See Figure 7). We validate our intuition about the distinct short-term and long-term repetition patterns. *Wikipedia* displays bright areas around the diagonal line, showing that recurring events happen within a short period. *REDDIT* display vastly different patterns, which indicates that recurring events are highly periodic with long intervals in between.

We further develop a metric, the temporal recency ratio ϕ_τ, to quantitatively assess the positions of datasets along the axis from short- to long-term repetition patterns. This metric is defined based on the τ-recurrence rate [33], denoted as $RR_\tau = \frac{1}{N-\tau} \sum_{i=1}^{B-\tau} \mathbf{R}_{i,i+\tau}$, where B denotes time bins. The τ-recurrence rate calculates the likelihood of an event repeating after τ time steps. A distribution of RR_τ heavily biased towards smaller τ values indicates short-term repetition patterns, whereas a more uniform distribution on varying τ signifies long-term repetition patterns. We define the temporal recency ratio as follows:

$$\phi_\tau(\mathcal{G}) = \sum_{i=1}^{B} \frac{RR_i}{\sum_{j=1}^{i} RR_j} \tag{6}$$

where B is the bin number. ϕ_τ is small if RR_τ decreases sharply with time (indicative of short-term repetition patterns). It is large if RR_τ maintains a stable value across different τ values (indicative of long-term repetition patterns). Additionally, our definition of ϕ_τ brings two advantageous characteristics:

(1) Independence of time scale: Our datasets have different time scales for their varied sources (Table 3). ϕ_τ uniformly measures the repetition regardless of the dilation or shrinkage of time scales.
(2) Independence of repetition frequency: The strength of repetition may vary depending on how events are recorded. ϕ_τ is also independent of repetition strength, focusing solely on distinguishing between long-term and short-term repetition patterns.

We summarize the repetition patterns in the recurrence matrices and *temporal recency ratio* $\phi_\tau(\mathcal{G})$ of our temporal datasets in

Table 4 and visualize representative datasets in Figure 7. There are two noteworthy datasets: 1. *MOOC* dataset is considered an exception by encompassing very sparse repetitions, as shown in Figure 7. The better performance of **emb** on the dataset could be attributed to its role in feature augmentation. 2. *SuperUser* is a dataset with both *short-term* and *long-term repetition patterns*, as shown by the recurrence matrix in Figure 7 and the median value of temporal recency ratio. **emb** and **RNN** have similar performance on *SuperUser*. We found that the recurrence matrices and the temporal recency ratio provide a consistent illustration of dataset repetition patterns, which account for its preference for either dynamic or static node memory.

Table 4: Preferred node memory of datasets and the repetition patterns summarized from recurrence matrices and temporal recency ratio $\phi_\tau(\mathcal{G})$.

Datasets	preferred node memory	repetition patterns	$\phi_\tau(\mathcal{G})$
Wikipedia	**RNN**	short-term	0.039
UCI	**RNN**	short-term	0.194
REDDIT	**emb**	long-term	4.201
Flights	**emb**	long-term	4.396
LASTFM	**emb**	long-term	3.114
MOOC	**emb**	sparse	0.333
GDELT	**emb**	long-term	5.705
SuperUser	**RNN**	mixed	2.704

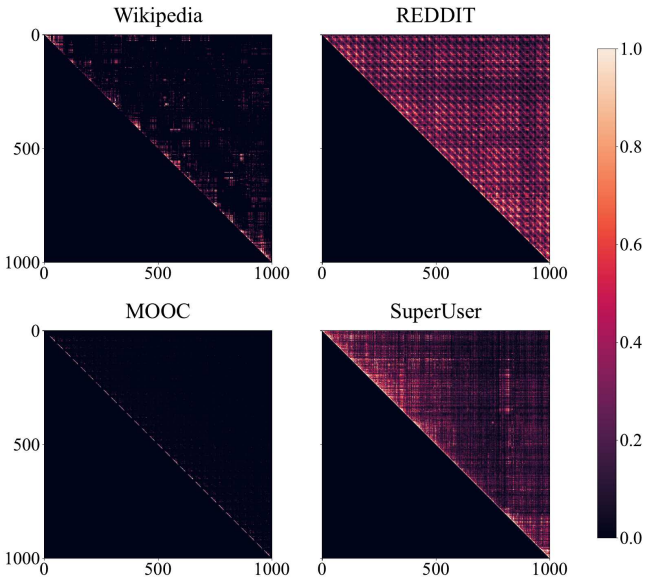

Figure 6: Recurrence matrices of *Wikipedia*, *REDDIT*, *MOOC*, and *SuperUser*. Extreme values are truncated, and matrices have been normalized. The x-axis and y-axis are bins mapped from timestamps. The value at position i, j in the matrices stands for the count of recurring events at time bins i and j. In the heatmap, brighter positions represent larger values, while darker positions represent smaller values.

We also analyze the patterns of session lengths of the datasets and view repetition from the perspective of sessions. We measure periods of consecutive events between the same sources and destinations by $\Delta_t = t_{last} - t_{first}$. Following the literature of session identification [17], we plot histograms of Δ_t on a logarithmically-scaled x-axis. Session identification methods usually leveraged an inactivity threshold t_T as the estimated cut-off where the probabilities of between-session and within-session reach equality [17]. We derive that threshold by fitting the histogram under the scaled x-axis to a two-component Gaussian mixture model with a minimal squared difference and finding the intersection point (Figure 7). Due to space limits, we only display the results from datasets *Wikipedia* and *REDDIT*. We find that the datasets with distinguishable long and short sessions (i.e., with a clear intersection point) are those with short-term repetition patterns. In contrast, the datasets with vague or unrecognizable sessions display a long-term periodic repetition or little repetition. This provides a deeper interpretation of *short-term repetition patterns* as sessions and enables us to transfer existing methodology in this area of research to dealing with temporal graph datasets.

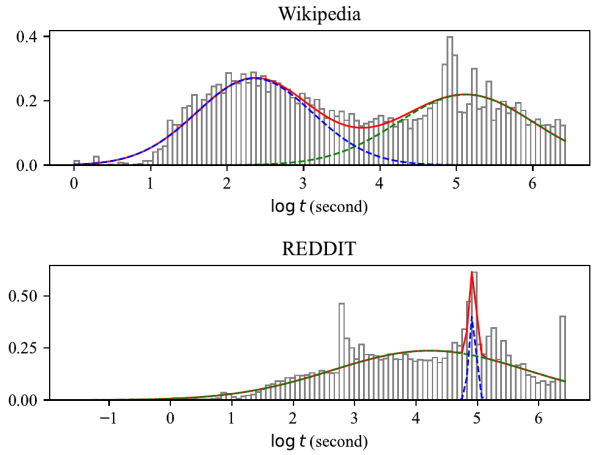

Figure 7: Session length distributions of datasets. The x-axis represents time on a logarithmic scale, and the y-axis represents the probability distribution density. The blue and green dashed lines represent two Gaussian distributions, while the red line represents the mixed distribution.

Finally, we summarize our analyses and restate our claim. With the measurement of dataset repetition using both recurrence matrices and temporal recency ratio, we categorize the dataset repetition into *short-term* and *long-term* accordingly. We observe an alignment between the repetitions and a preference towards **RNN** or **emb** node memory: datasets with *long-term repetition patterns* have a strong preference towards **emb** node memory, while those with *short-term repetition patterns* tend to favor **RNN**.

4.2.3 Validation on synthetic datasets. To further minimize interferences and substantiate our claim, we generate synthetic datasets with a more controllable experimental setting than often noisy and biased real-world datasets. We construct the dataset by sequentially generating events at every time step. First, we sample the

Algorithm 2: Synthetic temporal graph generation

Input : node count N, edge count E, timesteps T, cluster coefficient $C_i \sim \mathrm{N}(\mu, \sigma^2)$, long/short-term controller $\alpha \in (0, 1]$, repetition strength controller $\beta \in (0, 1]$, temperature $\mathbf{T}$

Output : $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with events $\{(i, j, \boldsymbol{e}_{ijt}, t_{\boldsymbol{e}})\}$

for $(t, k) \leftarrow (1, 1)$ **to** $(T, \frac{E}{T})$ **do**
  $\delta_{\boldsymbol{e}} \leftarrow \delta_e \sim \mathrm{N}(0, \sigma_t^2)$;
  $t_{\boldsymbol{e}} = t + \delta_{\boldsymbol{e}}$;
  Sample source node i with $p(i) \propto C_i$;
  if $\mathcal{N}(i, t) = \emptyset$ **then**
    Sample destination node j with $p(j) \propto C_i$;
  else
    With probability β, sample destination node j from $\{dst(\boldsymbol{e}), \boldsymbol{e} \in \mathcal{N}(i, t)\}$ with $p(j) \propto \alpha^{\frac{t - t_{\boldsymbol{e}}}{\mathrm{T}}}$;
    With probability $1 - \beta$, sample destination node j with $p(j) \propto C_i$;
  end if
end

timestamp of the events by adding a small deviation to the current time step. Second, the source node is sampled from a predefined distribution f. Third, the destination node is sampled either from historical events or the entire node set. The process for generating a synthetic temporal graph dataset is described in Algorithm 2.

Using α, we construct an exponential-decaying probability distribution of sampling historical neighbors. α approaching 1 represents uniformly sampling all historical neighbors, while α approaching 0 represents only sampling the most recent temporal neighbor. The bias of α towards these two extremes corresponds to *long-term* and *short-term* repetition, respectively. We use β to control the intensity of repetition, where a larger β represents stronger repetition. We show recurrence matrices of the synthetic datasets with different combinations of α and β in Figure 8.

Using synthetic datasets, we conduct controllable experiments on how dataset repetition patterns affect different node memory performance. Our results are summarized in Figure 9. We observe that with a larger α (corresponding to long-term repetitive pattern), **RNN** node memory tends to have a decreased performance, while **emb** node memory performance gradually increases, supporting our previous hypothesis on dataset repetition affecting node memory performance.

4.2.4 Discussions in the inductive setting. We note that **emb** node memory has a drawback in the inductive experimental setting. As the node memory is only updated during the training stage, no information from validation or test can be integrated into node memory to assist future tasks. The new nodes appearing only in the test stage would have only randomized vectors as their node embeddings. Our experiments show a significant drop in test accuracy compared to the transductive setting.

Given the strength of **emb** node memory shown on multiple datasets, we propose a preliminary measure to alleviate the drawbacks of **emb** node memory. We leverage the mailbox in **RNN** node memory to store the messages when events are observed

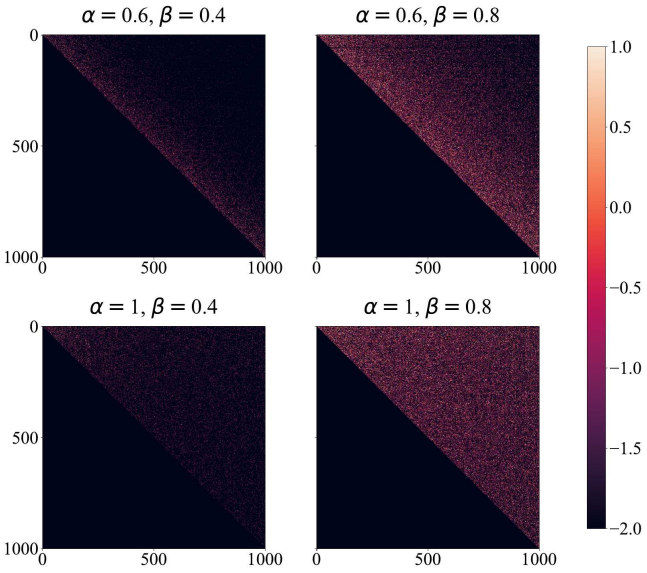

Figure 8: Recurrence matrices of synthetic datasets with different α and β. In every column, larger α corresponds to longer-term repetition patterns. In every row, larger β corresponds to more repetitions.

for inductive nodes. We use the node memory of 1-hop or 2-hop neighbors as messages, depending on whether the graph is bipartite. When the mailbox has enough mails (in our experiments, we set $\#mails = 3$), we smooth the messages as the initial memory of the inductive node. This initialization trick improves inductive MRRs by $\sim 3\%$ at best (Figure 10). Deliberately designed strategies that combine static and dynamic node embeddings in transductive and inductive settings will likely yield better performance. It should be mentioned that online fine-tuning [57] with streaming batches is also applicable in the continuous learning setting.

4.3 Connections and choices between modules

After introducing the neighbor sampling and node memory modules, we explore strategies for effective combined utilization and emphasize the importance of incorporating both modules. Specifically, we evaluate the cost-effectiveness of incorporating node

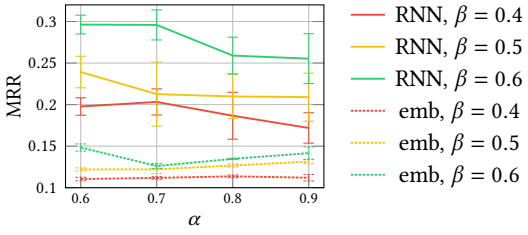

Figure 9: MRRs of TGNN models with emb and RNN node memory, with respect to α and β. We use a 1-layer model with 10 neighbors, MR neighbor sampler, and atten neighbor aggregator.

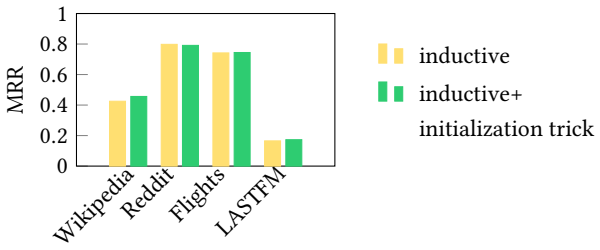

Figure 10: MRRs of models with embedding table-based node memory tested on inductive settings. We fix models to be 1-layer with 10 neighbors, using attention neighbor aggregators and the most recent neighbor sampling. MRRs of models w/ and w/o the embedding initialization trick for inductive settings are displayed.

memory compared to models that are only based on neighbor aggregation(e.g., TGAT [51]). We compare the cost-effectiveness between two approaches: 1. Adopting a deeper model and increasing the size of supporting neighbors; 2. Maintaining node memory in a shallow model with a small supporting neighborhood. Our results in Figure 11 demonstrate that models with node memory generally produce better results than pure sampling with less computation cost.

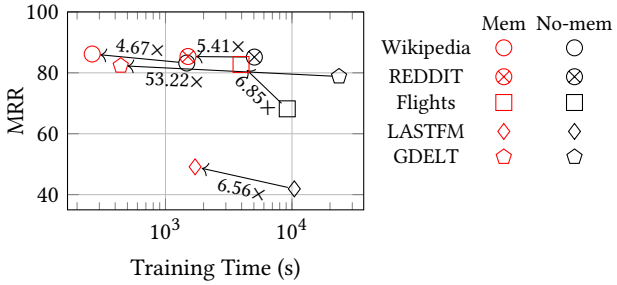

Figure 11: Total training time and MRRs of models w/ and w/o node memory. We use MR neighbor sampler and atten aggregator for both models. We compare the 1-layer model with 5 neighbors w/ memory to the 2-layer model 40 supporting neighbors per layer w/o node memory. On each dataset, we select the better-performing node memory between emb and RNN.

4.4 Discussions on other modules

Finally, we provide clear conclusions on neighbor aggregators and time encodings. While this section includes limited discussion, our results highlight equally meaningful factors to consider in TGNN designs as the previous sections.

4.4.1 Neighbor aggregators. Despite that **mixer** is a method proposed more recently, we found that **atten** neighbor aggregator generally outperforms **mixer** in a variety of settings (See Figure 12). This phenomenon is universally observed on all datasets. We observe that the solid lines standing for **atten** aggregators usually outperform their dashed-line counterparts standing for **mixer** aggregators by a large margin. While the superiority of **atten** neighbor aggregator is only displayed on two datasets due to space limitations, this phenomenon has been observed on all experimented datasets. Despite the theoretical quadratic complexity of **atten**, **atten** has a shorter runtime (< 50% when training in large scale) than **mixer** for our sequence length is usually limited.

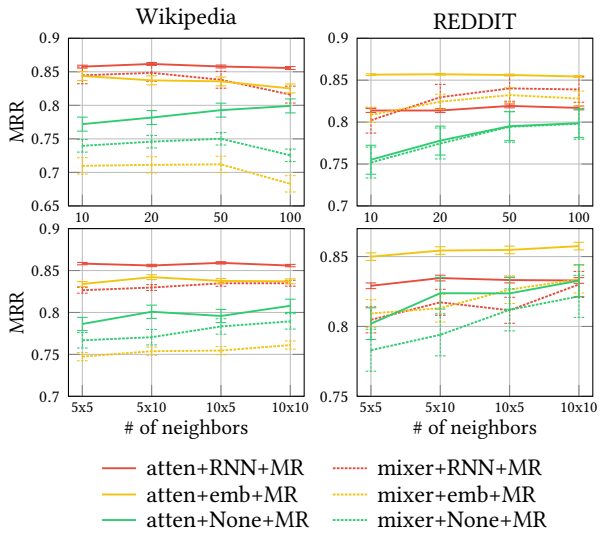

Figure 12: MRRs and runtimes of models with an increasing size of sampled neighbors on datasets *Wikipedia* and *REDDIT*. We show results of atten and mixer neighbor aggregators using solid and dashed lines under various settings on 1-layer (first row) and 2-layer (second row) models.

4.4.2 Time encoding. Most existing works [41, 47, 51] have employed a learnable time encoding $\mathbf{\Phi}$:

$$\mathbf{\Phi}(\Delta t) = \cos(\Delta t \boldsymbol{w} + \boldsymbol{b}), \tag{7}$$

where $\boldsymbol{w} \in \mathbb{R}^{d_T}$ and $\boldsymbol{b} \in \mathbb{R}^{d_T}$ are learnable parameters.

GraphMixer [12] proposed a non-learnable time encoding for events and claims the encoder exhibits smoother optimization, faster convergence, and a better generalization:

$$\mathbf{\Phi}(\Delta t) = \cos(\Delta t \boldsymbol{\omega}), \quad \boldsymbol{\omega} = \left\{\alpha^{-(i-1)/\beta}\right\}_{i=1}^{d_T}, \tag{8}$$

Our experiment results align with the claim of [12] that non-learnable time encoding outperforms learnable time encoding.

5 CONCLUSION

In this work, we proposed a modular TGNN comparison framework with reasonable optimizations, addressing the problem of insufficient design space search and unfair comparison. While neighbor sampling and aggregation are problems discussed in both temporal and static graphs, we focus on temporal graphs to reveal domain knowledge about the TGNN modules. Through extensive experiments, we drew conclusions on the cost-effectiveness of TGNN modules with different sizes and choices, the dependence between

module preferences and datasets, and the relations between different modules. Based on our findings, we now revisit the three research questions presented in Section 1.

RQ1. Efficiency and Cost-effectiveness of Module Designs: What designs strike the optimal balance between effectiveness and resource efficiency? We demonstrate that **TGNNs should adopt most recent neighbor sampling, attention-based neighbor aggregator, and maintain a node memory for effectiveness and efficiency**. We showed that: 1. With a similar or lower runtime, most recent neighbor sampling consistently outperforms uniform neighbor sampling. 2. Despite the fact that MLP-Mixer is proposed later than attention aggregator, attention neighbor aggregator outperforms the MLP-Mixer neighbor aggregator. 3. With comparable or even better model performance, we demonstrated that maintaining node memory is more efficient than increasing the number and layer of neighbor sampling to achieve comparable results.

RQ2. Universality of Module Effectiveness across Datasets: Do the best-performing modules on some datasets maintain their effectiveness universally across all datasets, or is module performance dependent on dataset characteristics? For neighbor sampling, neighbor aggregator, and time encoding, our experiments show consistent results of accuracy and runtime comparisons across all datasets. For node memory, our results show that **datasets exhibit a preference for different types of node memory based on whether their repetition patterns are long-term or short-term**. We demonstrate that the former kind of dataset prefers embedding table-based node memory, while the latter prefers RNN-based node memory.

RQ3. Interplay between Different Modules in the Model: Does the integration of various modules enhance or undermine performance? We observe that **when applying node memory, a deeper or wider neighbor sampling often brings little performance gain** to the model. This can be attributed to the overlapping effect of an enlarged receptive field brought by using node memory and sampling a large temporal neighborhood. We also find that combining RNN-based node memory and uniform neighbor samplers yields a poor performance for potentially not using the latest information to update the RNN-based node memory.

6 RELATED WORK

Benchmarking temporal graph learning. Recent Temporal Graph Learning benchmark works provide platforms for comparisons and conducting empirical comparisons of models.

EdgeBank [39] illustrates the limitations of tasks and metrics by showing the high prediction accuracy of a simple baseline. They also analyze dataset deficiency and enhance evaluation. TGB [20], BenchTemp [19], and DyGLib [55] provide evaluation pipelines for temporal graph learning. Their comparisons found that the performance of models drastically varies on tasks and datasets.

On the other hand, some works focus on conducting empirical model comparisons and drawing meaningful conclusions from results. Chen et al. [8] conduct a detailed bottleneck profiling of the runtime of TGNN models. Gravina et al. [16] conduct experiments on node- and edge-level tasks using recent TGNNs.

However, existing works lack the support for design space exploration. Moreover, benchmarking is performed using different model implementations, often without a reasonable amount of optimization. In this work, we made extensive efforts to fill this gap.

Improving the efficiency of temporal graph neural networks. A variety of works have laid emphasis on increasing the efficiency of TGNNs. TGL [59] focuses on resolving scalability challenges by exploring parallelizing opportunities using time-centric data structures, mechanisms, and graph engines. Gangda et al. [13] introduce an adaptive temporal graph sampling algorithm to achieve better results with smaller sampling sizes. Alomrani et al. [1] leverage temporal edge encodings and window-based subgraph sampling to generate embeddings.

On top of the TGNN model, various methods accelerate TGNNs by leveraging the characteristics of some TGNNs. DistTGL [60] and GNNFlow [57] are two distributed frameworks designed for faster training paradigms and comparable performance. GNNFlow [57] also presents support for continuous TGNN training. TGOpt [50] reduces the redundancies to accelerate the inference performance of TGAT [51]. Orca [29] caches intermediate embeddings for reusing to reduce redundant computations. Shihong et al. [15] accelerate TGNN training by improving the data batching and accessing manner to enlarge the batch size and reduce access volume.

7 FUTURE DIRECTIONS

Exploring additional benchmarking aspects. Future works can extend the benchmarking efforts of our work, such as exploring the effects of i) sampling with exponential decaying probabilities, ii) combining static and dynamic node memories, and iii) using different methods for negative sample selection. Comparison and validation of modules may also be carried out on random walk-based TGNN models. In addition to prediction accuracy, evaluation can be expanded to model robustness, generalization, and privacy.

Improving the design of TGNN modules. Existing modules exhibit limited adaptability on datasets. For example, one node memory type only performs well on short- or long-term repetitive datasets. Given that valuable information is distributed in various forms over past time periods, it is important to explore how more universal TGNN modules can be developed to effectively capture this information across diverse datasets and learning settings.

Examining dataset patterns. We demonstrated that TGNN dataset patterns remain largely unexplored, potentially preventing us from designing useful and efficient neighbor aggregators and neighbor sampling methods. Besides dataset repetition patterns discussed in this work, dataset heterophily and frequency domain patterns may be investigated.

Temporal neighbor modeling and sampling methods. Our findings indicated an orderly distribution of historical neighbors that could be fitted to some probability distributions. This insight can enhance temporal neighbor modeling or enable smart sampling that aggregates more informative neighbors with a lower budget.

ACKNOWLEDGMENTS

This work was supported by the DEVCOM Army Research Lab (ARL) under grants W911NF2220159 and W911NF2320186 and the National Science Foundation (NSF) under grant SPX-2333009.

REFERENCES

[1] Mohammad Alomrani, Mahdi Biparva, Yingxue Zhang, and Mark Coates. 2023. DyG2Vec: Efficient Representation Learning for Dynamic Graphs. *Transactions on Machine Learning Research* (2023).

[2] Ting Bai, Youjie Zhang, Bin Wu, and Jian-Yun Nie. 2020. Temporal graph neural networks for social recommendation. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 898–903.

[3] Ali Behrouz, Farnoosh Hashemi, Sadaf Sadeghian, and Margo Seltzer. 2024. CAT-Walk: Inductive Hypergraph Learning via Set Walks. *Advances in Neural Information Processing Systems* 36 (2024).

[4] Khac-Hoai Nam Bui, Jiho Cho, and Hongsuk Yi. 2022. Spatial-temporal graph neural network for traffic forecasting: An overview and open research issues. *Applied Intelligence* 52, 3 (2022), 2763–2774.

[5] Lei Cai, Zhengzhang Chen, Chen Luo, Jiaping Gui, Jingchao Ni, Ding Li, and Haifeng Chen. 2021. Structural temporal graph neural networks for anomaly detection in dynamic graphs. In *Proceedings of the 30th ACM international conference on Information & Knowledge Management*. 3747–3756.

[6] Defu Cao, Yujing Wang, Juanyong Duan, Ce Zhang, Xia Zhu, Congrui Huang, Yunhai Tong, Bixiong Xu, Jing Bai, Jie Tong, et al. 2020. Spectral temporal graph neural network for multivariate time-series forecasting. *Advances in neural information processing systems* 33 (2020), 17766–17778.

[7] Jianxin Chang, Chen Gao, Yu Zheng, Yiqun Hui, Yanan Niu, Yang Song, Depeng Jin, and Yong Li. 2021. Sequential recommendation with graph neural networks. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval*. 378–387.

[8] Hanqiu Chen, Yahya Alhinai, Yihan Jiang, Eunjee Na, and Cong Hao. 2022. Bottleneck Analysis of Dynamic Graph Neural Network Inference on CPU and GPU. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 130–145. https://doi.org/10.1109/IISWC55918.2022.00021

[9] Xu Chen, Qiu Qiu, Changshan Li, and Kunqing Xie. 2022. GraphAD: a graph neural network for entity-wise multivariate time-series anomaly detection. In *Proceedings of the 45th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 2297–2302.

[10] Xinshi Chen, Yan Zhu, Haowen Xu, Mengyang Liu, Liang Xiong, Muhan Zhang, and Le Song. 2021. Efficient Dynamic Graph Representation Learning at Scale. *arXiv preprint arXiv:2112.07768* (2021).

[11] Weilin Cong, Jian Kang, Hanghang Tong, and Mehrdad Mahdavi. 2024. On the Generalization Capability of Temporal Graph Learning Algorithms: Theoretical Insights and a Simpler Method. *arXiv preprint arXiv:2402.16387* (2024).

[12] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. 2023. Do We Really Need Complicated Model Architectures For Temporal Networks?. In *ICLR*.

[13] Gangda Deng, Hongkuan Zhou, Hanqing Zeng, Yinglong Xia, Christopher Leung, Jianbo Li, Rajgopal Kannan, and Viktor Prasanna. 2024. TASER: Temporal Adaptive Sampling for Fast and Accurate Dynamic Graph Representation Learning. *arXiv preprint arXiv:2402.05396* (2024).

[14] Jianfei Gao and Bruno Ribeiro. 2022. On the equivalence between temporal and static equivariant graph representations. In *International Conference on Machine Learning*. PMLR, 7052–7076.

[15] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. 2024. ETC: Efficient Training of Temporal Graph Neural Networks over Large-scale Dynamic Graphs. *Proc. VLDB Endow.* 17, 5 (2024), 1060–1072. https://www.vldb.org/pvldb/vol17/p1060-gao.pdf

[16] Alessio Gravina and Davide Bacciu. 2024. Deep learning for dynamic graphs: models and benchmarks. *IEEE Transactions on Neural Networks and Learning Systems* (2024).

[17] Aaron Halfaker, Os Keyes, Daniel Kluver, Jacob Thebault-Spieker, Tien Nguyen, Kate Grandprey-Shores, Anuradha Uduwage, and Morten Warncke-Wang. 2015. User session identification based on strong regularities in inter-activity time. In *Proceedings of the 24th International Conference on World Wide Web*. 410–418.

[18] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[19] Qiang Huang, Jiawei Jiang, Xi Susie Rao, Ce Zhang, Zhichao Han, Zitao Zhang, Xin Wang, Yongjun He, Quanqing Xu, Yang Zhao, et al. 2023. BenchTemp: A General Benchmark for Evaluating Temporal Graph Neural Networks. *arXiv preprint arXiv:2308.16385* (2023).

[20] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2024. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems* 36 (2024).

[21] Wenjian Jiang, Xuhong Wang, Ping Cui, Bin Huang, Yupu Yang, and Qifu Fan. 2022. Adaptive Sampling Temporal Graph Network. In *2022 4th International Conference on Advances in Computer Technology, Information Science and Communications (CTISC)*. IEEE, 1–8.

[22] Ming Jin, Yuan-Fang Li, and Shirui Pan. 2022. Neural temporal walks: Motif-aware representation learning on continuous-time dynamic graphs. *Advances in Neural Information Processing Systems* 35 (2022), 19874–19886.

[23] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. 2019. Time2vec: Learning a vector representation of time. *arXiv preprint arXiv:1907.05321* (2019).

[24] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research* 21, 70 (2020), 1–73.

[25] Sumit Kumar, Yiming Gu, Jerrick Hoang, Galen Clark Haynes, and Micol Marchetti-Bowick. 2021. Interaction-Based Trajectory Prediction Over a Hybrid Traffic Graph. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 5530–5535. https://doi.org/10.1109/IROS51168.2021.9636143

[26] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 1269–1278.

[27] Jianing Li and Xin Yao. 2022. Corporate investment prediction using a weighted temporal graph neural network. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 12, 6 (2022), e1472.

[28] Mengzhang Li and Zhanxing Zhu. 2021. Spatial-temporal fusion graph neural networks for traffic flow forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 35. 4189–4196.

[29] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Orca: Scalable Temporal Graph Neural Network Training with Theoretical Guarantees. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[30] Yiming Li, Yanyan Shen, Lei Chen, and Mingxuan Yuan. 2023. Zebra: When Temporal Graph Neural Networks Meet Temporal Personalized PageRank. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1332–1345.

[31] Jie Liu, Jiamou Liu, Kaiqi Zhao, Yanni Tang, and Wu Chen. 2024. TP-GNN: Continuous Dynamic Graph Neural Network for Graph Classification. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 2848–2861.

[32] Yuhong Luo and Pan Li. 2022. Neighborhood-aware scalable temporal network representation learning. In *Learning on Graphs Conference*. PMLR, 1–1.

[33] Norbert Marwan, M Carmen Romano, Marco Thiel, and Jürgen Kurths. 2007. Recurrence plots for the analysis of complex systems. *Physics reports* 438, 5-6 (2007), 237–329.

[34] Shengjie Min, Zhan Gao, Jing Peng, Liang Wang, Ke Qin, and Bo Fang. 2021. Stgsn—a spatial–temporal graph neural network framework for time-evolving social networks. *Knowledge-Based Systems* 214 (2021), 106746.

[35] Tore Opsahl and Pietro Panzarasa. 2009. Clustering in weighted networks. *Social networks* 31, 2 (2009), 155–163.

[36] Santosh Pandey, Lingda Li, Adolfy Hoisie, Xiaoye S Li, and Hang Liu. 2020. C-SAW: A framework for graph sampling and random walk on GPUs. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–15.

[37] Pietro Panzarasa, Tore Opsahl, and Kathleen M Carley. 2009. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology* 60, 5 (2009), 911–932.

[38] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. 2020. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34. 5363–5370.

[39] Farimah Poursafaei, Shenyang Huang, Kellin Pelrine, and Reihaneh Rabbany. 2022. Towards better evaluation for dynamic link prediction. *Advances in Neural Information Processing Systems* 35 (2022), 32928–32941.

[40] Farimah Poursafaei and Reihaneh Rabbany. 2023. Exhaustive Evaluation of Dynamic Link Prediction. In *2023 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 1121–1130.

[41] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. In *ICML 2020 Workshop on Graph Representation Learning*.

[42] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*. 519–527.

[43] Ilya O Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, et al. 2021. Mlp-mixer: An all-mlp architecture for vision. *Advances in neural information processing systems* (2021), 24261–24272.

[44] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. Dyrep: Learning representations over dynamic graphs. In *International conference on learning representations*.

[45] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[46] Daixin Wang, Zhiqiang Zhang, Jun Zhou, Peng Cui, Jingli Fang, Quanhui Jia, Yanming Fang, and Yuan Qi. 2021. Temporal-aware graph neural network for

credit risk prediction. In *Proceedings of the 2021 SIAM International Conference on Data Mining (SDM)*. SIAM, 702–710.

[47] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, et al. 2021. Apan: Asynchronous propagation attention network for real-time temporal graph embedding. In *Proceedings of the 2021 international conference on management of data*. 2628–2638.

[48] Xiaoyang Wang, Yao Ma, Yiqi Wang, Wei Jin, Xin Wang, Jiliang Tang, Caiyan Jia, and Jian Yu. 2020. Traffic flow prediction via spatial temporal graph neural network. In *Proceedings of the web conference 2020*. 1082–1092.

[49] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive representation learning in temporal networks via causal anonymous walks. *arXiv preprint arXiv:2101.05974* (2021).

[50] Yufeng Wang and Charith Mendis. 2023. TGOpt: Redundancy-aware optimizations for temporal graph attention networks. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 354–368.

[51] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *ICLR*.

[52] Menglin Yang, Min Zhou, Marcus Kalander, Zengfeng Huang, and Irwin King. 2021. Discrete-time temporal network embedding via implicit hierarchical learning in hyperbolic space. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1975–1985.

[53] Haoteng Yin, Muhan Zhang, Yanbang Wang, Jianguo Wang, and Pan Li. 2022. Algorithm and system co-design for efficient subgraph-based graph representation learning. *Proc. VLDB Endow.* 15, 11 (jul 2022), 2788–2796. https://doi.org/10.14778/3551793.3551831

[54] Jiaxuan You, Tianyu Du, and Jure Leskovec. 2022. ROLAND: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD conference on knowledge discovery and data mining*. 2358–2366.

[55] Le Yu, Leilei Sun, Bowen Du, and Weifeng Lv. 2024. Towards better dynamic graph learning: New architecture and unified library. *Advances in Neural Information Processing Systems* 36 (2024).

[56] Xianlin Zeng, Yalong Jiang, Wenrui Ding, Hongguang Li, Yafeng Hao, and Zifeng Qiu. 2021. A hierarchical spatio-temporal graph convolutional neural network for anomaly detection in videos. *IEEE Transactions on Circuits and Systems for Video Technology* 33, 1 (2021), 200–212.

[57] Yuchen Zhong, Guangming Sheng, Tianzuo Qin, Minjie Wang, Quan Gan, and Chuan Wu. 2023. GNNFlow: A Distributed Framework for Continuous Temporal GNN Learning on Dynamic Graphs. *arXiv preprint arXiv:2311.17410* (2023).

[58] Huachi Zhou, Qiaoyu Tan, Xiao Huang, Kaixiong Zhou, and Xiaoling Wang. 2021. Temporal augmented graph neural networks for session-based recommendations. In *Proceedings of the 44th International ACM SIGIR conference on research and development in information retrieval*. 1798–1802.

[59] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs. *Proc. VLDB Endow.* 15, 8 (apr 2022), 1572–1580. https://doi.org/10.14778/3529337.3529342

[60] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. 2023. DistTGL: Distributed Memory-Based Temporal Graph Neural Network Training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.