Cascade: A Dependency-Aware Efficient Training Framework for Temporal Graph Neural Networks

Yue Dai
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA, USA
yud42@pitt.edu

Xulong Tang
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA, USA
tax6@pitt.edu

Youtao Zhang
University of Pittsburgh
Department of Computer Science
Pittsburgh, PA, USA
youtao@pitt.edu

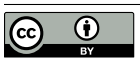
Abstract

Temporal graph neural networks (TGNN) have gained significant momentum in many real-world dynamic graph tasks. These models use graph changes (i.e., events) as inputs to update nodes' status vectors (i.e., memories), which are then exploited to assist predictions. Despite their improved accuracies, the efficiency of TGNN training is significantly limited due to the inherent temporal relationship between the input events. Although larger training batches can improve parallelism and speed up TGNN training, they lead to infrequent memory updates, which cause outdated information and reduced accuracy. This trade-off forces current methods to use small batches, resulting in high latency and underutilized hardware. To address this, we propose an efficient TGNN training framework, Cascade, to adaptively boost TGNN training parallelism based on nodes' spatial and temporal dependencies. Cascade adopts a topology-aware scheduler that includes as many spatial-independent events in the same batches. Moreover, it leverages node memories' similarities to break temporal dependencies on stabilized nodes, enabling it to pack more temporal-independent events in the same batches. Additionally, Cascade adaptively decides nodes' update frequencies based on runtime feedback. Compared to prior state-of-the-art TGNN training frameworks, our approach can averagely achieve 2.3× (up to 5.1×) speed up without jeopardizing the resulted models' accuracy.

CCS Concepts: • **Computing methodologies** → **Artificial intelligence**; **Learning paradigms**; **Parallel algorithms**.

Keywords: Temporal Graph Neural Network, Dynamic Graph, Efficient Deep Learning, Parallel Computing

ACM Reference Format:
Yue Dai, Xulong Tang, and Youtao Zhang. 2025. Cascade: A Dependency-Aware Efficient Training Framework for Temporal Graph Neural Networks. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3676641.3716250



ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1079-7/2025/03.
https://doi.org/10.1145/3676641.3716250

1 Introduction

Dynamic graphs are widespread across various domains, such as social media networks [20], knowledge graphs [22], autonomous systems [24], and traffic networks [29]. Unlike static graphs, whose nodes and edges remain constant, dynamic graphs evolve over time, introducing challenging tasks [1, 3, 6, 16, 20, 21, 31]. Inspired by the successes of Graph Neural Networks (GNNs) [8, 9, 15, 19, 39, 48], Temporal graph neural networks (TGNNs) have attracted growing attention for their improved accuracies in many real-world dynamic graph tasks [17, 21, 31, 33, 38, 43, 45, 47, 50, 52, 55]. On top of native GNNs, recent TGNN models keep a state vector for each node, called *node memory*, to encode the temporal dynamics and spatial relationships around the node. These memory vectors are continually updated and serve as the basis for making predictions, allowing TGNNs to achieve extraordinary prediction accuracy. With growing demands and interests in TGNN-based models, there is an escalating demand for developing training schemes that can swiftly adapt TGNNs to the ever-changing landscapes of dynamic graphs, ensuring that these models can be deployed quickly and effectively in real-world scenarios.

However, TGNN training faces significant challenges due to the sequential dependencies of input events, which substantially limit throughput. Recent studies utilize Continuous-Time Dynamic Graphs (CTDGs) to model the evolving dynamics of graphs by viewing them as sequences of event updates, such as changes to nodes or edges. These events are typically represented as edges connecting one node to another and are chronologically ordered by their timestamps. Existing TGNN training approaches segment these event sequences into batches for parallel processing [31, 45, 55, 56]. The computation within a single batch generally involves three steps: First, TGNNs predict edge presence or node classes based on the latest node memories, then compare these predictions to events within the batch (as the ground truth) to calculate losses and update model weights. Second, the model uses events within the batch to generate

messages for updating node memories. Lastly, it uses these messages to update nodes' memories for future usage. However, this batching process often overlooks the sequential occurrence order of events within the same batch. Consequently, larger batches may speed up processing but rely on potentially outdated node memories, thus compromising prediction accuracy. To maintain high accuracy, smaller batches are preferred to ensure updates incorporate the most recent memories. Nonetheless, these smaller batches lead to more training iterations and under-utilize the underlying hardware resources, leading to inefficiencies. For example, while training the Temporal Graph Network [31] on the Wikipedia [20] dataset, a batch size of 900 results in a 25% better validation loss compared to a batch size of 6000 but slows training on a Nvidia A100 GPU by 3.5×.

Fortunately, there exist significant opportunities to strategically increase the size of training batches in TGNNs without compromising the freshness of the nodes' memories. Specifically, the potential stems from two key observations: First, input events tend to occur sparsely across different parts of the dynamic graph. In particular, while specific nodes may experience frequent incoming and outgoing events, leading to quickly outdated memories, others may see much fewer events and retain their memory up-to-date over time. Consequently, when events impact distinct areas of the graph, they will have weak dependencies, presenting an opportunity to process them in parallel without losing accuracy. Second, memories within specific nodes could remain stabilized within a period. In particular, some nodes' memories may not change much by their outgoing or incoming events and, therefore, can be updated less frequently. For example, a consistently popular product in an e-commerce graph may have stable states (e.g., rating) despite frequent purchases. Events related to such stable nodes possess low dependencies between each other, hence allowing for their parallel processing. These insights suggest a pathway to optimizing TGNN training: By recognizing and leveraging the spatial relationship of events and the temporal stability of node memories, it's feasible to expand batch sizes adaptively.

Inspired by these observations, we propose a TGNN training framework, Cascade, to adaptively boost TGNN training parallelism based on input events' spatial and temporal dependencies. Cascade adaptively increases batch size during TGNN training in three folds: First, it uses a topology-aware batching algorithm to pack as many spatially independent events as possible into single batches, maximizing parallel processing while maintaining memory freshness. Second, it identifies stabilized nodes with minimal memory variations and excludes their updates from batching decisions, bypassing their temporal dependencies for more flexible batch configurations. Third, it dynamically adjusts the frequency of node memory updates based on runtime feedback during the training. We summarize our contribution as follows,

- We investigate the trade-off between parallelism and accuracy in TGNN training and recognize the potential of boosting parallelism of TGNN training by batching spatial and temporal independent events adaptively.
- We propose a TGNN training framework, Cascade, to dynamically identify the spatial and temporal dependencies between input events and pack as many events as possible without worsening node memories' freshness.
- We evaluate our approach on various real-world benchmarks. The experimental results show that our proposed training framework can achieve up to 5.1× speedup (2.3× on average) over the state-of-the-art TGNN training framework without increasing model losses.

2 Background

2.1 Dynamic Graphs

In contrast to static graphs, which are characterized by a constant set of nodes and edges $G = (V, E)$, dynamic graphs embody nodes and edges that evolve over time. There are two primary representations of dynamic graphs: Discrete-time dynamic Graphs (DTDGs) describe them as a sequence of static graph snapshots taken periodically, while Continuous-Time Dynamic Graphs (CTDGs) view them as a sequence of events, each detailing updates like edge changes. Recent studies have shown a preference for CTDGs due to their superior capacity for capturing detailed temporal variations over the static time frames inherent to DTDGs [17, 31, 54, 55]. In fact, DTDGs are often considered specific instances of CTDGs, distinguished by the segmentation of events into uniform time intervals [55]. In the CTDGs, dynamic graphs are denoted as dynamic graphs as $G = \{e(t_1), e(t_2), ...\}$, where each $e(t_i)$ indicates an event happened at timestamp t_i, typically represented as an edge with a timestamp. The prediction tasks for CTDGs can be depicted in Equation 1.

$$y_i = f_\theta(G_i^-, t_i) = f_\theta(\{e(t_1), e(t_2), ...e(t_{i-1})\}, t_i) \quad (1)$$

At the prediction time t_i, the model $f_\theta(\cdot)$ takes all previous events $G_i^- = \{e(t_1), e(t_2), ...e(t_{i-1})\}$ as inputs and predicts the testing nodes' classes or the presence of future edges.

2.2 Temporal Graph Neural Networks

The Temporal Graph Neural Networks (TGNNs) are widely studied and achieve state-of-the-art accuracies in CTDG tasks [17, 21, 31, 33, 38, 43, 47, 50, 52]. In addition to embedding nodes' neighborhood information like Graph Neural Networks (GNNs) [8, 9, 15, 19, 39, 48], TGNNs maintain a state vector, usually referred as *node memory*, for each node. This memory encodes the node's history and is used for predictions. The node memory is updated once the node is the destination or the source of a new event. Specifically, TGNNs produce node embedding for the predictions in three steps:

First, if an event $e(t)$ adds an edge e_{uv} from $node_u$ to $node_v$ (i.e., $e(t) = e_{uv}$), the **message generating** step will be triggered, in which two messages are generated as Equation 2. For simplicity, we only present the updating and following operations of $node_u$, which is the same for $node_v$.

$$m_{vu} = msg(s_v^-, s_u^-, \Delta T, e_{uv}) \tag{2}$$

The $msg(\cdot)$ is a learnable module such as Multi-Layer Perceptions (MLPs). The s_u^- and s_v^- denote the memories of $node_u$ and $node_v$ at their last updated times, e_{uv} denotes the edge features, and ΔT is the difference between the event's occurring timestamp and $node_u$'s last updated time.

Second, when TGNN models trigger a **memory updating** step, nodes u and v aggregate messages generated by previous events, then update their memories as Equation 3.

$$s_u^+ = UPDT(s_u^-, AGGR(m_{ku}^-|k \in N(u)), \tag{3}$$

The $N(u)$ denotes neighbors of $node_u$. The $AGGR(\cdot)$ is usually implemented by a *mean* (i.e., averaging sampled messages), *most_recent* (i.e., directly using the latest message) function to aggregate messages from the node's neighbors [21, 31, 42]. The $UPDT(\cdot)$ uses aggregated results to update the node's memory, which is usually implemented by a recurrent neural network such as Gated-Recurrent-Unit (GRU) [7].

Lastly, when TGNNs make a prediction that involves $node_u$, the **node embedding** step is triggered, in which TGNNs use a GNN module, such as Graph Attention Network (GAT) [39], to embed the node's and its neighbors' memories into its final node embedding, as depicted in Equation 4.

$$h_u = GNN(s_u, s_k|k \in N(u)), \tag{4}$$

The resulting node embedding h_u is fed into a final MLP module to get the prediction results.

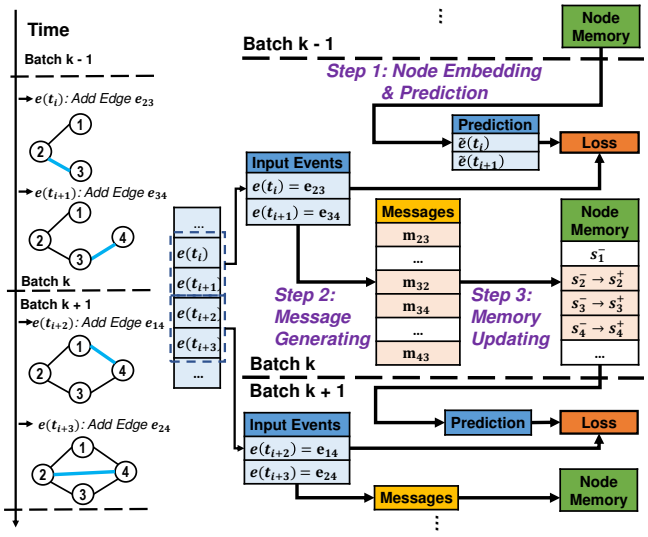


Figure 1. TGNN training steps: First start with node embedding and prediction, then message generating, and finally memory updating.

2.3 TGNN Training on CTDG

In recent developments, training methods for TGNNs have evolved from traditional snapshot-based approaches, which process the dynamic graph in a snapshot-by-snapshot manner (i.e., DTDG) [29, 32, 33, 40, 44, 47, 50, 52], to event-batching training methods, which segment the input event sequence of CTDGs into batches and then process events within a single batch in parallel [21, 31, 43, 46, 55, 56]. Each batch's events serve a dual purpose: they act as the ground truth for calculating prediction losses and the inputs for updating node memories.

Using edge prediction task as an example, as illustrated in Figure 1, a TGNN model takes a sequence of events (i.e., graph changes) as training inputs, divides the sequence into batches, then processes each batch in three steps as follows:

(1) First, it uses the node memories updated in the previous batches to embed node final representations and use the resulting node features to predict the events in the current batch. The trainer will then calculate losses based on the predictions and the input events, back-propagate losses, and update model weights accordingly. For instance, as shown in the figure, if there is an event $e(t_i) = e_{23}$ in the batch k, the TGNN will use s_2^-, s_3^- from updated before the batch as s_u, s_k to compute h_2 as Equation 4 and predict the probability of the edge $\hat{e}(t_i)$. The trainer will then compute the Binary-Cross-Entropy Loss to measure how much the probability of this real edge is higher than a wrong edge, such as e_{28}, and use the optimizer like Adam Optimizer [18] to backward propagate the loss and update the model weights.

(2) Second, the messages are generated based on the input events within the batch. For instance, if there is an event $e(t_i) = e_{23}$ in the batch k, the model will generate m_{23} and m_{32} based on the event e_{23}, its timestamp t_i, and its source node's current memory s_2^-, and destination node's current memory s_3^- following Equation 2).

(3) Lastly, for each event within the current batch, the trainer updates its source and destination nodes' memories. For instance, since $e(t_i) = e_{23}$ in the batch k involves $node_2$ and $node_3$, the models will update s_2^- to s_2^+ and s_3^- to s_3^+ as Equation 3) to ensure that they have up-to-date information.

In the batched training diagram, all the events within the same batch will be processed in parallel to finish the abovementioned steps. For example, for batch k, the events e_{23} and e_{34}, will be processed in parallel: First, the s_2^-, s_3^-, s_4^- will be used to compute the probability e_{23} and e_{34} (i.e., $\hat{e}(t_i)$ and $\hat{e}(t_{i+1})$); next, messages m_{23}, m_{32}, m_{34} and m_{43} will be generated in parallel as well; lastly, s_2^-, s_3^-, s_4^- will be updated to s_2^+, s_3^+, s_4^+ in parallel using the previous node memories.

3 Motivation

3.1 Challenge in Batched TGNN Training

While batching as many events during TGNN training enhances training efficiency by parallel processing input events,

it risks using outdated information and neglecting the temporal sequence of these events within a single batch. This

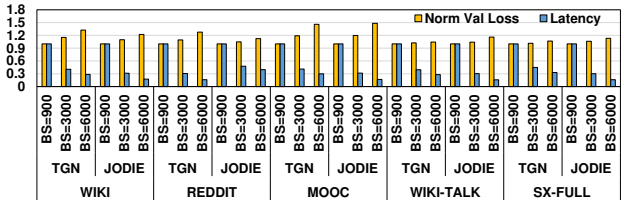


Figure 2. Normalized training latency and validation loss in TGN and JODIE trained under different batch sizes.

oversight makes TGNN models insensitive to intra-batch graph dynamics, potentially compromising their awareness of changes occurring within the current training batch. A major consequence of batching too many events is the potential expiration of node memories, leading to outdated embeddings, stale messages, and inaccurate memory updates. Moreover, concurrent event processing may disrupt their temporal sequence, which is crucial for capturing the graph's evolution. For example, a trending article's recommendation (as an event) may trigger rapidly increased product purchases in follows (as following events), showcasing how the temporal order of events can signal significant shifts in the graph's structure. As such, large batches could potentially jeopardize model accuracies, leading to compromised training results.

However, opting for small batches to preserve the temporal integrity of events may inevitably slow down the training process since it increases the number of training iterations required per epoch. To explore the impact of training batch sizes on training results and latencies, we employ a state-of-the-art training framework, TGL [55], to train two TGNN—Temporal Graph Network (TGN) [31] and JODIE [21]—on the datasets listed in Table 2. More details about the models and datasets are included in Section 5.1. Specifically, we train the models in different training batch sizes on a Nvidia A100 GPU and then evaluate their performance at a batch size of 900. As shown in Figure 2, while larger batches effectively reduce training latency, the resulting models' validation loss significantly increases. For instance, compared to `BS=900` (using a batch size of 900), although `BS=6000` reduces 71% TGN training latency on `WIKI`, the corresponding valuation loss is increased by 35%. Small batches, while helping accuracies, could cause poor training latencies. Moreover, the hardware utilization is significantly low in small batches. For instance, when training TGN on `WIKI` with `BS=900`, the streaming multiprocessor and memory utilization are as low as 17.2% and 15.2%, respectively. In contrast, `BS=6000` increases these values to 39.8% and 34.2%. To this end, finding a solution that balances TGNN training efficiency and effectiveness is significant yet challenging.

3.2 Spatial-independence in Scattered Events

Our first observation is that *training batch can be enlarged without accuracy loss by adding events from different subgraph regions.* Specifically, events within the input sequence often occur in distinct subgraphs and impact diverse sets of nodes. They are independent of each other and can be added to the same batch for two reasons: First, not all nodes will experience as many events around them during a period; thus, they do not expire simultaneously. This staggered expiration allows us to continue relying on nodes that remain unaffected since their last updates. Second, because events in different subgraphs typically exert minimal influence on each other, they can be processed in parallel without jeopardizing the integrity of the temporal information they carry.

To assess the potential of scattered events, we segment the training sets of datasets in Table 2 using a batch size of 900 and analyze the distribution of node degrees (i.e., the number of events outgoing from and incoming to each node) within these batches. As shown in Figure 3, most nodes are involved in far fewer events than a subset of highly connected nodes—the majority have only 0 to 25 events per batch. Even the most connected nodes have only 140 to 175 events, far less than the batch size. Hence, by significant chance, we can pack more events into the batches if they are spatially independent of current batched events.

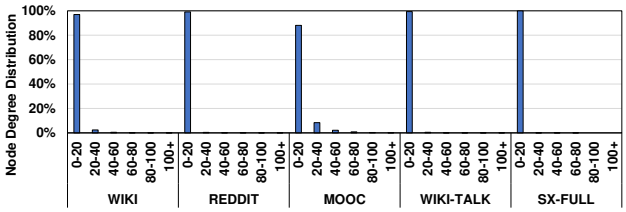


Figure 3. The distribution of nodes' degree within the batch size of 900 in different datasets.

However, the fixed batching strategy used by existing approaches cannot fully exploit the opportunities from this spatial-independent input. While large batches may potentially include more spatial-independent events, they could potentially pack too many dependent events if there are events extensively occur around specific nodes; conversely, although small batches may mitigate too aggressive batching on those high-degree nodes, they could potentially miss the opportunity of packing spatial-independent events. As such, an ideal batching scheme should adaptively increase and decrease batch sizes to include as many spatial-independent events as possible while avoiding packing too many spatial-dependent events on those high-degree nodes. We illustrate an example in Figure 4, in which an original batch contains events related to $node_1$ and its neighbors: If the following events continue to affect the same set of nodes (e.g., e_{16}, e_{15}, e_{13} continues to affect $node_1$ and its neighbors), then

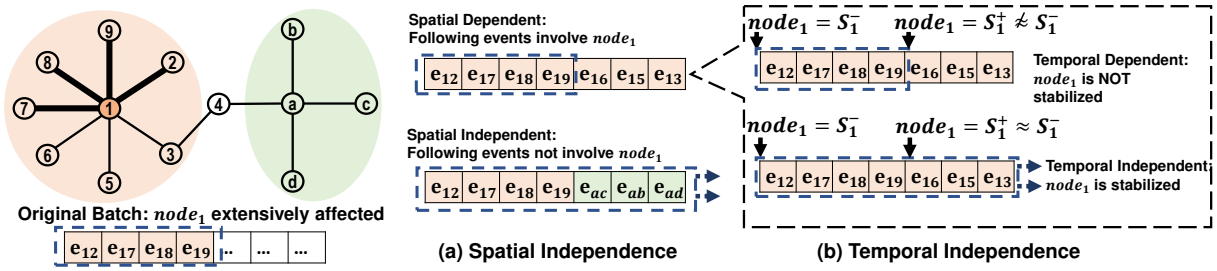


Figure 4. An example illustration of (a) Spatial Independence due to events scattered in different subgraphs; and (b) Temporal Independence due to stabilized node memories with their consequent potential in increasing batch sizes.

they are spatial dependent to current batch and the batch cannot be increased. Conversely, if the following event affects other nodes (e.g., e_{ac}, e_{ab}, e_{ad} affect $node_a$ and its neighbors instead of $node_1$), we may expand the batch to include them.

3.3 Temporal-independence in Stabilized Memories

Our second observation is that *training batch can also be enlarged without accuracy loss by adding events related to stabilized node memory.* Specifically, during the memory updating phase in TGNNs, many nodes reach a state of stability for extended periods. These stabilized nodes provide reliable, up-to-date memories, and their stability enables associated events to be processed in parallel without missing important temporal information. The intuition behind this is that nodes in the real world may usually stabilize and show similar behaviors over a period. For instance, a Reddit user may consistently show interest in specific topics, such as a particular game, and frequently engage in related discussions.

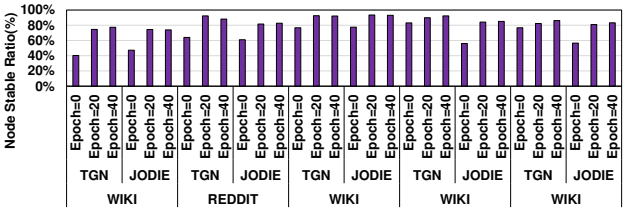


Figure 5. The ratio of stable node updates in different epochs when TGN and JODIE.

As depicted in Figure 5, while training TGN [31] and JODIE [21] (on the datasets specified in Table 2), on average, over 84% of the nodes maintain similar memory before and after updates (i.e., with a cosine similarity higher than 0.9) when models are trained after 20 epochs. To this end, by measuring runtime information and identifying these stabilized nodes, it is highly possible to adaptively neglect unnecessary temporal dependencies among events, thereby batching more events related to the same but stabilized nodes into a single batch without sacrificing the integrity of temporal data. We illustrate the cases using the same example in Figure 4: For the extensively affected $node_1$, if it is not stabilized (i.e., it has dissimilar memories before and after the node update in the original batch), then we need to update it before conduct following computations on e_{16}, e_{15}, e_{13}. Conversely, suppose it is stabilized (i.e., it has highly similar memories before and after the node update in the original batch). In that case, we may expand the batch to conduct computations on e_{16}, e_{15}, e_{13} as they can use similar input no matter with or without updating nodes' memories.

4 Design

4.1 Overview of Cascade

We introduce Cascade, an efficient training framework to increase batch sizes while keeping model accuracy. It consists of three designs: First, we propose a ***Topology-aware Graph Diffuser (TG-Diffuser)*** to incorporate spatial-independent events into batches. Second, we design a ***Similarity-aware Graph Filter (SG-Filter)*** to add temporal-independent events into batches. Lastly, we introduce an ***Adaptive Batch Sensor (ABS)***, a profile-based auto-tuner to analyze input training data and automatically control the TG-Diffuser.

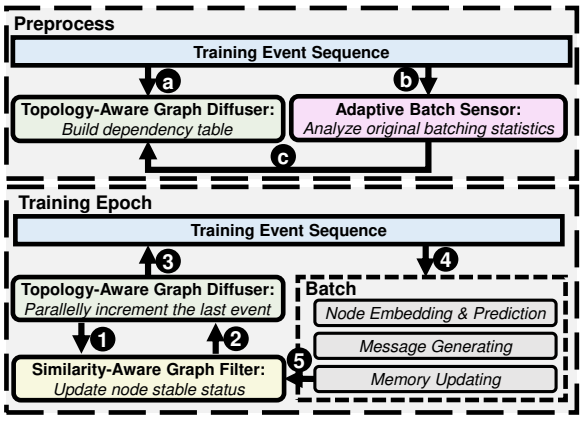


Figure 6. The workflow overview of Cascade.

The complete workflow of the Cascade framework is illustrated in Figure 6. We also detail it in Algorithm 1. Before training, the TG-Diffuser and ABS collaboratively preprocess the sequence of events in three steps: Initially, the

TG-Diffuser analyzes the training events (❶) to construct a dependency table that captures both spatial and temporal dependencies among the events (i.e., line 5 in Algorithm 1). Subsequently, the ABS processes these events as input (❷) and profiles the batching patterns using an originally defined sample batch size, which is small enough to ensure the training proceeds without deteriorating the model's performance (i.e., line 6 in Algorithm 1). Based on this analysis, the ABS sets the appropriate hyper-parameters for the TG-Diffuser (❸) (i.e., line 7 in Algorithm 1), optimizing the training setup. During training, the TG-Diffuser collaborates

Algorithm 1: TGNN training in Cascade

```
   Input : G: Input dynamic graph as event sequence; N
           Number of input training events; B_0: pre-defined
           batch size; E: Training epochs.
1  Initialize: Topology-Aware_Graph_Diffuser(TG-Diffuser);
2  Initialize Adaptive_Batch_Sensor(ABS);
3  Initialize Similarity-Aware_Graph_Filter(SG-Filter);
4  Initialize TGNN model(TGNN);
   // Preprocessing before training:
5  TG-Diffuser.build_dependency_table(G);
6  ω = ABS.max_endurance_profiling(G);
7  TG-Diffuser.set_parameters(ω);
8  st_idx = ed_idx = 0;
   // Training:
9  for e = 0, 1, ··· , E − 1 do
10     SG-Filter.reset();
11     while ed_idx < N do
12         st_idx = ed_idx;
           // Signify stable nodes:
13         S = SG-Filter.get_stable_nodes();
           // Get current batch:
14         ed_idx = TG-Diffuser.get_last_event_index(S);
15         ŷ = G[st_idx : ed_idx];
           // Model Training:
16         H = TGNN.Node_Embedding(ŷ);
17         y = MLP(H);
18         L(y, ŷ).backward();
19         TGNN.Generate_Message(ŷ);
20         TGNN.Update_Node_Memory(ŷ);
           // Update stable node flags:
21         SG-Filter.update_stable_nodes_flags(TGNN);
22 return TGNN;
```

with the SG-Filter to dynamically increase the training batch sizes through a five-step process: Initially, the TG-Diffuser sends a request to the SG-Filter (❶) and retrieves (❷) the node stable flags(i.e., line 12 in Algorithm 1), which are reset to all-false at the start of each epoch. Then, the TG-Diffuser ignores those stable nodes and identifies the last tolerable events for the current batch using the previously established dependency table (i.e., line 13 in Algorithm 1). This information is then used to segment a new batch from the training event sequence (❸) (i.e., line 14 in Algorithm 1). Following this, the TGNN models access the relevant events (❹) and proceed with the designated training steps (i.e., lines 15-19 in Algorithm 1). Lastly, the SG-Filter dynamically updates the node stable flags based on the node memories before and after the updates within the current batch (❺) (i.e., line 20 in Algorithm 1), ensuring that the node stable information is dynamically adjusted over training.

4.2 Topology-Aware Graph Diffuser

The TG-Diffuser efficiently integrates spatially independent events into batches through a two-step process: Initially, before training, it builds a dependency table that maps the spatial relationships between input events and nodes. The table reflects all related events around nodes. Next, for each batch, the TG-Diffuser independently identifies the last tolerable event on different nodes, which signifies the necessity to update node memories, and then includes all preceding unprocessed events up to this point into the current batch.

Build Dependency Table. Given a training dynamic graph of N nodes, the TG-Diffuser first builds a N-entries Dependency Table to reflect the spatial dependency between the training events and their related nodes. Each entry within the table contains two fields: `Node Idx` describes a node, and `Event Idx` consists of a sequence of event indices that indicate the events that may affect the node and, conversely, potentially may rely on the node. We illustrate the workflow of building the dependency table in Figure 7(a) and show the detailed algorithm in Algorithm 2. For each node (i.e., each table entry), the TG-Diffuser fills its `Event Idx` in two steps. First, it inserts all incoming and outgoing events indices of the current node into the `Event Idx`. For instance, as shown by the *Step 1*, for $node_1 = n_1$, its incoming and outgoing events $\{e(0), e(1), e(2), e(3), e(8), e(10)\}$ are added into the entry. The reason behind this is straightforward—all these events will be directly used to update the node's memory as m_{ku}^{-} in Equation 3, and the prediction about them will directly use the node's memory as s_u in Equation 4. Second, the TG-Diffuser looks up the node's neighbors and adds all their future events to the current node's `Event Idx`. As shown by the *Step 2*, where $node_1$ has $node_3$ as its neighbors due to $e(8) = e_{13}$, we add the events of $node_3$ after $e(8)$ into $node_1$'s `Event Idx`. These events are relevant to the current node because they update the neighbors' memories, influencing the current node's future memory updating and embedding; reversely, predicting them relies on features of the current node. It is worth mentioning that we do not include the past events in neighbors before they are connected to the current node (e.g., do not add events of $node_3$ before index 8)) since these neighbors are independent with the current node before there is an event building a connecting between them. We only consider events from the current node's 1-hop neighbors since they directly affect the current node's memories and propagate information from further

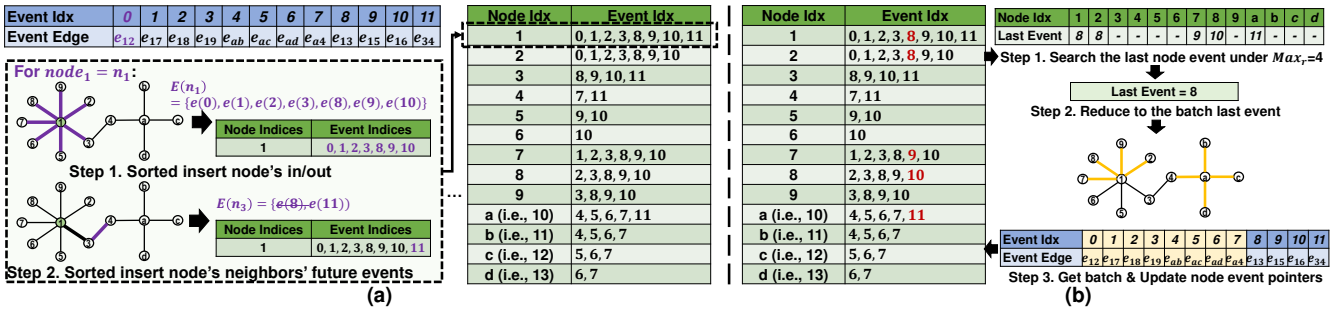


Figure 7. The workflow of TG-Diffuser: (a) Building dependency Table during preprocessing and (b) Looking up the last tolerable event during training.

Algorithm 2: Build Dependency Table

Output: D: Node-event dependency Table
Input : G: Input dynamic graph as event sequence; N Number of nodes.

Initialize $D = \{D_0, D_1, \cdots, D_{N-1}\}$;
// Loop Parallel:
for $n = node_0, node_1, \cdots, node_{N-1}$ **do**
  for $e_{nq} \in OutEvents(n) \cup InEvents(n)$ **do**
    // e_{nq}: Event from node n to node q
    $D_n.sorted\_insert(e_{nq})$;
    for $e_{qk} \in OutEvents(q) \cup InEvents(q)$ **do**
      // Insert future events in q
      if $e_{qk}.index > e_{nq}.index$ **then**
        $D_n.sorted\_insert(e_{qk})$;
return D;

distant neighbors. For instance, if $node_x$ has neighbor $node_y$ and $node_y$ has neighbor $node_z$, the updates in $node_z$ will not affect $node_x$ unless $node_y$ is updated; otherwise, $node_x$ will always use the same version of $node_y$, even if $node_y$ has an expired neighbor $node_z$. We implement the process using OpenMP [4] to enable parallel building; to ensure that the resulting `Event Idx` contains unique events sorted by their occurrence, we use sets in C++ to implement the `Event Idx` entries. The dependency table is stored in the host memory and will not be updated once built.

Get Last Tolerable Event. During training, TG-Diffuser looks into each node and finds the last tolerable event for the current batch independently. Intuitively, the process includes more events on those less affected nodes without introducing more events to those mostly affected nodes. To quantitatively measure the extension of being affected, we introduce a new parameter, namely, *Maximum Revisit Endurance* (Max_r). It defines the maximum number of relevant events (i.e., events in `Event Idx`) for a node within the batch. With higher Max_r, the nodes will be affected/used more before updating. The Adaptive Batch Sensor will analyze and control this parameter, as specified in Section 4.4. The TG-Diffuser increases the batch size under the limit of Max_r in three steps, as illustrated in Figure 7(b) and specified in Algorithm 3: First, at

Algorithm 3: Lookup Last Tolerable Event

Input : D: Node-event dependency Table; P: Node's current latest event ptr; N Number of nodes; Max_r maximum revisit endurance.
Output: K: the last tolerable event index for current batch.

Initialize $K = MAX\_INT$;
// Loop Parallel: get last tolerable event
for $n = node_0, node_1, \cdots, node_{N-1}$ **do**
  $D_n = D[n]$;
  $cur\_ptr = P[n]$; // cur_ptr points to node's latest relevant event in its event index
  $max\_perm\_ptr = min(cur\_ptr + Max_r, D_n.length - 1)$;
  $e_n = D_n[max\_perm\_ptr]$;
  $K = min(e_n, K)$;
// Loop Parallel: update nodes' event pointers
for $n = node_0, node_1, \cdots, node_{N-1}$ **do**
  if $D[n][P[n]] < K$ **then**
    $P[n]++$;
return K;

each node, the TG-Diffuser begins with the node's earliest unprocessed events indicated by the node's current event pointer. It increments this pointer by Max_r to determine the node's last tolerable event, at which this node is involved too excessively in the current batch and requires an update. For instance, as shown in the figure, for $node_0$, it starts from $e(0)$ and gets the last event at $e(8)$, meaning that the $node_0$ is affected by too many events (i.e., $Max_r = 4$) and should be updated at $e(8)$. For those events that have all their events bypassed, we set their result as MAX_INT to indicate that all remaining events in their entries can be processed safely. Second, the TG-Diffuser reduces the last event indices from different nodes and gets the smallest index among them. Intuitively, we would like to have a batch processed once a node cannot tolerate more related events. Using the same example, as shown in the figure, the batch's last event is $e(8)$ since any events after this one may use intolerably expired information on $node_1$ or $node_2$. Lastly, the TG-Diffuser returns the last event index as depicted in Algorithm 3 and updates all nodes' last event pointers, making them point to the next unprocessed event within the related nodes.

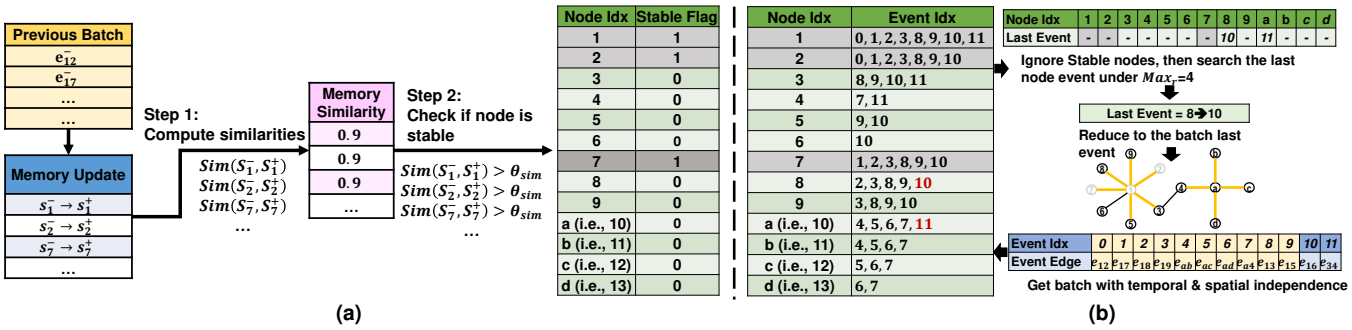


Figure 8. The workflow of SG-Filter: (a) Update node stable flag and (b) Guide TG-Diffuser to ignore stable nodes while looking up the last tolerable event.

Chunk-based Optimization for Large-Scale Graphs. While TG-Diffuser has low overhead for moderate-sized dynamic graphs, its overhead can increase with larger graphs (as we quantified on large-scale graphs in Section 5.5). To address this, we propose a chunk-based table-building strategy to reduce the overheads and enhance scalability. For large event sequences (e.g., billions of events) in large-scale graphs, we apply a two-step divide-and-conquer approach: (1) We split the sequence into smaller chunks, each containing a subset of consecutive events; and (2) we build tables independently for each chunk, considering only within-chunk dependencies. The final event in each chunk serves as a boundary to limit dependencies. Training is performed sequentially across chunks, ensuring node memories update in the correct order. This optimization boosts TG-Diffuser efficiency in two ways. First, processing smaller chunks improves data locality, significantly reducing cache misses and consequent memory latencies during the table building process. Specifically, instead of repeatedly accessing large memory sections that exceed cache capacity, processing smaller chunks allows each thread (i.e., node) to work with data that is more likely to remain in the cache. Second, by pipelining table building and training, training in each chunk can start as soon as its table is ready and training in the previous chunk has finished. This approach speeds up overall processing by pipelining and overlapping table building with training tasks.

4.3 Similarity-Aware Graph Filter

As discussed in Section 3.3, if a node exhibits stable memory—meaning its memory status changes minimally over time—events associated with this node will consistently retrieve similar input memories. Consequently, we can neglect these stabilized nodes when assessing dependencies within the input sequence. The Similarity-Aware Graph Filter (SG-Filter) is designed to identify temporal independence among node memories, thereby mitigating unnecessary constraints imposed by these temporal dependencies.

The operation of the SG-Filter unfolds in two main steps, as depicted in Figure 8. First, the SG-Filter maintains and updates node stable flags once the node memory is updated in two steps: At *step 1*, it calculates the similarities between nodes' memories before/after their updates. As illustrated in Figure 8(a), when memories in $node_1, node_2, node_7$ are updated from s_1^-, s_2^-, s_7^- to s_1^+, s_2^+, s_7^+ due to input events e_{12}^-, e_{17}^-, the SG-Filter computes the similarities between s_1^- and s_1^+, s_2^- and s_2^+, s_7^- and s_7^+, respectively. Next, at *step 2*, it compares the similarities with predefined threshold θ_{sim}—if the similarity is higher than the threshold, the node is considered stable, and vice versa—and updates the nodes' flags accordingly. For instance, $node_1, node_2, node_7$ are considered stable since $sim(s_1^-, s_1^+), sim(s_2^-, s_2^+), sim(s_7^-, s_7^+)$ are higher than $\theta_{sim} = 0.9$, and their flags are updated to 1. Lastly, based on the node stable flag, the SG-Filter guides the TG-Diffuser to ignore the stable nodes straightforwardly: It signifies the stable node indices to TG-Diffuser, and the TG-Diffuser will no longer look up the last event in those stable nodes' entires. For instance, as illustrated in Figure 8(b), the TG-Diffuser will ignore their entries if $node_1, node_2, node_7$ is specified as stable. Consequently, the barriers (i.e., Node last events as 8) posed by $node_1$ and $node_2$ no longer exist, and we can further expand batch size from 8 to 10.

4.4 Adaptive Batch Sensor

As discussed in Section 4.2, we employ the concept of Maximum Revisit Endurance (Max_r) to quantitatively control how many events a node can tolerate before its updating. A higher Max_r value allows nodes to participate in more events per batch, increasing the risk of incorporating outdated information. Consequently, a high Max_r potentially leads to more broken input dependencies. To ensure the TG-Diffuser operates within thresholds that maintain training efficiency without sacrificing the quality of the input data as the originally defined small batch sizes, we introduce the Adaptive Batch Sensor (ABS), a profile-based module, to gather statistics on Max_r using the original batch sizes.

Maximum Endurance Profiling. As depicted in Figure 9, the ABS begins by segmenting the input sequence into batches using a predefined small batch size. It then randomly

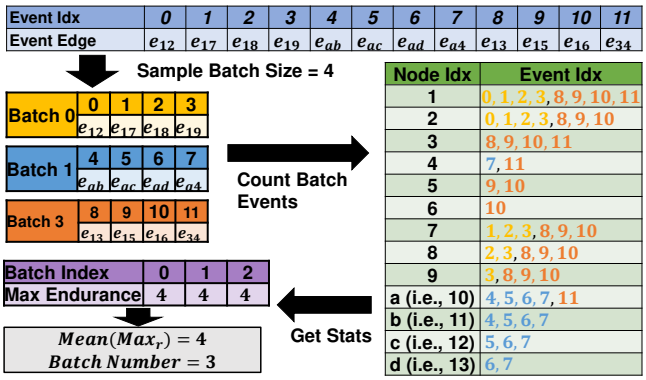


Figure 9. The workflow of maximum endurance profiling .

selects several batches to gather statistics on Max_r through a two-step process. Initially, for each batch, the ABS counts the number of relevant events for each node and identifies the highest count, which is termed `Max Endurance`. For example, in Batch 1, nodes $node_4$, $node_a$, $node_b$, $node_c$, and $node_d$ are involved in 1, 4, 4, 3, and 2 relevant events respectively, resulting in a `Max Endurance` of 4. Subsequently, ABS compares statistics from various batches, calculating the maximum, mean, and minimum values of `Max Endurance` and counting the batch number under the small batch settings. These statistics are then communicated to the TG-Diffuser to establish the upper limits on node involvement in each batch.

Logarithmic-Decaying Endurance. During training, the TG-Diffuser employs a logarithmic decaying strategy to subtly tune Max_r between the max and minimum values of `Max Endurance` configured by the ABS. In particular, ABS decays MAX_r once convergence halts (training loss stops decreasing for ten batches) as smaller batches can provide fresher node memories, aiding convergence. When triggered, the decaying step size is decided by the batch index—To avoid introducing errors in early timestamps, we adopt larger reduction steps in early batches and smaller reductions in later batches. The resulting new MAX_r will be sent to the TG-Diffuser to control how many events one node can endure before its update. Upon receiving the value, the TG-Diffuser will use the newly updated value to look up the last tolerable event for each node. The adjustment of Max_r occurs in three steps: Initially, Max_r is set to two times the mean value of `Max Endurance` (i.e., mr_{mean}). We empirically set the value for two reasons: the maximum value is too aggressive due to potential information loss, while the mean can be too conservative if the batch size is insufficient. Also, we cap Max_r all the time to ensure it is within the range of the analyzed maximum(i.e., mr_{max}) and minimum(i.e., mr_{min}). Next, the ABS further monitors training loss throughout the epoch, and periodic checks determine if there is no reduction in loss. If the training loss does not decline, we reduce Max_r toward the minimum value of `Max Endurance` through a logarithmic decay, a method commonly used in the deep learning domain [12, 27, 35]. Specifically, for the batch i, the TG-Diffuser will get the Max_r following Equation 5,

$$\begin{cases} Max_r = 2 \times mr_{mean} - \alpha \times log(\frac{i}{\beta} + 1) & (5) \\ \alpha = \frac{mr_{min} \times mr_{min}}{mr_{max}}, \beta = \frac{B}{\alpha} & (6) \\ Max_r = max(mr_{max}, min(mr_{min}, Max_r)) & (7) \end{cases}$$

in which mr_{mean}, mr_{min} and mr_{max} refer to the maximum, mean, and minimum values of the `Max Endurance`, respectively; and B refers to the batch numbers under preset batch sizes.

5 Evaluation

5.1 Methodology

Models. We evaluate Cascade using five recent TGNN models. Specifically, we include the following CTDG-based TGNN models: (1) JODIE [21] applies a normal Recurrent-Neural-Network(RNN) [34] to update node memories and uses a time-decay coefficient to scale them before classification. (2) TGN [31] uses a GRU [7] to update node memories and uses a Graph Attention Network (GAT) [39] to embed node memories. (3) APNN [43] adopts an asynchronous mailbox to store and update node memories and then directly use memories for predictions. To assess our approach's adaptability to DTDG-based models, we also include the following two DTDG-based TGNN models: (2) DySAT [33] uses RNN to update and combine node memories from different time graph snap-shots. (3) TGAT [47] adopts positional encoding to abstract edge temporal information and uses an attention-based module to collect messages from nodes' neighbors during memory updating. We follow the model configuration used in TGL [55] as shown in Table 1.

Table 1. Details of TGNN models.

	Sample Message	Memory Update	Node Embedding
JODIE	most recent (num = 1)	RNN (out size=100)	Identity (out size=100)
TGN	most recent (num = 1)	GRU (out size=100)	GAT (out size=100)
APAN	most recent (num = 10)	Transformer (out size=100)	Identity (out size=100)
DySAT	uniform (num = 10)	GAT (out size=100)	RNN (out size=100)
TGAT	uniform (num = 10)	Identity (out size=100)	2-layers GAT (out size=100)

Datasets. We use the following five real-world datasets to evaluate Cascade: (1) Wikipedia (`WIKI`), (2) Reddit (`REDDIT`), (3) MOOC student drop-out (`MOOC`) [21] are relatively small-scale datasets; (4) Wikipedia Talk network (`WIKI-TALK`) [23] and (5) Stack overflow temporal network (`SX-FULL`) [25] are large-scale datasets with millions nodes and events. We also include two billion-edge graph datasets to evaluate the scalability of Cascade: (1) `GDELT` [55] is originated from the

Event Database in GDELT 2.0, containing 0.2 billion events as news and articles. (2) MAG [55] is a paper citation graph containing 1.3 billion events as citations between papers. The statistics of the datasets are shown in Table 2. For datasets with no edge features, we randomly generate edge features following the setup in TGL [55] (denoted by *). For large-scale graphs with millions of nodes and edges, we set the edge feature size to 32 to avoid OOM issues on GPUs.

Table 2. Statistics of Datasets.

	# Nodes	# Edges	# Edge Features
WIKI	9,227	157,474	172
REDDIT	11,000	672,447	172
MOOC	7,047	411,749	128*
WIKI-TALK	2,394,385	5,021,410	32*
SX-FULL	2,601,977	63,497,050	32*
GDELT	16,682	191,290,882	186
MAG	121,751,665	1,297,748,926	32*

Platforms and Implementations. We run our experiments on a server with an AMD EPYC 7742 64-Core Processor CPU and a Nvidia A100 40GB GPU with CUDA 11.6 [28]. Our experiment compares the following approaches:

- **TGL (baseline)** [55] is a state-of-the-art TGNN training framework that achieves better training efficiency and accuracy than the vanilla version of the TGNN models. It adopts a parallel sampler to speed up the sampling process in TGNN and introduces a random batch shuffling strategy to improve the resulting models' losses.
- **TGLite** [45] is a state-of-the-art TGNN framework that provides core abstractions and building blocks for implementing TGNN. It speeds up TGNN training by integrating several optimization schemes and providing lightweight implementations of TGNN models.
- **Cascade.** We implement TG-Diffuser and ABS using C++ to parallel the table building and last event looking up. The SG-Filter is implemented by Python to directly leverage the parallel matrix operation in PyTorch [30]. We adopt the same sampling and model implementation as the baselines since these components are orthogonal to our designs.
- **Cascade-Lite.** In this version, we equip Cascade with optimized TGNN model implementation as the TGLite to evaluate its effectiveness in collaborating with various existing TGNN frameworks.

In addition, we compare Cascade with recent TGNN and Dynamic-GNN training frameworks that adopt dynamic parallelization schemes as described below:

- **NeutronStream** [5] is a DGNN training framework designed for windowed Dynamic Graph Neural Network (DGNN) training. It builds a dependency graph for the input events sequence, and then sequentially processes dependent events and only allows parallelizing events without dependence.
- **ETC** [11] is a TGNN training framework that uses an information-loss-bounded batching scheme to enlarge batch sizes without increasing information loss, which quantifies how many times nodes in the batch are expected to be updated. Additionally, it employs a pipelined data access strategy to improve data transfer efficiency between the CPU and GPUs during TGNN training.

Training Setup. We train the models for the link prediction tasks following setup in the baseline [55]. For WIKI, REDDIT, and MOOC, we train TGNN models with 100 epochs. For WIKI-TALK and SX-FULL, we train TGNN models with 50 epochs. We use a batch size of 900 for training the baselines, as the preset small batch size for ABS in Cascade, and for evaluating all resulting models. We set the similarity threshold θ_{sim} in SG-Filter to 0.9 (more discussion in Section 5.3) and CPU thread numbers in TG-Diffuser and ABS as 32. We set the adaptive decaying period to be 20 for all benchmarks (i.e., ABS makes decisions after each of the 20 batches).

5.2 Overall Performance

Speedup. To facilitate visualizing, we normalize all results by the baseline performances. As shown in Figure 10, Cascade achieves 1.3× to 5.1× (averagely 2.3×) speedups over the baseline. Moreover, Cascade-Lite achieves 1.2× to 5.0× (averagely 2.3×) speedups over the TGLite, indicating the potential of adapting Cascade to various existing TGNN frameworks. The acceleration is particularly notable in sparser dynamic graphs. Specifically, in WIKI, WIKI-TALK, and SX-FULL, which have average degrees of approximately 17.5, 2.5, and 24.4 respectively, Cascade achieves average speedups of 2.5×, 2.4×, and 3.0× over TGL. In comparison, in REDDIT and MOOC, whose average degrees are 61.1 and 58.4, Cascade achieves 1.8× and 1.7× average speedups over the baseline, respectively. This is because events in sparser graphs are more likely to be spatial independent of each other due to weaker connectivity among nodes. Regarding the models, Cascade demonstrates higher speedups for TGNN models that depend less on neighboring nodes for node updates or computing embeddings. Within CTDG-based models, TGN and JODIE, which update node memories using the most recent message, receive average speedups of 2.4× and 2.5×, respectively. In contrast, APAN, which utilizes the ten most recent messages, achieves a lower average speedup of 1.7×. For DTDG-based TGNNs, DySAT, which employs a single-layer GAT, achieves a 3.1× average speedup compared to 1.7× for TGAT, which uses a two-layer GAT that embeds more neighborhoods. The potential reason is that those slower models cost more time on the neighbor sampling step. With larger batches, the sampling step takes longer, compromising the benefit of higher parallelism and fewer iterations. Cascade effectively increases batch sizes in diverse benchmarks—as showcased in Figure 12(a), it increases the batch size from 900 to 4200 across WIKI, REDDIT and REDDIT-TALK.

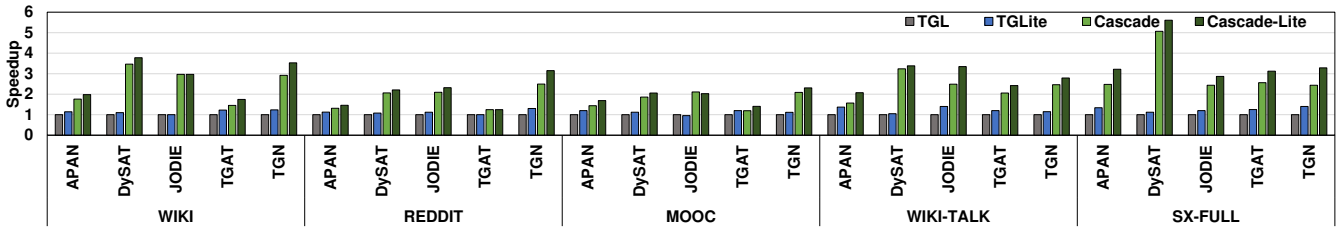


Figure 10. Training speed-ups introduced by Cascade and Cascade-Lite compared to baseline (TGL) and TGLite.

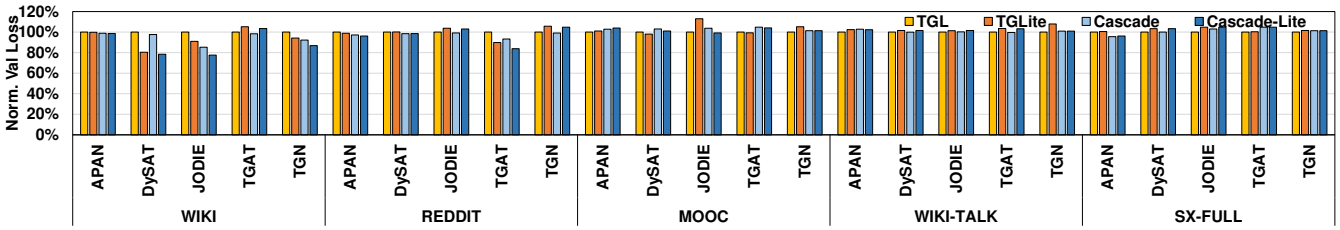


Figure 11. Validation losses (normalized to baseline) of TGNN models trained in Cascade and Cascade-Lite.

Model Losses. Unlike simply increasing batch sizes, Cascade accelerates the training without worsening the resulting models' performances. As shown in Figure 11, on average, models trained by Cascade and Cascade-Lite are validated to have 99.4% and 97.9% average losses compared to the baseline and TGLite, respectively. In those datasets with real-world edge features (i.e., `WIKI` and `REDDIT`), Cascade decreases the average losses of the resulting model by 5.5% (up to 15%) and 2.5% (up to 7.3%), respectively. To investigate its capabilities to maintain model performances, we train the TGNNs with baseline while increasing their batch sizes to the same as the average batch size in Cascade. We compared the resulting model losses with those in Cascade, as shown in figure 12(b), using larger batches (i.e., TGL-LB) causes 1% to 83% loss increases than using batch size of 900. In contrast, Cascade introduces 1% to 15% loss decreases than the baseline, leading to ~80% accuracy improvement over large batch sizes.

5.3 Optimization Analysis

To further investigate the effectiveness of the TG-Diffuser and SG-Filter, we evaluate the performances when Cascade only enables the TG-Diffuser and ABS without SG-Filter (referred to as Cascade-TB), on `WIKI` and `REDDIT`. As shown in Figure 12(c), Cascade-TB achieves 1.8× speedup over the baseline by equipping TG-Diffuser and ABS. Similar to overall performance, it benefits more on relatively sparser dynamic graphs—the average speedup of Cascade-RB is 1.9× on `WIKI`, and is lower as 1.7× on `REDDIT`. The speedup is more significant for those models relying less on neighbors. For instance, on JODIE and TGN, Cascade achieves 2.3× and 2.5× average speedup than in APAN, which is 1.2×. With the help of SG-Filter, the average speedup in Cascadeis further boosted to 2.2×. Compared to the TG-Diffuser, the SG-Filter can further boost the performance of models that use more neighbors for their computing. For instance, in APAN, Cascade achieves 1.7× speed up compared to 1.1× in Cascade. This is because APAN uniformly samples more messages from the past instead of using the most recent message; there might be more overlapping in these sampled past messages, which leads to similar inputs for memory updating. Consequently, there is a higher possibility of having temporal independent node memories.

In terms of the model losses, as shown in Figure 12(d), Cascade-TB is capable of maintaining validation losses and, in some cases, even reduces more loss than the Cascade. For instance, in JODIE, the Cascade reduces model losses to 84% and 97%; in comparison, Cascade reduces losses to 85% and 99%. This is because the stable detecting scheme in SG-Filter decides node stable status based on past updates and may mispredict in some cases. To better understand the potential impact of SG-Filter, we measure Cascade under different choices of similarity thresholds. As shown in Figure 13(a), using lower similarity can improve latencies yet harm the model accuracy. For instance, while using $\theta_{sim} = 0.85$ achieves 2.7× average speedup, it increases loss by 8%. Conversely, using higher similarity can help maintain model accuracy yet achieve fewer benefits in latencies. For instance, using $\theta_{sim} = 0.95$ causes no loss drops yet lowers the speedup to 2×.

5.4 Overhead Analysis

To measure the impact of these overheads, we investigate the time and space breakdown of Cascade under datasets `WIKI`, `REDDIT`, and `WIKI-TALK`. As shown in Figure 13(b), on average, Cascade causes 17% latency overhead in these

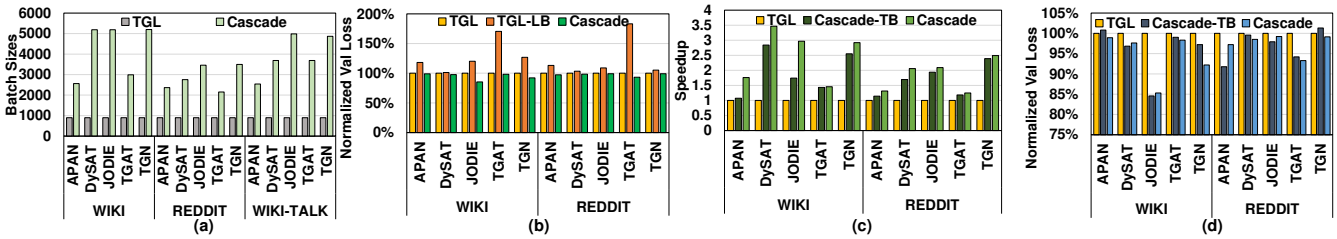


Figure 12. (a) Batch sizes of TGNNs in Cascade compared to the baseline. (b) Validation losses of TGNNs trained by Cascade and TGL-LB (baselines with larger batches)—results are normalized by the losses in the baseline. (c) Speedups of Cascade-TB and Cascade over baseline. (d) Validation losses of TGNNs trained in Cascade-TB and Cascade compared to baseline.

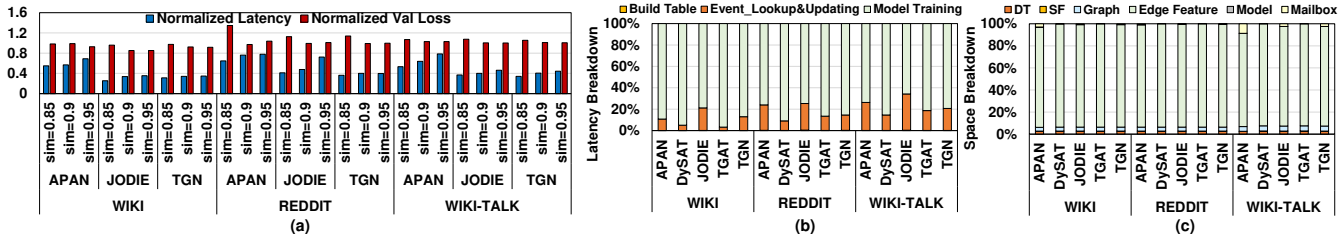


Figure 13. (a) Latency and validation loss of Cascade under different similarity threshold for SG-Filter (i.e., θ_{sim}). (b) The latency breakdown in Cascade. (c) The space consumption ratio in Cascade (DT as dependency Table, SF as node stable flag).

moderate-sized graphs, which is far less than the model training time—compared to the original model training latency in baselines, the overhead is less than 10%. Building the dependency table causes ignorable overheads, which are as low as 0.1% on average, as it is only conducted once throughout the training process. In contrast, the batching `Event_lookup` takes a heavier part of the overheads, which takes 16% latency on average. This is because we need to compare the last event for each node and then update their pointers in each batch. The node stable flag checking and updating causes ignorable overhead since similarity computing is considerably faster on GPUs. In terms of the space overhead, as shown in Figure 13(c), the dependency table (DF) and the node stable flag (SF) takes less than 3% space overhead in total—even in large graphs such as `WIKI_TALK`, they consume much less space than the `edge Feature`, which takes the majority of the space consumption. Adaptive batching (ABS) introduces two minimal overheads: (i) profiling costs for detecting max, min, and average revisit limits, which are negligible (<1% in preprocessing) as they involve sampling a few batches (50 in our implementation) and checking node-related events without computation; and (ii) reconfiguration costs for calculating and assigning new Max_r, which are minimal (a few cycles) as they require only a few scalar operations.

5.5 Scalability on Large-scale Graphs

We compared Cascade to the baseline on two billion-event datasets: `GDELT` and `MAG`, and report the results in Figure 14. For `MAG` dataset, APAN throws out-of-memory (OOM) errors in both baseline and Cascade as it stores the ten most recent neighbors for each node. From the Figure 14(a), Cascade (second bar) achieves average speedups of 1.7× on `GDELT` and 1.3× on `MAG` over the baseline. As shown in Figure 14(b), the resulting models have validation losses of 97.9% and 99.0% compared to the baseline, respectively. These results demonstrate that Cascade remains effective on large graphs.

However, one can observe that performance gain is lower on large-scale graphs than moderate-sized graphs (i.e., 1.7× on `GDELT` and 1.3× on `MAG` compared to an average of 2.3× in moderate-sized graphs in Figure 10). The reason is that the pre-processing overheads significantly increase in large-scale graphs. We report the latency breakdown in Figure 14(c). As one can observe, the pro-processing overheads can account for 36.6% of the entire execution time in large graphs (which is less than 1% in moderate-sized graphs).

To improve the scalability of Cascade on large-scale graphs, we propose an optimization by enabling chunk-based preprocessing described in detail in Section 4.2. In our experiment, we set the chunk size as one million events. We report the speedup, validation loss, and pre-processing overheads in Figure 14 labeled as Cascade_EX. Specifically, Cascade_EX achieves speedups of 2× on `GDELT` and 1.7× on `MAG` without increasing validation losses. This is higher than the speedups (i.e., 1.7× on `GDELT` and 1.3× on `MAG`) without chunk-based preprocessing. The reason is that this optimization significantly reduces the cache misses in table building and is able to pipeline and overlap the table building with model training, as we elaborated in Section 4.2. Results show that Cascade_EX reduces the preprocessing overhead by an average of 35% in two large-scale graphs.

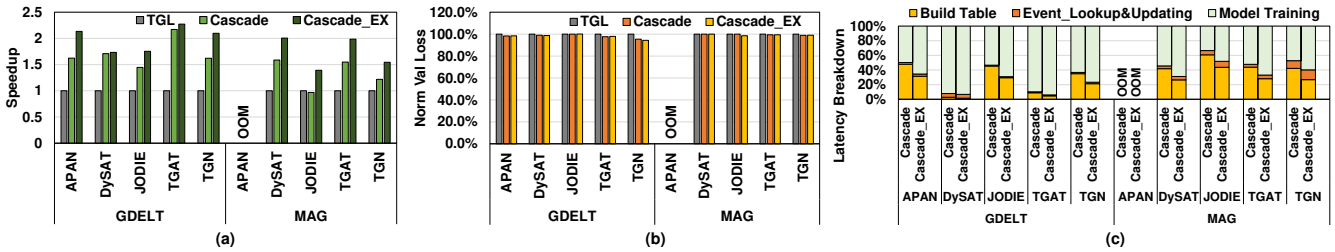


Figure 14. The evaluation results on GDELT and MAG including (a) The speedup and (b) resulting TGNNs' validation losses in Cascade and its optimized version with chunk_based optimization Cascade_EX over the baseline. (c) The latency breakdowns.

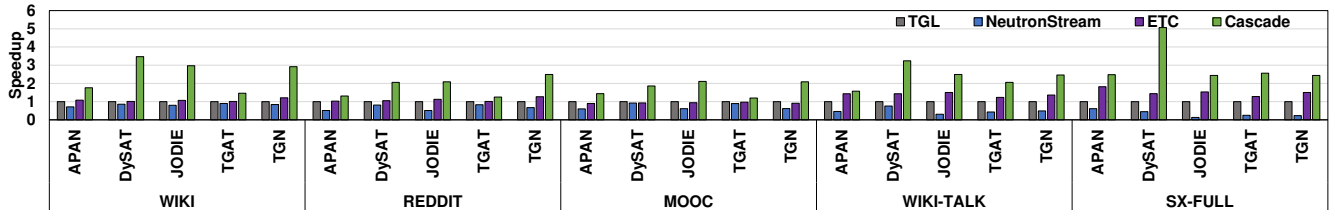


Figure 15. Training speed-ups introduced by Cascade, NeutronStream, ETC compared to baseline (TGL).

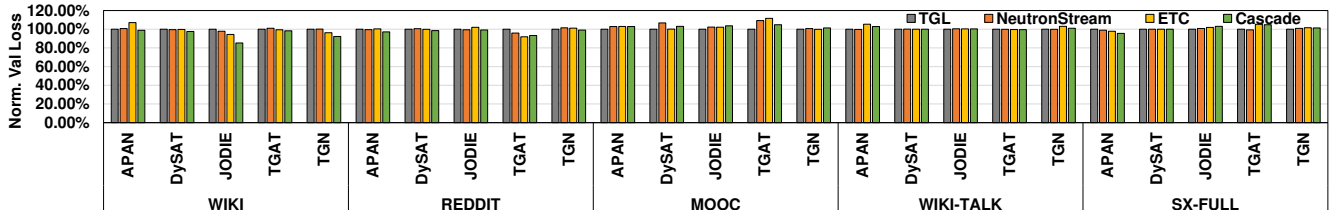


Figure 16. Validation losses (normalized to baseline) of TGNN models trained in by Cascade, NeutronStream, ETC.

5.6 Comparison with Prior Dynamic Batching

We compare Cascade with ETC [11] and NeutronStream [5], and report the results in Figure 15. As all compared approaches increase batch sizes from a basic batch size, we set the basic batch size for all approaches as 900 following the baseline (i.e., TGL [55]) configuration in Section 5.2. This size strikes a balance between accuracy and efficiency under fixed-sized batching.

Comparison with NeutronStream: We use the scheme in NeutronStream to check if the subsequent events are independent of existing events within the batch, then batch those independent ones into the current batch. In contrast, Cascade also employs the same base batch size and increases batch by batching subsequent events if they are related to less-frequently involved nodes or stable nodes. Our results show that Cascade achieves a 3.8× improvement over NeutronStream, with better validation losses on average. The performance gain is mainly because Cascade yields larger batch sizes and, therefore, more parallelism than NeutronStream. It is also worth mentioning that NeutronStream generally performs worse than the baseline as they spend a lot of time on constructing dependency graphs yet hardly increase batch sizes. Hence, even if we start with larger basic batch sizes that are larger than 900, it can hardly bring more parallelism than the baseline and may introduce more significant overhead.

Comparison with ETC: For each base batch, ETC expands by adding subsequent events as long as they do not increase the information loss (i.e., the total number of expected node updates) beyond a specified threshold. Specifically, to achieve comparable performances as using small batches, it automatically detects the information loss in the pre-defined small batch size (i.e., batch size = 900 in our experiments as the baseline). Then, it uses the upper bound of the detected information loss as the threshold to ensure that the information loss of the enlarged batches is not worse than the baseline. Our results show that Cascade achieves a 1.9× improvement over ETC, with better validation losses average on average. Similarly, the performance gains come from larger batch sizes: While Cascade increases the average batch size to 4255, ETC increases the batch size from 900 to an average of 1123. The improvements in ETC are limited since it stops expanding a batch once the information loss (i.e., the total number of expected node updates) reaches the

per-batch threshold. This can lead to situations in which, when specific high-degree nodes in the current batch are expected to have frequent updates and trigger the batching limit, subsequent events cannot be further batched since these high-degree nodes have raised the total number of expected updates beyond the threshold. In contrast, Cascade tracks the endurance score for each node (i.e., look up the last tolerable event in each node independently). Therefore, even if some high-degree nodes are already heavily involved in the current batch, they only raise their own endurance scores, but not those endurance scores in other nodes. As a result, Cascade can still include more subsequent events if they are related to those fresher nodes since the nodes still have low endurance scores.

6 Related Work

Although extensive studies have been conducted on accelerating GNN training [2, 14, 26, 36, 37, 41, 44, 49, 51, 53], they fail short in addressing the unique challenges in TGNN training due to their distinct computing diagrams. While some of the recent studies on TGNN training focus on DTDG graphs [10, 13, 32, 40], these approaches are tailored to DTDG contexts, where graph snapshots fixedly determine batches and whole-graph update are conducted. Noticing the unique challenge in CTDG-based TGNN training, TGL [55] introduces a parallel sampler to speed the sampling process for CTDG and proposes a chunk scheduling approach to increase the resulting models' accuracy. On top of TGL, DistTGL [55] further proposes heuristic-guided parallelism to speed up the distributed TGNN training. More recently, TGLite [45] provides core abstractions and building blocks for implementing optimized TGNNs. Additionally, ETC [11] and Neutronstream [5] explore adopting dynamic batching in CTDG-related training. However, none of the prior methods adaptively quantify and leverage the spatial and temporal relationships between events to dynamically increase batch sizes, thereby limiting their ability to enhance parallelism without significant information loss.

7 Conclusion

In this work, we proposed an efficient TGNN training framework, Cascade, to speed up temporal graph neural network (TGNN) training by adaptively increasing training batch sizes without breaking input dependency. Experimental results show Cascade can achieve up to 5.1× speedup over the state-of-the-art TGNN training frameworks.

Acknowledgments

The authors would like to thank the anonymous ASPLOS reviewers for their constructive feedback and suggestions. This work is supported in part by NSF grants #2154973, #2334628, and #2312157.

References

[1] Luca Belli, Sofia Ira Ktena, Alykhan Tejani, Alexandre Lung-Yut-Fon, Frank Portman, Xiao Zhu, Yuanpu Xie, Akshay Gupta, Michael Bronstein, Amra Delić, et al. Privacy-preserving recommender systems challenge on twitter's home timeline. *arXiv preprint arXiv:2004.13715*, 2020.

[2] Zhenkun Cai, Xiao Yan, Yidi Wu, Kaihao Ma, James Cheng, and Fan Yu. Dgcl: An efficient communication library for distributed gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 130–144, 2021.

[3] Augustin Chaintreau, Pan Hui, Jon Crowcroft, Christophe Diot, Richard Gass, and James Scott. Impact of human mobility on opportunistic forwarding algorithms. *IEEE Transactions on Mobile Computing*, 6(6):606–620, 2007.

[4] Rohit Chandra. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[5] Chaoyi Chen, Dechao Gao, Yanfeng Zhang, Qiange Wang, Zhenbo Fu, Xuecang Zhang, Junhua Zhu, Yu Gu, and Ge Yu. Neutronstream: A dynamic gnn training framework with sliding window for graph streams. *Proceedings of the VLDB Endowment*, 17(3):455–468, 2023.

[6] Jinyin Chen, Jian Zhang, Xuanheng Xu, Chenbo Fu, Dan Zhang, Qingpeng Zhang, and Qi Xuan. E-lstm-d: A deep learning framework for dynamic network link prediction. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 51(6):3699–3712, 2019.

[7] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[8] Yue Dai, Xulong Tang, and Youtao Zhang. Flexgm: An adaptive runtime system to accelerate graph matching networks on gpus. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pages 348–356. IEEE, 2023.

[9] Yue Dai, Youtao Zhang, and Xulong Tang. Cegma: Coordinated elastic graph matching acceleration for graph matching networks. In *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 584–597. IEEE, 2023.

[10] Kaihua Fu, Quan Chen, Yuzhuo Yang, Jiuchen Shi, Chao Li, and Minyi Guo. Blad: Adaptive load balanced scheduling and operator overlap pipeline for accelerating the dynamic gnn training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2023.

[11] Shihong Gao, Yiming Li, Yanyan Shen, Yingxia Shao, and Lei Chen. Etc: Efficient training of temporal graph neural networks over large-scale dynamic graphs. *Proceedings of the VLDB Endowment*, 17(5):1060–1072, 2024.

[12] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT Press, 2016.

[13] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. Dynagraph: dynamic graph neural networks at scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–10, 2022.

[14] Deniz Gurevin, Caiwen Ding, and Omer Khan. Exploiting intrinsic redundancies in dynamic graph neural networks for processing efficiency. *IEEE Computer Architecture Letters*, 2023.

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[16] Zhen Jia, Abdalghani Abujabal, Rishiraj Saha Roy, Jannik Strötgen, and Gerhard Weikum. Tequila: Temporal question answering over knowledge bases. In *Proceedings of the 27th ACM international conference on information and knowledge management*, pages 1807–1810, 2018.

[17] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for

dynamic graphs: A survey. *The Journal of Machine Learning Research*, 21(1):2648–2720, 2020.

[18] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[19] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[20] Srijan Kumar, William L Hamilton, Jure Leskovec, and Dan Jurafsky. Community interaction and conflict on the web. In *Proceedings of the 2018 world wide web conference*, pages 933–943, 2018.

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

[22] Julien Leblay and Melisachew Wudage Chekol. Deriving validity time in knowledge graph. In *Companion Proceedings of the The Web Conference 2018*, pages 1771–1776, 2018.

[23] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Signed networks in social media. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 1361–1370, 2010.

[24] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, pages 177–187, 2005.

[25] Jure Leskovec and Rok Sosič. Snap: A general-purpose network analysis and graph-mining library. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 8(1):1–20, 2016.

[26] Zhiqi Lin, Cheng Li, Youshan Miao, Yunxin Liu, and Yinlong Xu. Pagraph: Scaling gnn training on large graphs via computation-aware caching. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 401–415, 2020.

[27] Grégoire Montavon, Geneviève Orr, and Klaus-Robert Müller. *Neural networks: tricks of the trade*, volume 7700. springer, 2012.

[28] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, 2008.

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

[30] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[31] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *arXiv preprint arXiv:2006.10637*, 2020.

[32] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, Guzman Lopez, Nicolas Collignon, et al. Pytorch geometric temporal: Spatiotemporal signal processing with neural machine learning models. In *Proceedings of the 30th ACM international conference on information & knowledge management*, pages 4564–4573, 2021.

[33] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 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*, pages 519–527, 2020.

[34] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

[35] Noam Shazeer and Mitchell Stern. Adafactor: Adaptive learning rates with sublinear memory cost. In *International Conference on Machine Learning*, pages 4596–4604. PMLR, 2018.

[36] Nishil Talati, Di Jin, Haojie Ye, Ajay Brahmakshatriya, Ganesh Dasika, Saman Amarasinghe, Trevor Mudge, Danai Koutra, and Ronald Dreslinski. A deep dive into understanding the random walk-based temporal graph learning. In *2021 IEEE International Symposium on Workload Characterization (IISWC)*, pages 87–100. IEEE, 2021.

[37] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, et al. Dorylus: Affordable, scalable, and accurate {GNN} training with distributed {CPU} servers and serverless threads. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 495–514, 2021.

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

[39] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[40] Chunyang Wang, Desen Sun, and Yuebin Bai. Pipad: pipelined and parallel dynamic gnn training on gpus. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 405–418, 2023.

[41] Lei Wang, Qiang Yin, Chao Tian, Jianbang Yang, Rong Chen, Wenyuan Yu, Zihang Yao, and Jingren Zhou. Flexgraph: a flexible and efficient distributed framework for gnn training. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 67–82, 2021.

[42] Xiaoyun Wang, Minhao Cheng, Joe Eaton, Cho-Jui Hsieh, and Felix Wu. Attack graph convolutional networks by adding fake nodes. *arXiv preprint arXiv:1810.10751*, 2018.

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

[44] Yufeng Wang and Charith Mendis. 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*, pages 354–368, 2023.

[45] Yufeng Wang and Charith Mendis. Tglite: A lightweight programming framework for continuous-time temporal graph neural networks. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 1183–1199, 2024.

[46] Yaqi Xia, Zheng Zhang, Hulin Wang, Donglin Yang, Xiaobo Zhou, and Dazhao Cheng. Redundancy-free high-performance dynamic gnn training with hierarchical pipeline parallelism. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*, pages 17–30, 2023.

[47] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *arXiv preprint arXiv:2002.07962*, 2020.

[48] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[49] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. Gnnlab: a factored system for sample-based gnn training over gpus. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 417–434, 2022.

[50] Jiaxuan You, Tianyu Du, and Jure Leskovec. Roland: graph learning framework for dynamic graphs. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 2358–2366, 2022.

[51] Haiyang Yu, Limei Wang, Bokun Wang, Meng Liu, Tianbao Yang, and Shuiwang Ji. Graphfm: Improving large-scale gnn training via feature

momentum. In *International Conference on Machine Learning*, pages 25684–25701. PMLR, 2022.

[52] Yao Zhang, Yun Xiong, Yongxiang Liao, Yiheng Sun, Yucheng Jin, Xuehao Zheng, and Yangyong Zhu. Tiger: Temporal interaction graph embedding with restarts. *arXiv preprint arXiv:2302.06057*, 2023.

[53] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. Bytegnn: efficient graph neural network training at large scale. *Proceedings of the VLDB Endowment*, 15(6):1228–1242, 2022.

[54] Ying Zhong and Chenze Huang. A dynamic graph representation learning based on temporal graph transformer. *Alexandria Engineering Journal*, 63:359–369, 2023.

[55] Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. Tgl: A general framework for temporal gnn training on billion-scale graphs. *arXiv preprint arXiv:2203.14883*, 2022.

[56] Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. Disttgl: Distributed memory-based temporal graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2023.