# Caerus: A Scalable Caching-based Framework for Temporal Graph Neural Networks


## ABSTRACT

Representation learning over dynamic graphs is critical for many real-world applications such as social network services and recommender systems. Temporal graph neural networks (T-GNNs) are powerful representation learning methods and have achieved remarkable effectiveness on continuous-time dynamic graphs. However, T-GNNs still suffer from high time complexity, which increases linearly with the number of timestamps and grows exponentially with the model depth, causing them not scalable to large dynamic graphs. To address the limitations, we propose Caerus, a novel framework that accelerates T-GNN training by non-trivally caching and reusing intermediate embeddings. We design a brand new and optimal cache replacement algorithm, called MRD, under a practical cache limit. MRD not only improves the efficiency of training T-GNNs by maximizing the number of cache hits but also reduces the approximation errors by avoiding keeping and reusing extremely stale embeddings. Meanwhile, we develop profound theoretical analysis on the approximation error introduced by our reuse schemes and offer rigorous convergence guarantees. Extensive experiments have validated that Caerus can obtain two orders of magnitude speedup over the state-of-the-art baselines while achieving higher precision on large dynamic graphs.

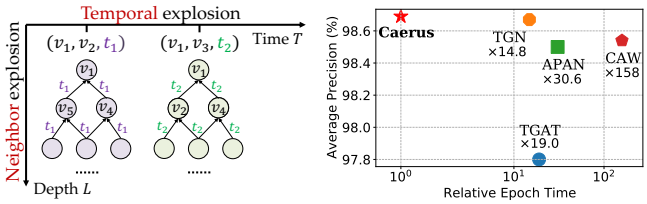


**ACM Reference Format:**

. 2023. Caerus: A Scalable Caching-based Framework for Temporal Graph Neural Networks. In *Proceedings of Proceedings of the 2023 International Conference on Management of Data (SIGMOD '23).* ACM, New York, NY, USA, 19 pages. https://doi.org/XXXXXXX.XXXXXXX


## 1 INTRODUCTION

Dynamic graphs serve as cornerstones in a broad spectrum of domains, including social network services, citation network analysis, e-commerce, and recommender systems. Real-world dynamic graphs can be generally modeled as temporal interactions between nodes. For instance, in social networks like Reddit, users interact with each other by leaving a comment or becoming a follower; in an e-commerce platform, customers interact with items by making a purchase. The above applications require learning representation on evolving graphs, where nodes and edges are dynamically updated, to support downstream tasks such as temporal link prediction or recommendation. However, the majority of previous methods [25, 28, 47, 74] assume that the input graph is *static*. Although it would be possible to apply static graph methods to dynamic graphs by ignoring the evolutionary patterns of dynamic structures, such solutions have been shown to be suboptimal [50, 71].




(a) (b)

**Figure 1: (a) An illustration of the computational cost of T-GNNs, which increases linearly with the number of timestamps (temporal explosion) and grows exponentially with the model depth (neighbor explosion). (b) Comparison of Caerus with the state-of-the-art baselines on Reddit dataset.**

Temporal graph neural networks (T-GNNs) [35, 50, 65, 66, 71] are state-of-the-art methods for learning representations on dynamic graphs. In general, T-GNNs mimic the message passing of static GNNs [28, 62] by *temporal sampling* and *recursive neighborhood aggregation*. Specifically, TGAT [71] encodes continuous time using random Fourier features and applies attention modules for neighborhood aggregation; TGN [50] is a generic framework that includes previous works [35, 59, 71] as special cases; CAW [66] further improves the inductive performance through anonymized random walks. It has been shown in [50, 66] that T-GNNs can outperform the static graph methods [25, 28, 34, 47, 62] and snapshot-based models [22, 26, 46] by a large margin in average precision.

Unfortunately, the high time complexity of T-GNNs hinders their applicability to large dynamic graphs. For instance, it takes over 12 hours for CAW [50] to finish one epoch of training on a large dynamic graph (7.8 million interactions) with an NVIDIA RTX 2080 Ti GPU; T-GNNs including JODIE [35] and TGN [50] cannot scale to large graphs due to high memory usage. Consequently, existing methods are typically evaluated on small dynamic graphs.

The computational bottlenecks of T-GNNs lie in *temporal explosion* and *neighbor explosion*. As shown in Figure 1a, given a batch of two edges $(v_1, v_2, t_1)$ and $(v_1, v_3, t_2)$, T-GNNs generally compute temporal embeddings of target nodes at $t_1$ and $t_2$ with two *time-dependent* computation graphs. On the one hand, unlike static GNNs where the representation of a node is computed only once, T-GNNs need to calculate the *intermediate embeddings* of the same node at *all the involved timestamps* in a data batch. For instance, the embeddings of node $v_4$ at an intermediate layer are computed twice. We refer to this problem as *temporal explosion*. On the other hand, T-GNNs suffer from *neighbor explosion*, which means that the computation graph of T-GNNs grows exponentially with the model depth due to recursive neighborhood aggregation [50]. The above two factors make it challenging to scale T-GNNs to large graphs.




*SIGMOD '23, June 18 - June 23, 2023, Washington, USA*





ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

To solve the above challenges, a straightforward approach would be to cache the embeddings at each intermediate layer and accelerate model training by judiciously reusing these cached results. Take Figure 1a as an example. After executing the computation graph at $t_1$, we cache the intermediate embedding of node $v_4$ on GPU. Then at $t_2$, we avoid recursive neighborhood aggregation for the node $v_4$ by reusing its cached embedding. When new data batches come, we can also reuse historical embeddings computed in previous batches. In this way, we effectively reduce the computational cost of T-GNNs and thus accelerate model training. However, such a caching and reuse scheme could have the following shortcomings.

- **Limited GPU memory.** Ideally, we can maximize the training efficiency by caching and reusing all the intermediates. However, this is impractical for large dynamic graphs due to limited GPU memory. For instance, it takes 32 GB of memory to cache all the intermediates for a 3-layer T-GNN with embedding size 1024 on a 4M-node graph. Besides, edge features, model parameters, and gradients consume extra space. In contrast, a high-end GeForce RTX 3090 Ti GPU is equipped with only 24 GB of memory.
- **Performance drop.** While prior works on static GNNs [18,45] simply cache all the intermediates, this practice is unfavorable for large dynamic graphs. Since model parameters and the graph structure evolve over time, cached embeddings would get stale. Reusing an incredibly stale embedding could introduce huge approximation errors and thus hurt model performance badly. The staleness problem can be especially serious on real-world dynamic graphs where the average frequency of each node's embedding being updated is quite low.
- **Gradient explosion.** The computation graphs of vanilla T-GNNs at different timestamps are independent. However, sharing intermediate embeddings across multiple timestamps further complicates the computation graph. It could lead to exploding gradients when updating model parameters in back-propagation, thus severely hurting model accuracy. According to our experiments, exploding gradients can degrade the accuracy of a 2-layer T-GNN by 4% on the Wikipedia dataset [4].

In this work, we propose Caerus, a scalable **Ca**ching-bas**e**d f**r**amework for temporal graph ne**u**ral network**s**. Caerus solves the *temporal explosion* and *neighbor explosion* problems by sharing *newly computed* intermediates across multiple timestamps in a data batch and by reusing *previously cached* historical embeddings, respectively. The technical challenge is how to manage caches such that we can effectively reduce the recomputation cost without compromising model accuracy. We realize that the classical cache replacement policies like LRU [14] tend to cache all the newly computed embeddings even if they may not be reused in the near future. Thus, the caching policies could suffer from low cache hit ratios and the stale embeddings that would cause large approximation errors. To address the problems, we have introduced the novel concept *reuse distance* for intermediate embeddings by analyzing the distribution of subsequent training batches. We then develop a non-trival caching policy MRD that can maximize the number of cache hits for offline T-GNN training and implicitly filter out extremely stale embeddings. We further theoretically prove that our proposed reuse schemes have a bounded approximation error on the model outputs. Moreover, Caerus addresses the exploding gradient issue with an effective *gradient blocking* strategy. Although our gradient blocking approach and reuse schemes introduce approximated gradient values, we provide rigorous *convergence guarantees* for our proposed techniques.

In summary, we have made the following contributions:

- We present Caerus, a framework that accelerates T-GNN training by caching and reusing intermediates. Caerus reduces the time complexity of T-GNNs from $O(|E| \cdot k^L)$ to $O(|E| \cdot kL)$, where $|E|$ is the batch size, $L$ is the model depth, and $k$ is the neighbor size.
- We formulate the cache replacement problem for training T-GNNs under practical cache limits and propose an optimal caching policy MRD. It not only improves the training efficiency by maximizing the number of cache hits but also boosts model performance by resolving the staleness problem.
- We propose a novel gradient blocking strategy to avoid exploding gradients caused by reusing intermediates, thus stabilizing model update. Moreover, we have theoretically investigated that T-GNNs trained with the approximated gradients can converge to a local optimum as vanilla T-GNNs.
- We develop theoretical analysis on the approximation errors of model outputs. The theory indicates that the errors of approximated temporal embeddings can be bounded as long as the cached embeddings do not get too stale.
- We conduct extensive experiments to validate the efficiency and effectiveness of Caerus. Specifically, Caerus can be two orders of magnitude faster than the state-of-the-art baselines while achieving higher precision and having no noticeable impact on the convergence rate. Besides, Caerus demonstrates robust performance on various graph distributions.

The remainder of this paper is organized as follows. We introduce the technical background in Section 2. Then, we present our Caerus framework in Section 3 and study Caerus under cache limits in Section 4. We develop approximation and convergence guarantees in Section 5. Finally, we conduct experiments in Section 6, review the related work in Section 7, and conclude this paper in Section 8.

## 2 BACKGROUND

In this section, we introduce the background of representation learning on dynamic graphs and a generic T-GNN architecture. Table 1 summarizes the major notations used throughout this paper.

### 2.1 Dynamic Graphs

In this work, we focus on *continuous-time* dynamic graphs (CTDGs) due to their generality and flexibility. CTDGs can come with edge features [35] or simply be non-attributed [1, 2]. Without loss of generality, we define the CTDG as follows.

Definition 1 (Continuous-Time Dynamic Graph (CTDG)). *A CTDG is represented as a sequence of interaction events* $G = \{\alpha(t_1), \alpha(t_2), \dots\}$ *arranged in increasing order of time. Each event is a tuple* $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$ *representing a (directed) temporal edge, where* $v_i$ *and* $v_j$ *are nodes,* $e_{ij}(t)$ *is an edge feature vector, and* $t \in \mathbb{N}^+$ *is the timestamp at which the interaction happens.*

In social networks, for instance, a user $v_i$ can follow, unfollow, or comment on another user $v_j$ at timestamp $t$. Here, different behaviors are represented by different edge feature vectors $e_{ij}(t)$.

## 2.2 Embedding Learning on Dynamic Graphs

Embedding learning (also known as representation learning) problems on dynamic graphs typically follow an *encoder-decoder* framework [27]. Given an interaction event $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$, an encoder function $f : \alpha(t) \rightarrow h_i(t), h_j(t)$ maps the event features to node embeddings, where $h_i(t) \in \mathbb{R}^d$ and $h_j(t) \in \mathbb{R}^d$ are temporal embeddings of node $v_i$ and $v_j$ at timestamp $t$, respectively. A decoder function (e.g., MLP [49]) then takes these temporal embeddings as input and makes prediction for downstream tasks such as link prediction [50, 65, 66] or dynamic node classification [35, 65].

## 2.3 Temporal Graph Neural Networks

Temporal graph neural networks (T-GNNs) are powerful models for learning temporal embeddings on CTDGs. Existing T-GNNs [50, 59, 71] follow a *generic* architecture and usually maintain a state vector $s_i(t)$ for each node $v_i$. The state vector $s_i(t)$ evolves with time and summarizes all the interactions that involve node $v_i$ until timestamp $t$. Given a new interaction $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$, an $L$-layer T-GNN works in the following two sequential steps, i.e., *temporal message passing* and *state update*.

**Temporal Message Passing.** In this step, T-GNN computes the temporal embedding $h_i^L(t)$ of node $v_i$ by *recursively aggregating* its $L$-hop neighborhood information as follows:

$$N_i^l(t) = \text{SAMPLE}(G, v_i, t), \tag{1}$$

$$g_i^l(t) = \text{AGGREGATE}(\{(h_j^{l-1}(t), e_{ij}(\tau), \phi(t-\tau)) | (j, \tau) \in N_i^l(t)\}), \forall l = 1, \ldots, L, \tag{2}$$

$$h_i^l(t) = \text{COMBINE}(h_i^{l-1}(t), g_i^l(t)), \forall l = 1, \ldots, L, \tag{3}$$

$$h_i^0(t) = s_i(t^-). \tag{4}$$

At each layer $l$, T-GNN first samples from graph $G$ a set of temporal neighbors $N_i^l(t)$, where each $(j, \tau) \in N_i^l(t)$ indicates that node $v_j$ interacted with node $v_i$ at timestamp $\tau$ ($\tau < t$) (Eq.1). Then, it aggregates the sampled neighborhood information (e.g., edge features and time encoding) of node $v_i$ (Eq.2). Here, the function $\phi(\cdot)$ encodes the delta time between the current interaction $\alpha(t)$ and a sampled interaction $\alpha(\tau)$. Thus, the computation graphs of T-GNNs are time-dependent. A special case is that T-GNN aggregates the latest state vectors before $t$ when computing the first layer embeddings (Eq.4). Finally, T-GNN computes the $l$-th layer output $h_i^l(t)$ by combing the previous layer output $h_i^{l-1}(t)$ and the aggregated neighborhood information $g_i^l(t)$ (Eq.3).

**State Update.** After receiving a new event $\alpha(t) = (v_i, v_j, e_{ij}(t), t)$, T-GNN updates the state vectors of nodes $v_i$ and $v_j$. Specifically, the state vector of node $v_i$ is updated as follows:

$$s_i(t) = \text{UPDATE}(s_i(t^-), s_j(t^-), e_{ij}(t), t), \tag{5}$$

where $s_i(t^-)$ and $s_j(t^-)$ are the latest state vectors before time $t$.

## 2.4 Training T-GNNs

T-GNNs are trained using batches of interactions. Given a new batch of interactions $E$, we first compute temporal embeddings of nodes appeared in these edges through *forward propagation*. Then, we use supervision signals from a downstream task to update model parameters via *backward propagation*. We illustrate the training procedure of T-GNNs as follows.

**Table 1: Summary of major notations.**

| Notation | Description |
|---|---|
| $G$ | a continuous-time dynamic graph |
| $\alpha(t)$ | an interaction that occurs at timestamp $t$ |
| $v_i$ | a node in the dynamic graph |
| $e_{ij}(t)$ | feature of the edge occurring between $v_i$ and $v_j$ at $t$ |
| $E$ | a batch of interactions events |
| $B$ | a set of target nodes that appear in $E$ |
| $L$ | model depth |
| $k$ | limit on the sampled neighbor size |
| $M^l$ | cache at the $l$-th layer |
| $m$ | cache size limit at each layer |
| $N_i^l(t)$ | sampled neighbors of node $v_i$ at layer $l$ and time $t$ |
| $g_i^l(t)$ | aggregated neighborhood of node $v_i$ at layer $l$ and time $t$ |
| $h_i^l(t)$ | exact embedding of node $v_i$ at layer $l$ and time $t$ |
| $\tilde{h}_i^l(t)$ | approximated embedding of $v_i$ at layer $l$ and time $t$ |
| $\bar{h}_i^l$ | cached embedding of node $v_i$ at layer $l$ ($\bar{h}_i^l \in M^l$) |
| $s_i(t)$ | the state vector of node $v_i$ at time $t$ |

*Example 1.* Figure 2a shows an example of training a T-GNN for a link prediction task. There comes a new mini-batch of two interactions $\alpha(t_5) = (v_1, v_2, e_{12}(t_5), t_5)$ and $\alpha(t_6) = (v_1, v_3, e_{13}(t_6), t_6)$. In forward propagation, the T-GNN model computes temporal embeddings of the target nodes $\{v_1, v_2, v_3\}$ at $t_5$ and $t_6$ with two *time-dependent* computation graphs, where each edge is associated with a *time encoding*. The T-GNN model incorporates the new coming event $\alpha(t_5)$ when calculating the embedding $h_1^3(t_6)$ since $\alpha(t_5)$ happens before time $t_6$, while the calculation of $h_1^3(t_5)$ is purely based on historical interactions before time $t_5$. The temporal embeddings $h_1^3(t_5)$ and $h_2^3(t_5)$ are then fed into a decoder module to predict the existence of the true links and negatively sampled false links. Finally, the parameters of T-GNN are updated by stochastic gradient descent (SGD) [52] based on a predefined loss function.

Note that in the above example, node $v_4$ is sampled as a neighbor node at different timestamps. Then, T-GNN need to calculate its intermediate embeddings in two independent computation graphs. Compared to static GNNs, the computational cost of T-GNNs scales not only with the number of layer $L$ but also with the number of timestamps. Hence, T-GNNs are much more computationally expensive, and it is difficult to scale them to large dynamic graphs.

**Time Complexity.** Let $k$ be the maximum number of neighbors sampled for each target node in Eq.1. Then, given an $L$-layer T-GNN, there are $O(k^{L-l})$ ($1 \leq l \leq L$) nodes at the $l$-th layer in the computation graph. To calculate the representation of each node, a T-GNN needs to aggregate the neighborhood information of at most $k$ neighbors. Therefore, the time complexity of T-GNNs is $O(|E| \cdot \sum_{l=1}^{L} k \cdot k^{L-l}) = O(|E| \cdot k^L)$, where $|E|$ is the batch size.

# 3 THE CAERUS FRAMEWORK

As discussed in Section 2.4, the training cost of T-GNNs increases linearly with the number of unique timestamps (*temporal explosion*) and grows exponentially with the model depth $L$ (*neighbor explosion*). To address the computational bottlenecks, we propose Caerus, a scalable **Ca**ching-bas**e**d f**r**amework for temporal graph ne**u**ral network**s**. Caerus accelerates T-GNN training by caching

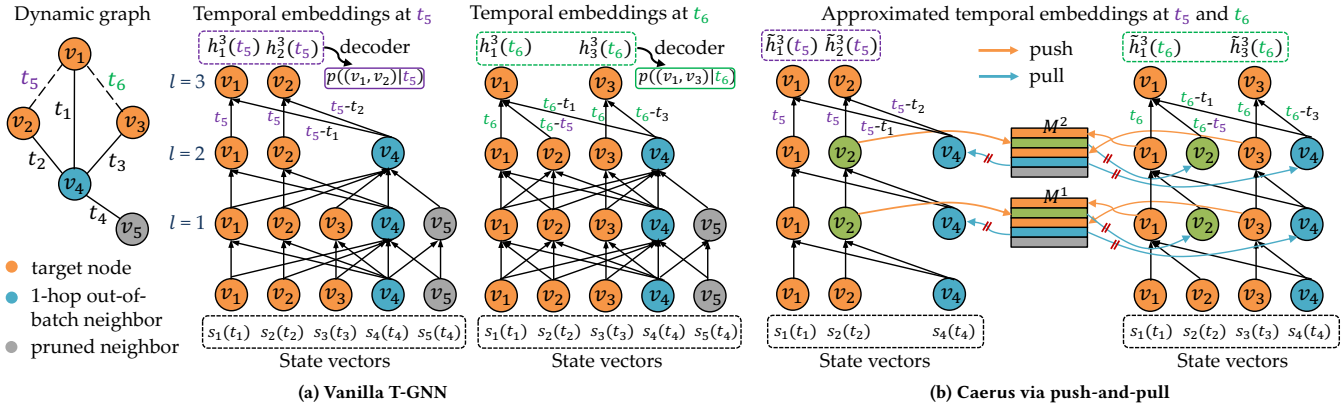



**Figure 2: (a) Example of training a 3-layer T-GNN for a link prediction task. In the dynamic graph, the dotted lines with timestamps $t_5$ and $t_6$ ($t_5 < t_6$) represent a new batch of two edge interactions, while the solid lines with timestamps indicate historical interactions. To predict the existence of these two links, a T-GNN computes temporal embeddings of target nodes at $t_5$ and $t_6$ with two independent computation graphs, where each edge is associated with a time encoding. The temporal embeddings are then decoded to get link probabilities $p((v_1, v_2)|t_5)$ and $p((v_1, v_3)|t_6)$. Based on the predicted probabilities and the labels of the newly arrived edges, parameters of the T-GNN are updated via stochastic gradient descent (SGD). (b) Caerus accelerates model training by reusing cached embeddings. After obtaining intermediate results at the $l$-th layer ($1 \le l < L$), Caerus first pushes the embeddings of target nodes $\{v_1, v_2, v_3\}$ to the cache $M^l$ and then pulls embeddings of the neighbor nodes $v_2$ and $v_4$ from the cache $M^l$ to calculate the $(l+1)$-th layer output. Particularly, the green nodes indicate that the newly computed intermediates of the *in-batch neighbor node* $v_2$ is shared across multiple timestamps, while the blue nodes suggest that the previously cached historical embeddings of the *out-of-batch neighbor node* $v_4$ can be reused in the current batch. In back-propagation, parameters of the T-GNN are updated via SGD. Particularly, as shown by the red slash symbols, gradients will not propagate through the reused embeddings in order to avoid the gradient explosion issue.**

and reusing intermediate embeddings in *forward propagation* (Section 3.1). Such an aggressive reuse scheme may cause severe accuracy drop due to the gradient explosion issue in *backward propagation*. Caerus then solves this problem through a novel gradient blocking strategy (Section 3.2).

## 3.1 Acceleration via Reusing Embeddings

The basic idea of Caerus is to aggressively prune the computation graph via a *push-and-pull* strategy. Without loss of generality, we consider an $L$-layer ($L \ge 2$) T-GNN with caches $M^1, \ldots, M^{L-1}$. Here, $M^l$ dynamically maintains a set of intermediate embeddings that are computed at the $l$-th layer of the T-GNN. In this section, we assume that the storage space is big enough to accommodate all the intermediates. We will remove this assumption and study Caerus under practical cache size limit in Section 4.

Given a batch of events $E$, Caerus works as described in Algorithm 1. Specifically, the first layer is treated as a special case, where Caerus aggregates neighborhood features for target nodes $v_i$ and $v_j$ as vanilla T-GNN (line 3). This is because both Caerus and vanilla T-GNN aggregate features based on state vectors without reusing historical embeddings. The difference is that Caerus then pushes the first layer embeddings $h_i^1(t), h_j^1(t)$ to the cache $M^1$ (line 4).

For a subsequent layer $l = 2, \ldots, L$, Caerus generally works in three steps: ① pull cached neighbor embeddings from the cache $M^{l-1}$ (line 8), ② approximate the current layer output $\tilde{h}_i^l(t)$ with pulled embeddings (line 9), and ③ push the newly computed embeddings to the cache $M^l$ ($l \ne L$) (lines 10-11). We elaborate the above three steps as follows.

**Pull.** Given a target node $v_i$ at timestamp $t$, Caerus samples its 1-hop temporal neighbors $N_i^l(t)$ before feature aggregation (Eq.1). For each sampled neighbor node $v_j$, Caerus pulls the corresponding cached embedding $\bar{h}_j^{l-1}$ if there is one in the cache $M^{l-1}$. Otherwise, Caerus uses zero embeddings for *cold-start* [54] nodes whose intermediate representations have not been cached yet. In this way, Caerus prunes the computation graphs by *avoiding recursively computing the embedding for each neighbor node*. Note that the pulled embeddings are either computed in prior training iterations or newly computed in the current batch. Therefore, Caerus not only reuses *historical* embeddings but also shares the *newly computed* embeddings across *all the possible timestamps* in a data batch.

**Approximated Message Passing.** In analogous to the vanilla T-GNN, Caerus conducts approximated message passing based on cached embeddings as follows:

$$\tilde{g}_i^l(t) = \text{AGGREGATE}(\{(\bar{h}_j^{l-1}, e_{ij}(\tau), \phi(t-\tau)) \mid (j,\tau) \in N_i^l(t)\}), \forall l = 2, \ldots, L, \quad (6)$$

$$\tilde{h}_i^l(t) = \text{COMBINE}(\tilde{h}_i^{l-1}(t), \tilde{g}_i^l(t)), \forall l = 2, \ldots, L, \quad (7)$$

$$\tilde{h}_i^1(t) = h_i^1(t). \quad (8)$$

Specifically, Caerus approximates the neighborhood information $\tilde{g}_i^l(t)$ by aggregating the pulled embeddings (Eq.6). Then, it combines the previous layer embedding $\tilde{h}_i^{l-1}(t)$ with $\tilde{g}_i^l(t)$ to get the $l$-th layer output $\tilde{h}_i^l(t)$ (Eq.7).

**Push.** After obtaining the representation $\tilde{h}_i^l(t)$ at the intermediate layer $l$ ($l \ne L$), Caerus pushes it to the cache $M^l$ and *replaces* the

**Algorithm 1:** Caerus via push-and-pull

**input :** Batch of interactions $E$, caches $\{M^l\}_{l=1}^{L-1}$
**ouput :** Approximated temporal embeddings

```
// first layer
for α(t) = (v_i, v_j, e_ij(t), t) ∈ E do
    h_i^1(t), h_j^1(t) ← temporal message passing (Eq.1-Eq.4)
    push h_i^1(t), h_j^1(t) to the cache M^1
// subsequent layers
for l = 2, ..., L do
    for α(t) = (v_i, v_j, e_ij(t), t) ∈ E do
        pull 1-hop neighbor embeddings of v_i, v_j from M^{l-1}
        h̃_i^l(t), h̃_j^l(t) ← approximated temporal message
          passing (Eq.6-Eq.8)
        if l ≠ L then
            push h̃_i^l(t), h̃_j^l(t) to the cache M^l
```

previously cached intermediate of node $v_i$ (lines 10-11). Note that the node $v_i$ may appear in multiple interactions in the current batch $E$. For example, in Figure 2b, node $v_1$ appears as a target node at timestamps $t_5$ and $t_6$ ($t_5 < t_6$). In such a case, only the intermediate embedding $\tilde{h}_1^l(t_6)$ with the latest timestamp $t_6$ will be kept in the cache $M^l$ since the cache maintains at most one intermediate embedding for each node.

**Time Complexity.** The complexity of Algorithm 1 is $O(|E| \cdot kL)$, where $k$ is the limit on sampled neighbor size, and $|E|$ is the batch size. In contrast, the vanilla T-GNN has time complexity $O(|E| \cdot k^L)$. Hence, our Caerus framework is exponentially faster. Furthermore, to avoid the communication cost caused by frequent data transfers between the host (i.e., CPU) and the device (e.g., GPU), we deploy all the caches on the device.

**Reuse Chance Analysis.** The reuse chances of Caerus come from two sources: *out-of-batch neighbor nodes* (e.g., $v_4$ in Figure 2b) and *in-batch neighbor nodes* (e.g., $v_2$ in Figure 2b). For out-of-batch neighbor nodes, Caerus avoids *neighbor explosion* by directly reusing the *historical* embeddings computed in previous iterations. For in-batch neighbor nodes, Caerus avoids *temporal explosion* by sharing the *newly computed* embeddings across multiple timestamps. According to Figure 7, there can be more in-batch neighbors than out-of-batch neighbors on real-world dynamic graphs due to *temporal locality*.

**The Staleness Problem.** Unlike classical caching problems in database systems, reusing previously cached intermediates to speed up T-GNN training will inevitably introduce approximation errors. This is because model parameters and the graph structure keep changing during training. Hence, given a target node, the input features, neighbor set, and model parameters required to compute its intermediate embeddings are different in different training batches. Under the circumstances, previously computed intermediates could get stale with respect to the current batch and thus introduce errors. Intuitively, more stale embeddings, which were computed a long time ago, are more likely to cause performance drop. Hence, the technical challenge here is how to reuse intermediates without compromising model accuracy. To address the staleness problem, we propose a cache replacement algorithm to filter out those extremely stale embeddings in Section 4.3. Moreover, we theoretically investigate how the staleness problem affects model outputs and the convergence rate in Section 5.

## 3.2 Effective Training via Gradient Blocking

Although our push-and-pull scheme (Algorithm 1) reduces the computational cost of T-GNNs in *forward propagation*, it brings new challenges to gradient update in *backward propagation*. To achieve stable and effective model training, the critical problem here is how to manage gradient flows with cached embeddings. We discuss this problem in two cases, i.e., reusing *newly computed* embeddings and reusing *historical* embeddings.

**Gradient Flow with Newly Computed Embeddings.** Consider a newly computed embedding $\bar{h}_i^l \in M^l$ that is shared across multiple timestamps in forward propagation. By the chain rule of differentiation, gradients from multiple links can propagate through it in back-propagation. However, such a gradient flow scheme may cause *exploding gradients* [29], thus hurting model accuracy. For instance, in Figure 2b, the embedding $h_2^1(t_5)$ is used to compute $\tilde{h}_2^2(t_5)$ and $\tilde{h}_1^2(t_6)$. Therefore, in back-propagation, gradients from these two links will accumulate on $h_2^1(t_5)$ and may produce a large gradient value (i.e., exploding gradient). In contrast, for the vanilla T-GNN in Figure 2a, the embedding $h_2^1(t_5)$ will not partipate in the computation at other timestamps. Hence, the exploding gradient issue will not occur on vanilla T-GNNs. According to Figure 5, the gradient explosion issue can be especially serious if embeddings are aggressively shared across many timestamps in a data batch.

**Gradient Flow with Historical Embeddings.** Given a historical embedding $\bar{h}_i^l \in M^l$ that is reused in forward propagation, it is impractical for the gradient reaching $\bar{h}_i^l$ to propagate through it in back-propagation. This is because $\bar{h}_i^l$ was computed in a previous batch and the computation graph for calculating it was already discarded after finishing model training on that batch. Moreover, recursively propagating gradients across multiple training batches can also cause unstable model update.

**Gradient Blocking.** Based on the above discussions, we propose to block all the gradient flows reaching the reused embeddings in back-propagation. Specifically, as shown by the red slash symbols, we do not allow gradients to propagate through the reused embeddings. Hence, gradients that originate from different timestamps will not accumulate anymore. In this way, we can effectively avoid exploding gradients, thus stabilizing model training. Although our blocking approach results in incomplete gradient values, we show later through Theorem 5 that the approximated gradient is asymptotically unbiased. Therefore, T-GNNs trained with such approximated gradients can still converge to a local optimum.

# 4 CAERUS UNDER CACHE SIZE LIMIT

Note that Algorithm 1 assumes that the intermediates of all the nodes in a dynamic graph can be cached. In this section, we scale Caerus to large dynamic graphs with limited cache size. The motivation of restricting the cache size is to fit intermediates in limited GPU memory and mitigate the staleness problem of cached embeddings. We introduce our *reuse-or-recompute* scheme in Section 4.1, formally define the cache replacement problem for T-GNNs in Section 4.2, develop an optimal caching policy called MRD in Section 4.3, and prove the optimality of MRD in Section 4.4.

**Algorithm 2:** Caerus via reuse-or-recompute

**input :** Batch of events $E$, caches $\{M_{T-1}^l\}_{l=1}^{L-1}$, cache size $m$
**ouput:** Approximated temporal embeddings

1 $B_T = \{(v_i, t), (v_j, t) | (v_i, v_j, e_{ij}(t), t) \in E_T\}$ // target nodes
2 $\tilde{H}_{B_T}^L \leftarrow$ Recursion($B_T$,$L$, $m$) // model output
3 **return** $\tilde{H}_{B_T}$

**Algorithm 3:** Recursion

**input :** Target nodes $B$, depth $l$, cache size $m$
**ouput:** Approximated embeddings $\tilde{H}_B^l$

1 **if** $l = 1$ **then**
2  $H_B^l \leftarrow$ temporal message passing (Eq.1-Eq.4)
3 **else**
4  $O \leftarrow$ sampled out-of-batch neighbor nodes of $B$
5  $U \leftarrow$ uncached neighbor nodes
6  $B' \leftarrow B \cup U$
7  $\tilde{H}_{B'}^{l-1} \leftarrow$ Recursion($B'$, $l-1$, $m$)
8  $\tilde{H}_O^{l-1}, M_T^{l-1} \leftarrow$ push_and_pull ($M_{T-1}^{l-1}$, $\tilde{H}_{B'}^{l-1}$, $O$, $m$)
9  $\tilde{H}_B^l \leftarrow$ approximated message passing (Eq.6-Eq.8)
10 **return** $\tilde{H}_B^l$

## 4.1 Reuse-or-Recompute

To compute the embedding of a target node $v$, Caerus adopts a *reuse-or-recompute* scheme under cache limit. The basic idea is to reuse intermediates of $v$'s neighbor nodes if they are cached; otherwise, Caerus recomputes the corresponding representations.

Algorithm 2 depicts our reuse-or-recompute algorithm, which recursively invokes Algorithm 3 to compute the representations at each layer. Specifically, given a set of target nodes $B$ at layer $l$, we first collect the corresponding *1-hop out-of-batch neighbor* nodes $O$ and *uncached 1-hop neighbor* nodes $U \subseteq O$ (lines 4-5 in Algorithm 3). Then, we calculate the $(l-1)$-layer representations of $B' = B \cup U$ by recursion (line 7). We next push the newly computed intermediate embeddings $\tilde{H}_{B'}^{l-1}$ into the cache $M_T^{l-1}$ and pull the cached embeddings $\tilde{H}_O^{l-1}$ of out-of-batch neighbor nodes $O$ (line 8). Meanwhile, the cache is updated with $\tilde{H}_{B'}^{l-1}$ following the predefined cache size limit $m$ (see Section 4.3). Finally, we get the $l$-th layer output by approximated message passing (line 9). Figure 3 gives an example of how our reuse-or-recompute scheme works.

**Time Complexity.** Compared to Algorithm 1, it takes extra time for Algorithm 2 to compute the intermediates of *out-of-batch neighbor nodes* whose historical embeddings are not cached. Let $n$ be the total number of such uncached nodes when training a T-GNN on the batch $E$. Then, the time complexity of Algorithm 2 is $O(|E| \cdot kL + nk)$. We will empirically show in Section 6.2 that the recomputation cost of Algorithm 2 can be marginal since the number of our-of-batch neighbors can be smaller than that of in-batch neighbors.

## 4.2 Cache Replacement Problem

A critical problem in Algorithm 3 is how to dynamically manage the caches so that we can mitigate the staleness problem of cached intermediates and improve the efficiency of *offline* model training by minimizing the amount of recomputation.

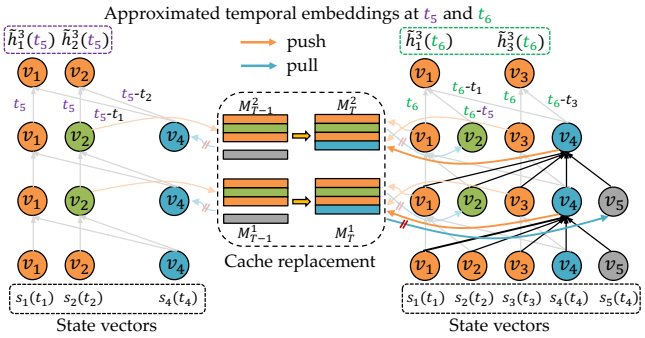



**Figure 3: Caerus via reuse-or-recompute under cache size limit. Consider the link prediction task in Figure 2a and our Caerus with cache size constraint $m = 4$. Assuming the historical embeddings of node $v_4$ are not cached, Caerus needs to recompute its intermediate representations. Compared to the case without cache limit (Figure 2b), the extra computation and data flow are highlighted.**

**Problem Setup.** In offline training, we consider a sequence of batches $E_1, E_2, \ldots, E_T$, where each batch $E_t$ contains a set of interactions. Let $B_t$ be the set of target nodes that appear in $E_t$ and $O_t$ be the set of *1-hop out-of-batch neighbors* sampled for $B_t$. Caerus needs to recompute the representations of uncached nodes $U_t$ ($U_t \subseteq O_t$) and then updates the cache with the *newly computed intermediates* of $B_t \cup U_t$. Given the above preliminaries, we formally define the cache replacement problem as follows.

Definition 2 (Cache Replacement Problem). *Consider a sequence of offline training batches $E_1, \ldots, E_T$. Let $O_t$ be the set of 1-hop out-of-batch neighbors sampled for the target nodes $B_t$. Denote by $M_t$ the set of cached embeddings after finishing model training on the batch $E_t$. We aim to find a cache replacement strategy $f(\cdot)$ that maximizes the number of cache hits under the cache size limit $m$:*

$$\max_f \sum_{t=1}^{T} \sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M_{t-1}\} = \min_f \sum_{t=1}^{T} |U_t|, \tag{9}$$

$$M_t = f(M_{t-1}, H_{B_t} \cup H_{U_t}), \sum_{\bar{h}_i \in M_t} cost(\bar{h}_i) \leq m, \forall t = 1, \ldots, T,$$

*where $H_{B_t}$ and $H_{U_t}$ are the newly computed embeddings of the target nodes $B_t$ and uncached 1-hop neighbor nodes $U_t \subseteq O_t$, respectively.*

Without loss of generality, we drop the superscript of layer $l$ in the above problem definition since we cache intermediate results of *the same set of nodes* at different layers. In real-world applications, models are often updated *offline* and then deployed to production if they pass the A/B test [5, 11, 38, 68]. Our basic idea is to minimize the amount of recomputation by maximizing the total number of cache hits in all the offline training batches. In practice, we consider $m$ as the maximum number of items allowed and set the cost of each cached embedding $\bar{h}_i$ to be $cost(\bar{h}_i) = 1$.

**Hardness.** The cache replacement problem (CRP) is a variant of the general caching problem [6, 8] and the multidimensional knapsack problem [19]. CRP is NP-hard in the *general* form, where different items can have different storage costs. However, in T-GNNs, all the intermediate representations at a specific layer $l$ have the same

**Algorithm 4:** Minimum reuse distance (MRD) algorithm

**input :** Offline training batches $\{E_t\}_{t=1}^{T}$, initial cache $M_0$, cache size $m$
**ouput:** Cache plans $\{M_t\}_{t=1}^{T}$

1 **for** $t = 1, \ldots, T$ **do** `//data preparation`
2 $B_t \leftarrow$ target nodes in the batch $E_t$
3 $N_t \leftarrow$ sampled neighbor nodes of $B_t$ (Eq.1)
4 $O_t \leftarrow N_t \setminus B_t$
5 **for** $t = 1, \ldots, T$ **do** `//generate cache plans`
6 $M' \leftarrow M_{t-1} \cup B_t \cup O_t$ `//candidate set`
7 $M_t \leftarrow$ nodes in $M'$ with the top-$m$ minimum reuse distance (ties are broken arbitrarily)
8 **return** $\{M_t\}_{t=1}^{T}$

dimensionality. Therefore, we consider the *uniform* case and then develop an optimal cache replacement algorithm in Section 4.3.

## 4.3 Minimum Reuse Distance Algorithm

In this section, we develop an *optimal* algorithm for the cache replacement problem. Given an intermediate embedding $h_v$, we define its *maximum active time* and *reuse distance* as follows.

Definition 3 (Maximum Active time). *Consider an intermediate embedding $h_v$ of node $v$ that is created at the data batch $E_i$. Let $E_j$ $(i < j)$ be the latest batch where the node $v$ appears as a target node (i.e., $v \in B_j$). Then, the maximum active time of the intermediate $h_v$ is $\gamma(h_v) = j - 1$.*

As the node $v$ appears in batch $E_j$ as a target node, our Caerus will compute its latest intermediate representations. Hence, the historical embedding $h_v$ computed at the previous batch $E_i$ will never be reused when training a model on the batch $E_j$ and afterward.

Definition 4 (Reuse Distance). *Consider an intermediate embedding $h_v$ of node $v$ after updating model on the batch $E_i$. Let $E_j$ $(i < j)$ be the latest batch where the node $v$ appears as a 1-hop out-of-batch neighbor node (i.e., $v \in O_j$). Then, the reuse distance of $h_v$ right after finishing model training on the batch $E_i$ is*

$$\Delta(h_v, i) = \begin{cases} j - i, j \le \gamma(h_v) \\ \infty, j > \gamma(h_v). \end{cases}$$

Based on the above definition, we propose the minimum reuse distance (MRD) algorithm that can generate optimal cache plans. As shown in Algorithm 4, the basic idea of MRD is to cache intermediates that are most likely to be reused in the near future. Given a sequence of offline training batches, MRD first extracts the target nodes $B_t$ and out-of-batch neighbor nodes $O_t$ (lines 1-4). The sampling results (line 3) will be reused in offline training. Then, in each iteration $t$, MRD selects nodes with the top-$m$ minimum reuse distance from the candidate set $M_{t-1} \cup B_t \cup O_t$ with *ties broken arbitrarily* (lines 5-7). Note that MRD only generates *cache plans* instead of a set of cached embeddings. Based on these plans, we dynamically update caches at *all the intermediate layers* in the offline training stage.

**Time Complexity.** The time complexity of the data preparation step in Algorithm 4 is dominated by the sampling operation (line 3). The sampling cost can be amortized by reusing the sampled results in the offline training stage. For each node in the candidate set, it takes $O(1)$ time to get the reuse distance (line 6). Then, it takes $O(|M'|)$ time to select the top-$m$ nodes (line 7). Therefore, the time complexity of Algorithm 4 is $O(T \cdot (|E| \cdot c + |M|))$, where $|E|$ is the maximum batch size, $c$ is the sampling cost given a target node, and $|M|$ is the maximum candidate size.

**Comparison with Classical Caching Policies.** Note that the classical cache replacement policies like LRU [14] and 2Q [32] are designed for single-point query. In contrast, MRD can make a better cache plan by analyzing the dependencies between different items in a data batch and the distribution of subsequent training batches. On the other hand, LRU tends to cache all the newly computed embeddings even if they may not be reused in the near future. Thus, LRU could fill the cache with stale embeddings of some cold nodes. Instead, by its principle, our MRD policy does not cache intermediates with large reuse distances (i.e., embeddings of cold nodes) in the first place. Therefore, MRD cam implicitly *avoid keeping and reusing extremely stale embeddings* that would introduce large approximation errors. We will empirically compare MRD with classical caching policies in Section 6.4.

## 4.4 Proof of Optimality

We follow the dynamic programming technique [51] to prove that our MRD policy (Algorithm 4) is optimal in maximizing the number of cache hits. Given a cache $M_{t-1}$, let $J_t(M_{t-1})$ be the maximum possible number of cache hits starting with the batch $E_t$. Specifically, $J_t(M_{t-1})$ is recursively defined as

$$J_t(M_{t-1}) = \sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M_{t-1}\} + \max_{M_t \in C_t(M_{t-1})} J_{t+1}(M_t), \quad (10)$$

$$C_t(M_{t-1}) = \{M \subseteq M_{t-1} \cup H_{B_t} \cup H_{U_t} : |M| = m\}, \quad (11)$$

where $C_t(M_{t-1})$ is the set of candidate cache states.

Consider two caches $M_{t-1}$ and $M'_{t-1}$ ($|M_{t-1}| = |M'_{t-1}| = m$), we define the *potential advantage* of $M'_{t-1}$ over $M_{t-1}$ as

$$d_t(M_{t-1}, M'_{t-1}) = \min_g \{\bar{h} \in M_{t-1} | \Delta(\bar{h}, t-1) > \Delta(g(\bar{h}), t-1)\}, \quad (12)$$

where $g : M_{t-1} \to M'_{t-1}$ is a bijection between items in the two cache states $M_{t-1}$ and $M'_{t-1}$. $d_t(M_{t-1}, M'_{t-1})$ compares the reuse distance of all the items in $M_{t-1}$ and $M'_{t-1}$. A larger $d_t(M_{t-1}, M'_{t-1})$ value indicates that items in $M'_{t-1}$ generally have smaller reuse distance and thus may lead to more cache hits than $M_{t-1}$. We next prove the following lemma.

Lemma 2. *$J_t(M'_{t-1}) - J_t(M_{t-1}) \le d_t(M_{t-1}, M'_{t-1})$ for all $1 \le t \le T$ and $|M_{t-1}| = |M'_{t-1}|$.*

Proof. If $t = T$, $J_t(M'_{t-1}) = J_t(M_{t-1}) = 0$ and $d_t(M_{t-1}, M'_{t-1}) = 0$. Therefore, the inequality holds for the special case $t = T$. If $t < T$, we proceed by induction. Assume that $J_{t+1}(M'_t) - J_{t+1}(M_t) \le d_{t+1}(M_t, M'_t)$ for all $|M_t| = |M'_t|$. Let $M'_t \in C_t(M'_{t-1})$ be chosen by an optimal strategy and $M_t \in C_t(M_{t-1})$ be chosen by our MRD algorithm, respectively. We study the relationship between $d_{t+1}(M_t, M'_t)$ and $d_t(M_{t-1}, M'_{t-1})$ by analyzing whether the historical embedding $\bar{h}_i$ of each node $v_i \in O_t$ is cached.

- $\bar{h}_i \in M_{t-1}$ and $\bar{h}_i \in M'_{t-1}$. After reusing $\bar{h}_i$, we can either keep $\bar{h}_i$ or evict it. Since MRD evicts items with the largest reuse distance, then $d_{t+1}(M_t, M'_t) \le d_t(M_{t-1}, M'_{t-1})$.

- $\bar{h}_i \notin M_{t-1}$ and $\bar{h}_i \notin M'_{t-1}$. After recomputing $h_i$, we can either cache $h_i$ or discard it. Since MRD evicts items with the largest reuse distance, then $d_{t+1}(M_t, M'_t) \leq d_t(M_{t-1}, M'_{t-1})$.
- $\bar{h}_i \in M_{t-1}$ and $\bar{h}_i \notin M'_{t-1}$. Before updating model with the batch $E_t$, $\bar{h}_i$ contributes to the advantage of $M_{t-1}$ by 1 in Eq.12. After that, $M_{t-1}$ loses its relative advantage by 1. Then, $d_{t+1}(M_t, M'_t) \leq d_t(M_{t-1}, M'_{t-1}) + \sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M_{t-1} \text{ and } \bar{h}_i \notin M'_{t-1}\}$.
- $\bar{h}_i \notin M_{t-1}$ and $\bar{h}_i \in M'_{t-1}$. Before updating model with the batch $E_t$, $\bar{h}_i$ contributes to the advantage of $M'_{t-1}$ by 1 in Eq.12. After that, $M'_{t-1}$ loses its relative advantage by 1. Then, $d_{t+1}(M_t, M'_t) \leq d_t(M_{t-1}, M'_{t-1}) - \sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \notin M_{t-1} \text{ and } \bar{h}_i \in M'_{t-1}\}$.

Summarize all the above four cases, we get

$$\begin{aligned} &d_{t+1}(M_t, M'_t) + \textstyle\sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M'_{t-1}\} \\ \leq &d_t(M_{t-1}, M'_{t-1}) + \textstyle\sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M_{t-1}\}. \end{aligned} \tag{13}$$

Based on the above discussion, we can obtain

$$\begin{aligned} J_t(M'_{t-1}) &= \textstyle\sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M'_{t-1}\} + J_{t+1}(M'_t) \\ &\leq \textstyle\sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M'_{t-1}\} + J_{t+1}(M_t) + d_{t+1}(M_t, M'_t) \\ &\leq \textstyle\sum_{v_i \in O_t} \mathbf{1}\{\bar{h}_i \in M_{t-1}\} + J_{t+1}(M_t) + d_t(M_{t-1}, M'_{t-1}) \\ &\leq J_t(M_{t-1}) + d_t(M_{t-1}, M'_{t-1}), \end{aligned} \tag{14}$$

where the first inequality follows the inductive assumption, the second inequality follows Eq.13, and the last inequality follows the definition of $J_t(M_{t-1})$. The proof of Lemma 2 is completed. □

Theorem 3. *The MRD algorithm is optimal in maximizing the number of cache hits for the cache replacement problem (Def. 2).*

Proof. Consider a cache $M_{t-1}$. Let $M_t$ and $M'_t$ be the successive cache states selected by our MRD algorithm and the optimal algorithm, respectively. Since $M_t$ minimizes $d_{t+1}(M_t, M'_t)$ over the candidate set $C_t(M_{t-1})$, then $d_{t+1}(M_t, M'_t) = 0$. Finally, we get

$$0 \leq J_{t+1}(M'_t) - J_{t+1}(M_t) \leq d_{t+1}(M_t, M'_t) = 0, \tag{15}$$

where the first inequality follows the optimality of $M'_t$ and the second inequality follows Lemma 2. Then, Eq.15 indicates that our MRD algorithm is optimal. □

## 5 THEORETICAL ANALYSIS

In this section, we develop theoretical analysis on the approximation error introduced by our reuse schemes in Section 5.1, and offer convergence guarantees for our Caerus framework in Section 5.2.

### 5.1 Approximation Guarantee

We propose a *unified* theoretical framework for bounding the approximation error introduced by *cache-unlimited* Caerus (Algorithm 1) and *cache-limited* Caerus (Algorithm 2). Specifically, given an $L$-layer T-GNN and a target node $v_i$ at timestamp $t$, we aim to bound the approximation error between the exact embedding $h_i^L(t)$ and the approximated embedding $\tilde{h}_i^L(t)$.

**Layer-wise Approximation Error.** Let $f_1(\cdot) = \text{COMBINE}(\cdot)$ and $f_2(\cdot) = \text{AGGREGATE}(\cdot)$ be $\lambda_1$ and $\lambda_2$ Lipschitz continuous, i.e., $\|f_1(x_1) - f_1(x_2)\| \leq \lambda_1 \|x_1 - x_2\|$ and $\|f_2(y_1) - f_2(y_2)\| \leq \lambda_2 \|y_1 - y_2\|$. The Lipschitz properties are widely used in model analysis. We refer readers to [7, 18, 20, 63] for the Lipschitz constants of various combination and aggregation functions used in GNNs. Based on the Lipschitz property, we have the following upper bound on the $l$-th layer approximation error:

$$\begin{aligned} \|h_i^l(t) - \tilde{h}_i^l(t)\| =& \|f_1(h_i^{l-1}(t), g_i^l(t)) - f_1(\tilde{h}_i^{l-1}(t), \tilde{g}_i^l(t))\| \\ \leq& \lambda_1(\|h_i^{l-1}(t) - \tilde{h}_i^{l-1}(t)\| + \|g_i^l(t) - \tilde{g}_i^l(t)\|). \end{aligned} \tag{16}$$

Moreover, since the aggregation function $f_2(\cdot)$ is $\lambda_2$-Lipschitz continuous, we can obtain the following upper bound on the approximation error of the aggregated neighborhood feature:

$$\begin{aligned} \|g_i^l(t) - \tilde{g}_i^l(t)\| \leq& \lambda_2 \textstyle\sum_{(j,\tau) \in R_i^l(t)} \|h_j^{l-1}(\tau) - \bar{h}_j^{l-1}\| \\ &+ \lambda_2 \textstyle\sum_{(j,\tau) \in N_i^l(t) \setminus R_i^l(t)} \|h_j^{l-1}(\tau) - \tilde{h}_j^{l-1}(t)\|, \end{aligned} \tag{17}$$

where $N_i^l(t)$ is a set of sampled temporal neighbors, $R_i^l(t) \in N_i^l(t)$ denotes the neighbors whose intermediate embeddings are cached and reused, and $N_i^l(t) \setminus R_i^l(t)$ refers to the nodes whose embeddings are newly computed.

For any node $v_i$ at timestamp $t$, assume that the error of the approximated embedding $\tilde{h}_i^{l-1}(t)$ and the staleness of the cached embedding $\bar{h}_i^{l-1}$ are bounded, i.e., $\|h_i^{l-1}(t) - \tilde{h}_i^{l-1}(t)\| \leq \delta^{(l-1)}$ and $\|h_i^{l-1}(t) - \bar{h}_i^{l-1}\| \leq \epsilon^{(l-1)}$. Let $k$ be the maximum number of neighbors sampled in Eq.1. Plug Eq.17 into Eq.16, then

$$\begin{aligned} \|h_i^l(t) - \tilde{h}_i^l(t)\| &\leq \lambda_1(\|h_i^{l-1}(t) - \tilde{h}_i^{l-1}(t)\| + \|g_i^{l-1}(t) - \tilde{g}_i^{l-1}(t)\|) \\ &\leq \lambda_1(\delta^{(l-1)} + k\lambda_2(\delta^{(l-1)} + \epsilon^{(l-1)})) \\ &= \lambda_1(1 + k\lambda_2)\delta^{(l-1)} + k\lambda_1\lambda_2\epsilon^{(l-1)} = \delta^{(l)}. \end{aligned} \tag{18}$$

The above recursion formula indicates that the $l$-th layer approximation error $\delta^{(l)}$ is composed of the $(l-1)$-layer approximation error $\delta^{(l-1)}$ and the staleness $\epsilon^{(l-1)}$ of cached embeddings. There is one special case. The 0-layer embedding $h_i^0(t)$ corresponds to the state vector $z_i(t^-)$ of node $v_i$, i.e., $h_i^0(t) = z_i(t^-)$. On the other hand, the stale embedding $\bar{h}_i^0(t)$ and the approximated embedding $\tilde{h}_i^0(t)$ correspond to the stale state vector $\bar{z}_i(t^-)$, i.e., $\bar{h}_i^0(t) = \tilde{h}_i^0(t) = \bar{z}_i(t^-)$. Hence, the input error can be bounded by

$$\|h_i^0(t) - \tilde{h}_i^0(t)\| = \|h_i^0(t) - \bar{h}_i^0(t)\| = \|z_i(t^-) - \bar{z}_i(t^-)\| \leq \epsilon^{(0)}. \tag{19}$$

Since Caerus updates the state vectors of all the target nodes as the vanilla T-GNN, there is no staleness in node states, i.e., $\epsilon^{(0)} = 0$.

**The Upper Bound.** Based on the recursion formula Eq.18, we stack all the layers together and develop the following theorem, which holds with and without cache limit.

Theorem 4. *Consider an $L$-layer T-GNN with COMBINE$(\cdot)$ and AGGREGATE$(\cdot)$ functions that are $\lambda_1$ and $\lambda_2$ continuous, respectively. Assume that for any node $v_i$ at timestamp $t$, its cached embedding $\bar{h}_i^{l-1}$ has bounded staleness, i.e., $\|h_i^{l-1}(t) - \bar{h}_i^{l-1}\| \leq \epsilon^{(l-1)}$. Then, the output error of the approximated temporal embedding $\tilde{h}_i^L(t)$ is bounded by*

$$\|h_i^L(t) - \tilde{h}_i^L(t)\| \leq \textstyle\sum_{l=1}^{L-1} k(1 + k\lambda_2)^{L-l-1}\lambda_1^{L-l}\lambda_2\epsilon^{(l)}, \tag{20}$$

*where $k$ is the limit on the sampled neighbor size in Eq.1.*

Proof. Eq.20 can be obtained by unrolling the recursive approximation error according to Eq.18 and Eq.19. □

**Implication.** As have discussed in Section 3.1 and Section 4.3, since model parameters and the graph structure keep changing, intermediates computed in previous training iterations may become out-of-date and introduce huge approximation errors. Theorem 4 suggests that more stale embeddings are more likely to hurt model accuracy. The staleness problem can be especially serious on large dynamic graphs where the average frequency of each node being updated is quite low. Fortunately, our cache replacement policy MRD (Algorithm 4) can implicitly prevent the cache from keeping extremely stale embeddings that would hurt model accuracy. Therefore, cache-limited Caerus, equipped with our MRD algorithm, is supposed to achieve better predictive performance than cache-unlimited Caerus on large dynamic graphs.

## 5.2 Convergence Guarantee

In this section, we analyze how our Caerus framework affects the convergence rate of model training. T-GNNs are updated using batches of interactions. Given a new data batch $B_t$, the parameters of a vanilla T-GNN are updated through $W_t = W_{t-1} - \eta \nabla \mathcal{L}(W_{t-1})$, where $\mathcal{L}(\cdot)$ is a loss function, $\nabla \mathcal{L}(W_{t-1})$ is the gradient obtained on the batch $B_t$ with parameters $W_{t-1}$, and $\eta$ is the learning rate. Similarly, Caerus updates model parameters through $W_t = W_{t-1} - \eta \nabla \tilde{\mathcal{L}}(W_{t-1})$, where $\nabla \tilde{\mathcal{L}}(W_{t-1})$ denotes the approximated gradient caused by our gradient blocking and reuse schemes. Given the above preliminaries, we work out the convergence guarantee for our Caerus framework as follows.

Theorem 5. *Consider a T-GNN updated by Caerus through gradient descent $W_t = W_{t-1} - \eta \nabla \tilde{\mathcal{L}}(W_{t-1})$. Assume that (1) the loss function $\mathcal{L}(\cdot)$ is $\rho$-smooth, (2) the activation function in T-GNN is Lipschitz continuous, and (3) the gradients of model parameters have bounded Frobenius norm. Then, after training the T-GNN for $T$ iterations, we have*

$$\frac{1}{T} \sum_{t=1}^{T} \mathbb{E} \|\nabla \mathcal{L}(W_{t-1})\|_F^2 = O(1/\sqrt{T}) \tag{21}$$

*by setting the learning rate $\eta = \min(\frac{1}{\rho}, \frac{1}{\sqrt{T}})$.*

The above theorem demonstrates that a T-GNN trained through our Caerus framework can converge to a local optimum as the vanilla T-GNN does with convergence rate $O(1/\sqrt{T})$. Specifically, Caerus introduces the approximated gradient $\nabla \tilde{\mathcal{L}}$ while vanilla T-GNN is updated via the exact gradient $\nabla \mathcal{L}$. Theorem 5 shows that a T-GNN trained with the approximated gradient $\nabla \tilde{\mathcal{L}}$ can evetually achieve a small exact gradient $\nabla \mathcal{L}$, i.e., the gradient that is obtained without reusing intermediates or gradient blocking. This theoretical finding indicates that the approximated gradient $\nabla \tilde{\mathcal{L}}$ computed by Caerus is asymptotically unbiased as the number of training iterations $T \to \infty$. Note that the assumptions in Theorem 5 are common and widely used in the convergence analysis [12, 45, 64]. We put the proof in the appendix of the full paper[1].

# 6 EXPERIMENTS

## 6.1 Experimental Setup

**Datasets.** Table 2 summarizes the datasets used in experiments. The first three datasets are common benchmarks for evaluating T-GNNs [35, 50, 66, 71]. Specifically, the MOOC dataset consists of student-course interactions; the Wikipedia dataset records one month of edits made by users on Wikipedia pages; the Reddit dataset contains one month of posts made by users on subreddits. Besides, AskUbuntu, Superuser, and Wiki-Talk consist of timestamped user-user interactions collected over six years from the corresponding platforms. All the above six graphs are non-stationary because the patterns of user-user or user-item interactions shift over time. To empirically study the impact of stationarity, we build two synthetic graphs using the stochastic block model (SBM) [24, 46, 67]. Given $n$ vertices from $r$ disjoint communities $C_1, \ldots, C_r$, SBM randomly samples edges based on a $r \times r$ matrix $P$. For any two vertices $u \in C_i$ and $v \in C_j$, SBM generates an edge with probability $P_{i,j}$. Consider $10k$ nodes from five communities with $P_{i,i} = 0.2$ and $P_{i,j} = 0.01 (i \neq j)$. We build stationary and non-stationary graphs in four stages following [24, 46]. To create the stationary graph, we sample 250k edges with fake timestamps at each stage. To create the non-stationary graph, we randomly pick 10% of nodes, move them to another community, and then sample 250k edges at each stage.

**Table 2: Summary of datasets used in experiments. $|V|$ and $|E|$ denote the number of nodes and temporal links, respectively. $d_v$ and $d_e$ denote the dimensionality of node and edge features, respectively. The last column indicates whether the corresponding dataset is a bipartite graph or not.**

| Dataset | $\|V\|$ | $\|E\|$ | $d_v$ | $d_e$ | Bipartite |
|---|---|---|---|---|---|
| MOOC [35] | 7,144 | 411,749 | 172 | 4 | True |
| Wikipedia [35] | 9,227 | 157,474 | 172 | 172 | True |
| Reddit [35] | 10,984 | 672,447 | 172 | 172 | True |
| AskUbuntu [1] | 159,316 | 964,437 | 172 | 0 | False |
| SuperUser [2] | 194,085 | 1,443,339 | 172 | 0 | False |
| Wiki-Talk [3] | 1,140,149 | 7,833,140 | 172 | 0 | False |
| Stationary | 10,000 | 1,000,000 | 172 | 0 | False |
| Non-stationary | 10,000 | 1,000,000 | 172 | 0 | False |

We chronologically split all the graphs in Table 2 into training (70%), validation (15%), and test (15%) sets. To test model performance on unseen nodes, we randomly sample 10% of nodes from each dataset and mask them during model training as in [71].

**Compared Models.** We apply our Caerus framework to TGN [50] and compare our techniques with the state-of-the-art T-GNNs.

- **Caerus-P** caches intermediate results without cache limit and prunes the computation graphs via **p**ush-and-**p**ull (Algorithm 1).
- **Caerus-R** scales to large dynamic graphs under cache size constraints via **r**euse-or-**r**ecompute (Algorithm 2).
- JODIE [35] updates the representations of two nodes involved in each edge interaction using double RNN models.
- TGAT [71] encodes continuous time using random Fourier features and mimics the message passing of static GNNs.
- TGN [50] is a generic T-GNN framework that includes previous works [35, 59, 71] as special cases.
- CAW [66] adopts anonymized random walks for temporal sampling and aggregates sampled results using attention modules.
- APAN [65] decouples model inference and graph computation and accelerates inference by asynchronous message propagation.

**Evaluation Metrics.** As for effectiveness, we compare average precision (AP) of models on the test set. More specifically, we consider

[1] https://anonymous.4open.science/r/SIGMOD23_Caerus-7D47/Caerus.pdf

**Table 3: Comparison of 2-layer models on small dynamic graphs. We report the performance in transductive average precision (%), inductive average precision (%), and convergence time (s). Besides, we report the number of epochs required for models to converge in parentheses. The best and second-best results are marked in bold and underlined, respectively.**

| Model | Wikipedia | | | Reddit | | | MOOC | | |
|---|---|---|---|---|---|---|---|---|---|
| | Trans AP | Induct AP | Convergence (#) | Trans AP | Induct AP | Convergence (#) | Trans AP | Induct AP | Convergence (#) |
| JODIE | 95.16 | 93.13 | 2356.4 (18) | 95.83 | 93.20 | 9243.64 (14) | 83.26 | 81.77 | 5160.30 (15) |
| TGAT | 94.26 | 92.88 | 2881.4 (29) | 97.80 | 96.08 | 8524.2 (15) | 70.22 | 70.83 | 7838.5 (25) |
| TGN | 98.58 | 98.05 | 1846.9 (22) | 98.66 | 97.55 | 8868.4 (20) | 88.88 | 88.17 | 5086.83 (21) |
| APAN | 96.41 | 96.06 | 1605.0 (21) | 98.50 | 97.38 | 16431.8 (18) | 87.02 | 86.74 | 3374.1 (14) |
| CAW | 98.18 | **98.24** | 10175.5 (13) | 98.54 | **97.97** | 75585.1 (16) | 80.60 | 80.18 | 34063.9 (14) |
| Caerus-P | **98.62** | 98.09 | **148.81 (23)** | **98.68** | 97.54 | **597.2 (20)** | **89.34** | **89.33** | **303.1 (18)** |

**Table 4: Comparison of 2-layer models on large dynamic graphs. We report the performance in transductive average precision (%), inductive average precision (%), and convergence time (s). Besides, we report the number of epochs required for models to converge in parentheses. The best and second-best results are marked in bold and underlined, respectively. Caerus-R is implemented with a cache limit $m = 1,000$. "–" denotes CUDA memory error, and "~" denotes time limit exceed such that we cannot finish one epoch of model training in 12 hours.**

| Model | AskUbuntu | | | SuperUser | | | Wiki-Talk | | |
|---|---|---|---|---|---|---|---|---|---|
| | Trans AP | Induct AP | Convergence (#) | Trans AP | Induct AP | Convergence (#) | Trans AP | Induct AP | Convergence (#) |
| JODIE | – | – | – | – | – | – | – | – | – |
| TGAT | 87.57 | 84.21 | 11728.5 (21) | 86.40 | 83.12 | 17129.3 (19) | 91.72 | 85.38 | 189420.8 (34) |
| TGN | 94.51 | 92.73 | 36202.6 (20) | 93.18 | 91.76 | 81747.6 (24) | – | – | – |
| APAN | 89.17 | 88.43 | 21606.1 (14) | 87.07 | 85.50 | 48724.2 (19) | ~ | ~ | ~ |
| CAW | 90.87 | 90.63 | 61903.7 (14) | 88.92 | 88.32 | 111744.8 (15) | ~ | ~ | ~ |
| Caerus-P | 94.58 | 93.02 | **715.7 (18)** | 93.53 | 91.93 | 1309.6 (21) | 95.57 | 90.02 | **8575.4 (18)** |
| Caerus-R | **95.57** | **93.06** | 972.3 (21) | **94.49** | **92.06** | **1173.5 (17)** | **95.79** | **90.37** | 9929.0 (19) |

two types of tasks: *transductive* learning and *inductive* learning. The transductive task examines model performance on nodes that have been observed in training. In contrast, the inductive task tests the learning ability of models on unseen nodes. As for efficiency, we evaluate the per-epoch training time (i.e., epoch time) and the total training time till convergence (i.e., convergence time) of models. Following [50, 66, 71], we say a model converges if its validation performance does not increase for five epochs.

**Training Configurations.** Our experimental configurations generally follow the previous works [50, 66, 71]. In experiments, we focus on link prediction tasks that forecast whether an interaction happens between two given nodes at a future timestamp. Since the original datasets only contain true observations, we randomly sample an equal number of false links. For all the datasets, we set the batch size to be 200 for model training, validation, and testing. Moreover, we set the maximum number of training epochs to be 50 with early stopping. Specifically, we stop model training if the validation performance does not increase for five epochs. As for hyperparameters, we tune the learning rate for each method over the space $\{1, 0.1, 0.01, 0.001\} \times 10^{-4}$. We conducted experiments on a server with 72 Intel(R) Xeon(R) 2.60 GHz CPUs, 8 GeForce RTX 2080 GPUs, and 256 GB of memory. Each experiment was repeated five times, and the average results were reported.

**Implementation of Models.** Caerus-P and Caerus-R are built on top of the generic framework [50]. Given a target node at time $t$, our models sample its top-$k$ recent neighbors (Eq.1), aggregate the neighborhood information with a multi-head attention module (Eq.6), and then combine the intermediate results using a 2-layer MLP model (Eq.7). The critical difference is that Caerus uses intermediate embeddings as key values in attention modules, while TGN uses state vectors. For the baselines, we adapt the officially released codes to our evaluation pipeline. To achieve a fair comparison, we set the maximum neighbor size $k = 10$ for APAN, TGAT, TGN, Caerus-P, and Caerus-R. For each target node in CAW, we sample 64 length-2 anonymized random walks as it is the default setting in the original implementation.

## 6.2 Key Results

Table 3 and Table 4 compare our methods with the state-of-the-art baselines on small and large dynamic graphs, respectively. We omit the results of Caerus-R in Table 3 since we can easily cache all the intermediates on small graphs. Moreover, we set the cache limit $m = 1,000$ for Caerus-R on large graphs since Caerus-R can *already achieve considerable cache hits with such a small cache*. For a fair comparison, all the models except JODIE are implemented in 2 layers. JODIE cannot go deep since it does not involve recursive neighbor aggregation. In the following, we compare the methods in terms of efficiency, effectiveness, and convergence rate.

**Time Efficiency of Caerus.** As shown in Table 3 and Table 4, our Caerus-P and Caerus-R are *one or two orders of magnitude faster* than the compared baselines. Notably, compared to Caerus-P, the recomputation cost of Caerus-R is marginal on large dynamic graphs. This observation indicates that our optimal cache replacement algorithm MRD (Algorithm 4) can achieve high cache hit ratios even under a small cache limit $m = 1,000$. In principle, Caerus-P and Caerus-R are much faster because they avoid *temporal explosion*

and *neighbor explosion* by sharing *newly computed* embeddings across multiple timestamps and by reusing *previously cached* historical embeddings, respectively. Although JODIE is computationally simple, it partitions the dynamic graph into very small batches to avoid overlapping between interactions. Small batches make GPU resources underutilization, thus slowing down model training. TGAT and TGN follow the generic framework described in Section 2.3. They cannot resolve the temporal and neighbor explosion problems such that the time complexity still increases linearly with the number of unique timestamps and grows exponentially with the number of layers. Besides, CAW and APAN are very sensitive to the available CPU resources and RAM. As shown in Table 4, these two models cannot even finish one epoch of training in 12 hours on the Wiki-Talk dataset.

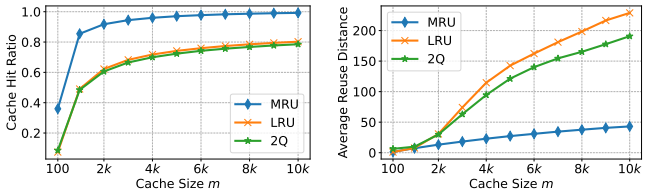
**Memory Efficiency of Caerus.** Caerus-P and Caerus-R can also reduce the memory consumption during model training. Although these two algorithms cache intermediates in GPU memory, our reuse schemes *save not only the computational cost but also the memory required to finish the corresponding computations*. In contrast, the memory cost of the baselines still scales with the number of nodes in the computation graph. As shown by the "–" marks in Table 4, JODIE and TGN cannot scale to large dynmaic graphs due to CUDA memory error. Our Caerus-R can scale to large graphs without incurring memory error by setting a practical cache limit.

**Convergence Rate of Caerus.** Table 3 and Table 4 show the number of epochs required for different models to converge. Compared to the baselines that do not reuse intermediates, Caerus-P and Caerus-R demonstrate better generalization performance while having no noticeable impact on the convergence rate. This finding is consistent with our theoretical analysis (Theorem 5) on the convergence guarantee of the Caerus framework. Although our gradient blocking and reuse schemes cause approximated gradients, Theorem 5 shows that the gradient is asymptotically unbiased. Therefore, our Caerus-P and Caerus-R approaches can converge to a local optimum as vanilla T-GNNs do.

**Table 5: Training loss and test AP on large dynamic graphs.**

| Model | AskUbuntu | | SuperUser | | Wiki-Talk | |
|---|---|---|---|---|---|---|
| | Train loss | Test AP | Train loss | Test AP | Train loss | Test AP |
| TGN | **0.622** | 94.51 | **0.578** | 93.18 | – | – |
| Caerus-P | 0.679 | 94.58 | 0.617 | 93.53 | 0.634 | 95.57 |
| Caerus-R | 0.654 | **95.57** | 0.594 | **94.49** | **0.618** | **95.79** |

**Effectiveness of Caerus.** Table 3 compares the average precision (AP) of models in both transductive and inductive settings. Caerus-P matches or outperforms all the compared baselines in the transductive setting. CAW is specifically designed for *inductive* representation learning on dynamic graphs. Although it achieves better inductive AP than Caerus-P on Wikipedia and Reddit datasets, our Caerus-P is more efficient with only 0.15%~0.43% accuracy loss. Table 4 shows model performance on large dynamic graphs. Surprisingly, our Caerus-P and Caerus-R outperform all the baselines, which do not reuse intermediates, in both transductive and inductive settings. Furthermore, Caerus-R achieves higher AP than Caerus-P in all the cases. The reason is that Caerus-R is equipped with our MRD algorithm. MRD can *avoid keeping and reusing incredibly stale embeddings* that would introduce significant approximation errors, thus improving model precision.




**(a) Cache hit ratio** **(b) Average reuse distance**

**Figure 4: Comparison of caching policies on AskUbuntu.**

Table 5 further investigates why our reuse schemes can achieve superior effectiveness on large dynamic graphs. Specifically, Table 5 shows that the best baseline TGN outperforms Caerus-P and Caerus-R in the training stage, i.e., TGN achieves the smallest training loss. Conversely, Caerus-P and Caerus-R obtain higher test AP. This finding suggests that although our reuse schemes introduce approximation errors, slight noises could overcome the overfitting problem and thus improve models' generalization ability [31, 45].

**Table 6: Training Caerus-R with various caching policies on AskUbuntu when the cache limit is 5*k*. We report the preparation overhead (s) of caching policies, convergence time (s) of models, transductive AP (%), and inductive AP (%).**

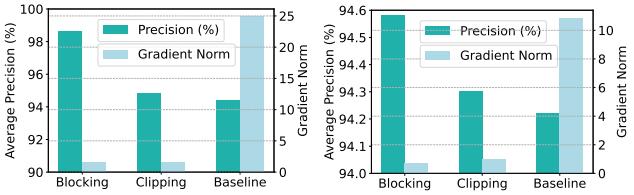
| Algorithm | Preparation | Convergence | Trans AP | Induct AP |
|---|---|---|---|---|
| LRU | 1.49 | 988.8 | 95.32 | 92.97 |
| 2Q | 2.01 | 1032.6 | 95.25 | 92.94 |
| MRD | 3.86 | 823.8 | 95.73 | 93.06 |

## 6.3 Comparison of Cache Replacement Policies

**Cache Hit Ratio and Reuse distance.** Figure 4 compares our MRD algorithm with the classical cache replacement policies 2Q [32] and LRU [14]. Since MRD is optimal in maximizing the number of cache hits, it achieves higher cache hit ratios. In contrast, LRU and 2Q tend to cache all the newly computed embeddings even if they may not be reused in the near future. On the other hand, MRD leads to significantly smaller reuse distance with the increase of cache size. Here, the reuse distance of a cached embedding is defined as the number of passed training iterations from its being cached to its being reused. Hence, smaller reuse distance indicates less staleness of the cached embedding.

**Training Efficiency.** Table 6 further demonstrates the overhead of different caching policies and how they affect the model training time till convergence. Since MRD achieves higher cache hit ratios than LRU and 2Q, training Caerus-R models with MRD is generally faster. Besides, MRD only takes 3.86 seconds to make cache plans on AskUbuntu with a cache limit of 5,000. The preparation overhead is negligible compared to the total training cost of 823.8 seconds. Therefore, our MRD policy itself is efficient and accelerates Caerus-R by reducing the amount of recomputation.

**Model Effectiveness.** Table 6 also shows that MRD can lead to better model effectiveness than LRU and 2Q on AskUbuntu. The same conclusion can be drawn on other datasets. This is consistent with our observation in Figure 4b, which demonstrates that MRD results in significantly smaller average resue distance. Therefore, training Caerus-R model with MRD can achieve better predictive performance by filtering out those stale embeddings that could introduce large approximation errors.




(a) Wikipedia (b) AskUbuntu

**Figure 5: Training Caerus-P with different gradient update strategies on Wikipedia and AskUbuntu.**

## 6.4 Comparison of Gradient Update Strategies

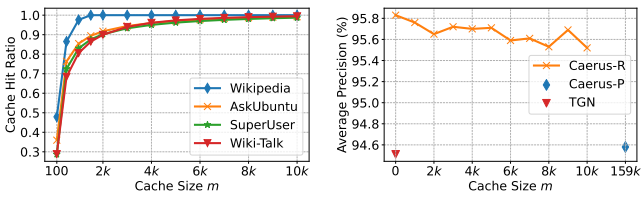
**Model Effectiveness.** In Section 3.2, we have discussed that simply reusing intermediates could cause the exploding gradient issue. Figure 5 further investigates the impact of various gradient update strategies on the Frobenius norm of gradients and the performance of Caerus-P. Note that the simple baseline, which does not attempt to reduce gradient values, indeed suffers from exploding gradients and thus cannot achieve satisfactory model precision. On the other hand, the gradient clipping method [76], a common practice for avoiding gradient explosion, can project gradient values into a normal range. However, our gradient blocking approach is more effective than the gradient clipping as the latter only results in a marginal gain in model precision. This is because simply clipping gradient values cannot prevent gradients from accumulation. Therefore, gradients from multiple timestamps would still aggregate and cause unstable model update.

**Impact of Neighbor Distribution.** We observe that the gradient explosion issue on Wikipedia is more severe than that on AskUbuntu. As shown in Figure 5, the gradient norm of the simple baseline on Wikipedia and AskUbuntu is 24.9 and 10.8, respectively. The phenomenon is caused by the difference in these two data distributions. Specifically, in-batch neighbors account for 77.9% of the total 1-hop neighbors on Wikipedia, while this value is 61.2% on AskUbuntu. As a result, the newly computed embeddings are aggressively shared across more timestamps in a data batch when training Caerus-P on Wikipedia than those on AskUbuntu. By the chain rule of differentiation, gradients from multiple timestamps would aggregate and result in gradient explosion. The above observation again validates our analysis in Section 3.2 that simply reusing intermediates can cause exploding gradients.

## 6.5 Other Ablation Studies

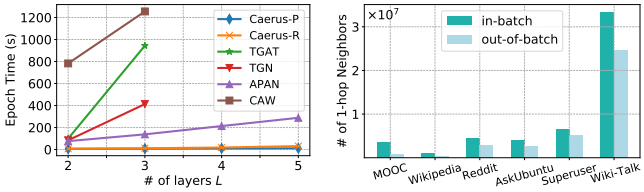
**Impact of cache size $m$.** Figure 6a demonstrates the cache hit ratios achieved by MRD on four dynamic graphs. Note that MRD can lead to 100% cache hits on Wikipedia with a tiny cache size $m = 1.5k$. For the other three large dynamic graphs, MRD achieves a cache hit ratio of about 90% with a cache size $m = 2k$. Particularly, the training set of Wiki-Talk involves 624,057 different nodes. The cache size $m = 2k$ only accounts for 0.32% of the unique training nodes. This is because dynamic graphs usually follow the *power-law distribution*, i.e., a few nodes appear in most temporal interactions. Therefore, we can achieve a high cache hit ratio by caching intermediates of some frequently occurred nodes.

Figure 6b shows the average precision of Caerus-R on AskUbuntu. As the cache size $m$ increases, the performance of Caerus-R slightly decreases but is consistently better than Caerus-P and TGN. The




(a) Cache hit ratio (b) Caerus-R on AskUbuntu

**Figure 6: (a) Cache hit ratios achieved by MRD varying cache size $m$. (b) Average precision of Caerus-R varying cache size $m$. Since Caerus-R can share newly computed embeddings across timestamps without caching, $m = 0$ indicates that only intermediates of in-batch neighbors are reused.**




(a) Epoch time (b) Reuse chance breakdown

**Figure 7: (a) Epoch time of models varying the number of layers on Wikipedia. Caerus-R is implemented with per-layer cache limit $m = 500$. Some results of baselines are absent due to CUDA memory error. (b) The total number of 1-hop in-batch and out-of-batch neighbors during model training.**

performance drop is in line with our expectation because a larger cache maintains more stale embeddings and increases the reuse distance, thus introducing larger approximation errors. Note that Caerus-R with $m = 10k$ achieves almost 100% cache hits but still outperforms Caerus-P, which caches all the historical embeddings, by a large margin. This finding suggests that the performance gap between Caerus-P and Caerus-R is caused by a few incredibly unreliable historical intermediates. Moreover, both Caerus-P and Caerus-R beat the baseline TGN. As discussed in Section 6.2, this is because our reuse scheme can act as *regularization* techniques [21, 31, 39, 72] and therefore improve models' generalization performance by overcoming the *overfitting* issue [18, 45].

**Impact of model depth $L$.** Figure 7a compares the epoch time of our Caerus-P and Caerus-R with baselines on the Wikipedia dataset. Specifically, the epoch time of Caerus-P remains stable as we increase of the number of layers $L$. In contrast, the cost of Caerus-R increases a bit faster as the number of recomputed representations generally grows exponentially with the number depth $L$. Although Caerus-R incurs extra recomputation cost, it is still significantly faster than the compared baselines. Moreover, with the increase of model depth $L$, TGAT, TGN, and CAW fail due to CUDA memory error. In contrast, our Caerus-P and Caerus-R are memory efficient and scalable. In general, the *efficiency advantage of our methods over baselines increases with the model depth L*.

**Reuse chance breakdown.** Figure 7b shows the total number of *1-hop in-batch neighbors* (e.g., $v_2$ in Figure 2b) and *1-hop out-of-batch neighbors* (e.g., $v_4$ in Figure 2b) during model training. All the six dynamic graphs exhibit a high degree of *temporal locality*, i.e., the number of in-batch neighbors exceeds that of out-of-batch neighbors. This observation also indicates that the dynamic graphs

follow the *power-law distribution*, where some common neighbor nodes frequently appear in different data batches. Caerus prunes the computation graph by reusing the newly computed embeddings of in-batch neighbors and the cached intermediates of out-of-batch neighbors. According to the reuse scheme, Figure 7b suggests that *in-batch neighbors contribute more reuse chances* to Caerus.

**Table 7: Ablation study on graph distributions. We report model performance in average precision (%) and epoch time (s). Caerus-R is implemented with a cache limit of 1,000.**

| Model | Stationary | | Non-stationary | |
|---|---|---|---|---|
| | AP | Epoch Time | AP | Epoch Time |
| TGN | **96.87** | 413.62 | 93.76 | 419.62 |
| Caerus-P | 96.76 | **46.15** | 93.73 | **45.83** |
| Caerus-R | 96.81 | 87.50 | **93.93** | 83.33 |

**Impact of Stationarity.** Table 7 compares Caerus-P, Caerus-R, and the best performing baseline TGN on two synthetic graphs. It is clear that Caerus-P and Caerus-R achieve comparable average precision as TGN on both Stationary and Non-stationary datasets. Note that the six real-world dynamic graphs shown in Table 3 and Table 4 are also non-stationary because the patterns of user-user and user-item interactions in these graphs significantly shift over time. In summary, the experiments show that Caerus-P and Caerus-R are effective on both stationary and non-stationary graphs.

**Impact of Graph Topology.** The Stationary dataset shown in Table 7 has uniform degree distribution. Compared to scale-free graphs like Wikipedia and Reddit, it generally takes a larger cache size to achieve a comparable cache hit ratio on the graph with uniform degree distribution. For instance, MRD obtains a cache hit ratio of 97.6% and 40.9% on Wikipedia and Stationary with a small cache limit of 1,000, respectively. The reason is that there is no strong temporal locality in Stationary. However, Caerus-R and Caerus-P are still significantly faster than the best baseline TGN on Stationary. This is because, aside from reusing cached intermediates, Caerus-R can further reduce the computational cost by sharing newly computed embeddings of in-batch neighbors across multiple timestamps in a data batch. Besides, our implementation of the Caerus framework is highly efficient.

## 7 RELATED WORK

**Representation Learning on Dynamic Graphs.** Prior works [22, 23, 40, 46, 53, 55, 73] mainly focus on *discrete-time dynamic graphs* (DTDGs) that are represented by graph snapshots taken at different time intervals. The random walk-based methods [40, 73] incrementally sample time-dependent walks and minimize the pairwise distance of learned representations of adjacent vertices in each walk. Another line of works [46, 53, 55] encodes the structural information of each graph snapshot using a GNN model and then learns the temporal dynamics of consecutive snapshots using a sequence model such as RNN. However, all the DTDG-based solutions need to aggregate graph snapshots with a predefined time granularity. Thus, they are not suitable for many real-world applications where interactions can happen at various time granularities.

T-GNNs [35, 50, 59, 65, 66, 71] have been developed to learn representations on *continuous-time dynamic graphs* (CTDGs) that are represented by sequences of timestamped edges. Given a new edge, the sequence-based approach JODIE [35] updates representations of the two nodes involved in the edge using RNNs. DyRep [59] further considers 2-hop neighborhood information for updating node representations. TGAT [71] encodes continuous timestamps using random Fourier features and mimics the message passing of static GNNs. TGN [50] is a generic T-GNN framework that includes [35, 59, 71] as special cases. To improve the inductive performance on unseen nodes, CAW [66] adopts an anonymization strategy that replace node identities with the number of node occurrences based on a set of sampled walks. In addition, APAN [65] decouples model inference and graph computation and accelerates model inference through asynchronous message propagation.

**Reuse in Static GNNs.** A few recent works have considered using historical embeddings to minimize sampling variance [12, 15] or accelerate computation [18, 37, 45, 58] for *static* GNNs. To avoid neighbor explosion, VR-GCN [12] reuses cached intermediate embeddings in forward propagation and develops control variate based neighbor sampling algorithm. MVS-GNN [15] further improves the sampling efficiency and reduces the variance of mini-batch construction. On the other hand, GNNAutoScale [18] prunes the computation graph by reusing historical embeddings of 1-hop out-of-batch neighbors. IGLU [45] adopts a more aggressive caching strategy where the intermediate results are lazily updated at regular intervals. Dorylus [58] reduces the communication cost in distributed GPU servers by caching the embeddings of boundary nodes. CacheGNN [37] accelerates the inference of static GNNs over graph snapshots. In contrast to the above works, we improve the training efficiency of T-GNNs under practical cache limits.

**Reuse in ML Systems.** Materialization and reuse in ML systems are reminiscent of the classical view materialization problems [13, 30, 41] in database systems. Such techniques have been adopted in a wide range of ML lifecycle management systems such as data versioning [9, 10], model management [42, 61], feature selection [44, 56, 57, 75], model training [43, 64], model diagnosis [60], and model serving [16, 33, 36]. Specifically, HELIX [69, 70] and Collaborative Optimizer [17] optimize the execution cost across ML pipelines by caching and reusing intermediates. LIMA [48] further provides more fine-grained lineage tracing and reuse inside ML systems. Note that the above works cannot be applied to T-GNNs because they do not support approximate reuse. Moreover, HET [43] reduces the communication overhead of training large embedding models by caching hot embeddings on local machines with consistency guarantees. In contrast to prior works, Caerus captures the approximate reuse chances for T-GNN training with approximation and convergence guarantees.

## 8 CONCLUSION

In this paper, we present Caerus, a scalable caching-based framework that significantly reduces the computational cost of T-GNNs on dynamic graphs. To scale Caerus to large dynamic graphs without compromising model accuracy, we propose an optimal cache replacement algorithm under practical cache limits. Moreover, we develop theoretical analysis on the approximation errors introduced by our reuse mechanisms and offer rigorous convergence guarantees. Extensive experiments have validated that Caerus can be two orders of magnitude faster than the state-of-the-art baselines while achieving higher precision on various dynamic graphs.

# REFERENCES


[1] [n.d.]. Ask Ubuntu. http://snap.stanford.edu/data/sx-askubuntu.html.
[2] [n.d.]. Super User. http://snap.stanford.edu/data/sx-superuser.html.
[3] [n.d.]. wiki-talk. http://snap.stanford.edu/data/wiki-talk-temporal.html.
[4] [n.d.]. Wikipedia edit history dump. https://meta.wikimedia.org/wiki/Data_dumps.
[5] Bilge Acun, Matthew Murphy, Xiaodong Wang, Jade Nie, Carole-Jean Wu, and Kim M. Hazelwood. 2021. Understanding Training Efficiency of Deep Learning Recommendation Models at Scale. In *HPCA*. IEEE, 802–814.
[6] Susanne Albers, Sanjeev Arora, and Sanjeev Khanna. 1999. Page Replacement for General Caching Problems. In *SIAM*. ACM/SIAM, 31–40.
[7] Raghu Arghal, Eric Lei, and Shirin Saeedi Bidokhti. 2021. Robust Graph Neural Networks via Probabilistic Lipschitz Constraints. *CoRR* abs/2112.07575 (2021).
[8] Laszlo A. Belady. 1966. A Study of Replacement Algorithms for Virtual-Storage Computer. *IBM Syst. J.* 5, 2 (1966), 78–101.
[9] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *CIDR*. www.cidrdb.org.
[10] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya G. Parameswaran. 2015. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *PVLDB* 8, 12 (2015), 1346–1357.
[11] Matthias Boehm, Arun Kumar, and Jun Yang. 2019. *Data Management in Machine Learning Systems.* Morgan & Claypool Publishers.
[12] Jianfei Chen, Jun Zhu, and Le Song. 2018. Stochastic Training of Graph Convolutional Networks with Variance Reduction. In *ICML*, Vol. 80. PMLR, 941–949.
[13] Rada Chirkova and Jun Yang. 2012. Materialized Views. *Foundations and Trends in Databases (TODS)* 4, 4 (2012), 295–405.
[14] Marek Chrobak and John Noga. 1998. LRU is Better than FIFO. In *PODS*. ACM/SIAM, 78–81.
[15] Weilin Cong, Rana Forsati, Mahmut T. Kandemir, and Mehrdad Mahdavi. 2020. Minimal Variance Sampling with Provable Guarantees for Fast Training of Graph Neural Networks. In *SIGKDD*. ACM, 1393–1403.
[16] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*. USENIX Association, 613–627.
[17] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker Markl. 2020. Optimizing Machine Learning Workloads in Collaborative Environments. In *SIGMOD*. ACM, 1701–1716.
[18] Matthias Fey, Jan Eric Lenssen, Frank Weichert, and Jure Leskovec. 2021. GNNAutoScale: Scalable and Expressive Graph Neural Networks via Historical Embeddings. In *ICML*, Vol. 139. 3294–3304.
[19] Arnaud Fréville. 2004. The multidimensional 0-1 knapsack problem: An overview. *European Journal of Operational Research* 155, 1 (2004), 1–21.
[20] Fernando Gama, Joan Bruna, and Alejandro Ribeiro. 2020. Stability Properties of Graph Neural Networks. *IEEE Trans. Signal Process.* 68 (2020), 5680–5695.
[21] Federico Girosi, Michael J. Jones, and Tomaso A. Poggio. 1995. Regularization Theory and Neural Networks Architectures. *Neural Computation* 7, 2 (1995), 219–269.
[22] Palash Goyal, Sujit Rokka Chhetri, and Arquimedes Canedo. 2020. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge Based System* 187 (2020).
[23] Palash Goyal, Sujit Rokka Chhetri, Ninareh Mehrabi, Emilio Ferrara, and Arquimedes Canedo. 2018. DynamicGEM: A Library for Dynamic Graph Embedding Methods. *CoRR* abs/1811.10734 (2018).
[24] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. DynGEM: Deep Embedding Method for Dynamic Graphs. *CoRR* abs/1805.11273 (2018).
[25] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. In *SIGKDD*. ACM, 855–864.
[26] Ehsan Hajiramezanali, Arman Hasanzadeh, Krishna R. Narayanan, Nick Duffield, Mingyuan Zhou, and Xiaoning Qian. 2019. Variational Graph Recurrent Neural Networks. In *NeurIPS*. 10700–10710.
[27] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *IEEE Data Eng. Bull.* 40, 3 (2017), 52–74.
[28] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *NIPS*. 1024–1034.
[29] Boris Hanin. 2018. Which Neural Net Architectures Give Rise to Exploding and Vanishing Gradients?. In *NeurIPS*. 580–589.
[30] Eric N. Hanson. 1987. A Performance Analysis of View Materialization Strategies. In *SIGMOD*. ACM Press, 440–453.
[31] Kam-Chuen Jim, C. Lee Giles, and Bill G. Horne. 1996. An analysis of noise in recurrent neural networks: convergence and generalization. *IEEE Transactions on Neural Networks* 7, 6 (1996), 1424–1438.
[32] Theodore Johnson and Dennis E. Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *VLDB*. Morgan Kaufmann, 439–450.
[33] Daniel Kang, John Emmons, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2017. NoScope: Optimizing Deep CNN-Based Queries over Video Streams at Scale. *PVLDB* 10, 11 (2017), 1586–1597.
[34] Thomas N. Kipf and Max Welling. 2016. Variational Graph Auto-Encoders. *CoRR* abs/1611.07308 (2016).
[35] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *SIGKDD*. ACM, 1269–1278.
[36] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. 2018. PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems. In *OSDI*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 611–626.
[37] Haoyang Li and Lei Chen. 2021. Cache-based GNN System for Dynamic Graphs. In *CIKM*. ACM, 937–946.
[38] Edo Liberty, Zohar S. Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Sundar Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *SIGMOD*. ACM, 731–737.
[39] Dongsheng Luo, Wei Cheng, Wenchao Yu, Bo Zong, Jingchao Ni, Haifeng Chen, and Xiang Zhang. 2021. Learning to Drop: Robust Graph Neural Network via Topological Denoising. In *WSDM*. ACM, 779–787.
[40] Sedigheh Mahdavi, Shima Khoshraftar, and Aijun An. 2018. dynnode2vec: Scalable Dynamic Network Embedding. In *BigData*. IEEE, 3762–3765.
[41] Imene Mami and Zohra Bellahsene. 2012. A survey of view selection methods. *SIGMOD Record* 41, 1 (2012), 20–29.
[42] Hui Miao, Ang Li, Larry S. Davis, and Amol Deshpande. 2017. ModelHub: Deep Learning Lifecycle Management. In *ICDE*. IEEE Computer Society, 1393–1394.
[43] Xupeng Miao, Hailin Zhang, Yining Shi, Xiaonan Nie, Zhi Yang, Yangyu Tao, and Bin Cui. 2021. HET: Scaling out Huge Embedding Model Training via Cache-enabled Distributed Framework. *PVLDB* 15, 2 (2021), 312–320.
[44] Supun Nakandala and Arun Kumar. 2020. Vista: Optimized System for Declarative Feature Transfer from Deep CNNs at Scale. In *SIGMOD*. ACM, 1685–1700.
[45] S. Deepak Narayanan, Aditya Sinha, Prateek Jain, Purushottam Kar, and Sundararajan Sellamanickam. 2021. IGLU: Efficient GCN Training via Lazy Updates. *CoRR* abs/2109.13995 (2021).
[46] 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 *AAAI*. 5363–5370.
[47] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: online learning of social representations. In *SIGKDD*. ACM, 701–710.
[48] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *SIGMOD*. ACM, 1426–1439.
[49] Hassan Ramchoun, Mohammed Amine Janati Idrissi, Youssef Ghanou, and Mohamed Ettaouil. 2016. Multilayer Perceptron: Architecture Optimization and Training. *International Journal of Interactive Multimedia and Artificial Intelligence* 4, 1 (2016), 26–30.
[50] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. *CoRR* abs/2006.10637 (2020).
[51] Benjamin Van Roy. 2007. A short proof of optimality for the MIN cache replacement algorithm. *Information Process. Letter* 102, 2-3 (2007), 72–73.
[52] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *CoRR* abs/1609.04747 (2016).
[53] 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 *WSDM*. ACM, 519–527.
[54] Andrew I. Schein, Alexandrin Popescul, Lyle H. Ungar, and David M. Pennock. 2002. Methods and metrics for cold-start recommendations. In *SIGIR*. ACM, 253–260.
[55] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. 2018. Structured Sequence Modeling with Graph Convolutional Recurrent Networks. In *ICONIP*, Vol. 11301. Springer, 362–373.
[56] Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim Kraska. 2019. Democratizing Data Science through Interactive Curation of ML Pipelines. In *SIGMOD*. ACM, 1171–1188.
[57] Evan R. Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J. Franklin, and Benjamin Recht. 2017. KeystoneML: Optimizing Pipelines for Large-Scale Advanced Analytics. In *ICDE*. IEEE Computer Society, 535–546.
[58] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. 2021. Dorylus: Affordable, Scalable, and Accurate GNN Training with Distributed CPU Servers and Serverless Threads. In *OSDI*. USENIX Association, 495–514.

[59] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. 2019. DyRep: Learning Representations over Dynamic Graphs. In *ICLR*.
[60] Manasi Vartak, Joana M. F. da Trindade, Samuel Madden, and Matei Zaharia. 2018. MISTIQUE: A System to Store and Query Model Intermediates for Model Diagnosis. In *SIGMOD*. ACM, 1285–1300.
[61] Manasi Vartak and Samuel Madden. 2018. MODELDB: Opportunities and Challenges in Managing Machine Learning Models. *IEEE Data Eng. Bull.* 41, 4 (2018), 16–25.
[62] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *ICLR*.
[63] Aladin Virmaux and Kevin Scaman. 2018. Lipschitz regularity of deep neural networks: analysis and efficient estimation. In *NeurIPS*. 3839–3848.
[64] Cheng Wan, Youjie Li, Cameron R. Wolfe, Anastasios Kyrillidis, Nam Sung Kim, and Yingyan Lin. 2022. PipeGCN: Efficient Full-Graph Training of Graph Convolutional Networks with Pipelined Feature Communication. *CoRR* abs/2203.10428 (2022).
[65] Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. 2021. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *SIGMOD*. ACM, 2628–2638.
[66] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. 2021. Inductive Representation Learning in Temporal Networks via Causal Anonymous Walks. In *ICLR*.
[67] Yuchung J. Wang and George Y. C. Wong. 1987. Stochastic Block Models for Directed Graphs. *J. Amer. Statist. Assoc.* 82 (1987), 8–19.
[68] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya G. Parameswaran. 2018. Accelerating Human-in-the-loop Machine Learning: Challenges and Opportunities. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD*. ACM, 9:1–9:4.
[69] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Accelerating Human-in-the-loop Machine Learning. *PVLDB* 11, 12 (2018), 1958–1961.
[70] Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya G. Parameswaran. 2018. Helix: Holistic Optimization for Accelerating Iterative Machine Learning. *PVLDB* 12, 4 (2018), 446–460.
[71] Da Xu, Chuanwei Ruan, Evren Körpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive representation learning on temporal graphs. In *ICLR*.
[72] Han Yang, Kaili Ma, and James Cheng. 2021. Rethinking Graph Regularization for Graph Neural Networks. In *AAAI*. AAAI Press, 4573–4581.
[73] Wenchao Yu, Wei Cheng, Charu C. Aggarwal, Kai Zhang, Haifeng Chen, and Wei Wang. 2018. NetWalk: A Flexible Deep Embedding Approach for Anomaly Detection in Dynamic Networks. In *SIGKDD*. ACM, 2672–2681.
[74] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *ICLR*.
[75] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization Optimizations for Feature Selection Workloads. *ACM Transactions on Database Systems (TODS)* 41, 1 (2016), 2:1–2:32.
[76] Jingzhao Zhang, Tianxing He, Suvrit Sra, and Ali Jadbabaie. 2020. Why Gradient Clipping Accelerates Training: A Theoretical Justification for Adaptivity. In *ICLR*. OpenReview.net.

# A PROOF OF THEOREM 5

We finish the proof of Theorem 5 in four steps. (1) We first bound the change of model parameters during one training epoch. (2) We next show that the staleness of cached embeddings and the approximation error of newly computed embeddings can be bounded if the change of model parameters can be bounded. (3) Then, we relate the bias in approximated gradients to the staleness of reused embeddings. (4) Finally, we establish the convergence guarantee using properties like smoothness and Lipschitz continuous.

## A.1 Preliminaries

**Generic T-GNN Architecture.** The analysis of gradient update and convergence rate is always model-dependent. Without loss of generality, we consider vanilla T-GNNs in the most generic form as follows:

$$Z^l = P^l H^{l-1} W^l, \quad H^l = \sigma(Z^l), \quad l = 1, \ldots, L, \tag{22}$$

where $H^l$ is the embeddings at the $l$-layer, $W^l$ is the learnable weight matrix used to compute the $l$-th layer embeddings, $P^l$ is a weighted adjacency matrix for feature aggregation, and $\sigma(\cdot)$ is an activation function. For ease of simplicity, we omit edge features and time encodings in Eq.22. However, as we will show later, our theoretical analysis can be easily extended to incorporate edge and time features. Similarly, we can express Caerus-P and Caerus-R models in the following generic form:

$$\tilde{Z}^l = (\tilde{P}^l \tilde{H}^{l-1} + \bar{P}^l \bar{H}^{l-1}) W^l, \quad \tilde{H}^l = \sigma(\tilde{Z}^l), \quad l = 1, \ldots, L, \tag{23}$$

where $\tilde{H}^l$ and $\bar{H}^l$ denote the newly computed embeddings and reused embeddings, respectively. Besides, $\tilde{P}$ and $\bar{P}$ denote matrices used to aggregate $\tilde{H}^{l-1}$ and $\bar{H}^{l-1}$ features, respectively.

It is worth noting that the exact embeddings $H^l$ computed by vanilla T-GNN and the approximated embeddings $\tilde{H}^l$ computed by Caerus can have different shapes. This is because Caerus prunes the computation graph by calculating the embeddings for fewer nodes. Let $\text{row}(H)$ denotes the number of rows in the matrix $H$. Then, $\text{row}(H^l) = \text{row}(\tilde{H}^l)$ if $l = L$. Otherwise, $\text{row}(H^l) \geq \text{row}(\tilde{H}^l)$. It is generally difficult to compare the gradients of embedding matrices in different shapes. For ease of analysis, we rewrite Caerus as follows:

$$\tilde{Z}^l = (\tilde{P}^l (\tilde{H}^{l-1} - \bar{H}^{l-1}) + P \bar{H}^{l-1}) W^l, \quad \tilde{H}^l = \sigma(\tilde{Z}^l), \tag{24}$$

where $\text{row}(\tilde{H}^{l-1}) = \text{row}(\bar{H}^{l-1}) = \text{row}(H^{l-1})$, and $\tilde{P}^l$ is an adjacency matrix used to aggregate the features of newly computed embeddings $\tilde{H}^{l-1}$. Specifically, in Eq.24, we conceptually augment the computation graph of Caerus so that Caerus computes embeddings for the same set of nodes as vanilla T-GNNs at each layer $l$ ($l = 1, \ldots, L$). More importantly, we can remove the impact of extra conceptually computed embeddings by setting the corresponding terms in the adjacency matrix $\tilde{P}^l$ to zero. As a result, Eq.23 and Eq.24 are mathematically equivalent and produce the same temporal embeddings.

**Updating T-GNN via Gradient Descent.** T-GNNs are updated using batches of interactions. Consider a batch of target nodes $B_\tau$, where $(v_i, t) \in B_\tau$ denotes node $v_i$ at timestamp $t$. For each $(v_i, t) \in B_\tau$, an $L$-layer vanilla T-GNN computes its temporal embedding $h_i^L(t)$ and then gets the loss value $f(h_i^L(t), y_i(t))$, where $y_i(t)$ is the true label of node $v_i$ at $t$, and $f(\cdot, \cdot)$ is a loss function. Finally, parameters of the vanilla T-GNN are updated through

$$W_\tau = W_{\tau-1} - \eta \nabla \mathcal{L}(W_{\tau-1}), \tag{25}$$

where $\nabla \mathcal{L}(W_{\tau-1}) = \frac{1}{|B_\tau|} \sum_{(v_i,t) \in B_\tau} \nabla f(h_i^L(t), y_i(t))$ is the gradient obtained on the data batch $B_\tau$ with parameters $W_{\tau-1}$, $W_{\tau-1} = [W_{\tau-1}^1, \ldots, W_{\tau-1}^L]$ is the model parameters obtained after updating T-GNN on the batch $B_{\tau-1}$, and $\eta$ is the learning rate.

Similarly, for each target node $(v_i, t) \in B_\tau$, an $L$-layer Caerus-P or Caerus-R model computes its temporal embedding $\tilde{h}_i^L(t)$ and then gets the loss value $f(\tilde{h}_j^i(t), y_i(t))$. Finally, parameters of our Caerus model are updated through

$$W_\tau = W_{\tau-1} - \eta \nabla \tilde{\mathcal{L}}(W_{\tau-1}), \tag{26}$$

where $\nabla \tilde{\mathcal{L}}(W_{\tau-1}) = \frac{1}{|B_\tau|} \sum_{(v_i,t) \in B_\tau} \nabla f(\tilde{h}_i^L(t), y_i(t))$ is the approximated gradient obtained on the batch $B_\tau$ with parameters $W_{\tau-1}$.

**Backpropagation Rules.** Let $f_i(t) = f(h_i^L(t), y_i(t))$ and $\tilde{f}_i(t) = f(\tilde{h}_i^L(t), y_i(t))$. Then, for vanilla T-GNNs in the form of Eq.22, we have the following backpropagation rules:

$$\begin{aligned}\nabla_{H^{l-1}} f_i(t) &= (P^l)^T \nabla_{Z^l} f_i(t) (W^l)^T \\ \nabla_{W^l} f_i(t) &= (P^l H^{l-1})^T \nabla_{Z^l} f_i(t) \\ \nabla_{Z^l} f_i(t) &= \sigma'(Z^l) \circ \nabla_{H^l} f_i(t).\end{aligned} \tag{27}$$

Moreover, for Caerus-P and Caerus-R in the form of Eq.24, we have the following backpropagation rules:

$$\begin{aligned}\nabla_{\tilde{H}^{l-1}} \tilde{f}_i(t) &= (\tilde{P}^l)^T \nabla_{\tilde{Z}^l} \tilde{f}_i(t) (W^l)^T \\ \nabla_{W^l} \tilde{f}_i(t) &= (\tilde{P}^l (\tilde{H}^{l-1} - \bar{H}^{l-1}) + P^l \bar{H}^{l-1})^T \nabla_{\tilde{Z}^l} \tilde{f}_i(t) \\ \nabla_{\tilde{Z}^l} \tilde{f}_i(t) &= \sigma'(\tilde{Z}^l) \circ \nabla_{\tilde{H}^l} \tilde{f}_i(t).\end{aligned} \tag{28}$$

Note that we do not calculate partial dervatives with repect to the cached embeddings $\bar{H}^l$ $(l = 1, \ldots, L-1)$ in the above rules. This is because Caerus adopts a gradient blocking strategy so that gradients will not propagate through the cached and reused embeddings.

## A.2 Bounding the Parameter Change

We bound the change of model parameters in one training epoch. Specifically, the parameters of T-GNN are updated in severe training epochs, where each epoch consists of a sequence of training batches. Let the gradient of model parameters be bounded, i.e., $\|\nabla \tilde{\mathcal{L}}(W_\tau)\|_F \leq G$ and $\|\nabla \mathcal{L}(W_\tau)\|_F \leq G$. Besides, let $n$ be the number of bathes in an epoch. Then, for any two weight matries $W_\tau$ and $W_{\tau'}$ $(\tau < \tau')$ in an epoch, we can bound the change of model parameters by

$$\begin{aligned}\|W_\tau - W_{\tau'}\|_F &\leq \sum_{z=\tau}^{\tau'-1} \|W_z - W_{z+1}\|_F = \sum_{z=\tau}^{\tau'-1} \|\eta \nabla \tilde{\mathcal{L}}(W_z)\|_F \\ &\leq (\tau' - \tau)\eta G \leq nG\eta,\end{aligned} \tag{29}$$

where $\eta$ is the learning rate.

## A.3 Bounding the Staleness

According to the lemma 1 in [12], the staleness of cached embeddings can be bounded by a constant multiplied by the change of parameters. Moreover, we have already shown in Eq.29 that the parameter change $\|W_i - W_j\|_F \leq nG\eta$. Therefore, for $l = 1, \ldots, L$, there exists a constant $k_s$ such that:

$$\begin{aligned}&\|h_i^l(t) - \bar{h}_i^l\|_2 \leq k_s\eta, \quad \|h_i^l(t) - \tilde{h}_i^l(t)\|_2 \leq k_s\eta \\ &\|z_i^l(t) - \bar{z}_i^l\|_2 \leq k_s\eta, \quad \|z_i^l(t) - \tilde{z}_i^l(t)\|_2 \leq k_s\eta,\end{aligned} \tag{30}$$

where $h_i^l(t)$ is the exact embedding, $\tilde{h}_i^l(t)$ is the approximated embedding computed by Caerus, $\bar{h}_i^l$ is the cached embedding, and $z_i^l(t), \tilde{z}_i^l(t), \bar{z}_i^l$ denote the corresponding features before activation. In the following analysis, we denote

$$\epsilon = k_s\eta \tag{31}$$

as the upper bound on the staleness and approximation error for notation convention.

## A.4 Bounding the Gradient Bias

**Bounded Frobenius Norm.** Assume that the weight matrices, adjacency matrices, embedding matrices, and gradient matrices have bounded Frobenius norm, i.e., there exists a constant $S$ so that

$$\begin{aligned}&\|P^l\|_F \leq S, \quad \|W^l\|_F \leq S, \quad \|\tilde{H}^l\|_F \leq S, \quad \|\sigma'(\tilde{Z}^l)\|_F \leq S, \\ &\|\nabla_{H^l} f_i(t)\|_F \leq S, \quad \|\nabla_{Z^l} f_i(t)\|_F \leq S \quad \|\nabla_{\tilde{Z}^l} \tilde{f}_i(t)\|_F \leq S.\end{aligned} \tag{32}$$

Note that bounded norm is a common assumption and widely used in the convergence analysis [12, 45, 64].

**Continuous and Smoothness Properties.** Let the loss function $f(h^L, y)$ be $\lambda_{loss}$-Lipschitz continuous and $\rho_{loss}$-smooth with respect to the temporal embedding $h^L$, i.e., $|f(h^L, y) - f(\tilde{h}^L, y)| \leq \|h^L - \tilde{h}^L\|_2$ and $\|\nabla_{h^L} f(h^L, y) - \nabla_{\tilde{h}^L} f(\tilde{h}^L, y)\|_2 \leq \rho_{loss}\|h^L - \tilde{h}^L\|_2$. Moreover, assume that the activation function $\sigma(\cdot)$ is $\rho_{act}$-smooth, i.e., $\|\sigma'(Z) - \sigma(\tilde{Z})\|_F \leq \rho_{act}\|Z - \tilde{Z}\|_F$.

**Bounding $\mathbb{E}\|\nabla_{Z^L} f_i(t) - \nabla_{\tilde{Z}^L} \tilde{f}_i(t)\|_F$.** We prove by induction that there exists a constant $k_Z^l$ $(l = 1, \ldots, L)$ such that

$$\mathbb{E}\|\nabla_{Z^l} f_i(t) - \nabla_{\tilde{Z}^l} \tilde{f}_i(t)\|_F \leq k_Z^l \epsilon. \tag{33}$$

We start from the case $l = L$. Since the loss function $f(h^L, y)$ is $\rho_{loss}$-smooth, then

$$\begin{aligned}&\|\nabla_{H^L} f_i(t) - \nabla_{\tilde{H}^L} \tilde{f}_i(t)\|_F \\ =&\|\nabla_{h_i^L(t)} f(h_i^L(t), y_i(t)) - \nabla_{\tilde{h}_i^L(t)} f(\tilde{h}_i^L(t), y_i(t))\|_2 \\ \leq&\rho_{loss}\|h_i^L(t) - \tilde{h}_i^L(t)\|_2 \leq \rho_{loss}\epsilon.\end{aligned} \tag{34}$$

Besides, by Eq.30 and Eq.31,

$$\|Z^L - \tilde{Z}^L\|_F = \|z_i^L(t) - \tilde{z}_i^L(t)\|_2 \leq \epsilon. \tag{35}$$

Then, we have

$$\begin{aligned}&\|\nabla_{Z^L} f_i(t) - \nabla_{\tilde{Z}^L} \tilde{f}_i(t)\|_F \\ =&\|\sigma'(Z^L) \circ \nabla_{H^L} f_i(t) - \sigma'(\tilde{Z}^L) \circ \nabla_{\tilde{H}^L} \tilde{f}_i(t)\|_F \\ \leq&\|\sigma'(Z^L) \circ \nabla_{H^L} f_i(t) - \sigma'(\tilde{Z}^L) \circ \nabla_{H^L} f_i(t)\|_F \\ &+ \|\sigma'(\tilde{Z}^L) \circ \nabla_{H^L} f_i(t) - \sigma'(\tilde{Z}^L) \circ \nabla_{\tilde{H}^L} \tilde{f}_i(t)\|_F \\ \leq&\|\sigma'(Z^L) - \sigma'(\tilde{Z}^L)\|_F \cdot \|\nabla_{H^L} f_i(t)\|_F \\ +&\|\sigma'(\tilde{Z}^L)\|_F \cdot \|\nabla_{H^L} f_i(t) - \nabla_{\tilde{H}^L} \tilde{f}_i(t)\|_F \\ \leq&S\rho_{act}\epsilon + S\rho_{loss}\epsilon \\ \leq&S(\rho_{act} + \rho_{loss})\epsilon = k_Z^L \epsilon.\end{aligned} \tag{36}$$

Hence, Eq.33 holds for $l = L$, where $k_Z^L = S(\rho_{act} + \rho_{loss})$. If the statement holds for $l + 1$, we get

$$\begin{aligned}&\|\nabla_{Z^l} f_i(t) - \nabla_{\tilde{Z}^l} \tilde{f}_i(t)\|_F \\ \leq&\|\sigma'(Z^l) \circ (P^{l+1})^T \nabla_{Z^{l+1}} f_i(t) (W^{l+1})^T \\ &- \sigma'(\tilde{Z}^l) \circ (\tilde{P}^{l+1})^T \nabla_{\tilde{Z}^{l+1}} \tilde{f}_i(t) (W^{l+1})^T\|_F \\ \leq&\|\sigma'(Z^l) \circ (P^{l+1})^T \nabla_{Z^{l+1}} f_i(t) \\ &- \sigma'(\tilde{Z}^l) \circ (\tilde{P}^{l+1})^T \nabla_{\tilde{Z}^{l+1}} \tilde{f}_i(t)\|_F \cdot \|(W^{l+1})^T\|_F \\ \leq&S(\|(\sigma'(Z^l) - \sigma'(\tilde{Z}^l)) \circ (P^{l+1})^T \nabla_{Z^{l+1}} f_i(t)\|_F \\ &+ \|\sigma'(\tilde{Z}^l) \circ (P^{l+1})^T (\nabla_{Z^{l+1}} f_i(t) - \nabla_{\tilde{Z}^{l+1}} \tilde{f}_i(t))\|_F \\ &+ \|\sigma'(\tilde{Z}^l) \circ (P^{l+1} - \tilde{P}^{l+1})^T \nabla_{\tilde{Z}^{l+1}} \tilde{f}_i(t)\|_F).\end{aligned} \tag{37}$$

Note that the randomness in Eq.37 comes from the adjacency matrix $\tilde{P}^{l+1}$. It captures which embeddings are newly computed. Assume that the distribution of nodes whose embeddings are newly computed by Caerus follows the distribution of nodes whose embeddings are computed by the vanilla T-GNN, i.e., $\mathbb{E}[\tilde{P}^{l+1}] = P^{l+1}$. Then,

$$\begin{aligned}
&\mathbb{E}\|\nabla_{Z^l} f_i(t) - \nabla_{\tilde{Z}^l}\tilde{f}_i(t)\|_F \\
\leq& S\cdot\mathbb{E}[\|\sigma'(Z^l)-\sigma'(\tilde{Z}^l)\|_F\cdot\|(P^{l+1})^T\|_F\cdot\|\nabla_{Z^{l+1}}f_i(t)\|_F \\
&+\|\sigma'(\tilde{Z}^l)\|_F\cdot\|(P^{l+1})^T\|_F\cdot\|\nabla_{Z^{l+1}}f_i(t)-\nabla_{\tilde{Z}^{l+1}}\tilde{f}_i(t)\|_F \\
&+\|\sigma'(\tilde{Z}^l)\|_F\cdot\|(P^{l+1}-\tilde{P}^{l+1})^T\|_F\cdot\|\nabla_{\tilde{Z}^{l+1}}\tilde{f}_i(t)\|_F] \\
\leq& S(\rho_{act}\epsilon\sqrt{\text{row}(Z^l)}\cdot S^2 + S^2\cdot k_Z^{l+1}\epsilon + 0) \\
=& S^3(\rho_{act}\sqrt{\text{row}(Z^l)} + k_Z^{l+1})\epsilon = k_Z^l\epsilon,
\end{aligned} \tag{38}$$

where $k_Z^l = S^3(\rho_{act}\sqrt{\text{row}(Z^l)} + k_Z^{l+1})$. Eq.33 holds by induction.

**Bounding** $\mathbb{E}\|\nabla_{W^l} f_i(t) - \nabla_{W^l}\tilde{f}_i(t)\|_F$**.** We next bound the bias in partial dervatives with respect to weight matrices $W^l$:

$$\begin{aligned}
&\mathbb{E}\|\nabla_{W^l} f_i(t) - \nabla_{W^l}\tilde{f}_i(t)\|_F \\
=&\mathbb{E}\|(P^lH^{l-1})^T\nabla_{Z^l}f_i(t) - (\tilde{P}^l(\tilde{H}^{l-1}-\bar{H}^{l-1}) + P^l\bar{H}^{l-1})^T\nabla_{\tilde{Z}^l}\tilde{f}_i(t)\|_F \\
\leq&\mathbb{E}\|((P^lH^{l-1})^T - (\tilde{P}^l(\tilde{H}^{l-1}-\bar{H}^{l-1}) + P^l\bar{H}^{l-1})^T\nabla_{Z^l}f_i(t)\|_F \\
&+\mathbb{E}\|(\tilde{P}^l(\tilde{H}^{l-1}-\bar{H}^{l-1}) + P^l\bar{H}^{l-1})^T(\nabla_{Z^l}f_i(t) - \nabla_{\tilde{Z}^l}\tilde{f}_i(t))\|_F \\
\leq&\mathbb{E}\|(P^lH^{l-1})^T - (\tilde{P}^l(\tilde{H}^{l-1}-\bar{H}^{l-1}) + P^l\bar{H}^{l-1})^T\|_F\cdot S \\
&+\mathbb{E}\|(\tilde{P}^l(\tilde{H}^{l-1}-\bar{H}^{l-1}) + P^l\bar{H}^{l-1})^T\|_F\cdot k_Z^l\epsilon \\
\leq& S\sqrt{\text{row}(H^{l-1})}\cdot\epsilon\cdot S + S^2\cdot k_Z^l\epsilon \\
\leq& S^2(\sqrt{\text{row}(H^{l-1})} + k_Z^l)\epsilon = k_W^l\epsilon,
\end{aligned} \tag{39}$$

where $k_W^l = S^2(\sqrt{\text{row}(H^{l-1})} + k_Z^l)$.

**Bounding** $\mathbb{E}\|\nabla\mathcal{L}(W) - \nabla\tilde{\mathcal{L}}(W)\|_F$**.** Finally, we bound the difference between the approximated gradient obtained by Caerus and the exact gradient obtained by the vanilla T-GNN:

$$\begin{aligned}
&\mathbb{E}\|\nabla\mathcal{L}(W) - \nabla\tilde{\mathcal{L}}(W)\|_F \\
\leq&\frac{1}{|B|}\sum_{l=1}^{L}\mathbb{E}\|\nabla_{W^l}f_i(t) - \nabla_{W^l}\tilde{f}_i(t)\|_F \\
\leq&\frac{1}{|B|}\sum_{l=1}^{L}k_W^l\epsilon = k_W\epsilon,
\end{aligned} \tag{40}$$

where $k_W = \frac{1}{|B|}\sum_{l=1}^{L}k_W^l$, and the second inequality follows Eq.39.

## A.5 Convergence Guarantee

We next establish the convergence guarantees for Caerus using properties like smoothness and Lipschitz continuous. Assume that the loss function $\mathcal{L}(W)$ is $\rho$-smooth, i.e.,

$$|\mathcal{L}(W_2) - \mathcal{L}(W_1) - \langle\nabla\mathcal{L}(W_1), W_2 - W_1\rangle| \leq \frac{\rho}{2}\|W_2 - W_1\|_F^2. \tag{41}$$

Let $\delta_i = \nabla\tilde{\mathcal{L}}(W_i) - \nabla\mathcal{L}(W_i)$. By the smoothness property, we have

$$\begin{aligned}
\mathcal{L}(W_{i+1}) &\leq \mathcal{L}(W_i) + \langle\nabla\mathcal{L}(W_i), W_{i+1} - W_i\rangle + \frac{\rho}{2}\eta^2\|W_{i+1} - W_i\|_F^2 \\
&= \mathcal{L}(W_i) - \eta\langle\nabla\mathcal{L}(W_i), \nabla\tilde{\mathcal{L}}(W_i)\rangle + \frac{\rho}{2}\eta^2\|\nabla\tilde{\mathcal{L}}(W_i)\|_F^2 \\
&= \mathcal{L}(W_i) - \eta\langle\nabla\mathcal{L}(W_i), \delta_i + \nabla\mathcal{L}(W_i)\rangle + \frac{\rho}{2}\eta^2\|\delta_i + \nabla\mathcal{L}(W_i)\|_F^2 \\
&= \mathcal{L}(W_i) - (\eta - \rho\eta^2)\langle\nabla\mathcal{L}(W_i), \delta_i\rangle \\
&\quad - (\eta - \frac{\rho\eta^2}{2})\|\nabla\mathcal{L}(W_i)\|_F^2 + \frac{\rho}{2}\eta^2\|\delta_i\|_F^2.
\end{aligned} \tag{42}$$

Let the Frobenius norm of gradients be bounded by a constant, i.e., $\|\nabla\mathcal{L}(W_i)\|_F \leq G$ and $\|\nabla\tilde{\mathcal{L}}(W_i)\|_F \leq G$. By Eq.31 and Eq.40,

$$\mathbb{E}\langle\nabla\mathcal{L}(W_i), \delta_i\rangle \leq \|\nabla\mathcal{L}(W_i)\|_F\cdot\mathbb{E}\|\delta_i\|_F \leq Gk_Wk_s\eta = k_1\eta, \tag{43}$$

$$\mathbb{E}\|\delta_i\|_F^2 \leq \|\nabla\tilde{\mathcal{L}}(W)\|_F^2 + \nabla\mathcal{L}(W)\|_F^2 \leq 2G^2 = k_2, \tag{44}$$

where $k_1 = Gk_Wk_s$. Plug Eq.43 and Eq.44 into Eq.42, and sum up the loss values, then

$$\begin{aligned}
&(\eta - \frac{\rho\eta^2}{2})\sum_{i=1}^{T}\mathbb{E}\|\nabla\mathcal{L}(W_{i-1})\|_F^2 \\
\leq&\mathcal{L}(W_0) - \mathcal{L}(W^*) + k_1T(\eta - \rho\eta^2)\eta + \frac{k_2T\rho\eta^2}{2},
\end{aligned} \tag{45}$$

where $W^*$ denotes a set of local optimal parameters, and $T$ is the number of training iterations. Furthermore, taking the learning rate $\eta = \min(\frac{1}{\rho}, \frac{C}{\sqrt{T}})$, then we get

$$\begin{aligned}
&\frac{1}{T}\sum_{i=1}^{T}\mathbb{E}\|\nabla\mathcal{L}(W_{i-1})\|_F^2 \\
\leq&2\frac{\mathcal{L}(W_0) - \mathcal{L}(W^*) + k_1T(\eta - \rho\eta^2)\eta + \frac{k_2T\rho\eta^2}{2}}{T\eta(2 - \rho\eta)} \\
\leq&2\frac{\mathcal{L}(W_0) - \mathcal{L}(W^*) + k_1T(\eta - \rho\eta^2)\eta + \frac{k_2T\rho\eta^2}{2}}{T\eta} \\
\leq&\frac{2(\mathcal{L}(W_0) - \mathcal{L}(W^*))}{T\eta} + 2k_1(\eta - \rho\eta^2) + k_2\rho\eta.
\end{aligned} \tag{46}$$

We next discuss the value of $C$ in two cases. If $\sqrt{T} > \rho$, we set $C = 1$. Hence, $\eta = \frac{1}{\sqrt{T}}$, and

$$\begin{aligned}
&\frac{1}{T}\sum_{i=1}^{T}\mathbb{E}\|\nabla\mathcal{L}(W_{i-1})\|_F^2 \\
\leq&\frac{2(\mathcal{L}(W_0) - \mathcal{L}(W^*))}{\sqrt{T}} + \frac{2k_1}{\sqrt{T}} + \frac{k_2\rho}{\sqrt{T}} \\
\leq&2\frac{\mathcal{L}(W_0) - \mathcal{L}(W^*) + k_1 + k_2\rho}{\sqrt{T}}.
\end{aligned} \tag{47}$$

If $\sqrt{T} < \rho$, we set $C = \frac{\sqrt{T}}{\rho}$. It is clear that $\frac{1}{\rho} \leq C \leq 1$. Hence,

$$\begin{aligned}
&\frac{1}{T}\sum_{i=1}^{T}\mathbb{E}\|\nabla\mathcal{L}(W_{i-1})\|_F^2 \\
\leq&\frac{2(\mathcal{L}(W_0) - \mathcal{L}(W^*))}{C\sqrt{T}} + \frac{2k_1C}{\sqrt{T}} + \frac{k_2\rho C}{\sqrt{T}} \\
\leq&2\frac{\rho(\mathcal{L}(W_0) - \mathcal{L}(W^*)) + k_1 + k_2\rho}{\sqrt{T}}.
\end{aligned} \tag{48}$$

Based on Eq.47 and Eq.48, we have

$$\frac{1}{T}\sum_{i=1}^{T}\mathbb{E}\|\nabla\mathcal{L}(W_{i-1})\|_F^2 = O(1/\sqrt{T}) \tag{49}$$

if we set the learning rate $\eta = \min(\frac{1}{\rho}, \frac{1}{\sqrt{T}})$. Then, the proof of Theorem 5 is completed.

# B ABLATION STUDY

## B.1 Comparison in Model Accuracy

Table 8 and Table 9 compare our methods with all the baselines in transductive accuracy (%), inductive accuracy (%), and convergence time (seconds). It is clear that Caerus-P and Caerus-R can match or outperform the best performing baseline TGN in both transductive accuracy and inductive accuracy. In contrast, CAW, TGAT, and APAN fail to achieve comparable model accuracy.

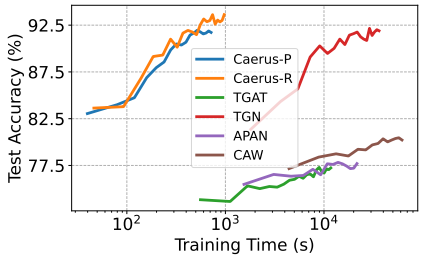



**Figure 8: Comparison of convergence time on AskUbuntu.**

## B.2 Comparison in Convergence Rate

Figure 8 gives a more intuitive comparison of convergence time on the AskUbuntu dataset. Table 8, Table 9, and Figure 8 demonstrate that Caerus-P and Caerus-R do not have a noticeable impact on the number of epochs required for models to converge. On the other hand, the per-epoch training time of Caerus-P and Caerus-R is much shorter than the other baselines. Therefore, Caerus-P and Caerus-P converge one or two orders of magnitude faster.

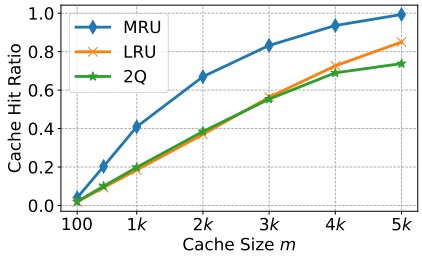



**Figure 9: Comparison of caching replacement policies on a synthetic graph with uniform degree distribution.**

## B.3 Impact of Graph Topology

Figure 9 demonstrates the cache hit ratios obtained by various caching policies on the synthetic graph with uniform degree distribution (i.e., the Stationary dataset shown in Table 7). Compared to scale-free graphs like Wikipedia and Reddit, it generally takes a larger cache size to achieve the same cache hit ratio on such uniform degree distribution. This is because there is no apparent temporal locality on such uniform distributions. However, our MRD algorithm still clearly leads to more cache hits than the classical cache replacement algorithms LRU and 2Q.

**Table 8: Comparison of 2-layer models on small dynamic graphs. We report the performance in transductive accuracy (%), inductive accuracy (%), and convergence time (s). Besides, we report the number of epochs required for model convergence in parentheses. The best and second-best results are marked in bold and underlined, respectively.**

| Model | Wikipedia | | | Reddit | | | MOOC | | |
|---|---|---|---|---|---|---|---|---|---|
| | Trans Acc | Induct Acc | Convergence (#) | Trans Acc | Induct Acc | Convergence (#) | Trans Acc | Induct Acc | Convergence (#) |
| JODIE | 86.71 | 84.35 | 2356.4 (18) | 91.87 | 90.93 | 9243.6 (14) | 82.91 | 82.26 | 5160.3 (15) |
| TGAT | 86.32 | 83.74 | 2881.4 (29) | 92.19 | 89.24 | 8524.2 (15) | 67.86 | 66.94 | 7838.5 (25) |
| TGN | 96.37 | 94.75 | 1846.9 (22) | **97.68** | **96.32** | 8868.4 (20) | 91.76 | 91.13 | 5086.83 (21) |
| APAN | 90.93 | 89.90 | 1605.0 (21) | 96.18 | 95.04 | 16431.8 (18) | 87.81 | 86.97 | 3374.1 (14) |
| CAW | 93.05 | 92.89 | 10175.5 (13) | 93.61 | 92.43 | 75585.1 (16) | 72.25 | 71.84 | 34063.9 (14) |
| Caerus-P | **96.46** | **94.85** | **148.81 (23)** | 97.59 | 96.27 | **597.2 (20)** | **92.07** | **91.52** | **303.1 (18)** |

**Table 9: Comparison of 2-layer models on large dynamic graphs. We report the performance in transductive accuracy (%), inductive accuracy (%), and convergence time (s). Besides, we report the number of epochs required for model convergence in parentheses. The best and second-best results are marked in bold and underlined, respectively. The Caerus-R models are implemented with cache limit $m = 1,000$. "–" denotes CUDA memory error, and "~" denotes time limit exceed such that we cannot finish one epoch of model training in 12 hours.**

| Model | AskUbuntu | | | SuperUser | | | Wiki-Talk | | |
|---|---|---|---|---|---|---|---|---|---|
| | Trans Acc | Induct Acc | Convergence (#) | Trans Acc | Induct Acc | Convergence (#) | Trans Acc | Induct Acc | Convergence (#) |
| JODIE | – | – | – | – | – | – | – | – | – |
| TGAT | 77.30 | 68.12 | 11728.5 (21) | 75.42 | 71.01 | 17129.3 (19) | 81.73 | 74.32 | 189420.8 (34) |
| TGN | 92.16 | 88.26 | 36202.6 (20) | 87.60 | 86.36 | 81747.6 (24) | – | – | – |
| APAN | 77.83 | 75.32 | 21606.1 (14) | 73.90 | 71.97 | 48724.2 (19) | ~ | ~ | ~ |
| CAW | 80.35 | 78.97 | 61903.7 (14) | 76.95 | 76.34 | 111744.8 (15) | ~ | ~ | ~ |
| Caerus-P | 91.95 | 88.73 | **715.7 (18)** | 90.84 | 88.14 | 1309.6 (21) | 94.08 | 88.80 | **8575.4 (18)** |
| Caerus-R | **93.51** | **89.66** | 972.3 (21) | **92.29** | **88.72** | **1173.5 (17)** | **94.49** | **89.58** | 9929.0 (19) |