Temporal Graph Rewiring with Expander Graphs

Katarina Petrović [1] **Shenyang Huang** [2 3] **Farimah Poursafaei** [2 3] **Petar Veličković** [4]

Abstract

Evolving relations in real-world networks are often modelled by temporal graphs. Graph rewiring techniques have been utilised on Graph Neural Networks (GNNs) to improve expressiveness and increase model performance. In this work, we propose Temporal Graph Rewiring (TGR), the first approach for graph rewiring on temporal graphs. TGR enables communication between temporally distant nodes in a continuous time dynamic graph by utilising expander graph propagation to construct a message passing highway for message passing between distant nodes. Expander graphs are suitable candidates for rewiring as they help overcome the oversquashing problem often observed in GNNs. On the public `tgbl-wiki` benchmark, we show that TGR improves the performance of a widely used TGN model by a significant margin, our code repository is accessible at: `https://github.com/kpetrovicc/TGR`.

1. Introduction

Graph representation learning (Hamilton, 2020) aims to learn node representations on graph structured data to solve tasks such as node property prediction (Hamilton et al., 2017), link prediction (Ying et al., 2018) and graph property prediction (Gilmer et al., 2017). Graph neural networks (GNN) (Kipf & Welling, 2016; Veličković et al., 2017; Xu et al., 2018) capture relationships between nodes following message passing paradigm (Veličković, 2022). GNNs have been successfully applied to model many real-world networks such as biological networks (Johnson et al., 2023; Zitnik et al., 2018) and social networks (Ying et al., 2018).

[1]Instititute for Artificial Intelligence of Serbia, Novi Sad, Serbia [2]School of Computer Science, McGill University, Montréal, Canada [3]Mila – Quebec AI Institute, Montréal, Canada [4]Google DeepMind, London, United Kingdom. Correspondence to: Katarina Petrović <katarina.petrovic@ivi.ac.rs>.

Accepted as an extended abstract for the Geometry-grounded Representation Learning and Generative Modeling Workshop at the 41st International Conference on Machine Learning, ICML 2024, Vienna, Austria.

Oversquashing in GNNs. GNNs operate through message passing mechanism which aggregates information over a node's *direct* neighbourhood at each GNN layer. Hence, propagating information between nodes that are at k-hop distance requires k GNN layers. This leads to **information bottleneck**, since the node's receptive field in most underlying graph topologies grows *exponentially* with an increase of number of GNN layers. This information is further stored in fixed length embedding vectors, leading to a phenomena called **oversquashing**, causing a severe loss of information. In *static* GNNs, bottlenecks and oversquashing are addressed using **graph rewiring**, where underlying graph topology is altered such that it connects distant nodes. There are several methods applied to rewire static GNNs such as diffusion-based graph rewiring (Gasteiger et al., 2019) or reducing negative Ricci curvature (Topping et al., 2021), leading to an impressive boost in performance.

Temporal graph learning (TGL) (Kazemi et al., 2020) emerged from graph representation learning to study evolving relationships in temporal graph data. Given the dynamic nature of temporal graphs, temporal graph neural networks (TGNNs) (Longa et al., 2023) are developed to capture the evolution of graph topology introducing novel components such as temporal memory (Kazemi et al., 2020) and time-encoding (Xu et al., 2020). We provide related work on TGNNs in appendix A.2. TGNNs are underpinned by a static GNN message passing mechanism, meaning they are vulnerable to bottlenecks and oversquashing. Given additional temporal dimension, receptive field of a node in a temporal graph grows faster than in static graphs, leading to a higher presence of oversquashing and bottlenecks in the structure. A natural question arises: *Can we improve TGNN performance by introducing temporal graph rewiring aimed to relieve oversquashing and ensure global propagation between temporally distant nodes?*

Why rewiring on temporal graphs? Rewiring fundamentally calls for a dynamic nature of how the data exists in the real world. There might be a process different from the input graph structure that governs the message passing and information exchange in a temporal graph. Additionally, temporal graph rewiring can be seen as a way to resolve so-called *memory staleness* problem (Kazemi et al., 2020). By memory staleness we refer to a process occurring in TGNN

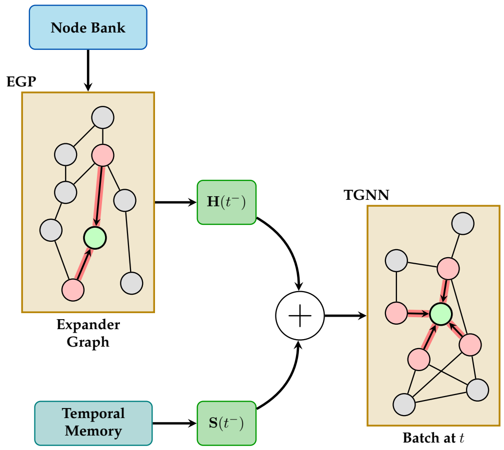

Figure 1: Node Bank stores all nodes *observed* up until batch t. In figure, timestamp t^- represents timestamp before t. Temporal memory stores observed node states mixed through expander message passing layer to generate expander embeddings $\mathbf{H}(t^-)$. TGNN input features are constructed by combining expander embeddings $\mathbf{H}(t^-)$ for previously observed nodes and temporal memory node states $\mathbf{S}(t^-)$ for unobserved nodes. The **green** node is a node of interest shown alongside its 1-hop neighbourhood.

temporal memory. Temporal memory is only updated if a node of interest interacts with another node in a graph, causing inactive nodes to appear as *stale*. This set-up poses significant issues in e.g social networks where if a user or a node is inactive for a period of time, it loses connections to its active neighbours. We propose temporal graph rewiring to add connections between temporally distant nodes, allowing information flow between nodes that might be inactive for a longer period of time.

In this work, we propose a novel TGR framework for *Temporal Graph Rewiring* with expander graphs. We leverage recent work by Deac et al. (2022) on expander graph propagation in combination with a TGNN base model to ensure global message passing between *temporally* distant nodes. It is shown that expander graphs satisfy four desirable criteria for graph rewiring: 1). the ability to propagate information globally within the input graph, 2). relieving bottlenecks and oversquashing, 3). subquadratic space and time complexity and 4). no additional pre-processing of the input graph (Deac et al., 2022).

Main Paper Contributions. Our main paper contributions include:

- **First Rewiring Method on Temporal Graphs.** In this work, we propose *Temporal Graph Rewiring* or TGR. To the best of our knowledge, TGR is the first approach which applies graph rewiring techniques on temporal graphs. TGR combines expander graph propagation (Deac et al., 2022) to rewire a base TGNN such as TGN (Rossi et al., 2020), while operating with minimal computational overhead.
- **Model-agnostic.** TGR is agnostic to the chosen expander message passing layer and helps alleviate the *over-squashing* and *memory staleness* problems in TGNNs. TGR is also agnostic to the base TGNN model as TGR provides dynamic node features thus has the potential to be applied to a wide range of TGL models.
- **Improved Performance**. We test TGR on a temporal link prediction task using `tgbl-wiki` data taken from a publicly available Temporal Graph Benchmark (TGB) (Huang et al., 2023). In section 4 we show that using TGR across four different expander message passing layers outperforms the base TGNN model.

2. Background

2.1. Static Graph Rewiring

In static GNNs, bottlenecks and oversquashing have been addressed with graph rewiring leading to an impressive boost in performance. There has been an extensive work done on applying graph rewiring to static graphs. For example diffusion-based graph rewiring (Gasteiger et al., 2019) diffuses additional edges in the graph with use of kernels such as PageRank (Brin, 1998). However as stated in (Topping et al., 2021), these models generally fail to reduce bottlenecks in the input graph. (Topping et al., 2021) introduces a method to modify a portion of edges with negative Ricci curvature to reduce oversquashing. However, this is shown to come with higher pre-processing cost and it is sub-optimal for analysing large graphs such as continuous time dynamic graphs which are the focus of our paper.

Rewiring with expander graphs. In this work we opt to work with expander graph propagation (Deac et al., 2022) as a starting point. Unlike other rewiring methods, expander graph propagation involves propagating messages over a graph that is *independent* of the input graph topology, by replacing input graph with an expander graph. In section 2.3, we describe in detail set-up by Deac et al. (2022), where expander and input graph GNN layers are alternated to compute node embeddings. In recent work by Giovanni et al. (2024), it is formally validated that such approach is effective in relieving oversquashing by decreasing overall effective resistance or commute time (Chandra et al., 1989). In fact, commute time is shown to be closely aligned with oversquashing, and it is stated that commute time in expander graphs grows linearly with the number of edges. In section 2.2, we provide further information on expander graph properties, showing they have high spectral gap while maintaining high sparsity making them an ideal candidate for graph rewiring. We are aware that expander graph propagation may not be the *only* way to do temporal graph rewiring. In this work, we aim to bring perspective of using expander graph propagation in combination with a temporal graph learning model and create pathway for further avenues to be explored in future work.

2.2. Expander Graphs

Following terminology laid out in (Deac et al., 2022), we provide a non-exhaustive list of most important expander graph topological properties in relation to temporal graph rewiring. Expander graphs are a fundamentally sparse family of graphs with number of edges scaling linearly with number of nodes ($|E| = O(n)$). They are characterised by two main properties: (**P1**) they have no bottlenecks and maintain high connectivity or spectral gap and (**P2**) they are efficiently precomputable through use of group operators and act as an *independent* graph topology over which input graph is rewired. Furthermore, their commute time or effective resistance scales linearly with the number of edges as stated in (Giovanni et al., 2024) showing that expander graphs are effective in reducing oversquashing.

Expander graphs have no bottlenecks. Spectral gap of the graph G with n nodes is related to the connectivity and existence of bottlenecks within its structure. We observe the eigenvalues of a graph-Laplacian L given as:

$$\lambda_0 < \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n, \tag{1}$$

where the first eigenvalue λ_1 is referred to as *spectral gap* or measure of connectivity. From Cheeger constant, it follows that a larger λ_1 leads to a better connectivity within graph G. Expander graph eigenvalues are extensively discussed in (Alon & Milman, 1985; Alon, 1986; Dodziuk, 1984; Tanner, 1984). From (Deac et al., 2022) it is known that Cheeger constant of expander graph families $\{G_i\}$ is lower-bounded by a positive constant $\epsilon > 0$, leading to high connectivity.

Expander graphs are efficiently precomputable. We follow construction given in (Deac et al., 2022) and we consider *Cayley graph* expander family. Cayley graphs are constructed through a use of the *special linear group* $\mathrm{SL}(2, \mathbb{Z}_n)$ as a generating set. Full details on pre-computing Cayley graphs can be found in the work done by (Deac et al., 2022; Kowalski, 2019; Selberg, 1965; Davidoff et al., 2003). It is known that a lower bound on the first eigenvalue of Cayley graph Laplacian is at least $3/16$ with full proofs given in (Kowalski, 2019; Davidoff et al., 2003). This means that Cayley graphs do not exhibit bottlenecks in their structure (satisfying **P1**).

Computing Cayley graphs comes with a low computational cost and no additional pre-processing of the input. It is shown in (Deac et al., 2022) that for an expander family $\{G_i\}$ of finite graphs with uniform upper bound on their vertex degree, the following inequality holds:

$$\mathrm{diam}(G_i) \leq k \log V(G_i), \tag{2}$$

where constant $k > 0$ and $\mathrm{diam}(G_i)$ is the diameter of graph G_i, yielding subquadratic time complexity. Due to their *low-diameter*, two expander nodes can reach each other within a *small* number of hops, relieving bottlenecks and offering efficient structure for global information propagation.

It is also important to note that the number of nodes in Cayley graphs grows cubically. As shown in (Deac et al., 2022), the number of nodes is computed as:

$$|V(\mathrm{Cay}(SL(2, \mathbb{Z}_n); S_n))| = n^3 \prod_{\text{prime } p|n} \left(1 - \frac{1}{p^2}\right). \tag{3}$$

2.3. Expander Graph Propagation

Expander Graph Propagation (EGP) framework is simplistic in nature and shows incredibly favourable properties for rewiring on static graphs: relieving information bottleneck and oversquashing, while maintaining low computational cost and subquadratic time. For a set of input node features $\mathbf{X}^{n \times d}$, where n is the number of nodes and d is the feature vector size, EGP operates by alternating GNN layers and propagating them over the adjacency matrix of the input graph and Cayley graph in a following manner:

$$\mathbf{H} = \mathrm{GNN}(\mathrm{GNN}(\mathbf{X}, \mathbf{A}), \mathbf{A}^{Cay}). \tag{4}$$

Here, $\mathbf{H}$ represents node embedding vector generated after EGP forward pass, and $\mathbf{A}$ and $\mathbf{A}^{Cay}$ are the adjacency matrices for the input and Cayley graph respectively. A GNN can be any classical GNN layer, such as graph attentional network (GAT) (Veličković et al., 2017):

$$\mathbf{h}_i = \Big\Vert_{k=1}^{K} \sigma \left(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^k \mathbf{W}^k \mathbf{x}_j \right). \tag{5}$$

In the equation above for multi-head-attention layer, i is a node of interest and $\mathcal{N}_i$ is its direct neighbourhood, α_{ij}^k are the attention weights of the k-th attention head and $\mathbf{W}^k$ is corresponding input linear transformation's weight matrix.

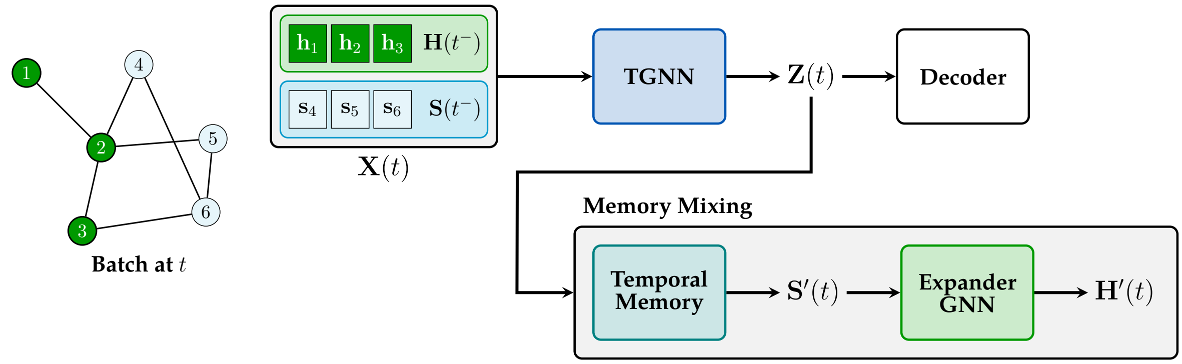

Figure 2: TGR batch processing. **Green** nodes are nodes that have been previously observed and **blue** nodes are the new nodes. Input node features $\mathbf{X}(t)$ are constructed by concatenating expander embeddings $\mathbf{H}(t^-)$ (for previously observed nodes), and node states $\mathbf{S}(t^-)$ (for new nodes). After computing temporal embeddings $\mathbf{Z}(t)$, temporal memory is updated and mixed with other previously observed node states to compute expander embeddings $\mathbf{H}'(t)$ and update expander memory.

3. Temporal Graph Rewiring

TGR combines expander graph propagation with use of memory module to reduce memory staleness and alleviate oversquashing in a base TGNN model. A detailed mechanism behind our framework is given in Figure 2 and algorithm in appendix A.4. During training, TGR stores observed nodes in Node Bank. After TGNN forward pass, TGR extracts observed node states from TGNN temporal memory and passes them to the expander message passing layer, which can be any classic static GNN layer. This process is called *memory mixing* and it is used to reduce memory staleness. Expander graphs introduce edges between temporally distant nodes in TGNN temporal memory, regardless whether they interacted with other nodes in the batch. Embeddings generated through memory mixing are then passed as input features to the base TGNN model.

As TGR essentially provides dynamic node features, it is agnostic to the underlying TGNN architecture thus it can be easily extended to other TGL models. In this work, we demonstrate how TGR integrates with a widely used TGN architecture (further described in appendix A.3). It is important to note that in our method, TGNN construction remains *intact*. We add memory mixing as a separate process to TGNN after TGNN temporal memory has been updated. This a difference to most graph rewiring set-ups, where rewiring alters individual GNN layers in the architecture.

Node Bank. Node Bank stores all nodes observed up to a temporal batch t. We define two classes of nodes in the batch: *previously observed* and *unobserved nodes* depending whether they have been observed by the TGNN model. Node Bank module is dynamically updated with new nodes at every temporal batch.

Memory Mixing. TGR introduces expander graph propagation through *memory mixing* to reduce memory staleness. After TGNN forward pass, TGNN temporal memory module is updated with new node states. Consider $\mathbf{S}'(t)$ which consists of updated node states for all nodes in the Node Bank. Expander embeddings $\mathbf{H}'(t)$ are generated through an expander message passing layer:

$$\mathbf{H}'(t) = \text{GNN}(\mathbf{S}'(t), \mathbf{A}^{Cay}). \tag{6}$$

Here, $\mathbf{A}^{Cay}$ is the expander graph adjacency matrix. In our set-up, we pre-compute a large expander graph from *Cayley graph* family with N nodes, where N is the number of nodes in the training dataset. We use entire expander graph for passing messages in the expander message passing layer. When expander graph is smaller than number of nodes in the node bank, we compute a new expander graph of size $(1+m)N$, where $m < 1$.

Expander Memory. Expander memory $\mathbf{H}'(t)$ stores computed expander embeddings and provides input features for previously observed nodes for TGNN forward pass.

Input Features for TGNN. The input feature vector $\mathbf{X}(t)$ for TGNN forward pass is constructed through a concatenation ($\|$) of expander embeddings $\mathbf{H}(t^-)$ (for previously observed nodes) from expander memory and TGNN node states $\mathbf{S}(t^-)$ (for new nodes) from temporal memory:

$$\mathbf{X}(t) = \mathbf{H}(t^-) \| \mathbf{S}(t^-). \tag{7}$$

TGNN Embedding. TGNN embedding module generates temporal embeddings $\mathbf{Z}(t)$ by aggregating information from node's direct temporal neighbourhood:

Table 1: Results for *dynamic link property prediction* on `tgbl-wiki`.

Method	**MRR (%)**		**Improvement (%)**		**Compute Time** (Sec)	
	Val.	Test	Val.	Test	Val.	Test
TGN	51.5±2.9	46.8±2.2	-	-	87	86
TGR-GCN	56.5±1.2	50.3±1.2	5.0	3.6	92	94
TGR-GAT	**57.7**±1.2	**53.1**±1.1	**6.3**	**6.3**	95	99
TGR-GIN	55.9±1.5	51.3±1.9	4.5	4.5	93	94
TGR-GATv2	54.6±0.9	46.7±3.4	3.2	-0.1	96	97

$$\mathbf{Z}(t) = \text{TGNN}(\mathbf{X}(t), \mathbf{A}(t)). \quad (8)$$

We define temporal neighbourhood N_i^t, of node i at time t as a set of temporal events where every edge $(i, j) \in N_i^t$. In this work we opt for a widely used TGN (Rossi et al., 2020) for a choice of a TGNN embedding. For node i its TGN embedding $\mathbf{z}_i(t)$ is computed as:

$$\mathbf{z}_i(t) = \text{MLP}(\mathbf{x}_i(t) \| \tilde{\mathbf{z}}_i(t)), \quad (9)$$

$$\tilde{\mathbf{z}}_i(t) = \text{MultiHeadAttention}(\mathbf{q}(t), \mathbf{K}(t), \mathbf{V}(t)), \quad (10)$$

$$\mathbf{q}(t) = \mathbf{x}_i(t) \| \phi(0), \quad (11)$$

$$\mathbf{K}(t) = \mathbf{V}(t) = \mathbf{C}(t), \quad (12)$$

$$\mathbf{C}(t) = [\mathbf{x}_1 \| \mathbf{e}_{i1} \| \phi(t - t_1), ..., \mathbf{x}_N \| \mathbf{e}_{iN} \| \phi(t - t_N)] \quad (13)$$

MultiHeadAttention represents multi-head-attention layer (Vaswani et al., 2023), and ϕ is the generic time encoding (Xu et al., 2020). Query $\mathbf{q}(t)$ relates to node of interest and the keys $\mathbf{K}(t)$ and values $\mathbf{V}(t)$ relate to its direct neighbours.

4. Experiments

Evaluation Setting. In this section, we evaluate TGR framework on the dynamic link property prediction task from TGB (Huang et al., 2023) on the `tgbl-wiki` dataset. Predicting links in continuous time dynamic graphs consists of computing temporal node embeddings and predicting a probability of an edge formation and associated edge features. We compare TGR performance to that of the TGN base model using *four* different baselines: *TGR-GCN* (Kipf & Welling, 2016), *TGR-GAT* (Veličković et al., 2017), *TGR-GIN* (Xu et al., 2018) and *TGR-GATv2* (Brody et al., 2022).

TGR Implementation. We implement TGR on top of the TGN base model used in TGB (Huang et al., 2023) to include memory mixing, expander memory and node bank module. Expander memory is initialised as an empty tensor which matches temporal memory size in (Rossi et al., 2020) set-up. We also initialise node bank and expander graph of size equal to the size of training dataset. In all experiments we set learning rate $lr = 5 \cdot 10^{-4}$ and run models with a tolerance of 50 epochs for a maximum of 100 epochs during training, test and validation. We maintain same batch size as in (Rossi et al., 2020) which contains 200 temporal events.

Results. We show that TGR significantly outperforms TGN on a temporal link prediction task. The best performing result is achieved using TGR-GAT showing 6% improvement across Validation and Test MRR comparison to TGN trained with our hyperparameters. Full table of results is given in table 1. Best results using GAT reiterate importance of prioritising most important neighbours in random selection of expander edges between observed nodes.

5. Conclusion

In this work, we proposed TGR to address gaps in TGNNs: bottlenecks, oversquashing and memory staleness. We show that using expander graphs to rewire temporal graphs is an optimal method to relieve oversquahsing while adding minimal cost overhead. We are aware that expander graph propagation may not be the *best* way of doing temporal graph rewiring. The purpose of this work is to conceptually propose temporal graph rewiring as a solution to oversquashing using method we know is efficient in relieving oversquashing in static GNNs. In future work, we hope to explore other graph rewiring methods and compare their performance to TGR. We have empirically tested TGR on `tgbl-wiki` dataset and shown that TGR significantly outperforms base TGN model. In future work, we intend to test TGR on tasks such as node and and graph property prediction and larger datasets available on TGB Benchmark.

Acknowledgements

We thank João Araújo and Simon Osindero for their comments and for reviewing the paper prior to submission. We thank the TGB team for creating the platform which allowed us to rapidly test and benchmark our method. We also thank Emanuele Rossi, author of TGN, for his useful comments and feedback while we were developing TGR.

References

Alon, N. Eigenvalues and expanders. *Combinatorica*, 6(2): 83–96, 1986.

Alon, N. and Milman, V. D. $\lambda 1$, isoperimetric inequalities for graphs, and superconcentrators. *Journal of Combinatorial Theory, Series B*, 38(1):73–88, 1985.

Brin, S. The pagerank citation ranking: bringing order to the web. *Proceedings of ASIS, 1998*, 98:161–172, 1998.

Brody, S., Alon, U., and Yahav, E. How attentive are graph attention networks?, 2022.

Chandra, A. K., Raghavan, P., Ruzzo, W. L., and Smolensky, R. The electrical resistance of a graph captures its commute and cover times. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pp. 574–586, 1989.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Davidoff, G. P., Sarnak, P., and Valette, A. *Elementary number theory, group theory, and Ramanujan graphs*, volume 55. Cambridge university press Cambridge, 2003.

Deac, A., Lackenby, M., and Veličković, P. Expander graph propagation. In Rieck, B. and Pascanu, R. (eds.), *Proceedings of the First Learning on Graphs Conference*, volume 198 of *Proceedings of Machine Learning Research*, pp. 38:1–38:18. PMLR, 09–12 Dec 2022. URL https://proceedings.mlr.press/v198/deac22a.html.

Dodziuk, J. Difference equations, isoperimetric inequality and transience of certain random walks. *Transactions of the American Mathematical Society*, 284(2):787–794, 1984.

Gasteiger, J., Weißenberger, S., and Günnemann, S. Diffusion improves graph learning. *Advances in neural information processing systems*, 32, 2019.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *International conference on machine learning*, pp. 1263–1272. PMLR, 2017.

Giovanni, F. D., Rusch, T. K., Bronstein, M. M., Deac, A., Lackenby, M., Mishra, S., and Veličković, P. How does over-squashing affect the power of gnns?, 2024.

Hamilton, W., Ying, Z., and Leskovec, J. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Hamilton, W. L. *Graph representation learning*. Morgan & Claypool Publishers, 2020.

Hochreiter, S. and Schmidhuber, J. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Huang, S., Poursafaei, F., Danovitch, J., Fey, M., Hu, W., Rossi, E., Leskovec, J., Bronstein, M., Rabusseau, G., and Rabbany, R. Temporal graph benchmark for machine learning on temporal graphs, 2023.

Johnson, R., Li, M. M., Noori, A., Queen, O., and Zitnik, M. Graph ai in medicine. *arXiv preprint arXiv:2310.13767*, 2023.

Kazemi, S. M., Goel, R., Jain, K., Kobyzev, I., Sethi, A., Forsyth, P., and Poupart, P. Representation learning for dynamic graphs: A survey. *Journal of Machine Learning Research*, 21(70):1–73, 2020.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Kowalski, E. *An introduction to expander graphs*. Société mathématique de France Paris, 2019.

Longa, A., Lachi, V., Santin, G., Bianchini, M., Lepri, B., Lio, P., Scarselli, F., and Passerini, A. Graph neural networks for temporal graphs: State of the art, open challenges, and opportunities, 2023.

Rossi, E., Chamberlain, B., Frasca, F., Eynard, D., Monti, F., and Bronstein, M. Temporal graph networks for deep learning on dynamic graphs, 2020.

Selberg, A. On the estimation of fourier coefficients of modular forms. In *Proceedings of Symposia in Pure Mathematics*, pp. 1–15. American Mathematical Society, 1965.

Tanner, R. M. Explicit concentrators from generalized n-gons. *SIAM Journal on Algebraic Discrete Methods*, 5 (3):287–293, 1984.

Topping, J., Di Giovanni, F., Chamberlain, B. P., Dong, X., and Bronstein, M. M. Understanding over-squashing and bottlenecks on graphs via curvature. *arXiv preprint arXiv:2111.14522*, 2021.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., and Bengio, Y. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Veličković, P. Message passing all the way up, 2022.

Xu, D., Ruan, C., Korpeoglu, E., Kumar, S., and Achan, K. Inductive representation learning on temporal graphs, 2020.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*, pp. 974–983, 2018.

Zitnik, M., Agrawal, M., and Leskovec, J. Modeling polypharmacy side effects with graph convolutional networks. *Bioinformatics*, 34(13):i457–i466, 2018.

A. Appendix

A.1. Temporal Graphs

Temporal graphs are characterised by an evolving underlying graph topology, thus making them suitable for modelling complex dynamic networks and evolving interactions between nodes. In this work we focus on continuous time dynamic graphs as they model events which can occur at *any* point in time, thus making them suitable for modelling a majority of real-world networks. Continuous time dynamic graphs (CTDGs) are defined as a sequence of chronologically sorted events that can be split into node-wise or edge-wise events. Node-wise events $\mathbf{v}_i(t)$ include addition and deletion of node i as well as change in node features at time t. Edge-wise events $\mathbf{e}_{i,j}(t)$ relate to an edge ij with endpoint nodes i and j and include formation and deletion of edges in a graph at time t.

A.2. Temporal Graph Neural Networks

TGL has been a popular topic in graph machine learning community focusing on modelling evolutionary behaviour of complex dynamic interaction networks and leading to development of temporal graph neural network (TGNN) architectures (Longa et al., 2023). Following terminology introduced in (Longa et al., 2023) TGNNs can be classified into snapshot-based architectures operating over discrete time dynamic graphs (DTDGs) and event-based architectures operating over continuous time dynamic graphs (CTDGs). CTDGs capture events that can happen at any time, while DTDGs capture events that happen in discrete time intervals. In this work we focus on CTDGs, as most real-world networks operate over a continuous time domain.

Event-based TGNNs such as Neighborhood-aware Scalable Temporal Network (NAT) (Johnson et al., 2023) or Temporal Graph Network (TGN) (Rossi et al., 2020) are characterised by use of graph-based operators in combination with novel model components such as temporal memory (Kazemi et al., 2020) and time-encoding (Xu et al., 2020). TGNNs use temporal memory to store long-standing relationships between observed node states used to generate dynamic node embeddings. Similarly to static GNNs, TGNNs operate through aggregating information from a local temporal neighbourhood to generate temporal embeddings following a message passing paradigm.

A.3. Temporal Graph Network

Temporal Graph Network (TGN) (Rossi et al., 2020) serves as a base model to TGR framework to generate temporal embeddings $\mathbf{Z}(t)$. TGN proposes a combination of temporal memory and graph-based operators for learning on continuous time dynamic graphs represented as chronologically sorted sequences of time-based events.

A unique advantage of TGN is use of temporal memory to store and update node states. TGN memory module $\mathbf{S}(t)$ stores memory state $\mathbf{s}_i(t)$ for each node i in the graph. When a node interacts with another node through a node-wise or edge-wise event, node states of the nodes involved in the event are updated in the memory through a recurrent neural network such as LSTM (Hochreiter & Schmidhuber, 1997) or GRU (Cho et al., 2014). The purpose of a memory module is to store historical information and long-term dependencies of a node in a compressed format.

A.4. TGR: Algorithm

This section provides full algorithm for TGR batch processing.

Get new and observed node IDs. At the beginning of batch processing, we extract 200 node IDs and store them in n_{id}. We then compare n_{id} to node bank node IDs n_{id}^{obs} and extract n_{id}^{seen} where we store previously observed nodes and n_{id}^{new} where we store unobserved nodes. After extracting n_{id}^{seen} and n_{id}^{new}, we update node bank n_{id}^{obs} to use in the next batch.

Compute input node features $\mathbf{X}(t)$. We compute input node features $\mathbf{X}(t)$ as previously discussed in section 3 through concatenating node n_{id}^{seen} features from expander memory and n_{id}^{new} from TGNN temporal memory.

TGNN Forward Pass. In TGN implementation $\mathbf{X}(t)$ includes node states for all node IDs in n_{id}. The main difference between TGN implementation and ours is a different set of input node features $\mathbf{X}(t)$. After forward pass, TGN temporal memory is updated with new node states.

Extract node states from Temporal Memory. We extract updated node states $\mathbf{S}'(t)$ for node IDs n_{id}^{obs} from node bank. We pad $\mathbf{S}'(t)$ to match expander graph size. Note that $\mathbf{A}^{Cay}$ remains the same throughout batch processing unless the node bank size grows larger than the size of expander graph, as previously described in section 3.

Memory Mixing Memory mixing module computes expander embeddings $\mathbf{H}'(t)$ by passing $\mathbf{S}'(t)$ through an expander message passing layer. After memory mixing expander memory is updated to include new expander embeddings.

Algorithm 1 TGR Batch Processing

Require: ExpanderMemory, TemporalMemory, Node Bank

1. Get new and observed node IDs

$n_{id}^{new} \leftarrow n_{id} \setminus n_{id}^{obs}$ {Identify new nodes.}

$n_{id}^{seen} \leftarrow n_{id} \cap n_{id}^{obs}$ {Identify observed nodes.}

$n_{id}^{obs} \leftarrow n_{id}^{obs} \cup n_{id}^{new}$ {Update the Node Bank.}

2. Compute input node features $\mathbf{X}(t)$

$\mathbf{X}(t)[n_{id}^{new}]$: $\mathbf{S}(t^-) \leftarrow$ TemporalMemory(n_{id}^{new})

$\mathbf{X}(t)[n_{id}^{seen}]$: $\mathbf{H}(t^-) \leftarrow$ ExpanderMemory(n_{id}^{seen})

3. TGNN Forward Pass

$\mathbf{Z}(t) \leftarrow$ TGNN$(\mathbf{X}(t), \mathbf{A}(t))$

4. Extract node states from Temporal Memory

$\mathbf{S}'(t) \leftarrow$ TemporalMemory(n_{id}^{obs})

Pad $\mathbf{S}'(t)$ to match $\mathbf{A}^{Cay}$ size: $\mathbf{S}'(t) \leftarrow$ Pad$(\mathbf{S}'(t))$

5. Memory Mixing

$\mathbf{H}'(t) \leftarrow$ GNN$(\mathbf{S}'(t), \mathbf{A}^{Cay})$

Update ExpanderMemory $\leftarrow \mathbf{H}'(t)$

A.5. Experiments

A.5.1. Datasets

We leverage availability of temporal graph data in the TGB Benchmark (Huang et al., 2023) to validate TGR performance. TGB Benchmark collects a variety of real-world dynamic datasets which contain temporal interactions in many real-world networks such as flights, transactions and beyond.

tgbl-wiki. In this work we explore `tgbl-wiki` dataset which stores dynamic information about a co-editing network on Wikipedia pages over a span of one month. The data is stored in a bi-partite temporal graph where nodes represent either editors or wiki-pages they interact with. An edge represents an action user takes when editing a Wikipedia page and edge features contain textual information about a page of interest. The goal is to predict existence and nature of links between editors and Wikipedia pages at a future timestamp.

A.5.2. Model Parameters

We do not perform extensive hyperparameter tuning in comparison to TGN implementation (Huang et al., 2023) (see reported table 2). We match expander memory and embedding dimension to temporal memory.

Table 2: Model Hyperparameters.

	Value
Temporal Memory Dimension	100
Node Embedding Dimension	100
Time Embedding Dimension	100
Expander Memory Dimension	100
Expander Embedding Dimension	100
# Attention Heads	2
Dropout	0.1