# NeutronStream: A Dynamic GNN Training Framework with Sliding Window for Graph Streams

Chaoyi Chen
Northeastern
University, China
chenchaoy@
stumail.neu.edu.cn

Dechao Gao
Northeastern
University, China
gaodechao@
stumail.neu.edu.cn

Yanfeng Zhang
Northeastern
University, China
zhangyf@
mail.neu.edu.cn

Qiange Wang
Northeastern
University, China
wangqiange@
stumail.neu.edu.cn

Zhenbo Fu
Northeastern
University, China
fuzhenbo@
stumail.neu.edu.cn

Xuecang Zhang
Huawei Technologies
Co., Ltd.
zhangxuecang@
huawei.com

Junhua Zhu
Huawei Technologies
Co., Ltd.
junhua.zhu@
huawei.com

Yu Gu
Northeastern
University, China
guyu@
mail.neu.edu.cn

Ge Yu
Northeastern
University, China
yuge@
mail.neu.edu.cn

## ABSTRACT

Existing Graph Neural Network (GNN) training frameworks have been designed to help developers easily create performant GNN implementations. However, most existing GNN frameworks assume that the input graphs are static, but ignore that most real-world graphs are constantly evolving. Though many dynamic GNN models have emerged to learn from evolving graphs, the training process of these dynamic GNNs is dramatically different from traditional GNNs in that it captures both the spatial and temporal dependencies of graph updates. This poses new challenges for designing dynamic GNN training frameworks. First, the traditional batched training method fails to capture real-time structural evolution information. Second, the time-dependent nature makes parallel training hard to design. Third, it lacks system supports for users to efficiently implement dynamic GNNs. In this paper, we present NeutronStream, a framework for training dynamic GNN models. NeutronStream abstracts the input dynamic graph into a chronologically updated stream of events and processes the stream with an optimized sliding window to incrementally capture the spatial-temporal dependencies of events. Furthermore, NeutronStream provides a parallel execution engine to tackle the sequential event processing challenge to achieve high performance. NeutronStream also integrates a built-in graph storage structure that supports dynamic updates and provides a set of easy-to-use APIs that allow users to express their dynamic GNNs. Our experimental results demonstrate that, compared to state-of-the-art dynamic GNN implementations, NeutronStream achieves speedups ranging from 1.48X to 5.87X and an average accuracy improvement of 3.97%.

## 1 INTRODUCTION

Graph Neural Networks (GNNs) [6, 18, 26, 30, 52, 60, 62, 64, 69] are a class of deep learning models designed to learn from graph data. GNNs have been widely adopted in various graph applications, including social networks analytics [8, 59], recommendation systems [35, 67], and knowledge graphs [39, 49]. Most of the existing GNN models assume that the input graph is static. However, real-world graphs are inherently dynamic and evolving over time. Recently, many dynamic GNN models [20, 27, 28, 34, 45, 51, 57, 63, 79] are emerged as a promising method for learning from dynamic graphs. These models capture both the *spatial* and *temporal* information, which makes them outperform traditional GNNs in real-time applications, such as real-time fraud detection [57], real-time recommendation [20], and many other tasks.

In dynamic GNNs, the dynamic graph is modeled as a sequence of time-stamped *events*, each event representing a graph update operation. Each event is associated with a timestamp indicating when it occurs and an update type, *e.g.,* an addition/deletion of a node/edge or an update of a node/edge's feature. Dynamic GNNs encode the information of each event into dynamic node embeddings chronologically. The training process of dynamic GNNs is dramatically different from traditional GNNs in that it has to consider the temporal dependency of events. Existing dynamic GNNs [19, 28, 51] are implemented on top of general DNN training frameworks, *e.g.,* Tensorflow [5] and PyTorch [41]. However, the complex spatial-temporal dependencies among events pose new challenges for designing dynamic GNN frameworks.

First, the traditional batched training mode adopted by existing DNN frameworks may fail to capture the real-time structural evolution information. Batched training mode periodically packs new arrival events into a training batch and trains the model using these batches incrementally. However, this method forcibly cuts off the stream and ignores the spatial locality between events in two consecutive batches, which may lead to a decrease in model accuracy. Figure 1(a) illustrates a motivating example on a dynamic social network graph, which contains six consecutive interaction
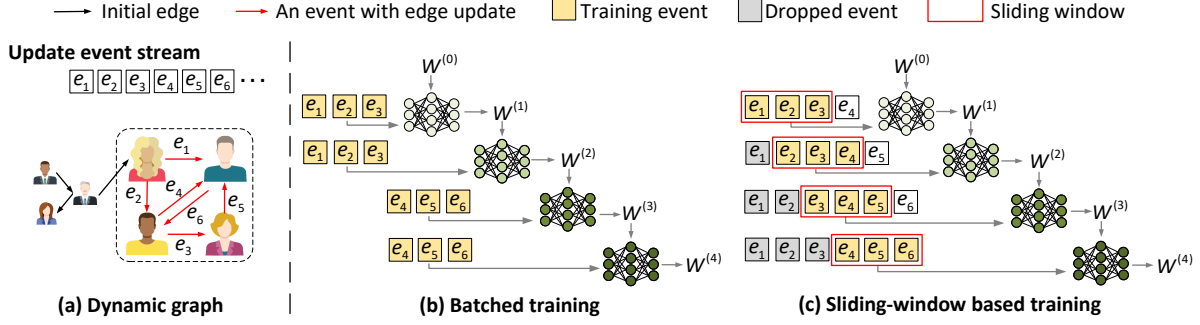
Figure 1: Traditional batched training vs. sliding-window based training for graph streams.

events concentrated in the rectangular area. In the batched training mode, as shown in Figure 1(b), these six events with spatial locality are split into two independent batches. With an initial parameter matrix $W^{(0)}$, the first batch with events $\{e_1, e_2, e_3\}$ is trained in two epochs, and the parameter matrix is updated twice to obtain $W^{(2)}$. The second batch with events $\{e_4, e_5, e_6\}$ is then trained in two epochs to obtain $W^{(4)}$. In this way, the spatial dependency between cross-batch events (*e.g.,* $e_3$ and $e_4$) cannot be captured, thereby impacting the model accuracy.

Second, the sequential nature of time-dependent events makes parallelism optimization hard to design. The existing open-source dynamic GNN implementations [19, 28, 51] are based on the mature DNN or GNN frameworks (*e.g.,* Pytorch [41] and DGL[55]). They adopt vanilla sequential iterative processing, which suffers from sub-optimal performance and can hardly benefit from modern parallel architectures. For example, the DyRep [51] implementation in PyTorch takes 60.45 seconds with a single thread for an epoch on the GitHub dataset [4], but it takes 60.89 seconds with 20 threads, which is even longer than single-thread execution, indicating the lack of parallelization support.

Third, there lacks a user-friendly programming abstraction for implementing dynamic GNNs. The training of dynamic GNNs highly relies on efficient dynamic storage for maintaining dynamic graphs, which is not a standard module in traditional GNN frameworks. Users have to implement their own store to maintain graph topology and multi-versioned embeddings. For example, in the Dyrep [51] model, users have to implement data structures to support the storage of timestamps, the update of graph and node embeddings, and complex graph operations such as accessing a node's neighbor nodes and their embeddings. This requires much redundant coding work and is error-prone. Therefore, it is desirable to have a set of specific APIs for implementing dynamic GNNs and an efficient framework that supports common modules used in dynamic GNN training.

To address the above problems for dynamic GNN training, we make the following contributions.

**Contribution 1.** We propose a new incremental learning mode with a sliding window for training on graph streams. We rely on a sliding window to select consecutive events from the graph stream feeding to the model training. The window is sliding as new update events arrive and the processed stale events outside of the window can be dropped, so that the freshness of the model can be guaranteed and the evolving information and temporal dependencies can be well captured. For example, in Figure 1(c), a window of 3 events as

training samples are trained for one epoch, and then the window slides with a sliding stride of 1. The window size can be adaptively adjusted for capturing the complete spatial dependencies.

**Contribution 2.** We propose a fine-grained event parallel execution scheme. We leverage the observation that each event only affects a small subgraph, and there is no read-write conflict between multiple events if their affected subgraphs do not intersect. Based on this observation, we build a dependency graph analysis method that identifies the events having no node-updating conflicts and processes them in parallel. In this way, training performance can be enhanced through fine-grained event parallelism.

**Contribution 3.** We deliver a dynamic GNN training framework NeutronStream. NeutronStream integrates sliding-window training method and dependency-graph-driven event parallelizing method. Moreover, NeutronStream integrates a built-in graph storage structure that supports dynamic updates and provides a set of easy-to-use APIs that allow users to express their dynamic GNNs with lightweight engineering work.

We evaluate NeutronStream on three popular dynamic GNN models, DyRep, LDG, and DGNN. The experimental results demonstrate that our optimized sliding-window-based training brings 0.46% to 6.31% accuracy improvements. Compared with their open-sourced implementations on PyTorch, NeutronStream achieves speedup ranging from 1.48X to 5.87X.

## 2 PRELIMINARIES

### 2.1 Graph Neural Networks

GNNs operating on graph-structured data aim to capture the topology and feature information simultaneously. Given the input graph $G = (V, E)$ along with the features of all nodes $z^{(0)}$, GNNs stack multiple graph propagation layers to learn a representation for each node. In each layer, GNNs aggregate the neighbor information from the previous layer, following an AGGREGATE-UPDATE computation pattern:

$$h_v^{(l)} = \text{AGGREGATE}\left(\{z_u^{(l-1)} \mid (u,v) \in E\}, W_a^{(l)}\right), \quad (1)$$

$$z_v^{(l)} = \text{UPDATE}\left(z_v^{(l-1)}, h_v^{(l)}, W_u^{(l)}\right), \quad (2)$$

where $z_v^{(l)}$ represents the embedding of node $v$ in the $l$-th layer. The AGGREGATE operation collects and aggregates the embedding of $v$'s incoming neighbors to generate the intermediate aggregation result $h_v^{(l)}$. The UPDATE operation uses $h_v^{(l)}$ and the $(l-1)$-th layer

embedding of $v$ to compute node $v$'s new embedding in the $l$-th layer, where $W_a^{(l)}$ and $W_u^{(l)}$ are model parameters in the $l$-th layer.

## 2.2 Graph Streams

A graph stream is used to abstract the dynamic behavior of a graph over time. It can be formalized as a pair $(G^{(0)}, Evt)$, where $G^{(0)} = (V^{(0)}, E^{(0)})$ is the initial state of a graph at time $t_0$ and $Evt$ is a streaming of graph update events $\{e[0], e[1], e[2], \ldots\}$ ordered by occurrence time. Each update event can be represented as a tuple $(u, v, t, T)$ where $u$ and $v$ are the nodes involved in the event (an edge update involves two nodes and a node update only involves a single node), $t$ is the event occurred timestamp, and $T$ indicates the type of update operations, such as node/edge's addition/deletion and the update of node/edge's feature.

Another commonly used method for modeling a dynamic graph is to represent it as a sequence of static graph snapshots, denoted as $G = (G^{(t_0)}, G^{(t_1)}, G^{(t_2)}, \cdots)$. Each snapshot $G^{(t_i)} = (V^{(t_i)}, E^{(t_i)})$ denote the graph consisting of nodes $V^{(t_i)}$ and edges $E^{(t_i)}$ formed before time $t_i$. Each snapshot $G^{(t_i)}$ is created by incorporating dynamic information $\Delta G^{(t_i)}$ between time intervals $t_{(i-1)}$ and $t_{(i-1)} + \Delta t$ on the previous snapshot $G^{(t_{i-1})}$.

This snapshot-based method cannot represent the complete evolution process of a graph [17, 37, 45, 79]. A snapshot only represents the structure at one particular time period, which can result in the loss of dynamic information. For example, if there are two update operations to add and then delete an edge between two nodes, the snapshot can only record that there is no edge between them, without capturing the dynamic change. In addition, this method cannot capture temporal dependency information within time intervals $\Delta t$ because the temporal dependency of intra-snapshot updates is not maintained. The temporal dependency information is important for modeling and prediction of dynamic graphs [36]. For example, in user-item networks, recent purchases and browsing records are more likely to represent a user's latest favor, which can provide more accurate real-time recommendations for this user [20]. Compared with the snapshot-based method, graph streams can record all dynamic information of a graph and the temporal dependency between them, so graph streams are more general. Therefore, in this work, we focus on dynamic GNNs on graph streams.

## 2.3 Dynamic GNN Training Abstraction

We find that the dynamic GNN models [20, 27, 28, 34, 45, 51, 57, 63, 79] designed for graph streams adopt the traditional batched training mode to learn parameters. Each batch consists of a sequence of historical events $\{e[1], e[2], \ldots, e[n]\}$. The learning process can be abstracted as an event-based dynamic GNN training process, which is summarized in Algorithm 1.

It starts from a initial state with graph $G^{(0)}$ and the corresponding node embedding $Z^{(0)}$ at time $t_0$. During training, the events are processed sequentially chronologically (Line 1-14). First, the algorithm updates the graph topology based on the event nodes and event type of the currently scheduled event (Line 2-3) and gets the event-affected subgraph (Line 4). The computation of an event involves a subgraph, including the event nodes and their neighbors. Neighbors typically consist of one-hop neighbors, while a few algorithms consider neighbors up to $k$-hop neighbors such as JODIE

---

**Algorithm 1:** Event-based Dynamic GNN Training (Per Epoch)

**Input** : Initial parameters $W = \{W_a, W_u, W_p\}$ for aggregation, update, and propagation, respectively, initial graph $G^{(0)} = (V^{(0)}, E^{(0)})$, initial node embeddings $Z^{(0)} = \{z_v^{(0)} \mid v \in V^{(0)}\}$, a sequence of events $\{e[1], e[2], \ldots, e[n]\}$ where $e[i] = (u, v, t^i, T)$, and a set of training sample labels $Y$

**Output**: Updated parameters $W' = \{W_a', W_u', W_p'\}$, updated graph $G^{(n)}$, and updated node embeddings $Z^{(n)}$

  **Forward:**
1  **for** $e[i] \leftarrow e[1]$ **to** $e[n]$ **do**
2  $\quad \{u, v, t^i, T\} = parse(e[i])$;
3  $\quad G^{(i)} = \text{UpdateGraph}(G^{(i-1)}, (u, v), T)$;
4  $\quad \mathcal{E}_e(V_e, E_e) = \text{GetSubgraph}(G^{(i)}, (u, v))$;
5  $\quad \Delta t_u = t^i - t_u; \Delta t_v = t^i - t_v; t_u = t^i; t_v = t^i$;
6  $\quad h_u = \text{Aggregate}(\{z_w^{(i-1)} \mid (w, u) \in E_e\}, W_a)$;
7  $\quad h_v = \text{Aggregate}(\{z_w^{(i-1)} \mid (w, v) \in E_e\}, W_a)$;
8  $\quad z_u^{(i)} = \text{UpdateEmb}(h_u, z_u^{(i-1)}, h_v, \Delta t_u, T, W_u)$;
9  $\quad z_v^{(i)} = \text{UpdateEmb}(h_v, z_v^{(i-1)}, h_u, \Delta t_v, T, W_u)$;
10 $\quad$ **for** each $w \in V_e \setminus \{u, v\}$ **do**
11 $\quad\quad \lfloor\ z_w^{(i)} = \text{PropUpdate}(z_w^{(i-1)}, z_u^{(i)}, z_v^{(i)}, W_p)$
12 $\quad$ **for** each $w \in V^{(i)} \setminus V_e$ **do**
13 $\quad\quad \lfloor\ z_w^{(i)} = z_w^{(i-1)}$
14 $\quad \hat{y}^{(i)} = \text{Predict}(z_u^{(i)}, z_v^{(i)})$;
15 $\mathcal{L} = loss\_func(\{\hat{y}^{(1)}, \hat{y}^{(2)}, \ldots, \hat{y}^{(n)}\}, Y)$;
  **Backward:**
16 $\{\nabla W_a, \nabla W_u, \nabla W_p\} \leftarrow \mathcal{L}.backward()$;
17 update $\{W_a', W_u', W_p'\}$ based on $\{\nabla W_a, \nabla W_u, \nabla W_p\}$;

---

[20], TGN [45]. Next, the algorithm computes the time intervals of the event nodes $u, v$ and updates their time points, respectively (Line 5). $\Delta t_u$ and $\Delta t_v$ represent the time elapsed since $u$'s previous interaction and $v$'s previous interaction, respectively, and are used as input to indicate the frequency of their interaction. Then, the algorithm computes the aggregated neighborhood embedding for each event node (Line 6-7). The aggregated neighborhood embeddings capture rich structural information, which is key to any representation learning task over graphs. The algorithm then computes the new embeddings $\{z_u^{(i)}, z_v^{(i)}\}$ for the event nodes based on the aggregated neighbor representation $\{h_u, h_v\}$, old embeddings $\{z_u^{(i-1)}, z_v^{(i-1)}\}$ and the time interval $\{\Delta t_u, \Delta t_v\}$ (Line 8-9).

The updated embeddings are then propagated to their neighbors (Line 10-11). This step is specific to dynamic GNN diffusion models like DGNN [28]. The updated event node embeddings are used as the input for downstream tasks such as link prediction and node classification (Line 14). The set of predicted labels $\hat{y}^{(i)}$ and the set of ground-truth labels $Y$ are used to calculate the loss value after executing $n$ events (Line 15). The link prediction task refers to predicting new edges between nodes based on historical events, typically without ground-truth labels. A common approach is to use negative sampling to construct negative samples and take the edge between event nodes as the positive sample. After generating positive and negative samples, the training of link prediction can be transformed into a binary classification problem. The parameters of the model are updated based on the loss (Line 16-17). Compared with traditional GNNs, learning dynamic node embeddings is more challenging due to complex time-varying graph structures.

**Algorithm 2:** Incremental Training with Sliding Window

**Input** : $W^{(0)}$, $G^{(0)}$, $Z^{(0)}$, $\{e[1], e[2], \ldots, e[m]\}$, $Y$, fixed window size $s$, and stride $d$

**Output** : $W^{(m)}$, $G^{(m)}$, and $Z^{(m)}$

1   $i \leftarrow 0$;   //window start position

2   **while** $i \leq m$ **do**

3     $j = (i + s < m)?(i + s) : m$;   //window end position

4     $\{W^{(j)}, G^{(j)}, Z^{(j)}\} = \text{DyGNN}(W^{(i)}, G^{(i)}, Z^{(i)}, \{e[i], e[i+1],$
     $\ldots, e[j]\}, Y)$;   //Algorithm 1

5     $i \leftarrow i + d$;   //window start position in next round

## 2.4 Limitations of Existing Frameworks

In many applications, a graph stream is a sequence of events that grows over time. To keep the model fresh, continuous training is crucial. However, existing dynamic GNN models are mainly implemented on general DNN or GNN frameworks [5, 41, 55]. They only support offline training based on events within a past time window because they lack an effective data structure to support the evolving graph topology. TGL [75] is a training framework designed for temporal GNNs, but it is also an offline training framework that requires storing the entire dynamic graph statically before training. Therefore, TGL needs to obtain all event information in advance to build the graph structure, which loses the flexibility to support dynamic graph updates. Additionally, TGL adopts an intra-batch parallel method, which breaks temporal dependencies between events and reduces model accuracy. To address these limitations, we propose NeutronStream, the first dynamic GNN framework supporting graph streaming training. We also propose a parallel execution scheme that guarantees temporal dependencies between events to accelerate model training.

## 3 DYNAMIC GNN TRAINING WITH SLIDING WINDOW

In this section, we propose an event-based training mode with a sliding window, which helps capture the spatial and temporal dependencies between events in a timely manner. Furthermore, we improve the basic sliding window method by adaptively adjusting the window size, aiming to include all relevant events in a window.

### 3.1 Incremental Mode with Sliding Window

In a streaming scenario, we assume update events are coming continuously. The key design to support training in this scenario is that we design a sliding window to select a set of training events chronologically from the input event stream. This training method is a continuous process until all events are trained or can be designed as an online process continuously accepting new update events. Algorithm 2 depicts the training process on a finite event stream $\{e[1], e[2], \ldots, e[m]\}$ given the initial model parameters $W^{(0)}$, the initial graph structure $G^{(0)}$, and the initial node embeddings $Z^{(0)}$.

There are two key parameters to control the sliding window, including the window size $s$ and the stride $d$. For example, in Figure 2 (a), the size $s = 4$ and the stride $d = 2$. The stride $d$ determines how far the window slides each time. The setting of $d$ needs to balance training efficiency and retaining information. Specifically, when $d$ is larger, the model trains faster but loses more information. The window size $s$ determines how much data can be fed into the
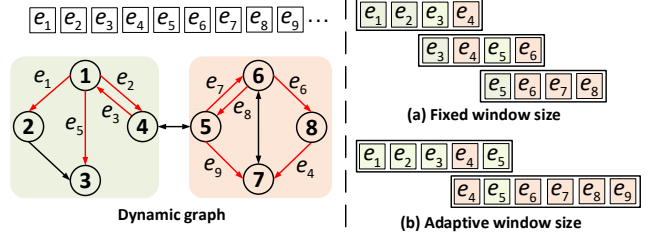


**Figure 2: Fixed window vs. adaptive window.**

model. The naive idea is to set the window size to a fixed value, and we have a fix-sized event set from the event stream each time.

The algorithm first determines the start and end positions ($i$ and $j$) of a window. Then the sequence of events in this window as input, *i.e.*, $\{e[i], \ldots, e[j]\}$, are fed into the dynamic GNN model DyGNN as described in Algorithm 1. The model parameters $W^{(i)}$, the graph topology $G^{(i)}$, and the node embeddings $Z^{(i)}$ at time $t_i$ are also the input of DyGNN, which will output the updated model parameters $W^{(j)}$, the updated graph topology $G^{(j)}$, and the updated node embeddings $Z^{(j)}$. Then the window slides with a stride $d$ to obtain the start position and end position of the next window. We repeat the above process until meeting the stream end position $m$. For an infinite stream, it will keep receiving new events and launch the DyGNN for a new window of events continuously.

### 3.2 Adaptive Adjustment of Window Size

The setting of the window size $s$ is a key issue. If $s$ is too small, the window contains little information. If $s$ is too large, the window may also introduce noise information, disturbing inference performance [44]. We notice that there is spatial locality among continuous events. A continuous segment of events is usually concentrated in a local area in the graph. For example, in a user-item network, a person may focus on browsing and purchasing certain items in a certain period of time, which will generate continuous interaction events. The model needs to encode the information of these highly related events into the user's embedding in time, so this sequence of correlated events should all be contained in a window, which cannot be satisfied by a fixed window size. As shown in Figure 2, there are 9 events where event set $\{e_1, e_2, e_3, e_5\}$ occurs in the green area and event set $\{e_4, e_6, e_7, e_8, e_9\}$ occurs in the orange area. However, if the window size is a fixed value of 4, the model cannot learn the spatial locality relations among these events. Figure 2 (a) shows that none of the three windows can contain one of the event sets. Therefore, the window size should be dynamically adjustable to capture the spatial locality, as shown in Figure 2 (b). Therefore, we propose an algorithm to adaptively adjust the window size for capturing the spatial locality as shown in Algorithm 3.

The idea of Algorithm 3 is to iteratively judge whether there is a correlation between the following event outside the window and the events already contained in the window. The inputs include an event stream $\{e[i], e[i+1], \ldots, e[m]\}$, minimum window size $L$, and maximum window size $H$. $L$ and $H$ are two hyperparameters that limit the number of events the window can contain. We adopt these two hyperparameters to control the amount of events input into the model training each time. If there is no association between a sequence of events, we use the minimum size $L$ to ensure the least amount of training data. If there are many associations among sequential events, this results in more and more events being added

**Algorithm 3:** Determination of a Window of Events

**Input** : Event stream $\{e[i], e[i+1], \ldots, e[m]\}$, minimum window size $L$, and maximum window size $H$

**Output** : A window of events $\{e[i], e[i+1], \ldots, e[i+s]\}$

1  $s \leftarrow 0$;  //window size
2  $nodes \leftarrow \emptyset$;  //a node set for capturing event dependencies
3  **while** $s \leq H$ **do**
4  $\quad j \leftarrow i + s$;  //window end position
5  $\quad \{u, v, t^j, T\} = parse(e[j])$;
6  $\quad$ **if** $s \leq L$ **then**
7  $\quad\quad\mid\;\; nodes \leftarrow nodes \cup \{u, v\}$;  //not enough events
8  $\quad$ **else if** $\{u, v\} \cap nodes \neq \emptyset$ **then**
9  $\quad\quad\mid\;\; nodes \leftarrow nodes \cup \{u, v\}$;  //spatial dependent event
10 $\quad$ **else**
11 $\quad\quad\mid\;\;$ **break**;
12 $\quad s \leftarrow s + 1$;  //updated window size
13 **return** $\{e[i], e[i+1], \ldots, e[i+s]\}$

to the window, and the window can be infinitely long. Therefore, we use the maximum size $H$ to limit the maximum length of the window. The output is a window of events $\{e[i], e[i+1], \ldots, e[i+s]\}$.

The algorithm first initializes the set $nodes$ and $s$, where $nodes$ represents a set of event nodes already in the window and $s$ represents the number of events already added (Line 1-2). Then the algorithm sequentially detects whether each event has spatial locality to events already in the window (Line 3-12). We get an event $e[j]$ from the event stream based on the index $j$ each time (Line 4-5). Next, we perform a judgment for the event $e[j]$. If the number of added events in the window is less than the minimum size $L$, this event is directly added to the window, and the event nodes $\{u, v\}$ are added to the set $nodes$ (Line 6-7). Otherwise, the algorithm judges whether there is an intersection between event nodes $\{u, v\}$ of event $e[j]$ and the set $nodes$. If there is an intersection, it means a locality exists between the $e[j]$ and the events already in the window. Therefore, this event should be added to the window (Line 8-9). If neither condition is met, break out of the loop (Line 10-11). If the number of events in the window reaches the maximum size $H$, break out of the loop (Line 3). Finally, we can get the range $[i, i+s]$ of events in this window, and then we return the window (Line 13).

Based on Algorithm 3, we can put a segment of events with spatial locality into a window so that the model can learn the locality between events. This algorithm can replace the fixed window size in Algorithm 2. Another method is to check whether there is an intersection between consecutive events based on the event subgraph. However, this method makes dependency analysis more time-consuming. Moreover, most dynamic GNNs only update event nodes and do not update their neighbors. Therefore, it is reasonable to use Algorithm 3 to determine the window.

## 4 DEPENDENCY-GRAPH-DRIVEN EVENT PARALLELIZING

When training the dynamic GNN model with a window, the events in the window are executed chronologically, as shown in Algorithm 1. For example, in Figure 3 (a), a window contains five events. Figure 3 (b) illustrates that the model executes these events sequentially, leading to a long training time. However, we notice that the computation of an event only involves a subgraph, typically consisting

of the event nodes and their neighbors. If there is no intersection of subgraphs between two events, they can be executed in parallel. For example, in Figure 3 (a), the purple area is the subgraph of events $e_1$ and $e_3$, and the green area is the subgraph of events $e_2$ and $e_4$. The two areas are disjoint, so events $e_1$ and $e_3$ can be executed in parallel with events $e_2$ and $e_4$. This fact brings the opportunity of parallelism in event processing. In this section, we first analyze the conditions of event parallelism and give related definitions. We then propose an algorithm to automatically analyze the dependencies of events and identify the events that can be executed in parallel.

### 4.1 Dependency Analysis of Event Processing

The parallel processing of events can be explained by answering two questions.

**Question 1: Why are there dependencies among events?**

If there are no read-write conflicts between two operations, the two operations can be processed in parallel. Therefore, we first define an event's read set and write set. As shown in Algorithm 1, An event $e = (u, v, t, T)$ involves event nodes $\{u, v\}$ and their $k$-hop neighbors where $k$ is determined by the model. The computation of an event requires reading and writing information about these nodes from the graph topology and node embeddings. For example, Aggregate requires reading the node embeddings of neighbors, while UpdateEmb requires updating the embeddings of event nodes. As shown in Figure 3 (a), the event $e_1$ involves event nodes 1, 2 and their one-hop neighbors 3, 4. Therefore, Aggregate requires reading the embeddings of neighbors 3, 4, while UpdateEmb requires writing the embeddings of event nodes 1, 2.

Based on the above analysis, we define **Event-Affected Subgraph** and **Event-Triggered Update Set** to denote the read range and the write range of an event, respectively.

DEFINITION 1. **Event-Affected Subgraph.** *For an event* $e = (u, v, t, T)$, *it generally needs to read the information of the subgraph composed of the $k$-hop neighbors of the event nodes $\{u, v\}$. We refer to this subgraph as the $k$-hop Event-Affected Subgraph $\mathcal{E}_e^k(V_e^k, E_e^k)$, where $V_e^k = \{\mathcal{N}_u^k \cup \mathcal{N}_v^k\}$ and $E_e^k$ is the set of edges between them.*

DEFINITION 2. **Event-Triggered Update Set.** *After processing an event $e = (u, v, t, T)$, the embeddings of the nodes around $u$ and $v$ need to be updated. This set of nodes is referred to as the event-triggered update set and is denoted by $\mathcal{U}_e$.*

For example in Figure 3 (a), the 1-hop event-affected subgraph of event $e_1$ is $\mathcal{E}_{e_1}^1$ and $V_{e_1}^1 = \{1, 2, 3, 4\}$. Its event-triggered update set is $\mathcal{U}_{e_1} = \{1, 2\}$. The 1-hop event-affected subgraph of event $e_2$ is $\mathcal{E}_{e_2}^1$ and $V_{e_2}^1 = \{5, 6, 7, 8\}$. Its event-triggered update set is $\mathcal{U}_{e_2} = \{5, 8\}$. Based on Definition 1 and Definition 2, we define a dependency between two events as having a read-write conflict as follows.

DEFINITION 3. **Dependencies between events.** *For any two events $e_i = (u_i, v_i, t_i, T_i)$ and $e_j = (u_j, v_j, t_j, T_j)$ where $t_i < t_j$, we have their corresponding event-affected subgraphs, i.e., $\mathcal{E}_{e_i}^k(V_{e_i}^k, E_{e_i}^k)$ and $\mathcal{E}_{e_j}^k(V_{e_j}^k, E_{e_j}^k)$, and event-triggered update sets, i.e., $\mathcal{U}_{e_i}$ and $\mathcal{U}_{e_j}$. If $V_{e_j}^k \cap \mathcal{U}_{e_i} \neq \emptyset$, we say that event $e_j$ depends on event $e_i$, i.e., $e_j \Rightarrow e_i$.*

For example, in Figure 3 (a), the 1-hop event-affected node set of event $e_3$ is $V_{e_3}^1 = \{1, 2, 3, 4\}$. The event-triggered update set of $e_1$
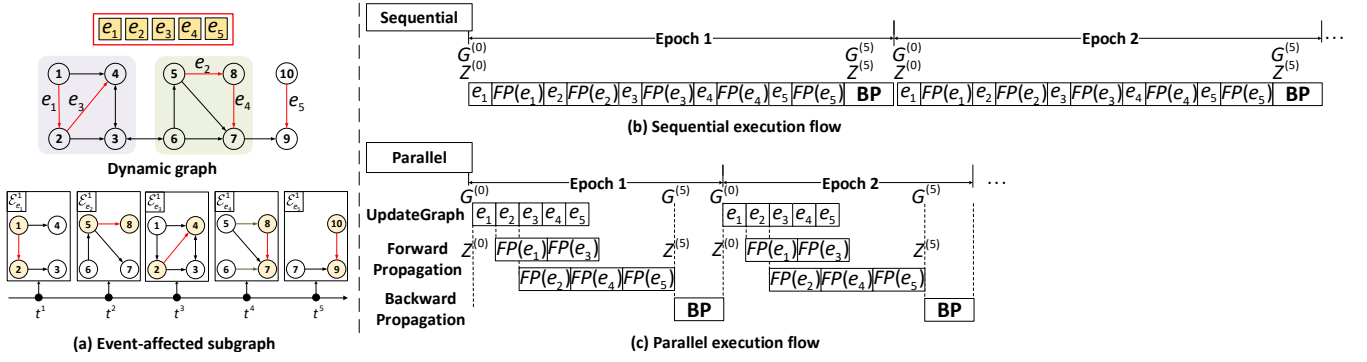
**Figure 3: Execution flow comparison of sequential execution events and parallel execution events. The red arrows represent the newly added edge. FP represents forward propagation, while BP represents backward propagation. Figure (a) is the 1-hop event-affected subgraph $\mathcal{E}_e^1$ of event $e_1, e_2, e_3, e_4, e_5$. The purple area is the affected subgraph of events $e_1$ and $e_3$, and the green area is the affected subgraph of events $e_2$ and $e_4$.**

is $\mathcal{U}_{e_1} = \{1, 2\}$. $V_{e_3}^1 \cap \mathcal{U}_{e_1} = \{1, 2\}$, which is not an empty set, so event $e_3$ depends on event $e_1$, marked as $e_3 \Rightarrow e_1$.

The dependency relationship $e_j \Rightarrow e_i$ indicates that event $e_j$ must wait for event $e_i$ to complete before executing. Similarly, if another event $e_k$ depends on event $e_j$, event $e_k$ must also wait for event $e_j$ to complete. Therefore, Event $e_k$ must also wait for event $e_i$. Event $e_k$ further depends on event $e_i$, resulting in a dependency chain, ie. $e_k \Rightarrow e_j \Rightarrow e_i$. The dependencies between events satisfy the transitivity theorem. The theorem is as follows.

THEOREM 1. **Dependency transitivity.** *For any three events $e_i$, $e_j$ and $e_k$, if $e_j \Rightarrow e_i$, $e_k \Rightarrow e_j$, then $e_k \Rightarrow e_i$.*

For a dependency chain $e_k \Rightarrow e_j \Rightarrow e_i$, non-adjacent events do not necessarily satisfy Definition 3. For example, in Figure 3 (a), the event-triggered update set of $e_2$ is $\mathcal{U}_{e_2} = \{5, 8\}$. The event-affected node set of event $e_4$ is $V_{e_4}^1 = \{5, 6, 7, 8\}$. $\mathcal{U}_{e_2} \cap V_{e_4}^1 = \{5, 8\}$, which is not an empty set, so event $e_4$ depends on event $e_2$. Similarly, the event-triggered update set of $e_4$ is $\mathcal{U}_{e_4} = \{7, 8\}$. The event-affected node set of event $e_5$ is $V_{e_5}^1 = \{7, 9, 10\}$. $\mathcal{U}_{e_4} \cap V_{e_5}^1 = \{7\}$, which is not an empty set, so event $e_5$ depends on event $e_4$. Therefore, the dependencies of these three events form a dependency chain, ie. $e_5 \Rightarrow e_4 \Rightarrow e_2$. However, event $e_2$ has no read-write conflict with event $e_5$, ie. $\mathcal{U}_{e_2} \cap V_{e_5}^1 = \varnothing$. Therefore, Definition 3 is a sufficient condition for the dependency. Two adjacent events in a dependency chain always satisfy Definition 3, so we refer to this dependency as a direct dependency.

**Question 2: Under what conditions can events be parallelized?**

Definition 3 and Theorem 1 indicate that if two events are not in the same dependency chain, they can be executed in parallel. As shown in Figure 3, the 1-hop event-triggered update set of event $e_1$ is $\mathcal{U}_{e_1} = \{1, 2\}$. The event-affected node set of event $e_3$ is $V_{e_3}^1 = \{1, 2, 3, 4\}$. Since $\mathcal{U}_{e_1} \cap V_{e_3}^1 = \{1, 2\}$, which is not an empty set, so event $e_3$ depends on event $e_1$. Therefore, these two events form a dependency chain, ie. $e_3 \Rightarrow e_1$. Similarly, event $e_2$, event $e_4$ and event $e_5$ form another dependency chain, ie. $e_5 \Rightarrow e_4 \Rightarrow e_2$. Therefore, the events of these two chains can be executed in parallel, as shown in Figure 3 (c). Parallel training can obviously speed up model training. However, existing frameworks [5, 41, 55] cannot analyze the dependencies between events and can only process these events sequentially, causing degraded performance. Therefore, we propose an algorithm to automatically analyze event dependencies, achieving parallel execution between events while strictly guaranteeing temporal dependency.
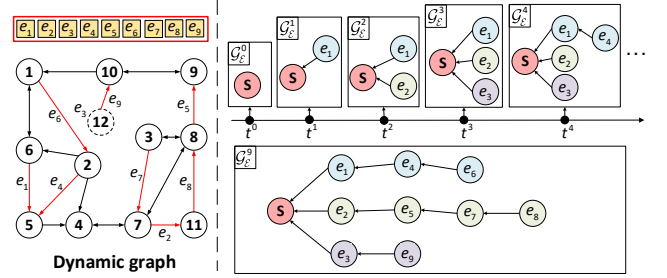


**Figure 4: Event dependency graph. The red arrows represent the newly added edge, and the dotted line represents the newly added node. The upper right corner is the evolved Event Dependency Graph. The lower right corner is the complete event dependency graph, where nodes of the same color represent a dependency chain.**

## 4.2 Parallelization based on Event Dependency Graph

We define the **Event Dependency Graph** to represent dependencies between events as follows.

DEFINITION 4. **Event Dependency Graph.** *An event dependency graph is denoted as $\mathcal{G}_\mathcal{E} = (\mathcal{V}, \mathcal{E})$. Here $\mathcal{V}$ is a set of nodes where each node represents an event. $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges where each edge represents a dependency. For example, a directed edge $(e_j, e_i)$ indicates that event $e_j$ directly depends on event $e_i$, i.e., $e_j \Rightarrow e_i$.*

We introduce a super-node marked $S$ as the root node in $\mathcal{G}_\mathcal{E}$, and all events depend on it. For example, in Figure 4, a window contains nine events. Figure 4 depicts the event dependency graph $\mathcal{G}_\mathcal{E}^9$ of these events. For example, edge $(1, S)$ indicates that $e_1$ depends on $S$. Similarly, edge $(4, 1)$ indicates that $e_4$ depends on $e_1$, and edge $(6, 4)$ indicates that $e_6$ depends on $e_4$. Since event dependencies are transitive, $e_6$ also depends on $e_1$ and $S$. Therefore, these three events form a dependency chain, i.e., $e_6 \Rightarrow e_4 \Rightarrow e_1 \Rightarrow S$. Similarly, event set $\{e_2, e_5, e_7, e_8\}$ forms a dependency chain, and event set $\{e_3, e_9\}$ forms another chain. Therefore, these nine events form three dependency chains. We can execute events in parallel based on this dependency graph. Events can be executed in parallel if they do not belong to the same dependency chain. Therefore, the events in these three chains can be executed in parallel. To infer the event dependency graph, we propose an event dependency graph generation algorithm, as shown in Algorithm 4.

In a dependency chain, the direct dependencies between two adjacent events always satisfy Definition 3. Therefore, the idea of
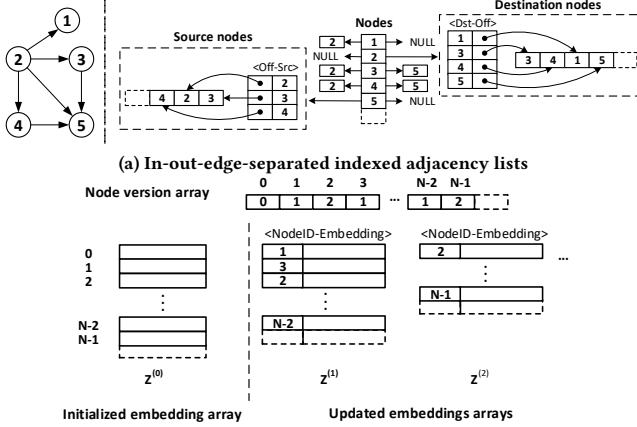
**Algorithm 4:** Generate Event Dependency Graph

---

**Input** : A window of events $\{e[1], e[2], \ldots, e[s]\}$
**Output**: Event dependency graph $\mathcal{G}$

1   initialize $\mathcal{G}$ with a root $e[0]$;   //for any $e[i], e[i] \Rightarrow e[0]$
2   **for** $e[i] \leftarrow e[1]$ **to** $e[s]$ **do**
3     **for** $e[j] \leftarrow e[i-1]$ **to** $e[0]$ **do**
4       **if** $e[i] \Rightarrow e[j]$ **then**
5         add event dependency $(e[i] \Rightarrow e[j])$ to $\mathcal{G}$;
6         add $e[j]$ to $e[i].deps[]$;

7   **return** $\mathcal{G}$;

---



(a) In-out-edge-separated indexed adjacency lists

(b) Multi-version node embedding storage

**Figure 5: Dynamic graph storage design.**

Algorithm 4 is to construct a dependency graph by identifying all direct dependencies. The input is a window of events, and the output is the event dependency graph of this window. The algorithm first initializes the dependency graph with a root event $e[0]$, where all events depend on $e[0]$ (Line 1). Next, the algorithm sequentially detects dependencies between each event $e[i]$ and the preceding events $e[i-1]$ to $e[0]$ (Line 2-6). The algorithm iteratively judge the dependencies between $e[i]$ and the events $e[i-1]$ to $e[0]$ (Line 3-6). If $e[i]$ depends on $e[j]$, we add an edge from $e[i]$ to $e[j]$ in the graph and add $e[j]$ to $e[i]$'s the dependency set $e[i].deps[]$. Finally, we can obtain the dependency graph of this window (Line 7).

Next, we can execute in parallel based on this generated dependency graph. We maintain a flag for each event indicating whether it has been completed. We use two queues, UnExec and Exec, to manage the execution state of events. UnExec stores events that have not been executed, while Exec stores events that can be executed in parallel. First, we traverse the generated graph and put all events into UnExec. Then, the scheduling thread continuously detects whether the dependency set $deps[]$ of each event in UnExec is empty. If the dependency set of an event is empty, it indicates that the event can be executed immediately. The scheduling thread moves the event to Exec. Multiple execution threads take events from Exec and execute them in parallel.

## 5 NEUTRONSTREAM

### 5.1 Dynamic Graph Storage

As shown in Algorithm 1, the complex computations of dynamic GNNs require storage structures that can provide efficient read and write operations. In the graph streaming training process, the storage structure also needs to support dynamic updates, such as node additions and edge additions. However, the mature DNN/GNN frameworks [5, 41, 55] lack support for dynamic graphs, requiring users to design and manage dynamic graphs, increasing the burden. Manually implemented dynamic graphs are typically inefficient and inflexible. Although there are some dynamic graph storages [10, 22, 77], they focus on graph analysis workloads and do not natively support storing node vectors. Dynamic GNNs require storage structures supporting node embedding storage and multi-version embedding storage for backpropagation. Therefore, we design a dynamic graph data structure *in-out-edge-separated indexed adjacency lists* and a node embedding data structure *multi-version node embedding* to address these requirements, as shown in Figure 5.

**Dynamic Graph Storage.** We propose a dynamic graph storage as shown in Figure 5 (a). It efficiently and flexibly stores incoming and outgoing edges separately for each node because some models need to access both incoming and outgoing edges like DGNN [28].

Each node has two dynamic arrays to store the incoming and outgoing edges, including source/destination node IDs, edge weights, and timestamps. Two dynamic arrays ensure that a node's incoming and outgoing edges can be visited continuously. However, arrays suffer from edge lookups by scanning for fine-grained access. For example, when an event needs to update the timestamp of a specified edge, we need to scan this array sequentially, and the time complexity is $O(n)$. To accelerate lookups, we maintain Key-Value pairs of $\langle NeighborID \rightarrow Offset \rangle$ for edges, indicating edge locations in arrays. We use Hash Table as the default index because our data structure with Hash Table provides an average $O(1)$ time complexity for each fine-grained access. Furthermore, the index does not hurt the analysis performance of traversing a node's neighbors because we can directly access the dynamic array to get all neighbors without involving the index. The structure indexes can directly locate edges without scanning, which is friendly for insertions and deletions of an edge. Since we use dynamic arrays to store nodes and edges, dynamic graph updates are naturally supported.

**Multi-Version Node Embedding Storage.** To support batch backpropagation, storing multiple versions of node embeddings for nodes that are updated multiple times within a batch is necessary. Figure 5 (b) illustrates our proposed multi-version embedding storage structure, which optimizes storage usage and enables efficient read and write operations. The embedding storage contains three components: initialized embedding array, updated embedding arrays, and node version array.

Before training starts, the initial node embeddings are stored in a dynamic array $Z^{(0)}$, marked as version 0. The version number is incremented when the node embedding is updated. We store updated embeddings with the same version together. Each embedding is stored as a key-value pair $\langle NodeID \rightarrow Embedding \rangle$. We use Hash Table as the default index to speed up access. The node version array records the latest node version, indicating in which dynamic array the latest embedding of each node is stored. For example, the latest version of node 0 is 0, meaning that we need to find the latest embedding in $Z^{(0)}$. Similarly, the latest version of node 2 is 2, meaning that we need to find the latest embedding in $Z^{(2)}$.

Based on this structure, we can quickly access and store the specific node embedding through two indexes with a time complexity

of $O(1)$. For example, if we want to query the embedding of node 2 with version number 1. We first need to get the latest version number of node 2 according to the node ID in the node version array. We can get that the version is 2, which indicates that there is an embedding with version number 1. We can then locate the embedding in $Z^{(1)}$ according to the node ID. The multi-version node embedding storage can support efficient access and modification, avoid redundant storage and provide users with more flexibility.

## 5.2 Programming Model and APIs

Implementing dynamic GNN models based on existing NN/GNN frameworks [5, 41, 55] requires users to implement storage structures and various complex graph operators manually, which requires high human efforts and is also tedious and error-prone. To facilitate the management of dynamic graphs, we provide a set of APIs for accessing the underlying graph storage and node embedding storage. To enable users to focus on implementing the algorithm logic, we abstract the execution process of an event into four programming interfaces according to Algorithm 1: **UpdateGraph**, **Aggregate**, **UpdateEmb**, and **PropUpdate**. Users only need to customize these four functions to implement the execution logic of an event. The details of these APIs are listed in the following.

- **DynGraph** is the underlying graph storage. It provides various functions, such as adding an edge (add_edge(uid, vid)), deleting an edge (del_edge(uid, vid)), adding a node (add_node(nid)), getting an event subgraph (get_subgraph(evt)), and querying the latest interaction time of a node (query_time (nid)).
- **DynEmb** is the underlying node embedding storage. It provides various functions, such as querying a node embedding (index (nid)) and updating a node embedding (update(nid, upd_emd)).
- **UpdateGraph** is a user-defined function that updates the graph structure based on event types, such as adding/removing edges. The parameters include the dynamic graph dynGraph and an event evt.
- **Aggregate** is a user-defined function that computes aggregated neighbor information for an event node. The parameters include event subgraph dynSubgraph, the node embedding dynEmb, and an event node nid.
- **UpdateEmb** is a user-defined function that computes updated embedding for an event node. The parameters include event subgraph dynSubgraph, the node embedding dynEmb, an event node nid, aggregated neighbor information h_agg, and event timestamp time.
- **PropUpdate** is a user-defined function that propagates an event node's embedding to neighbors. This function appears in diffusion models, such as DGNN [28]. The parameters include dynSubgraph, dynEmb, and an event node ID nid.

We can easily represent existing dynamic GNN models with the above interfaces. For each event node, DyRep [51] needs to update its embedding based on three parts: Localized Embedding Propagation, Self-Propagation, and Exogenous Drive. We use UpdateGraph to update the graph structure based on the update event. Localized Embedding Propagation needs to aggregate neighbor information, which can be represented by Aggregate. Self-Propagation and Exogenous Drive refer to computations based on previous embedding information and time interval of event nodes, respectively. We

```
Class DyRep(nn.module):
    def __init__(self, h_dim):
        # initial parameters include W_h, W_struct, W_rec, W_t
and attention martix S

# update graph structure based an event
    def UpdateGraph(evt, dynGraph):
        # compute intensity
        # update attention matrix self.S
        if evt.type == 'Association'
            dynGraph.add_edge(evt.u,evt.v, undir=True)
        return dynGraph.get_subgraph(evt)
# compute aggregated neighbor information of an event node
    def Aggregate(dynSubgraph, dynEmb, nid):
        nbr = dynSubgraph.get_nbr(nid, dir='in')
        nbr_emb = self.W_h(dynEmb.index(nbr))
        q_ni = torch.exp(self.S[nid, nbr])
        q_ni = q_ni/(torch.sum(q_ni) + 1e-7)
        agg = torch.max(nn.sigmoid(q_ni.view(1,1)*nbr_emb),dim=0)[0]
        return self.W_struct(agg)
# compute updated embedding of an event node
    def UpdateEmb(dynSubgraph, dynEmb, nid, h_agg, time):
        delta_time = time - dynSubgraph.query_time(nid)
        h_exogenous = self.W_t(delta_time)
        h_self = self.W_rec(dynEmb.index(nid))
        upd_emb = nn.sigmoid(h_agg + h_self + h_exogenous)
        dynEmb.update(nid, upd_emb)
# forward function
    def forward(evt_list, dynGraph, dynEmb):
        for evt in evt_list:
            dynSubgraph = UpdateGraph(evt, dynGraph)
            h_u_agg = Aggregate(dynSubgraph, dynEmb, evt.u)
            h_v_agg = Aggregate(dynSubgraph, dynEmb, evt.v)
            UpdateEmb(dynGraph, dynEmb, evt.u, h_v_agg, evt.time)
            UpdateEmb(dynGraph, dynEmb, evt.v, h_u_agg, evt.time)
```

**Figure 6: DyRep implementation with NeutronStream APIs.**

can use UpdateEmb to compute Self-Propagation and Exogenous Drive and combine the three parts to obtain the updated event node embeddings. Figure 6 shows the logic of computing event node embeddings in the Dyrep model on NeutronStream. Users only need to define **UpdateGraph**, **Aggregate**, and **UpdateEmb**.

## 5.3 Event Processing Engine and Pipelining

The Event Processing Engine is responsible for window determination and parallel training. The framework consists of four modules: Adaptive sliding window module, Event parallel analysis module, Event parallel scheduling module, and Event execution module. Each module performs different training tasks. First, the adaptive sliding window module determines a window of events from the event stream according to Algorithm 3. Then, the event parallel analysis module generates the event dependency graph of the window according to Algorithm 4. Based on the generated dependency graph, the event parallel scheduling module generates a parallel execution plan and dispatches executable events to the event execution module. Next, the execution module utilizes the thread pool to execute events in parallel. The sequential nature of time-dependent events requires that events be processed sequentially, making it difficult to divide training tasks and implement pipeline optimization. Based on our event parallel approach, we can divide the training process into different tasks, making pipeline optimization possible.

We propose a pipelining optimization to fully use each module, as shown in Figure 7. The pipeline enables concurrent execution of different windows. Once the first window is obtained, the adaptive sliding window module can continue to determine the second window. After the event analysis module has generated the event dependency graph of the first window, it can continue to analyze
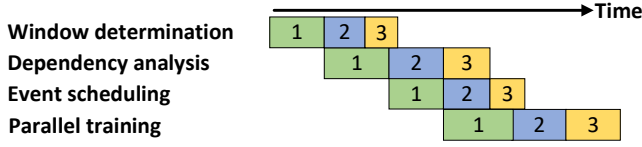
Figure 7: Pipeline optimization. The colors green, blue, and yellow indicate different windows of training events.

the second window. Similarly, the event parallel scheduling module can generate a scheduling plan for the second window without waiting for the first window to complete. The design can reduce the waiting time of each module and speed up training. Furthermore, dependency is important. Ignoring dependencies and executing events completely in parallel could cause the model to fail to capture dynamic information correctly and reduce accuracy.

**Evaluation of the Impact on Temporal Dependency.** We evaluate the impact of temporal dependency on DyRep [51] and DGNN [28] with the Social dataset, as shown in Table 1 and Figure 8. With Temporal Dependency indicates that the model executes events according timestamps. We use our parallel approach to run this pattern because our approach can strictly guarantee dependency among events (Algorithm 4). Without Temporal Dependency indicates ignoring the temporal dependency among events and executing them completely in parallel. We perform training on the first 80% of events and testing on the final 20%. As shown in Table 1, Without Temporal Dependency achieves a higher speedup. However, the speedup is limited due to conflicts in reading and writing node embeddings. As shown in Figure 8, Without Temporal Dependency completely loses time information and therefore has lower accuracy. With Temporal Dependency retains accurate dependency information, enabling the model to capture dynamic information correctly and achieve higher accuracy. Therefore, it is necessary to maintain the dependencies among events. We also record the time of performing dependency analysis according to Algorithm 4, as shown in Figure 9. The execution time of Algorithm 4 only takes up a small ratio of the total time, but the training can be parallelized. (Note that this time is recorded when the pipeline optimization is turned off. With pipeline optimization, this dependency analysis process will be overlapped.) Therefore, the dependency analysis will not become a bottleneck.

Table 1: Performance comparison with dependency and without dependency

| Dataset | Runtime of 80 epochs (s) | | Speedup |
| --- | --- | --- | --- |
| | With Temporal Dependency | Without Temporal Dependency | |
| DyRep | 8592.48 | 4544.05 | 1.89X |
| DGNN | 25521.12 | 16714.96 | 1.53X |


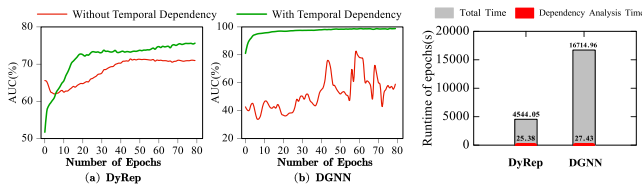
Figure 8: AUC Comparison with dependency and without dependency.
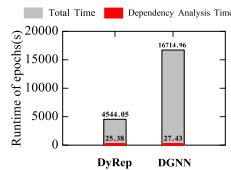


Figure 9: Ratio of dependency analysis time.

Table 2: Dataset statistics

| Dataset | $|V|$ | $|E|$.init | $|E|$.final | $evt.num$ |
| --- | --- | --- | --- | --- |
| Social Evolution [29] | 84 | 575 | 794 | 54,369 |
| Github [4] | 284 | 298 | 4,131 | 20,726 |
| DNC [21] | 2,029 | 0 | 5,598 | 39,264 |
| UCI [21] | 1,899 | 0 | 20,296 | 59,835 |
| Reality [7] | 6,809 | 0 | 9,484 | 52,052 |
| Slashdot [12] | 51,083 | 0 | 131,175 | 140,778 |

## 6 EXPERIMENTAL EVALUATION

### 6.1 Experimental Setup

**Environments.** Our experiments are conducted on an Aliyun ECS server (ecs.g5.6xlarge instance) equipped with 24 vCPU cores, 96GB DRAM. The server runs Ubuntu 18.04 LTS OS with GCC-7.5 as the host compiler. Libraries OpenMP-4.0.3, PyTorch v1.12.1 backend [41] are used in the server.

**Models.** We select three Dynamic GNN models: DyRep [51] , LDG [19] and DGNN [28]. Most existing stream-based dynamic GNNs models can be divided into two categories: *temporal point-based model* and *RNN-based model*. DyRep and LDG are two representative temporal point-based models, while DGNN is a representative RNN-based model. In our experiments, we focused on the one-hop neighbors of event nodes because DyRep [51] and LDG [19] aggregate the one-hop neighbor information through the attention mechanism, while DGNN [28] diffuses the updated event node embedding to one-hop neighbors. We set the dynamic node embedding size to be 64.

**Datasets.** Table 2 shows the six datasets used in our experiments. Social Evolution [29] and Github [4] are two social network datasets. DNC [21], UCI [21], Reality [7] and Slashdot [46] are communication network datasets. In Social Evolution and Github, nodes represent users, and edges represent various type of interaction events. In DNC, UCI, Reality, and Slashdot, nodes represent users, and edges represent communication events. For Social Evolution graph, we followed DyRep [51] and LDG [19], configuring *Close-Friend* events as association events and other types of events as communication events. For Github, We use *Follow* events as association events and other types of events as communication events.

**Baselines.** We implement three training methods on Neutron-Stream: batch, sliding window, and adaptive sliding window. Batch training refers to accumulating a batch of new events from the event stream as the training set input to the model, denoted as NS-Batch. We set the batch size to 200. Sliding window refers to sliding a certain stride each time and selecting a fixed window event as a training set, denoted as NS-Slide (Algorithm 2). We set the stride to 40 and the window size to 200. The adaptive sliding window refers to adaptively adjusting the window size based on the sliding window method to capture continuous events with locality, denoted as NS-AdaSlide (Algorithm 3). We set the range of a window from 100 to 300, with a stride of 20 % of the window size.

In NS-Batch, we take the 200 events adjacent to each batch as the test set and divide it into 5 test units. We feed a test unit into the model for testing after every 20 epochs of training. Subsequently, we feed these 200 events into the model as the next training batch. In NS-Slide and NS-AdaSlide, we perform 20 training epochs on each window. In the sliding window, we select the 40 events adjacent to each window as the test set. Subsequently, we add these

40 events to the next window for training. In `NS-AdaSlide`, we select the last 20% of events within a window as the test set. Based on the above settings, we ensure equal training iterations for each event in each method for fair comparison. We use link prediction as the downstream task and set the number of negative samples to 5. In performance experiments, we also compare the batch method based on the open-source implementations of the three models: [1], [2], [3]. These models are implemented based on PyTorch [41], so we denote them as `Torch-Batch`. NeutronStream also uses the C++ version LibTorch of PyTorch as the underlying operator library. Therefore, it is fair to compare NeutronStream with PyTorch. The open-source codes cannot be further optimized in PyTorch, because the key to affecting the training performance of dynamic GNNs is the sequential nature of time-dependent events.

## 6.2 Runtime Comparison

We first compare the overall performance of the three training methods implemented based on NeutronStream and the batch method implemented based on open-source codes. Figure 10 reports the runtime of training one epoch on different datasets. All the per-epoch runtime results are measured by averaging results of 10 epochs.

Compared to `Torch-Batch`, `NS-Batch` can achieve 1.53X-2.26X speedups over DyRep, 1.48X-2.33X speedups over LDG, and 3.01X-5.87X speedups over DGNN, demonstrating the effectiveness of our system optimization. LDG and DyRep share the same underlying code and storage structures, so their performances are similar. However, DGNN involves more complex computations and performs more read and write operations, so the speedup of our system is more obvious. In `NS-AdaSlide`, the window captures the locality between events, resulting in lower parallelism, thus requiring a longer training time. However, thanks to our system optimization, `NS-AdaSlide` can also achieve speedup improvements compared to `Torch-Batch`. `NS-AdaSlide` achieves 1.28X-1.71X speedups over DyRep, 1.27X-1.72X speedups over LDG, and 2.53X-4.44X speedups over DGNN. DyRep and LDG use uncompressed matrices to store graph topology and node embeddings. They need to perform multiple memory allocation and copy operations to store the updated embeddings. Therefore, they report the "Out-Of-Memory" error on two large datasets "Reality" and "Slashdot". Compared with the open-source codes of DGNN, `NS-Batch` achieves speedup 3.20X and 5.87X on these two datasets, respectively. Our framework can support training on larger datasets with more events, which demonstrates the scalability of our approach.

## 6.3 AUC Comparison

We use the Area Under the ROC Curve (AUC) metric on the test set to evaluate the performance of models. We sum the highest AUC for each batch or window and compute the average as a result, as shown in Table 3. `Torch-Batch` and `NS-Batch` represent the batch training implemented based on PyTorch and our system, respectively. They use the same training method, so their AUC performance is identical. Among the three methods, the batch method has the lowest AUC because it cuts off the data stream, resulting in the loss of training information. The sliding window method reduces information loss and improves AUC compared with the batch method. The adaptive sliding method achieves the highest AUC by effectively capturing the spatial-temporal locality between events. The experimental

**Table 3: AUC comparison of three training methods**

| Model | Dataset | Training Method | | | |
|---|---|---|---|---|---|
| | | Torch-Batch | NS-Batch | NS-Slide | NS-AdaSlide |
| DyRep | Social | 83.01% | 83.01% | 86.15% | **89.32%** |
| | Github | 73.46 % | 73.46% | 77.54% | **79.28%** |
| | DNC | 63.15% | 63.15% | 63.49% | **66.18%** |
| | UCI | 62.46% | 62.46% | 63.53% | **65.68%** |
| LDG | Social | 87.07% | 87.07% | 88.52% | **92.98%** |
| | Github | 74.34% | 74.34% | 78.10% | **79.16%** |
| | DNC | 64.62% | 64.62% | 66.50% | **69.41%** |
| | UCI | 62.16% | 62.16% | 64.38 % | **66.57%** |
| DGNN | Social | 97.21% | 97.21% | 97.61% | **97.67%** |
| | Github | 81.94% | 81.94% | 82.54% | **84.45%** |
| | DNC | 86.04% | 86.04 % | 87.72% | **88.81%** |
| | UCI | 78.45% | 78.45% | 81.14 % | **82.09%** |

results demonstrate the effectiveness of the sliding window method and the adaptive sliding window method.

## 6.4 Performance Gain Analysis

We quantify the gains of event-parallel execution and pipeline optimization separately in the adaptive window method. We take the non-optimized version (`Base`) as the baseline and gradually integrate these two optimization methods into the baseline version. DyRep, LDG, and DGNN show similar trends, so we show the results of DyRep. Figure 12 shows the normalized speedup over the raw base processing. Compared with `Base`, the parallel optimization (`Base+Parallel`) achieves 1.19X-1.43X speedups. On the Social dataset, the parallel speedup is lower because it is a small dataset with more dependencies between events. On the DNC dataset, the parallel speedup is higher because the dataset is sparse, and there are fewer dependencies between events. In addition, since our experiments are performed in the adaptive window method, this further limits parallel optimization. The pipeline optimization (`Base+Parallel +Pipeline`) can further achieve 1.08X-1.28X speedups compared to the parallel optimization. As the graph size increases, the time required for dependency analysis also increases, making the pipeline speedup more obvious. Therefore, the speedup from pipelining is more significant on the UCI dataset. Regarding the Social dataset, the speedup is relatively low due to its small size.

## 6.5 Evaluation the Impact on Parameters

**Varying Sliding Stride in `NS-AdaSlide`.** The stride determines the sliding stride of the window each time. We set the sliding stride as 0.2, 0.5, and 0.8 of the window size $W$ respectively to study the effect of sliding stride. As shown in Figure 15, the smaller the sliding stride, the less information is lost in model training. Therefore, the model AUC is the highest when the stride is $0.2 * W$, and the model AUC is the lowest when the stride is $0.8 * W$. Although a smaller stride leads to higher model AUC, it also results in a longer training time. Our results show that setting the stride to $0.5 * W$ achieves an average speedup of 2.55X compared to setting the stride to $0.2 * W$. The stride of $0.8 * W$ achieves an average speedup of 4.09X over the stride of $0.2 * W$. Therefore, the sliding stride setting balance the training efficiency and the training accuracy.

**Varying Sliding Stride in `NS-Slide`.** We set the sliding stride as 0.2, 0.5, and 0.8 of the window size $W$, where $W$ equals 200. `NS-AdaSlide` and `NS-Slide` show the same trends in accuracy and performance when varying sliding stride. The smaller the stride, the less information is lost and the accuracy is higher, but it also
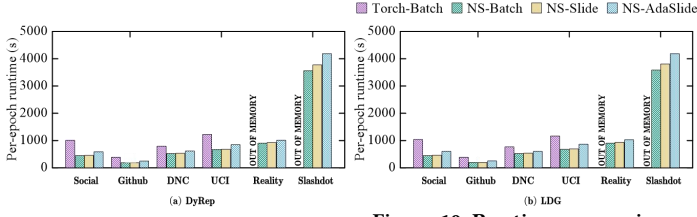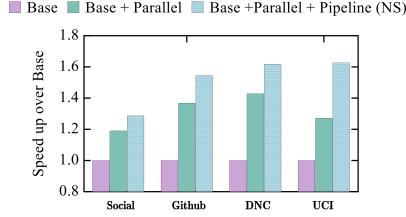
**Figure 10: Runtime comparison.**



**Figure 11: Varying window size in `NS-Slide`.**
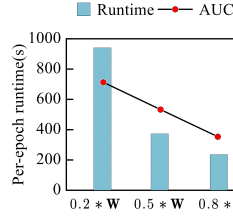


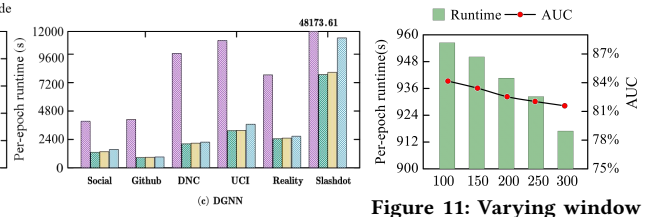**Figure 12: Performance breakdown analysis.**



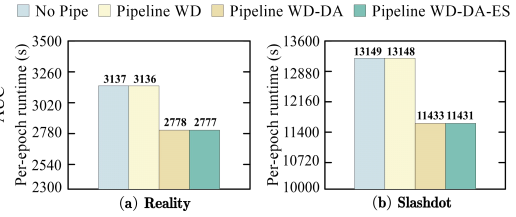**Figure 13: Varying sliding stride in `NS-Slide`.**



**Figure 14: Pipeline ablation.**

results in longer training time. As shown in Figure 13, the accuracy is the highest when the stride is 0.2, but it also takes the most training time. The accuracy is the lowest when the stride is 0.8, but it takes the shortest training time. In addition, the accuracy of varying stride is 84.45%, 79.60%, and 74.58% in NS-AdaSlide, respectively. The accuracy of varying stride is 82.54%, 77.85%, and 73.19% in NS-Slide, respectively. NS-AdaSlide achieves higher accuracy than NS-Slide when the ratio of stride is the same, which demonstrates the effectiveness of the adaptive sliding window method.

**Varying Window Size.** We vary the window size between 100 and 300, and set the sliding stride to 0.2 of the window size. We show the AUC and the per-epoch runtime of DGNN on Github, as shown in Figure 11. The accuracy increases when the window size decreases from 300 to 100. This is because Github has low spatial-temporal locality, and smaller windows can sufficiently capture the locality among events. Smaller windows also have a smaller stride, reducing information loss. In addition, when the window size varies from 100 to 300, the sliding window method achieves the highest accuracy of 84.17% when the window size is 100. The adaptive sliding window method can reach 84.45%, which demonstrates the effectiveness of the adaptive method. In performance, different window sizes show similar running times as their training events are similar. In addition, as the window size increases, the parallelism increases, so the window size of 300 shows the shortest runtime.

**Varying Parameters $H$ and $L$.** We set five ranges: [50, 300], [100, 300], [150, 300], [100, 250], [100, 350]. The left number indicates the setting of $L$, and the right number indicates the setting of $H$. We set the sliding stride to 0.2 of the window size. We show the AUC and the per-epoch runtime of DGNN on Github, as shown in Table 4. When fixing $H$ and varying $L$, a smaller $L$ leads to higher accuracy. This is because Github has low spatial-temporal locality, and smaller $L$ can better capture the locality among events. However, when fixing $L$ and varying $H$, they show the same accuracy since each window does not contain more than 250 events. Therefore, the parameter $L$ significantly impacts accuracy for datasets with low temporal-spatial locality. In performance, different settings show similar runtimes as their training events are similar. In addition, as the parameter $L$ increases, the parallelism increases, so the setting with [150,300] shows the shortest runtime.

**Table 4: Performance and accuracy when varying $L$ and $H$ (DGNN on Github)**

| | Varying Window Size Parameters $H$ and $L$ in `NS-AdaSlide` | | | | |
| --- | --- | --- | --- | --- | --- |
| | [50, 300] | [100, 300] | [150, 300] | [100, 250] | [100, 350] |
| Accuracy (%) | 85.61 | 84.45 | 83.47 | 84.45 | 84.45 |
| Peformance (s) | 1015.13 | 963.02 | 950.15 | 966.59 | 959.68 |

**Table 5: Graph query performance comparison**

| Model | Dataset | Query Time (s) | |
| --- | --- | --- | --- |
| | | Torch_Query | NeutronStream_Query |
| DyRep | Social | 281.40 | 114.38 |
| | Github | 101.76 | 47.29 |
| | DNC | 143.78 | 101.62 |
| | UCI | 223.04 | 141.03 |
| LDG | Social | 278.46 | 114.85 |
| | Github | 108.2 | 46.88 |
| | DNC | 146.75 | 101.68 |
| | UCI | 220.71 | 140.73 |
| DGNN | Social | 762.69 | 329.30 |
| | Github | 767.71 | 200.58 |
| | DNC | 1968.62 | 419.90 |
| | UCI | 2601.55 | 509.09 |

## 6.6 Performance of Dynamic Graph Storage

We evaluate the graph storage performance of NeutronStream by comparing it with open-source codes in the batch training. We regard reading and writing graph topology and node embeddings during training as graph queries. `NS-Batch` maintains the same sequential processing events as the PyTorch-based to ensure fairness, denoted as `NeutronStream_Query`. Table 5 shows the per-epoch query time. NeutronStream can achieve 1.42X-2.46X speedups over DyRep, 1.44X-2.42X speedups over LDG, and 2.32X-5.11X speedups over DGNN. LDG and DyRep have the same underlying storage, so NeutronStream demonstrates similar speedups over these two models. DyRep and LDG use uncompressed matrices to store graph topology and node embeddings. They are highly inefficient in storing updated embeddings, requiring multiple memory allocations and copy operations. When computing each event, they first need to create a new matrix and copy from the previous event's embedding matrix, resulting in redundant storage. DGNN uses a row-based sparse matrix (`scipy.sparse.lil_matrix`) to store graph topology. During graph query operations, DGNN needs to first convert the `lil_matrix` to a COO-format matrix and then perform query operations. This model maintains multiple embeddings for each node. During the computation of an event, this model needs to read
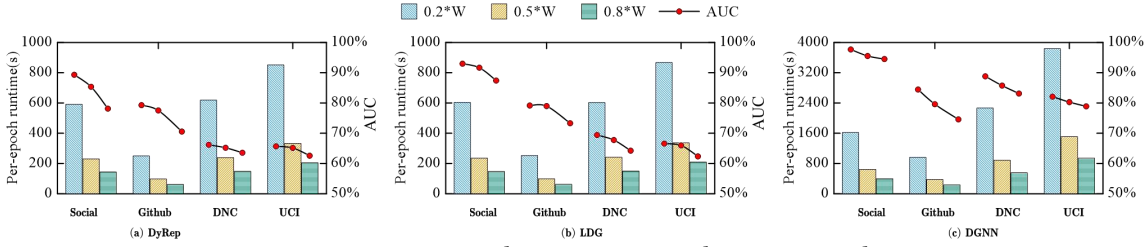
**Figure 15: Runtime and AUC comparison when varying stride**

and write the embeddings of the event nodes and their neighbors, resulting in complex read-and-write operations. In addition, these models all require determining the total number of nodes before training, so they do not support adding/removing nodes.

### 6.7 Analysis of Pipeline Benefits

The training process of a window can be divided into four components: Window determination (WD), Dependency analysis (DA), Event scheduling (ES), and Parallel training (PT). We perform pipelined ablation experiments of DGNN on Reality and Slashdot. Figure 14 reports the per-epoch runtime when adding each component sequentially to the pipeline. No Pipe indicates four components are executed sequentially. Pipeline WD indicates the window determination component is added to the pipeline. Pipeline WD-EA indicates both the window determination and the dependency analysis are added to the pipeline. Pipeline WD-DA-ES indicates the window determination, the dependency analysis, and the event scheduling are added to the pipeline. No Pipe is the slowest one due to its totally sequential execution. Pipeline WD does not reduce the runtime obviously since WD is fast. DA is a costly step, so adding DA to the pipeline can significantly reduce the runtime. Similar to WD, ES is also a lightweight step, so adding ES to the pipeline only slightly reduces the runtime. The results show that the benefits of pipeline optimization mainly come from overlapping DA.

## 7 RELATED WORK

**Static GNN Frameworks.** Static GNNs are a class of deep learning models to learn from static graphs [6, 18, 26, 30, 33, 52, 60, 62, 64, 69]. Many frameworks are designed specifically for static GNNs [9, 11, 16, 23, 31, 32, 40, 42, 43, 55, 61, 65, 66, 70–73, 76]. PyG [9] is a GNN library based on PyTorch, which provides a general message-passing interface for users. DGL [55] provides the graph as the programming abstraction. It provides users with flexible APIs to implement arbitrary message-passing computation. It also provides a distributed framework DistDGL [74]. Sancus [42] is a decentralized full-graph GNN training framework. It introduces history embedding and actively creates asynchrony, which avoids a lot of communication and speeds up training. PASCA [70] proposes a new Scalable Graph Neural Architecture Paradigm. It implements an automated search engine to systematically explore high-performance and scalable GNN structures. NeutronStar [56] proposes a hybrid dependency management approach which can adaptively select an appropriate dependency strategy for different dependent neighbors to accelerate distributed GNN training. Although these frameworks can effectively support static GNN training, their underlying storage and execution engines cannot effectively support dynamic graph updates and training of dynamic GNNs.

**Dynamic GNN Frameworks.** Dynamic graphs represent graphs evolving over time. They can be categorized into snapshot-based dynamic graphs and continuous-time dynamic graphs based on

discrete or continuous timestamps [17, 78]. The snapshot-based dynamic graph is a list of snapshots, each of which keeps the graph state at a certain moment. For a graph whose nodes or edges are updated frequently, it is more efficient to use this method for storage. There is a class of dynamic GNN models specifically designed for the snapshot-based dynamic graph, such as EvolveGCN [38], DySAT [48], and DynGEM [13]. The continuous-time dynamic graph can be viewed as a stream of graph update events. This representation can capture all temporal information compared with the snapshot-based dynamic graph. There is a class of dynamic GNN models designed for the continuous-time dynamic graph, such as DyRep [51], DGNN [28], TGAT [63], Zebra [24] and TE-DyGE [54]. There are a limited number of frameworks available for dynamic GNNs. PyGT [47], DynaGraph [14] and PiPAD [53] are specifically designed for training on snapshot-based dynamic graphs. Cambricon-G [50] designs a specialized accelerator to optimize various variant models of GCN [18], such as GraphSage [15], DiffPool [68], DGMG [25], and EdgeConv [58]. They do not support training models designed for graph streams. TGL [75] is a temporal GNN training framework. It introduces Temporal-CSR, a data structure that stores temporal graphs and supports parallel sampling. This structure requires storing the entire dynamic graph statically before training. To speed up the training, TGL proposes an intra-batch parallel training method, disregarding the temporal dependencies between events within a batch.

## 8 CONCLUSIONS AND FUTURE WORK

We present NeutronStream, a dynamic GNN training framework. The effectiveness of NeutronStream and its high performance are contributed by several components, including a sliding-window-based method to incrementally train models for capturing the spatial-temporal dependency of events, a fine-grained event parallel processing scheme and a series of system optimizations. We evaluate NeutronStream on three dynamic GNN models, DyRep, LDG, and DGNN. The experimental results demonstrate that our optimized sliding-window-based training brings 3.97% accuracy improvements on average. Compared with their open-sourced implementations with PyTorch, NeutronStream achieves 2.26X-12.35X speedups. The learning models for graph streams are more desirable in real-world application scenarios. We believe that the online training support for GNNs is promising and can boost the wide applications of GNNs. Our future work will be tackling the challenges of online training for dynamic graphs.

# REFERENCES

[1] [n.d.]. https://github.com/Harryi0/dyrep_torch.git.

[2] [n.d.]. https://github.com/uoguelph-mlrg/LDG.git.

[3] [n.d.]. https://github.com/alge24/DyGNN.git.

[4] [n.d.]. Github. https://www.gharchive.org/.

[5] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

[6] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *NIPS*.

[7] Natha1n Eagle and Alex Pentland. 2006. Reality mining: sensing complex social systems. *Pers. Ubiquitous Comput.* 10, 4 (2006), 255–268. https://doi.org/10.1007/s00779-005-0046-3

[8] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph neural networks for social recommendation. In *The world wide web conference*. 417–426.

[9] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[10] Per Fuchs, Domagoj Margan, and Jana Giceva. 2023. Sortledton: a Universal Graph Data Structure. *SIGMOD Rec.* 52, 1 (2023), 17–25.

[11] Swapnil Gandhi and Anand Padmanabha Iyer. 2021. P3: Distributed Deep Graph Learning at Scale. In *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, Angela Demke Brown and Jay R. Lorch (Eds.). USENIX Association, 551–568.

[12] Vicenç Gómez, Andreas Kaltenbrunner, and Vicente López. 2008. Statistical analysis of the social network and discussion threads in slashdot. In *Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, 2008*, Jinpeng Huai, Robin Chen, Hsiao-Wuen Hon, Yunhao Liu, Wei-Ying Ma, Andrew Tomkins, and Xiaodong Zhang (Eds.). ACM, 645–654.

[13] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. DynGEM: Deep Embedding Method for Dynamic Graphs. *CoRR* abs/1805.11273 (2018). arXiv:1805.11273 http://arxiv.org/abs/1805.11273

[14] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: dynamic graph neural networks at scale. In *GRADES-NDA '22: Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), Philadelphia, Pennsylvania, USA, 12 June 2022*, Vasiliki Kalavri and Semih Salihoglu (Eds.). ACM, 6:1–6:10.

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

[16] Zhihao Jia, Sina Lin, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2020. Improving the Accuracy, Scalability, and Performance of Graph Neural Networks with Roc. In *In Proceedings of Machine Learning and Systems*.

[17] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. 2020. Representation Learning for Dynamic Graphs: A Survey. *J. Mach. Learn. Res.* 21, 70 (2020), 1–73.

[18] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=SJU4ayYgl

[19] Boris Knyazev, Carolyn Augusta, and Graham W Taylor. 2021. Learning temporal attention in dynamic graphs with bilinear interactions. *Plos one* 16, 3 (2021), e0247936.

[20] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2018. Learning dynamic embeddings from temporal interactions. *arXiv preprint arXiv:1812.02289* (2018).

[21] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web* (Rio de Janeiro, Brazil) *(WWW '13 Companion)*. Association for Computing Machinery, New York, NY, USA, 1343–1350. https://doi.org/10.1145/2487788.2488153

[22] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066.

[23] Hongzheng Li, Yingxia Shao, Junping Du, Bin Cui, and Lei Chen. 2022. An I/O-efficient disk-based graph system for scalable second-order random walk of large graphs. *arXiv preprint arXiv:2203.16123* (2022).

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

[25] Yujia Li, Oriol Vinyals, Chris Dyer, Razvan Pascanu, and Peter Battaglia. 2018. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324* (2018).

[26] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. 2016. Gated Graph Sequence Neural Networks. In *Proceedings of ICLR'16*.

[27] Yuanfu Lu, Xiao Wang, Chuan Shi, Philip S Yu, and Yanfang Ye. 2019. Temporal network embedding with micro-and macro-dynamics. In *Proceedings of the 28th ACM international conference on information and knowledge management*. 469–478.

[28] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. 2020. Streaming graph neural networks. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 719–728.

[29] Anmol Madan, Manuel Cebrian, Sai Moturu, Katayoun Farrahi, et al. 2011. Sensing the" health state" of a community. *IEEE Pervasive Computing* 11, 4 (2011), 36–45.

[30] D Marcheggiani and I Titov. 2017. Encoding sentences with graph convolutional networks for semantic role labeling. In *EMNLP 2017-Conference on Empirical Methods in Natural Language Processing, Proceedings*. 1506–1515.

[31] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: scaling out huge embedding model training via cache-enabled distributed framework. *arXiv preprint arXiv:2112.07221* (2021).

[32] Xupeng Miao, Wentao Zhang, Yuezihan Jiang, Fangcheng Fu, Yingxia Shao, Lei Chen, Yangyu Tao, Gang Cao, and Bin Cui. 2022. P2CG: a privacy preserving collaborative graph neural network training framework. *The VLDB Journal* (2022), 1–20.

[33] Xupeng Miao, Wentao Zhang, Yingxia Shao, Bin Cui, Lei Chen, Ce Zhang, and Jiawei Jiang. 2021. Lasagne: A multi-layer graph convolutional network framework via node-aware deep architecture. *IEEE Transactions on Knowledge and Data Engineering* (2021).

[34] Alice Moallemy-Oureh, Silvia Beddar-Wiesing, Rüdiger Nather, and Josephine M. Thomas. 2022. FDGNN: Fully Dynamic Graph Neural Network. *CoRR* abs/2206.03469 (2022). https://doi.org/10.48550/arXiv.2206.03469

[35] Federico Monti, Michael M. Bronstein, and Xavier Bresson. 2017. Geometric Matrix Completion with Recurrent Multi-Graph Neural Networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 3697–3707.

[36] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, Pierre-Antoine Champin, Fabien Gandon, Mounia Lalmas, and Panagiotis G. Ipeirotis (Eds.). ACM, 969–976. https://doi.org/10.1145/3184558.3191526

[37] Yan Pang, Ai Shan, Zhen Wang, Mengyu Wang, Jianwei Li, Ji Zhang, Teng Huang, and Chao Liu. 2022. Sparse-Dyn: Sparse dynamic graph multirepresentation learning via event-based sparse temporal attention network. *International Journal of Intelligent Systems* 37, 11 (2022), 8770–8789.

[38] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. 2020. EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 5363–5370. https://ojs.aaai.org/index.php/AAAI/article/view/5984

[39] Namyong Park, Andrey Kan, Xin Luna Dong, Tong Zhao, and Christos Faloutsos. 2019. Estimating Node Importance in Knowledge Graphs Using Graph Neural Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2019, Anchorage, AK, USA, August 4-8, 2019*, Ankur Teredesai, Vipin Kumar, Ying Li, Rómer Rosales, Evimaria Terzi, and George Karypis (Eds.). ACM, 596–606.

[40] Yeonhong Park, Sunhong Min, and Jae W. Lee. 2022. Ginex: SSD-enabled Billion-scale Graph Neural Network Training on a Single Machine via Provably Optimal In-memory Caching. *Proc. VLDB Endow.* 15, 11 (2022), 2626–2639.

[41] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: an imperative style, high-performance deep learning library. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. 8026–8037.

[42] Jingshu Peng, Zhao Chen, Yingxia Shao, Yanyan Shen, Lei Chen, and Jiannong Cao. 2022. SANCUS: Staleness-Aware Communication-Avoiding Full-Graph Decentralized Training in Large-Scale Graph Neural Networks. *Proc. VLDB Endow.* 15, 9 (2022), 1937–1950.

[43] Jingshu Peng, Yanyan Shen, and Lei Chen. 2021. GraphANGEL: Adaptive aNd Structure-Aware Sampling on Graph NEuraL Networks. In *2021 IEEE International Conference on Data Mining (ICDM 21)*. 479–488.

[44] Jiarui Qin, Weinan Zhang, Xin Wu, Jiarui Jin, Yuchen Fang, and Yong Yu. 2020. User Behavior Retrieval for Click-Through Rate Prediction. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval, SIGIR 2020, Virtual Event, China, July 25-30, 2020*, Jimmy X. Huang, Yi Chang, Xueqi Cheng, Jaap Kamps, Vanessa Murdock, Ji-Rong Wen,

and Yiqun Liu (Eds.). ACM, 2347–2356.

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

[46] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[47] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Sinziana Astefanoaei, Oliver Kiss, Ferenc Béres, Guzmán López, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021*, Gianluca Demartini, Guido Zuccon, J. Shane Culpepper, Zi Huang, and Hanghang Tong (Eds.). ACM, 4564–4573.

[48] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. 2020. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 519–527.

[49] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *European semantic web conference*. Springer, 593–607.

[50] Xinkai Song, Tian Zhi, Zhe Fan, Zhenxing Zhang, Xi Zeng, Wei Li, Xing Hu, Zidong Du, Qi Guo, and Yunji Chen. 2022. Cambricon-G: A Polyvalent Energy-Efficient Accelerator for Dynamic Graph Neural Networks. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 41, 1 (2022), 116–128.

[51] Rakshit S. Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR*.

[52] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[53] Chunyang Wang, Desen Sun, and Yuebin Bai. 2023. PiPAD: Pipelined and Parallel Dynamic GNN Training on GPUs. *arXiv preprint arXiv:2301.00391* (2023).

[54] Liping Wang, Yanyan Shen, and Lei Chen. [n.d.]. TE-DyGE: Temporal Evolution-Enhanced Dynamic Graph Embedding Network. In *Database Systems for Advanced Applications: 28th International Conference, DASFAA 2023, Tianjin, China, April 17–20, 2023, Proceedings, Part III*. 183–198.

[55] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv preprint arXiv:1909.01315* (2019).

[56] Qiange Wang, Yangfeng Zhang, Hao Wang, Chaoyi Chen, Xiaodong Zhang, and Ge Yu. 2022. NeutronStar:Distributed GNN Training with Hybrid Dependency Management. In *ACM International Conference on Management of Data (SIGMOD 22)*. 770–778.

[57] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2628–2638.

[58] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E Sarma, Michael M Bronstein, and Justin M Solomon. 2019. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (tog)* 38, 5 (2019), 1–12.

[59] Zhouxia Wang, Tianshui Chen, Jimmy S. J. Ren, Weihao Yu, Hui Cheng, and Liang Lin. 2018. Deep Reasoning with Knowledge Graph for Social Relationship Understanding. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, Jérôme Lang (Ed.). ijcai.org, 1021–1028.

[60] Shiwen Wu, Wentao Zhang, Fei Sun, and Bin Cui. 2020. Graph Neural Networks in Recommender Systems: A Survey. *arXiv preprint arXiv:2011.02260* (2020).

[61] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chengguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: vertex-centric programming for graph neural networks. In *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar (Eds.). ACM, 359–375.

[62] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.

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

[64] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2018. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.

[65] Jianbang Yang, Dahai Tang, Xiaoniu Song, Lei Wang, Qiang Yin, Rong Chen, Wenyuan Yu, and Jingren Zhou. 2022. GNNLab: a factored system for sample-based GNN training over GPUs. In *EuroSys '22: Seventeenth European Conference on Computer Systems, Rennes, France, April 5 - 8, 2022*, Yérom-David Bromberg, Anne-Marie Kermarrec, and Christos Kozyrakis (Eds.). ACM, 417–434.

[66] Hongbo Yin, Yingxia Shao, Xupeng Miao, Yawen Li, and Bin Cui. 2022. Scalable Graph Sampling on GPUs with Compressed Graph. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 2383–2392.

[67] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 974–983.

[68] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. 2018. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems* 31 (2018).

[69] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 793–803.

[70] Wentao Zhang, Yu Shen, Zheyu Lin, Yang Li, Xiaosen Li, Wen Ouyang, Yangyu Tao, Zhi Yang, and Bin Cui. 2022. Pasca: A graph neural architecture search system under the scalable paradigm. In *Proceedings of the ACM Web Conference 2022*. 1817–1828.

[71] Xin Zhang, Yanyan Shen, and Lei Chen. 2022. Feature-Oriented Sampling for Fast and Scalable GNN Training. In *2022 IEEE International Conference on Data Mining (ICDM)*. 723–732.

[72] Xin Zhang, Yanyan Shen, Yingxia Shao, and Lei Chen. 2023. DUCATI: A Dual-Cache Training System for Graph Neural Networks on Giant Graphs with GPU. In *ACM International Conference on Management of Data (SIGMOD 23)*. 479–488.

[73] Chenguang Zheng, Hongzhi Chen, Yuxuan Cheng, Zhezheng Song, Yifan Wu, Changji Li, James Cheng, Hao Yang, and Shuai Zhang. 2022. ByteGNN: Efficient Graph Neural Network Training at Large Scale. *Proc. VLDB Endow.* 15, 6 (2022), 1228–1242.

[74] Da Zheng, Chao Ma, Minjie Wang, Jinjing Zhou, Qidong Su, Xiang Song, Quan Gan, Zheng Zhang, and George Karypis. 2020. DistDGL: Distributed Graph Neural Network Training for Billion-Scale Graphs. *arXiv preprint* arXiv 2010.05337 (2020).

[75] Hongkuan Zhou, Da Zheng, Israt Nisa, Vassilis N. Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A General Framework for Temporal GNN Training onBillion-Scale Graphs. *Proc. VLDB Endow.* 15, 8 (2022), 1572–1580.

[76] Rong Zhu, Kun Zhao, Hongxia Yang, Wei Lin, Chang Zhou, Baole Ai, Yong Li, and Jingren Zhou. 2019. AliGraph: A Comprehensive Graph Neural Network Platform. *In Proceedings of the VLDB Endowment* 12, 12 (2019), 2094–2105.

[77] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034.

[78] Yuecai Zhu, Fuyuan Lyu, Chengming Hu, Xi Chen, and Xue Liu. 2022. Encoder-Decoder Architecture for Supervised Dynamic Graph Learning: A Survey. *CoRR* abs/2203.10480 (2022).

[79] Yuan Zuo, Guannan Liu, Hao Lin, Jia Guo, Xiaoqian Hu, and Junjie Wu. 2018. Embedding temporal network via neighborhood formation. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2857–2866.