Mind the truncation gap: challenges of learning on dynamic graphs.

Anonymous authors
Paper under double-blind review

Abstract

Systems characterized by evolving interactions, prevalent in social, financial, and biological domains, are effectively modeled as continuous-time dynamic graphs (CTDGs). To manage the scale and complexity of these graph datasets, machine learning (ML) approaches have become essential. However, CTDGs pose challenges for ML because traditional static graph methods fail to account for event timings naturally. Newer approaches, such as graph recurrent neural networks (GRNNs), are inherently time-aware and offer improvements over static methods for CTDGs. Yet, GRNNs face another issue: the short truncation of backpropagation-through-time (BPTT) whose impact has never been properly examined until now. In this work, we demonstrate that this truncation can limit the learning of temporal dependencies, resulting in reduced performance. Through experiments on a novel synthetic task as well as real-world datasets, we reveal that there exists a performance gap between full backpropagation-through-time (F-BPTT) and truncated backpropagation-through-time (T-BPTT) which we term the "truncation gap". As the importance of CTDGs grows, understanding and addressing this truncation gap is essential and we discuss potential future directions to address this issue.

1 Introduction

In many domains, data naturally forms a sequence of interactions between various entities, which can be represented as an evolving network. Examples include interactions between users in social networks, card payments from users to merchants in payment networks, or bank transfers between entities in financial networks. These interactions typically involve only a pair of entities and are often modeled as a dynamic graph, where each entity is represented by a node and each interaction by an edge with an associated timestamp.

Unlike static graphs, where links between entities are persistent, dynamic graphs capture the sequential aspect of interactions, which can provide valuable information for different inference tasks. In these settings, the latent variables associated with nodes, such as user preferences in social networks, can evolve over time. Tasks on dynamic graphs involve predicting certain edge or node properties or the likelihood of an event involving two nodes at a specific point in time based on past information.

Recent works (Dai et al., 2017; Trivedi et al., 2019; Kumar et al., 2019; Rossi et al., 2020) have proposed treating such dynamic graphs as dynamical systems where a state for each entity evolves over time. The dynamics of these states are based on recurrent neural network (RNN) cells which use the previous states of both interacting nodes when computing the new states, therefore coupling both sides of an interaction at the time of an event.

As a **first contribution of this work**, we provide a **survey of architectures involving recurrent cells to process dynamic graph-based data**, naming them graph recurrent neural networks (GRNNs), and argue that this formulation deals more naturally with time without loosing expressivity compared to static GNN methods. While there are differences in how various GRNN approaches handle the training, all of them have in common that batches are formed using sequential time windows over the events. The sequential ordering is needed because future states can only be computed from past information in the correct order. Backpropagation is applied to each batch to obtain the gradients with respect to the learnable parameters

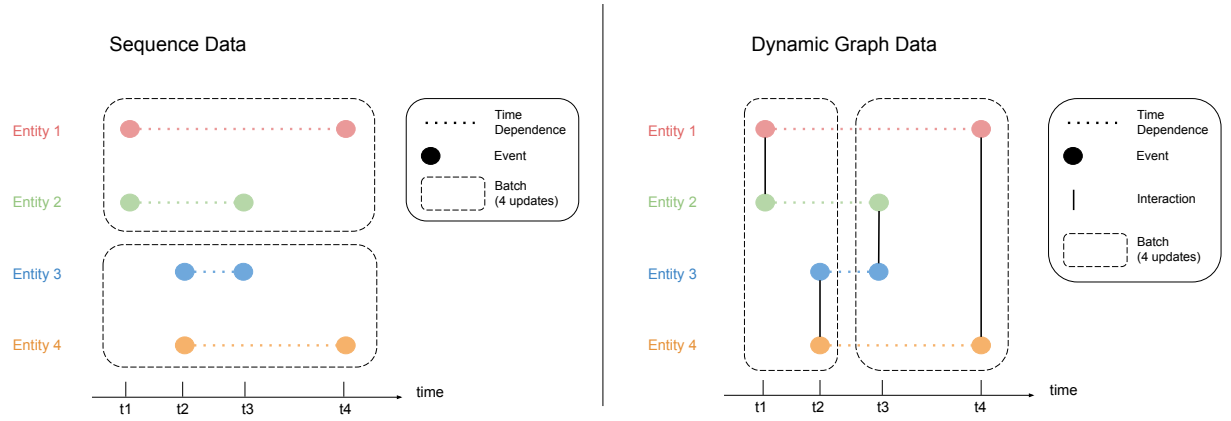

Figure 1: **Truncation of temporal history becomes severe in dynamic graphs.** (left) Sequence based data can be grouped by sequence when defining batches. In this specific example of sequences with two events, with a batch capacity of 4 entity updates, we can include two sequences per batch. Temporal dependencies between the events (horizontal dotted lines) are not broken by batching. (right) Due to the interactions between states, we cannot consider isolating a subset of entities in a batch on dynamic graphs as we need the counterparty entity's state to update an entity's state when an event occurs. Batches are defined by time instead of by entity but this leads to more extreme gradient truncation along the time axis (time dependencies are broken by batching). In the example, with the same capacity of 4 entity updates, each batch now includes only a single event per entity.

and the learning signal therefore cannot cross batch boundaries. In other words, this *truncation* of the backpropagation ignores the dependence of information older than the current batch.

In sequence modelling tasks, one typically has many sequences that are fully independent. If small enough, these sequences can be included in their entirety in a batch, or alternatively, they can be broken down into sizeable chunks thus mitigating the impact of truncating the backpropagation (Figure 1, left). On dynamic graphs, however, it is not possible to split the data into different sequences per entity due to the coupled dynamics. Effectively, we have *a single global sequence* and batches need to be defined over globally ordered interaction events, from oldest to newest (Figure 1, right). Because such a sequential batch of events can involve completely different entities, if the number of entities is large enough one can find themselves in a situation where batches include a single update per entity. This in turn means that, from the entity point of view, only temporal dependencies spanning a single hop in the graph can be learned accurately. Such drastic truncation could lead to incorrect training gradients and a failure in learning to leverage longer term dependencies existing in the data.

As a **second contribution of this work**, we **investigate the impact of this truncation** by comparing the training approach proposed in previous works with F-BPTT. We propose a synthetic edge regression task on a dynamic graph that truncated backpropagation fails to learn when longer temporal dependencies are required. We also show that a gap between the performance achievable with truncated vs. full backpropagation exists on real world dynamic graph datasets by comparing both training methods on popular benchmarks. We call this drop in performance the *truncation gap*, since it is solely caused by the truncated backpropagation.

While F-BPTT is not a practical solution for large dynamic graph datasets due to its memory scaling, our results show that current training approaches based on truncating backpropagation are not able to fully exploit the capacity of GRNN models. We believe that further investigation into better training approaches for these models is warranted and propose possible future research directions.

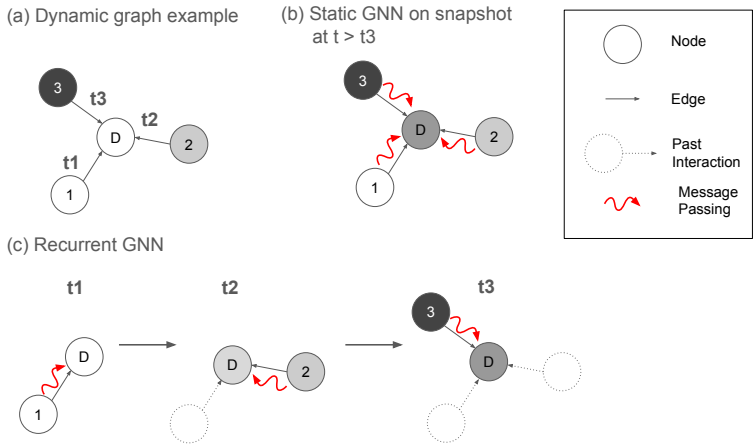

Figure 2: **GRNNs are natural solutions for dynamic graph tasks.** (a) An example of a dynamic graph. Three source nodes interact with the central destination node of interest (node D). The interactions happen at different timesteps (t1<t2<t3). Colors of the nodes are denoting different embedding features. (b) A static GNN on a snapshot of the dynamic graph described in panel (a) aggregates all neighboring nodes of the destination node D simultaneously. (c) A recurrent GNN aggregates the neighboring nodes of the destination node D spread over various timesteps. The final embedding of the destination node D will represent an aggregated view of its neighborhood similarly as in the static GNN (see panel b). Therefore, we gained a more natural way to deal with time without compromising the graph nature of the computations.

2 Background

A graph neural network (GNN) is typically thought of as a network that maps a node embedding and (a sample of) its neighbour embeddings to a new node embedding. The mapping is permutation invariant, such that the specific order of the neighbour embeddings does not affect the output. Applying such GNN layers simultaneously to all node embeddings in a graph can be seen as message-passing, where messages are passed over d hops where d is equal to the number of GNN layers. In this formulation, the GNN acts on all nodes simultaneously and therefore only works in a static graph setting. One workaround typically used for dynamic graphs is to consider various *snapshots* of the graph on some chosen timepoints, and apply the static GNN on each of the snapshots (Figure 2b). However, this approach is very dependent on the choice of the snapshots and does not naturally take temporal information within each snapshot into account.

Using a recurrent cell to perform the computations essentially spreads out the graph aggregations over time (Figure 2c). In the end, one achieves similar graph-based representations as with the static GNN approaches, but with the added benefit that temporal information is more naturally dealt with and one has updated embeddings at any timepoint (as opposed to only at the chosen snapshot times). There are however two important differences with the static approaches:

- Permutation invariance over the neighbours is broken, since swapping the order of neighbors implies a different temporal sequence. This is in fact a desirable property in the dynamic setting since it allows to distinguish between interactions with the same nodes but in different temporal order.

- Messages flow strictly from past to future. One of the potential downsides of this is that a node state can become stale when there are no events for that node for an extended period of time. In such a scenario, leveraging more recent information from the node's neighbours could prove valuable. This was the motivation of Rossi et al. (2020) to add a GNN operation on top of the recurrent model.

> Another option could be to keep the recurrent setting and randomly add a few 'virtual' past edges in each batch to achieve a similar effect.

Another property of using recurrent cells for temporal graphs, is that the graph itself does not need to be explicitly stored in the simplest setting. Instead, it suffices to store the node embeddings and update them using the recurrent architecture whenever an edge arrives. Furthermore, no neighborhood sampling is needed. These properties hugely simplify the storage and operations required.

We will refer to such graph recurrent neural networks as GRNNs (other names have been used in the literature, such as memory-based temporal graph neural networks (Zhou et al., 2023) and message passing temporal graph networks (Souza et al., 2022)). A general framework for GRNNs is discussed in Appendix A. Given the discussed advantages of the GRNNs, we will focus on these approaches in the remainder of the work.

3 Related Work

In most previously proposed GRNN architectures, the updates of hidden states $\mathbf{h}_k^{(s_k)}$ of node s and $\mathbf{h}_k^{(d_k)}$ of node d for interaction k at time t_k are defined by a pair of functions, h_s and h_d:

$$\begin{aligned} \mathbf{h}_k^{(s_k)} &= h_s\left(\omega\left(\delta t_k^{(s_k)}\right), \mathbf{h}_{k-1}^{(s_k)}, \mathbf{h}_{k-1}^{(d_k)}, \mathbf{x}_k\right) , \\ \mathbf{h}_k^{(d_k)} &= h_d\left(\omega\left(\delta t_k^{(d_k)}\right), \mathbf{h}_{k-1}^{(d_k)}, \mathbf{h}_{k-1}^{(s_k)}, \mathbf{x}_k\right) , \end{aligned} \tag{1}$$

with $\omega(\delta t)$ an optional function producing an embedding for the time elapsed since the last update for each node, and $\mathbf{x}_k$ the message of the current interaction (which could itself be a combination of edge and node features).

CoEvolve, Deep CoEvolve and Know-Evolve. Wang et al. (2016) propose a temporal point-process model for event prediction where a latent state for each entity in the network is kept and updated on each event with a coupled dynamic model based on a Hawkes process. Dai et al. (2017) extend this model with a more generic update based on a standard RNN cell and Trivedi et al. (2017) apply a similar model to dynamic knowledge graphs.

In both later works, the authors assume that the node states are constant between interactions. The update for the source node from equation 1 takes the form

$$\mathbf{h}_k^{(s_k)} = \mathrm{RNN}\left(\mathbf{h}_{k-1}^{(s_k)}, \left[\delta t_k^{(s_k)}, \mathbf{h}_{k-1}^{(d_k)}, \mathbf{x}_k\right]\right), \tag{2}$$

where $[\cdot]$ denotes concatenation and a similar update is used for the destination node.

JODIE. Kumar et al. (2019) propose a similar model that foregoes predicting the time of interaction. I.e., rather than predicting a conditional intensity function for observing an event between any two entities, it tries to rank how plausible interactions between pairs of entities are at a given time.

The dynamics are governed by equations identical to 2, but the authors then propose an evolution function to project the embedding to any time t. Note however that this proposed projection is only used for prediction and is not part of the dynamics, which are only governed by equation 2.

DyRep. Trivedi et al. (2019) propose a recurrent model where the messages coming from the counterparty are itself the result of a graph attention operation over the counterparty graph neighborhood. In other words, instead of equation 2 the authors use

$$\mathbf{h}_k^{(s_k)} = \mathrm{RNN}\left(\mathbf{h}_{k-1}^{(s_k)}, \left[\delta t_k^{(s_k)}, \mathbf{GA}\left(\mathcal{N}(d_k)\right)\right]\right), \tag{3}$$

where $\mathbf{GA}$ denotes a graph attention operation applied over a graph neighborhood, $\mathcal{N}(d_k)$, of the counterparty node.

TGN. Rossi et al. (2020) tackle the same problem as Kumar et al. (2019) but introduce a few modifications on the model. Namely, they introduce a message function, $\mathbf{M}$, as an arbitrary function of the elapsed time, the hidden states of both nodes involved in an interaction and $\mathbf{x}_k$, resulting in the update:

$$\mathbf{h}_k^{(s_k)} = \text{RNN}\left(\mathbf{h}_{k-1}^{(s_k)}, \mathbf{M}\left(\delta t_k^{(s_k)}, \mathbf{h}_{k-1}^{(s_k)}, \mathbf{h}_{k-1}^{(d_k)}, \mathbf{x}_k\right)\right). \tag{4}$$

Moreover, an *embedding module* (which the authors propose to be an attention based graph neural network similar to Xu et al. (2020)) is added on top of the recurrent *memory* model to deal with the staleness problem of the hidden states in the absence of events. While this module is not aimed at tackling the truncation problem (the gradient propagation through the node embeddings that feed this module is still truncated), it could still help alleviate it since edge features for a number of past edges of a node are explicitly considered, affecting the prediction through an alternative path to the nodes' hidden states.

APAN. The main motivation of Wang et al. (2021) is to have feasible real-time inference using a temporal graph attention module. For this purpose, they introduce an architecture that makes use of a hidden state and a fixed-size mailbox per node that is updated asynchronously. Unlike JODIE or TGN, the node states are not updated based on RNN dynamics, but instead by an attention based encoder operating on the mailbox where the previous state helps determine the attention weights:

$$\mathbf{h}_k^{(s_k)} = \text{Encoder}\left(\mathbf{h}_{k-1}^{(s_k)}, \mathcal{M}^{(s_k)}(t_k^-)\right). \tag{5}$$

Here $\mathcal{M}^{(s_k)}(t_k^-) = \left[\mathbf{m}_1^{(s_k)}(t_k^-), \ldots, \mathbf{m}_M^{(s_k)}(t_k^-)\right]$ is the mailbox for node s_k at the time of the interaction. Note that this means that the update for the state s_k does not consider the counterparty state for edge k directly.

A new message for the event is delivered asynchronously to the mailboxes of each node in a temporal neighbourhood which act as FIFO queues. This message is a simple, non-learnable function of the new embeddings of the nodes involved in the event as well as the edge features (such as their sum). Similar to the GNN embedding module used on top of TGN, this type of architecture might help alleviate the truncation problem to some extent since it provides a direct path from (a function of) past edge features of each node to the prediction. However, it still suffers from a similar truncation problem since both the previous node state and all the messages in the mailbox are considered fixed inputs for backpropagation and thus, their dependence on the encoder parameters and information from multiple hops away is ignored.

NLB. Luo & Li (2024) consider a model similar to the TGN, with a GRNN used in conjunction with a temporal graph attention module. Instead of implementing recency sampling, they propose a novel forward temporal neighbour sampling method in order to improve the inference speed of the graph attention module without relying on a fixed set of the most recent temporal neighbours.

3.1 Training GRNNs: Batch Processing Strategies

The usual way to train RNNs is using the Back Propagation Through Time (BPTT) algorithm. When sequences become too long, one reverts to a truncated version, T-BPTT. In dynamic graphs, however, there is no good option to define sequences, since nodes interact with each other (Figure 1). Because future events are influenced by past events, the natural way to define batches is by adopting a sliding window over the globally ordered events. In other words, no random shuffling of events or sequences is possible, and each epoch will contain batches that sequentially slide from older to newer events. There are then different approaches in the literature on how to define and process these batches:

Deep CoEvolve and DyRep. Dai et al. (2017) and Trivedi et al. (2019) consider sequential batches of interactions. A computation graph for the forward pass is built for each batch, processing each event in the batch in sequence (Figure 3.1, left). This graph is then back-propagated through. Gradient propagation is therefore restricted to each batch, ignoring the model parameter dependencies of the hidden states of each entity that serve as inputs to this computational graph. When batches are large enough, this sequential processing of each event in a batch can allow for the learning of multi-hop temporal dependencies, but at

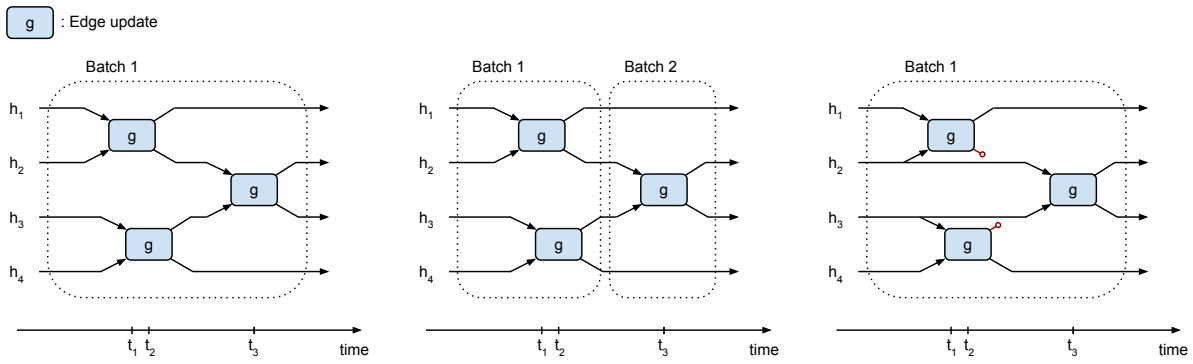

Figure 3: Three different batching strategies illustrated. Four nodes with respective states $h_1 \dots h_4$ interact as in Figure 1 until time t_3. Each blue box denotes an interaction, where the hidden states of the two interacting nodes are updated. On the left the approach of Dai et al. (2017) where a different computation graph is built for each mini-batch. Due to the sequential processing within a batch, computational efficiency is lost. In the middle the *t-Batching* strategy of Kumar et al. (2019) which uses variable sized batches to guarantee no sequential dependencies within a batch, allowing parallel processing. On the right, the approach of Rossi et al. (2020) that uses fixed size batches and parallel processing at the cost of correctness, leading to inconsistent histories where the latent states used for the third event ignore the updates of the previous two events.

the cost of being very slow (essentially each edge in the dataset is processed sequentially). Furthermore, for large networks and if events are well distributed across many entities, a single update per entity would be processed in each batch and therefore only a single hop temporal dependency is learned.

JODIE. Kumar et al. (2019) propose instead using variable sized batches so that a single update per entity is present in each batch (which they call *t-Batches)*, Figure 3.1, middle). This makes the training more efficient since all updates in a single batch can now occur in parallel, but also exacerbates the problem of gradient truncation because it guarantees a truncation horizon of a single hop per entity.

TGN. Rossi et al. (2020) consider fixed size batches of temporally ordered events, but ignore any sequential dependencies within each batch by processing all updates in parallel (Figure 3.1, right). When multiple events involving a single entity are present in a batch, this leads to an inconsistent history where the last edge effectively determines the final state of the node, ignoring previous edges that occurred in the same batch. While this approach makes the processing more efficient and straightforward, it (1) guarantees a truncation horizon of a single hop and (2) introduces an approximation that can potentially deteriorate the results. This forces the authors to propose keeping batch sizes large enough to be more efficient than (Deep)CoEvolve (Dai et al., 2017), but small enough to avoid serious deterioration of results. Empirically, they found that a batch size of 200 provided a reasonable trade-off. We also note that the GNN module on top of the recurrent cell may help to alleviate the deterioration, since it provides an alternative way to model dependencies in the data to the GRNN component.

Finally, Zhou et al. (2023) propose efficient ways to distribute the training of the above models over multiple GPUs, but its proposed methods do not affect the truncation problem.

In summary, all discussed methods suffer from truncating backpropagation with typically small horizons. Because this flaw is tightly linked to the use of backpropagation, and due to the success of backpropagation in other settings and the ease of implementation in current deep learning frameworks, it has been accepted as an unavoidable shortcoming. In fact, none of the above works discusses this truncation's effect on learning. In the next section, we propose a synthetic task to benchmark performance of the various approaches on learning temporal dependencies. We identify severe degradation of performance due to the truncation, and subsequently discuss future research directions to mitigate this issue.

4 Synthetic Task

In order to investigate the effect of limiting the backpropragation to a single batch in GRNNs on their ability to learn longer term dependencies we propose a novel edge regression synthetic task. We construct the task to achieve the following desirable properties:

- A parameter M should regulate the amount of past edge memory necessary to solve the task (larger M would denote older information is needed to correctly predict the label).
- The latent state of the source node is updated using the destination node's latent state and vice-versa. Otherwise, non-graph based methods would be able to solve the task.

The code to generate the data will be made available upon acceptance, such that this task can be used in future research to benchmark the developed methods.

4.1 Task Specification

We took inspiration from the *adding task* (Hochreiter & Schmidhuber, 1997), a benchmark frequently used for recurrent neural networks, and construct a variation for graphs that satisfies the above properties. We consider an internal state for each node $i \in [1..N]$ in a graph, consisting of a memory buffer of size M, $\mathbf{h}^{(i)} \in \mathbb{R}^M$. Each edge, $e_k = (s_k, d_k, x_k, y_k)$, is defined by two nodes, s_k and d_k, an input numeric feature x_k and a target y_k. The target is computed by adding the last two elements of the buffers of the nodes (prior to the state update for event k):

$$y_k = \left(\mathbf{h}_{k-1}^{(s_k)}\right)_M + \left(\mathbf{h}_{k-1}^{(d_k)}\right)_M .$$

The memory buffer of each node is updated after each edge arrival by shifting the values in the buffer using a first-in-first-out (FIFO) principle. Importantly, the newly stored value depends on the edge feature and the *counterparty* node, such that the second property above is satisfied:

$$\begin{aligned}\left(\mathbf{h}_{k}^{(s_k)}\right)_0 &= \frac{1}{2}\left(\mathbf{h}_{k-1}^{(d_k)}\right)_M + \frac{1}{2}x_k \\ \left(\mathbf{h}_{k}^{(s_k)}\right)_i &= \left(\mathbf{h}_{k-1}^{(s_k)}\right)_{i-1} , \quad i \in [1 \dots M] .\end{aligned}$$

The update is symmetrical for $\mathbf{h}^{(d_k)}$. A schematic representation of the task is depicted in Figure 4.1.

This task becomes more challenging as M increases since it determines a delay (in number of edges) between an input being observed and it affecting an output for the same node. A recurrent model will have to learn the underlying dynamics of the task from these input/output pairs in order to correctly solve it. It will, therefore, have to memorize older information for each node as M increases.

To generate dynamic graphs for this synthetic task we sample edges randomly by picking a random pair of nodes, s_k and d_k, uniformly and a corresponding random edge feature, $x_k \sim_{i.i.d.} \mathcal{N}(0, 1)$, drawn from a standard normal distribution. The states for all nodes are initialized to the zero vector.

4.2 Results

For our experiments we consider dynamic graphs consisting of 1000 edges and using a total of 100 nodes, which we refer to as one epoch. We generate a new dynamic graph per epoch.

All models are based on Equation 1 using a single GRU cell for both updates (since the dynamics are symmetric) and without time encoding.

We train a first model with F-BPTT by back-propagating through an entire epoch (i.e. the entire dynamic graph) in order to compute the gradient of the total loss (sum of losses for all edges). One gradient step is taken per epoch using an AdamW optimizer Loshchilov & Hutter (2019) with a learning rate of 1e-3 and weight decay of 1e-4 for a total of 5000 epochs.

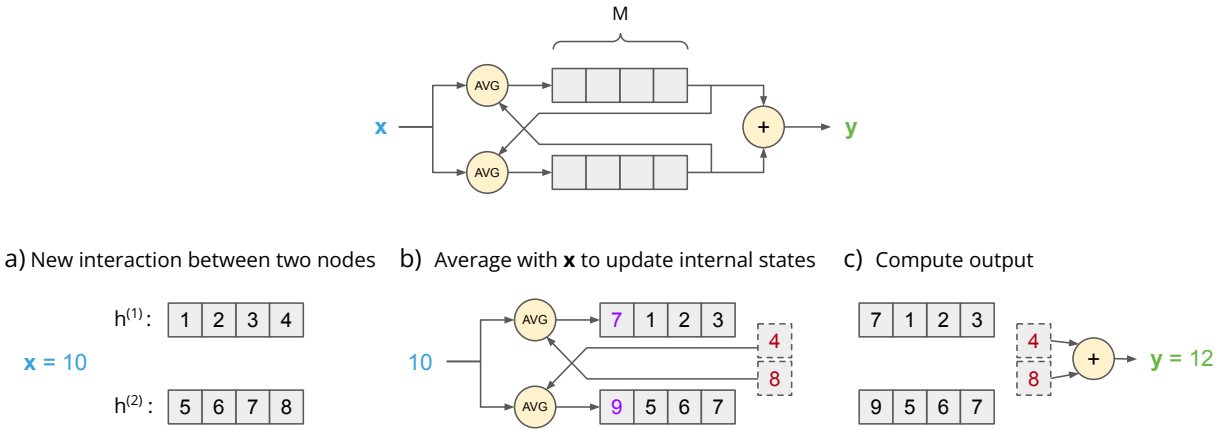

Figure 4: Visual depiction of synthetic task dynamics. A fixed size FIFO buffer with length M (4 in the picture) is used to store the internal state of each node. When a new edge between two nodes occurs (a), the input is averaged with the last elements of each counterparty node's buffer in order to determine the new number to be stored in each buffer (b). The sum of these last elements also determines the output for the edge (c).

For the model using T-BPTT we compute truncated gradients for each edge and accumulate these (i.e., sum them) over an entire epoch, performing a single gradient step at the end. We use the same optimizer and hyperparameters for the same number of epochs as with F-BPTT. The reasoning behind this setup is to be directly comparable with F-BPTT but, in this instance, employing an online setup where a gradient step is taken per edge did not seem to significantly alter the results for T-BPTT.

We train models using F-BPTT and T-BPTT and with hidden state sizes of 32, 64 and 128. For all models, we compute the average loss over the last 100 epochs across various values of the memory parameter M. For comparison, we also plot the performance of a trivial baseline that predicts $\hat{y}_k = 0$ for every edge.

While for $M = 1$ the models using T-BPTT are able to approximately solve the task, their performance rapidly degrades as M is increased. In contrast, models trained using F-BPTT are able to fully solve the task (the final loss is orders of magnitude smaller than the target variance) for every value of M tested (Figure 4.2).

5 Dynamic Graph Benchmark Results

In order to show that the *truncation gap* is also present in real world dynamic graphs we conduct a simple experiment on three publicly available dynamic graph datasets from Kumar et al. (2019) (Reddit, Wikipedia and MOOC).

We consider a GRNN model based on a GRU cell with a hidden state size of 64. Similar to Rossi et al. (2020) we use the same GRU for the updates to both source and destination nodes (note that despite these nodes being heterogeneous, using a GRNN model with non-symmetric dynamics did not lead to better performance). We also disregard any timestamp information on the edges by not having our update function depend on any δt, therefore assuming that there are no autonomous dynamics between events. These choices are made to simplify the experimental setup since our goal is not to challenge the state of the art results for dynamic graphs. Our choice of state size is limited by the memory required to run full backpropagation on the training set.

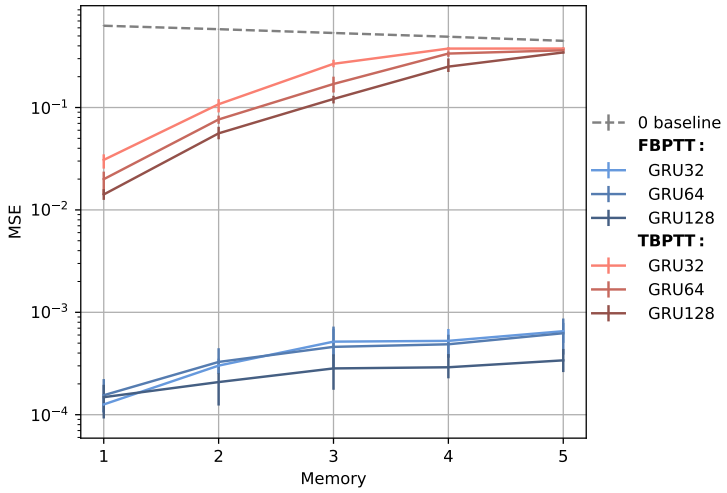

Figure 5: Mean squared error (MSE) obtained for different sized GRU models trained with both F-BPTT and T-BPTT. The depicted error bars correspond to the min and max MSE values over 5 different random seeds used for parameter initialization and dynamic graph sampling. The solid lines correspond to the mean MSE over the different seeds.

We train the GRNN model with both truncated and full BPTT, using the same batching strategy of Rossi et al. (2020) (see Section 3.1) with a batch size of 200. We also employ the same negative sampling strategy where for every edge in the graph, a negative edge is obtained by uniformly sampling a random destination node. The training loss is then the binary cross-entropy for classifying positive and negative edges. For F-BPTT we backpropagate through all the batches in the training data, computing the gradient of the total training loss for the epoch. This is possible for the selected datasets due to their relatively small size. In order to enable a fair comparison, we also accumulate the truncated gradients for every batch computed with T-BPTT, performing a single parameter update per training epoch. This allows us to have a common training setup with the same number of epochs and the same number of parameter updates for both methods with the same potential choices of hyperparameters.

We use the Adamw optimizer (Loshchilov & Hutter, 2019), and tune the learning rate and weight decay parameters using 25 trials of random search. We also tune the values for the dropout of the multilayer perceptron classifier as well as a state dropout that is applied to the node states modified in each batch. For this state dropout we experiment with two approaches: (i) a regular dropout layer applied directly to all the states updated at each batch or (ii) a layer that keeps each element of the previous state with a given dropout probability instead of updating to the newly computed state (which we call recurrent dropout). The search space used for these tuned parameters is summarized in Table 1. We repeat our experiments using 3 different seeds determining the hyperparameter sampling, model initialization, dropout and negative edge selection during training for the Wikipedia and MOOC datasets. A single seed is used for Reddit, which is the largest dataset, due to it being substantially slower to run.

For evaluation we use a similar setup to Kumar et al. (2019) and Huang et al. (2023). We process the validation and test sets using the same batching strategy used during training, and then rank for every edge all the possible destination nodes according to the probabilities predicted by the model. Note that since the number of destination nodes for these datasets is at most 1000 we don't use any negative sampling for evaluation. Mean Reciprocal Rank (MRR) and Recall@10 values are computed based on the rank of the true destination node for each edge. We stop each trial when neither metric has improved for 250 epochs.

The results are reported in Table 2, where we can observe a large *truncation gap* on the Reddit and MOOC datasets representing at minimum a 10% improvement when using F-BPTT. For Wikipedia, while a 3%

Table 1: Search space for the 25 trials of random search used to tune the hyperparameters of the optimizer and the model dropouts. The last parameter refers to a uniform at random choice between the two types of state dropout.

Parameter	Domain	Distribution
`learning_rate`	$[10^{-3}, 10^{-2}]$	Log-Uniform
`weight_decay`	$[10^{-5}, 1]$	Log-Uniform
`mlp_dropout`	$[0, 0.3]$	Uniform
`state_dropout`	$[0, 0.3]$	Uniform
`state_dropout_type`	{Regular, Recurrent}	Uniform

Table 2: **Truncated BPTT results in worse performance across most datasets and metrics.** Results on the benchmark dynamic graph datasets showing that there is a performance gap between the models trained with truncated and full backpropagation. For Wikipedia and MOOC, the reported values are the means and standard errors on the test set over the different seeds.

	Reddit		Wikipedia		MOOC	
	MRR	Recall@10	MRR	Recall@10	MRR	Recall@10
T-BPTT	0.549	0.717	$0.534_{\pm 0.011}$	$0.677_{\pm 0.007}$	$0.281_{\pm 0.006}$	$0.643_{\pm 0.021}$
F-BPTT	0.648	0.803	$0.525_{\pm 0.012}$	$0.698_{\pm 0.006}$	$0.343_{\pm 0.008}$	$0.708_{\pm 0.007}$
Truncation Gap	0.010	0.085	$-0.009_{\pm 0.012}$	$0.022_{\pm 0.006}$	$0.062_{\pm 0.006}$	$0.064_{\pm 0.020}$
Improvement (%)	18.1	11.9	$-1.8_{\pm 2.2}$	$3.2_{\pm 0.8}$	$22.2_{\pm 2.6}$	$10.1_{\pm 3.5}$

improvement in the Recall metric was also observed, a non-statistically significant (negative) result is obtained for MRR.

6 Beyond Backpropagation

While we argued that recurrent neural network architectures are well-suited to handle tasks on dynamic graph data, we have shown that the truncation present in state-of-the-art approaches can lead to the failure of learning tasks requiring larger temporal dependencies. Since the truncation is inevitable in approaches using backpropagation, solving the issue warrants looking beyond backpropagation-based methods.

Viable research directions could include approximations to real-time recurrent learning (RTRL). More specifically, recent work (Tallec & Ollivier, 2017; Mujika et al., 2018; Benzing et al., 2019) describe unbiased stochastic low-rank approximations to the true Jacobians in recurrent neural networks on sequential data, which make online learning feasible. Adapting such methods for the dynamic graph settings would be an interesting avenue to resolve the temporal dependency learning issue. Another avenue would be to adapt the recurrent architecture itself, to make full RTRL feasible.

7 Conclusions

We surveyed current methods leveraging recurrent architectures for inference tasks on dynamic graphs, discussing the strengths and weaknesses of various training and batching strategies. We identified that backpropagation may reach its limits in this dynamic graph setting, due to a severe truncation of the history, confirming that it is holding back GRNN models both in a proposed novel synthetic task and in publicly available real-world datasets. Therefore, we believe that methods beyond backpropagation warrant more attention in this context.

References

Frederik Benzing, Marcelo Matheus Gauy, Asier Mujika, Anders Martinsson, and Angelika Steger. Optimal Kronecker-Sum Approximation of Real Time Recurrent Learning. In *Proceedings of the 36th International Conference on Machine Learning*, pp. 604–613. PMLR, May 2019. URL https://proceedings.mlr.press/v97/benzing19a.html. ISSN: 2640-3498.

Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural Ordinary Differential Equations, December 2019. URL http://arxiv.org/abs/1806.07366. arXiv:1806.07366 [cs, stat].

Hanjun Dai, Yichen Wang, Rakshit Trivedi, and Le Song. Deep Coevolutionary Network: Embedding User and Item Features for Recommendation, February 2017. URL http://arxiv.org/abs/1609.03675. arXiv:1609.03675 [cs].

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

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, September 2023. URL http://arxiv.org/abs/2307.01026. arXiv:2307.01026 [cs].

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*, KDD '19, pp. 1269–1278, New York, NY, USA, July 2019. Association for Computing Machinery. ISBN 978-1-4503-6201-6. doi: 10.1145/3292500.3330895. URL https://doi.org/10.1145/3292500.3330895.

Ilya Loshchilov and Frank Hutter. Decoupled Weight Decay Regularization, jan 2019. URL http://arxiv.org/abs/1711.05101. arXiv:1711.05101 [cs, math].

Yuhong Luo and Pan Li. No Need to Look Back: An Efficient and Scalable Approach for Temporal Network Representation Learning, February 2024. URL http://arxiv.org/abs/2402.01964. arXiv:2402.01964 [cs] version: 1.

Asier Mujika, Florian Meier, and Angelika Steger. Approximating Real-Time Recurrent Learning with Random Kronecker Factors. *arXiv:1805.10842 [cs, stat]*, December 2018. URL http://arxiv.org/abs/1805.10842. arXiv: 1805.10842.

Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. Temporal Graph Networks for Deep Learning on Dynamic Graphs, October 2020. URL http://arxiv.org/abs/2006.10637. arXiv:2006.10637 [cs, stat].

Amauri H. Souza, Diego Mesquita, Samuel Kaski, and Vikas Garg. Provably expressive temporal graph networks. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/d029c97ee0db162c60f2ebc9cb93387e-Abstract-Conference.html.

Corentin Tallec and Yann Ollivier. Unbiased Online Recurrent Optimization. *arXiv:1702.05043 [cs]*, May 2017. URL http://arxiv.org/abs/1702.05043. arXiv: 1702.05043.

Rakshit Trivedi, Hanjun Dai, Yichen Wang, and Le Song. Know-Evolve: Deep Temporal Reasoning for Dynamic Knowledge Graphs, June 2017. URL http://arxiv.org/abs/1705.05742. arXiv:1705.05742 [cs].

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*. OpenReview.net, 2019. URL https://openreview.net/forum?id=HyePrhR5KX.

Xuhong Wang, Ding Lyu, Mengjian Li, Yang Xia, Qi Yang, Xinwen Wang, Xinguang Wang, Ping Cui, Yupu Yang, Bowen Sun, and Zhenyu Guo. APAN: Asynchronous Propagation Attention Network for Real-time Temporal Graph Embedding. In *Proceedings of the 2021 International Conference on Management of Data*, pp. 2628–2638, June 2021. doi: 10.1145/3448016.3457564. URL http://arxiv.org/abs/2011.11545. arXiv:2011.11545 [cs].

Yichen Wang, Nan Du, Rakshit Trivedi, and Le Song. Coevolutionary Latent Feature Processes for Continuous-Time User-Item Interactions. In *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016. URL https://proceedings.neurips.cc/paper_files/paper/2016/hash/53ed35c74a2ec275b837374f04396c03-Abstract.html.

Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. Inductive Representation Learning on Temporal Graphs, February 2020. URL http://arxiv.org/abs/2002.07962. arXiv:2002.07962 [cs, stat].

Hongkuan Zhou, Da Zheng, Xiang Song, George Karypis, and Viktor Prasanna. Disttgl: Distributed memory-based temporal graph neural network training. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701092. doi: 10.1145/3581784.3607056. URL https://doi.org/10.1145/3581784.3607056.

A Graph Recurrent Neural Networks (GRNNs)

We will consider an inference problem on sequences of events between entities in a network. We restrict ourselves to one or two types of entities and a single type of events for simplicity, although the models presented in this section would be trivial to generalize to more types of entities and events. We will first discuss previously proposed architectures within a general framework that can be applied to different tasks and training schemes. We will then review the various batching and training strategies proposed in past work.

A.1 A General Framework for GRNNs

One can define a dynamical system over the graph with a hidden state comprising a hidden state for every node, $i \in [1 \dots N]$, in the network: $\mathbf{h}(t) = \left(\mathbf{h}^{(1)}(t), \dots, \mathbf{h}^{(N)}(t)\right)$. Between edges (i.e. interaction events), this state can be subject to an autonomous evolution which we will assume to be time-invariant and similar and independent for each node in the network:

$$\mathbf{h}^{(i)}(t) = f\left(t - \tau, \mathbf{h}^{(i)}(\tau)\right) . \tag{6}$$

We can visualize this evolution as the blue trajectories in Figure 6. Here we express this time evolution in integral form, but it could instead be given as an ordinary differential equation such as a neural ordinary differential equation (NeuralODE) (Chen et al., 2019). Note that if the network is heterogeneous, (i.e., there are different types of entities), we could have different evolution laws for each entity type.

An edge, $e_k = (s_k, d_k, t_k, \mathbf{x}_k)$, is defined by a source entity, $s_k \in \mathcal{H}$, a destination entity, $d_k \in \mathcal{X}$, a timestamp, $t_k \in \mathbb{R}$ and a set of input features, $\mathbf{x}_k \in \mathcal{X}$. The states for both nodes associated with the edge are updated by a function, $g : \mathcal{H} \times \mathcal{H} \times \mathcal{X} \to \mathcal{H} \times \mathcal{H}$. This update allows information encoded in the state of one node to be propagated to the counter-party node of the edge and vice-versa. We assume this update to be time

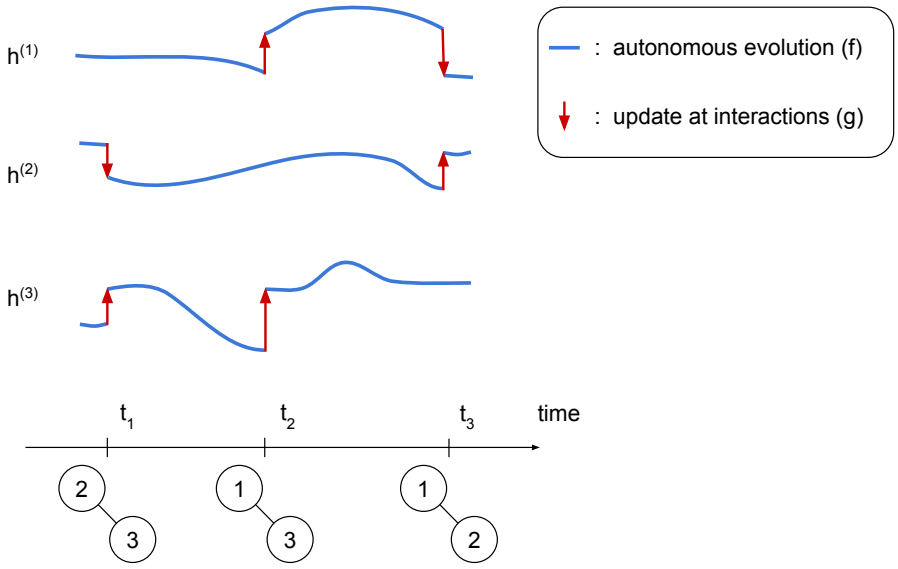

Figure 6: The dynamical system over the hidden states can in general contain two components: a function f encoding the evolution between interactions (blue), and a function g encoding the updates due to interaction events (red). In this example interactions happen at times t1, t2 and t3 between nodes 2-3, 1-3 and 2-3 respectively.

invariant for simplicity:

$$\left(\mathbf{h}^{(s_k)}(t_k^+), \mathbf{h}^{(d_k)}(t_k^+)\right) = g\left(\mathbf{h}^{(s_k)}(t_k^-), \mathbf{h}^{(d_k)}(t_k^-), \mathbf{x}_k\right) \tag{7}$$

$$\mathbf{h}^{(i)}(t_k^+) = \mathbf{h}^{(i)}(t_k^-), \quad i \notin \{s_k, d_k\}\ . \tag{8}$$

We can visualize this update as the red arrows in Figure 6.

In most settings, the state of a node is only needed for the end task at a time of an interaction involving that node. In this case, we can simplify the above model by merging the autonomous evolution f together with the state update g into a single update function h. There are two possibilities for this, depending on whether one needs the state immediately before or after a potential interaction for the end-task. As an example, for an edge classification or regression task where the target for the edge may depend directly on the edge features we could use the node states after the update. We can denote by $\tau^{(i)}(t)$ the last timestamp before t where an edge involving node i was observed and $\delta t_k^{(i)} = t_k - \tau^{(i)}(t_k)$ and write:

$$\left(\mathbf{h}_k^{(s_k)}, \mathbf{h}_k^{(d_k)}\right) = h\left(\delta t_k^{(s_k)}, \delta t_k^{(d_k)}, \mathbf{h}_{k-1}^{(s_k)}, \mathbf{h}_{k-1}^{(d_k)}, \mathbf{x}_k\right)\ , \tag{9}$$

where we also defined $\mathbf{h}_k^{(s_k)} = \mathbf{h}^{(s_k)}\left(t_k^{(s_k)+}\right)$. Here h is given by applying first f starting from the state immediately after the previous update of each node and then g for the edge update. This is depicted on the left (a) of Figure 7 and can be written as

$$h\left(\delta t_k^{(s_k)}, \delta t_k^{(d_k)}, \mathbf{h}_{k-1}^{(s_k)}, \mathbf{h}_{k-1}^{(d_k)}, \mathbf{x}_k\right) = g\left(f\left(\delta t_k^{(s_k)}, \mathbf{h}_{k-1}^{(s_k)}\right), f\left(\delta t_k^{(d_k)}, \mathbf{h}_{k-1}^{(d_k)}\right), \mathbf{x}_k\right)\ . \tag{10}$$

For a link prediction scenario one would instead like to answer the question of whether a link between two nodes is likely at a given time (or simply rank how likely potential links between different nodes are). For this task, the node states immediately before a (potential) interaction are necessary since no information about the (potential) edge can be considered. We can thus redefine $\mathbf{h}_k^{(s_k)} = \mathbf{h}^{(s_k)}\left(t_k^{(s_k)-}\right)$ and $\delta t_k^{(i)}$ to be

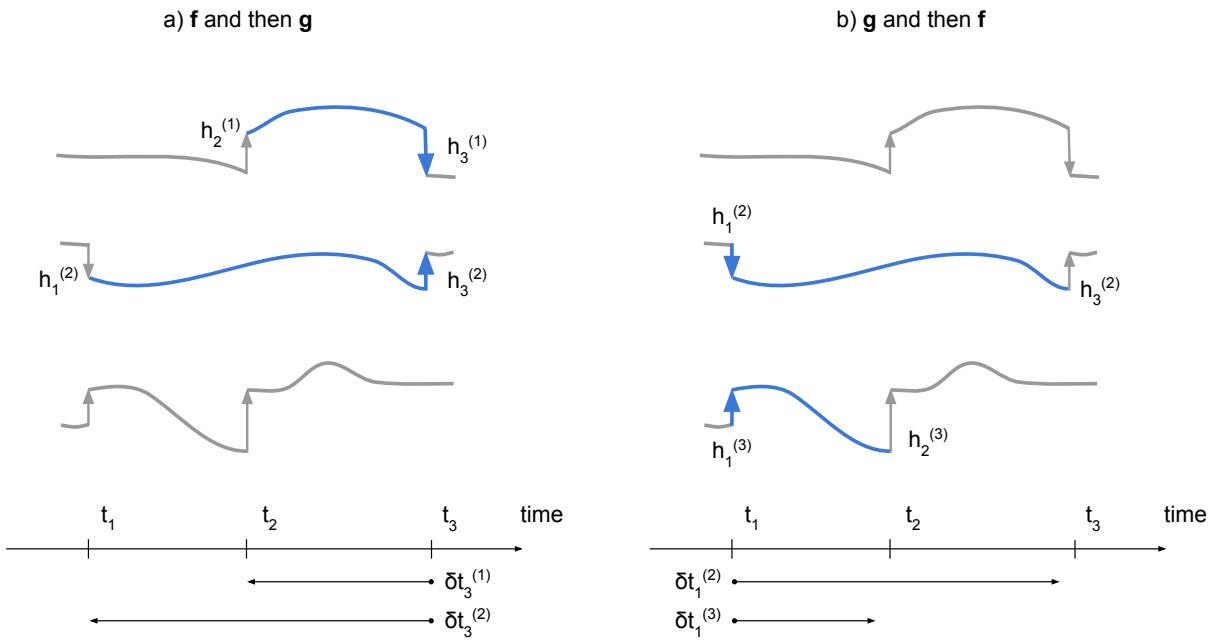

Figure 7: The two different possibilities for merging the autonomous evolution and event update into a single function. On the left (a) in blue is the update corresponding to the third event between node 1 and 2. The state of each node immediately after its last update is first propagated using the autonomous evolution, f, up to t_3 and then the update, g, is computed. On the right (b) in blue is the update corresponding to the first event between nodes 2 and 3. The state of each node immediately before the event is first updated using g and then propagated using f up until the next (potential) event where the state is needed. Note the different definitions of $h_k^{(i)}$ and $\delta t_k^{(i)}$. On the left they refer to the state immediately after event k and the time elapsed since the last update of the node before event k respectively. On the right they refer to the state immediately before event k and the time it will take until the next (potential) update of each node after event k respectively.

the time between edge k and a potential future interaction in Equation 9. h is then given by first applying the node updates, g, and then propagating the states using f. This is depicted on the right (b) of Figure 7. Note that in a practical application, the update for event k for a node i would only be computed at the time of a future potential event for node i, thus requiring that extra memory of the last event for each node be maintained. In the example of Figure 7, the update for event 1 could be computed at time t_3 for node 2 and t_2 for node 3. This is the approach taken in Rossi et al. (2020) where the update is potentially recomputed several times with different values of δt corresponding to the times that node i is evaluated as a possible link for another node (i.e., it is selected as a negative example) and when a future edge involving node i occurs.