TGOpt: Redundancy-Aware Optimizations for Temporal Graph Attention Networks

Yufeng Wang
University of Illinois at Urbana-Champaign, USA
yufengw2@illinois.edu

Charith Mendis
University of Illinois at Urbana-Champaign, USA
charithm@illinois.edu

Abstract

Temporal Graph Neural Networks are gaining popularity in modeling interactions on dynamic graphs. Among them, Temporal Graph Attention Networks (TGAT) have gained adoption in predictive tasks, such as link prediction, in a range of application domains. Most optimizations and frameworks for Graph Neural Networks (GNNs) focus on GNN models that operate on static graphs. While a few of these optimizations exploit redundant computations on static graphs, they are either not applicable to the self-attention mechanism used in TGATs or do not exploit optimization opportunities that are tied to temporal execution behavior.

In this paper, we explore redundancy-aware optimization opportunities that specifically arise from computations that involve temporal components in TGAT inference. We observe considerable redundancies in temporal node embedding computations, such as recomputing previously computed neighbor embeddings and time-encoding of repeated time delta values. To exploit these redundancy opportunities, we developed TGOpt which introduces optimization techniques based on deduplication, memoization, and precomputation to accelerate the inference performance of TGAT. Our experimental results show that TGOpt achieves a geomean speedup of 4.9× on CPU and 2.9× on GPU when performing inference on a wide variety of dynamic graphs, with up to 6.3× speedup for the Reddit Posts dataset on CPU.

CCS Concepts: • **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Software performance**.

Keywords: Temporal Graph Neural Networks, Redundancy-Aware Optimizations, Memoization, Dynamic Graphs

PPoPP '23, February 25-March 1, 2023, Montreal, QC, Canada

ACM ISBN 979-8-4007-0015-6/23/02…$15.00
https://doi.org/10.1145/3572848.3577490

1 Introduction

Graph Neural Networks (GNNs) [24] have gained rapid adoption across a wide range of application domains, including social network analysis [40], financial fraud detection [4, 30], and drug discovery [13, 27]. GNNs perform predictive modeling on graph-structured data, typically by learning node embedding representations using node features and their neighborhood information. Many of the popular GNN models [9, 16, 29] were developed to primarily operate on static graphs. Subsequently, there have been many efforts at optimizing GNNs for CPUs [15], GPUs [33, 34, 39], and hardware accelerators [1, 38, 43] aimed at minimizing training and inference times. Furthermore, multiple frameworks such as DGL [31], PyTorch Geometric [5], and NeuGraph [19] have enabled practitioners to more easily experiment with different GNN topologies while achieving high performance. However, most of these frameworks and optimizations focus on GNNs that solely operate on static graphs.

Recently, Temporal GNNs (TGNNs) have gained popularity for their ability to learn predictive models on dynamic graphs, which are graphs that evolve their topologies over time with new nodes or edges. Among these TGNN models, Temporal Graph Attention Networks (TGAT) [37] have seen adoption in applications ranging from modeling temporal knowledge graphs [10, 11] to temporal propagation-based fake news detection [26]. TGAT performs predictive modeling on continuous-time dynamic graphs, processing batches of edge interactions that occur across time. Compared to its static counterparts, TGAT training and inference is computationally more expensive as it relies on a complex attention mechanism, along with other encoding operations, to capture the temporal nature of neighborhoods [41]. Therefore, optimizing TGAT remains challenging, yet important to increase end-user productivity.

In this paper, we introduce techniques to increase TGAT inference performance by primarily focusing on eliminating redundant computations. Redundancy elimination has been explored in the context of static GNNs. For example, HAG [12] and ReGNN [2] are methods to reduce redundancies by identifying and reusing aggregations on overlapping neighbors. These methods require conversion to a custom computation graph representation and are restricted to simple aggregation operators, thus not applicable to attention-based models like TGAT. In the temporal context, there is limited

effort at optimizing TGNNs. One such work is [41] which replaces self-attention with simpler calculations and focuses on FPGA-based accelerators. Comparatively, in this paper, we take a different approach of focusing on the time aspects of TGAT and exploiting temporal redundancies that occur across time and inference executions. We observe and leverage redundancies in temporal embedding and time-encoding computations to considerably reduce TGAT inference runtime. In contrast to prior work, the redundancy elimination techniques we consider are not restricted to simple aggregations and do not require replacing self-attention.

We observe three main sources of redundant computations during TGAT inference. First, processing the source and destination nodes of edge interactions within a batch can lead to the same node embedding calculations. For some graphs, there can be as much as 55% of these same calculations in a batch. Second, calculating embeddings for a given node and timestamp will explore the same temporal neighborhoods that have been explored in a previous time step, resulting in recalculations of the same embeddings. In our analysis, we have found that some dynamic graphs can repeat calculations for 89.9% of the total embeddings that get generated during their evolutionary lifetime. Third, the time-encoding operation in TGAT is frequently invoked with the same time delta values. We elaborate on these observations in §3.

With these key observations, we developed TGOpt to exploit these opportunities by reusing values instead of recomputing them during TGAT inference. TGOpt accelerates inference computation by performing: 1) deduplication of repeated nodes/timestamps when processing a batch of edge interactions, 2) memoization of previously computed node embeddings, and 3) precomputation of time-encodings for a select window of time delta values. These techniques are semantic-preserving redundancy-aware transformations that preserve the computation semantics and end-user interfaces of the baseline version, leading to transparent performance improvements from the end-user's perspective. Node embeddings generated by TGOpt are the same, within floating-point tolerance, as from the baseline and thus retains model accuracy. The reuse exploiting techniques in TGOpt require additional memory to cache computed values for later reuse. To balance between performance from these reuses and memory consumption, TGOpt offers settings to limit the memory footprint of its computed values cache.

To evaluate our approach, we compared the inference performance of TGOpt against the baseline implementation of TGAT on both CPU and GPU environments. Our results show that TGOpt achieves a geomean speedup of 4.9× on CPU and 2.9× on GPU for a wide variety of dynamic graphs, with up to 6.3× speedup for the Reddit Posts dataset on CPU. We elaborate on our results and findings in §5. Overall, our results show that TGOpt yields substantial speedups for TGAT inference across datasets and machine environments while preserving model semantics and accuracy.

This paper makes the following specific contributions:

- **Exploration of redundancies that exist in Temporal Graph Attention Networks.** We perform a systematic study that examines the temporal redundancies that exist in TGAT in three different areas: within edge interaction batches, within embedding calculations that explore the temporal neighborhood, and within time-encodings.
- **Optimizations that exploit redundant computations in the context of TGAT inference.** We minimize redundant computations by introducing techniques that perform deduplication, memoization, and precomputation to reuse previously computed values.
- **Implementation and evaluation on a wide variety of dynamic graphs on both CPU and GPU.** We built TGOpt that implements the redundancy-aware optimizations for TGAT to allow transparent inference performance benefits to end-users. We show that TGOpt is capable of yielding $3-6\times$ speedup on CPU and $2-3\times$ on GPU, while incurring reasonably low overhead and memory usage.

2 Background

In this section, we provide an overview of Temporal GNN notations, concepts, and abstractions of their computations.

GNN Operators. GNNs are neural network models that operate on graph data. A (static) graph is a 2-tuple $G = (V, E)$ where $V = \{v_1, v_2, ..., v_i\}$ is a set of nodes and $E \subseteq V \times V$ is a set of edges. Each node v_i has a feature vector $x_i \in \mathbb{R}^{d_v}$, and each edge a feature vector $e_{ij} \in \mathbb{R}^{d_e}$. GNNs act as graph operators that aggregate node features with local neighborhood information. They output node embedding vectors $h_i \in \mathbb{R}^{d_v}$ used for downstream tasks such as node classification and edge prediction. Node features are inherent attributes, while these node embeddings are computed representations.

Dynamic Graphs. While the graph structure of a static graph is stable, a *dynamic graph* evolves over time with new nodes or edges. We define a dynamic graph as a 2-tuple $G = (V(T), E(T))$ where the node/edge sets are parameterized by time. An edge $e_{ij}(t_j) \in E(T)$ represents an interaction between nodes v_i and v_j with edge timestamp $t_j \in T$. We also define the *temporal neighborhood* of a node v_i at time t to be $\mathcal{N}(i, t) = \{j : e_{ij}(t_j) \in E(T) \wedge t_j < t\}$, i.e. neighbor j is in the temporal neighborhood if it has an interaction with node v_i where the edge timestamp t_j is less than t. Two nodes can have multiple edges at different times, making a dynamic graph a multi-graph. So the temporal neighborhood may contain the same neighbor j but with different edge interaction timestamps. Since an edge is uniquely identified by the node indices i, j and time t_j, we will also use $e_{ij}(t_j)$ as notation for the edge feature vector. Also note that node features may change in a dynamic graph, but we assume they are static in this work.

Graph Representation. Dynamic graphs are represented in two ways [14]: discrete-time dynamic graphs (DTDGs)

or continuous-time dynamic graphs (CTDGs). A CTDG is a stream of timestamped graph events $\mathcal{G} = \{\delta(t_1), \delta(t_2), ...\}$ where the timestamps $0 \leq t_1 \leq t_2 \leq ...$ are ordered [32]. Event $\delta(t)$ indicates a change event, such as a change to a node's features or a new edge between two nodes (i.e. a new edge interaction). A DTDG is a sequence of static graph snapshots $\mathcal{S} = \{\mathcal{G}(t_1), \mathcal{G}(t_2), \dots\}$ taken at intervals, where $\mathcal{G}(t) = (V[0,t], E[0,t])$ can be derived from a CTDG [22]. Thus, DTDGs lose some time information that may be crucial for certain applications. In this work, we focus on CTDGs as it is the data that TGAT operates on and because it captures richer temporal information.

GNNs for Dynamic Graphs. There is a growing body of work on extending static GNNs to model and learn on dynamic graphs. Early work on these dynamic GNNs focus on generating embedding representations for discrete-time dynamic graphs. Their computations mainly involve applying a structural operator on individual DTDG snapshots and a temporal operator across snapshots, or a combination of these operators [8, 21, 23, 25]. Recent work focuses more on continuous-time dynamic graphs, with computations mainly based on structural neighborhood aggregations and temporal encoding methods involving granular timestamps or time deltas for new edge interactions in CTDGs [22, 32, 37, 42]. We follow these later works and adopt Temporal GNNs in this paper as the term for GNN models for dynamic graphs.

Temporal Message Passing. Many GNN models can be expressed in the message-passing style [6], which abstracts a GNN operator as three steps. To facilitate later discussions, we extend message-passing to capture the notion of time, which is a critical parameter for dynamic graphs:

$$m_j(t) = \text{msg}\left(h_i^{(l-1)}(t),\ h_j^{(l-1)}(\tau),\ e_{ij}(t_j)\right) \tag{1}$$

$$r_i(t) = \text{agg}\left(\{m_j(t) : j \in \mathcal{N}(i,t)\}\right) \tag{2}$$

$$h_i^{(l)}(t) = \text{upd}\left(h_i^{(l-1)}(t),\ r_i(t)\right) \tag{3}$$

where $\tau \leq t$. Common choices for τ are time t or edge timestamp t_j. $h_i^{(l-1)}(t)$ and $h_j^{(l-1)}(\tau)$ are the *temporal embeddings*, which are computed for a target node and time. We will refer to this target node-timestamp pair with the notation $\langle i, t\rangle$.

In brief, Eq. (1) describes *message creation* where a "message" vector is created for each neighbor j. Eq. (2) is *message aggregation* where messages are reduced into a single neighborhood vector. And Eq. (3) is *feature update* where it combines this vector with the node features to produce the embedding $h_i^{(l)}(t)$. The specific functions may be learnable or non-learnable, e.g. $\text{agg}(\cdot)$ might be a summation while $\text{upd}(\cdot)$ can be a neural network.

Furthermore, many GNN models use a layered architecture where the same GNN operator is stacked into L layers, thereby recursively aggregating neighbor information from L-hops away. Thus, the current layer l computes the output $h_i^{(l)}(t)$ using embeddings from the $(l-1)^{th}$ layer.

Temporal Graph Attention Network. The TGAT model is a Temporal GNN that can generate temporal embeddings for CTDG data. It learns a function $\Phi : T \to \mathbb{R}^{d_t}$ that maps a time value to a d_t-dimensional vector. This time-encoding technique allows it to capture temporal patterns of the graph. The time-encoding vector is then injected into the input features of a GNN operator, thereby incorporated into the output embeddings. TGAT's computations can be expressed using the temporal message-passing model:

$$z_i(t) = h_i^{(l-1)}(t) \,\|\, \Phi(0) \tag{4}$$

$$z_j(t) = h_j^{(l-1)}(t_j) \,\|\, e_{ij}(t_j) \,\|\, \Phi(t - t_j) \tag{5}$$

$$r_i(t) = \text{Attn}\left(z_i(t),\ \{z_j(t) : j \in \mathcal{N}(i,t)\}\right) \tag{6}$$

$$h_i^{(l)}(t) = \text{FFN}\left(r_i(t) \,\|\, h_i^{(l-1)}(t)\right) \tag{7}$$

where $\|$ is the concatenation operator, and FFN is a feed-forward neural network. Eqs. (4, 5) is the message creation step in Eq. (1), while Eq. (6) is the $\text{agg}(\cdot)$ in Eq. (2), and Eq. (7) is the update function seen in Eq. (3).

$\text{Attn}(\cdot)$ in Eq. (6) is the self-attention mechanism from [28], except now parameterized by time. For the optimizations we consider, the specifics are not essential and we abstractly refer to this attention mechanism as M (see [37] for details).

These equations make up one layer in the TGAT model architecture. The input to a layer is simply the target $\langle i, t\rangle$ while everything else is computed. Computing the embedding starts at the L^{th} layer (the starting layer) and works back through the layers until the first layer where $h_i^{(0)} = x_i$.

More importantly, note that TGAT's choice for τ is edge timestamp t_j, and the time value being encoded is the delta between the target time t and t_j (thus it is 0 for the target node). The time-encoding function is formulated as:

$$\Phi(\Delta t) = \cos\left(\omega \cdot \Delta t + \phi\right) \tag{8}$$

where ω, ϕ are learnable vectors and Δt is a time delta [41].

Temporal Sampling. While computing $h_i^{(l)}(t)$ for a target $\langle i, t\rangle$, a layer needs previous layer embeddings for the node and neighbors. All of the neighbors could be considered, but in practice, this is rarely done due to scalability concerns. Rather, a model typically limits to a max number of neighbors via sampling [9]. Common strategies for this include uniform/random sampling and most-recent neighbors sampling. While both work with the TGAT model, we only focus on most-recent sampling due to better performance characteristics as shown in [22].

Batched Training and Inference. The equations above are formulated from the perspective of a single target node. In practice, the TGAT model processes a batch of edge interactions together and generates embeddings for both the source and destination nodes. The batching procedure allows processing nodes in parallel by packing them into tensors.

3 Redundancies & Reuse Opportunities

We start by sharing observations and insights on redundancies that motivated our optimization work. We identify three main places where redundancy and the potential for reuse occur: duplication when processing edges in batches (§3.1), performing the same computations on the same temporal subgraphs (§3.2), and encoding the same time deltas (§3.3).

3.1 Duplication From Batched Edges

We observed that the specific way edges are packed into a batch leads to duplicate target $\langle i, t\rangle$ pairs for which duplicate embeddings are being computed. For CTDG data, a list of edges is grouped into a batch. Each edge $e_{ij}(t_j)$ is decoupled into their source and destination nodes, using the same t_j as the target timestamp in the resulting node-timestamp pairs.

In a graph structure, nodes often share common neighbors and this can lead to duplicate $\langle i, t\rangle$ pairs. For instance, suppose source nodes a, b have an edge to the same neighbor c at time t. As seen in Figure 1, this will result in $\langle c, t\rangle$ appearing in the batch twice, one being a duplicate. To be precise, we define the rule that given a batch of edges as a list of node-timestamps $\mathcal{B} = \{\langle v_i, t_i\rangle, \langle v_j, t_j\rangle, \dots\}$, a duplicate exists if there is a pair of elements such that $v_i = v_j \wedge t_i = t_j$.

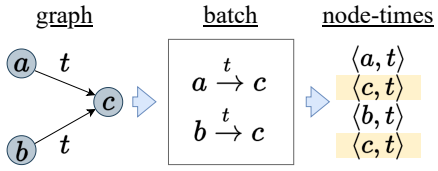

Figure 1. Example of new edges in a graph being grouped into a batch, resulting in a duplicate node-timestamp pair.

We reasoned that there are several scenarios where duplicates occur. First, they may exist in the batch $\mathcal{B}$ a model receives as input at the starting layer. Second, as a model recursively computes embeddings, the batching process means it will pool together neighbors of all the target nodes in the batch to serve as input to the previous layer, leading to potential duplicates. Lastly, when the recursion hits layer 0 where node features are retrieved, the target timestamps are irrelevant since features are static, so the rule above only needs to check the target node and this can result in duplicates when there might not have been any before. To inform our intuition, we have analyzed several datasets and found that duplicates can account for an average of 55% of the batch, and can rapidly increase down the layers as Table 1 shows.

3.2 Temporally Redundant Embedding Calculations

One of our main insights is that the model will inevitably compute the same embeddings for the same nodes over time. Our intuition is that not all the nodes in a dynamic graph are changing at once. When coupled with the fact that model parameters and weights do not change during inference time,

Table 1. Percentage of duplication per batch of 200 edges for each model layer, averaged over all batches in the dataset. See Table 2 for a summary of these datasets.

Dataset	TGAT layer		
	0	1	2
jodie-lastfm	94%	48%	0%
jodie-mooc	96%	74%	2%
jodie-reddit	88%	41%	0%
jodie-wiki	96%	68%	0%
snap-email	96%	55%	19%
snap-msg	96%	70%	16%
snap-reddit	83%	35%	8%

a layer will end up performing the same computations. In this case, a previously generated embedding could be reused instead of redoing the computation.

As the model recursively computes embeddings through the previous layers, so too will it recursively sample the neighbors. This in effect induces a L-hop subgraph for a L-layer model that starts at the target node. As neighbors are sampled, it needs to uphold the temporal constraint that $t_j < t$. During recursion, time t becomes t_j and this results in the property that all neighbors in the L-hop subgraph have edge timestamp less than the initial target time. We will refer to this induced L-hop subgraph as the *temporal subgraph*.

While examining this sampling process, we realized that a node's neighborhood can remain mostly the same despite a node changing. In particular, the choice of using most-recent sampling provides the nice property that the most recent neighbors of a node are mostly retained in the same relative order. This means that as a node evolves new interactions, older neighbors can still remain in the sampled temporal subgraph as long as they uphold the constraint of $t_j < t$ where t is now the new interaction time.

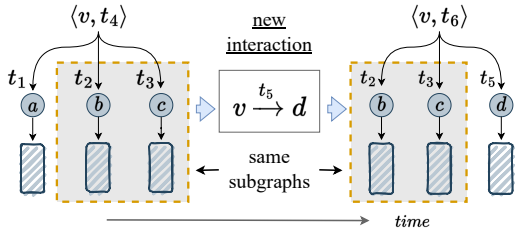

Figure 2. Example of a node v retaining most of its temporal subgraph over time when sampling 3 most-recent neighbors.

Take, for example, the scenario presented in Figure 2. When sampling for the target pair $\langle v, t_4\rangle$, the temporal subgraph will include neighbors a, b, and c with edge timestamps t_1, t_2, and t_3 respectively. Suppose now node v has a new interaction with a new neighbor d at time t_5. When we perform sampling again at time t_6, this new neighbor will be included

in the subgraph. However, notice that neighbors b and c are still retained in node v's neighborhood, and their sampled subgraphs will remain unchanged as well since their edge interaction timestamps are still t_2 and t_3.

In fact, we can generalize from this example and claim that given the same target $\langle i, t\rangle$, the sampled temporal subgraph will be exactly the same. In Figure 2, if we sample for $\langle v, t_4\rangle$ a second time before the new interaction, then this is trivially true since there have not been any changes. When we sample $\langle v, t_4\rangle$ again after the new interaction, the new neighbor will not be considered since it violates the temporal constraint (i.e. edge time must be $< t_4$). Thus, the resulting temporal subgraph will be the same as before.

When the induced temporal subgraph is the same, it implies that the model will be performing the same computations on the same set of nodes, neighbors, and timestamps. To validate if we can exploit this observation in order to reuse previously generated embeddings, we performed an analysis on the embeddings being computed in dynamic graphs. Specifically, we ran model inference on each edge of the graph and tracked the embeddings being generated and how many could be reused based on our insight above.

From this initial analysis, we observed that there are compelling and favorable opportunities for reusing computed embeddings. Figure 3 illustrates the trend of embeddings being reused versus recomputed for one of the datasets we studied. This trend also showed us that as a graph evolves over time, the amount of embeddings that could be reused also increases. At its peak, the ratio of reused embeddings to recomputed ones is about 8.9 to 1, or about 89.9% of the total embeddings. In fact, we see that the number of reuses already exceeds the number of recomputed embeddings after the first few days of its temporal evolution.

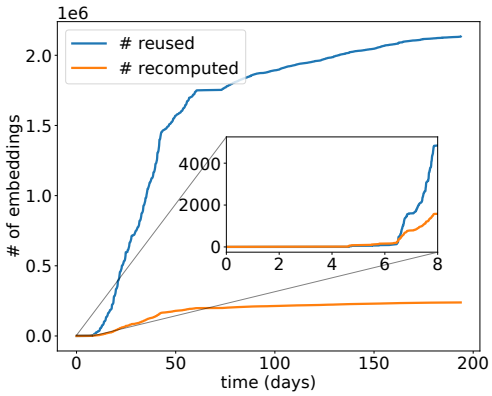

Figure 3. Embeddings reused vs. recomputed over time for the `snap-msg` dataset (x-axis based on edge timestamps).

3.3 Repeated Time Encodings

One last insight is in regards to how TGAT encodes and injects time information into the embedding computation. Returning to Eqs. (4, 5), we immediately notice that it always uses 0 for the time-encoding of $z_i(t)$. Performing this computation every time is unnecessary—since the weights for the time-encoder are fixed at inference time, we can compute this once ahead-of-time and reuse it indefinitely.

Meanwhile, for $z_j(t)$ we noticed that the time-encoder is repeatedly encoding the same time deltas (Δt). In fact, we observed that Δt follows a power-law distribution and clusters near 0, as Figure 4 shows. We note that the authors in [41] reported the same observation about the nature of Δt but it was made with a different interpretation and context. In our case, we reasoned that because the model uses most-recent sampling, the time difference $t - t_j$ will be relatively small and close to 0. Plus, sampling the same neighbors and inducing the same temporal subgraphs means the time-encoder will frequently encounter the same Δt. Once again, we see this as an opportunity to avoid doing computations by reusing previously computed values.

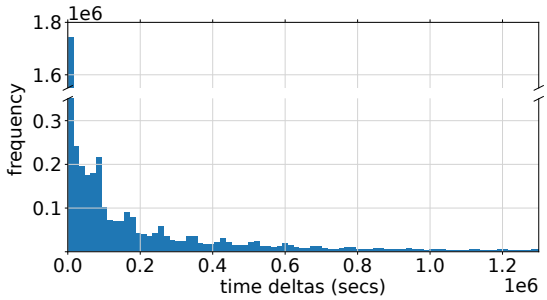

Figure 4. Distribution of time delta values processed by the time-encoder for the `snap-msg` dataset.

4 Redundancy-Aware Optimizations

In this section, we propose optimizations to exploit the redundancies we presented in §3. All our optimizations preserve model semantics and produce the same outputs as before, thereby conserving model accuracy as well.

Algorithm 1 outlines the inference function in our TGOpt system. It uses a memoization cache in the computation of temporal embeddings for a batch of target $\langle i, t\rangle$ pairs, and is a drop-in replacement for the original TGAT inference function. Cache lookup on line 8 is performed before any computations are done. When there are cache misses, it samples the neighborhood (the `NghLookup` operation) and carries out the original calculations, before finally storing the results on line 17 (§4.2). Deduplication filtering and inverse mapping on lines 6 and 20 is executed before and after the cache operation (§4.1). And the `TimeEncode` operation on line 14 will internally use precomputed time-encodings (§4.3).

4.1 Deduplicating Nodes

As we explored in §3.1, many duplicates can exist when performing batch processing of edge interactions.

Algorithm 1: End-to-end redundancy-aware calculation of TGAT temporal embeddings with TGOpt.

Data: Dynamic graph $\mathcal{G}$, node features X, edge features E, GNN attention operator M, sampling N number of neighbors
Input: Current layer l, target nodes and times (ns, ts)
Output: Node embedding features H for layer l

1 **function** ***Embed***(l, ns, ts)
2 **if** $l == 0$ **then**
3 $H \leftarrow$ lookup node features for ns in X;
4 **return** H;
5 **end**
6 $ns, ts, inv\_idx \leftarrow$ `DedupFilter`(ns, ts); ▷ §4.1
7 $keys \leftarrow$ `ComputeKeys`(ns, ts); ▷ §4.2
8 $hits, H \leftarrow$ `CacheLookup`$(keys)$; ▷ §4.2
9 **if** *not all hits* **then**
10 shrink $keys, ns, ts$ lists to only the misses;
11 $ns_{ngh}, ts_{ngh} \leftarrow$ `NghLookup`$(\mathcal{G}, N, ns, ts)$;
12 $H^{(l-1)} \leftarrow$ ***Embed***$(l-1, ns \cup ns_{ngh}, ts \cup ts_{ngh})$;
13 $\Delta t \leftarrow ts - ts_{ngh}$;
14 $H_t \leftarrow$ `TimeEncode`$(0, \Delta t)$; ▷ §4.3
15 $H_e \leftarrow$ lookup edge features in E;
16 $H_m \leftarrow M(H^{(l-1)}, H_t, H_e)$; ▷ Eqs.(6,7)
17 `CacheStore`$(keys, H_m)$; ▷ §4.2
18 copy H_m into H;
19 **end**
20 $H \leftarrow$ `DedupInvert`(H, inv_idx); ▷ §4.1
21 **return** H;
22 **end**

The deduplication filter (`DedupFilter`) preprocesses the input batch $\mathcal{B}$ and produces unique elements. In practice, batch $\mathcal{B}$ is represented as two arrays: one is a list of nodes and the other is a list of edge timestamps, both of the same length. The source and destination node of each edge in the batch is concatenated to form one node list. Notice that TGOpt only applies this filter to the input node and timestamp lists for layers $l > 0$. Although Table 1 indicates that many duplicates exist in the input to layer 0, since it only needs to lookup the node features, there is no need to apply `DedupFilter`.

In addition, an inverse index (`inv_idx`) is used to map the unique items back to the original arrays. This index is used after computations are done in order to produce output embeddings of the expected shape and with duplicated results, so to preserve semantics and produce results that are comparable with the baseline implementation.

Optimizing the Filter. One can easily implement the operations `DedupFilter` and `DedupInvert` using the NumPy and PyTorch libraries. However, doing so for `DedupFilter` will incur unnecessary overhead and memory footprint. Since uniqueness is determined by both the node and timestamp,

Algorithm 2: Deduplication of target nodes and timestamps, and building the inverse index.

Input: Target nodes and times (ns, ts)
Output: Unique target lists and inverse index

1 $ns_{uniq}, ts_{uniq}, inv\_idx \leftarrow \{\}, \{\}, \{\}$;
2 $processed \leftarrow$ mapping from key to index;
3 **for** $i \in 0..size(ns)$ **do**
4 $n, t \leftarrow$ retrieve element i from (ns, ts);
5 $key \leftarrow$ `Hash`(n, t);
6 **if** *key already in processed* **then**
7 $idx \leftarrow$ get index from *processed*;
8 $inv\_idx \leftarrow inv\_idx \cup \{idx\}$;
9 **else**
10 $idx \leftarrow$ current size of ns_{uniq};
11 $inv\_idx \leftarrow inv\_idx \cup \{idx\}$;
12 $ns_{uniq} \leftarrow ns_{uniq} \cup \{n\}$;
13 $ts_{uniq} \leftarrow ts_{uniq} \cup \{t\}$;
14 store key, idx in *processed*;
15 **end**
16 **end**
17 **return** $ns_{uniq}, ts_{uniq}, inv\_idx$;

one will need to first construct a 2-D tensor by concatenating the two arrays, which consumes memory.

We outline an approach in Algorithm 2 that jointly operates on the two separate arrays in order to avoid creating intermediate tensors. It checks uniqueness by using a `Hash` function that yields collision-free hash values (`key`). Since the node and timestamp are 32-bit values, `Hash` constructs a 64-bit value by bitwise shifting and OR-ing the two values, which is efficient and provides the collision-free property.

4.2 Memoization of Embeddings

Our goal is to persist computed embeddings so they could be reused at a later time, which requires a way to identify each embedding and to retrieve them when needed. We develop an efficient scheme based on memoization techniques to achieve this. Generally, a memoization cache maps inputs to an output, with different inputs resulting in different outputs. In our case, the output is the embedding tensor $H^{(l)}$ and the inputs will be the values needed to compute the embedding. We first discuss computing cache keys for the inputs, then go into details about storing/looking up values in §4.2.2.

4.2.1 Computing Cache Keys. Lines 11-15 in Algorithm 1 suggest that we will need to consider at least the list of neighbors, their edge timestamps, edge features, and $H^{(l-1)}$ as inputs. The inputs will generally be combined into a single key value by hashing, but doing this for the full set of inputs will be detrimental to performance.

As we have established in §3.2, sampling for the same target $\langle i, t \rangle$ will yield the same temporal subgraph, and the same

computations will be performed. As a consequence, we can exploit this to exclude everything else except for the target $\langle i, t \rangle$ pair in our consideration. Therefore, the `ComputeKeys` operation only needs to consider these two values in order to produce a key that maps the inputs to stored embeddings. The same hashing function introduced for the deduplication filter is used for this purpose as well.

Also note that since each $\langle i, t \rangle$ pair in a batch is independent of each other, the `ComputeKeys` operation can be performed in parallel across the pairs, which we exploit in our implementation of TGOpt.

4.2.2 Cache Storage and Lookup.

The storage scheme for keeping the computed embeddings will affect how cache lookup operates. TGOpt currently has a simple scheme of using a hash table that maps a key to an embedding vector. With this scheme, cache lookup will only need to search a single data structure and this also makes other bookkeeping simple as well, but more sophisticated design choices are possible. Algorithm 3 presents the `CacheStore` operation.

Algorithm 3: Store operation for the TGOpt cache.

Input: List of cache *keys*, embeddings H_m
(each key in *keys* corresponds to a vector in H_m)

```
size ← current cache table size + size of keys;
if size > limit then
    evict items from the cache table;
end
if using GPU then
    move H_m to CPU device;
end
for key ∈ keys do
    h_m ← next embedding vector in H_m;
    store key, h_m in cache table;
end
```

As for the `CacheLookup` operation, TGOpt searches the cache table for hits using the given list of keys and returns an embedding tensor. To avoid creating intermediate tensors, it will construct the final embedding tensor (H) of the expected shape, and partially fills it in during the search. Any missing embeddings will be indicated by the hit index that is returned alongside H. Finally, if using a GPU device for computations it will move the tensor to the GPU before returning it.

We also note that each of the keys can be operated on independently, so the main loop in both `CacheStore` and `CacheLookup` can be parallelized, given that TGOpt uses a concurrent hash table implementation. We selectively parallelize these operations depending on the hardware.

Storage Memory Limit. In order to be conscious of memory usage, we impose a memory budget on TGOpt's cache. The `CacheStore` operation will check the size of the cache against this limit, and evict items as necessary. Currently, it uses a simple FIFO eviction policy. We further reduce memory usage by only caching the $L - 1$ layers. This is based on the fact that $H^{(l-1)}$ is required to compute $H^{(l)}$. While on the flip side, $H^{(L)}$ of the last layer is not required for other computations. What cache limit to adopt is a decision that balances between performance and memory usage. By setting a lower limit, the performance gain will be lower since more computations will need to be done, while a higher limit allows for more reuse but higher memory usage as well. We explore this tension in our experiments in §5.2.4.

Storage Memory Location. Another design decision is where to store the embeddings, whether on CPU or GPU memory. When running inference on GPU, tensors need to reside on the same device. In this work, we propose to store the cached embeddings solely on the CPU rather than GPU. This means the `CacheStore` and `CacheLookup` operations will incur data movement costs. But we reasoned that the data size during cache lookup is often small, and the current lookup approach emphasizes doing many small data copies as hits are found, all of which are not favorable to GPUs. We present an analysis of our design choice in §5.2.5.

4.3 Precomputing Time Encodings

To reduce the redundancy mentioned in §3.3, TGOpt precomputes time-encoding vectors in advance before running inference. Since the time-encoder parameters are fixed during inference and Δt are simple values with no other dependencies, we can directly apply the $\Phi(\cdot)$ function and keep the time vectors for later use.

Unlike the lookup table described in [41] which is split into 128 time intervals, TGOpt will instead precompute a select window of Δt values starting at 0. Because this time window is contiguous from 0, the Δt value itself can serve as an index into a dense tensor that stores the time-encoding vectors, allowing for a simple lookup process. However, we will still need to account for any misses and perform the original computation for those. Thus, the `TimeEncode` function operates similarly to `CacheLookup`, in that it constructs the final time-encoding tensor (H_t) of the expected shape and partially fills it in with hit vectors.

5 Evaluation

We evaluated TGOpt against the baseline on a wide variety of dynamic graph datasets, achieving geomean speedups of 4.9× on CPU and 2.9× on GPU. We summarize our experimental setup, results, and analysis in this section.

5.1 Experimental Setup

Our main metric is model runtime on a standard inference task. To simulate the evolution of a graph over time, the inference task is set up to iterate through all edges in a graph ordered chronologically and in batches of 200. For each batch, we run the model to generate temporal embeddings. We

measure runtime as the total time to iterate through all the edges in the dynamic graph dataset. The model is trained according to standard training procedures for link prediction, and the trained parameters are then used during inference.

5.1.1 Baseline Model and Datasets. Our baseline is the official TGAT implementation[1], which we updated to use more recent versions of Python (v3.7) and PyTorch (v1.12). The TGAT model used is 2 layers, 2 attention heads, sampling of 20 most-recent neighbors, and the rest are default settings.

We evaluated on bipartite and homogeneous dynamic graphs of various sizes. Following the baseline, bipartite graphs are treated as homogeneous, and all graphs are considered as undirected. Further description of the datasets follow, while Table 2 summarizes their data statistics.

- Bipartite: `jodie-lastfm`, `jodie-mooc`, `jodie-reddit` (Reddit Posts), and `jodie-wiki` are widely-used bipartite graphs with user and item node types. We use the curated datasets from JODIE [17] for all four.
- Homogeneous: `snap-email` is a dataset of emails sent at a research institute. `snap-msg` captures messages between users on a college social media-like network. `snap-reddit` represents hyperlink references between subreddits made within a post's title or body. All three are from the SNAP data repository [18], under the names CollegeMsg, email-Eu-core-temporal, and soc-RedditHyperlinks, respectively.

Table 2. Datasets for our evaluation of TGOpt. Node features use a zero-vector with the same dimension as the edge features. †A randomly generated 100-dimensional vector is used when missing edge features.

Dataset	$\|V\|$	$\|E\|$	$d_e^\dagger$	$\max(t)$
`jodie-lastfm`	1,980	1,293,103	-	1.4e8
`jodie-mooc`	7,144	411,749	4	2.6e6
`jodie-reddit`	10,984	672,447	172	2.7e6
`jodie-wiki`	9,227	157,474	172	2.7e6
`snap-email`	986	332,334	-	6.9e7
`snap-msg`	1,899	59,835	-	1.1e9
`snap-reddit`	67,180	858,488	86	1.5e9

5.1.2 Machine Environment. We conducted our experiments on two different machines. One is a CPU server equipped with 2x Intel Xeon Gold 6348 @ 2.6GHz, 28 cores each, and 1TB Mem. Our GPU machine is an AWS p3.2xlarge instance provisioned with 8 vCPUs @ 2.3GHz, 61GB Mem, and an Nvidia Tesla V100 GPU (16GB).

5.1.3 Implementation. TGOpt uses Python and PyTorch (same versions as the baseline). Portions of TGOpt are implemented as a PyTorch C++ custom extension, which is then exposed as a Python module via PyBind11. All three cache operations in TGOpt are parallelized on the GPU machine to help spread work across the slower CPU cores, while on the CPU server only `CacheLookup` is parallelized so to minimize on synchronization overhead. For cache size, we limit to 2 million embedding items. Considering that most embeddings are a 100-dimensional vector of 32-bit floats, this limit roughly translates to a limit of less than 1GB. For the time-encodings, we chose 10,000 as the time window.

We use a custom C++ parallel neighborhood sampler that we implemented instead of the official implementation for both TGOpt and the baseline. We observed that the original sampler in Python has noticeable runtime fluctuations, which is not ideal for experimental measurements. Our sampler is inspired by the one proposed in [42]. The custom sampler helps speed up graph operations and provides more consistent performance, which aids in experimentation.

All the optimizations that TGOpt implements are semantic-preserving and produce the same final embedding values as the baseline, within floating-point tolerance of 1e−5 or 1e−6. To validate this and to confirm the correctness of TGOpt, we collected and compared the embeddings from both TGOpt and the baseline for all of our datasets.

5.2 Results

We present our experimental findings below. We report inference performance results for all graphs and our in-depth analyses on two select datasets. For our ablation study (§5.2.2) and other analyses (§5.2.3, 5.2.4, 5.2.5), we focus on `jodie-lastfm` and `snap-msg` as representative datasets of large/small and bipartite/homogeneous graphs, respectively.

5.2.1 Inference Performance. As Figure 5 demonstrates, TGOpt significantly outperforms the baseline on all datasets and machine environments. It yields speedups in the range of $3-6\times$ on CPU and $2-3\times$ on GPU.

As the results indicate, performance gain tends to be higher on CPU than GPU. As we show in our breakdown analysis (§5.2.3), tensor operations on GPU are much more efficient than CPU, and by reusing values TGOpt is able to minimize the expensive recomputation of embedding tensors on CPU. Among the CPU results, we see that the bipartite `jodie-*` datasets attain higher speedups than the `snap-*` datasets. In [17], the JODIE authors mention that the bipartite datasets were specifically curated for users' repetitive behavior of interacting with the same item consecutively. Naturally, this results in new interactions with a single neighbor but the rest of the neighborhood is mostly unchanged, which is exactly the kind of redundancy TGOpt exploits. Specifically, one implication of this is that there will be more repeated time-encodings of the same time deltas for the unchanged neighborhoods. As our ablation study shows, the precomputation optimization is able to exploit this to yield higher speedups for the `jodie-*` datasets on CPU.

[1] https://github.com/StatsDLMathsRecomSys/Inductive-representation-learning-on-temporal-graphs

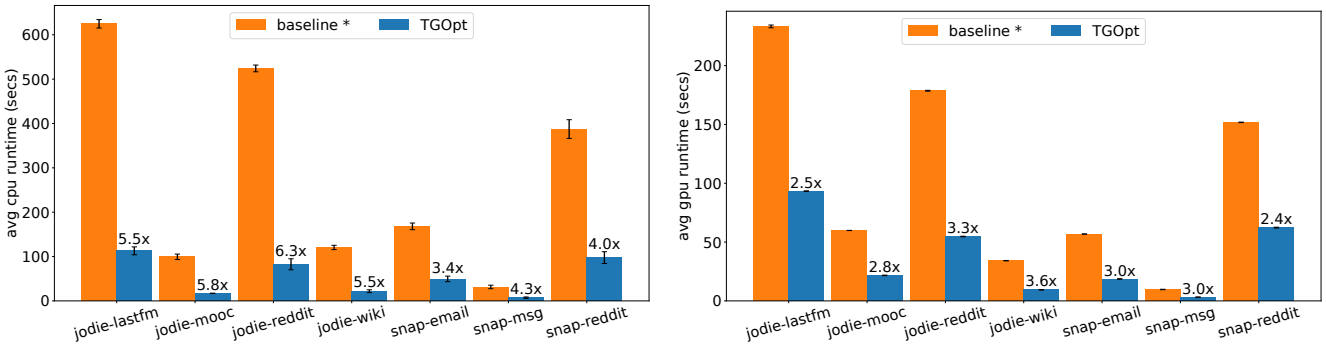

Figure 5. Inference performance across various datasets (left is CPU server, right is GPU machine). Runtime is averaged over 10 runs. Line at top of bars is its standard deviation. Bar label for TGOpt is speedup. *baseline is the TGAT official code.

Meanwhile, the results on GPU are more consistent, with lower standard deviations than the CPU results. Since TGOpt uses CPU memory for its cache, this allows for better utilization of each device's physical cache hierarchies and less memory contention, thereby yielding more consistent GPU speedups. Although everything took longer to run on CPU, the relative performance of the datasets against each other is similar for both machines (e.g. `jodie-lastfm` takes about 20 times longer to run than `snap-msg` on both).

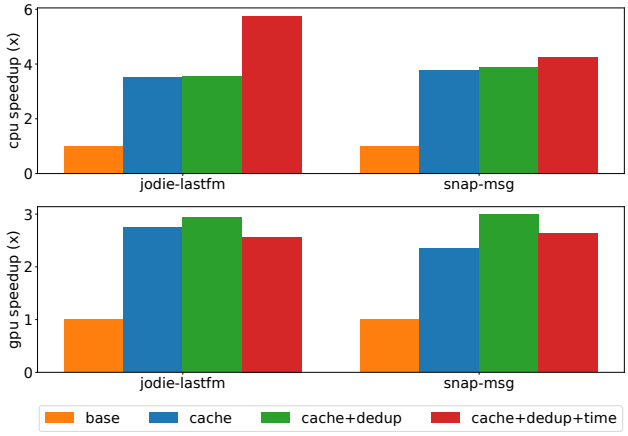

Figure 6. Accumulative inference speedup when applying optimizations sequentially.

5.2.2 Ablation Study. To better understand the contribution from each of our three optimizations, we carried out an ablation study where we incrementally enabled each one. As Figure 6 (top) shows, we saw a speedup of at least 3× just by applying the cache optimization on CPU. Enabling deduplication contributed a slight increase to the speedup, while enabling time precomputation yielded a bigger boost to performance. In fact, the `jodie-lastfm` dataset saw a significant jump in speedup. Since tensor computations are more expensive on CPU (as shown in Table 3), the avoidance of recomputations afforded by the precomputation optimization outweighs the incurred overheads. When coupled with the repetitive behavior of interactions as mentioned in §5.2.1, this leads to more noticeable performance gains for the `jodie-*` datasets.

To gain another perspective, we ran the same ablation study on the GPU machine (Figure 6 bottom). Enabling the cache/dedup optimizations yielded at least 2× speedup for both datasets, but time precomputation produced a small regression. As the cost breakdown in Table 3 shows, the time-encoding lookup contributes overhead to the `TimeEncode` operation, which is negatively impacting performance on the GPU machine. Since GPUs are efficient at performing the time-encoding tensor operations, the cost savings from reused time-encoding values become marginal while lookup overheads become more noticeable. From this, it appears that tuning the optimizations to the hardware (and perhaps the dataset as well) could yield better performance gains.

5.2.3 Breakdown Analysis. We conducted a breakdown analysis of the cost of each major operation in TGOpt, along with other metrics. We have three main conclusions.

First, by caching and reusing embeddings TGOpt is able to avoid heavy operations while lowering the cost of other operations. In Table 3, for the GPU machine we noticed the `NghLookup` operation took the most time on the baseline, which TGOpt is able to significantly reduce. For instance, `jodie-lastfm` saw a decrease of 132 seconds while incurring just 44 seconds of overhead for the cache/dedup operations. `TimeEncode` of zeros involves creating intermediate zero tensors and data movement, thus having higher cost on GPU compared to `TimeEncode` of neighbor Δt which only involves tensor computations. TGOpt helps avoid most of the cost of `TimeEncode` (0) by using precomputed values, but incurs noticeable overhead on GPU for `TimeEncode` (Δt). Meanwhile, on the CPU server tensor computations tend to dominate the runtime. The "attention M" operation is the GNN operator of Eqs. (6, 7). TGOpt avoids running this operation as well as the `TimeEncode` operation by reusing values, while incurring similarly low overheads for the cache/dedup operations. The overhead associated with `TimeEncode` (Δt) is outweighed by the reduction of recomputations on CPU.

Table 3. Total runtime of operations from Algorithm 1 on both the CPU server and GPU machine. Two additional metrics are shown, with values the same on both machines.

Operation (secs)	jodie-lastfm		snap-msg	
	base	ours	base	ours
CPU server				
NghLookup	99.09	10.74	4.35	0.44
DedupFilter	-	5.19	-	0.18
DedupInvert	-	0.78	-	0.06
TimeEncode (0)	5.38	0.53	0.27	0.02
TimeEncode (Δt)	149.93	45.33	6.17	5.36
ComputeKeys	-	0.49	-	0.01
CacheLookup	-	10.57	-	0.30
CacheStore	-	3.73	-	0.39
attention M	314.21	32.50	19.25	4.39
GPU machine				
NghLookup	147.84	15.33	6.17	0.48
DedupFilter	-	10.65	-	0.23
DedupInvert	-	1.35	-	0.07
TimeEncode (0)	54.50	0.73	2.52	0.04
TimeEncode (Δt)	5.00	12.01	0.30	0.58
ComputeKeys	-	1.73	-	0.03
CacheLookup	-	27.73	-	0.99
CacheStore	-	2.72	-	0.14
attention M	22.53	15.15	1.75	1.30
average hit rate (%)	-	90.94	-	85.85
used cache size (MiB)	-	931	-	46.5

Second, the average hit rate shows that we are getting sustained cache usage from TGOpt. The hit rate in Table 3 shows the overall hit rate averaged across all batches. Looking at the hit rate trend in Figure 7, we see that the hit rate reaches about 80% very early on and continues to increase, indicating that cache reuse grows with time.

Lastly, although TGOpt has a cache limit, the datasets were still able to attain performance benefits. `snap-msg` stores at most 100,007 cache items, which is well within the limit, as its approximate used cache size of 46.5MiB shows. `jodie-lastfm`, being one of our larger datasets, stores at most 2,581,675 cache items which far exceeds the cache limit. Despite this limit, we were able to achieve a 2.5× speedup on GPU while consuming no more than 1GB of memory.

5.2.4 Cache Memory Usage. The balance between performance and memory usage is an important trade-off when considering TGOpt's cache. Table 4 shows results for other cache limits on the GPU machine aside from the 2M we used for inference runtime in §5.2.1 (similar trends can be observed for CPU). As the table shows, a lower limit means fewer cached values for reuse which leads to longer runtimes. The effect on a small dataset like `snap-msg` is marginal, but becomes significant for a larger dynamic graph like

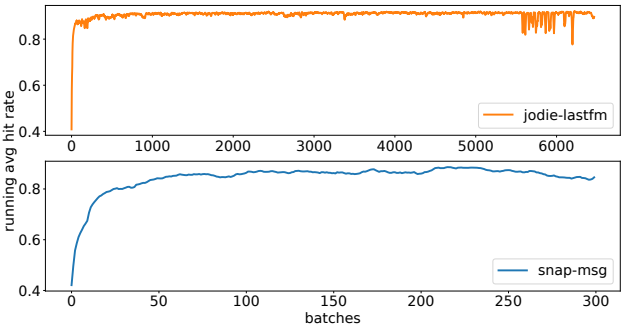

Figure 7. Evolution of TGOpt's cache hit rate, averaged over a sliding window of the last 10 batches.

Table 4. Runtime on GPU machine (top) and memory usage (bottom) when varying the TGOpt cache limit of embeddings.

Dataset	# of embeddings			
	10K	100K	1M	3M
jodie-lastfm	153.48s	103.52s	94.04s	89.41s
	4.7MiB	46.5MiB	465MiB	1201MiB
snap-msg	5.26s	4.13s	4.07s	4.04s
	4.7MiB	46.5MiB	46.5MiB	46.5MiB

`jodie-lastfm`. On the other hand, memory usage increases as we set higher limits. We have chosen 2M as it yields good performance while keeping memory footprint of the TGOpt cache to under 1GB for the various datasets we studied.

5.2.5 Cache Storage Analysis. In regards to memory, we further analyzed the trade-off of storing cache items on CPU versus GPU device memory. To shed light on this, we profiled all data movement overhead between CPU and GPU, using the time spend on CUDA `memcpy` as a proxy measure (as obtained from the `nvprof` tool[2]).

As Table 5 shows for the two select datasets, storing cache items on GPU memory incurs significant overhead. The three main data movement costs are: host-to-device (HtoD), device-to-host (DtoH), and device-to-device (DtoD). Storing on CPU incurs a noticeable increase in cross-device overhead, which is to be expected. But this is a reasonably low percentage of GPU activity time, about 13% for `jodie-lastfm`. But when storing on GPU, the DtoD cost saw a significant jump and dominates the time spend (~75% for `jodie-lastfm`). In other words, the GPU is spending a majority of its time doing data movement within device and less time on executing kernels.

This analysis corroborates that our current caching scheme is not favorable to GPUs and that keeping cache items on CPU memory yields better performance, although we note that this can change given a different design.

[2] https://docs.nvidia.com/cuda/profiler-users-guide/index.html

Table 5. Overall time spent on data movement between CPU and GPU. Parenthesis is % of all GPU activities.

Dataset	CUDA memcpy, storing on CPU			CUDA memcpy, storing on GPU		
	HtoD	DtoH	DtoD	HtoD	DtoH	DtoD
jodie-lastfm (s)	1.08 (08.37%)	0.61 (4.71%)	0.02 (0.16%)	0.45 (0.96%)	0.30 (0.64%)	34.81 (74.87%)
snap-msg (ms)	50.51 (10.11%)	11.66 (2.33%)	0.95 (0.19%)	23.34 (1.87%)	8.44 (0.68%)	780.61 (62.43%)

6 Related Work

Optimizing Temporal GNNs. Compared to static GNNs, there has been limited work on optimizing TGNNs. To the best of our knowledge, only [41] proposes applying a redundancy related optimization of precomputing a time-encoding lookup table, which is hardcoded to 128 intervals. More importantly, the self-attention in TGNNs was replaced with a simplified version, thereby altering the semantics, whereas our work retains the semantics and model accuracy. Another work proposes the TGL framework for efficient TGNN training [42]. The proposed contributions do not include any redundancy optimizations. Rather, the proposed T-CSR data structure and parallel sampler helps with graph operations and is otherwise orthogonal/complementary to TGOpt.

Caching for Temporal GNNs. A recent system called DynaGraph [8] utilizes a technique for caching intermediate message-passing results. This is different from our scheme of caching the final embeddings. Additionally, the DynaGraph system only supports models that operate on the DTDG representation, with caching done in the context of graph snapshots. It is not clear how this can be extended to models for CTDG data with embedding computations for streaming edge interactions, which is the focus of our work.

Redundancy Elimination for GNNs. There exists a body of work on optimizing static GNNs ([34–36, 39]), but few utilize redundancy-aware techniques, and none target dynamic graphs. Computation redundancy has been explored in [12] which proposes the HAG abstraction. The HAG representation eliminates redundant computations while retaining model accuracy by reusing intermediate aggregation results for overlapping subsets of neighbors. However, HAG is restricted to simple aggregation operators, thus not applicable to models that use the more complex self-attention mechanism such as the TGAT model that we studied. Along the same line of work is ReGNN [2], which proposes similar redundancy elimination techniques based on intermediate overlapping results. Further, ReGNN is more tailored to hardware accelerators, and is similarly restricted as HAG.

Redundancy Elimination for Deep Learning. Data redundancy optimizations have been well explored for Deep Neural Networks (DNN). One common trend is techniques for detecting redundancies in image, video, or text data and exploiting these to reduce computations and reuse intermediate results within or across DNN layers [3, 7, 20]. These often trade performance gains for a slight drop in model accuracy. For instance, Deep Reuse [20] reuses computation results for similar sub-vectors of image pixels, which are clustered by a similarity metric and have room for accuracy loss. Meanwhile, our work focuses on graph data, their redundancies arising from the temporal computations in TGAT, and optimizations that preserve model semantics and accuracy.

7 Future Work

While our work focuses on accelerating TGAT inference, an important future direction is exploring redundancy opportunities in the training process. During training, the model parameters will be updated via backpropagation, and so the effectiveness of the caching technique will be limited, but optimizations like deduplication can still be applied to bring benefits. There may be other computation or data redundancy opportunities in the training process.

Currently, our caching method assumes the most-recent neighbor sampling strategy. This makes reasoning about the temporal neighborhood straightforward and enables reuse without incurring overhead from comparing the neighborhood to what was used for previously computed embeddings. Future work can explore other sampling schemes and investigate efficient ways to compare the neighborhood for reuse. Other assumptions in our approach are that node features are static and the graph only evolves new edge interactions. Future work here will be to extend support for other graph change events, such as node feature changes and deletion of edges, in a way that efficiently updates the cache while maximizing reuse.

8 Conclusion

In this work, we introduced TGOpt, a system that targets redundancies arising from the temporal nature of dynamic graphs and TGAT computations. We detailed these redundancy opportunities and developed semantic-preserving optimization techniques of deduplication, memoization, and precomputation to eliminate them from the temporal embedding computations. In our experiments, we showed that TGOpt is able to yield considerable speedups for TGAT inference for a wide selection of dynamic graphs, while incurring reasonably low overhead and memory usage.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable feedback. We also thank Damitha Lenadora and Stefanos Baziotis for their proofreading and helpful comments.

References

[1] Adam Auten, Matthew Tomei, and Rakesh Kumar. 2020. Hardware Acceleration of Graph Neural Networks. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[2] Cen Chen, Kenli Li, Yangfan Li, and Xiaofeng Zou. 2022. ReGNN: A Redundancy-Eliminated Graph Neural Networks Accelerator. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 429–443.

[3] Jou-An Chen, Wei Niu, Bin Ren, Yanzhi Wang, and Xipeng Shen. 2022. Survey: Exploiting Data Redundancy for Optimization of Deep Learning. *ACM Comput. Surv.* (sep 2022).

[4] Dawei Cheng, Xiaoyang Wang, Ying Zhang, and Liqing Zhang. 2022. Graph Neural Network for Fraud Detection via Spatial-Temporal Attention. *IEEE Transactions on Knowledge and Data Engineering* 34, 8 (2022), 3800–3813.

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

[6] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70* (Sydney, NSW, Australia) *(ICML'17)*. JMLR.org, 1263–1272.

[7] Saurabh Goyal, Anamitra Roy Choudhury, Saurabh Raje, Venkatesan Chakaravarthy, Yogish Sabharwal, and Ashish Verma. 2020. PoWER-BERT: Accelerating BERT inference via progressive word-vector elimination. In *International Conference on Machine Learning*. PMLR, 3690–3699.

[8] Mingyu Guan, Anand Padmanabha Iyer, and Taesoo Kim. 2022. DynaGraph: Dynamic Graph Neural Networks at Scale. In *Proceedings of the 5th ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)* (Philadelphia, Pennsylvania) *(GRADES-NDA '22)*. Association for Computing Machinery, New York, NY, USA, Article 6, 10 pages.

[9] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[10] Zhen Han, Peng Chen, Yunpu Ma, and Volker Tresp. 2021. Explainable Subgraph Reasoning for Forecasting on Temporal Knowledge Graphs. In *International Conference on Learning Representations*.

[11] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. 2022. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. *IEEE Transactions on Neural Networks and Learning Systems* 33, 2 (2022), 494–514.

[12] Zhihao Jia, Sina Lin, Rex Ying, Jiaxuan You, Jure Leskovec, and Alex Aiken. 2020. Redundancy-Free Computation for Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA.

[13] Wengong Jin, Kevin Yang, Regina Barzilay, and Tommi Jaakkola. 2019. Learning multimodal graph-to-graph translation for molecular optimization. *ICLR* (2019).

[14] 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, 1, Article 70 (jan 2020), 73 pages.

[15] Taehyun Kim, Changho Hwang, KyoungSoo Park, Zhiqi Lin, Peng Cheng, Youshan Miao, Lingxiao Ma, and Yongqiang Xiong. 2021. Accelerating GNN Training with Locality-Aware Partial Execution. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems* (Hong Kong, China) *(APSys '21)*. Association for Computing Machinery, New York, NY, USA, 34–41.

[16] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

[17] 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.

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[19] Lingxiao Ma, Zhi Yang, Youshan Miao, Jilong Xue, Ming Wu, Lidong Zhou, and Yafei Dai. 2019. Neugraph: Parallel Deep Neural Network Computation on Large Graphs. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 443–457.

[20] Lin Ning and Xipeng Shen. 2019. Deep Reuse: Streamline CNN Inference on the Fly via Coarse-Grained Computation Reuse. In *Proceedings of the ACM International Conference on Supercomputing (ICS '19)*. Association for Computing Machinery, New York, NY, USA, 438–448.

[21] 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 Press, 5363–5370.

[22] 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*.

[23] 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* (Houston, TX, USA) *(WSDM '20)*. Association for Computing Machinery, New York, NY, USA, 519–527.

[24] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *Trans. Neur. Netw.* 20, 1 (jan 2009), 61–80.

[25] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. 2016. Structured Sequence Modeling with Graph Convolutional Recurrent Networks. *arXiv* (2016).

[26] Chenguang Song, Kai Shu, and Bin Wu. 2021. Temporally evolving graph neural network for fake news detection. *Information Processing & Management* 58, 6 (2021), 102712.

[27] Wen Torng and Russ B. Altman. 2019. Graph Convolutional Neural Networks for Predicting Drug-Target Interactions. *Journal of Chemical Information and Modeling* 59, 10 (2019), 4131–4149. PMID: 31580672.

[28] 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 *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[29] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. *ICLR* (2018).

[30] Daixin Wang, Jianbin Lin, Peng Cui, Quanhui Jia, Zhen Wang, Yanming Fang, Quan Yu, Jun Zhou, Shuang Yang, and Yuan Qi. 2019. A Semi-Supervised Graph Attentive Network for Financial Fraud Detection. In *2019 IEEE International Conference on Data Mining (ICDM)*. 598–607.

[31] 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).

[32] 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 '21)*. Association for Computing Machinery, New York, NY, USA, 2628–2638.

[33] Yuke Wang, Boyuan Feng, and Yufei Ding. 2022. QGTC: Accelerating Quantized Graph Neural Networks via GPU Tensor Core. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 107–119.

[34] Yuke Wang, Boyuan Feng, Gushu Li, Shuangchen Li, Lei Deng, Yuan Xie, and Yufei Ding. 2021. GNNAdvisor: An Efficient Runtime System for GNN Acceleration on GPUs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'21)*.

[35] Yidi Wu, Kaihao Ma, Zhenkun Cai, Tatiana Jin, Boyang Li, Chenguang Zheng, James Cheng, and Fan Yu. 2021. Seastar: Vertex-Centric Programming for Graph Neural Networks. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 359–375.

[36] Zhiqiang Xie, Minjie Wang, Zihao Ye, Zheng Zhang, and Rui Fan. 2022. Graphiler: Optimizing Graph Neural Networks with Message Passing Data Flow Graph. In *Proceedings of Machine Learning and Systems*, Vol. 4. 515–528.

[37] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *International Conference on Learning Representations (ICLR)*.

[38] Bingyi Zhang, Sanmukh R. Kuppannagari, Rajgopal Kannan, and Viktor Prasanna. 2021. Efficient Neighbor-Sampling-based GNN Training on CPU-FPGA Heterogeneous Platform. In *2021 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7.

[39] Hengrui Zhang, Zhongming Yu, Guohao Dai, Guyue Huang, Yufei Ding, Yuan Xie, and Yu Wang. 2022. Understanding GNN Computational Graph: A Coordinated Computation, IO, and Memory Perspective. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 467–484.

[40] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.), Vol. 31. Curran Associates, Inc.

[41] Hongkuan Zhou, Bingyi Zhang, Rajgopal Kannan, Viktor Prasanna, and Carl Busart. 2022. Model-Architecture Co-Design for High Performance Temporal GNN Inference on FPGA. In *36rd International Parallel and Distributed Processing Symposium*.

[42] Hongkuan Zhou, Da Zheng, Israt Nisa, Vassilis N. Ioannidis, Xiang Song, and George Karypis. 2022. TGL: A general framework for temporal GNN training on billion-scale graphs. In *VLDB 2022*.

[43] Zhe Zhou, Bizhao Shi, Zhe Zhang, Yijin Guan, Guangyu Sun, and Guojie Luo. 2021. BlockGNN: Towards Efficient GNN Acceleration Using Block-Circulant Weight Matrices. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1009–1014.

A Artifact Description

A.1 Overview

Our artifact packages together the source code, scripts, and data necessary to reproduce the results in the paper. It also contains scripts for setting up a Docker image or AWS instance for those familiar and prefer those environments.

For evaluating the artifact, we provide with the artifact the datasets used in our paper and already-trained models. Users can download/preprocess their own datasets and train the models themselves using our scripts as well. Running the full experiments for CPU and GPU takes approximately 10 hours in total on the machines we used in our paper.

A.2 Requirements

Running the experiments requires the following:

- **Operating System:** Our instructions assumes Linux, but should be adaptable to macOS and Windows.
- **Hardware:**
  - For CPU experiments: Multicore x86_64 CPU, preferably 2-socket NUMA machine (we tested on a 2 NUMA server with Intel Xeon Gold 6348).
  - For GPU experiments: Nvidia GPU (we tested on Tesla V100 16GB via an AWS instance).
  - Disk space: Artifact is 1.6 GBs (uncompressed 3.2 GBs).
  - Memory: At least 6 GBs of free memory.
- **Software:**
  - Python 3.7, PyTorch 1.12, and required dependencies in `requirements.txt` (we recommend using conda).
  - g++ >= 7.2, OpenMP >= 201511, Intel TBB >= 2020.1 (for the TGOpt C++ extension).
- **Datasets:** Provided under the `data/` subdirectory. See A.6 for preparing your own datasets.
- **Models:** Provided under the `saved_models/` subdirectory. See A.6 for training your own models.

Before running the experiments, we recommend minimizing background applications and processes, and quiescing the machine to put it in a more controlled and consistent state (e.g. disable hyper-threading, turbo-boost, etc).

A.3 Artifact Access

The artifact is archived and publicly available on Zenodo: https://doi.org/10.5281/zenodo.7328505.

A.4 Setup Instructions

We provide different instructions depending on your preferences and available environment/resources. Please first download, uncompress, and `cd` into the artifact directory.

- **Docker Setup:** Build and run the docker image (it should start a bash shell with conda `base` environment activated).

```
$ docker build -t tgopt:artifact .
$ docker run -it --rm
  -v /path/to/artifact:/tgopt tgopt:artifact
```

- **AWS GPU Setup:** For using AWS EC2 with GPU, we provide the `setup-aws-ec2.sh` script under the `scripts/` subdirectory. See the script for the recommended settings and provisioning. Upload the artifact and run the script inside the instance to set it up.
- **Manual Setup:** Most software dependencies can be installed via your system's package manager. For Python, we recommend using conda.

```
$ conda create -n tgopt python=3.7
$ conda activate tgopt
$ pip install -r requirements.txt
$ pip install torch==1.12.0+cu116
  --extra-index-url
  https://download.pytorch.org/whl/cu116
```

See PyTorch's website[3] for CPU-only or other CUDA versions. Next, compile the C++ extension.

```
$ cd extension && make && cd ..
```

When the environment is ready, kick the tires by running a small experiment. The `inference.py` script is the main entry-point for our experiments. Add `--gpu 0` if GPU is available (use `-h` to see all available options).

```
$ python inference.py -d snap-msg --model tgat
  --prefix test --opt-all
```

A.5 Running Experiments

The following instructions walk through reproducing the results and plots in §5.2 of the paper. The artifact writes results to the terminal and generates PDF files (for plots). For NUMA machines, you might need to set a NUMA policy by prepending the `python` commands (or editing the provided scripts) with: `numactl --interleave all`.

Step 1: Inference Performance (§5.2.1). To reproduce results similar to Figure 5 in the paper, use the `run-exp.sh` script. It will run all the datasets using the baseline and then our optimized version, taking an average over 10 runs, and generating a PDF plot file.

```
$ ./scripts/run-exp.sh <cpu | gpu>
```

Step 2: Ablation Study (§5.2.2). To reproduce our ablation study, use the provided `run-ablation.sh` script. This will save runtimes into various csv files and generate a plot of the speedup comparison. When GPU is available, Figure 6 in our paper can be reproduced by running the ablation for both CPU and GPU, and then using the `plot-ablation-both.py` script to generate the plot.

```
$ ./scripts/run-ablation.sh cpu
# If GPU is available:
$ ./scripts/run-ablation.sh gpu
$ python ./scripts/plot-ablation-both.py
  logs/ab-cpu.csv logs/ab-gpu.csv
```

[3] https://pytorch.org/get-started/previous-versions/#v1120

Step 3: Breakdown Analysis (§5.2.3). The cost breakdown in Table 3 can be observed using the `--stats` flag. Note that the output from the artifact uses different naming (e.g. `t_cache_keys` in the output is `ComputeKeys` in the table, `t_attn` is `attention M`, etc). One easy way to get a quick view of the comparison is to run `inference.py`, saving the output to a text file, and using a diff viewer. Additionally, the `--stats` flag also collects hit rates and outputs a csv file under the `logs/` subdirectory. Use the `plot-hit-rate.py` script to generate a figure similar to Figure 7 in our paper.

```
# Run for jodie-lastfm and snap-msg:
$ python inference.py -d <name> --model tgat
  --prefix bd --stats 2>&1
  | tee out-bd-base.txt
$ python inference.py -d <name> --model tgat
  --prefix bd --stats --opt-all 2>&1
  | tee out-bd-opt.txt
# ... use diff viewer ...
$ python ./scripts/plot-hit-rate.py
  jodie-lastfm logs/bd-jodie-lastfm-hits.csv
  snap-msg logs/bd-snap-msg-hits.csv
```

Step 4: Cache Memory Usage (§5.2.4). To acquire the results in Table 4, run the `inference.py` script with the `--stats` flag. This will output more detailed statistics and measurements. Use the `--cache-limit` flag to change the number of stored embeddings. For our paper, we ran using limits of {1e4, 1e5, 1e6, 3e6} for the `jodie-lastfm` and `snap-msg` datasets.

```
$ python inference.py -d <name> --model tgat
  --prefix exp-limit --opt-all --stats
  --cache-limit <limit>
```

Step 5: Cache Storage Analysis (§5.2.5, GPU Only). This section requires having a GPU device and Nvidia's profiling tool called `nvprof`[4]. Note: `nvprof` produces profiling data that can take up significant disk space, so you may want to delete it after each run (it should be located at `/tmp/.nvprof`).

By default, TGOpt will store embeddings on CPU memory (even if GPU device is available). First, collect results using this default setting, then re-compile the C++ extension so it stores embeddings on GPU. Lastly, collect results for this new setting. The results in Table 5 of our paper are generated by looking for the `[CUDA memcpy ...]` statistics in the output files from `nvprof`.

```
$ nvprof --openacc-profiling off
  --unified-memory-profiling off
  --csv --log-file nvprof-cpu-<name>.csv
  python inference.py -d <name> --model tgat
  --prefix store-cpu --opt-all --gpu 0
$ cd extension
$ make clean
$ env tgopt_embed_store_dev=1 make
$ cd ..
$ nvprof --openacc-profiling off
  --unified-memory-profiling off
  --csv --log-file nvprof-gpu-<name>.csv
  python inference.py -d <name> --model tgat
  --prefix store-gpu --opt-all --gpu 0
```

[4] https://docs.nvidia.com/cuda/profiler-users-guide/index.html

A.6 Reuse and Repurposing

Our artifact and the accompanying scripts can be adapted for other use cases. Below are a few suggestions.

Custom Datasets. Instead of using the dataset and models bundled with the artifact, you may prepare the datasets and train the models yourself using the following:

```
$ ./data-download.sh <name>
$ python data-reformat.py -d <name>
$ python data-process.py -d <name>
$ python train.py -d <name> --model tgat [--gpu 0]
```

To use your own dataset, you will need to create the following three files in the `data/` subdirectory (see the TGAT readme[5] for more details):

- `ml_{name}.csv` - List of temporal edges.
- `ml_{name}.npy` - Edge features with shape $(|E| + 1, d_e)$.
- `ml_{name}_node.npy` - Node features, shape $(|V| + 1, d_e)$.

We recommend editing the `data-reformat.py` script to preprocess your dataset into a csv file with the expected format, then the `data-process.py` file can be used unchanged.

Custom Models. You can train TGAT models with different settings, e.g. a model with 3 layers and sampling of 8 neighbors (be sure to use the same `--model` flag in later inference commands):

```
$ python train.py -d <name> --model my-model
  --n-layer 3 --n-degree 8 [--gpu 0]
```

Custom Experiments. The `inference.py` script offers various command-line flags, some of which control the optimizations offered by TGOpt. You can use these flags, along with editing our other provided scripts, to try different experimental settings.

Customizing TGOpt Extension. Finally, the TGOpt C++ extension has several compilation options to customize its behavior, mostly via environment variables and related to parallelization. In the `extension/` subdirectory, see the `Makefile` and `setup.py` for the supported env vars. For example, to force all the TGOpt operations to be single-threaded, try:

```
$ env tgopt_force_single=1 make
```

[5] https://github.com/StatsDLMathsRecomSys/Inductive-representation-learning-on-temporal-graphs#use-your-own-data