Leveraging Temporal Graph Networks Using Module Decoupling

Or Feldman
Ben-Gurion University of the Negev
Technion - Israel Institute of Technology
orfel@post.bgu.ac.il

Chaim Baskin
Ben-Gurion University of the Negev
chaimbaskin@bgu.ac.il

Abstract

Current memory-based methods for dynamic graph learning use batch processing to efficiently handle high stream of updates. However, the use of batches introduces a phenomenon we term *missing updates*, which adversely affects the performance of memory-based models. In this work, we analyze the negative impacts of *missing updates* on dynamic graph learning models, and propose the decoupling strategy to mitigate these effects. Based on this strategy, we develop the Lightweight Decoupled Temporal Graph Network (LDTGN), a memory-based model with a minimal number of learnable parameters that deals with high frequency of updates. We validated our proposed model across diverse dynamic graph benchmarks. LDTGN surpassed the average precision of previous methods by over 20% in scenarios demanding frequent graph updates. In the vast majority of the benchmarks, LDTGN achieves better or comparable results while operating with significantly higher throughput than existing baselines. The code to replicate our experiments is available at this url.

1 Introduction

Dynamic graphs are commonly used to describe real-world dynamic systems, where the interacting elements are modeled as nodes, and the interactions between two elements are represented as edges. Each edge is usually labeled with a timestamp indicating its time of occurrence. Item recommendation on e-commerce platforms [1], friendship suggestion on social networks [2, 3], anomaly detection on communication networks [4], and traffic forecasting [5] are all practical tasks that can be modeled using dynamic graphs.

Although most real-world graph-related tasks have time-evolving data, deep learning approaches typically focus on problems described by static graphs, which do not change over time. Moreover, it has also been shown that ignoring the dynamic nature of a system by abstracting it with static graphs is suboptimal [6, 7]. A dynamic representation of a system, on the other hand, is often able to learn its time-evolving behavior [8–12].

Dynamic graph approaches are often based on discrete-time [13–15] or continuous-time [16–18] settings. In discrete-time setting, data are received as a sequence of snapshots describing the full graph structure at specific times, while in the flexible continuous-time setting, a single update on the graph can happen at any moment. The setting in which deep learning models for dynamic graphs operate at inference time can be roughly divided into the following types: streaming, deployed, and live update [19]. In this work, we focus on continuous-time dynamic graphs in the context of streaming, in which the models may be updated upon receiving new information, but cannot perform backpropagation due to the high throughput required.

Memory-based models for dynamic graphs are designed to support the assimilation of new information through graph updates during the inference phase. To do this, they manage a memory unit that represents the current state of the dynamic graph. This memory unit usually includes the current structure of the dynamic graph, data-specific information such as node and edge features, timestamps of previous updates, and learnable information computed by the model.

In the streaming setting of continuous-time dynamic graphs, memory-based networks have to use batches to keep up with the stream of incoming updates. Using batches, these models process multiple updates and predictions in parallel. This situation introduces a new problem in which updates for the models are not being considered for the predictions inside their mutual batch. In Section 3, we formally define this undesirable phenomenon as *missing updates*. In this work, we suggest the decoupling strategy to minimizes the negative impacts of *missing updates*, while still using batches. Guided by this strategy, we develop the Lightweight Decoupled Temporal Graph Network (LDTGN) – an efficient memory-based model for dynamic graph learning that outperforms most established baselines both in terms of running time and performance.

To summarize, this work makes the following contributions:

- We introduce and analyze the problem of *missing updates* in memory-based models.
- We suggest the decoupling methodology for building memory-based models for dynamic graph learning.
- Based on the suggested methodology, we propose LDTGN, a new lightweight model for dynamic graph learning tasks that can operate at high streaming rates and with a significantly smaller number of parameters compared to other baselines.
- We evaluate LDTGN on various transductive and inductive benchmarks for dynamic graph learning. LDTGN achieves better or comparable performance on the tested benchmarks and outperforms previous methods in terms of throughput.

2 Background

Static graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a tuple of nodes $\mathcal{V} = \{1, ..., \boldsymbol{n}\}$ and edges $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. $\mathcal{G}$ is often equipped with two feature functions: $F_{\mathcal{V}}$ and $F_{\mathcal{E}}$ that map a node or an edge into a vector representing their matching features. Continuous-Time Dynamic Graph (CTDG) is a sequence $\mathcal{Q} = \{u_{t_1}, u_{t_2}, ..., u_{t_m}\}$ of m timestamped updates on the graph. An update u_t that occurs at time t can be one of the following: node addition, node removal, edge addition, and edge removal. In CTDG, the features of nodes and edges can change over time. Hence, feature functions also take timestamp as an input, where the functions' output represents the features at that timestamp. $\mathcal{G}[\mathcal{Q}(t)]$ is the static graph received by applying all the updates from $\mathcal{Q}$ on $\mathcal{G}$ that occur until time t. The k-hop neighborhood of a node v_i at time t is defined by:

$$\mathcal{N}_i^0(t) = \{v_i\} \tag{1}$$

$$\mathcal{N}_i^k(t) = \{v_j | v_u \in \mathcal{N}_i^{k-1}(t), (v_u, v_j) \in \mathcal{G}[\mathcal{Q}(t)]\} \tag{2}$$

As a result of the growing interest in CTDGs that update frequently, several techniques have been recently developed [7, 16, 18, 20–22]. Many of these methods are specific instances of the Temporal Graph Network (TGN) model [6]. TGN is a general memory-based network designed to learn on CTDGs while achieving throughput suitable for streaming tasks. The primary concept of TGN is to maintain node features, or states, that are updated with each change to the graph. To achieve this, TGN utilizes two central modules: memory and prediction.

Memory module. The memory module is responsible for applying the updates on the graph and update the states accordingly. When a new batch of updates arrives, the memory module applies a message function that generates a vector for each node involved in each update. For example, the model would generate the following message vectors upon receiving the update of adding the edge $e_{i,j}$ from node v_i to node v_j at time t:

$$m_i(t) = \mathrm{msg_s}(s_i(t^-), s_j(t^-), \Delta t_i, F_{\mathcal{E}}(e_{i,j}, t)) \tag{3}$$

$$m_j(t) = \mathrm{msg_d}(s_j(t^-), s_i(t), \Delta t_j, F_{\mathcal{E}}(e_{i,j}, t)) \tag{4}$$

where $s_u(t^-)$ is the state of v_u prior to t and Δt_u is the time elapsed since v_u received an update. $\mathrm{msg_s}$ and $\mathrm{msg_d}$ may have learnable parameters. Then, all the messages in the batch are aggregated into a single message per node, s.t., if node v_i is involved in updates at $t_1, t_2, ..., t_n$ where $t_1 \leq t_2 \leq ... \leq t_n$ then:

$$\overline{m}_i(t_n) = \mathrm{agg}(m_i(t_1), m_i(t_2), ..., m_i(t_n)) \tag{5}$$

The aggregation function, for example, can consider only $m_i(t_n)$ and neglect any previous messages in the batch. Finally, the messages are used to update the state of the nodes:

$$s_i(t) = \mathrm{mem}(\overline{m}_i(t), s_i(t^-)) \tag{6}$$

The mem function is a memory-based neural network such as LSTM [23] or GRU [24].

Prediction module. The prediction module computes the model's predictions based on the given inputs and the current task. For example, the prediction module calculates the probability of edge existence between two nodes at a specified future time in future edge prediction tasks. The prediction process starts with reading from the memory module the states of all the nodes in the neighborhood of each node in the input. Then, the prediction module generates new embedding for each input node based on its state and the states of its neighbors. Let $[\cdot||\cdot]$ denote the operation of vector concatenation. The embedding formulation based on the 1-hop neighborhood of a node v_i is:

$$z_i(t) = \Sigma_{v_j \in \mathcal{N}_i^1(t)} h(v_i(t), v_j(t), F_{\mathcal{E}}(e_{i,j})) \tag{7}$$

where $v_u(t) = [F_{\mathcal{V}}(v_u, t)||s_u(t^-)||\Delta t_u]$ and h is a learnable function. Using the neighborhood of a node to compute its embedding mitigates the staleness problem [25]. For the task of predicting the future existence of an edge $e_{i,j}$ at time t, TGN computes the edge's probability to exist by:

$$p_{i,j}(t) = \mathrm{merge}(z_i(t), z_j(t)) \tag{8}$$

where merge is a learnable function, such as an MLP.

3 Problem statement

Batches used by memory-based models contain both updates to the graph and input queries for the model to predict. This mechanism allows memory-based models to achieve a reasonable throughput during inference time [6, 18, 20, 22]. In the streaming setting, where the graph receives new updates at extremely high frequency, it is crucial for the model to maintain sufficient throughput; otherwise, the buffer containing the new inputs and updates may overflow.

Memory-based models have a well-defined flow of operation upon receiving a new batch. First, they compute their predictions for the inputs in the batch. This operation is performed in parallel by using the current states and the current graph structure as saved in their memory. Next, they process all the updates in the batch and update their inner memory accordingly. This flow of operations introduces the undesirable phenomenon we call *missing updates*.

Formally, given a batch $\mathcal{B} = \{x_{t_1}, x_{t_2} ... x_{t_m}\}$ of size m where x_{t_i} can be either an update to the graph u_{t_i} or an input query q_{t_i}, e.g., whether there exists an interaction between nodes v_k and v_j at time t_i. We say that u_{t_i} is a *missing update* if there exists an input query q_{t_j} such that $i < j$ and at least one of the nodes in u_{t_i} is in the neighborhood of the nodes in q_{t_j}. Inputs to the model that depend on *missing updates* are harder to predict, as their states and their neighbors' states are outdated at the time of the prediction.

3.1 Empirical analysis

We examined the incidence and impact of the *missing updates* phenomenon across various real-world datasets for dynamic graphs. We measured the average ratio of inputs in a batch that depend on at least one *missing update*. In addition, we calculated the average number of *missing updates* in a single batch. We conducted both measurements for various batch sizes. In both cases, we focused on *missing updates* within the 1-hop neighborhood of the input nodes and report the results in Figure 1(a) and Figure 1(b), respectively. We also tested the TGN model trained on these datasets with the examined batch sizes. In Figure 1(c), we report the average precision of TGN for each batch size, normalized by the average precision achieved with a batch size of 10. In Appendix A, we supply the full *missing updates* statistics for all the datasets used in this work.

In Figure 1, we can see that an increase in batch size correlates with a higher incidence of *missing updates*. Furthermore, the occurrence of *missing updates* varies across different datasets. The findings

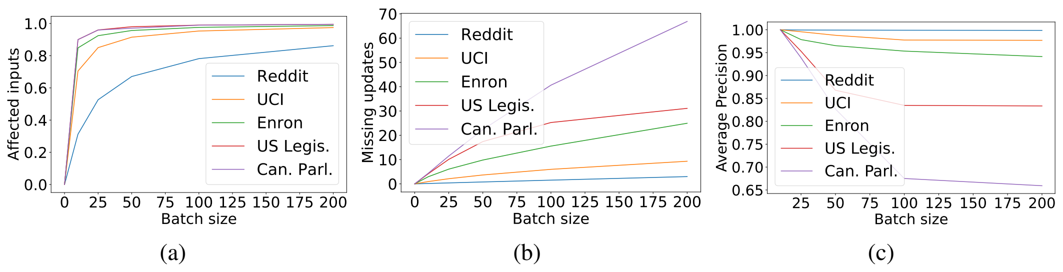

Figure 1: The incidence of *missing updates* in real-world datasets as a function of the batch size, and their impact on the performance of TGN. In (a), the ratio of inputs that depend on at least one *missing update* increases significantly as the batch size increases. In (b), the average number of *missing updates* per input increases as the batch size increases. In (c), the performance of TGN is correlated with the number of *missing updates*, where a high incidence of *missing updates* indicates a significant performance decrease.

from Figure 1 suggest a negative correlation between the occurrence of *missing updates* in a dataset and the performance of a model trained on it. Consequently, achieving optimal performance in memory-based models necessitates smaller batch sizes. This result highlights a trade-off in memory-based models: while smaller batch sizes improve performance, larger batch sizes are needed to attain high throughput in streaming scenarios.

4 Related work

Handling *missing updates*. The t-Batch algorithm [20] was initially intended to improve the running-time performance of memory-based networks for dynamic graphs that process updates one after the other (i.e., batch size equal to 1). It combines multiple updates into a single batch as long as the batches do not contain the same nodes, where the batches are temporally sorted, allowing the networks to apply the updates in parallel. Using t-Batch, JODIE's memory-based model becomes X9.2 faster than similar methods without suffering from *missing updates* [20]. The t-batch algorithm, however, suffers from two main flaws. First, large batch sizes for t-Batch are often impossible since temporal locality is a common characteristic of dynamic graphs [26]. In addition, many modern deep learning networks for dynamic graphs, such as TGN, depend on the neighborhood of the nodes to compute their predictions, causing t-Batch to perform complicated neighborhood-independent batches instead of node-independent batches, which are significantly smaller.

Efficient methods for streaming. According to Huang et al. [19], EdgeBank is currently an order of magnitude faster than other well-known methods for dynamic graph learning. EdgeBank [26] is a memorization algorithm that saves any seen update and predicts according to a simple decision rule that can be one of the following: whether the input was seen in the last few iterations or whether the input has already been seen a sufficient number of times. The algorithm's simplicity allows it to perform extremely fast, even without batches, thus not suffering from *missing updates*. Nevertheless, EdgeBank was developed to serve as a baseline for testing and comparing other methods for dynamic graphs [26], therefore, its performance lags significantly behind the state-of-the-art [19, 27].

5 Proposed method

In this section, we describe our proposed method to balance the batch size trade-off discussed earlier in Section 3. The method decouples the modules of memory-based models by ensuring that each module operates with a different batch size. In general, the memory module will utilize a smaller batch size for frequent updates, while the prediction module will employ a larger batch size for efficiency.

Next, we describe our proposed lightweight model for dynamic graph learning tasks. The model is a specific instance of TGN, but with decoupled modules that are implemented using efficient functions. Specifically, we parameterize the EdgeBank [26] model to allow it to learn. Following that, we introduce extra parameters to consider single-node information in the prediction instead of solely relying on edge temporal information.

5.1 The decoupling strategy

We propose to *decouple* the core modules of memory-based models: the prediction and memory modules. The decoupled modules will operate on different data using different batch size. Given a batch containing updates to apply and inputs to predict, the model divides the batch into smaller consecutive batches we term memory batches. The memory module operates on the memory batches, and thus, it can perform memory updates more frequently. After processing a memory batch but before proceeding to the next one, the memory module extracts and temporally saves the temporal neighborhood information. This information encompasses the neighborhood state relevant to the nodes in the subsequent memory batches, preventing it from being overridden. The neighborhood state is defined by:

$$S_{\mathcal{N}_i^k(t)}(t) = \{s_j(t)\,|v_j \in \mathcal{N}_i^k(t)\} \tag{9}$$

The saved states create views of the model's memory at each time a memory batch starts. After processing all memory batches, the prediction module retrieves the extracted states for each node associated with a given input. The states of each input are retrieved from the view that is immediately preceding its timestamp. Subsequently, the prediction module computes predictions for all inputs within the complete batch simultaneously. Note that the total memory required to save the information for the views is equal to the memory used in standard memory-based models with the same batch size.

Figure 2 demonstrates the effectiveness of a decoupled model compared to a standard memory-based model. In Figure 2, the edge at t_4 is given as an input for the models. A standard memory-based model computes the embeddings of v_3 and v_6 based on their neighborhood states before t_1 and only then updates its inner memory with the edges at t_1, t_2 and t_3. On the other hand, a decoupled model initially performs memory updates of the two memory batches. Then, its prediction module uses the states extracted before t_3 that include the updates in the first memory batch. The *missing update* that affects the interaction between v_3 and v_6 is avoided by using the decoupling strategy since the prediction module is aware of the interaction between v_2 and v_3.

Decoupling the modules of memory-based models offers two immediate benefits. First, by decoupling the memory module from the prediction module and setting the memory batch size to 1, the number of *missing updates* in each batch is guaranteed to be 0. In addition, by using decoupling, we can accelerate the execution time of an existing model without compromising its accuracy. This is achieved by decoupling its modules, setting the memory batch size to match the model's original batch size, and increasing the batch size of the prediction module substantially. Using the original batch size for the memory batches ensures the same frequency of *missing updates*, and the new larger batch size improves the runtime performance. Figure 3 illustrates the running time improvement of decoupled TGN with a memory batch size of 50 when using growing batch sizes. Notably, the decoupling strategy enhances TGN's running time by 12.5% without compromising its performance, as the frequency of *missing updates* depends only on the memory batch size. Furthermore, transferring additional computations from the memory module to the prediction module will lead to an additional improvement in running time. Further analysis of the potential speedup of the decoupling strategy is discussed in Appendix E.

5.2 Lightweight Decoupled Temporal Graph Network

We propose the Lightweight Decoupled Temporal Graph Network (LDTGN), an efficient model designed for dynamic graph learning tasks. LDTGN operates with high throughput, crucial for the streaming setting, while also achieving superior performance in dynamic graph learning tasks.

We develop LDTGN by enhancing EdgeBank and incorporating the decoupling strategy. Although EdgeBank's performance falls short of the current state-of-the-art, it achieves commendable results with exceptionally high throughput, making it a suitable foundation for our model. We construct LDTGN by finding the deficiencies in EdgeBank that need addressing to attain top-tier performance, and integrating improvements to resolve these issues effectively. In this subsection, we describe LDTGN for future edge prediction tasks and assume only edge addition updates. Comprehensive

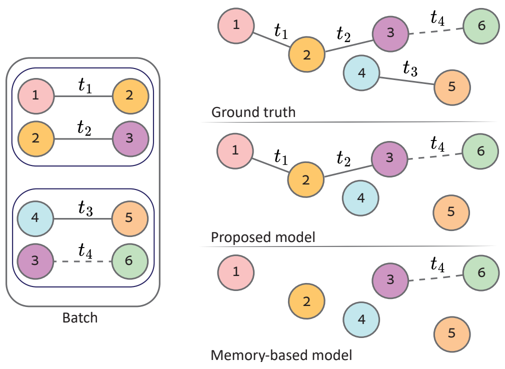

Figure 2: Illustration of a dynamic graph at t_4 for the task of predicting the edge (v_3, v_6). The state of a memory-based model is compared to the state of a model operating using the proposed decoupling strategy. The memory-based model was updated prior to t_1 and, therefore, does not contain (v_1, v_2), (v_2, v_3) and (v_4, v_5). The model that follows the decoupling strategy and applies inner memory batch updates was previously updated at t_2 and, therefore, closely resembles the ground truth and is missing only (v_4, v_5).

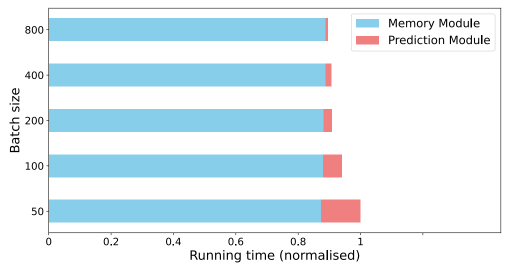

Figure 3: Comparison of running times for decoupled TGN with a constant memory batch size of 50 and varying batch sizes, tested on the Wikipedia dataset. The running times are normalized by the baseline scenario where both the memory batch size and the batch size are set to 50.

details about applying node addition, node removal, and edge removal updates, as well as adjustments for node classification tasks, are provided in Appendix C.

Poursafaei et al. [26] formulated EdgeBank as a memory-based algorithm that stores the last T updates. EdgeBank predicts 'positive' only for edges in the memory and 'negative' for any other edge. We can also describe this memory-based prediction rule as a linear function that maps a time-based difference into a prediction. Equation (10) details the linear function of EdgeBank with a decision function that predicts 'positive' for an edge $e_{i,j}$, only if it was updated in the last T updates.

$$p_{i,j}(t) = -(t - t_{i,j}) + T \tag{10}$$

In Equation (10), $t_{i,j}$ is the last time the edge $e_{i,j}$ received an update, and t is the current time. $t_{i,j}$ is set to 0 if $e_{i,j}$ has not been received yet. Poursafaei et al. [26] suggested to use a constant value of 1000 for T. The equation should be parameterized to allow the model to learn the most appropriate value of T for every dataset. To do this, we add a bias b and a coefficient w as detailed in Equation (11).

$$p_{i,j}(t) = (t - t_{i,j})w + b \tag{11}$$

Using Equation (11), we can learn the appropriate threshold for each task. As in EdgeBank, this function does not incorporate the nodes themselves into the prediction. We solve it by adding the time differences of each node in the potential edge and appropriate coefficients as detailed in Equation (12).

$$p_{i,j}(t) = (t - t_{i,j})w_1 + (t - t_i)w_2 + (t - t_j)w_3 + b \tag{12}$$

In Equation (12), t_i and t_j are the last times the nodes v_i and v_j received an update, respectively. Equation (12) is missing topological and data-specific information such as node and edge features. Moreover, the prediction function is linear, which often causes the learned function to be distant from the ground truth prediction function. To solve this issue, we first create embeddings for the nodes in the potential edge and a preliminary embedding for the edge itself as described in Equations (13) and (14):

$$z_i(t) = \Sigma_{k \in \mathcal{N}_i^1(t)} \alpha_k [v_i(t) || v_k(t) || F_{\mathcal{E}}(e_{i,k})] \tag{13}$$

$$z_{i,j}(t) = \text{TDE}(t - t_{i,j}) \tag{14}$$

where $v_i(t) = [F_{\mathcal{V}}(v_i, t) || \text{TDE}(t - t_i)]$ and α_k is attention weight computed as in GAT [28]. TDE is a non-linear time difference embedding function such as Time2Vec [29]. Equation (15) uses the

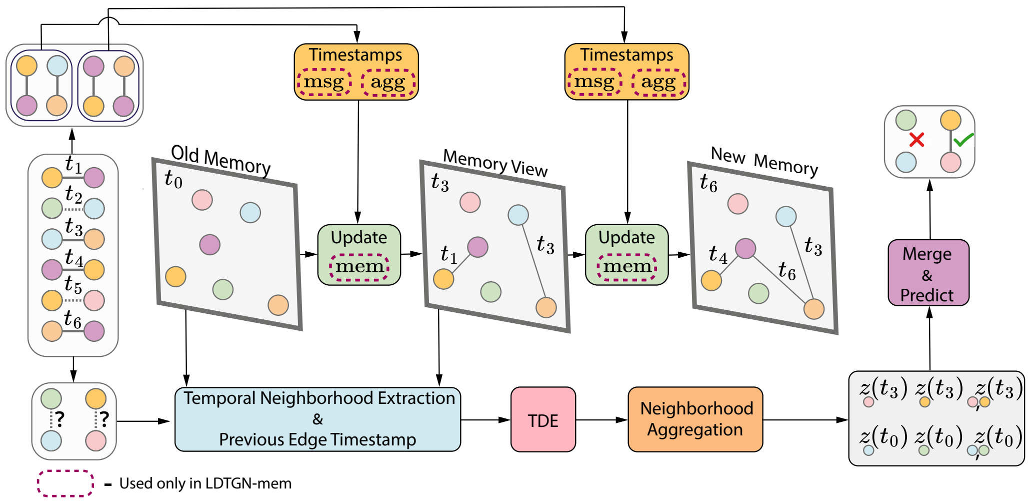

Figure 4: Framework of the proposed model. The batch of updates and inputs is first divided into memory batches and a single batch of inputs. Then, the new edges and their appropriate timestamps are saved in the memory. In LDTGN-mem, the state of each node in the memory batch is updated using the msg, agg, and mem functions. Before each update, the relevant information is saved in a memory view to prevent it from being overridden. Next, the information of each input node is extracted from the appropriate memory view. Then, TDE is applied to the time differences between the inputs and the time of the extracted timestamps. Neighborhood information is aggregated using learnable attention weights to create a single encoding for each node. Finally, the node encoding and the edge encoding are merged using the merge function, and the combined encoding is used to get the final prediction.

embeddings of the nodes, edge time difference, and a non-linear merge function to give the final prediction.

$$p_{i,j}(t) = \mathrm{merge}(z_i(t), z_j(t), z_{i,j}(t)) \tag{15}$$

Equations (13) to (15) constitute the prediction module of LDTGN. The prediction module only requires t_i, t_j and $t_{i,j}$ from the memory module, making these timestamps the sole data needed by the memory module. In the experiments, we implemented LDTGN with a memory batch size of 1, thus eliminating the adverse effects of *missing updates*. This design choice not only mitigates these negative impacts but also enables the model to process updates without the message aggregator of TGN. The message aggregator may lead to loss of information as detailed in Section 2. LDTGN operates with a minimal memory batch size and with a high throughput thanks to the removal of msg_s, msg_d and mem from the memory module. Compared to TGN, LDTGN saves states for the edges in addition to the states of the nodes. This does not add an additional memory to LDTGN over TGN, since TGN already saves the full graph to incorporate topological information in the prediction.

In the scenarios where the throughput is allowed to be smaller, and the *missing updates* negative effects are neglectable for small memory batch size, LDTGN can incorporate long-term dependencies. We refer to this variant of LDTGN as LDTGN-mem. To achieve long-term dependencies, LDTGN-mem is implemented with a heavier memory module. This memory module generates the following messages:

$$m_i(t) = [s_i(t^-)||s_j(t^-)||TDE(t - t_i)] \tag{16}$$

$$m_j(t) = [s_j(t^-)||s_i(t^-)||TDE(t - t_j)] \tag{17}$$

The aggregation function for the messages takes only the most recent message per node, and the mem function is set to be a GRU cell:

$$s_i(t) = \mathrm{GRU}(\overline{m}_i(t), s_i(t^-)) \tag{18}$$

To incorporate the long-term memory in the prediction module, LDTGN-mem adds the current learned state to the data of each node:

$$v_i(t) = [F_{\mathcal{V}}(v_i, t)||\mathrm{TDE}(t - t_i)||s_i(t^-)] \tag{19}$$

In contrast to LDTGN, LDTGN-mem has to operate with a memory batch size larger than 1 to ensure a reasonable throughput. We chose to implement LDTGN-mem with a memory batch size of 50. This is because we observed earlier in Figure 1 that the incidence of *missing updates* with a batch size of 50 is not severe. In addition, LDTGN-mem operates with an acceptable throughput when using this memory batch size, as presented later in Section 6. The adjustments required for LDTGN-mem are detailed in the illustration of our model in Figure 4.

6 Experiments

All the experiments were performed using DyGLib [27] – the unified library for dynamic graph learning evaluation. DyGLib contains various real world datasets including large-scale dynamic graphs with millions of edges. We additionally evaluated our model on the Temporal Graph Benchmark (TGB) [19]. The results for TGB are available in Appendix D. The experiments were performed for the task of future edge prediction with random negative edge sampling on the following datasets: Wikipedia, Reddit, MOOC, LastFM, Enron, Social Evo., UCI, Flights, Can. Parl., US Legis., UN Trade, UN Vote, and Contacts. The datasets were collected by Poursafaei et al. [26]. Additional information and statistics regarding the datasets can be found in Appendix A. We used seven well-known methods as baselines for the task of future edge prediction: DyRep [16], TGAT [7], TGN [6], CAWN [22], EdgeBank [26], GraphMixer [18] and DyGFormer [27]. Additional information regarding the baselines can be found in Appendix B. We adopted the approach used in previous works and split the dataset into training, validation, and test sets by performing a chronological split of 70%–15%–15%. We report the mean and standard deviation of the Average Precision (AP) on the test set. Results for Areas Under the Receiver Operating Characteristic Curve (AUC-ROC) are detailed in Appendix D.

6.1 Future edge prediction

In the first experiment, we tested transductive future edge prediction with random negative edge sampling, i.e., for each positive edge in the datasets, a negative edge with the same source and a random destination is sampled. The results for this experiment are presented in Table 1. We also performed an experiment for the inductive future edge prediction task, in which all the edges in the validation and test sets contain only nodes that have not been previously seen in the training set. The results for this experiment are reported in Table 2. The baselines' results were computed with DyGLib using the hyperparameters configurations as described in [27]. As in [27], all the models were trained and tested using a batch size of 200. Additional implementation-specific details of LDTGN and LDTGN-mem and their training methodology are detailed in Appendix C. We report in Tables 1 and 2 the best results among the LDTGN variations. The full results for each LDTGN variation are in Appendix D. LDTGN achieves better or comparable results compared to the baselines for the setting of transductive and inductive future edge prediction. In benchmarks where the negative effects of the *missing updates* are insignificant for small batch sizes, LDTGN achieve comparable performance to DyGFormer. In the benchmarks where *missing updates* have substantial influence, such as US Legis, LDTGN considerably outperforms the compared baselines since it completely removes all the *missing updates* when using a memory batch size of 1.

Table 1: AP for transductive future edge prediction with random negative sampling over five runs. The significantly best result for each benchmark appears in bold font.

Dataset	DyRep	TGAT	TGN	CAWN	EdgeBank	GraphMixer	DyGFormer	LDTGN (ours)
Wikipedia	94.86±0.06	96.94±0.06	98.45±0.06	98.76±0.03	90.37±0.00	97.25±0.03	**99.03±0.02**	**98.99±0.03**
Reddit	98.22±0.04	98.52±0.02	98.63±0.06	99.11±0.01	94.86±0.00	97.31±0.01	99.22±0.01	**99.28±0.02**
MOOC	81.97±0.49	85.84±0.15	89.15±1.60	80.15±0.25	57.97±0.00	82.78±0.15	87.52±0.49	**91.73±0.65**
LastFM	71.92±2.21	73.42±0.21	77.07±3.97	86.99±0.06	79.29±0.00	75.61±0.24	**93.00±0.12**	91.22±0.31
Enron	82.38±3.36	71.12±0.97	86.53±1.11	89.56±0.09	83.53±0.00	82.25±0.16	92.47±0.12	**98.10±0.01**
Social Evo.	88.87±0.30	93.16±0.17	93.57±0.17	84.96±0.09	74.95±0.00	93.37±0.07	94.73±0.01	**95.45±0.51**
UCI	65.14±2.30	79.63±0.70	92.34±1.04	95.18±0.06	76.20±0.00	93.25±0.57	95.79±0.17	**97.05±0.01**
Flights	95.29±0.72	94.03±0.18	97.95±0.14	98.51±0.01	89.35±0.00	90.99±0.05	**98.91±0.01**	98.76±0.06
Can. Parl.	66.54±2.76	70.73±0.72	70.88±2.34	69.82±2.34	64.55±0.00	77.04±0.46	97.36±0.45	**99.47±0.03**
US Legis.	75.34±0.39	68.52±3.16	75.99±0.58	70.58±0.48	58.39±0.00	70.74±1.02	71.11±0.59	**92.08±0.09**
UN Trade	63.21±0.93	61.47±0.18	65.03±1.37	65.39±0.12	60.41±0.00	62.61±0.27	66.46±1.29	**97.82±0.07**
UN Vote	62.81±0.80	52.21±0.98	65.72±2.17	52.84±0.10	58.49±0.00	52.11±0.16	55.55±0.42	**80.94±1.43**
Contacts	95.98±0.15	96.28±0.09	96.89±0.56	90.26±0.28	92.58±0.00	91.92±0.03	98.29±0.01	**98.78±0.04**

Table 2: AP for inductive future edge prediction with random negative sampling over five different runs. The significantly best result for each benchmark appears in bold font.

Dataset	DyRep	TGAT	TGN	CAWN	GraphMixer	DyGFormer	LDTGN (ours)
Wikipedia	92.43±0.37	96.22±0.07	97.83±0.04	98.24±0.03	96.65±0.02	98.59±0.03	**98.74±0.02**
Reddit	96.09±0.11	97.09±0.04	97.50±0.07	98.62±0.01	95.26±0.02	98.84±0.02	**98.86±0.02**
MOOC	81.07±0.44	85.50±0.19	89.04±1.17	81.42±0.24	81.41±0.21	86.96±0.43	**90.61±0.32**
LastFM	83.02±1.48	78.63±0.31	81.45±4.29	89.42±0.07	82.11±0.42	**94.23±0.09**	92.62±0.59
Enron	74.55±3.95	67.05±1.51	77.94±1.02	86.35±0.51	75.88±0.48	89.76±0.34	**96.06±0.09**
Social Evo.	90.04±0.47	91.41±0.16	90.77±0.86	79.94±0.18	91.86±0.06	93.14±0.04	**94.37±0.68**
UCI	57.48±1.87	79.54±0.48	88.12±2.05	92.73±0.06	91.19±0.42	94.54±0.12	**94.92±0.01**
Flights	92.88±0.73	88.73±0.33	95.03±0.60	97.06±0.02	83.03±0.05	**97.79±0.02**	97.31±0.16
Can. Parl.	54.02±0.76	55.18±0.79	54.10±0.93	55.80±0.69	55.91±0.82	87.74±0.71	**97.83±0.06**
US Legis.	57.28±0.71	51.00±3.11	58.63±0.37	53.17±1.20	50.71±0.76	54.28±2.87	**83.76±0.44**
UN Trade	57.02±0.69	61.03±0.18	58.31±3.15	65.24±0.21	62.17±0.31	64.55±0.62	**97.43±0.07**
UN Vote	54.62±2.22	52.24±1.46	58.85±2.51	49.94±0.45	50.68±0.44	55.93±0.39	**81.29±1.41**
Contacts	92.18±0.41	95.87±0.11	93.82±0.99	89.55±0.30	90.59±0.05	**98.03±0.02**	**97.94±0.13**

6.2 Memory and running time performance

We calculated the average number of learnable parameters required for each model to achieve its best performance and reported it in Figure 5. We also measured the average throughput at inference time for each model across all the datasets, where the throughput is defined as the number of edges the model can process in a single second. The results are shown in Figure 6. In both Figure 5 and Figure 6, LDTGN surpasses the other baselines by a large margin in terms of efficiency. The throughput of the baselines was measured using a batch size that is at least the batch size used for LDTGN; hence, the results in Figure 6 are also proportionate to the latency of LDTGN compared to the other baselines.

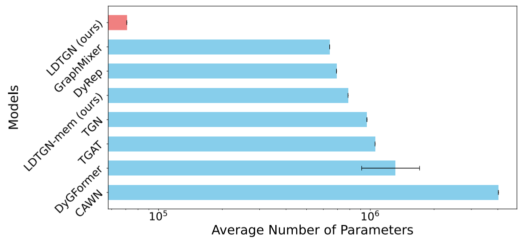

Figure 5: Average number of learnable parameters used by the baselines and our model. The black ranges indicate the standard deviation of the average number of learnable parameters.

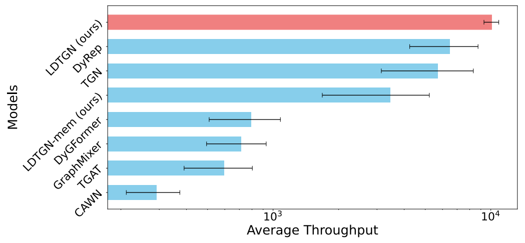

Figure 6: Average throughput (processed edges per second) of the baselines and our model. The black ranges indicate the standard deviation of the average throughput.

7 Conclusion

In this work, we introduced the *missing updates* phenomenon caused by the batching technique in memory-based models for dynamic graph learning. We showed negative correlation between the frequency of *missing updates* in datasets and the performance of the memory-based models. Since larger batch sizes lead to more *missing updates*, yet are necessary for reasonable throughput, there is a trade-off regarding the batch size in memory-based models. To balance this trade-off, we presented the decoupling strategy for designing memory-based models. Decoupling enables two types of batches – one for the memory module and the other for the prediction module. In this way, models for dynamic graphs can increase the frequency of the updates while still handling their arrival streams. In addition, we introduced LDTGN, a lightweight model for future edge prediction that is highly efficient in terms of time and memory. When feasible, LDTGN can be equipped with a heavier memory module, allowing it to better capture long-term dependencies. We also showed by extensive experiments that LDTGN has outstanding performance for both transductive and inductive tasks, achieving better or comparable performance than well-known baselines on the tested benchmarks.

References

[1] Linlin Ding, Baishuo Han, Shu Wang, Xiaoguang Li, and Baoyan Song. User-centered recommendation using us-elm based on dynamic graph model in e-commerce. *International Journal of Machine Learning and Cybernetics*, 10:693–703, 2019. 1

[2] Lars Backstrom and Jure Leskovec. Supervised random walks: predicting and recommending links in social networks. In *Proceedings of the fourth ACM international conference on Web search and data mining*, pages 635–644, 2011. 1

[3] Sogol Haghani and Mohammad Reza Keyvanpour. A systemic analysis of link prediction in social network. *Artificial Intelligence Review*, 52:1961–1995, 2019. 1

[4] Wenchao Yu, Wei Cheng, Charu C Aggarwal, Kai Zhang, Haifeng Chen, and Wei Wang. Netwalk: A flexible deep embedding approach for anomaly detection in dynamic networks. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2672–2681, 2018. 1

[5] Andrea Cini, Ivan Marisca, Filippo Maria Bianchi, and Cesare Alippi. Scalable spatiotemporal graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 37(6), pages 7218–7226, 2023. 1

[6] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal graph networks for deep learning on dynamic graphs. *CoRR*, abs/2006.10637, 2020. 1, 2, 3, 8, 16, 17

[7] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive representation learning on temporal graphs. *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30*, 2020. 1, 2, 8, 16

[8] Georg Simmel. *The sociology of georg simmel*, volume 92892. Simon and Schuster, 1950. 1

[9] Mark S Granovetter. The strength of weak ties. *American journal of sociology*, 78(6):1360–1380, 1973.

[10] Shmoolik Mangan and Uri Alon. Structure and function of the feed-forward loop network motif. *Proceedings of the National Academy of Sciences*, 100(21):11980–11985, 2003.

[11] Riitta Toivonen, Jussi M Kumpula, Jari Saramäki, Jukka-Pekka Onnela, János Kertész, and Kimmo Kaski. The role of edge weights in social networks: modelling structure and dynamics. In *Noise and Stochastics in Complex Systems and Finance*, volume 6601, pages 48–55. SPIE, 2007.

[12] Thomas E Gorochowski, Claire S Grierson, and Mario Di Bernardo. Organization of feed-forward loop motifs reveals architectural principles in natural and engineered networks. *Science advances*, 4(3):eaap9751, 2018. 1

[13] David Liben-Nowell and Jon Kleinberg. The link prediction problem for social networks. In *Proceedings of the twelfth international conference on Information and knowledge management*, pages 556–559, 2003. 1

[14] Aravind Sankar, Yanhong Wu, Liang Gou, Wei Zhang, and Hao Yang. Dysat: Deep neural representation learning on dynamic graphs via self-attention networks. In *Proceedings of the 13th international conference on web search and data mining*, pages 519–527, 2020.

[15] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao Schardl, and Charles Leiserson. Evolvegcn: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34(04), pages 5363–5370, 2020. 1

[16] Rakshit Trivedi, Mehrdad Farajtabar, Prasenjeet Biswal, and Hongyuan Zha. Dyrep: Learning representations over dynamic graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9*, 2019. 1, 2, 8, 16

[17] Yao Ma, Ziyi Guo, Zhaocun Ren, Jiliang Tang, and Dawei Yin. Streaming graph neural networks. In *Proceedings of the 43rd international ACM SIGIR conference on research and development in information retrieval*, pages 719–728, 2020.

[18] Weilin Cong, Si Zhang, Jian Kang, Baichuan Yuan, Hao Wu, Xin Zhou, Hanghang Tong, and Mehrdad Mahdavi. Do we really need complicated model architectures for temporal networks? *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5*, 2023. 1, 2, 3, 8, 16

[19] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems*, 36, 2023. 1, 4, 8

[20] Srijan Kumar, Xikun Zhang, and Jure Leskovec. Predicting dynamic embedding trajectory in temporal interaction networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1269–1278, 2019. 2, 3, 4, 13

[21] Lu Wang, Xiaofu Chang, Shuang Li, Yunfei Chu, Hui Li, Wei Zhang, Xiaofeng He, Le Song, Jingren Zhou, and Hongxia Yang. Tcl: Transformer-based dynamic graph modelling via contrastive learning. *CoRR*, abs/2105.07944, 2021.

[22] Yanbang Wang, Yen-Yu Chang, Yunyu Liu, Jure Leskovec, and Pan Li. Inductive representation learning in temporal networks via causal anonymous walks. *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7*, 2021. 2, 3, 8, 16

[23] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8): 1735–1780, 1997. 3

[24] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734, 2014. 3

[25] Seyed Mehran Kazemi, Rishab Goel, Kshitij Jain, Ivan Kobyzev, Akshay Sethi, Peter Forsyth, and Pascal Poupart. Representation learning for dynamic graphs: A survey. *The Journal of Machine Learning Research*, 21(1):2648–2720, 2020. 3

[26] Farimah Poursafaei, Shenyang Huang, Kellin Pelrine, and Reihaneh Rabbany. Towards better evaluation for dynamic link prediction. *Advances in Neural Information Processing Systems*, 35:32928–32941, 2022. 4, 6, 8, 16

[27] Le Yu, Leilei Sun, Bowen Du, and Weifeng Lv. Towards better dynamic graph learning: New architecture and unified library. *Advances in Neural Information Processing Systems*, 36: 67686–67700, 2023. 4, 8, 13, 16

[28] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. 6

[29] Seyed Mehran Kazemi, Rishab Goel, Sepehr Eghbali, Janahan Ramanan, Jaspreet Sahota, Sanjay Thakur, Stella Wu, Cathal Smyth, Pascal Poupart, and Marcus Brubaker. Time2vec: Learning a vector representation of time. *CoRR*, abs/1907.05321, 2019. 6

[30] James W Pennebaker, Martha E Francis, and Roger J Booth. Linguistic inquiry and word count: Liwc 2001. *Mahway: Lawrence Erlbaum Associates*, 71(2001):2001, 2001. 13

[31] Jitesh Shetty and Jafar Adibi. The enron email dataset database schema and brief statistical report. *Information sciences institute technical report, University of Southern California*, 4(1): 120–128, 2004. 13

[32] Anmol Madan, Manuel Cebrian, Sai Moturu, Katayoun Farrahi, et al. Sensing the" health state" of a community. *IEEE Pervasive Computing*, 11(4):36–45, 2011. 13

[33] Pietro Panzarasa, Tore Opsahl, and Kathleen M Carley. Patterns and dynamics of users' behavior and interaction: Network analysis of an online community. *Journal of the American Society for Information Science and Technology*, 60(5):911–932, 2009. 13

[34] Martin Strohmeier, Xavier Olive, Jannis Lübbe, Matthias Schäfer, and Vincent Lenders. Crowdsourced air traffic data from the opensky network 2019–2020. *Earth System Science Data*, 13 (2):357–366, 2021. 13

[35] Shenyang Huang, Yasmeen Hitti, Guillaume Rabusseau, and Reihaneh Rabbany. Laplacian change point detection for dynamic graphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 349–358, 2020. 13

[36] James H Fowler. Legislative cosponsorship networks in the us house and senate. *Social networks*, 28(4):454–465, 2006. 13

[37] Graham K MacDonald, Kate A Brauman, Shipeng Sun, Kimberly M Carlson, Emily S Cassidy, James S Gerber, and Paul C West. Rethinking agricultural trade relationships in an era of globalization. *BioScience*, 65(3):275–289, 2015. 13

[38] Erik Voeten, Anton Strezhnev, and Michael Bailey. United Nations General Assembly Voting Data. *Harvard Dataverse*, 2009. URL `https://doi.org/10.7910/DVN/LEJUQZ`. 13

[39] Piotr Sapiezynski, Arkadiusz Stopczynski, David Dreyer Lassen, and Sune Lehmann. Interaction data from the copenhagen networks study. *Scientific Data*, 6(1):315, 2019. 13

[40] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010. 17

A Datasets statistics and descriptions

In our experiments we used the following dynamic graph datasets:

• Wikipedia [20]: Wikipedia edit requests log over one month, where the editing users and Wikipedia pages are represented as nodes and the edit requests are modeled as edges. The edges are timestamped and contain LIWC feature vectors [30] of the requested text to post.

• Reddit [20]: Reddit post requests log over one month where the posting users and subreddits are represented as nodes and the posting requests are modeled as edges.

• MOOC [20]: Students' access records to MOOC online courses, where students and content units (e.g., videos, answers, etc.) are described as nodes and the access actions (viewing a video, submitting an answer, etc.) are modeled as edges. The edges are timestamped and have four features describing the action.

• LastFM [20]: LastFM listening records over one month, where the LastFM users and the songs are represented as nodes and there is an edge between the users and the songs to which they listened. The edges are timestamped and do not contain any features.

• Enron [31]: Email logs of Enron employees over a period of three years, where the employees are modeled as nodes and a single edge represents an email sent between two employees. The edges are timestamped and do not contain any features.

• Social Evo. [32]: Documentation of the everyday life of undergraduate students living in dormitories from October 2008 to May 2009. Represented as a mobile phone proximity network where each edge has two features.

• UCI [33]: Message log of the online community of students from the University of California, Irvine, where the students are modeled as nodes and a single edge represents a message sent between two students. The edges are timestamped with a granularity of seconds.

• Flights [34]: Tracked air traffic during the COVID-19 pandemic, where the airports are modeled as nodes and the edges are the tracked flights between two airports. The edges are timestamped and weighted. The weight of the edges indicates the number of flights between the airports in a day.

• Can. Parl. [35]: Documented interactions between Canadian members of parliaments from 2006 to 2019, where the members of parliaments are described as nodes, two of which are connected by an edge if they both voted "yes" on a bill. The edges are timestamped and weighted. The weight of the edges indicates the number of times that one member voted "yes" for another member's bill within one year.

• US Legis. [36]: Documented interactions in the US Senate, where legislators are modeled as nodes, and two of which are connected by an edge if they co-sponsored a bill. The edges are timestamped and weighted. The weight of the edges indicates the number of times that two members of the US Congress co-sponsored a bill in a given term.

• UN Trade [37]: Documented global food and agriculture trading connections spanning over 30 years, where nations are represented as nodes, two of which are connected by an edge if they have an agriculture import or export relations. The edges are timestamped and weighted. The weight of the edges is the sum of normalized agriculture import or export values between two countries.

• UN Vote [38]: Documentation of roll-call votes in the United Nations General Assembly from 1946 to 2020 where nations are represented as nodes, two of which are connected by an edge if they both voted "yes" for an item. The edges are timestamped and weighted. The weight of the edges is the number of times the two countries vote "yes" on a call.

• Contact [39]: Physical proximity records documenting around 700 university students over a period of four weeks, where the students are modeled as nodes. Two students are connected by an edge if they are within a close proximity to each other. The edges are timestamped and weighted. The weight of the edges specifies the physical proximity between two students.

The full statistics of the datasets as collected by Yu et al. [27] are reported in Table 3.

Table 3: Datasets statistics.

Dataset	Domain	#Nodes	#Edges	#Node Features	#Edge Features	Bipartite	Duration
Wikipedia	Social	9,227	157,474	-	172	True	1 month
Reddit	Social	10,984	672,447	-	172	True	1 month
MOOC	Interaction	7,144	411,749	-	4	True	17 months
LastFM	Interaction	1,980	1,293,103	-	–	True	1 month
Enron	Social	184	125,235	-	–	False	3 years
Social Evo.	Proximity	74	2,099,519	-	2	False	8 months
UCI	Social	1,899	59,835	-	–	False	196 days
Flights	Transport	13,169	1,927,145	-	1	False	4 months
Can. Parl.	Politics	734	74,478	-	1	False	14 years
US Legis.	Politics	225	60,396	-	1	False	12 terms
UN Trade	Economics	255	507,497	-	1	False	32 years
UN Vote	Politics	201	1,035,742	-	1	False	72 years
Contact	Proximity	692	2,426,279	-	1	False	1 month

In Table 4 we report the ratio of inputs that depend on at least a single *missing update* in their 1-hop neighborhood. In Table 5 we report the average number of *missing updates* affecting the 1-hop neighborhood of the nodes. Tables 4 and 5 contain the *missing updates* statistics for all the datasets used in this work for various batch sizes.

Table 4: Ratio of inputs that depend on at least a single *missing update* in their 1-hop neighborhood.

Dataset	1	10	25	50	100	200
Wikipedia	0	0.23	0.42	0.55	0.67	0.76
Reddit	0	0.31	0.52	0.67	0.78	0.86
MOOC	0	0.88	0.95	0.98	0.99	0.99
LastFM	0	0.74	0.88	0.94	0.97	0.98
Enron	0	0.85	0.92	0.95	0.98	0.99
Social Evo.	0	0.90	0.96	0.98	0.99	0.99
UCI	0	0.70	0.85	0.91	0.95	0.97
Flights	0	0.82	0.90	0.94	0.96	0.98
Can. Parl.	0	0.90	0.96	0.98	0.99	0.99
US Legis.	0	0.90	0.96	0.98	0.99	0.99
UN Trade	0	0.90	0.96	0.98	0.99	0.99
UN Vote	0	0.90	0.96	0.98	0.99	0.99
Contacts	0	0.86	0.94	0.97	0.98	0.99

Table 5: Average number of *missing updates* affecting the 1-hop neighborhood of each input node.

Dataset	1	10	25	50	100	200
Wikipedia	0	0.25	0.65	1.19	2.02	3.28
Reddit	0	0.15	0.39	0.78	1.53	2.99
MOOC	0	0.72	1.45	2.38	3.90	6.57
LastFM	0	0.45	1.16	2.10	3.59	5.93
Enron	0	2.95	6.07	9.81	15.55	24.95
Social Evo.	0	1.86	3.15	5.22	9.77	19.22
UCI	0	0.95	2.12	3.67	5.98	9.31
Flights	0	2.82	5.77	8.79	11.97	14.55
Can. Parl.	0	4.41	11.46	22.33	40.65	66.87
US Legis.	0	4.19	10.10	17.41	25.26	31.09
UN Trade	0	4.37	11.21	21.35	37.41	57.00
UN Vote	0	3.75	8.56	14.35	21.47	28.01
Contacts	0	1.74	2.60	3.15	3.85	5.13

A.1 Missing updates and sparse dynamic graphs

Table 6 contains all the datasets we used in our work, including datasets of large and sparse graphs from TGB (tgbl-review, tgbl-coin, and tgbl-comment). We excluded tgbl-wiki and tgbl-flight because they are based on the Wikipedia and Flights datasets, which we have already included. The table is sorted by the average degree. MUX stands for the average number of missing updates of a node in an interaction to predict using a memory-based model with a batch size of X. In general, graphs with a smaller average degree, i.e., sparse graphs, tend to have fewer missing updates. This makes sense since they depend on fewer neighbors who can potentially receive updates. However, this is not always the case. For example, UCI, which is sparser than tgbl-comment, has more missing updates. Moreover, increasing the batch size, even for large and sparse graphs (such as tgbl-review, tgbl-coin, and tgbl-comment), increases the number of missing updates, making them non-negligible. Even a low number such as 0.3 in tgbl-review means that, on average, for 30% of the nodes in the graph, a memory-based model would not consider the node's recent update when making a prediction.

Table 6: Datasets and their average number of *missing updates* affecting the 1-hop neighborhood of input nodes, sorted by sparsity.

Dataset	Average Degree	MU200	MU100	MU50
Social Evo.	28371.8	19.2	9.7	5.2
UN Vote	5152.9	28.0	21.4	14.3
Contact	3506.1	5.1	3.8	3.1
UN Trade	1990.1	57.0	37.4	21.3
Enron	680.6	24.9	15.5	9.8
LastFM	653.0	5.9	3.5	2.1
US Legis.	268.4	31.0	25.2	17.4
Flights	146.3	14.5	11.9	8.7
Can. Parl.	101.4	66.8	40.6	22.3
Reddit	61.2	2.9	1.5	0.7
MOOC	57.6	6.5	3.9	2.3
tgbl-comment	44.5	0.4	0.4	0.1
tgbl-coin	35.7	4.2	2.6	1.6
UCI	31.5	9.3	5.9	3.6
Wikipedia	17.0	3.2	2.0	1.1
tgbl-review-v2	13.8	0.3	0.2	0.1

B Baselines descriptions

We used the following baselines in the experiments:

•DyRep [16]: DyRep is an RNN-based architecture that utilizes a temporal attention mechanism to exploit the dynamic structure of the graphs.

• TGAT [7]: TGAT uses a time-encoding function and aggregates neighborhood information using self-attention to compute the embedding for each node.

• TGN [6]: TGN is a general architecture for CTDG learning tasks. It uses both a prediction module and a memory module to get relevant and accurate predictions for each input at each moment in time. It does this by aggregating information from the neighborhood of each node and maintain learnable updated memory which is based on RNN, and thus also solves the staleness problem.

• CAWN [22]: The CAWN model is based on causal anonymous walks that are generated for each node. The walks are encoded using RNNs and aggregated to achieve the node representation.

• EdgeBank [26]: EdgeBank is a memorization algorithm that saves any seen update and, given an input, it predicts according to a simple decision rule that can be one of the following: whether the input was seen in the last few iterations (EdgeBank$_{th}$) or in the last few time units (EdgeBank$_{tw}$), or whether the input has already been seen a sufficient number of times (EdgeBank$_{re}$). While EdgeBank can also have a decision rule that is based on infinite memory i.e., predicts positive for any previously seen edge and predicts negative otherwise (EdgeBank$_{inf}$). The algorithm's simplicity allows it to perform extremely fast, making it significantly faster than any other model for dynamic graph learning. In our experiments, we report the best results of EdgeBank among all of its decision rule variations.

• GraphMixer [18]: GraphMixer uses three components for the task of future edge prediction: a link-encoder that is based on MLP and a fixed time-encoding function, a node-encoder that only performs neighborhood mean-pooling and another MLP for edge prediction.

• DyGFormer [27]: DyGFormer is a transformer-based architecture. To generate an encoding for a given interaction, DyGFormer generates a co-occurrence embedding of the interaction in addition to a neighborhood representation for each interacting node. Then, it uses a patching technique on historical representations of the interacting nodes to better capture long-term temporal dependencies. The patches are then sent to a transformer and its outputs are averaged to create the final representation.

C Additional Implementation Details

C.1 Supporting additional update types

In Section 5.2 we described how to handle edge addition updates. In case of an update that removes the edge $e_{i,j}$, $t_{i,j}$ should be set to 0. Similarly, when a node v_i is added to the graph, t_i should be set to the current time. t_i should be set to 0 when the node is removed.

C.2 Node classification

To adjust LDTGN for dynamic node classification, the merge function needs to be removed, s.t., the prediction operation is applied directly on the node embedding. The Wikipedia dataset can also be used for dynamic node classification, therefore we used it to evaluate our model compared to other baselines:

Table 7: AUC-ROC for node prediction task on the Wikipedia dataset.

Dataset	DyRep	TGAT	TGN	CAWN	GraphMixer	DyGFormer	LDTGN (ours)
Wikipedia	86.39±0.98	84.09±1.27	86.38±2.34	84.88±1.33	86.80±0.79	87.44±1.08	86.71±0.44

For this task of node classification LDTGN achieves comparable performance to previous state-of-the-art while still being the most efficient in terms of throughput and latency.

C.3 Further implementation details

For the TDE of LDTGN we used an MLP with two hidden layers and two activation layers of ReLU [40]. Each linear layer of TDE outputs vector of length 100. Before applying TDE the time difference need to be normalised to ease the learning process. We used Equation (20) to normalise the time difference, where C is the length of the dataset.

$$\text{normalise}(t) = \frac{log(1+t)}{log(1+C)} \tag{20}$$

For LDTGN-mem, we used Time2Vec as the TDE function. Time2Vec utilizes the cosine function, thus omitting the need for normalization.

The merge function of LDTGN and LDTGN-mem is an MLP that maps multiple input vectors into a single value that represents the probability of the edge to be positive. The MLP first applies linear layer that maps the three vectors into a single vector. Then, it reduces the vector's dimension to 80, 10 and finally to 1. After each dimensionality reduction, ReLU is being applied. Finally, a sigmoid function is applied on the result to obtain the probability of an edge to be positive.

In practice, it is challenging to utilize the full neighborhood of input nodes to compute the predictions and withstand a reasonable throughput, since the neighborhood of each node is expected to grow overtime. Thus, we implemented our models using the recent neighbors sampling strategy that was suggested by Rossi et al. [6] in which only the k neighbors of each hop which were recently involved in an update are used for computing the predictions. Thus, the computation time of the model becomes independent of the graph's scale. For our models, we used $k = 20$.

C.4 Choosing the memory batch size

The optimal batch size of the memory module depends on the model's requirements. If there is a demand for the model to operate in a certain throughput, the smallest batch size that enables that should be selected, but if, on the other hand, precision is the priority, we will select the largest batch size that achieves this. This trade-off is discussed earlier in Section 3. It is recommended that the general batch size will be a multiplication of the memory batch size for convenient implementation, but it does not have to. As demonstrated in Figure 1(b) different datasets can have different frequencies of missing updates. Datasets with larger frequencies of missing updates tend to require smaller memory batch sizes. In our experiments, we set the memory batch size of LDTGN-mem to be 50 as it provides a good trade-off between precision and throughput.

C.5 Training

We trained the models for 100 epochs with a patience of 20 epochs before early stopping. We used binary cross entropy loss as the objective function and optimized the models using Adam's algorithm with a learning rate of 10^{-4}. To ensure that all models were trained and tested on exactly the same negative edges, a constant seed was used. Additionally, the sampling function ensured that the size of the sampled batches of negative edges was the same for every model, equal to the batch size, which also prevented the sampled negative edges from differing between different model evaluations.

All the experiments were performed on Intel(R) Xeon(R) Gold 6130 CPU @ 2.10GHz and NVIDIA GeForce RTX 3090.

D Additional results

In Tables 8 to 11 we report the AP and AUC-ROC of our proposed model and the baselines for the tasks of transductive and inductive future edge prediction.

In Table 12 we report the results for TGB. The metric used in TGB is Mean Reciprocal Rank (MRR). '-' in the results table denotes scenarios where a specific method was either not officially applied to the dataset or was unable to complete the validation and testing phases within a reasonable time. The full implementation of LDTGN for TGB can be found in this url.

Table 8: AP for transductive future edge prediction with random negative sampling over five runs. The significantly best result for each benchmark appears in bold font.

Dataset	DyRep	TGAT	TGN	CAWN	EdgeBank	GraphMixer	DyGFormer	LDTGN (ours)	LDTGN-mem (ours)
Wikipedia	94.86±0.06	96.94±0.06	98.45±0.06	98.76±0.03	90.37±0.00	97.25±0.03	**99.03±0.02**	98.86±0.02	**98.99±0.03**
Reddit	98.22±0.04	98.52±0.02	98.63±0.06	99.11±0.01	94.86±0.00	97.31±0.01	99.22±0.01	98.61±0.01	**99.28±0.02**
MOOC	81.97±0.49	85.84±0.15	89.15±1.60	80.15±0.25	57.97±0.00	82.78±0.15	87.52±0.49	83.34±1.47	**91.73±0.65**
LastFM	71.92±2.21	73.42±0.21	77.07±3.97	86.99±0.06	79.29±0.00	75.61±0.24	**93.00±0.12**	90.81±0.01	91.22±0.31
Enron	82.38±3.36	71.12±0.97	86.53±1.11	89.56±0.09	83.53±0.00	82.25±0.16	92.47±0.12	**98.10±0.01**	92.28±0.32
Social Evo.	88.87±0.30	93.16±0.17	93.57±0.17	84.96±0.09	74.95±0.00	93.37±0.07	94.73±0.01	**95.45±0.51**	94.02±0.16
UCI	65.14±2.30	79.63±0.70	92.34±1.04	95.18±0.06	76.20±0.00	93.25±0.57	95.79±0.17	**97.05±0.01**	95.75±0.04
Flights	95.29±0.72	94.03±0.18	97.95±0.14	98.51±0.01	89.35±0.00	90.99±0.05	**98.91±0.01**	97.50±0.07	98.76±0.06
Can. Parl.	66.54±2.76	70.73±0.72	70.88±2.34	69.82±2.34	64.55±0.00	77.04±0.46	97.36±0.45	**99.47±0.03**	72.82±9.17
US Legis.	75.34±0.39	68.52±3.16	75.99±0.58	70.58±0.48	58.39±0.00	70.74±1.02	71.11±0.59	**92.08±0.09**	80.93±0.48
UN Trade	63.21±0.93	61.47±0.18	65.03±1.37	65.39±0.12	60.41±0.00	62.61±0.27	66.46±1.29	**97.82±0.07**	96.65±0.19
UN Vote	62.81±0.80	52.21±0.98	65.72±2.17	52.84±0.10	58.49±0.00	52.11±0.16	55.55±0.42	**80.94±1.43**	71.21±1.14
Contacts	95.98±0.15	96.28±0.09	96.89±0.56	90.26±0.28	92.58±0.00	91.92±0.03	98.29±0.01	98.19±0.03	**98.78±0.04**

Table 9: AP for inductive future edge prediction with random negative sampling over five different runs. The significantly best result for each benchmark appears in bold font.

Dataset	DyRep	TGAT	TGN	CAWN	GraphMixer	DyGFormer	LDTGN (ours)	LDTGN-mem (ours)
Wikipedia	92.43±0.37	96.22±0.07	97.83±0.04	98.24±0.03	96.65±0.02	98.59±0.03	**98.74±0.02**	98.40±0.04
Reddit	96.09±0.11	97.09±0.04	97.50±0.07	98.62±0.01	95.26±0.02	98.84±0.02	98.00±0.04	**98.86±0.02**
MOOC	81.07±0.44	85.50±0.19	89.04±1.17	81.42±0.24	81.41±0.21	86.96±0.43	82.73±1.52	**90.61±0.32**
LastFM	83.02±1.48	78.63±0.31	81.45±4.29	89.42±0.07	82.11±0.42	**94.23±0.09**	92.17±0.01	92.62±0.59
Enron	74.55±3.95	67.05±1.51	77.94±1.02	86.35±0.51	75.88±0.48	89.76±0.34	**96.06±0.09**	88.07±0.56
Social Evo.	90.04±0.47	91.41±0.16	90.77±0.86	79.94±0.18	91.86±0.06	93.14±0.04	**94.37±0.68**	91.31±0.22
UCI	57.48±1.87	79.54±0.48	88.12±2.05	92.73±0.06	91.19±0.42	94.54±0.12	**94.92±0.01**	93.00±0.12
Flights	92.88±0.73	88.73±0.33	95.03±0.60	97.06±0.02	83.03±0.05	**97.79±0.02**	95.60± 0.10	97.31±0.16
Can. Parl.	54.02±0.76	55.18±0.79	54.10±0.93	55.80±0.69	55.91±0.82	87.74±0.71	**97.83±0.06**	58.05±3.08
US Legis.	57.28±0.71	51.00±3.11	58.63±0.37	53.17±1.20	50.71±0.76	54.28±2.87	**83.76±0.44**	65.75±1.57
UN Trade	57.02±0.69	61.03±0.18	58.31±3.15	65.24±0.21	62.17±0.31	64.55±0.62	**97.43±0.07**	89.21±1.15
UN Vote	54.62±2.22	52.24±1.46	58.85±2.51	49.94±0.45	50.68±0.44	55.93±0.39	**81.29±1.41**	63.54±2.09
Contacts	92.18±0.41	95.87±0.11	93.82±0.99	89.55±0.30	90.59±0.05	**98.03±0.02**	97.85±0.03	**97.94±0.13**

Table 10: AUC-ROC for transductive future edge prediction with random negative sampling over five runs. The significantly best result for each benchmark appears in bold font.

Dataset	DyRep	TGAT	TGN	CAWN	EdgeBank	GraphMixer	DyGFormer	LDTGN (ours)	LDTGN-mem (ours)
Wikipedia	94.37 ± 0.09	96.67 ± 0.07	98.37 ± 0.07	98.54 ± 0.04	90.78 ± 0.00	96.92 ± 0.03	**98.91 ± 0.02**	98.67±0.01	**98.90±0.05**
Reddit	98.17 ± 0.05	98.47 ± 0.02	98.60 ± 0.06	99.01 ± 0.01	95.37 ± 0.00	97.17 ± 0.02	99.15 ± 0.01	98.20±0.02	**99.25±0.02**
MOOC	85.03 ± 0.58	87.11 ± 0.19	91.21 ± 1.15	80.38 ± 0.26	60.86 ± 0.00	84.01 ± 0.17	87.91 ± 0.58	82.43±1.72	**93.33±0.54**
LastFM	71.16 ± 1.89	71.59 ± 0.18	78.47 ± 2.94	85.92 ± 0.10	83.77 ± 0.00	73.53 ± 0.12	**93.05 ± 0.10**	90.79±0.01	91.68±0.51
Enron	84.89 ± 3.00	68.89 ± 1.10	88.32 ± 0.99	90.45 ± 0.14	87.05 ± 0.00	84.38 ± 0.21	93.33 ± 0.13	**98.31±0.01**	93.35±0.42
Social Evo.	90.76 ± 0.21	94.76 ± 0.16	95.39 ± 0.17	87.34 ± 0.08	81.60 ± 0.00	95.23 ± 0.07	96.30 ± 0.01	**96.82±0.25**	95.93±0.06
UCI	68.77 ± 2.34	78.53 ± 0.74	92.03 ± 1.13	93.87 ± 0.08	77.30 ± 0.00	91.81 ± 0.67	94.49 ± 0.26	**96.22±0.03**	94.79±0.07
Flights	95.95 ± 0.62	94.13 ± 0.17	98.22 ± 0.13	98.45 ± 0.01	90.23 ± 0.00	91.13 ± 0.01	**98.93 ± 0.01**	96.98±0.09	98.82±0.07
Can. Parl.	73.35 ± 3.67	75.69 ± 0.78	76.99 ± 1.80	75.70 ± 3.27	64.14 ± 0.00	83.17 ± 0.53	97.76 ± 0.41	**99.68±0.02**	77.66±7.92
US Legis.	82.28 ± 0.32	75.84 ± 1.99	83.34 ± 0.43	77.16 ± 0.39	62.57 ± 0.00	76.96 ± 0.79	77.90 ± 0.58	**94.88±0.10**	87.96±0.53
UN Trade	67.44 ± 0.83	64.01 ± 0.12	69.10 ± 1.67	68.54 ± 0.18	66.75 ± 0.00	65.52 ± 0.51	70.20 ± 1.44	**97.91±0.06**	97.16±0.17
UN Vote	67.18 ± 1.04	52.83 ± 1.12	69.71 ± 2.65	53.09 ± 0.22	62.97 ± 0.00	52.46 ± 0.27	57.12 ± 0.62	**86.81±0.87**	77.33±1.04
Contact	96.48 ± 0.14	96.95 ± 0.08	97.54 ± 0.35	89.99 ± 0.34	94.34 ± 0.00	93.94 ± 0.02	98.53 ± 0.01	98.58±0.01	**99.06±0.04**

Table 11: AUC-ROC for inductive future edge prediction with random negative sampling over 5 different runs. The significantly best result for each benchmark appears in bold font.

Dataset	DyRep	TGAT	TGN	CAWN	GraphMixer	DyGFormer	LDTGN (ours)	LDTGN-mem (ours)
Wikipedia	91.49±0.45	95.90±0.09	97.72±0.03	98.03±0.04	95.57±0.20	**98.48±0.03**	98.23±0.00	98.30±0.06
Reddit	96.05±0.12	96.98±0.04	97.39±0.07	98.42±0.02	93.80±0.07	**98.71±0.01**	97.30±0.03	98.56±0.05
MOOC	84.03±0.49	86.84±0.17	91.24±0.99	81.86±0.25	81.43±0.19	87.62±0.51	81.88±1.74	**92.36±0.30**
LastFM	82.24±1.51	76.99±0.29	82.61±3.15	87.82±0.12	70.84±0.85	**94.08±0.08**	91.75±0.01	92.57±0.86
Enron	76.34±4.20	64.63±1.74	78.83±1.11	87.02±0.50	72.33±0.99	90.69±0.26	**95.77±0.13**	88.46±0.79
Social Evo.	91.18±0.49	93.41±0.19	93.43±0.59	84.73±0.27	93.71±0.18	95.29±0.03	**96.03±0.37**	94.01±0.2
UCI	58.08±1.81	77.64±0.38	86.68±2.29	90.40±0.11	84.49±1.82	92.63±0.13	**92.83±0.02**	90.83±0.21
Flights	93.56±0.70	88.64±0.35	95.92±0.43	96.86±0.02	82.48±0.01	**97.80±0.02**	94.44±0.21	97.39±0.21
Can. Parl.	55.27±0.49	56.51±0.75	55.86±0.75	58.83±1.13	55.83±1.07	89.33±0.48	**98.73±0.05**	58.59±4.42
US Legis.	61.07±0.56	48.27±3.50	62.38±0.48	51.49±1.13	50.43±1.48	53.21±3.04	**88.19±0.24**	72.45±1.31
UN Trade	58.82±0.98	62.72±0.12	59.99±3.50	67.05±0.21	63.76±0.07	67.25±1.05	**97.47±0.07**	90.26±1.28
UN Vote	55.13±3.46	51.83±1.35	61.23±2.71	48.34±0.76	50.51±1.05	56.73±0.69	**86.99±0.86**	68.99±1.66
Contact	91.89±0.38	96.53±0.10	94.84±0.75	89.07±0.34	93.05±0.09	**98.30 ± 0.02**	98.26±0.02	98.22±0.14

Table 12: MRR for future edge prediction on TGB. The significantly best result for each benchmark appears in bold font. The second best result for each benchmark is underlined.

Dataset	DyRep	TGAT	TGN	CAWN	EdgeBank	GraphMixer	DyGFormer	LDTGN (ours)
tgbl-wiki-v2	0.050 ± 0.017	0.141 ± 0.007	0.396 ± 0.060	0.711 ± 0.006	0.571	0.118 ± 0.002	0.798 ± 0.004	**0.816 ± 0.001**
tgbl-review-v2	0.220 ± 0.030	0.355 ± 0.012	0.349 ± 0.020	0.193 ± 0.001	0.025	**0.521 ± 0.015**	0.224 ± 0.015	0.381 ± 0.009
tgbl-coin-v2	0.452 ± 0.046	-	0.586 ± 0.037	-	0.580	-	**0.752 ± 0.004**	0.649 ± 0.007
tgbl-comment	0.289 ± 0.033	-	0.379 ± 0.021	-	0.149	-	**0.670 ± 0.001**	0.400 ± 0.006
tgbl-flight-v2	0.556 ± 0.014	-	0.705 ± 0.020	-	0.387	-	-	**0.736 ± 0.015**

E Decoupling potential speedup analysis

In this section, we analyze the potential speedup that models can achieve by using the decoupling strategy. The decoupling strategy does not affect running time directly, but rather aids to accelerate the running time of models without compromising their precision. Figure 3 demonstrates this exact idea: one can decouple a memory-based model and increase its batch size significantly while maintaining a constant memory batch size. This will lead to running time improvement without compromising the precision of the decoupled model.

In the context of a sequence containing updates for a model to apply and inputs for it to predict, denote the time it takes for the memory module to apply all the given updates in the sequence as t_{memory} and denote the time it takes for the prediction module to finish computing the predictions for all the inputs in the sequence as $t_{prediction}$. The total time it takes for the model to finish processing the sequence is:

$$T_{total} = t_{memory} + t_{prediction} \tag{21}$$

By decoupling the network one can reduce this time to

$$T_{decouple\_total} = t_{memory} + \frac{t_{prediction}}{BS_{new}/BS_{old}} \tag{22}$$

Where BS_{old} is the batch size of the model before decoupling and BS_{new} is the batch size of the decouple model. Hence the potential speedup of the model is:

$$speedup = \frac{BS_{new} \cdot t_{prediction} + BS_{new} \cdot t_{memory}}{BS_{old} \cdot t_{prediction} + BS_{new} \cdot t_{memory}} \tag{23}$$