TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs [Scalable Data Science]

Hongkuan Zhou*
University of Southern California
hongkuaz@usc.edu

Da Zheng
AWS AI
dzzhen@amazon.com

Israt Nisa
AWS AI
nisisrat@amazon.com

Vasileios Ioannidis
AWS AI
ivasilei@amazon.com

Xiang Song
AWS AI
xiangsx@amazon.com

George Karypis
AWS AI
gkarypis@amazon.com

ABSTRACT

Many real world graphs contain time domain information. Temporal Graph Neural Networks capture temporal information as well as structural and contextual information in the generated dynamic node embeddings. Researchers have shown that these embeddings achieve state-of-the-art performance in many different tasks. In this work, we propose TGL, a unified framework for large-scale offline Temporal Graph Neural Network training where users can compose various Temporal Graph Neural Networks with simple configuration files. TGL comprises five main components, a temporal sampler, a mailbox, a node memory module, a memory updater, and a message passing engine. We design a Temporal-CSR data structure and a parallel sampler to efficiently sample temporal neighbors to form training mini-batches. We propose a novel random chunk scheduling technique that mitigates the problem of obsolete node memory when training with a large batch size. To address the limitations of current TGNNs only being evaluated on small-scale datasets, we introduce two large-scale real-world datasets with 0.2 and 1.3 billion temporal edges. We evaluate the performance of TGL on four small-scale datasets with a single GPU and the two large datasets with multiple GPUs for both link prediction and node classification tasks. We compare TGL with the open-sourced code of five methods and show that TGL achieves similar or better accuracy with an average of 13× speedup. Our temporal parallel sampler achieves an average of 173× speedup on a multi-core CPU compared with the baselines. On a 4-GPU machine, TGL can train one epoch of more than one billion temporal edges within 1-10 hours. To the best of our knowledge, this is the first work that proposes a general framework for large-scale Temporal Graph Neural Networks training on multiple GPUs.

PVLDB Reference Format:
Hongkuan Zhou, Da Zheng, Israt Nisa, Vasileios Ioannidis, Xiang Song, and George Karypis. TGL: A General Framework for Temporal GNN Training on Billion-Scale Graphs [Scalable Data Science]. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

*The work was performed during an internship at AWS AI Lab.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tedzhouhk/TGL.

1 INTRODUCTION

Graph Neural Networks (GNNs) have proven to be powerful and reliable method in representation learning on static graphs and are widely used in many academic and industrial problems. There exist well-developed libraries like DGL [22] and PyG [2] that allow users to quickly and efficiently implement GNN variants for static graphs and deploy them to CPUs, GPUs, or even distributed systems. There are also multiple benchmark dataset collections like OGB [4, 6] that provide large-scale and wide-ranging datasets to evaluate the performance of GNN variants for static graphs.

However, many real-world graphs are dynamic. For example, in a social network new users join over time and users interact with each other on posts and send messages. In a knowledge graph, new events appear and are only valid for specific periods of time. The dynamics in the user-item graph reveal important information in identifying abusive behaviors [20]. To capture the evolving nature on dynamic graphs, recently, researchers [1, 13, 15, 17] have developed Temporal Graph Neural Networks (TGNNs) which jointly learn the temporal, structural, and contextual relationships on dynamic graphs. Like static GNNs, TGNNs encode graph information at a given time into dynamic node embeddings. With the additional temporal information, TGNNs outperform static GNNs on link prediction tasks, dynamic node classification tasks, and many other tasks on dynamic graphs. These works on temporal graph representation learning are developed using different frameworks with different levels of optimizations and parallelization and are evaluated on small dynamic graphs which only contain less than ten thousand nodes and one million edges.

Many real-world dynamic graphs, such as social network graphs and knowledge graphs, usually have millions of nodes and billions of edges. It is challenging to scale TGNN training to large graphs for multiple reasons. First, the additional temporal dependency requires training to be done chronologically. TGNNs also use more expensive neighbor samplers which select temporal neighbors based on the timestamps of the interactions between nodes. Moreover, different TGNNs capture the temporal information using different strategies like node memory and snapshot. Existing graph deep learning frameworks like DGL and PyG do not provide efficient data structure, sampler, and message passing primitive for dynamic graphs, which requires users to implement extra modules to compose TGNN models. In addition, it is also challenging to design an efficient and versatile framework that is capable of unifying the different schemes of different TGNN variants. Recently, PyTorch Geometric Temporal (PyGT) [16] attempted to design a library for dynamic and temporal geometric deep learning on top of PyG. However, PyGT only supports discrete time snapshot-based method and full batch training on small-scale spatial-temporal graphs.

To fill these gaps, we develop TGL, the first general framework for large-scale offline TGNNs training. In this work, we focus on the widely used edge-timestamped dynamic graphs where each edge is associated with a timestamp. TGL supports all TGNN variants that aggregate and refine information from maintained states or features of selected temporal neighbors. The survey [9] categorizes dynamic graphs into Continuous Time Dynamic Graphs (CTDGs) and Discrete Time Dynamic Graphs (DTDGs) based on the continuous or discrete quantity of the timestamps. However, we believe that DTDGs are essentially CTDGs with granulated timestamps. Hence, we design TGL to support the more general CTDGs and evaluate TGL by comparing the performance of TGNN variants targeting both CTDGs and DTDGs in the experiments. Our main contributions are

- We design a unified framework that supports efficient training on most TGNN architectures by studying the characteristic of a diverse set of TGNNs variants including snapshot-based TGNNs [13, 17], time encoding-based TGNNs [1, 15, 23], and memory-based TGNNs [10, 15, 18, 23].
- We design a CSR-based data structure for rapid access to temporal neighbors and a parallel sampler that support different temporal neighbor sampling algorithms. Our parallel sampler can quickly locate the temporal edges to sample from by maintaining auxiliary pointer arrays.
- We propose a novel random chunk scheduling technique that overcomes the deprivation of intra-dependency when training with a large batch size for the methods using node memory, which enables multi-GPU training on large-scale dynamic graphs.
- To better compare the performance of various TGNN methods, we introduce two large-scale datasets with billions of edges – the GDELT and MAG datasets which represent dynamic graphs with long time duration and dynamic graphs with larger number of nodes.

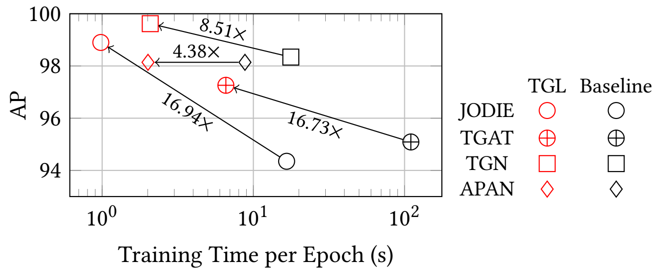

Figure 1: Accuracy and per epoch training time of TGL compared with the baselines on the Wikipedia dataset (600 batch size).

	[13]	[17]	[10]	[18]	[17]	[1]	[15]	[23]
snapshot	✓	✓			✓			
time encoding						✓	✓	✓
memory			✓	✓			✓	✓

Table 1: Strategies used by various TGNN variants.

- We compare the performance of TGL with the baseline open-sourced codes on two small-scale datasets. TGL achieves similar or higher accuracy for all baseline methods with an average speedup of 13× as shown in Figure 1. On the large-scale datasets, TGL achieves an average of 2.3× speedup when using 4 GPUs.

2 TEMPORAL GRAPH NEURAL NETWORKS

TGNNs generate dynamic node embeddings by adding components like time encoder and node memory in the message passing flow or combining information from multiple consecutive graph snapshots.. In general, TGNNs generate dynamic node embeddings by iteratively processing the information gathered from temporal neighbors, similarly to static GNNs. To capture the additional temporal dependencies, TGNNs usually process the neighbor information by three methods: 1) group the neighbors in the past according to their time and learn sequences from the groups (snapshot-based TGNNs), 2) add additional time encoding to each past neighbor, and 3) maintain node memory which summarizes the current state of each node (memory-based TGNNs). Table 1 shows different strategies used by different TGNN variants. Note that some TGNNs [15, 23] use combinations of multiple strategies to intensify the temporal relationships, while some other TGNNs only rely on a single strategy. For example, pure memory TGNNs [10, 18, 23] directly use the node memory as the dynamic node embeddings, potentially with complex COMB and UPDT function to update node memory. For example, in APAN [23], the mails are delivered to the mailboxes of hop-1 neighbors and the COMB function applies attention mechanism to update the node memory. After studying the architecture of different TGNNs, we identify three components that form a unified representation for most TGNN variants – the node memory, the attention aggregator, and the temporal sampler. For snapshot-based TGNNs [13, 17], each snapshot is treated independently while the output of each snapshot is combined to produce the final node embeddings.

2.1 Node Memory

For nodes with different history lengths, a fixed number of temporal neighbors may not provide enough information to generate the

Symbol	Description
u, v, i, j	Nodes in dynamic graphs
e	Edges in dynamic graphs
$\boldsymbol{v}_v, \boldsymbol{e}_{ij}$	Node feature of v and edge feature of edge ij
$\mathcal{N}(v)$	Set of past neighbors of node v
m_e^{uv}, m_e^{vu}	Mails generated at the source and destination nodes
s_v	Node memory of node v
t_v^-	Time when s_v is updated
$\Phi(\cdot)$	Time encoder

Table 2: Notation used in this paper

dynamic node embedding at the current state. To address this issue, many works [10, 15, 18, 23] use node memory to summarize the history of the nodes in the past. Later when this node is referenced as a temporal neighbor, its node memory serves as complimentary information and is combined with the node features as the input node features.

To maintain the node memory of each node, when an event indicates the appearing of a new edge, a sequence model (RNN or GRU) is used to update the corresponding node memory. If there is a new edge connecting from node u to node v at the current timestamp t, we generate two mails m_e^{uv} and m_e^{vu}

$$m_e^{uv} = \left(\boldsymbol{s}_u || \boldsymbol{s}_v || \Phi(t - t_v^-) || \boldsymbol{e}_{uv}\right) \tag{1}$$

$$m_e^{vu} = \left(\boldsymbol{s}_v || \boldsymbol{s}_u || \Phi(t - t_v^-) || \boldsymbol{e}_{uv}\right). \tag{2}$$

The time encoder Φ [1] encodes the time interval $\Delta t = t - t_v^-$ into vector

$$\Phi(\Delta t) = \cos(\boldsymbol{\omega}\Delta t + \boldsymbol{\phi}), \tag{3}$$

where $\boldsymbol{\omega}$ and $\boldsymbol{\phi}$ are two learnable vectors. The node memory is then updated by

$$s_v = \mathrm{UPDT}\left(s_v, \mathrm{COMB}\left(m_e^{ij} | v \in \mathcal{N}(i) \cup \mathcal{N}(j)\right)\right), \tag{4}$$

where UPDT is the RNN or GRU memory updater, and COMB is the combiner of all related neighbor input mails. The mails are delivered to the neighbors of the source and destination nodes.

When performing GNN message passing, the node memory is combined with the original node features $\boldsymbol{v}_v$ to serve as the new node features.

$$\boldsymbol{v}_v' = \boldsymbol{s}_v + \mathrm{MLP}(\boldsymbol{v}_v). \tag{5}$$

2.2 Attention Aggregator

TGNNs adopt the attention mechanism from Transformer [19] to gather and aggregate information from temporal neighbors. The attention aggregation of node u is computed by the queries, keys, and values from its hop-1 temporal neighbors $v \in \mathcal{N}(v)$.

2.3 Temporal Sampler

To ensure each node can access the relevant information from its supporting nodes or the mails are delivered to neighbor nodes, TGNNs need to consider the edge timestamps when sampling. There are two major sampling strategies, uniform sampling where neighbors in the past are sampled uniformly as supporting nodes and most-recent sampling where only the most recent neighbors are sampled. Note that in a dynamic graph, two nodes can have multiple edges at different timestamps. These nodes can also be sampled multiple times as supporting nodes with different timestamps.

3 TGL

In this section, we present TGL – a general framework for efficient TGNNs training on large-scale dynamic graphs.

Figure 2 shows the overview of the training of TGL on a single GPU. We split the modules with learnable and non-learnable parameters to store on GPU and CPU respectively. For datasets where the GPU memory is sufficient to hold all information, the non-learnable modules can also be stored and computed on GPU to speedup the training. To be compatible with different TGNN variants, we design five general components: the temporal sampler, the mailbox, the

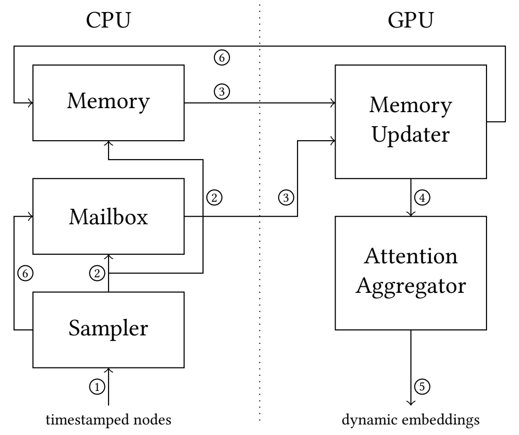

Figure 2: Overview (forward path) of the proposed framework. ① Sample neighbors for the root nodes with timestamps in the current mini-batch. ② Lookup the memory and the mailbox for the supporting nodes. ③ Transfer the inputs to GPU and update the memory. ④ Perform message passing using the updated memory as input. ⑤ Compute loss with the generated temporal embeddings. ⑥ Update the memory and the mailbox for next mini-batch.

node memory, the memory updater, and the attention aggregator. For snapshot-based TGNNs, the temporal sampler would sample in each snapshot individually. Note that in TGL, we do not treat graph snapshots as static windows. Instead, the graph snapshots are dynamically created according to the timestamp of the target nodes. This allows the snapshot-based TGNNs to generate dynamic node embeddings at any timestamps, instead of a constant embedding in a static snapshot.

TGNNs are usually self-supervised by the temporal edges, because it is hard to get dynamic graphs with enough dynamic node labels to supervise the training. Training with temporal edges causes the "information leak" problem where the edges to predict are already given to the models as input. The information leak problem in attention aggregator can be simply avoided by not sampling along the edges to predict. In node memory, the information leak problem is eliminated by caching the input from previous mini-batches [15], which enables the node memory to receive gradients. In TGL, we adopt the mailbox module [23] to store a fixed number of most recent mails for updating the node memory. When a new interaction appears, we first update the node memory of the involved nodes with the cached messages in the mailbox. The messages in the mailbox are updated after the dynamic node embeddings are computed. Note that to keep the node memory consistent, the same updating scheme is used at inference when updating the node memory is not necessary.

TGL users can easily configure TGL to train different TGNN variants by yaml configuration files. TGL supports a wide range of TGNN variants including vanilla attention-based TGNN TGAT [1], snapshot-based TGNNs like Evolve-GCN [13] and DySAT [17], memory-based TGNNs [15, 18], and pure memory-based TGNNs [10, 23].

Algorithm 1: Parallel Temporal Sampler

```
Data: sorted T-CSR $G$
Input: root nodes $\boldsymbol{n}$ with timestamp $\boldsymbol{t_n}$, number of layer $L$,
       number of neighbors in each layer $k_l$, number of
       snapshots $S$, snapshot length $t_s$
Output: DGL MFGs
1  advance the pointer of $\boldsymbol{n}$ to $\boldsymbol{t_n}$ in $pt(S+1)$ in parallel;
2  for $l$ in $0..L$ do
3      for $s$ in $0..S$ do
4          if $l \geq 0$ then
5              set $\boldsymbol{n}$ and $\boldsymbol{t_n}$ to sampled neighbors in $l-1$;
6          end
7          if $l == 0$ then
8              advance the pointer of $\boldsymbol{n}$ to $\boldsymbol{t_n} - s * t_s$ in
                 $pt(S-s-1)$ in parallel;
9          else
10             binary search in the snapshots $S_s$ for each node
                 $n \in \boldsymbol{n}$ in parallel;
11         end
12         foreach $n \in \boldsymbol{n}$ in parallel do
13             sample $k_l$ neighbors within the snapshot $S_s$;
14         end
15         generate DGL MFGs;
16     end
17 end
```

3.1 Parallel Temporal Sampler

Sampling neighbors on dynamic graphs is complex as the timestamps of the neighbors need to be considered. In the offline training process, TGL stores the entire dynamic graph statically where the timestamps are attached to the nodes and edges. For snapshot-based TGNNs, the temporal samplers need to identify the snapshots before sampling. Other TGNNs that either samples uniformly from all past neighbors or sample most recent neighbors can be treated as single snapshot TGNNs with infinite snapshot length. Their temporal samplers also needs to identify the candidate edges and their sampling probabilities. Hence, it is important to design a data structure that can rapidly identifies the dynamic candidate set of temporal neighbors to sample from. Combined with the fact that the mini-batches in TGNNs training follow chronological order (have non-decreasing timestamps), we propose the Temporal-CSR (T-CSR) data structure.

The T-CSR Data Structure Besides the indptr and indices array of the CSR data structure, for each node, T-CSR sorts the outgoing edges according to their timestamps as shown in Figure 3. After sorting all the edges in a dynamic graph, we assign edge ids according to their position (indexes) in the sorted indices and times arrays. In addition, for a TGNN model with n snapshots, we maintain $n+1$ pointers for each node that point at the first and last edges in these snapshots. Formally, the T-CSR data structure is defined by an indptr array of size $|V|+1$, an indices array and a time array of size $|E|$, and $n+1$ pointers array of size $|V|$, which leads to a total space complexity of $O(2|E|+(n+2)|V|)$. For dynamic graphs with inserting, updating, and deletion of edges and nodes, the T-CSR data

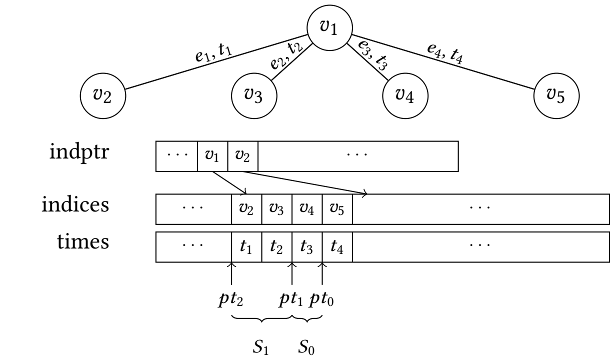

Figure 3: T-CSR representation of the node v_1 with four temporal edges e_1 to e_4 with timestamps t_1 to t_4 connected to neighbors v_1 to v_4. The indices and times arrays are sorted by the edge timestamps and indexed by the edge ids e_1 to e_4. S_0 and S_1 denote two snapshots of the temporal graph, designated by the pointers pt_0 to pt_2.

structure can treat them as standalone graph events and allocate their own entries in the indices and times array.

Sampling With the help of the T-CSR data structure, we can quickly choose an edge between two pointers uniformly, or pick edges closest to the end pointer for the most recent neighbors. These pointers are stored in arrays and takes an additional $O(n|V|)$ storage and $O(|E|)$ computation complexity to maintain in one epoch, but allows the sampler to identify candidate edges in $O(1)$. By contrast, performing binary search would lead to $O(|E|\log|E|)$ computation complexity to identify candidate edges in one epoch. Note that some TGNNs like TGAT [1] use the timestamp of the neighbors to sample multi-hop temporal neighbors, instead of using the timestamp of the root nodes. For these TGNNs, the proposed pointer only works for the hop-1 neighbors. Since the edges are sorted in T-CSR, we can still to use binary search to quickly find out the candidate edges before sampling.

Parallel Sampling To leverage the multi-core CPU resources in the host machine, we exploit data parallelism to sample on the root nodes in a mini-batch as shown in Algorithm 1. In each mini-batch, the target nodes are evenly distributed to each thread to update the pointers and sample the neighbors. Note that when updating the pointers in parallel, it is possible that multiple threads share the same target nodes with different timestamps, which causes race conditions. We add fine-grained locks to each node to avoid the pointers being advanced multiple times under such conditions. When same target nodes at different timestamp appears multiple time in one mini-batch, it is also possible that the target nodes with small timestamps sample temporal neighbors from the future. We prevent information leak in such situation by strictly enforcing that the sample temporal neighbors have smaller timestamps than the root nodes. After each thread finishes sampling in each mini-batch, we generate a DGL Message Flow Graph (MFG) for each layer [22] which contains all the input data needed in the forward and backward propagation and pass it to the trainer.

Algorithm 2: Random Chunk Scheduling

```
Data: training edges $E$, sorted T-CSR $G$, TGNN model $M$
Input: batch size $bs$, chunk size $cs$, training epochs $E$
for $e$ in $0..E$ do
    $e_s \leftarrow \mathrm{rand}(0, bs/cs) * bs$;
    $e_e \leftarrow e_s + bs$;
    while $e_e \leq |E|$ do
        sample MFGs from $E(e_s..e_e)$;
        train for one iteration with the current MFG;
        $e_s \leftarrow e_s + bs$;
        $e_e \leftarrow e_e + bs$;
    end
end
```

3.2 Parallel Training

In order to scale static GNN training to large graphs, recent works [21, 24] increase the batch size to take advantage of the massive data parallelism provided by multi-GPU servers or GPU clusters. However, training TGNN with a large batch size suffers from the intrinsic temporal dependency in the node memory. Defining the dependent edges as pairs of training edges who share common supporting nodes in the source or destination nodes, we can divide the edge dependencies into two types:

- **Intra-batch** dependencies refer to the dependent edges in the same mini-batch. In TGNN training, the intra-batch dependencies are discarded in order to process the edges in a mini-batch in parallel.
- **Inter-batch** dependencies refer to the dependent edges in different mini-batches. TGNNs take these inter-batch relations into account by updating the node memory and the mailbox after each mini-batch.

Since the total number of intra- and inter-batch dependencies is constant on one dynamic graph, training with a larger batch size discards more intra-batch dependencies and learns less inter-batch dependencies which leads to lower accuracy. To mitigate this issue, we propose a random chunk scheduling technique that divides the training edges into chunks and randomly picks one chunk as the starting point in each training epoch, which allows close chunks to be arranged in different mini-batches in different training epochs, hence learning more inter-batch dependencies. The random chunk training algorithm is shown in Algorithm 11.

To train TGL on multiple GPUs, we adopt the setup of multiple GPUs on a single node and store the node memory and the mailbox in the main memory. On n GPUs, we launch n training processes and one sampling process with inter-process communication protocols. For simplicity, the updates of the model weights, the node memory, and the mailbox are synchronized.

4 EXPERIMENTS

We perform detailed experiments to evaluate the performance of TGL. We implement TGL using PyTorch 1.8.1 [14] and DGL 0.6.1 [22]. The parallel temporal sampler is impletmented using C++ and integrated to the Python training script using PyBind11 [7]. The open-sourced code of TGL could be found at https://github.com/tedzhouhk/TGL.

Table 3: Dataset Statistic. The max(t) column shows the maximum edge timestamp (minimum edge timestamp is 0 in all datasets). $|d_v|$ and $|d_e|$ show the dimensions of node features and edge features, respectively. The * denotes randomized features.

	$\|V\|$	$\|E\|$	max(t)	Labels	Classes	$\|d_v\|$	$\|d_e\|$
Wikipedia	9K	157K	2.7e6	217	2	-	172
Reddit	11K	672K	2.7e6	366	2	-	172
MOOC	7K	412K	2.6e6	-	-	-	128*
LastFM	2K	1.3M	1.3e8	-	-	-	128*
GDELT	17K	191M	1.8e5	42M	81	413	186
MAG	122M	1.3B	120	1.4M	152	768	-

We select five representative TGNN variants as the baseline methods and evaluate their performance in TGL.

- JODIE [10] is a pure memory-based TGNN method that uses RNN to update the node memory by the node messages. We use the open-sourced code implemented as a baseline in TGN [15] as the baseline code.
- DySAT [17] is a snapshot-based TGNN that uses RNN to combine the node embeddings from different snapshots.
- TGAT [1] is a attention-based TGNN that gathers temporal information by the attention aggregator.
- TGN [15] is a memory-based TGNN that applies the attention aggregator on the node memory updated by GRU with the node messages.
- APAN [23] is a pure memory-based TGNN method that uses attention aggregator to update the node memory by the node messages delivered to the multi-hop neighbors.

For fair comparison, we set the receptive field to be 2-hop and fix the number of neighbors to sampler per hop at 10. The size of the mailbox is set to be 10 mails in APAN while 1 mail in other methods. For the COMB function in Equation 4, we use the most recent mail in all methods, as we do not see noticeable difference if switched to the mean of mails. We set the dimension of the output dynamic node embeddings to be 100. We apply the attention aggregator with 2 attention heads for the message passing step in all baseline methods. For DySAT, we use 3 snapshots with the duration of each snapshot to be 10000 seconds on the four small-scale datasets, 6 hours on GDELT, and 5 years on MAG. As mentioned in Section 3, TGL uses dynamic snapshot windows to ensure that the time resolution of the generated dynamic node embeddings is the same as the other TGNNs. For fairness, we add layer normalization to JOIDE and TGAT, which allows all methods to have layer normalization and in-between each layer. For all methods, we sweep the learning rate from {0.01,0.001,0.0001} and dropout from {0.1,0.2,0.3,0.4,0.5}. The TGNN models are trained with the link prediction task and directly used in the dynamic node classification task without fine-tuning [1, 15]. On all datasets, we follow the extrapolation setting that predict the links or node properties in the future given the dynamic graphs in the past. We provide comprehensive and nondiscriminatory benchmark results for various TGNNs by evaluating them in the TGL framework.

4.1 Datasets

Table 3 shows the statistic of the six datasets we use to evaluate the performance of TGL. As the Wikipedia [15], Reddit [15], MOOC [10], and LastFM [10] datasets are small-scale and bipartite dynamic graphs, in order to evaluate the performance on general and large-scale graphs, we introduce two large-scale datasets – GDELT and MAG. These two datasets contains 0.2 and 1.3 billion edges in multiple years and focus on testing the capability of TGNNs in two different dimensions.

4.1.1 GDELT. The GDELT dataset is a Temporal Knowledge Graph (TKG) originated from the Event Database in GDELT 2.0 [11] which records events happening in the world from news and articles in over 100 languages every 15 minutes. Previous event prediction work [8] pre-processed a small dataset from the same source with events happened in January 2018. Their version is a featureless graph where the features of actors and events are ignored. In this work, we propose a larger featured version with events happened from the beginning of 2016 to the end of 2020. Our GDELT dataset is a homogeneous dynamic graph where the nodes represent actors and temporal edges represent point-time events. Each node has a 413-dimensional multi-hot vector representing the CAMEO codes attached to the corresponding actor to server as node features. Each temporal edge has a timestamp and a 186-dimensional multi-hot vector representing the CAMEO codes attached to the corresponding event to server as temporal edge features. The link prediction task on the GDELT dataset predicts whether there will be an event happening between two actors at a given timestamp. For the node classification task, we use the countries where the actors were located when the events happened as the dynamic node labels. We remove the dynamic node labels for the nodes that have the same labels at their most recent timestamps to make this task more challenging. We use the events before 2019, in 2019, and in 2020 as training, validation, and test set, respectively. The GDELT datasets has dense temporal interactions between the nodes and requires TGNNs to be able to capture mutable node information for a long time duration.

4.1.2 MAG. The MAG dataset is a homogeneous sub-graph of the heterogeneous MAG240M graph in OGB-LSC [5]. We extract the paper-paper citation network where each node in MAG represents one academic paper. A directional temporal edge from node u to node v represents a citation of the paper v in the paper u and has a timestamp representing the year when the paper u is published. The node features are 768-dimensional vectors generated by embedding the abstract of the paper using RoBERTa [12]. The link prediction task on the MAG dataset predicts what papers will a new paper cite. For the node classification dataset, we use the arXiv subject areas as node labels. We use the papers published before 2018, in 2018, and in 2019 as training, validation, and test set. The MAG dataset test the capability of TGNN models to learn dynamic node embeddings on large graph with stable nodes and edges.

4.2 Parallel Temporal Sampler

The performance of our parallel temporal sampler is evaluated on the g4dn.8xlarge instance on AWS EC2 with 32 virtual CPUs and 64GB of main memory. We select three representative sampling algorithms

- DySAT 2-layer sampling represents the temporal graph sampling for snapshot-based methods. The supporting nodes are chosen uniformly from the temporal neighbors in each dynamic snapshots.
- TGAT 2-layer sampling represents the uniformly temporal graph sampling which selects supporting nodes uniformly from all past temporal neighbors.
- TGN 1-layer sampling represents the most recent temporal graph sampling which selects most recent temporal neighbors as supporting nodes. Most recent sampling algorithms are usually used in memory-based methods and hence requires one less supporting layers.

Table 4: Execution time and improvement with respect to baseline samplers on the Wikipedia dataset for one epoch.

	DySAT			TGAT			TGN		
#Threads	1	8	32	1	8	32	1	8	32
Time (s)	1.161	0.446	0.371	1.557	0.569	0.370	0.094	0.46	0.039
Improv.	-	-	-	23×	48×	57×	69×	188×	289×

Table 4 shows the improvement (speedup) of the temporal parallel sampler in TGL compared with the samplers in the open-sourced baselines using different number of threads. The baseline samplers sample the neighbors by performing single-thread vectorized binary search on sorted neighbors lists. We show the sampling time for one epoch with batch size of 600 positive and 600 negative edges. With our efficient T-CSR data structure, TGL spends less than 0.5 seconds in sampling on one epoch of the Wikipedia dataset for all three sampling algorithms. Using 32 threads, TGL achieves 57× and 289× speedup compared with the sampler in TGAT and TGN. The speedup is a result by combined factors of 1) the T-CSR data structure, 2) data parallelism, and 3) efficiency of C++ over Python.

Figure 4 shows the runtime and the runtime breakdown of our temporal parallel sampler using a different number of threads. TGL achieves 3.13×, 4.20×, and 2.42× speedup using 32 threads for the DySAT, TGAT, and TGN sampling algorithms. The reasons for the sub-linear speedup are 1) node-wise locks in updating the pointers

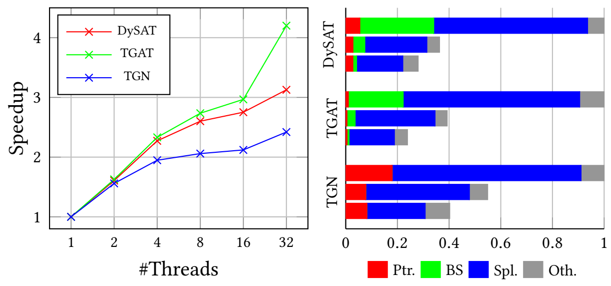

Figure 4: (a) Scalability of the temporal sampler on the Wikipedia dataset. (b) Runtime breakdown (normalized by single thread runtime) of the temporal sampler on the Wikipedia dataset with 1 (top), 8 (mid), and 32 (bottom) threads. Ptr., BS, Spl., and Oth. denote the time to update pointers (line 4 and 11), to perform binary search (line 13), to sample neighbors (line 16), and to generate DGL MFGs (line 18) in Algorithm 1, respectively.

Table 5: Link Prediction results on the Wikipedia, Reddit, MOOC, and LastFM datasets. The Time columns refer to the training time per epoch. (First second)

	Wikipedia					Reddit					MOOC		LastFM	
	Baseline		TGL			Baseline		TGL			TGL		TGL	
	AP	Time (s)	AP	Time (s)	Speedup	AP	Time (s)	AP	Time (s)	Speedup	AP	Time (s)	AP	Time (s)
JODIE	94.35	16.6	98.90	**1.0**	16.94×	96.56	89.0	99.45	**4.2**	21.24×	98.95	**2.8**	**78.78**	**8.7**
DySAT	-	-	96.37	6.4	-	-	-	98.57	21.5	-	98.76	19.5	76.39	48.4
TGAT	95.09	110.1	97.26	6.6	16.73×	97.82	576.2	99.48	39.9	14.45×	98.50	24.5	54.82	91.4
TGN	98.34	17.7	**99.62**	2.1	8.51×	98.47	91.9	**99.78**	10.5	8.33×	**99.59**	5.7	73.76	18.7
APAN	98.12	8.8	98.14	2.0	4.38×	99.22	121.7	99.24	8.8	13.85×	98.58	5.6	62.73	18.2

2) memory performance bottleneck when fetching the selected edge information 3) linear workload with respect to the number of threads when generating DGL MFGs.

4.3 Single-GPU Training

We evaluate the performance of TGL with a single GPU on the four small datasets. We use the same g4dn.8xlarge AWS EC2 instance with one Nvidia T4 GPU. We find that on all datasets, the batch size of 600 positive edges with 600 negative edges are a good balance point between the convergence rate and training speed for memory-based TGNNs. Hence, for a fair comparison, we use batch size of 600 for all five selected TGNN variants in TGL and their open-sourced baselines. For the MOOC and LastFM datasets, we randomly generate 128-dimensional edge features since the original datasets do not contain node or edge features. Due to the lack of authentic features and performance evaluation of the baseline code, we do not compare the performance of TGL and the baseline code on these two datasets. Since the 16GB GPU memory is enough to hold the node features, the edge features, the node memory, and the mailbox, we store these data on GPU instead of CPU to avoid the data transfer overhead. We use 32 threads in the temporal parallel sampler.

Table 5 shows the accuracy and per epoch training time of the five baselines and TGL in the link prediction task. We report the accuracy in Average Precision (AP) on both the positive and negative test edges. For all methods, TGL achieves similar or higher AP than the baselines with significantly faster runtime (see Figure 1). The accuracy improvement on TGAT and JODIE is because we use layer normalization in-between each layer. The accuracy of TGAT and TGN also benefits from better hyper-parameters and convergence. TGN achieves the highest AP in the link prediction task on all datasets except the LastFM dataset, followed by JODIE, DySAT and TGAT. The pure memory-based TGNN and JODIE achieves top-tier accuracy with the fastest training time. With efficiently implemented components and optimized data path, TGL achieves an average of 13× speedup in the per-epoch training time.

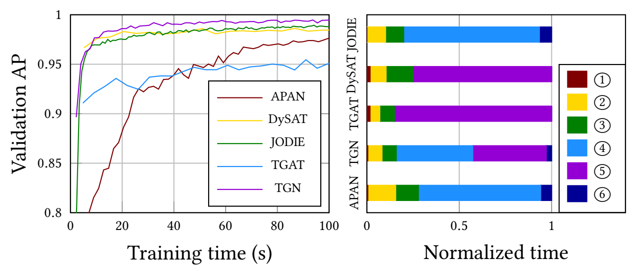

Figure 5: Validation AP with training time (left) and normalized runtime breakdown (right) on the Wikipedia dataset. The circled numbers refer to the six steps in Figure 2.

Figure 5 shows the convergence curve and runtime breakdown on the Wikipedia dataset. JODIE, DySAT, and TGN have faster convergence speed than APAN and TGAT. The runtime breakdown is measured by storing the node memory and mailbox in the main memory and forcing to use synchronized execution between the CPU and GPU, which leads to around 15% more execution time than asynchornized execution. With our temporal parallel sampler, the sampling overhead in TGL is negligible. For computation intensive two-layer TGNNs like DySAT and TGAT, the runtime is dominated by the computation on GPU. For memory-based models, the time spent in updating the node memory and the mailbox takes up to 30% of the total training time.

Table 6 shows the results of directly using the learned TGNN models on the dynamic node classification task. On the Wikipedia and the Reddit datasets, the node classification tasks are to identify banned users. Since the number of positive labels are small compared with the number of negative labels, we train the MLP classifiers with an equal number of randomly sampled negative labels, similar to training link prediction models. We also show the accuracy as AP on both the positive nodes and sampled the negative nodes. TGN and JODIE achieve the highest AP on the Wikipedia and the Reddit datasets, where JODIE achieves more than 7% AP than other methods on the Reddit dataset. We assume this is due to the noisy neighbors in the Reddit dataset, which prevent high-expressive model from learning general patterns on the graph structure.

4.4 Random Chunk Scheduling

To evaluate the effectiveness of the random chunk scheduling technique, we train the TGN model which has the best overall performance on the two small-scale datasets, as training with a small

Table 6: Dynamic node classification result (First second)).

	Wikipedia	Reddit	GDLET	MAG
	AP		F1-Micro	
JODIE	81.73	**70.91**	11.25	43.94
DySAT	86.30	61.70	10.05	50.42
TGAT	85.18	60.61	10.04	**51.72**
TGN	**88.33**	63.78	**11.89**	49.20
APAN	82.54	62.00	10.03	-

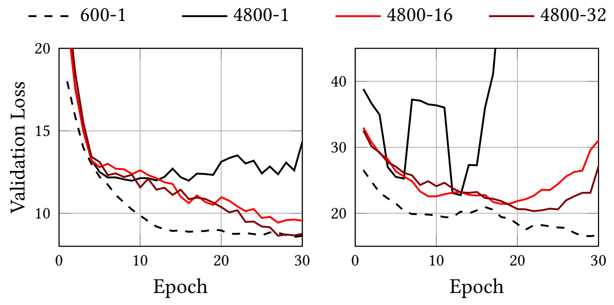

Figure 6: Validation loss (moving average of 5 epochs) with different chunk size when using random chunk scheduling algorithm with large batch size on Wikipedia (left) and Reddit (right) dataset. We denote batch size x and number of chunks per batch y as $x-y$ in the legends.

batch size and plot various convergence curves on the large-scale datasets is too slow. To make a fair comparison, we train the baseline models with the best group of hyperparameters (0.001 learning rate, 600 batch size). We then increase the batch size and also linearly increase the learning rate, as a larger batch size leads to a better approximation of the total loss [3]. Specifically, we train the same model with 8× the batch size and learning rate (0.008 learning rate, 4800 batch size) with the chunk sizes of 4800, 300, and 150 (number of chunks per batch size 1, 16, and 32). Since the node memory used in the validation process is inherited from the training process, when we compute the validation loss, we first reset the node memory and use a constant batch size of 600 to run one whole epoch on the training and validation set. Figure 6 shows the validation loss under different batch sizes and chunk sizes on the Reddit and Wikipedia datasets. The models trained with the batch size of 4800 and no random chunk scheduling cannot learn on both datasets after 5 to 10 epochs, due to the lost dependencies in the training mini-batches. On the Wikipedia dataset, the batch size of 4800 with 16 chunks per batch performs better than no chunks while the same batch size with 32 chunks per batch achieves similar convergence after 30 epochs. On the Reddit dataset, our random chunk scheduling technique also mitigates the overfitting issue and achieves close to baseline convergences within 30 epochs.

4.5 Multi-GPU Training

We evaluate the performance of TGL on the two large-scale datasets with multiple GPUs. We use the p3dn.24xlarge instance on EC2 with 96 virtual CPUs, 768GB of main memory, and 8 Nvidia V100

Table 7: Link Prediction results of TGL on GDELT and MAG. The Time columns refer to the training time per epoch (First second)).

	GDELT		MAG	
	AP	Time (s)	AP	Time (s)
JODIE	97.98	**599.2**	99.41	**4128.3**
DySAT	98.72	10651.4	98.27	19748.6
TGAT	96.49	8499.2	99.02	32104.5
TGN	**99.39**	915.9	**99.49**	8912.5
APAN	95.28	1358.5	-	-

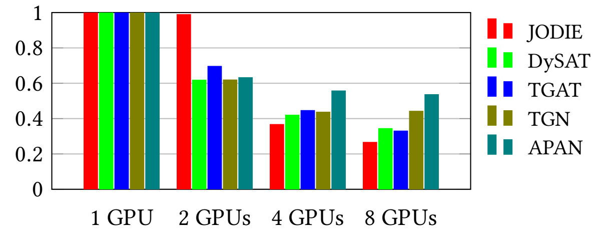

Figure 7: Normalized Training time per epoch with different number of GPUs on the GDELT dataset.

GPUs. We use 64 threads in the temporal parallel sampler and assign 8 threads for each trainer process. We use a local batch size of 4000 positive and 4000 negative edges on each GPU. The global copy of the node memory and the mailbox are stored in the shared memory. The trainer process then overlaps the MFG copy to GPU with the computation on GPU by creating additional copying threads on different CUDA streams. The gradients in each iteration are synchronized among the trainer processes through the NCCL backend.

Table 7 shows the AP and running time in the link prediction task. Similar to the single GPU results, TGN achieves the highest AP and JODIE has the fastest training time. On the GDELT datasets, the memory-based models can train one epoch within 30 minutes, while the non-memory based models need more than 3 hours. On the MAG dataset, APAN throws out of memory error as it requires the mailbox to store 10 most recent mails for each node in the graph. Figure 7 shows the scalability of TGL on multiple GPUs. TGL achieves 2.74×, 2.28×, 2.25×, 2.30× and 1.80× speedup by using 4 GPUs for JODIE, DySAT, TGAT, TGN, and APAN, respectively. For 8 GPUs, the bandwidth between CPU and main memory to slice the node and edge features and update the node memory and the mailbox and the number of PCI-E channels to copy the MFGs to the GPUs are saturated.

Table 6 shows the F1-Micro of the trained models in the multiple-class single-label dynamic node classification task. On the GDELT dataset, all models perform bad where JODIE and TGN has slightly better performance than others. On the MAG dataset, TGAT and DySAT with two complete graph attention layers achieves the highest and second highest accuracy while JODIE with no graph attention layer achieves the lowest accuracy.

5 CONCLUSION

In this work, we proposed TGL – the first unified framework for large-scale TGNN training. TGL allows users to efficiently train different TGNN variants on a single GPU and multiple GPUs by writing simple configuration files. We designed the T-CSR data structure to store the dynamic graphs and developed a temporal parallel sampler which greatly reduces the sampling overhead. We proposed the random chunk scheduling technique to mitigate the loss of dependencies when training with a large batch size. We processed two large-scale datasets to test the capability of TGNNs in two different dimensions. We evaluated the performance of five different TGNN variants on four small-scale datasets and two large-scale datasets with billions of edges. TGL achieves similar or better accuracy on all datasets with significantly faster training time compared with the open-sourced baselines.

REFERENCES

[1] da Xu, chuanwei ruan, evren korpeoglu, sushant kumar, and kannan achan. 2020. Inductive representation learning on temporal graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJeW1yHYwH

[2] Matthias Fey and Jan E. Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[3] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv:1706.02677 [cs.CV]

[4] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *arXiv preprint arXiv:2103.09430* (2021).

[5] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. OGB-LSC: A Large-Scale Challenge for Machine Learning on Graphs. *arXiv preprint arXiv:2103.09430* (2021).

[6] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open Graph Benchmark: Datasets for Machine Learning on Graphs. *arXiv preprint arXiv:2005.00687* (2020).

[7] Wenzel Jakob, Jason Rhinelander, and Dean Moldovan. 2016. pybind11 — Seamless operability between C++11 and Python. https://github.com/pybind/pybind11.

[8] Woojeong Jin, Meng Qu, Xisen Jin, and Xiang Ren. 2020. Recurrent Event Network: Autoregressive Structure Inferenceover Temporal Knowledge Graphs. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Online, 6669–6683. https://doi.org/10.18653/v1/2020.emnlp-main.541

[9] 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.

[10] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of the 25th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM.

[11] Kalev Leetaru and Philip A. Schrodt. 2013. GDELT: Global data on events, location, and tone. *ISA Annual Convention* (2013). http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.686.6605

[12] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. *arXiv preprint arXiv:1907.11692* (2019).

[13] 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 *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*.

[14] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

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

[16] Benedek Rozemberczki, Paul Scherer, Yixuan He, George Panagopoulos, Alexander Riedel, Maria Astefanoaei, Oliver Kiss, Ferenc Beres, , Guzman Lopez, Nicolas Collignon, and Rik Sarkar. 2021. PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*.

[17] 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.

[18] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HyePrhR5KX

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

[20] Andrew Z Wang, Rex Ying, Pan Li, Nikhil Rao, Karthik Subbian, and Jure Leskovec. 2021. Bipartite Dynamic Representations for Abuse Detection. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 3638–3648.

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

[22] 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).

[23] 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 *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD/PODS '21)*. Association for Computing Machinery, New York, NY, USA, 2628–2638. https://doi.org/10.1145/3448016.3457564

[24] 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. *2020 IEEE/ACM 10th Workshop on Irregular Applications: Architectures and Algorithms (IA3)* (Nov 2020). https://doi.org/10.1109/ia351965.2020.00011

6 RESPONSE TO REVIEWER COMMENTS

We thank the reviewers for the time and effort they dedicated to provide valuable feedbacks on our paper. We arrange our response as follows. We first highlight the major improvements and new experiment results. Then, we respond to each comment in detail.

6.1 Highlights in Revised Paper

The highlights in the revised paper are

- **Background and Motivation** In the updated Section 1, we have added the definition of Continuous Time Dynamic Graphs (CTDGs) and Discrete Time Dynamic Graphs (DTDGs) and explained why we aim at supporting the more general CTDGs in TGL. We have also explained why it is hard to implement TGNNs on existing graph frameworks and why we need a new framework for TGNNs.
- **Parallel Temporal Sampler** In the updated Section 3.1, we have added a formal definition of the proposed T-CSR data structure as well as how the T-CSR data structure is maintained during the sampling process. We have also restructured the Section into three parts with the T-CSR data structure, sampling, and parallel sampling. We have also clarified how our T-CSR data structure can be used to insert and delete events of nodes and edges.
- **Random Chunk Scheduling Technique** In the updated Section 3.2, we have replaced the Figure that was depicting the proposed random chunk scheduling technique with Algorithm 11 to enhance clarity on the developed technique. We have also added a more detailed description on the proposed technique.
- **Experiments** In the updated Section 4.2, we have added more details on the implementation of the baseline samplers and on the speedup provided by our parallel temporal sampler. For completion, in the updated Section 4.3 we also have included the results of two public small-scale datasets – the LastFM and MOOC datasets. We have also depicted the runtime of each individual step in the updated Figure 5.

6.2 Response to Meta-Reviewer

1. **Improve the technical exposition on the system design decisions and make the description of the techniques more rigorous and self-contained.**
   In the revised paper, we have added further details on the system design of TGL, especially the parallel temporal sampler and the random chunk scheduling technique. For more details, please refer to our responses to **R1.W1**, **R1.W2/D2**, **R2.D1**, **R3.D1**, and **R3.D3/D4/D5/D7**.
2. **Provide the missing details and further explanations of the empirical results. Provide more qualitative and quantitative justification of temporal GNN vs regular prior GNN methods/tools.**
   In the updated Section 4.2, we have added more details on the baseline samplers and explained the speedup achieved by TGL. Please refer to our response to **R2.D1/D2** for more details. In the updated Figure 5, we have included the detailed execution time of each component in TGL. Please refer to our response to **R2.D3** for more details. In the updated Section 4.3, we have justified the accuracy improvements of TGL compared with each baselines. Please refer to our response to **R3.D1** for more details.
3. **Add another standard public dataset from the literature for the experiments, e.g., LastFM song listens.**
   In the updated Section 4.3, we have added the results on two public datasets. Please refer to our response to **R3.D2** for more details.

6.3 Response to Reviewer #1

1. W1**: The challenging for temporal GNN is not well justified, given several public GNN framework (DGL and PyG) are available.**
   In the updated Section 1, we have justified the motivation to develop a unified framework for TGNN. These general GNN frameworks (DGL and PyG) are mainly designed for static graphs where their data structures are not designed to efficiently store and access the dynamic graphs. In addition, they do not provide important modules in TGNNs like temporal neighbor samplers, node memory, and message passing schemes on multiple snapshots. The users need to write at least hundreds of lines of code to compose a TGNN model using DGL or PyG. By contrast, TGL users can simply to write configuration files to compose various TGNN models. The lack of a TGNN framework could also be reflected by the fact that recent TGNN works like TGAT and TGN still purely rely on basic machine learning frameworks Tensorflow and PyTorch in their experiments.
2. D1**: As we know CSR is very difficult for dynamic updates, since all edges are stored in the consecutive array. I do not understand how the proposed temporal CSR handle the edge or node updates, including insertion and deletions. It seems that the temporal graph in in this paper refer to a graph with temporal information on edges; but they are still static.**
   In the updated Section 1 and Section 3.1, we have added the clarification that the proposed T-CSR data structure is designed to support efficient offline TGNN training. In the offline training setup, the entire training (dynamic) graph is known in advance. Hence, we can allocate exact space for all the nodes and edges in the training graphs before training. We have also added a note on handling inserting, updating, and deleting events on the proposed T-CSR data structure. For the dynamic graphs that contain updating and deleting events, we follow TGN [15] to treat each updating or deleting event independently and generate a standalone mail for each such event. However, we have also added the clarification that the T-CSR data structure could also supports dynamic graphs with the inserting and deleting events. We can treat the inserting and deleting events as standalone node or edges events with an additional validation array with size $|E|$ that marks whether an edge is valid or not. This validation array needs to be updated after finishing one mini-batch. During sampling, the invalid edges could be simply ignored.
   To address the online training or inference setting where the dynamic graph is streamed in, the T-CSR data structure could be modified to a List of List (LIL) or List of Array (LIA) data structure where the insert operation is cheaper than CSR. Our future work attempts to address this issue by implementing such

data structure and designing hardware accelerator for online sampling.

3. **W2/D2: Author should provide more system implementation details of TGL.**
In the revised paper, we have provide more details on the implementation details of TGL. Please refer to the **Parallel Temporal Sampler** and **Random Chunk Scheduling Technique** items in Section 6.1.

6.4 Response to Reviewer #2

1. **D1: Authors claim the speedup of TGL's sampler compared with the baselines' samplers in the paper. Though it is proven with the experimental results, more details about the baselines' samplers need to be provided, as well as the details for the reason of the time improvement.**
In the updated Section 4.2, we have added description of the baseline samplers in the open-sourced code and the reasons of the time improvement. The main factors for the time improvements are
(a) The proposed T-CSR data structure reduce the time complexity of identifying the candidate neighbors to sample from in one epoch from $O(|E|\log|E|)$ to $O(|E|)$. TGAT has lower speedup numbers than TGN because TGAT uses the timestamps of the neighbors to sample multi-hop neighbors. Please refer to Section 3.1 for more details. In addition, the T-CSR data structure uses contiguous memory to store the entire graph structure, which potentially leads to better cache performance.
(b) Our parallel temporal sampler exploits data parallelism by adding element-wise lock to avoid the race condition in the pointer arrays, which greatly reduces the time spent in selecting the neighbors (Figure 4.b).
(c) Our parallel temporal sampler is implemented in C++ and integrated to the main Python training program using PyBind11 where the baseline samplers are directly implemented in Python. C++ is more efficient than Python in such CPU heavy workloads.
2. **D2: In Table 4. (1) why the improvement ratio for DySAT is not included? (2) For TGN, the time for 8 threads is 0.46s but 0.039s with 32 threads. I think it is a typo here.**
(1) We pick DySAT as a representative sampling method for snapshot-based TGNNs. However, there is no baseline available for DySAT sampler since DySAT adopts full neighbor training strategy.
(2) Yes, it is a typo and we have corrected it.
3. **D3: Since there are five components in the framework TGL. I think the performance of different 'mailbox' and 'memory updater' can also be evaluated and included in the paper.**
In the updated Figure 5, we have further broke down the execution times to the six individual steps as shown in Figure 2. We have also added a clarification of the runtime breakdown in Section 4.3. In PyTorch, the GPU kernels are executed in asynchronous mode in order to overlap the operations on CPU and GPU. Measuring the time in each individual step requires the GPU kernels to be executed in synchronous mode, which leads to around 15% overhead in the execution time compared with the default asynchronous execution. The overhead primarily comes from the data copying from CPU to GPU (step ③) and the sampling and indexing (step ① and ②) which could either be overlapped with itself or with operations in the previous mini-batch. However, we believe the updated time breakdown provides more information on the runtime of each module in TGL.
4. **D4: There is a 404 error with the link provided by the authors. This paper proposed a framework for temporal neural networks and two large-scale temporal graphs. These can be important for the researchers. The authors are suggested to make the related code and data available as soon as possible to increase the impact of this work.**
There is an internal process in AWS before the code and data can be made public available. We are working on it and will release the code and data as soon as possible.

6.5 Response to Reviewer #3

1. **D1: The experiments did not give much insights into how could the accuracies of the TGNN methods be kept or even improved after TGL techniques are used.**
In the updated Section 4.3, we have explained the improvement in accuracy of TGL. From the algorithm perspective, TGL applies batch normalization in-between each layer, which increases the accuracy on JODIE and TGAT. Beside the added batch normalization on TGAT and JODIE, the other neural network architectures and training algorithms implemented in TGL are exactly the same as the baseline TGNNs. Due to the fast training speed of TGL, we are able to explore more hyper-parameters and achieve higher accuracy on TGN and TGAT due to better convergence. When training with large batch size, we have shown the effectiveness of our random chunk scheduling technique in Section 4.4.
2. **D2: There are many data sets used in the literature (such as Last FM song listens). Why are only two datasets considered? The proposed techniques are purely empirical. Testing TGL on more datasets is necessary to investigate the behaviors (such as robustness) of TGL. If page limitation is an issue, such results can be placed in a technical report.**
In the updated Section 4.3, we have included the results on two additional public datasets – the LastFM and MOOC datasets. The LastFM datasets is a user-song dataset from the popular music website last.fm while the MOOC datasets is a user-activity dataset from the MOOC social network. However, since all baseline code except JODIE [10] is not evaluated on these two datasets and does not support feature-less graphs, we believe there is no substantial value in comparing the performance of TGL with the baseline code on these two datasets. In the updated Section 1 and Section 4.1, we have justified our choice of the two small-scale datasets Wikipedia and Reddit as well as the motivation to look for larger datasets.
We choose the Wikipedia and Reddit datasets for the following reasons: (1) they both have an authentic 172-dimensional edge features while other datasets typically do not have edge features and have to use random features in the experiments, and (2) the state-of-the-art TGNN works (TGAT [1], TGN [15], APAN

[23]) are optimized and evaluated on these two datasets, which provides comprehensive and fair baselines.
On the other hand, these existing datasets only have few thousands of nodes and few millions of edges. Worse still, they are all bipartite graphs with only a few binary node labels. We believe these small-scale datasets do not reflect real-world production-scale dynamic graphs and are not capable of benchmarking various TGNN variants. Hence, we pre-process two large-scale datasets – the GDELT and MAG datasets with 0.2 billion and 1.3 billion edges. These large-scale datasets come with node and/or edge features and have more 42 million and 1.4 million node labels for the dynamic node classification task. We will also publish these two large-scale datasets to benefit future researches.

3. **D3: The proposed T-CSR data structure is not formally defined. It is only illustrated in Figure 3 and described casually.**
   In the updated Section 3.1, we have added more details of the proposed T-CSR data structure, including a formal definition and the space complexity analysis. We have also re-organize the Section and added three sub-titles – the T-CSR data structure, sampling, and parallel sampling to provide a clearer description of the proposed parallel temporal sampler. We have also added a note on the handling of inserting, updating, and deleting events. Please refer to our response to **R2.D2** for more details.
4. **D4: In Algorithm 1, the technical jargon DGL MFG is not even elaborated.**
   In the updated Section 3.1, we have added description about DGL Message Flow Graph (MFG). The DGL MFG is a data structure used by DGL which contains information needed in the forward and backward propagation in a graph block of one mini-batch.
5. **D5/6: The random chunk scheduling technique is only briefly sketched. The technique is not formally defined. Figure 4 is not understandable. Its caption contains much technical jargon.**
   To better explain the proposed random chunk scheduling technique, in the updated Section 3.2, we have removed Figure 4 and added a detailed algorithm Algorithm 11 describing how the random chunk scheduling technique is applied in the training process. We have also revised the Section with a new title that better summarizes the motivation of our random chunk scheduling technique. In the updated section, we first state that training with large batch sizes leads to less learned inter-batch dependencies in each epoch. Then, we point out that our random chunk scheduling technique allows the TGNN models to learn different inter-batch dependencies in different epochs. With enough number of training epochs and small chunk size, the TGNN models trained with random chunk scheduling technique and large batch size can learn the same amount of inter-batch dependencies as the TGNN models train with small batch size. Lastly, we sketched the multi-GPU training strategy, where more details are provided in Section 4.5.
6. **D7: Typos.**
   We have corrected these typos. For the snapshot durations, the numbers are for the snapshot duration of each dataset. In each dataset, the durations of the consecutive snapshots are constant. We have rewritten this sentence to eliminate the disambiguation.