APMTH 220 Geometric Methods for Machine Learning **Spring 2025**

Dada-DyGNN: Deletion-Augmented Streaming GNNs

Name: Jerry Liang *Submission Date: May 9, 2025*

1 Abstract

Graph neural networks (GNNs) have proved to be a formidable tool in learning useful representations for node and edge features. However, most existing work focus on static graphs, ignoring the complex dynamic nature of most real-world applications. In particular, existing temporal state-of-the-art methods only consider interactions in the form of edge insertions (e.g., molecular bonds, citation history, etc.) and neglect edge deletions. Many applications (e.g., social networks), however, contain rich information through edge deletions (e.g., "un-friending"). In Dada-DyGNN, we modify existing work in Ada-DyGNN to incorporate edge deletions with temporal decay in reinforcement learning context. Our work differs from existing literature since we do not want to explicitly "unlearn" that an interaction ever occurred– rather we wish to leverage all past history to learn a rich embedding useful for the downstream task of link-prediction. Our contributions are twofold: **(1)** a theoretical redesign of the model architecture and reward scheme; and **(2)** open-source implementation with deletion-augmentation and patches of existing bugs in Ada-DyGNN. Results are limited due to our compute constraints, but show promise given hyperparameter adjustments and ample resources in converging to a final neighbor update policy.

Our code base can be found here: https://github.com/J3rry-L/dada-dygnn

2 Introduction and Related Works

Graph neural networks (GNNs) are a type of machine learning model architecture acting on graph data. Characterized by a "message passing" process, node embeddings are iteratively updated as neighbors exchange messages. The learned representations reflect both structural and topological data (from neighborhood propagation), while also incorporating attribute data (from the nodes/edges themselves). As such, GNNs and their variants have empirically worked well for tasks ranging from social networks, to chemistry, to natural language processing.

However, standard GNNs assume that input data is static—in the real world, relationships are dynamic and change over time. This temporal data is crucial information ignored in the traditional GNNs. Therefore, we consider streaming-augmented GNNs, which update nodes in-real-time and on-the-fly as edges are inserted. We generalize this existing architecture to support edge deletion, a common graph update often ignored or unsupported by current state-of-the-art implementations.

Dynamic GNNs (DyGNNs) [1], first pioneered in 1997, were the first graph neural networks that allowed for streamed edges. Despite theoretical limitations [2], empirical performance is impressive, leading to revitalized interest and numerous novel variants of DyGNNs—continuous time networks [3], autoencoder models (DynGEM) [4], random walks (JODIE) [5], temporal graph attention [6], temporal graph networks [7], and reinforcement learning augmented methods (Ada-DyGNN) [8]. These methods, however, focus on an "insert-only" rather than an "insert-delete" paradigm [9], the latter of which is arguably more reflective of real world networks (e.g., social network friendships/follows). We extend Ada-DyGNN with **D**elete-Augmented **Ada**ptive **Dy**namic **GNN** (Dada-DyGNN) to allow for this insert-delete paradigm.

Edge deletions in graph streaming and link-prediction have been sparsely studied. The most pertinent publications in this area are GNNDelete [10] and Decoupled GNNs [11]. GNNDelete focuses on the task of "unlearning", which has the explicit goal that edge removals should leave node representations as if

the edge was never added in the first place. While a useful property, this is not what we wish to encode with Dada-DyGNN, since the timestamps and update history should ideally be kept as useful context for downstream tasks.

Decoupled GNNs are closer to our work—the core aim, however, of this paper is to offer a unified framework for continuous and discrete time dynamic graphs (CTDG/DTDG) through snapshots G_t. Deletions in this decoupled scheme correspond to purely negating a weight update (from $d(u) + w_{(u,v)}$ to $d(u) - w_{(u,v)}$), which does not offer much sophistication besides partial unlearning. Likewise, this paper focuses on the traditional formulation of link-prediction, which predicts only whether (at a future timestamp) an edge will be inserted, not also whether an existing edge will be deleted.

In Dada-DyGNN, we incorporate nuanced temporal update information (rather than direct unlearning) in specifically an RL context, and test harder downstream tasks of both insert & delete link-predictions.

3 Background and Notation

Dada-DyGNN is built on top of the existing Ada-DyGNN model architecture. Therefore, we briefly discuss the three core parts of this existing RL scheme of *Robust Knowledge Adaptation*: **(1)** the *Time-Aware Attentional Aggregating Module* (hereby T3AM); **(2)** the *Reinforced Neighbor Selection Module* (hereby RNSM); and **(3)** *Downstream Link-Prediction*. In brief, T3AM incorporates temporal data into the message passing paradigm; RNSM uses these temporal-augmented intermediate states to choose which neighbors to retain versus update; and link-prediction is done after representations are learned through cosine similarity (with highly similar nodes likelier to have an edge inserted at the next timestamp). We discuss each module in detail below, and our modified Proposed Approach in Section 4.

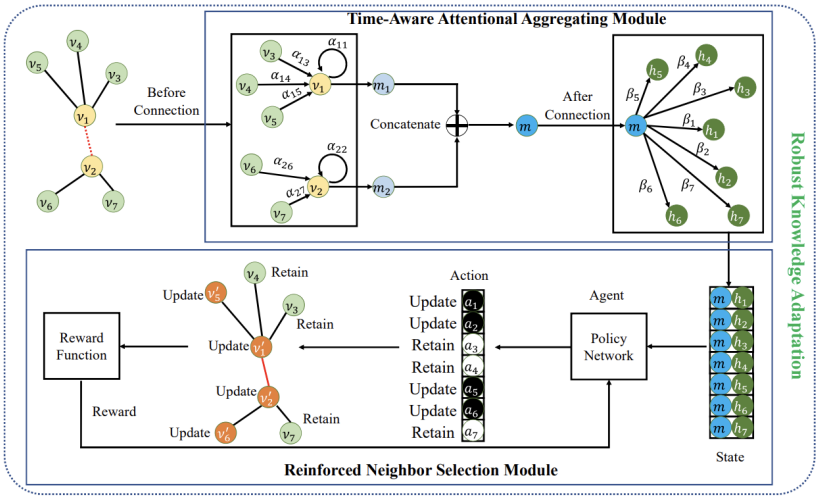

Figure 1: This figure summarizes the Robust Knowledge Adaption procedure in Ada-DyGNN.

3.1 Time-Aware Attentional Aggregating Module

T3AM combines the traditional attention mechanism of GAT with temporal-related prior information. In particular, given a new interaction, i.e., edge insertion, (v_s, v_d, t) with source v_s, destination v_d, at time t; and historical interactions with neighbors $(v_s, v_1, t_1), \ldots, (v_s, v_k, t_k)$, T3AM computes intermediate states $\mathbf{h}_i(t)$ used to update node embeddings $\mathbf{x}_i(t)$.

In detail, we define $\Delta t_i = t - t_i$ and use a simple time-decay function $\phi(\Delta t) = \frac{1}{1+\Delta t}$. The time-aware

attentional coefficient between v_s, v_i, capturing the importance of v_i on v_s, is defined as

$$\alpha_{si} = \frac{\exp\left(\sigma\left(\mathbf{a}^\top \left[\mathbf{W}_g \mathbf{x}_s(t) \,\|\, \phi(\Delta t_i)\mathbf{W}_g \mathbf{x}_i(t)\right]\right)\right)}{\sum_{j\in\mathcal{N}_s(t)} \exp\left(\sigma\left(\mathbf{a}^\top \left[\mathbf{W}_g \mathbf{x}_s(t) \,\|\, \phi(\Delta t_j)\mathbf{W}_g \mathbf{x}_j(t)\right]\right)\right)}$$

where $\mathbf{W}_g \in \mathbb{R}^{d_m/2 \times d_n}$ is a learnable weight matrix; $\mathbf{a} \in \mathbb{R}^{d_m}$ is a learnable weight vector; σ is the LeakyReLU activation function; and $\|$ represents concatenation. Then the message for v_s is

$$\mathbf{m}_s(t) = \sigma_1\left(\sum_{i\in\mathcal{N}_s(t)} \alpha_{si}\, \phi(\Delta t_i)\, \mathbf{W}_g \mathbf{x}_i(t)\right) \in \mathbb{R}^{d_m/2}$$

where σ_1 is the ReLU function, and similarly for $\mathbf{m}_d(t)$. The interaction message is hence

$$\mathbf{m}(t) = \mathbf{m}_s(t) \,\|\, \mathbf{m}_d(t) \,\|\, \sigma_1(\mathbf{W}_e \mathbf{e}(t)) \in \mathbb{R}^{2d_m}$$

where $\mathbf{W}_e \in \mathbb{R}^{d_m \times d_e}$ is a learnable weight matrix and $\mathbf{e}(t)$ are edge features. Finally, we have the attentional coefficient of node v_i to be

$$\beta_i = \frac{\exp\left(\sigma_2\left(\phi(\Delta t_i)\mathbf{x}_i(t) \cdot \mathbf{W}_p \mathbf{m}(t)\right)\right)}{\sum_{j\in\mathcal{N}_{s\cup d}(t)} \exp\left(\sigma_2\left(\phi(\Delta t_j)\mathbf{x}_j(t) \cdot \mathbf{W}_p \mathbf{m}(t)\right)\right)}$$

with intermediate state

$$\mathbf{h}_i(t) = \beta_i \mathbf{W}_p \mathbf{m}(t) \in \mathbb{R}^{d_n}$$

where $\mathbf{W}_p \in \mathbb{R}^{d_n \times 2d_m}$ is a learnable weight matrix. This final $\mathbf{h}_i(t)$ is passed onto the RNSM, and encodes both structural (attention mechanism) and temporal information through $\mathbf{m}(t)$'s reliance on time-decay $\phi(\Delta t)$. Note that the original paper erroneously omitted the dot product.

3.2 Reinforced Neighbor Selection Module

RNSM takes in the intermediate embedding from T3AM and calculates state $\mathbf{s}_i(t) = \mathbf{h}_i(t) \,\|\, \mathbf{x}_i(t)$. Then the RL agent, with policy π picks an action $a_i \in \{0, 1\}$, representing whether the agent decides to update the node v_i. The policy consists of two fully-connected layers

$$\pi(\mathbf{s}_i(t)) = \sigma_2(\mathbf{W}_1 \sigma_1(\mathbf{W}_2 \mathbf{s}_i(t))))$$

where σ_1 and σ_2 are ReLU and sigmoid activation functions, respectively; and $\mathbf{W}_1 \in \mathbb{R}^{1\times d_n}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_n \times 2d_n}$ are learnable weight matrices.

Then the embedding of node $v_i \in \mathcal{N}_{s\cup d}$ is updated as follows

$$\mathbf{x}_i(t+) = \begin{cases} \sigma_1(\mathbf{W}_u(\mathbf{x}_i(t) \,\|\, \mathbf{h}_i(t)) & \text{if } a_i = 1, \\ \mathbf{x}_i(t) & \text{if } a_i = 0 \end{cases}$$

where $\mathbf{W}_u \in \mathbb{R}^{d_n \times 2d_n}$ is learnable.

The reward for this policy (motivated by DynGEM) is designed to preserve historical topological information and local stability using cosine similarity as follows

$$r = \frac{\sum_{i\in\mathcal{N}_s^*(t)} \cos\left(\mathbf{x}_s(t+), \mathbf{x}_i(t+)\right)}{|\mathcal{N}_s^*(t)|} + \frac{\sum_{i\in\mathcal{N}_d^*(t)} \cos\left(\mathbf{x}_d(t+), \mathbf{x}_i(t+)\right)}{|\mathcal{N}_d^*(t)|}$$

where policies with parameters θ are updated with learning rate η through self-critical training as

$$\theta \leftarrow \theta + \eta \frac{1}{|\mathcal{N}_{s\cup d}(t)|} \sum_{i\in\mathcal{N}_{s\cup d}(t)} (r - \hat{r}) \nabla_\theta \log \pi_\theta(\mathbf{s}_i(t)).$$

Likewise, T3AM and RNSM are jointly-optimized. Node embeddings are randomly initialized and not treated as learnable, and are dynamically updated by Ada-DyGNN. The loss for T3AM with this process is defined as the cross-entropy

$$L_{ce} = -\log(\sigma_2(\mathbf{x}_s(t+)^\top \mathbf{x}_d(t+))) - \log(\sigma_2(1 - \mathbf{x}_s(t+)^\top \mathbf{x}_n(t+)))$$

where (x_d, x_n, t) is a randomly sampled negative edge.

3.3 Downstream Link-Prediction

Future link-prediction in Ada-DyGNN is of the node-based paradigm described in [12]. In particular, node representations are aggregated to pairwise representations used to predict edge insertions. This is in contrast to subgraph-based paradigms which extracts and learns a representations a local subgraph rather than on a node itself. Ada-DyGNN uses a simple cosine similarity measure for link-prediction—in particular, if $\cos(\mathbf{x}_i, \mathbf{x}_j)$ is large, there is a greater probability of a future link between them.

This is evaluated with the mean reciprocal rank (MRR) and area under the curve/average precision (AUC/AP) metrics. In MRR, for each test edge (v_s, v_d, t), we calculate its predicted rank (with rank 1 being the most probable) against all $|V| - 1$ negative samples (v_s, v_n, t) and average across the test set $MRR = \frac{1}{M}\sum_{i=1}^{M} \frac{1}{\text{rank}_i}$. In AUC/AP, we treat the task as a binary classification problem, generating one negative sample to compare against. MRR is a strictly harder metric since it compares against the entire set of possible negative samples.

4 Proposed Approach

To support an insert-delete streaming paradigm, we propose significant, but natural, changes to both the Ada-DyGNN model architecture and the downstream link-prediction task. For the former, we must incorporate another dimension of temporal information as an input into T3AM and RNSM. For the latter, we must generalize to support multiclass predictions (e.g., delete, add, keep) beyond simple cosine similarity of final embeddings. We discuss the many nuances of these modifications in the corresponding subsections. A summary table of all proposed changes is available in Appendix A.

4.1 T3AM Modifications

First, we must modify inputs to be of the form (v_s, v_d, t, f), where f is a flag that is 1 if the update is an edge insertion, and -1 if the update is an edge deletion. Edge insertions are only valid if the edge (v_s, v_d) exists, and vice versa for deletions.

This however, introduces an interesting quirk. With an insert-only paradigm, there are a finite number of possible updates (up to $\binom{|V|}{2}$) since once an edge is inserted, it cannot be deleted. On the other hand, for an insert-delete paradigm, edges can infinitely be inserted, then removed, then inserted again. The specific architecture we will describe below will only consider the most recent insert and deletion (if applicable). However, we will note below that our model can be extended to account for k-length history (i.e., up to k inserts and deletes each) by only a linear increase in embedding size.

Let $(v_s, v_d, t_i, 1)$ represent the most recent insertion for v_s, v_d, and $(v_s, v_d, t'_i, -1)$ represent the most recent deletion (if applicable). Note that if $t_i > t'_i$, then the edge (v_s, v_d) exists, but we still record that it was removed previously; and if $t_i < t'_i$ then there is no longer an edge, but still record the time it was previously added. We define $\Delta t_i = t - t_i$, $\Delta t'_i = t - t'_i$, and use the same time-decay function ϕ. If an insertion or deletion never occurs, we treat $t = \infty$, i.e. $\phi(\Delta t) = 0$.

Consequently, we make the following modifications to T3AM. The time-aware attentional coefficient between v_s, v_i is now defined as the softmax of the values

$$\sigma\left(\mathbf{a}^\top \left[\mathbf{W}_g\mathbf{x}_s(t) \,\|\, \phi(\Delta t_i)\mathbf{W}_g\mathbf{x}_i(t) \,\|\, \phi(\Delta t'_i)(-\mathbf{W}_g)\mathbf{x}_i(t)\right]\right)$$

where we rescale to have dimensions $\mathbf{W}_g \in \mathbb{R}^{d_m \times d_n}, \mathbf{x}_i(t) \in \mathbb{R}^{d_n}, \mathbf{a} \in \mathbb{R}^{3d_m}$. Note two key things: (1) while we share weights $\mathbf{W}_g$ and negate for deletion, this does not correspond to unlearning since we concatenate (rather than directly subtracting from the embedding) and scale by time-decay; and (2) this construction can easily be extended to concatenations of k-insertions and deletions where $\mathbf{a} \in \mathbb{R}^{(2k+1)d_m}$. Likewise, we modify messages as follows:

$$\mathbf{m}_s(t) = \sigma_1\left(\sum_{i \in \mathcal{N}_s(t)} \alpha_{si}\ [\phi(\Delta t_i)\,\mathbf{W}_g\mathbf{x}_i(t) \,\|\, \phi(\Delta t_i')\,(-\mathbf{W}_g)\mathbf{x}_i(t)]\right) \in \mathbb{R}^{2d_m}$$
$$\mathbf{m}(t) = \mathbf{m}_s(t) \,\|\, \mathbf{m}_d(t) \,\|\, \sigma_1(\mathbf{W}_e\mathbf{e}(t)) \in \mathbb{R}^{8d_m}$$

where we define $\mathbf{m}_d(t)$ analogously to $\mathbf{m}_s(t)$ and $\mathbf{W}_e \in \mathbb{R}^{4d_m \times d_e}$. The final attentional coefficient β_i and intermediate state $\mathbf{h}_i(t) = \beta_i \mathbf{W}_p \mathbf{m}(t)$ of node v_i are defined similarly to Ada-DyGNN (summing the two dot products for insert/delete) with no modifications, except $\mathbf{W}_p \in \mathbb{R}^{d_n \times 8d_m}$.

4.2 RNSM Modifications

Moving on to RNSM, we modify the state to include the flag f_i by concatenation, i.e., $\mathbf{s}_i(t) = \mathbf{h}_i(t) \,\|\, \mathbf{x}_i(t) \,\|\, f_i$. We keep the same two fully-connected layer policy $\pi(\mathbf{s}_i(t)) = \sigma_2(\mathbf{W}_1\sigma_1(\mathbf{W}_2\mathbf{s}_i(t)))$ but now $\mathbf{W}_2 \in \mathbb{R}^{d_n \times (2d_n+1)}$. We also use the same cosine reward scheme to incentivize the policy to retain local stability and the same policy update step. Note that cosine similarity still makes sense in the context of edge deletions, since the reward would optimize across two separate neighborhoods, i.e., $v_d \notin \mathcal{N}_s^*(t)$, $v_s \notin \mathcal{N}_d^*(t)$.

To optimize T3AM, we use cross-entropy loss, but we must adapt it for edge deletions. In particular, negative samples in deletion updates correspond to v_n such that there is an edge (v_d, v_n) at time t, and

$$L_{ce} = \begin{cases} -\log(\sigma_2(\mathbf{x}_s(t+)^\top \mathbf{x}_d(t+))) - \log(\sigma_2(1 - \mathbf{x}_s(t+)^\top \mathbf{x}_n(t+))) & \text{if } f = 1, \text{randomly sample } (v_s, v_n) \notin E, \\ -\log(1 - \sigma_2(\mathbf{x}_s(t+)^\top \mathbf{x}_d(t+))) - \log(\sigma_2(\mathbf{x}_s(t+)^\top \mathbf{x}_n(t+))) & \text{if } f = -1, \text{randomly sample } (v_s, v_n) \in E. \end{cases}$$

In particular, this intuitively pushes embeddings closer together when insertions occur, and farther apart when edges are deleted (where embeddings distance is proxied by dot product and transformed into probabilities with the sigmoid function).

4.3 Downstream Link-Prediction

Ada-DyGNN performs link-prediction using cosine similarity between node representations, treating this value as roughly the "probability" there will be an edge between these two nodes at time t. This is intrinsically a binary classification problem (insert/don't insert). However, with the introduction of edge deletion, there are now three distinct classes (insert/delete/no update). And while it's true that for any particular pair of nodes it is a binary decision (since multi-edges are prohibited), direct application of cosine similarity on embeddings is no longer fully robust/rich enough to characterize these differences. Furthermore, it is unclear whether the dot product (which is a simple unweighted sum of entrywise products) can learn nuanced temporal relationships embedded into the updated node representations.

As such, in Dada-DyGNN, we construct a simple MLP acting on the entrywise product $\mathbf{x}_i \odot \mathbf{x}_j$, as inspired by [13]. The value outputted by this MLP is then fed into tanh, which returns a number in the range $[-1, 1]$. We interpret -1 to mean full confidence that the edge will be deleted, 0 as perfectly indifferent, and 1 as full confidence that the edge will be inserted. The MLP is trained on the same Dada-DyGNN testset with cross-entropy loss, in particular, using the sigmoid function. This is a viable optimization since $\tanh(x) = 2\sigma(2x) - 1$ is simply a rescaling of the same objective. To summarize,

$$\text{MLP}(\mathbf{x}_i \odot \mathbf{x}_j) = \mathbf{W}_1^{\text{MLP}}\sigma_1(\mathbf{W}_2^{\text{MLP}}(\mathbf{x}_i \odot \mathbf{x}_j))$$
$$\text{Predict}(\mathbf{x}_i, \mathbf{x}_j) = \tanh(\text{MLP}(\mathbf{x}_i \odot \mathbf{x}_j)) \in [-1, 1]$$

where $\mathbf{W}_1^{\text{MLP}} \in \mathbb{R}^{1\times d_n}, \mathbf{W}_2^{\text{MLP}} \in \mathbb{R}^{d_n\times d_n}$ are learnable weights, and σ_1 is the ReLU non-linearity. Furthermore, we train the MLP with cross-entropy loss

$$L_{ce}^{\text{MLP}} = \begin{cases} -\log(\sigma_2(\text{MLP}(\mathbf{x}_i \odot \mathbf{x}_j))) - \log(\sigma_2(1 - \text{MLP}(\mathbf{x}_i \odot \mathbf{x}_k))) & \text{if } f = 1, \text{randomly sample } (v_i, v_k) \notin E, \\ -\log(1 - \sigma_2(\text{MLP}(\mathbf{x}_i \odot \mathbf{x}_j))) - \log(\sigma_2(\text{MLP}(\mathbf{x}_i \odot \mathbf{x}_k))) & \text{if } f = -1, \text{randomly sample } (v_i, v_k) \in E. \end{cases}$$

similar to the cross-entropy loss for T3AM but using the MLP instead of the dot product. We initialize $\mathbf{W}_1^{\text{MLP}} = \mathbf{1}_{d_n}^\top, \mathbf{W}_2^{\text{MLP}} = I_{d_n}$, which corresponds (roughly, ignoring non-linearities) to the dot product. We believe that this is a good initial parametrization since T3AM loss explicitly maximized the dot product of the embeddings of positive samples (i.e., linked nodes), and vice versa.

4.4 Implementation Details

Due to a (surprising) lack of publicly-available timestamped social network data with deletions, we benchmark Dada-DyGNN by testing future link prediction on two synthetic datasets.

First, we test on a deletion-augmented version of one the datasets used in [8]. In particular, we consider the UCI dataset [14] rather than Wikipedia and Reddit datasets for two reasons: (1) While the raw Wikipedia/Reddit data is given in [15], we could not find the exact 30 day excerpt that matches the number of nodes and edges for described in [8]; and (2) these datasets are far too large (on the order of 10 GB) to run locally given our compute setup. To augment the existing UCI dataset, we delete an inserted edge with probability p after $T \sim \text{Expo}(\lambda)$ from its first interaction. For our simulations, we chose $p = 0.2$ and $1/\lambda = 0.01 \times (T_{\max} - T_{\min})$, i.e., edges get deleted with 20% probability in approximately 1% of the entire timestamp range. This dataset has 1899 nodes with 71839 interactions out of which 12004 are deletions.

Second, we generate a test set based on a variation of the preferential attachment model [16] common in describing social networks/economics/biology. In particular, this variation assumes that at each time step, with probability π_1, a new node is connected to an existing node, chosen proportional to the degree of each existing node; with probability π_2, two existing nodes are chosen proportional to degree and connected; and with probability $\pi_3 = 1 - \pi_1 - \pi_2$, a random node is chosen proportional to degree and one of its edges deleted. Theoretical results show this converges to the power law distribution pervasive across real world phenomena. We choose $\pi_1 = \pi_3 = 0.05$, $\pi_2 = 0.9$, with 2471 nodes, 45588 interactions, and timestamps distributed as a Poisson process with $1/\lambda = 100$.

For each set, we consider only MRR (which is a strictly harder metric than AUC), with rankings determined by $\text{Predict}(\mathbf{x}_i, \mathbf{x}_j) \in [-1, 1]$ (reversed if sampling link-deletion prediction). We also use a similar hyperparameter setup to Ada-DyGNN. We use the Adam optimizer with 0.5 dropout rate, 0.0001 learning rate, and a maximum of 30 epochs. The last differs from Ada-DyGNN (which allows for up to 200 epochs) due to our compute limitations. Our lightweight MLP classifier is trained over 1000 epochs. We set $d_n = 64$ and $d_m = 16$ consistently throughout the experiment. This corresponds to messages $\mathbf{m}(t)$ and embeddings $\mathbf{x}_i$ of the same dimension as Ada-DyGNN for comparability. Furthermore, NSRM only samples the most $k = 200$ recent interaction neighbors. We study Dada-DyGNN in both transductive (predict edges between observed training nodes) and inductive (predict edges between unseen nodes) settings. In both, the first 80% of the edges are used as the training set, 10% are used as the validation set, and the remaining 10% as the testing set. Our experiments are run locally on a 8-core AMD Ryzen 7 5700U CPU with Radeon Graphics, taking 32660.34 seconds ($\sim$ 9 hours) for our deletion-augmented UCI dataset, and 14037.29 seconds ($\sim$ 4 hours) for our synthetic preferential attachment model.

Aside, we also patched numerous logical flaws in the given open-source implementation of Ada-DyGNN. These include: **(1)** a failure to properly concatenate $\mathbf{x}_i$ to calculate the state $\mathbf{s}_i$ (provided code instead duplicates $\mathbf{h}_i$); **(2)** incorrect sampling (while the original paper uses negative samples, in actuality, their sampling procedure returns any pair of nodes at random, which is only a decent heuristic if G is sparse); and (3) an inconsistent description of non-linearities (e.g., use of tanh rather than ReLU for $\mathbf{x}_i(t+)$).

5 Results

Full results are available under `log/DadaDyGNN` of our repository.

Below are the graphs showing MRR across training epochs (without a tuned MLP classifier). Unfortunately, our policies fail to converge within the 30 epoch maximum that we imposed, but even so, show decent results similar to traditional approaches, e.g., GCN/GraghSAGE/GAT/CTDNE. Dada-DyGNN achieves a MRR of around 0.007 for transductive (seen nodes) and 0.004 (new unseen nodes) for the UCI dataset, and around 0.004/0.003, respectively, for our preferential attachment dataset. We expect that with policy convergence, we would achieve far better performance, lower bounded by the maximum MRR achieved in each run (approximately 0.008/0.007 for UCI and 0.005/0.005 for preferential attachment). Besides a lack of convergence, we also attribute the comparatively poor performance to the SoTA/Ada-DyGNN to the inherent difficulty of simultaneously predicting link deletions and insertions. Predicting insertions alone degenerates to a simple binary classification with cosine similarity, which is no longer the case for our task at hand. Furthermore, it is possible that constraining messages to the same 64 dimension representation (while trying to incorporate double the information with deletion history) greatly reduces the expressivity of our GNN.

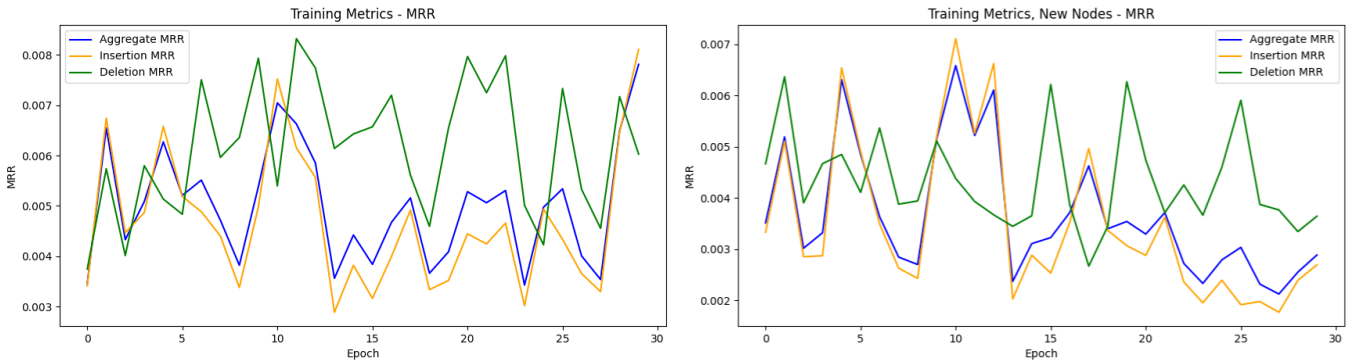

Figure 2: MRR for insertions/deletions/aggregate over training epochs for both transductive (left) and inductive (right) link-prediction tasks on the augmented UCI dataset.

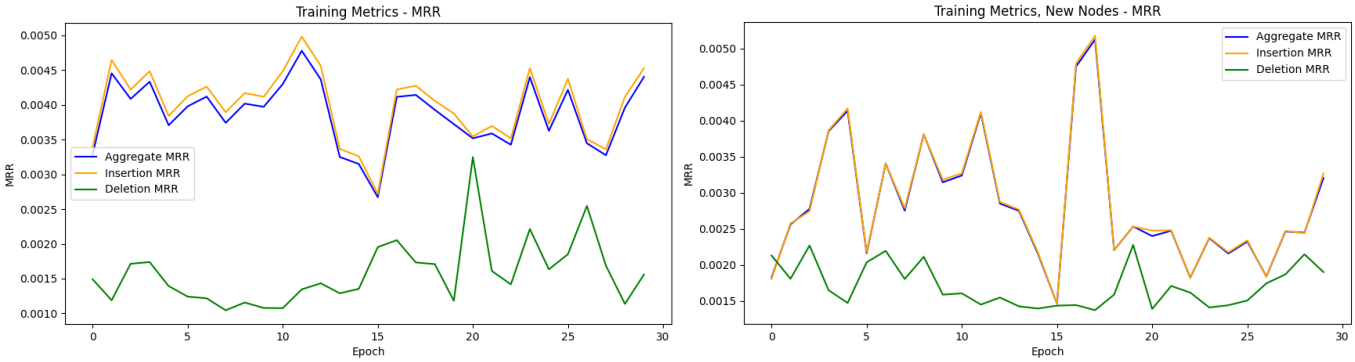

Figure 3: MRR for insertions/deletions/aggregate over training epochs for both transductive (left) and inductive (right) link-prediction tasks on the preferential attachment dataset.

When we look at the breakdown of the aggregate MRR into insertion and deletion tasks, we see that as expected, with a larger proportion of deletion updates, relative performance increases for deletion MRR compared to insertion MRR. This is most clearly seen in the preferential attachment model, where deletions only account for $\sim 5\%$ of interactions and such only achieve a MRR of roughly 0.002. Perhaps more surprisingly, with deletions accounting for $\sim 20\%$ updates in the UCI dataset, performance on link-deletion prediction is, on average, better than link-insertion prediction. This shows promise that link-deletion can be incorporated as a downstream task for dynamic GNNs.

Furthermore, while we anticipated that the final MLP classifier would improve performance, this does not seem to be the case from our experiments. In particular, after the MLP, we achieve MRRs of 0.0033/0.0040 for UCI and 0.0031/0.0027 for preferential attachment—lower than the raw MRRs. We hypothesize that this is due to overfitting over the 1000 training epochs (with classifier loss fairly stable in the last few hundred epochs), and anticipate that a shorter train cycle would achieve positive impact.

Finally, we acknowledge a painful limitation in our experimental results—the lack of a baseline comparison with existing methods. This is a difficult task, since as discussed, previous approaches only tackled link-insertion predictions, and thus it is unclear how to impartially benchmark against our deletion-augmented datasets. A naïve approach would be to set all predictions for deletions for these insertion-only methods as incorrect (i.e., 0 MRR). This, however, artificially and unfairly penalizes what may actually be a useful embedding. A slightly more refined approach would perhaps reverse the ranks before calculating MRR for link-deletion prediction, but this does not fundamentally address the issue that these methods are not designed for this downstream task. In particular, low similarity in these methods correspond to a "low chance of an edge-insertion", which is substantively different from a "high chance of edge-deletion"—a classic example of this is in our preferential attachment model, where more active nodes are both more likely to have insertions *and* deletions.

6 Discussions and Conclusion

Our work with Dada-DyGNN shows promise in constructing a nuanced, delete-augmented streaming GNN algorithm. In particular, our paper provides a new perspective on encoding edge deletion history—beyond (partial) unlearning—through time-decay with concatenation. These contributions are in two main areas: **(1)** the modified model architecture (T3AM/RNSM) and objective functions; and **(2)** open-source code improving upon Ada-DyGNN's logical bugs as well as support for edge deletions.

Clear next steps to build out Dada-DyGNN include more ablations discussed briefly in Section 5. In particular, training for more epochs to achieve true policy convergence; decreasing MLP training to prevent overfitting; constructing a comprehensive benchmark with existing methods; plotting performance across message/embedding dimensions (rather than constraining to the same dimensions in Ada-DyGNN); increasing history length (beyond the latest deletion/insertion); and testing on non-augmented real-world data. These experiments would demonstrate the robustness of our Dada-DyGNN architecture.

Beyond this, it would be worthwhile to explore how to extend more general classes of GNNs to incorporate temporal information through our design. In particular, how well does concatenation with time-decay and a look-back history work for general graph learning tasks? Can the loss function be improved upon where instead of cross-entropy, perhaps we use contrastive learning? There is also a question of whether time-decay should be explicitly predetermined (e.g., in our case, $O(1/\Delta t)$) or whether implicitly learning temporal decay would significantly improve the embeddings learned by a dynamic GNN. Overall, dynamic graph machine learning has been a drastically understudied area, and we believe that there is much progress to be made in designing architectures that succinctly capture rich structural, intrinsic, and temporal information.

References

[1] Yao Ma, Ziyi Guo, Zhaochun Ren, Eric Zhao, Jiliang Tang, and Dawei Yin. 1997. Streaming Graph Neural Networks. In *Proceedings of ACM Woodstock conference (WOODSTOCK'97)*, Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 11 pages.https://doi.org/10.475/123_4

[2] Zhe Wang, Tianjian Zhao, Zhen Zhang, Jiawei Chen, Sheng Zhou, Yan Feng, Chun Chen, and Can Wang. 2024. Towards Dynamic Graph Neural Networks with Provably High-Order Expressive Power. Retrieved from https://arxiv.org/abs/2410.01367

[3] Giang Hoang Nguyen, John Boaz Lee, Ryan A. Rossi, Nesreen K. Ahmed, Eunyee Koh, and Sungchul Kim. 2018. Continuous-Time Dynamic Network Embeddings. In *WWW '18 Companion: The 2018 Web Conference Companion*, April 23–27, 2018, Lyon, France. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3184558.3191526

[4] Palash Goyal, Nitin Kamra, Xinran He, and Yan Liu. 2018. DynGEM: Deep Embedding Method for Dynamic Graphs. Retrieved from https://arxiv.org/abs/1805.11273

[5] Srijan Kumar, Xikun Zhang, and Jure Leskovec. 2019. Predicting Dynamic Embedding Trajectory in Temporal Interaction Networks. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)*, Association for Computing Machinery, Anchorage, AK, USA, 1269–1278. DOI:https://doi.org/10.1145/3292500.3330895

[6] Da Xu, Chuanwei Ruan, Evren Korpeoglu, Sushant Kumar, and Kannan Achan. 2020. Inductive Representation Learning on Temporal Graphs. Retrieved from https://arxiv.org/abs/2002.07962

[7] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. Retrieved from https://arxiv.org/abs/2006.10637

[8] Hanjie Li, Changsheng Li, Kaituo Feng, Ye Yuan, Guoren Wang, and Hongyuan Zha. 2024. Robust Knowledge Adaptation for Dynamic Graph Neural Networks. Retrieved from https://arxiv.org/abs/2207.10839

[9] Andrew McGregor. 2014. Graph stream algorithms: a survey. SIGMOD Rec. 43, 1 (May 2014), 9–20. DOI:https://doi.org/10.1145/2627692.2627694

[10] Jiali Cheng, George Dasoulas, Huan He, Chirag Agarwal, and Marinka Zitnik. 2023. GNNDelete: A General Strategy for Unlearning in Graph Neural Networks. Retrieved from https://arxiv.org/abs/2302.13406

[11] Yanping Zheng, Zhewei Wei, and Jiajun Liu. 2023. Decoupled Graph Neural Networks for Large Dynamic Graphs. Retrieved from https://arxiv.org/abs/2305.08273

[12] Muhan Zhang. 2022. Graph Neural Networks: Link Prediction. In Graph Neural Networks: Foundations, Frontiers, and Applications, Lingfei Wu, Peng Cui, Jian Pei and Liang Zhao (eds.). Springer Singapore, Singapore, 195–223.

[13] https://medium.com/stanford-cs224w/online-link-prediction-with-graph-neural-networks-46c1054f2aa4

[14] J. Kunegis, "Konect: the koblenz network collection," in *The Web Conference*, 2013, pp. 1343–1350.

[15] https://snap.stanford.edu/data/index.html

[16] Maria Deijfen and Mathias Lindholm. 2009. Growing networks with preferential deletion and addition of edges. Physica A: Statistical Mechanics and its Applications 388, 19 (October 2009), 4297–4303. DOI:https://doi.org/10.1016/j.physa.2009.06.032

Appendix

A. Dada-DyGNN Modifications

Component	Symbol	Changes in Dada-DyGNN	Dimensions
Input	(v_s, v_d, t, f)	Add flag $f \in \{+1, -1\}$ corresponding to insert/delete	N/A
Time-Aware Attentional Coefficient	α_{si}	Concatenate edge deletion history $\phi(\Delta t_i')(-\mathbf{W}_g)\mathbf{x}_i(t)$ before softmax using shared weights $\mathbf{W}_g$	$\mathbf{W}_g \in \mathbb{R}^{d_m \times d_n}$ $\mathbf{a} \in \mathbb{R}^{3d_m}$
(Node) Messages	$\mathbf{m}_s(t), \mathbf{m}_d(t), \mathbf{m}(t)$	Concatenate edge deletion history $\phi(\Delta t_i')(-\mathbf{W}_g)\mathbf{x}_i(t)$	$\mathbf{m}_s(t), \mathbf{m}_d(t) \in \mathbb{R}^{2d_m}$ $\mathbf{W}_e \in \mathbb{R}^{4d_m \times d_e}$
(Intermediate) States	$\mathbf{h}_i(t), \mathbf{s}_i(t)$	No modification beyond weight matrix; Concatenate flag f_i	$\mathbf{W}_p \in \mathbb{R}^{d_n \times 8d_m}$ $\mathbf{s}_i(t) \in \mathbb{R}^{2d_n+1}$ $\mathbf{W}_2 \in \mathbb{R}^{d_n \times (2d_n+1)}$
T3AM Loss Function	L_{ce}	Different cases based on insert/delete; update negative sampling	N/A
Link-Prediction	N/A	Train MLP on $\mathbf{x}_i \odot \mathbf{x}_j$ with cross-entropy loss (σ_2); output "probability" of insert (+1)/delete (-1) as tanh	$\mathbf{W}_1^{\text{MLP}} \in \mathbb{R}^{1 \times d_n}$ $\mathbf{W}_2^{\text{MLP}} \in \mathbb{R}^{d_n \times d_n}$

Figure 4: This figure summarizes the modifications in Dada-DyGNN.

B. Generative Tools Disclosure

In this project, we leveraged two generative AI tools—Windsurf (formerly Codeium/Qodo) and OpenAI ChatGPT. The former is a coding-specific tool, and it was prompted to write certain snippets of our project; the latter was used mainly to explain the existing codebase. The experience in using these tools was mixed. While Windsurf consistently provided a very good skeleton to work off of, it was mostly unable to implement the complex algorithmic logic precisely as laid out in our paper. ChatGPT had similar issues, though was more helpful in giving broader verbal explanations of technical material. As such, an overwhelming majority of the logic in Dada-DyGNN was implemented manually, with the most significant changes in `DadaDyGNN.py` (model architecture) and `utils.py` (rewriting a more robust sampler and `NeighborFinder` class).