ONLINE GRAPH NETS

Anonymous authors
Paper under double-blind review

ABSTRACT

Temporal graph neural networks (T-GNNs) for continuous-time dynamic graphs sequentially update node states and use temporal message passing to predict novel events. While node states rest in the memory, the message-passing operations must be computed on-demand for each prediction. In practice, these operations are the computational bottleneck of T-GNNs as they require topologically exploring large temporal graphs. To circumvent this caveat (i.e., avoid temporal message passing), we propose Online Graph Nets (OGNs). OGN maintains a summary of the temporal neighbors of each node in a latent variable and updates it as events unroll, in an online fashion. At prediction time, OGN simply combines node states and their latents to obtain node-level representations. Consequently, the memory cost of OGN is constant with respect to the number of previous events. Remarkably, OGN outperforms most existing T-GNNs on temporal link prediction benchmarks while running orders of magnitude faster. For instance, OGN performs slightly better than state-of-the-art T-GNNs on Reddit, with a speedup of up to two orders of magnitude. Also, since OGNs store all relevant information in node states and neighborhood variables, there is no need to sweep through the graph structure at inference time. This feature makes OGN especially well-suited for applications that require on-device predictions (e.g., on mobile phones).

1 INTRODUCTION

Temporal graph neural networks (T-GNNs) (Kazemi et al., 2020; Nguyen et al., 2018; Trivedi et al., 2019; Wang et al., 2021; Rossi et al., 2020; Xu et al., 2020) have become popular due to their ability to learn time-dependent vector representations of real-world dynamic graphs, like citation and social networks. We can roughly categorize T-GNNs into two classes. The first deals with discrete-time dynamic graphs (Liben-Nowell & Kleinberg, 2007; Pareja et al., 2020; Zhu et al., 2016; Ahmed & Chen, 2016), often represented as a sequence of graph snapshots. The second class handles continuous-time dynamic graphs (CTDGs), represented as a sequence of timestamped events. T-GNNs for CTDGs — hereafter referred to simply as T-GNNs — leverage continuous dynamics by combining several building blocks, such as attention mechanisms (Vaswani et al., 2017; Velickovic et al., 2018), time encoding schemes (Xu et al., 2019a; Kazemi et al., 2019), recurrent models (Cho et al., 2014; Hochreiter & Schmidhuber, 1997), and convolutional layers (Bruna et al., 2014; Kipf & Welling, 2017). Intuitively, these model components allow for capturing meaningful structural and temporal patterns. However, as T-GNNs increase in complexity, understanding the key factors behind their success is more challenging, and applying them to large datasets becomes prohibitive.

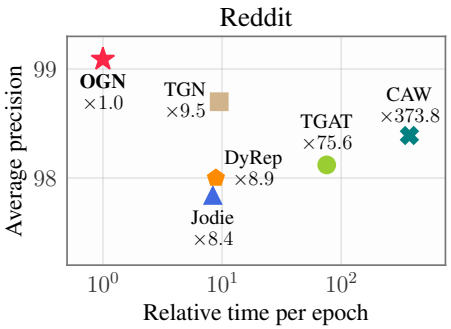


Figure 1: **OGN vs SOTA T-GNNs on Reddit (+500k interactions).** The horizontal axis shows the relative training time for each method as a multiple of OGN's running time. The vertical axis shows average precision. OGN clearly outperforms the SOTA but runs approximately 10 times faster than TGN (Rossi et al., 2020) and 374 times faster than CAW[1] (Wang et al., 2021).

[1]These results were obtained using batch-size equal to 200 (TGN and TGAT's default value) for all methods. Since CAW samples random walks in CPU, its running time might be significantly affected by the amount of RAM available (we used 40GB). For implementation details and results with batch size 32, see the Appendix.

In static settings, many works (e.g., Knyazev et al., 2019; Xu et al., 2019b; Garg et al., 2020; Loukas, 2020; Morris et al., 2019; Q. Li & Wu, 2018; Errica et al., 2020; Mesquita et al., 2020) have studied the power and limitations of GNNs. Meanwhile, simple models have also proved competitive, if not superior, to more complex GNNs (Wu et al., 2019; Huang et al., 2021; Chen et al., 2020). On the other hand, temporal settings remain much less explored from both theoretical and empirical perspectives. As a consequence, there are many open questions regarding *when* and *why* modern T-GNNs succeed. Additionally, it is unknown whether we can simplify T-GNNs without significantly compromising their high predictive performance.

Most T-GNNs follow a general framework (Section 2.2) that comprises up to three main steps for every prediction: (i) temporal sampling of nodes, (ii) temporal neighborhood aggregation, and (iii) recursive update of node states. We refer to steps (i) and (ii) jointly as temporal message passing due to its resemblance to message passing in traditional GNNs (Hamilton et al., 2017). One key issue in previous proposals is that the sampling phase needs to sweep through historical data (events in the past). In turn, every prediction depends on the number of previous events, which may considerably hurt time efficiency during training and testing (see Section 3.1). On top of that, the computational cost of neighborhood aggregations usually increases with the number of sampled nodes. This dependence on history length can be easily witnessed empirically. For instance, CAW (Wang et al., 2021) achieves impressive results on many benchmarks, however, it can take over 100 minutes per epoch in large datasets (+500K events) on a home computer with an NVIDIA GTX 1080 Ti 8Gb GPU. As many real-world networks can easily comprise billions of evolving links connecting hundreds of thousands of nodes, scalability is a crucial desideratum of learning models on temporal networks.

To mitigate these issues, we propose *Online Graph Nets* (OGN), a simple and fast approach for temporal graph learning. In OGN every node is represented as a combination of a state and a neighborhood summary. While previous works invest most of their computation in repeatedly sampling and aggregating neighbors, our neighborhood computation is embarrassingly simple and cheap: it is just a weighted average of all temporal neighbors states. To update the state of a node, we use a combination of its previous state, its neighborhood summary, and information about the novel event. In both computations we use the ordering of interactions between nodes as a proxy for time information, thus not considering any form of continuous time data. We also develop a scheme to efficiently and dynamically compute the node and neighborhood representations. With this, OGN does not need to store any information besides these two vectors per node, and the update can be performed in $O(1)$ time. Consequently, our method works as a fully online streaming algorithm without the need to store any previous interaction, which is especially useful for on-device predictions. Most importantly, OGN either outperforms or is competitive against state-of-the-art T-GNNs, while being much faster than previous methods (see Figure 1).

2 Background

2.1 Preliminaries

We consider continuous-time dynamic graphs (CTDGs) as sequences of timestamped events. These events are split into node and edge events. The former adds/deletes elements to/from the node set $\mathcal{V}$. The latter represent interactions between nodes in $\mathcal{V}$. Each edge event is a tuple $(u, v, \boldsymbol{e}, t)$ where u and v are nodes, $\boldsymbol{e} \in \mathbb{R}^\ell$ is a vector of edge features, and $t \in \mathbb{N}^+$ is a timestamp. In practice, adding/deleting nodes only expands/retracts the set of possible edge interactions. Therefore, without loss of generality, we represent a CTDG as a sequence of edge events $\mathcal{E} = \left((u_n, v_n, \boldsymbol{e}^{(n)}, t^{(n)})\right)_{n=1}^{N}$.

2.2 Temporal Graph Neural Networks

Temporal graph neural networks (T-GNNs) usually follow a three step pipeline to extract node representations: temporal sampling, neighborhood aggregation, and recursive state update. We refer to the two former jointly as temporal message passing. Figure 2 provides an overview of this process.

Temporal message passing. The first step in temporal message passing is temporal sampling, which involves iteratively selecting temporal neighbors of a node u to build a graph. Then, a neighborhood aggregation step takes the sampled graph and aggregates neighbors iteratively, along with their states/features and information about connecting edges, to compute an output graph. For an event at time t', the time embedding wrt a reference time t is computed through a time encoder $\Phi(t, t')$. Let $\mathcal{G}_u^{(0)}$ be a singleton graph, containing only node u annotated with its node state (and/or

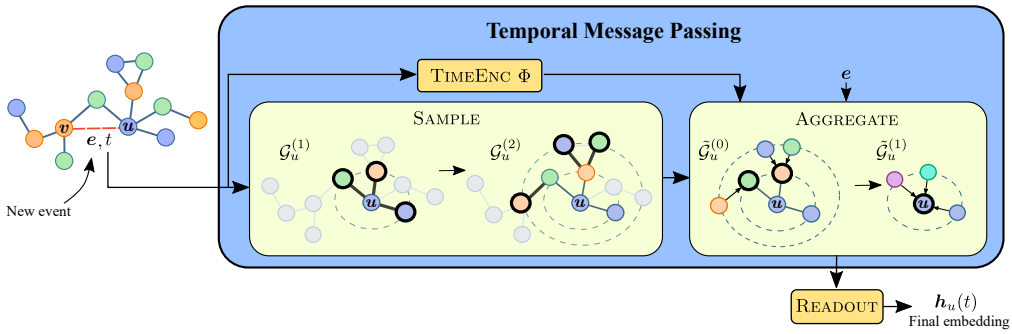


Figure 2: **T-GNNs: general framework to compute temporal node representations.** For clarity, we illustrate an example for node u with $L = 2$. For an event with node v at time t, T-GNNs (i) iteratively sample a L-hop temporal neighborhood of u (SAMPLE), (ii) encode the continuous timestamps (TIMEENC), (iii) iteratively aggregate the neighborhood information (AGGREGATE), and (iv) compute the final embedding of the resulting neighborhood information of node u (READOUT).

features) and no edges. In general, the updates that a T-GNN with L layers does to extract an embedding for node u go as follows:

$$\mathcal{G}_u^{(l)} \leftarrow \textsc{Sample}(\mathcal{G}_u^{(l-1)}), \quad \forall l = 1, \dots, L \tag{1}$$

$$\tilde{\mathcal{G}}_u^{(0)} \leftarrow \mathcal{G}_u^{(L)} \tag{2}$$

$$\tilde{\mathcal{G}}_u^{(l)} \leftarrow \textsc{Aggregate}(\tilde{\mathcal{G}}_u^{(l-1)}, \Phi), \quad \forall l = 1, \dots, L \tag{3}$$

where $\mathcal{G}_u^{(l)}$ is the graph obtained after l layers of sampling. When sampling a new node j from a node i, $\textsc{Sample}(\cdot)$ adds its node states $\boldsymbol{s}_j$, edge features $\boldsymbol{e}_{ij}$ and the timestamp of the event t_{ij} to the resulting graph $\mathcal{G}_u^{(l)}$. Similarly, $\tilde{\mathcal{G}}_u^{(l)}$ denotes the graph after l steps of aggregation, i.e., $\textsc{Aggregate}(\cdot)$, which assembles the sampled node states, edge features, and encoded timestamps $\Phi(t_{ij}, t)$ into a new representation. Since aggregation reduces the number of nodes, the graphs follow $\tilde{\mathcal{G}}_u^{(l)} \subseteq \tilde{\mathcal{G}}_u^{(l-1)} \dots \subseteq \tilde{\mathcal{G}}_u^{(0)}$, for all $l = 1, \dots, L$.

After L layers of aggregation, T-GNNs output a graph ($\tilde{\mathcal{G}}_u^{(L)}$) that contains information about the temporal neighborhood of node u. We embed $\tilde{\mathcal{G}}_u^{(L)}$ into a node vector $\boldsymbol{h}_u(t)$ using a readout function, i.e., $\boldsymbol{h}_u(t) = \textsc{Readout}(\tilde{\mathcal{G}}_u^{(L)})$. In methods such as TGN and TGAT where the output of the aggregation is a single node, the readout function simply returns its node embedding.

Recursive state update. Once the output embedding is computed, T-GNNs update the state vector for node u as follows:

$$\boldsymbol{s}_u \leftarrow \textsc{Update}(\boldsymbol{s}_u, \boldsymbol{s}_v, \boldsymbol{e}_{uv}, t_{uv}). \tag{4}$$

To better understand how the framework captures continuous time T-GNN models, take the case of Temporal Graph Networks (TGN) (Rossi et al., 2020). In the l-th SAMPLE function, TGN selects the last nodes that have interacted with the leaves of the graph $\mathcal{G}_u^{(l-1)}$. Subsequently, TGN encodes the timestamps using Time2Vec (Kazemi et al., 2019) and applies temporal graph attention (Xu et al., 2020) as the AGGREGATE function. The last aggregation layer outputs a single node, and READOUT returns its feature vector, which is used as the node's representation for inference. Finally, the node's state is updated by using its previous state, the interacting node's state, the edge features, and the time embedding.

3 Online Graph Nets

A core idea behind T-GNNs is to maintain a state $\boldsymbol{s}_u$ for each node u, updating it whenever an event involving u (or its temporal neighbors) takes place. These updates require probing temporal and topological information to aggregate states from (possibly multi-hop) neighbors. Nonetheless, this aggregation step is the computational bottleneck of T-GNNs. To address this limitation, we propose summarizing each node's neighborhood into an auxiliary variable, which is incrementally updated as

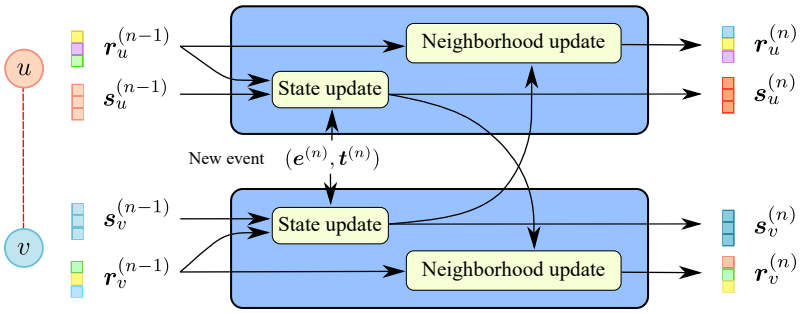


Figure 3: **Online Graph Nets (OGN).** OGN maintains a state vector and a neighborhood summary for each node. For a node u, to predict for a new event with node v, OGN updates the state vector of u using the neighborhood summary $r_u^{(n-1)}$ and the information of the event ($\boldsymbol{e}^{(n)}$ and $\boldsymbol{t}^{(n)}$), then updates its neighborhood information using the updated state vector of v. OGN performs this update for both nodes, and uses the resulting state vectors for inference.

events unfold. Combining this idea with minimal design choices, we develop OGN (*Online Graph Nets*) — a fast and simple model for representation learning on dynamic graphs.

Basic notation. We assign a neighborhood variable $\boldsymbol{r}_u \in \mathbb{R}^d$ and a state variable $\boldsymbol{s}_u \in \mathbb{R}^d$ to each node u. We annotate these variables with superscripts to account for their evolution in time. In our formulation, we replace timestamps by an enumeration of events over time, which is equivalent to counting the events up to (and including) each interaction. For instance, $\boldsymbol{s}_u^{(n)}$ denotes the state vector for u after the n-th edge event. If the n-th added edge does not have an endpoint in u, then we set $\boldsymbol{s}_u^{(n)} = \boldsymbol{s}_u^{(n-1)}$ and $\boldsymbol{r}_u^{(n)} = \boldsymbol{r}_u^{(n-1)}$ by default. Also, we denote by $\mathcal{N}_u^{(n)}$ the set of temporal neighbors of node u prior to the n-th event in history.

Expected neighborhood state. We define the neighborhood state $\boldsymbol{r}_u^{(n)}$ as a weighted average of the states of all nodes that u interacted with, *exactly at the time of those interactions*. Thus, if node u had two distinct interactions with node i, then $\boldsymbol{r}_u^{(n)}$ considers two states of i, which need not be identical. To favor recent neighbors, we make the log-weight for $i \in \mathcal{N}_u^{(n)}$ decay linearly with the number of events $(n - m_i)$ since the interaction between i and u, which is the m_i-th event in the history. More specifically, the neighborhood state $\boldsymbol{r}_u^{(n)}$ is given by

$$\boldsymbol{r}_u^{(n)} = \sum_{i \in \mathcal{N}_u^{(n)}} w_i \boldsymbol{s}_i^{(m_i)}, \quad \text{with} \quad w_i \coloneqq \frac{\exp\left(-\alpha\left(n - m_i\right)\right)}{\sum_j \exp\left(-\alpha\left(n - m_j\right)\right)}, \tag{5}$$

where α controls how fast the importance of temporal neighbors decays. Note that the weight vector $\boldsymbol{w} = (w_1, w_2, \ldots, w_{n-m_i})$ is the output of a temperature-scaled softmax. As consequence, $\boldsymbol{w}$ defines a categorical distribution over the neighborhood $\mathcal{N}_u^{(n)}$, and Equation 5 can be seen as the expected state of the neighbors of u. We provide more details in Appendix F.

Online computation of neighborhood states. Naively updating $\boldsymbol{r}_u^{(n)}$ requires a sweep over all previous neighbors every time a new edge event with endpoint in u occurs. Additionally, we would need to store the complete history of states for each node. Summing up, after the n-th event we would have an overhead of $O(n)$ time and memory for each novel update. To alleviate this cost, we propose updating $\boldsymbol{r}_u^{(n)}$ *online* as events unroll. Assume that the n-th event connects nodes u and v, and let m be the number of events since u last interacted with any node. We recursively compute

$$\boldsymbol{a}_u^{(n)} = \boldsymbol{s}_v^{(n)} + \exp(-\alpha m) \cdot \boldsymbol{a}_u^{(n-1)}, \tag{6}$$

$$b_u^{(n)} = 1 + \exp(-\alpha m) \cdot b_u^{(n-1)}, \tag{7}$$

with $\boldsymbol{a}_u^{(0)} = \mathbf{0}$ and $b_u^{(0)} = 0$. This recursion is particularly useful since $\boldsymbol{r}_u^{(n)} = \boldsymbol{a}_u^{(n)} / b_u^{(n)}$ — see Proposition 1, with a simple proof in Appendix G. Thus, to implement our method we simply store and update $\boldsymbol{a}_u^{(n)}$ and $b_u^{(n)}$. This scheme drops both time and memory complexity to $O(1)$ per update.

Proposition 1. *For any node $u \in \mathcal{V}$, let $\boldsymbol{a}_u^{(n)}$ and $b_u^{(n)}$ be defined according to Equation 6 and Equation 7, respectively, with $\boldsymbol{a}_u^{(0)} = \mathbf{0}$ and $b_u^{(0)} = 0$. Then, for all $n \in \mathbb{N}$, it holds that:*

$$\boldsymbol{r}_u^{(n)} = \frac{\boldsymbol{a}_u^{(n)}}{b_u^{(n)}}.$$

Remark 1 (Other weighting schemes)**.** *The updates with exponential decay in Equation 6 and Equation 7 can be adapted to arbitrary weighting schemes for $\boldsymbol{r}_u^{(n)}$ without breaking the identity in Proposition 1, that allows efficient updates neighborhood states. We can do so by replacing the unnormalized weight $\exp(-\alpha m)$ with another expression. The requirement for Proposition 1 to remain valid is that adding a novel event must preserve the unnormalized weights of previous events.*

Updating node states. We now describe how we update the state $\boldsymbol{s}_u^{(n)}$ when a new event involving u occurs, with edge features $\boldsymbol{e}^{(n)} \in \mathbb{R}^\ell$. We first compute an intermediate state $\bar{\boldsymbol{s}}_u^{(n)}$ by running the edge features, the previous node state $\boldsymbol{s}_u^{(n-1)}$, and the time embedding $\boldsymbol{t}^{(n)}$ of n through a linear layer followed by a non-linearity. Then, we feed $[\bar{\boldsymbol{s}}_u^{(n)} \| \boldsymbol{r}_u^{(n-1)}]$ to another single-layer net to obtain the updated state $\boldsymbol{s}_u^{(n)}$, where $\|$ denotes concatenation. The resulting node state update is:

$$\bar{\boldsymbol{s}}_u^{(n)} = \phi\Big(\boldsymbol{W}_1\big[\, \boldsymbol{e}^{(n)} \;\|\; \boldsymbol{s}_u^{(n-1)} \,\|\, \boldsymbol{t}^{(n)}\big]\Big), \tag{8}$$

$$\boldsymbol{s}_u^{(n)} = \phi\Big(\boldsymbol{W}_2\big[\bar{\boldsymbol{s}}_u^{(n)} \,\|\, \boldsymbol{r}_u^{(n-1)}\big]\Big), \tag{9}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d\times(\ell+2d)}$ and $\boldsymbol{W}_2 \in \mathbb{R}^{d\times 2d}$ are parameters, and ϕ is an element-wise non-linearity. OGN performs this update for both nodes u and v, and uses the resulting node states $\boldsymbol{s}_u^{(n)}$ and $\boldsymbol{s}_v^{(n)}$ as node representations for prediction purposes. A diagram for OGN is presented in Figure 3.

3.1 Complexity analysis

A key aspect of our formulation is that it incurs the same amount of computation for each edge added to the graph, no matter how many events have happened thus far. Notably, the computation time for each edge does not depend on any particular graph property, nor on the number of edges previously added or its distribution, nor in the degree of nodes, etc. More specifically, in terms of the graph size, our computational cost per addition is $O(1)$. Moreover, whenever an edge is added to the graph and values $\boldsymbol{s}_u$, $\boldsymbol{a}_u$ and b_u are updated for the nodes incident to that edge, we no longer need that edge information in any way for future computations and we can safely drop it. In this sense, our method is a fully *online streaming method*, i.e., its computation time does not increase with the number of previous events (graph size).

This clearly differs from previous methods that need access to some form of memory about the previous events, and whose time complexity for processing new edges or making predictions also depends on the total number of previous events. For concreteness, assume a dynamic graph with E edges added so far, and with d as its maximum degree (maximum amount of events for a single node). For instance, TGAT's implementation requires a binary search over the history of previous events for each node. This implies that every node should store information of all its previous events, thus having a $\Omega(E)$ requirement for total memory and a $O(\log d)$ time overhead to process each new event (Xu et al., 2020). A naive implementation of the path sampling in CAW would also need $\Omega(E)$ requirement for memory (potentially accessing any edge in the graph) and $O(d \times L)$ when sampling paths of length L. In the CAW paper (Wang et al., 2021), the authors state that sampling can be done with constant time and memory overhead, nevertheless our experiments using their own implementation exhibit the bigger running times of all methods we tested (Figure 4).

3.2 On-edge prediction

Dynamic graphs in real-life applications, such as social networks, may capture millions of new events every day. Thus, methods that require storing and updating the history of events for making predictions are not suitable for real-time applications. The updating must be performed in a centralized system to keep the last version of the complete graph, encompassing all the history, which becomes impractical when the graph size scales hugely. State-of-the-art T-GNNs, such as TGN, TGAT, and CAW, fall into this category. These methods rely on neighborhood sampling, thus requiring to at least update the (often multi-hop) neighborhood of the involving nodes as events unroll.

In contrast to the latest T-GNNs, predictions for OGN only depend on the nodes involved in the new event. Since OGN only relies on local information, it needs not an updated version of the graph nor even the graph structure to make a prediction. This property enables on-edge prediction. To illustrate, if nodes in the graph represent users in a social network and edges represent interactions between them, the predictions and the updates can be performed in each user's device by considering its state and neighborhood vectors and the ones from its interacting user, without requiring large computations and information exchange with a centralized system. Changing the predictions and updates from centralized to on-edge makes the models suitable for the aforementioned applications.

4 Related Works

Models for representation learning on dynamic graphs can be roughly categorized w.r.t. their ability to deal with discrete-time or continuous-time graphs. Discrete-time methods operate on snapshots of dynamic graphs sampled at regular intervals. Although these models come in many flavors, some strategies include snapshot aggregation schemes (Liben-Nowell & Kleinberg, 2007; Ahmed & Chen, 2016), time-dependent random walks (Mahdavi et al., 2018; Yu et al., 2018; Winter et al., 2018), and sequence models (e.g., recurrent nets or transformers) combined with GNNs (Seo et al., 2018; Pareja et al., 2020; Goyal et al., 2020; Sankar et al., 2020). For a review of these strategies, we refer the reader to the recent survey by Kazemi et al. (2020).

Our work mainly relates to methods for representation learning on continuous-time graphs. Notably, the majority of these methods employ deep learning ingredients. A particularly widespread design consists of applying sequence models to update node representations as new (edge) events occur. JODIE (Kumar et al., 2019) applies two mutually-recursive RNNs — one for each node (source and target) involved in an edge event — followed by a time-dependent embedding projection. DyRep (Trivedi et al., 2019) combines RNNs with a temporally attentive module, leveraging 2-hop neighborhood information for updating node representations. TGAT (Xu et al., 2020) samples a set of temporal neighbors and applies graph attention networks (Velickovic et al., 2018) over them. In addition, TGAT proposes using random Fourier features to encode timestamps. TGN (Rossi et al., 2020) introduces a general framework that includes some prior models (e.g., JODIE and TGAT) as particular cases. Another line of work applies random walks with transition probabilities conditioned on timestamps (Nguyen et al., 2018; Bastas et al., 2019; Wang et al., 2021). Causal anonymous walks network (CAW) (Wang et al., 2021) applies a recurrent model to multiple anonymized random walks, and then aggregates the latent states using attention or mean operations. Different from prior works, our proposal leverages time information by simply counting edge events. Also, we focus on obtaining a simple architectural design that naturally leads to fast prediction and training.

OGN applies a weighted average (Equation 5) over temporal neighbors to prioritize more recent interactions. These weights define a probability distribution over temporal neighbors. This reflects the notion that recent temporal neighbors deserve higher importance, which is a widespread idea in temporal graph learning (Sharan & Neville, 2008; Ahmed & Chen, 2016; Wang et al., 2021).

5 Experiments

We evaluate OGN in two tasks: temporal link prediction and node classification. We run all experiments using PyTorch (Paszke et al., 2017) and our code is available as additional material.

5.1 Temporal link prediction

Datasets. We assess the performance of OGN on four commonly used link prediction benchmarks: Reddit, Wikipedia, MOOC and Twitter. These datasets are attributed, i.e., they contain feature vectors for their events. The Twitter dataset is not publicly available, but we follow instructions from Rossi et al. (2020) to create a version of the dataset. Node features are absent in all datasets, thus we follow previous work (Xu et al., 2020; Rossi et al., 2020) and set them to zero. We provide datasets statistics in Appendix B. Additionally, we report results for non-attributed datasets in Appendix E.

Baselines. We compare OGN against five state-of-the-art temporal models: Jodie (Kumar et al., 2019), DyRep (Trivedi et al., 2019), TGAT (Xu et al., 2020), TGN (Rossi et al., 2020), and CAW (Wang et al., 2021). We also show results for two static graph methods, GAT (Velickovic et al., 2018) and GraphSage (Hamilton et al., 2017), which do not use temporal information. Most of the results for these baselines are available in previous works (Xu et al., 2020; Rossi et al., 2020). However, we re-run experiments for Jodie and DyRep due to conflicting numbers in different papers. We noticed

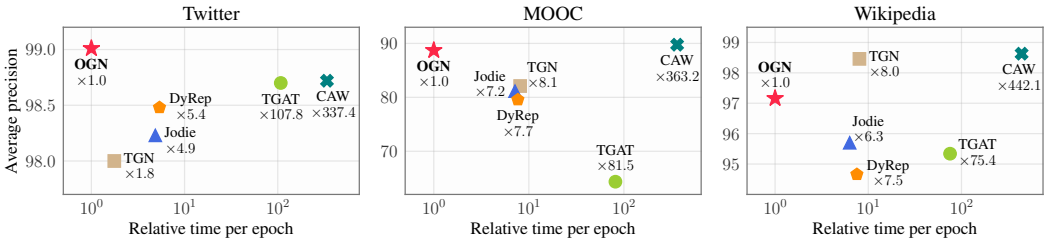


Figure 4: **Time/epoch vs average precision.** We normalize the running time per epoch (training+validation) of all models with respect to the execution time of OGN. In all cases, OGN nearly matches or surpasses the SOTA. Also, OGN is the fastest method overall. For instance, OGN is two orders of magnitude faster than CAW and at least one order of magnitude faster than TGAT.

Table 1: **Results in average precision (AP) on link prediction for datasets w/ edge features.** In all cases, OGN is either the best method or is closely behind. * denotes the original standard deviation 0.04 rounded to the first decimal. Boldface indicates the best one or two average results (two if they are less than a standard deviation away from each other), and underline the second best.

Model	Reddit		Wikipedia		MOOC		Twitter	
	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive
GAT	97.33±0.2	95.37±0.3	94.73±0.2	91.27±0.4	-	-	-	-
GraphSAGE	97.65±0.2	96.27±0.2	93.56±0.3	91.09±0.3	-	-	-	-
Jodie	97.84±0.3	93.97±1.3	95.70±0.2	93.61±0.2	81.16±1.0	78.77±1.6	98.23±0.1	96.06±0.1
DyRep	98.00±0.1	95.18±0.2	94.66±0.1	91.91±0.2	79.57±1.5	79.37±0.7	98.48±0.1	96.33±0.2
TGAT	98.12±0.2	96.62±0.3	95.34±0.1	93.99±0.3	64.36±3.3	61.74±3.2	98.70±0.1	96.33±0.1
TGN	98.70±0.1	97.55±0.1	98.46±0.1	97.81±0.1	82.10±0.4	77.70±0.3	98.00±0.1	95.76±0.1
CAW	98.39±0.1	97.81±0.1	**98.63±0.1**	**98.52±0.1**	**89.76±0.4**	**89.72±0.4**	98.72±0.1	**98.54±0.3**
OGN (ours)	**99.09±0.0***	**98.66±0.1**	97.16±0.2	**98.41±0.2**	**88.71±1.3**	85.65±1.5	**99.01±0.0***	**98.46±0.1**

an incorrect implementation of attention in the released code of CAW, thus we modify it and report the corrected numbers. We provide a complete description of this change in Appendix A.

Experimental setup. The goal in link prediction is to classify whether an interaction between two nodes happens at a given time. Since our datasets only contain positive observations, we follow previous works (Xu et al., 2020; Rossi et al., 2020) and create negative links artificially. For every edge event, we create a false event at the same time by re-assigning one of the edge endpoints to a random node. Following Xu et al. (2020) and Rossi et al. (2020), we consider a 70%-15%-15% (train-val-test) split. We evaluate all models in both transductive and inductive settings. In the transductive setting, we predict interactions involving nodes seen during training. In the inductive setting, we evaluate the models on nodes never observed before. We use average precision (AP) as performance metric and repeat each experiment for ten independent runs.

Results. Table 1 presents the results for link prediction task in the transductive and inductive settings. The results show that OGN achieves comparable performance to state-of-the-art methods. Notably, it obtains the highest AP on Reddit (transductive and inductive) and Twitter (transductive). For completeness, Appendix E reinforces our findings with results using an additional negative sampling procedure.

Table 2: **Inference time (seconds) in the transductive setting.** Similarly to training time (Figure 4), test time for OGN is much faster than TGAT, TGN, and CAW.

Model	Reddit	Wikipedia	MOOC	Twitter
TGAT	420	687	246	1,064
TGN	132	182	88	98
CAW	2,218	2,068	1,337	1,494
OGN	**35**	**5**	**12**	**24**

Figure 4 shows the performance and time per epoch for different T-GNNs. OGN is competitive with state-of-the-art methods, while being orders of magnitude faster than some of the methods. In particular, OGN is always two orders of magnitude faster than CAW, which is the best performing method in Wikipedia and MOOC. Further, OGN is the fastest among all methods. We observe similar results when measuring inference/test time (Table 2). We note however that CAW is sensible to CPU capabilities and amount of memory available, which can be mainly attributed to its inherent sampling procedure. Thus, we also report results for a different value of batch size in Appendix E.

5.2 NODE CLASSIFICATION

Datasets. We evaluate OGN on two node classification benchmarks: Reddit and Wikipedia. Reddit contains labeled links that indicate whether a user will be banned from a subreddit. Only 366 out of $672,447$ interactions result in a ban. Wikipedia contains labeled events that represents whether a user will be kept from editing a Wikipedia page. Again, only a small fraction (217 out of $157,474$) of edits lead to bans on Wikipedia (Kumar et al., 2019).

Baselines. We compare OGN against four temporal models: Jodie, DyRep, TGAT and TGN. We also evaluate two static methods: GAT and GraphSage. Most of the results for our baselines are available in the literature (Xu et al., 2020; Rossi et al., 2020).

Table 3: **Results for node classification (AUC)**. OGN is the best method on Wikipedia and second best on Reddit.

Model	Reddit	Wikipedia
GAT	$64.52_{\pm 0.5}$	$82.34_{\pm 0.8}$
GraphSage	$61.24_{\pm 0.6}$	$82.42_{\pm 0.7}$
Jodie	$61.83_{\pm 2.7}$	$84.84_{\pm 1.2}$
DyRep	$62.91_{\pm 2.4}$	$84.59_{\pm 2.2}$
TGAT	$65.56_{\pm 0.7}$	$83.69_{\pm 0.7}$
TGN	$\mathbf{67.06_{\pm 0.9}}$	$\underline{87.81_{\pm 0.3}}$
OGN	$\underline{65.59_{\pm 1.0}}$	$\mathbf{88.36_{\pm 0.4}}$

Experimental setup. Due to the imbalance between positive and negative examples in the dataset, we measure performance in terms of AUC (area under the receiver operating characteristic curve). We report average and standard deviation for ten repetitions of the experiments.

Results. Table 3 presents the results for the node classification task. For Wikipedia, OGN presents the best performance. For Reddit, OGN is the second best method in average AUC.

6 ABLATION STUDIES

The importance of fine-grained time information. We design simple experiments to challenge the need for fine-grained time information (timestamps), and use TGAT, TGN and CAW as running examples. Intuitively, this feature is essential to capture the dynamics of real-world applications (e.g., social networks), in which events naturally occur in continuous time. To challenge this intuition, we artificially discretize the time information and re-evaluate these methods. We do so by setting the gap between successive events to a fixed value $\Delta = 0.1$ in training and testing. We refer to this approach as $\mathcal{U}$-TIME.

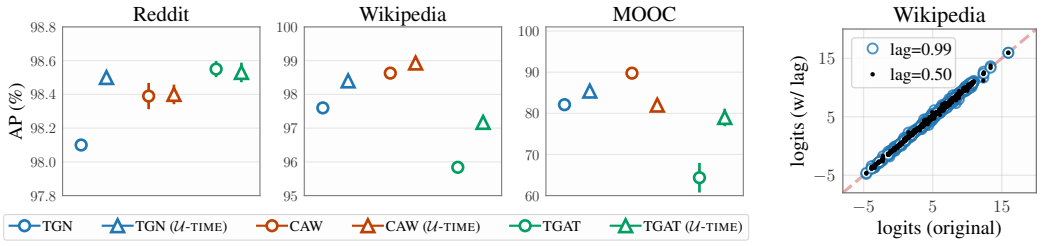
Figure 5a reports the performance of TGN, CAW, and TGAT with and without the discretization approach. Surprisingly, we find that continuous-time information generally does not improve and sometimes even hurts the performance of T-GNNs. For instance, $\mathcal{U}$-TIME leads to an increase of $\approx 3\%$ and 14% in AP for TGN and TGAT on the MOOC dataset, respectively. For all other datasets and methods, we only observe small fluctuations in performance, except for CAW on MOOC.

A possible explanation for this phenomenon is that T-GNNs are insensitive to the specific values of the timestamp, leveraging the ordering instead. To test this hypothesis, we take TGN (with full-fledged timestamps) and evaluate their predictions when each timestamp of the test set is shifted by a relative lag to approach the timestamp of the subsequent event. For example, with a relative lag of 0.5, we shift an event with timestamp 20000 to 15000 if the previous event happened at timestamp 10000. Note that this procedure preserves the original ordering of events. Figure 5b compares the logits of TGN with relative lags $\{0.5, 0.99\}$ on Wikipedia. Notably, TGN produces virtually the same predictions regardless of the amount of lag we apply. We describe further ablation studies regarding time information and the aggregation module in Appendix D.

Time information. To evaluate the effect of discretizing timestamps in OGN, we consider two alternatives: (i) using the original timestamps, and (ii) removing the vector representation of the time. Results in Table 4 show that using discretized time information generally leads to higher AP values. The exception is Reddit, in which using the original timestamps yields slightly better results.

Table 4: **Ablation study for OGN**. Neighborhood state and edge information play a crucial role on the model's performance. Discretized timestamps generally lead to better performance.

Model	Reddit	Wikipedia	MOOC
Original time	$\mathbf{99.13_{\pm 0.02}}$	$96.45_{\pm 0.97}$	$81.06_{\pm 2.82}$
No time	$98.96_{\pm 0.03}$	$95.63_{\pm 0.29}$	$86.33_{\pm 3.12}$
No edge	$96.11_{\pm 1.02}$	$96.57_{\pm 0.16}$	$69.87_{\pm 6.74}$
No neigh states	$97.00_{\pm 0.10}$	$89.33_{\pm 0.87}$	$83.40_{\pm 0.29}$
OGN	$\mathbf{99.09_{\pm 0.04}}$	$\mathbf{97.16_{\pm 0.21}}$	$\mathbf{88.71_{\pm 1.34}}$



(a) (b)

Figure 5: (**a**) **Effect of using regularly spaced timestamps ($\mathcal{U}$-TIME) on TGAT, TGN and CAW.** In general, using $\mathcal{U}$-TIME ($\triangle$ markers) does not harm performance. In fact, TGN and TGAT benefit from time discretization. Also, the models show smaller standard deviations overall. (**b**) **Predictions of TGN with shifted timestamps on test data for Wikipedia.** Even after applying a 0.99 relative lag, the logits remain rather similar to those from the original no-shifted timestamps (red line).

Similarly, using the discretized time also outperforms OGN with no time embeddings. These results highlight that the only information needed from the timestamps is the sequence of events.

Edge information. Since most datasets do not have node features, the edge features are the only source of information besides timestamps. To study their importance to our method, we remove them from the dataset by setting their values to zero. Results in Table 4 show that the performance deteriorates across all datasets when we remove the edge features, which demonstrates their importance for accurate predictions. The drop in performance without edge features is especially noticeable for the MOOC dataset, where the performance drop amounts to 18% when we remove edge features.

Neighborhood information. One of the key insights of our method consists of using a state vector $\boldsymbol{r}_u$ to summarize the neighborhood information for the nodes in the dataset. To study the impact of this component, we remove $\boldsymbol{r}_u$ from our model and only consider the node state and the time embedding to update the node states. The results in Table 4 show a significant decrease in performance when the neighborhood state is removed. These results indicate that neighborhood states are crucial components of our method.

Limitations and future works. As noted previously, edge features play an important role in the node state updates. Since node features are absent in the datasets, the performance of OGN mainly relies on edge information. Therefore, its performance drops when learning on non-attributed temporal graphs, which suggests that OGN does not fully exploit the structural information of the graph, given its simplicity. We report results on benchmarks with non-attributed events in Appendix E.

Although we observe that OGN's exponentially linear decay is a positive inductive bias for current benchmarks, we believe other weighting schemes might be useful for specific applications. As discussed in Remark 1, OGN can incorporate hand-crafted weight functions easily. However, using adaptive weights (e.g., with GRUs) could be useful when there is little information about the task at hand. We leave this direction for future investigation.

7 Conclusions

This work proposes OGN, a fast approach for representation learning on continuous-time dynamic graphs. Notably, OGN does not rely on the expensive message-passing step that T-GNNs like TGN, TGAT and CAW do. Instead, OGN relies on latent neighborhood variables that are updated in a streaming fashion, like node states are. At prediction time, we simply project node states and latents through feedforward mechanisms to obtain accurate predictions. As a consequence, OGN is often orders of magnitude faster than the remaining methods. OGN also surpasses or closely matches the SOTA in many large-scale benchmarks.

An attractive feature is that OGN is an online streaming method, i.e.: (i) updating the model does not require access to previous events; (ii) the computational cost of OGN does not increase with history length. Since OGN does not use neighborhood sampling, its predictions are extremely scalable and suitable for use on edge devices.

ETHICS AND REPRODUCIBILITY

In this work we follow the ICLR Code of Ethics. Given the experimental nature of this work, we have done our best effort to ensure our results are reproducible. In particular, we provide a detailed account of the training procedure and hyperparameters in the appendices. We also include the code for our experiments as supplementary material. After the discussion period, we will also upload this code to a github repository. Our experiments primarily focus on publicly available datasets. The exception is the Twitter dataset, for which we outline the necessary processing steps in the appendix.

REFERENCES

N. M. Ahmed and L. Chen. An efficient algorithm for link prediction in temporal uncertain social networks. *Information Sciences*, 331:120–136, 2016.

N. Bastas, T. Semertzidis, A. Axenopoulos, and P. Daras. evolve2vec: Learning network representations using temporal unfolding. In *International Conference on MultiMedia Modeling (MMM)*, 2019.

J. Bruna, W. Zaremba, A. Szlam, and Y. Lecun. Spectral networks and locally connected networks on graphs. In *International Conference on Learning Representations (ICLR)*, 2014.

M. Chen, Z. Wei, Z. Huang, B. Ding, and Y. Li. Simple and deep graph convolutional networks. In *International Conference on Machine Learning (ICML)*, 2020.

K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.

F. Errica, M. Podda, D. Bacciu, and A. Micheli. A fair comparison of graph neural networks for graph classification. In *International Conference on Learning Representations (ICLR)*, 2020.

V. Garg, S. Jegelka, and T. Jaakkola. Generalization and representational limits of graph neural networks. In *International Conference on Machine Learning (ICML)*, 2020.

P. Goyal, S. R. Chhetri, and A. Canedo. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems*, 187, 2020.

W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Q. Huang, H. He, A. Singh, S. Lim, and A. Benson. Combining label propagation and simple models out-performs graph neural networks. In *International Conference on Learning Representations (ICLR)*, 2021.

S. Kazemi, R. Goel, K. Jain, I. Kobyzev, A. Sethi, P. Forsyth, and P. Poupart. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research*, 21(70):1–73, 2020.

S. M. Kazemi, R. Goel, S. Eghbali, J. Ramanan, J. Sahota, S. Thakur, S. Wu, C. Smyth, P. Poupart, and M. Brubaker. Time2vec: Learning a vector representation of time. *ArXiv: 1907.05321*, 2019.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

B. Knyazev, G. W. Taylor, and M. Amer. Understanding attention and generalization in graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

S. Kumar, X. Zhang, and J. Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *International Conference on Knowledge Discovery & Data Mining (KDD)*, 2019.

D. Liben-Nowell and J. Kleinberg. The link prediction problem for social networks. *Journal of the American Society for Information Science and Technology*, 58(7):1019–1031, 2007.

A. Loukas. What graph neural networks cannot learn: depth vs width. In *International Conference on Learning Representations (ICLR)*, 2020.

S. Mahdavi, S. Khoshraftar, and A. An. dynnode2vec: Scalable dynamic network embedding. In *International Conference on Big Data*, 2018.

D. Mesquita, A. H. Souza, and S. Kaski. Rethinking pooling in graph neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2019.

H. Nguyen, J. B. Lee, R. A. Rossi, N. K. Ahmed, E. Koh, and S. Kim. Continuous-time dynamic network embeddings. In *The Web Conference (WWW)*, 2018.

A. Pareja, G. Domeniconi, J. Chen, T. Ma, H. Kanezashi T. Suzumura, T. Kaler, T. B. Schardl, and C. E. Leiserson. EvolveGCN: Evolving graph convolutional networks for dynamic graphs. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In *Advances in Neural Information Processing Systems (NeurIPS - Workshop)*, 2017.

Z. Han Q. Li and X. Wu. Deeper insights into graph convolutional networks for semi-supervised learning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 2018.

E. Rossi, B. Chamberlain, F. Frasca, D. Eynard, F. Monti, and M. Bronstein. Temporal graph networks for deep learning on dynamic graphs. *ArXiv: 2006.10637*, 2020.

A. Sankar, Y. Wu, L. Gou, W. Zhang, and H. Yang. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *International Conference on Web Search and Data Mining (WSDM)*, 2020.

Y. Seo, M. Defferrard, P. Vandergheynst, and X. Bresson. Structured sequence modeling with graph convolutional recurrent networks. In *International Conference on Neural Information Processing (ICONIP)*, 2018.

U. Sharan and J. Neville. Temporal-relational classifiers for prediction in evolving domains. In *International Conference on Data Mining (ICDM)*, 2008.

R. Trivedi, M. Farajtabar, P. Biswal, and H. Zha. DyRep: Learning representations over dynamic graphs. In *International Conference on Learning Representations (ICLR)*, 2019.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *International Conference on Learning Representations (ICLR)*, 2018.

Y. Wang, Y. Chang, Y. Liu, J. Leskovec, and P. Li. Inductive representation learning in temporal networks via causal anonymous walks. In *International Conference on Learning Representations (ICLR)*, 2021.

S. De Winter, T. Decuypere, S. Mitrović, B. Baesens, and J. De Weerdt. Combining temporal aspects of dynamic networks with node2vec for a more efficient dynamic link prediction. In *International Conference on Advances in Social Networks Analysis and Mining*, 2018.

F. Wu, A. H. Souza Jr, T. Zhang, C. Fifty, T. Yu, and K. Weinberger. Simplifying graph convolutional networks. In *International Conference on Machine Learning (ICML)*, 2019.

D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan. Self-attention with functional time representation learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019a.

D. Xu, C. Ruan, E. Korpeoglu, S. Kumar, and K. Achan. Inductive representation learning on temporal graphs. In *International Conference on Learning Representations (ICLR)*, 2020.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019b.

W. Yu, W. Cheng, C. AC. Aggarwal, K. Zhang, H. Chen, and W. Wang. Netwalk: A flexible deep embedding approach for anomaly detection in dynamic networks. In *International Conference on Knowledge Discovery & Data Mining (KDD)*, 2018.

L. Zhu, D. Guo, J. Yin, G. V. Steeg, and A. Galstyan. Scalable temporal latent space inference for link prediction in dynamic social networks. *IEEE Transactions on Knowledge and Data Engineering*, 28(10):2765–2777, 2016.

A CAUSAL ANONYMOUS WALK (CAW)

Wang et al. (2021) propose Causal Anonymous Walk-Networks (CAW-N), which compute node embeddings using sets of temporal random walks S_u and S_v, starting respectively from the endpoints u and v of an edge to be predicted at time t. The random walk is *time-aware* in the sense that the log-probability of sampling w as the next node in a walk from u, is inversely proportional to the difference between the current time and the time of the last interaction between w and u. The random-walk size is governed by an hyperparameter L.

Once S_u and S_v are computed, CAW-N anonymizes each walk by replacing every node w with a footprint vector $I_{\text{CAW}}(w) \in \mathbb{R}^{L+1}$ such that $I_{\text{CAW}}(w)_\ell$ stores how many times w was the ℓ-th node in a walk of $S_u \cup S_v$. Let $\widehat{S}_u$ and $\widehat{S}_v$ denote the anonymous versions of S_u and S_v. CAW-N then treats every anonymous walk $\widehat{W} \in \widehat{S}_u \cup \widehat{S}_v$ as a sequence of pairs $(I_{\text{CAW}}(w_i), t_i)$, with $i = 0, \ldots, L$, and produces an embedding $\text{emb}(\widehat{W})$ by first transforming each pair into a vector $[f_1(I_{\text{CAW}}(w_i)) \parallel f_2(t_{i-1} - t_i)]$, and then running a recurrent network over that sequence of vectors. In Wang et al. (2021), function $f_1(\cdot)$ is a combination of MLPs, and $f_2(\cdot)$ is a time encoding scheme. Finally, CAW-N combines the anonymous walk embeddings using either mean-pooling or self-attention.

A.1 THE BUG: ATTENTION OVER THE BATCH INSTEAD OF THE WALKS

In the official code[1] by Wang et al. (2021), we note that the self-attention used for aggregating the sampled random walks is computed over the incorrect dimension. Specifically, given the tensor that contains representations of the sampled walks $\boldsymbol{h} \in \mathbb{R}^{B,N,D}$, where B is the batch size, N the number of walks and D the embedding size, the self-attention should be computed over N, but in the official release it is instead computed over B.[2] Consequently, when predicting a link at time t_i, this implementation allows the model to get information from the other events in the batch, in particular, to events at times t_j with $j > i$. This effectively allows CAW to "look to the future".

To illustrate the problem, we replicated the results of Wang et al. (2021) in the transductive setting, and use the same training setup to evaluate the test set using batches of size 1 (we emphasize that we only change the batch size during testing). The prediction algorithm of CAW does not depend on the batch size, but the results suggest that the test performance drops noticeably using batch size 1. We further fixed the implementation error so that the self-attention is now computed over the dimension of the walks, N, and retrained CAW with the same hyperparameters. Again, there is an evident drop in performance compared to the results reported by the authors.

In addition, we note that in the officially released code of CAW, the pooling method that aggregates walks is always set as attention, even when using the flag of mean pooling[3]. Thus, we modified it to be a mean aggregation, and the performance drops w.r.t the numbers provided in (Wang et al., 2021) with mean aggregation. Table 5 summarizes the results for the aforementioned modifications.

Table 5: Results of CAW in transductive setting (average precision).

Model	**Reddit**	**Wikipedia**	**MOOC**	**UCI**
Original (bug in the attention)	$99.75_{\pm 0.12}$	$100.0_{\pm 0.0}$	$97.55_{\pm 0.45}$	$93.56_{\pm 1.33}$
Test batch size 1	$85.29_{\pm 1.08}$	$92.94_{\pm 0.52}$	$75.57_{\pm 2.52}$	$77.10_{\pm 1.31}$
Corrected attention	$97.08_{\pm 0.06}$	$98.20_{\pm 0.07}$	$73.40_{\pm 0.48}$	$81.66_{\pm 0.59}$
Mean	$96.53_{\pm 0.12}$	$97.83_{\pm 0.13}$	$72.15_{\pm 0.51}$	$77.96_{\pm 1.67}$

Because of the aforementioned bug we use the "corrected attention" version of CAW for the experiments in the paper.

[1] After seeing our work, Wang et al. (2021) have updated their github repository to fix the bug.

[2] The specific line of code that generates the problem can be found in the official released code (Line 51) `https://github.com/snap-stanford/CAW/blob/master/transformer.py#L51` when using the multi-head attention implemented by the PyTorch library. That implementation expects a tensor arranged as (sequence_length, batch, embedding_size), but Wang et al. (2021) provide a tensor arranged as (batch, sequence_length, embedding_size).

[3] In the official implementation of CAW (Lines 120-124) `https://github.com/snap-stanford/CAW/blob/be07783b59824fbc5ed666b3e885d4a6abc8d1a3/main.py#L120` the walk_pool argument is not fed into CAW, thus it always takes the default value, which is the attention aggregation.

B DATASETS

For temporal link prediction, we use six popular benchmarks:

- Reddit[4] is a dataset of posts made by users on subreddits over a month. Nodes correspond to either users or subreddits, and links denote posting requests from users to subreddits, annotated with timestamps.
- Wikipedia[5] is a network where links correspond to timestamped updates that users (nodes) make to wiki pages (nodes). The dataset only comprises the 1000 most edited pages, and users with at least 5 edits within a month.
- MOOC[6] is a dataset of students' actions on a massive open online course. Its nodes represent either students or course content units, and its temporal links represent student's access to course units.
- UCI[7] is a dataset of posts to the University of California Irvine forum. Its nodes denote either users or forums, and the links represent timestamped non-attibuted forum messages.
- Enron[8] is a dataset of email communications in Enron. Its nodes represent core employees of Enron and links represent emails between them.
- LastFM[9] records one month of who-listens-to-which song information. Its nodes correspond to either users or songs.

We also create a Twitter dataset following the work of Rossi et al. (2020). We describe the details of the Twitter dataset in Appendix B.1. Table 6 reports summary statistics for each dataset.

Table 6: Summary statistics for temporal link prediction datasets. * Corresponds to edge features filled with zero values.

Dataset	#Nodes	#Edges	#Edge feat.
Reddit	10,985	672,447	172
Wikipedia	9,227	157,474	172
MOOC	7,145	411,749	4
Twitter	8,925	406,564	768
UCI	1,899	59,835	100*
Enron	184	125,235	32*
LastFM	1,980	1,293,104	2*

B.1 TWITTER DATASET

We base our Twitter dataset in the description given in the TGN paper (Rossi et al., 2020). To generate the dataset we begin with the data from the 2021 Twitter RecSys Challenge. Then we take the 10,000 nodes with the highest number of interactions in the dataset — and respectively their edge events. Note that not all of the 10,000 nodes will be left in the dataset, since some might no have interactions with other nodes in the dataset. To compute the edge features, we use Multilingual BERT on the provided text tokens.

C IMPLEMENTATION DETAILS

C.1 TRAINING PROCEDURE

For training, we follow the setting of previous T-GNNs for continuous-time dynamic graphs (e.g., TGN, TGAT and CAW). The dataset is a long sequence of events. Then, we partition these events into non-overlapping batches that are contiguous in time. At each epoch, our method goes over all batches, in chronological order. Also, at the beginning of each epoch, we reset the memory (node

[4] http://snap.stanford.edu/jodie/reddit.csv
[5] http://snap.stanford.edu/jodie/wikipedia.csv
[6] http://snap.stanford.edu/jodie/mooc.csv
[7] http://konect.cc/networks/opsahl-ucforum/
[8] https://www.cs.cmu.edu/~./enron/
[9] http://snap.stanford.edu/jodie/lastfm.csv

states) and the neighborhood information. Algorithm 1 summarizes the training procedure for OGN. Importantly, OGN only updates the node states and neighborhood states for the nodes u, v involved in a positive/true event. Given an edge event between nodes u and v, we sample a random node w to create a fake interaction with u. Since w is sampled over all possible nodes, it can occasionally be v itself. We deal with this case in line 16 of Algorithm 1.

During training, the gradients of OGN are back-propagated through a single step of the state update. In other words, the gradients are computed considering the previous node states as constant (i.e., we do not propagate gradients through time). Notably, all related works employ the same procedure for backpropagation.

Algorithm 1 Training OGN

Input: Edge events $\mathcal{E}$

1: **for** epoch **do**
2: $\quad \boldsymbol{s}^{(0)}, \boldsymbol{r}^{(0)}, \boldsymbol{a}^{(0)}, b^{(0)} \leftarrow \mathbf{0}, \mathbf{0}, \mathbf{0}, 0$ ▷ Reset states
3: $\quad$ **for** batch $(u, v, e, t) \in \mathcal{E}$ **do**
4: $\quad\quad w \leftarrow$ sample negative node
5: $\quad\quad \boldsymbol{r}_u^{(n-1)} \leftarrow \boldsymbol{a}_u^{(n-1)} / b_u^{(n-1)}$ ▷ Compute neighborhood state
6: $\quad\quad \boldsymbol{r}_v^{(n-1)} \leftarrow \boldsymbol{a}_v^{(n-1)} / b_v^{(n-1)}$
7: $\quad\quad \boldsymbol{r}_w^{(n-1)} \leftarrow \boldsymbol{a}_w^{(n-1)} / b_w^{(n-1)}$
8: $\quad\quad \boldsymbol{z}_u^{(n)} \leftarrow \textsc{ComputeState}\left(\boldsymbol{s}_u^{(n-1)}, \boldsymbol{r}_u^{(n-1)}, \boldsymbol{e}^{(n)}, \boldsymbol{t}^{(n)}\right)$ ▷ Compute node states (Eqs. 8-9)
9: $\quad\quad \boldsymbol{z}_v^{(n)} \leftarrow \textsc{ComputeState}\left(s_v^{(n-1)}, \boldsymbol{r}_v^{(n-1)}, \boldsymbol{e}^{(n)}, \boldsymbol{t}^{(n)}\right)$
10: $\quad\quad \boldsymbol{z}_w^{(n)} \leftarrow \textsc{ComputeState}\left(s_w^{(n-1)}, \boldsymbol{r}_w^{(n-1)}, \boldsymbol{e}^{(n)}, \boldsymbol{t}^{(n)}\right)$
11: $\quad\quad \boldsymbol{s}_u^{(n)}, \boldsymbol{s}_v^{(n)} \leftarrow \boldsymbol{z}_u^{(n)}, \boldsymbol{z}_v^{(n)}$ ▷ Update node states
12: $\quad\quad \boldsymbol{a}_u^{(n)}, b_u^{(n)} \leftarrow \textsc{UpdateNeigh}\left(\boldsymbol{s}_v^{(n)}, \boldsymbol{a}_u^{(n-1)}, b_u^{(n-1)}\right)$ ▷ Update neighborhood variables (Eqs. 6-7)
13: $\quad\quad \boldsymbol{a}_v^{(n)}, b_v^{(n)} \leftarrow \textsc{UpdateNeigh}\left(\boldsymbol{s}_u^{(n)}, \boldsymbol{a}_v^{(n-1)}, b_v^{(n-1)}\right)$
14:
15: $\quad\quad \boldsymbol{p}_{\text{pos}}, \boldsymbol{p}_{\text{neg}} \leftarrow \textsc{Predict}\left(\boldsymbol{z}_u^{(n)}, \boldsymbol{z}_v^{(n)}\right), \textsc{Predict}\left(\boldsymbol{z}_u^{(n)}, \boldsymbol{z}_w^{(n)}\right)$ ▷ Get predictions
16: $\quad\quad l \leftarrow \text{BCE}\left(\boldsymbol{p}_{\text{pos}}, \mathbf{1}\right) + \text{BCE}\left(\boldsymbol{p}_{\text{neg}}, \mathbf{0} + \mathbb{1}_{v=w}\right)$ ▷ Binary cross-entropy (BCE) loss
17: $\quad$ **end for**
18: **end for**

C.2 Evaluation setup and hyperparameters

Real-world temporal networks only comprise true edge events, i.e., *positive links* (class 1). To generate *negative links* (class 0), we follow the standard methodology (Rossi et al., 2020; Xu et al., 2020; Wang et al., 2021): for each positive link $e_{u,v}(t)$, we create a negative link $e_{u,v'}(t)$ with $v' \neq v$ uniformly sampled from a set of candidate nodes, using the same feature vector and timestamp as $e_{u,v}(t)$.

We train the models using both positive and negative links, and the binary cross-entropy loss. For CAW, we use Adam with learning rate 10^{-3} during 50 epochs, with early stopping if there is no improvement greater than 10^{-3} (default value in the original repository) in validation average precision for 3 epochs. For the rest of the methods (including OGN), we use Adam with learning rate 10^{-4} during 50 epochs, with early stopping if there is no improvement greater than 10^{-5} in validation average precision for 5 epochs. We follow Xu et al. (2020); Rossi et al. (2020) and use batch-size 200 for all methods, unless explicitly stated otherwise.

For CAW, we perform a grid search over the time decay $\alpha \in \{0.01, 0.1, 0.25, 0.5, 1.0, 2.0, 4.0, 10.0, 100.0\} \times 10^{-6}$, number of walks $M \in \{1, 2, 3, 4, 5\}$ and walk length $L \in \{32, 64, 128\}$. We present the best combination of hyperparameters in Table 7. For TGN, we follow (Rossi et al., 2020) and sample twenty temporal neighbors. For TGAT, we

sample twenty immediate neighbors and twenty 2-hop temporal neighbors following the guidelines in the original work (Xu et al., 2020).

Table 7: Hyperparameters for CAW.

Dataset	**Time decay α**	**#Walks**	**Walk length**
Reddit	10^{-8}	32	3
Wikipedia	4×10^{-6}	64	4
MOOC	10^{-4}	64	3
UCI	10^{-5}	64	2
Enron	10^{-6}	64	5

C.3 Hardware

We run experiments using a set of machines comprising heterogeneous GPU resources including Nvidia Tesla P100, Tesla V100, GTX 1080Ti, and TITAN RTX cards. To ensure fairness in time comparison (Figures 1 and 4), we also run all methods on the machine equipped with a consumer-grade GPU (Nvidia GTX 1080Ti) and an Intel Xeon E5-2630 v4 CPU with up to 40 GB RAM.

D Further Ablation Study

D.1 The importance of time information

In section 6, we show that methods using $\mathcal{U}$-time (i.e., uniformly discretized time) generally outperform their counterparts that use original timestamps. We now take a step further and evaluate the performance of T-GNNs when no time information is available and only the ordering of the events is preserved. In particular, we first create the sequence of events ordered by time and then set the actual value of timestamps to zero before feeding them to TGAT and TGN. We refer to this approach as NO-TIME.

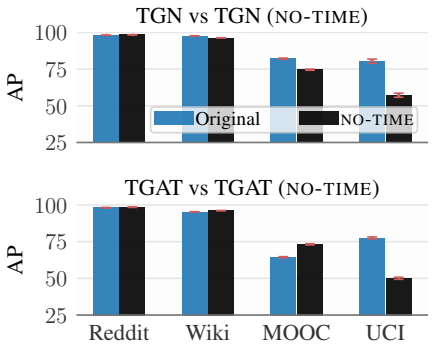


Figure 6: Removing timestamps can hurt the performance of T-GNNs. Both TGN (NO-TIME) and TGAT (NO-TIME) experience a significant performance drop on the UCI dataset.

Figure 6 shows the performance of representative T-GNNs with and without timestamps. Notably, the performance of TGN significantly decreases ($\approx$ 23% in AP) on the UCI dataset. These results show that there are cases in which leveraging the ordering of events alone is not enough to learn meaningful temporal node representations.

We note that the $\mathcal{U}$-TIME and NO-TIME approaches are fundamentally different. Unlike NO-TIME, $\mathcal{U}$-TIME still allows T-GNNs to count the total number of events between two interactions involving the same node. In summary, we conclude that, although fine-grained information is not crucial, some degree of time information is still important for T-GNNs.

D.2 The importance of attention

TGN, CAW and TGAT use attention as their aggregation layers. We evaluate the importance of this component in the performance of T-GNNs. For this purpose, we replace the attention module by an element-wise pooling (mean or max) followed by a linear layer.

Table 8 compares the performance of TGN and TGAT using attention modules against their counterparts using element-wise pooling (we report the best between mean and max). Interestingly, we do not see a significant gain by using attention over a mean or max layer in most datasets. In fact, the largest gap happens in favor of TGAT on MOOC. In this case, TGAT sees an accuracy boost of 7.53 when equipped with the mean/max layer. For both TGAT and TGN, attention only outperforms mean/max on Wikipedia.

Table 8: **Results with different aggregation modules (transductive setting).** Using attention either brings marginal gains or winds up hurting performance (TGAT on MOOC). The * denotes std < 0.05 rounded to the first decimal.

Model	Module	Reddit	Wikipedia	MOOC
TGN	Original	$98.1_{\pm 0.0}$*	$\mathbf{97.6_{\pm 0.1}}$	$82.1_{\pm 0.4}$
	Mean/Max	$98.1_{\pm 0.0}$*	$97.3_{\pm 0.1}$	$82.0_{\pm 0.2}$
TGAT	Original	$98.51_{\pm 0.1}$	$\mathbf{95.85_{\pm 0.1}}$	$66.09_{\pm 3.4}$
	Mean/Max	$98.47_{\pm 0.0}$*	$94.96_{\pm 0.2}$	$\mathbf{74.34_{\pm 0.5}}$

E Additional experiments

E.1 Unattributed datasets

In this section we present results for link prediction using three more datasets: UCI, Enron and LastFM (see Table 9). All of these datasets are unattributed, which means that they do not contain edge or node features. Notably, OGN consistently outperforms TGAT and there is no clear winner between TGN and OGN. Only CAW is consistently better than OGN on non-attributed data. While CAW's anonymous walks seem to better capture structural patterns in the graphs, CAW runs orders of magnitude slower (e.g., see Figure 1 and Figure 4).

Table 9: Average precision (AP) for link prediction on datasets that do not contain edge features.

Model	**UCI**		**Enron**		**LastFM**	
	Transductive	Inductive	Transductive	Inductive	Transductive	Inductive
Jodie	$86.73_{\pm 1.0}$	$75.26_{\pm 1.7}$	$77.31_{\pm 4.2}$	$76.48_{\pm 3.5}$	$69.32_{\pm 1.0}$	$80.32_{\pm 1.4}$
DyRep	$54.60_{\pm 3.1}$	$50.96_{\pm 1.9}$	$77.68_{\pm 1.6}$	$66.97_{\pm 3.8}$	$69.24_{\pm 1.4}$	$82.03_{\pm 0.6}$
TGAT	$77.51_{\pm 0.7}$	$70.54_{\pm 0.5}$	$68.02_{\pm 0.1}$	$63.70_{\pm 0.2}$	$54.77_{\pm 0.4}$	$56.76_{\pm 0.9}$
TGN	$80.40_{\pm 1.4}$	$74.70_{\pm 0.9}$	$79.91_{\pm 1.3}$	$\underline{78.96_{\pm 0.5}}$	$\underline{80.69_{\pm 0.2}}$	$\underline{84.66_{\pm 0.1}}$
CAW	$\mathbf{92.16_{\pm 0.1}}$	$\mathbf{92.56_{\pm 0.1}}$	$\mathbf{92.09_{\pm 0.7}}$	$\mathbf{91.74_{\pm 1.7}}$	$\mathbf{81.29_{\pm 0.1}}$	$\mathbf{85.67_{\pm 0.5}}$
OGN (ours)	$\underline{90.94_{\pm 0.3}}$	$\underline{81.60_{\pm 0.4}}$	$\underline{81.69_{\pm 3.2}}$	$77.71_{\pm 5.5}$	$71.02_{\pm 0.99}$	$83.41_{\pm 1.3}$

E.2 Small batch size

Table 10 reports absolute and relative training times for all methods using batch-size 32 on Reddit, Wikipedia, and MOOC. Notably, OGN is the fastest method by a margin. Nonetheless, the running time between OGN and CAW is smaller compared to when we use batch-size=200 Figure 4. In fact, CAW's running time is highly sensitive to the amount of available RAM. This is reasonable since CAW samples walks using the CPU and may require repeatedly accessing the main memory. Also, since the gap between all methods have reduced, it is reasonable to believe that additional communication between GPU and CPU (due to smaller batch size) might play an important role.

Table 10: **Time (seconds) per epoch with batch-size 32 (transductive setting).** We report the average value over 3 epochs. Numbers in parentheses indicate the relative training time wrt OGN.

Model	Reddit	Wikipedia	MOOC
TGAT	$2,113$ ($\times 6.4$)	362 ($\times 4.5$)	985 ($\times 6.9$)
TGN	$1,713$ ($\times 5.2$)	162 ($\times 2.0$)	640 ($\times 4.5$)
CAW	$8,723$ ($\times 26.6$)	$3,839$ ($\times 48.0$)	$5,591$ ($\times 39.4$)
OGN	328	80	142

E.3 Results with different negative sampling scheme

Table 11 shows results for link prediction using a different negative sampling scheme. For each node u, we only sample negative edges that have an endpoint in a temporal neighbor. For all models we use the same hyperparameters as in the original setting. In this case, the results are expected to be worse than those with the original sampling, that also samples edges that did not interact previously with u. OGN achieves the best performance in two out of three datasets (Reddit and MOOC). In Wikipedia, OGN is the second best.

Table 11: Results (AP) using negative sampling on visited nodes in the transductive setting.

Model	Reddit	Wikipedia	MOOC
TGAT	66.7	60.8	63.8
TGN	<u>95.6</u>	**97.7**	78.5
CAW	63.9	67.9	<u>82.3</u>
OGN	**97.6**	<u>94.5</u>	**91.1**

F FURTHER INTUITION BEHIND THE WEIGHTED AVERAGE

We show that the neighborhood state $\boldsymbol{r}_u^{(n)}$ in, Equation 5, can be seen as an expected value taken over the the neighbors $\mathcal{N}_u^{(n)}$ of node u. For this purpose, we define the probability mass function $p : \mathcal{N}_u^{(n)} \to [0, 1]$ given by

$$p(i) = \frac{e^{-\alpha \Delta m_i}}{\sum_j e^{-\alpha \Delta m_j}} \quad \forall i \in \mathcal{N}_u^{(n)}, \tag{10}$$

where $\Delta m_i = n - m_i$ Then, it follows by definition that:

$$\boldsymbol{r}_u^{(n)} = \mathbb{E}_{i \sim p}[\boldsymbol{s}_i^{(m_i)}] = \sum_{i \in \mathcal{N}_u^{(n)}} p(i) \boldsymbol{s}_i^{(m_i)} \tag{11}$$

Note also that $\boldsymbol{s}_i^{(m_i)}$ also encapsulates information from the neighborhood of node i (see Equation 8 and Equation 9). Therefore, $\boldsymbol{r}_u^{(n)}$ captures multi-hop information.

G PROOF: TRACTABLE AGGREGATION

We now show that the ratio between $\boldsymbol{a}_u^{(n)}$ and $\boldsymbol{b}_u^{(n)}$ equals $\boldsymbol{r}_u^{(n)}$. Recall that $\boldsymbol{r}_u^{(n)}$ is given by:

$$\boldsymbol{r}_u^{(n)} = \frac{\sum_{i \in \mathcal{N}_u^{(n)}} e^{-\alpha \Delta m_i} \boldsymbol{s}_i^{(m_i)}}{\sum_{j \in \mathcal{N}_u^{(n)}} e^{-\alpha \Delta m_j}}. \tag{12}$$

More specifically, we prove by induction on the number of events n that $\boldsymbol{a}_u^{(n)}$ and $\boldsymbol{b}_u^{(n)}$ equal the numerator and denominator of Equation 12, respectively. Both proofs are straightforward and follow the same structure.

Proposition 2. *For all $n \in \mathbb{N}^+ \cup \{0\}$, it holds that*

$$\boldsymbol{a}_u^{(n)} = \sum_{i \in \mathcal{N}_u^{(n)}} e^{-\alpha \Delta m_i} \boldsymbol{s}_i^{(m_i)}.$$

Proof. For $n = 0$, $\boldsymbol{a}_u^{(0)} = 0$ by definition and the identity holds. Assume the above identity holds for an arbitrary $n - 1 \geq 0$, i.e.,

$$\boldsymbol{a}_u^{(n-1)} = \sum_{i \in \mathcal{N}_u^{(n-1)}} e^{-\alpha \Delta m_i} \boldsymbol{s}_i^{(m_i)} = \sum_{i \in \mathcal{N}_u^{(n-1)}} e^{-\alpha (k - m_i)} \boldsymbol{s}_i^{(m_i)},$$

where k is the latest event for node u within the $n - 1$ first events. Applying the update to $\boldsymbol{a}_u^{(n-1)}$ (Equation 6), we get

$$\boldsymbol{a}_u^{(n)} = e^{-\alpha(n-n)} \boldsymbol{s}_v^{(n)} + e^{-\alpha(n-k)} \sum_{i \in \mathcal{N}_u^{(n-1)}} e^{-\alpha(k-m_i)} \boldsymbol{s}_i^{(m_i)} = \sum_{i \in \mathcal{N}_u^{(n)}} e^{-\alpha(n-m_i)} s_i^{(m_i)}$$

□

Proposition 3. *For all $n \in \mathbb{N}^+ \cup \{0\}$, it holds that*

$$\boldsymbol{b}_u^{(n)} = \sum_{j \in \mathcal{N}_u^{(n)}} e^{-\alpha \Delta m_j}.$$

Proof. For $n = 0$, $b_u^{(0)} = 0$ by definition and the identity holds. Assume the above identity holds for an arbitrary $n - 1 \geq 0$, i.e.,

$$b_u^{(n-1)} = \sum_{j \in \mathcal{N}_u^{(n-1)}} e^{-\alpha \Delta m_j} = \sum_{j \in \mathcal{N}_u^{(n-1)}} e^{-\alpha(k - m_i)}$$

where k is the latest event for node u within the $n - 1$ first events. Applying the update to $\boldsymbol{b}_u^{(n-1)}$ (Equation 7), we get

$$b_u^{(n)} = e^{-\alpha(n-n)} + e^{-\alpha(n-k)} \sum_{j \in \mathcal{N}_u^{(n-1)}} e^{-\alpha(k - m_i)} = \sum_{j \in \mathcal{N}_u^{(n)}} e^{-\alpha(n - m_j)}$$

□