

A Computational Time

Table 5: Test inference time comparison for snapshot and event based methods on DTDG datasets, we report the average result from 5 runs. Top three models are coloured by **First**, **Second**, **Third**.

	Method	UCI	Enron	Contacts	Social Evo.	MOOC
event	TGN [15]	1.07	1.71	137.57	24.04	50.04
	DyGFormer [28]	155.58	57.72	15423.99	349.22	OOM
	NAT [17]	3.82	8.39	596.22	148.43	299.00
	GraphMixer [37]	32.88	13.85	3542.88	132.39	OOM
	EdgeBank _∞ [16]	0.52	0.24	45.33	2.07	5.17
	EdgeBank _{tw} [16]	0.52	<u>0.25</u>	50.77	<u>2.45</u>	<u>6.12</u>
snapshot	HTGN (UTG) [12]	0.61	0.87	76.64	14.59	28.64
	GCLSTM (UTG) [13]	0.35	0.46	<u>40.83</u>	9.27	19.78
	EGCNo (UTG) [14]	<u>0.43</u>	0.45	40.62	7.35	15.49
	GCN (UTG) [1]	<u>0.50</u>	<i>0.31</i>	56.88	<i>6.40</i>	<i>13.30</i>

Table 5 shows the inference time for all methods on DTDG datasets. Table 6 shows the inference time for all methods on CTDG datasets. OOM means out of memory and OOT means out of time. We observe that snapshot-based models are at least one order of magnitude faster than event-based models such as NAT, DyGFormer and GraphMixer. In addition, the best performing model on most datasets, DyGFormer, is also consistently the slowest method.

B Evaluation Settings

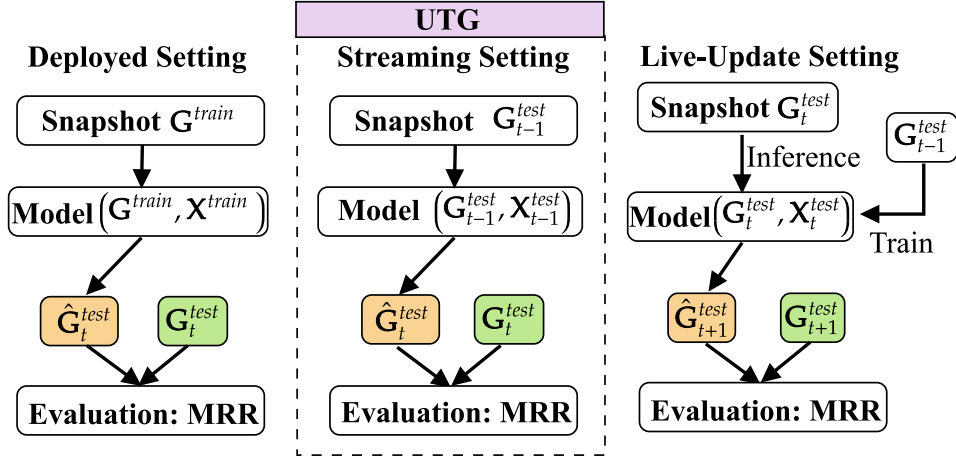


Figure 4: Different setting for evaluation of future link prediction include between *deployed*, *streaming* and *live-update* setting. UTG framework is designed for the streaming setting.

Deployed setting The *deployed setting* is often used as the evaluation setting for snapshot-based methods [12, 14]. In this setting, no information from the test set is passed to the model, and the node embeddings from the last training snapshot are used for predictions in all test snapshots.

Streaming setting event-based models often evaluate with the *streaming setting* [15, 17, 26, 28]. In this setting, information from the previously observed batches of events can be used to update the model however no information from the test set can be used to train the model.

Live-update Setting You et al. [18] proposed the *live-update setting* where the model weights are constantly updated to newly observed snapshots while predicting the next snapshot. To predict links

Table 6: Test inference time comparison for snapshot and event based methods on CTDG datasets, results reported from 5 runs. Top three models are coloured by **First**, **Second**, **Third**.

	Method	tgbl-wiki	tgbl-review	Reddit
event	TGN [15]	39.24	1137.69	286.79
	DyGFormer [28]	7196.52	26477.51	OOT
	NAT [17]	340.51	8925.21	1159.19
	GraphMixer [37]	1655.44	4167.63	7166.24
	EdgeBank _∞ [16]	20.06	140.25	26.91
	EdgeBank _{tw} [16]	20.67	143.49	27.08
snapshot	HTGN (UTG) [12]	28.96	718.17	117.05
	GCLSTM (UTG) [13]	20.54	436.30	82.88
	EGCNo (UTG) [14]	20.15	433.23	84.85
	GCN (UTG) [1]	18.25	384.51	78.78

in \mathbf{G}_{t+1} , first the observed snapshot \mathbf{G}_{t-1} are split into a training set and a validation set. The model is trained on \mathbf{G}_{t-1}^{train} while using \mathbf{G}_{t-1}^{val} for early stopping. Lastly, the trained model receives \mathbf{G}_t and predicts for \mathbf{G}_{t+1} .

Figure 4 illustrates the difference between these three settings. In this work, we focus on the streaming setting as it closely resembles the common use case where even after a model is trained, it is expected to incorporate new information from the data stream for accurate predictions.

C Temporal Graph Learning Methods

C.1 Snapshot-based Methods

snapshot-based methods receives a sequence of graph snapshots as input, representing the temporal graph at specific time intervals (hours, days, etc.). Therefore, DTDG methods are designed to process entire snapshot at once (often with a graph learning model) and then utilize mechanisms to learn temporal dependencies between snapshots. Example methods are as follows:

- **HTGN.** Many DTDG methods focus on learning structural and temporal dependencies in an Euclidean space thus omitting the complex and hierarchical properties which arises in real world networks. To address this, Yang et al. [12] proposed a Hyperbolic Temporal Graph Network (HTGN) which utilizes the exponential capacity and hierarchical awareness of hyperbolic geometric. More specifically, HTGN incorporates hyperbolic graph neural network and hyperbolic gated recurrent neural network to capture the structural and temporal dependencies of a temporal graph, implicitly preserving hierarchical information. In addition, the hyperbolic temporal contextual self-attention module is used to attend to historical states while the hyperbolic temporal consistency module ensures model stability and generalization.
- **GCLSTM.** To learn over a sequences of graph snapshots, Chen et al. [13] proposed a novel end-to-end ML model named Graph Convolution Network embedded Long Short-Term Memory (GC-LSTM) for the dynamic link prediction task. In this work, the LSTM act as the main framework to learn temporal dependencies between all snapshots if a temporal graph while GCN is applied on each snapshot to capture the structural dependencies between nodes. Two GCNs are used to learn the hidden state and the cell state for the LSTM and the decoder is a MLP mapping the feature at the current time back to the graph space. The design of GC-LSTM allows it to handle both link additions and link removals.
- **EGCN.** Existing approaches often require the knowledge of a node during the entire time span of a temporal graph while real world networks often changes its node set. To address this challenge, Pareja et al. [14] proposed the EvolveGCN (EGCN) model which captures the dynamic of the graph sequence by using an RNN to update the weight of a GCN. In this way, The RNN regulates the GCN model parameter directly and effectively performing model adaptation. This allows node changes because the learning is performed on the model itself, rather than specific sequence of node embeddings. Note that the GCN parameters are not trained and only computed from the RNN.

Empirically, the model achieves good performance for link prediction, edge classification and node classification on DTDGs.

- **PyG-Temporal.** PyTorch Geometric Temporal (PyG-Temporal) is an open-source Python library which combines state-of-the-art methods for neural spatiotemporal signal processing [22]. Many existing methods such as EGCN, GCLSTM and more are implemented directly in PyG-Temporal for research. PyG-Temporal is designed with a simple and consistent API following existing geometric deep learning library such as Pytorch Geometric [40]. Originally, PyG-Temporal are designed for node level regression tasks on datasets available exclusively within the framework. In this work, we apply PyG-Temporal models for the link prediction tasks on publicly available datasets.

C.2 Event-based Methods

Continuous Time Dynamic Graph (CTDG) methods receive a continuous stream of edges as input and make predictions over any possible timestamps. CTDG methods incorporate newly observed information into its predictions by updating its internal representation of the world. For efficiency, the stream of edges are divided into fixed size batches while predictions are made for each batch sequentially. To incorporate the latest information, edges from each batch becomes available to the model once the predictions are made. Different from DTDG, CTDG has no inherent notion of graph snapshots, models often track internal representations of a node over time and sample temporal neighborhoods surrounding the node of interest for prediction.

- **TGAT.** Xu et al. [41] argued that models for temporal graphs should be able to quickly generate embeddings in an inductive fashion when new nodes are encountered. The key component of the proposed Temporal Graph Attention (TGAT) layer is to combine the self-attention mechanism with a novel functional time encoding technique derived from Bochner’s theorem from classical harmonic analysis. In this way, a TGAT layer can efficiently learn from temporal neighborhood features as well as temporal dependencies. The functional time encoding provides a continuous functional mapping from the time domain to a vector space. The hidden vector of time then replaces positional encoding used in the self-attention mechanism.
- **TGN.** Rossi et al.. [15] introduce Temporal Graph Network (TGN), a versatile and efficient framework for dynamic graphs, represented as stream of timestamped events. TGN leverage a combination of memory modules and graph-based operators to improve computational efficiency. Essentially, TGN is a framework that subsumes several previous models as specific instances. When making predictions for a new batch, TGN first update the memory with messages coming from previous batches to allow the model to incorporate novel information from observed batches.
- **CAWN.** Causal Anonymous Walks (CAWs) [26] are proposed for representing temporal networks inductively to learn the laws governing the link evolution on networks such as the triadic closure law. CAWs, derived from temporal random walks, act as automatic retrievals of temporal network motifs, avoiding the need for their manual selection and counting. An anonymization strategy was also proposed to replace node identities with hitting counts from sampled walks, maintaining inductiveness and motif correlation. CAWN is a neural network model proposed to encode CAWs, paired with a CAW sampling strategy that ensures constant memory and time costs for online training and inference.
- **TCL.** TCL [42] effectively learns dynamic node representations by capturing both temporal and topological information. It features three main components: a graph-topology-aware transformer adapted from the vanilla Transformer, a two-stream encoder that independently extracts temporal neighborhood representations of interacting nodes and models their interdependencies using a co-attentional transformer, and an optimization strategy inspired by contrastive learning. This strategy maximizes mutual information between predictive representations of future interaction nodes, enhancing robustness to noise.
- **NAT.** In modeling temporal networks, the neighborhood of nodes provides essential structural information for interaction prediction. It is often challenging to extract this information efficiently. Luo et al. [17] propose the Neighborhood-Aware Temporal (NAT) network model that introduces a dictionary-type neighborhood representation for each node. NAT records a down-sampled set of neighboring nodes as keys, enabling fast construction of structural features for joint neighborhoods. A specialized data structure called N-cache is designed to facilitate parallel access and updates on GPUs.

- **EdgeBank.** EdgeBank [16] is a non-learnable heuristic baseline which simply memorizes previously observed edges. The surprisingly strong performance of EdgeBank in existing evaluation inspired the authors to also propose novel, more challenging and realistic evaluation protocols for dynamic link prediction.
- **DyGFormer DyGFormer.** Yu et al. [28] introduces a transformer-based architecture for dynamic graph learning. DyGFormer focuses on learning from nodes historical first-hop interactions and employs a neighbor co-occurrence encoding scheme to capture correlations between source and destination nodes through their historical sequences. A patching technique was also proposed to divide each sequence into patches for the transformer, enabling effective utilization of longer histories. *DyGLib* was also presented as a library for standardizing training pipelines, extensible coding interfaces, and thorough evaluation protocols to ensure reproducible dynamic graph learning research.

D Computing Resources

For our experiments, we utilized one of the following GPUs. The first option was NVIDIA A100 GPUs (40GB memory) paired with 4 CPU nodes. These nodes featured CPUs such as the AMD Rome 7532 @ 2.40 GHz with 256MB cache L3, AMD Rome 7502 @ 2.50 GHz with 128MB cache L3, or AMD Milan 7413 @ 2.65 GHz with 128MB cache L3, each equipped with 100GB memory. The second option was using NVIDIA V100SXM2 GPUs (16GB memory) alongside 4 CPU nodes, which housed Intel Gold 6148 Skylake CPUs @ 2.4 GHz, each with 100GB memory. Our last choice was to run experiments using NVIDIA P100 Pascal GPUs (12GB HBM2 memory) with 4 CPU nodes from Intel E5-2683 v4 Broadwell @ 2.1GHz with 100GB memory. Each experiment had a five-day time limit and was repeated five times, with results reported as averages and standard deviations. Notably, aside from methods adopted from the PyTorch Geometric library, several other models (assessed using their original source code or the [DyGLib repository](#)) encountered out-of-memory or out-of-time errors when applied to larger datasets.

E Model Configurations

For all methods and datasets, we employed the Adam optimizer with a two different learning rates namely 0.001 and 0.0002, and the configuration with the higher average performance was selected for reporting the results. Each experiment was repeated five times and the average and standard deviations were reported.

The train, validation, and test splits for *tgbl-wiki* and *tgbl-review* are provided by the TGB benchmark. For other datasets (namely, UCI, Enron, Contacts, Social Evo., MOOC, and Reddit), we used a chronological split of the data with 70%, 15%, and 15% for the training, validation, and test set, respectively, which is inline with previous studies [15–17, 25]. We set the batch size equal to 64 for NAT, and for all other models (i.e., TGN, DyGFormer, GraphMixer, EdgeBank, HTGN, GCLSTM, EGCNo, and GCN) the batch size was 200. For the experiments on CTDGs, we set the number of epoch equal to 40 and implemented an early stopping approach with a patience of 20 epochs and tolerance of 10^{-5} . For the experiments on DTDGs, the number of epochs was set to 200 with a similar early stopping approach. Dropout was set to 0.1. We set the number of attention heads equal to 2 for the models with an attention module, and node embedding size was fixed at 100. For TGN, the time embedding size was 100 and the memory dimension was specified as 172, with a message dimension of 100. For NAT, we set the *bias*= $1e-5$, and *replacement probability*=0.7. All other parameters were set according to the suggested values by Luo and Li [17]. The special hyperparameters of the DyGFormer and GraphMixer are set according to the recommendations presented by Yu et al. [28].