Reducing Resource Usage for Continuous Model Updating and Predictive Query Answering in Graph Streams

Qu Liu
Department of Computer Science
University of Massachusetts, Lowell
Massachusetts, USA
qliu@cs.uml.edu

Adam King
Department of Computer Science
University of Massachusetts, Lowell
Massachusetts, USA
kingamk3@gmail.com

Tingjian Ge
Department of Computer Science
University of Massachusetts, Lowell
Massachusetts, USA
ge@cs.uml.edu

***Abstract*—We observe the need for continuous, online training of dynamic graph neural network (DGNN) models while at the same time using them to answer continuous predictive queries as data streams in. This implies significant training-time and memory costs. Along with the DGNN model learning, we simultaneously learn a weight/priority distribution over the nodes via a randomized online algorithm. In turn, the DGNN is continuously trained/learned by sampling nodes from the learned distribution and performing the chosen nodes' partitions of training work. We also devise a novel graph Kernel Density Estimation technique to smooth the distribution and improve the learning quality. Our experiments show that continuous online learning is much needed for graph streams and our approach significantly improves the standard DGNN models—to achieve the same accuracy, the training time ranges from several times to two orders of magnitude shorter, and the maximum memory consumption is several times to 20 times smaller.**

***Index Terms*—online learning, continuous model updates, continuous predictive queries, dynamic graph neural networks, kernel density estimation**

I. Introduction

Graphs are a general model to represent data/information, especially when they are heterogeneous with various types, as data can be represented by entities (nodes) and relations (edges). The *relations* here may also broadly refer to interactions between entities. For this reason, graph representation learning has been studied in a wide range of applications, including e-commerce, recommender systems, social networks, biology, healthcare, communication networks, computational finance, and traffic. Many of these graphs are inherently dynamic and are modeled as graph streams, which contain streaming updates. Let us look at an example.

Example 1. *Medical data for patients in the Intensive Care Units (ICU) of hospitals is highly dynamic and rich, and is a heterogeneous graph stream. Specifically, we look into the MIMIC-III (Medical Information Mart for Intensive Care III) data [1] comprising de-identified health-related data associated with over forty thousand patients who stayed in the critical care units of the Beth Israel Deaconess Medical Center between 2001 and 2012. Figure 1 illustrates a snippet of the data from this dataset. First of all, there are many types of entities, shown as vertices in the graph, including patients, diagnosis, procedures, lab events, input events, and prescription drugs. They are shown as nodes of different colors.*

Interaction relations such as lab measurement events and input events (fluids administered to a patient) are very dynamic and timestamped (dashed edges in Figure 1). Some relations can be static, such as a patient's diagnosis (solid edges in the figure). Two patients may have relations with the same diagnosis, or they may have the same input events but at different times. Learning from such heterogeneous data sources is instrumental for inference and reasoning. For instance, a hospital manager may subscribe to the following continuous predictive query so that sufficient resources can be allocated: "Notify me when it is predicted that, in the next hour, grouped by the medical procedure, the number of patients tested with abnormal results is above a certain threshold".

This work is supported by NSF grants IIS-2124704 and OAC-2106740.

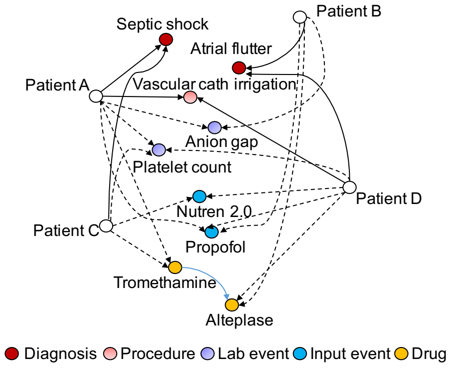


Fig. 1: Illustrating a graph stream in healthcare.

Graph streams [2] belong to the more general notion of *data streams* [3], where a major type of queries widely used in practice is *continuous queries* that monitor the dynamic data. For high-rate graph streams, data is often partially observed and incomplete, continuously predicting missing events and future events can be very useful, as shown in Example 1 where a hospital manager may subscribe to a number of predictive queries including the number of abnormal lab results exceeding a threshold. In general, it has been shown that monitoring such predictive events and predictive queries is very valuable in healthcare, business operations, network monitoring, among other domains [4], [5].

Graph neural networks have become the state-of-the-art deep learning model for data represented as graphs [6]–[8]. In recent years, there has been a number of proposals of *dynamic graph neural networks* (DGNN) [9], [10] that become the dominant approach for dynamic networks and graph streams, where node attributes and edges dynamically change over time, as is the case in the application of Example 1.

Interestingly, not only does such a system need to continuously answer a set of predictive queries, we find that (as shown later) it also needs to continuously perform graph machine learning model updates, analogous to what other researchers found for online machine learning in general [11]–[13]. Thus, the central questions that we try to answer in this paper are: *Do we need to continuously train a DGNN model as we continuously answer predictive queries? If so, since continuous learning can be resource demanding, how can we perform it in a time- and memory-efficient manner?*

With empirical evidence (Section VI), we find the answer to the first question to be affirmative. Then computation and memory efficiency of online (as data changes) and continuous training is much needed, given the real-time requirement nature of the continuous monitoring tasks. We reduce the resource usage of continuous model updates by the observation that the size of a DGNN model is independent of the size of $\mathcal{G}$. Thus, different parts of $\mathcal{G}$ may not have equal importance to online learning—***chosen subgraphs might be more effective for training given the same amount of compute resources***. In addition, which part of data to focus on in training also has to do with the continuous queries depending on what data is relevant to predicting their results. Therefore, we devise a novel method to learn a weight distribution over the nodes in $\mathcal{G}$ and perform adaptive online training over selected subgraphs. Our method is both ***data aware*** and ***workload aware***.

A. Our Contributions

Although there has been work on online machine learning [11]–[13], to our knowledge, we are the first to answer this question in the context of dynamic networks and graph streams, where the DGNN training work can be partitioned over nodes in the network and it is natural to adaptively learn "what to learn", i.e., which nodes are more profitable to train at a given time.

Our experimental study (Figure 4) clearly demonstrates the need for online continuous learning in graph streams. Towards our goal, we first propose a scheme for the *node-level partitioning* of incremental training work, which includes both *self-supervised learning* [6] using node/edge labels as targets and *supervised learning* based on the revealed results of the continuous predictive queries discussed above. Thus each node has its partition of training work.

Then we devise a randomized algorithm that simultaneously both **(a)** learns a weight distribution over the nodes and **(b)** does weighted online training of DGNN (based on any existing DGNN model)—by sampling over the nodes using the very weight distribution learned in (a), and then performing the sampled node's partition of training work. We prove that the running state of the continuous randomized algorithm, which is a weight distribution of nodes, follows a Markov chain that converges to its stationary distribution where the probability of a state s (i.e., weight distribution) is proportional to e^{u_s}, where u_s can be flexibly defined as the *utility* (e.g., accuracy in evaluation) of using the weight distribution s to sample and train. In other words, our continuous learning algorithm will converge to a state with high utility (corresponding to high query result accuracy).

Since message passing of graph neural networks is between each node and its neighborhood, the utilities of training at two neighboring nodes should not be too far apart. As a result, the weight distribution should be more or less smooth with respect to edge connections in the network. Inspired by the Kernel Density Estimation (KDE) technique [14], [15] originally for estimating continuous distributions, we propose a novel *graph KDE* based on *random-walk kernels* which is for discrete distributions and is aware of network topology. We analyze the weight distribution learning enhanced by the graph KDE, and show that the effective sampling distribution is a smoothed version of the original one based on a node's network neighborhood. Our experiments show that the graph-KDE enhanced version further makes continuous learning more efficient.

Our comprehensive empirical study using three real-world graph stream datasets and six baseline models demonstrates the need for online adaptive learning for graph streams and shows the salient advantages of our proposed weighted online learning enhanced by graph KDE over standard DGNN models in previous work. To achieve the same prediction accuracy, the training time of our proposed method is typically one to two orders of magnitude shorter, and the maximum memory consumption is several times to 20 times smaller.

In summary, our contributions include:

- We propose the problem of simultaneous resource-efficient model updates and continuous query answering in graph streams.
- We carefully study and empirically verify the need for continuous learning-model updates in graph streams.
- We devise a randomized algorithm with theoretical guarantees that adaptively learns a node distribution and performs *weighted* learning that reduces compute-resource usage.
- We further propose a novel *graph kernel density estimation* method that smooths the learned distribution and further enhances the resource efficiency.
- We perform a comprehensive experimental study that evaluates the motivation, the effectiveness of our approaches, and the compute-resource reduction.

II. Problem Statement

A *graph stream* $\mathcal{G} = (N, E)$ is an infinite sequence of *snapshots* $\mathcal{G}_1, \mathcal{G}_2, ..., \mathcal{G}_t, ...$, where each snapshot $\mathcal{G}_t$ corresponds to a *time step* t, N is the set of nodes, and E is the set of edges. Each snapshot $\mathcal{G}_t = (N_t, E_t)$ satisfies $N_t \subseteq N$ and $E_t \subseteq E$. Each node $v \in N$ may have a set of attributes $X = X_1, ..., X_a$ that may bear different values in different snapshots. In addition, each edge $e \in E$ may be of one of the r types. So this is a heterogeneous graph stream.

Note that there are other formulations of graph streams [2] such as a sequence of graph edges, and the formulation above that we use is a more general model that also incorporates attribute updates and relations/edges with certain durations (including static relations). For instance, in Example 1 and Figure 1, there are also more static edges such as those between patients and diagnosis nodes (e.g., Septic shock) and between patients and procedures (e.g., Vascular cath irrigation)—indicating the "properties" of ICU patients without a timestamp.

$\mathcal{G}$ has a *analytics workload* which contains a number of *continuous predictive/monitoring queries*, each of which is to *continuously* predict, at every time t, a function of the data in $\mathcal{G}_{t+\delta}$, where $\delta > 0$. That is, each query keeps predicting, at every step t, a value that is a function of the data in a future snapshot. This is shown on the right hand side of Figure 2. As an example, recall one such query near the end of Example 1. For such an analytics workload, there is a body of previous models called *dynamic graph neural networks* (DGNN) [9], [10] that can be used to answer the continuous queries. The problem we are focusing on in this paper is:

- Does the system indeed need to continuously update the dynamic graph neural network model in order to continuously get up-to-date answers to the queries?
- If so, find a strategy to continuously update an existing DGNN model so as to minimize the computational resource consumption for a fixed prediction error budget.

III. System Architecture and Node-level Partitioning of Learning

A. Architecture and Model

The system architecture is illustrated in Figure 2. Our work can adopt any of the dynamic graph neural network (DGNN) model in the literature [9], [10], which typically consists of a conventional graph neural network (GNN) model to capture message-passing at the graph-structure level and a sequence model for temporal correlations. In our evaluation in Section VI, we have experimented with seven such models, namely TGCN [16], DCRNN [17], GCLSTM [18], DyGrEncoder [19], ROLAND [20], WinGNN [21], and EvolveGCN [22].

The model operates on a graph stream $\mathcal{G}$, and dynamically learns the weights/importance of each node of the graph as a distribution $\mathcal{D}[v]$. Our ultimate approach is based on the graph-KDE that we design, as illustrated in Figure 2 and detailed in Section V. The main purpose is to handle

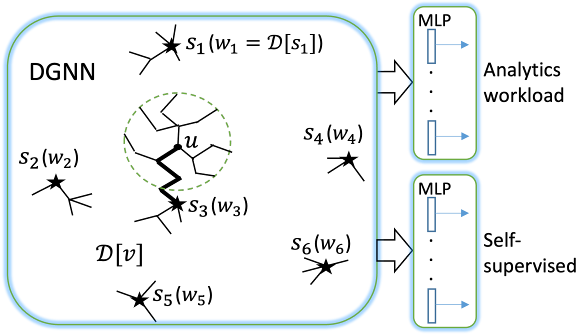


Fig. 2: The overall architecture of our model. The left-hand side is any dynamic graph neural network (DGNN) model, and the right-hand side shows two parts of training—supervised from continuous analytics workload and self-supervised from graph data (node/link labels). The online training work is partitioned at the node level, and we learn the weights/importance of each node of the graph (distribution $\mathcal{D}[v]$). Our ultimate approach is graph-KDE sampling (details in Section V) that selects a number of seed nodes (s_1 to s_6) based on their weights w_i, and performs a short random walk from a chosen seed to arrive at a node u for incremental training at a subgraph induced by the neighborhood of u.

the analytics workload, which contains a set of continuous predictive queries, for each of which we further use a multi-layer perceptron (MLP) model on top of the embeddings from DGNN to predict the results. The supervised and self-supervised learning part will be explained in Section III-B and Section III-C.

B. Two-Part Model Training

As commonly done for graph neural networks, the training consists of two parts:

1) *Supervised* learning based on the q continuous predictive queries in the analytics workload. The system knows the ground-truth result labels when the predicted time in a query later arrives—so the supervised learning has a slight delay to get the future result labels.
2) *Self-supervised* learning by predicting chosen nodes/links (labels) in the network [23], i.e., supervision signals within the graph, rather than external labels from the analytics workload.

This is illustrated in Figure 2. The self-supervised learning part is especially helpful when the external labels from the analytics workload are insufficient.

C. Node-level Partitioning of Training Work

Intuitions and Rationales. A key novelty in our work is to reduce the computational resource usage of continuous model updates by the observation that the size (i.e., number of parameters) of a dynamic graph neural network model is independent of the size of $\mathcal{G}$. Therefore, depending on the nature of the data (nodes, attributes, relations) and data updates, we may not need to spend the resources to uniformly train over all parts of $\mathcal{G}$, i.e., **some chosen subgraphs may be more "profitable" when investing the same amount of compute resources. So our selective training is *data aware***. Moreover, it is clear that which parts of the dynamic graph $\mathcal{G}$ have higher importance for training also depends on the continuous predictive queries—what subset of data is relevant to the queries. Hence **the selective training should also be *workload aware***.

We now discuss how to partition the online incremental training work for each node. Once we learn the weight distribution $\mathcal{D}$ over all nodes in the graph stream (Sections IV and V), we sample nodes from $\mathcal{D}$ and perform each chosen node's partition of work in their neighborhood subgraphs. This process is repeated, incremental, and adaptive to data changes. This is illustrated on the left of Figure 2. For a node v, its partition of incremental training work is as follows.

1) The self-supervised learning part for node v is to choose node v or any of its incident edges with node/link labels as training targets.
2) Node v's partition of supervised learning is to train over the continuous queries based on a smaller, induced subgraph $\mathcal{G}_v$ being fed into the DGNN model for back-propagation. Specifically, $\mathcal{G}_v$ is the induced subgraph of $\mathcal{G}$ by keeping only v and all its neighbors within L hop distance, where L is the number of *layers* associated with the underlying GNN model (usually no more than 3). Note that the inference (forward propagation) is still based on the whole graph $\mathcal{G}$.

The purpose of node-level online learning partition is to help us gauge the influence of incrementally selecting a node partition for training (at the current moment) to the accuracy/utility of the analytics workload.

IV. Adaptive Node-Weight Learning

We devise a randomized online algorithm to learn the node-weight distribution $\mathcal{D}$ adaptively from the data and the continuous analytics workload. The very first idea is that we consider online learning as *continuous discrete steps*, where we sample nodes based on a distribution $\mathcal{D}$ (that we are about to learn *at the same time!*), and for each chosen node, we perform its node-level partition of the training as described above. Our algorithm will sample one or more *pairs* of nodes in each training step, and perform their training partitions. There is a fixed time interval between two training steps.

A. Temporal Utility

1) Utility of a Training Action: We measure the effectiveness of an incremental training action (e.g., performing a node's training partition, as described in Section III-C) using a *utility* value, where a higher utility should lead to better accuracy when evaluating the analytics workload (all continuous queries). However, we cannot afford to evaluate the analytics workload every time we need to get the utility of training a node's partition.

Instead, inspired by previous work on focal loss [24] and on confidence-aware learning [25], the *hardness* of a node sample, as measured by the training loss/error before back-propagation, is indicative of how beneficial it is to use this node to train. This also has no extra overhead. Thus, we choose the utility function as such in our experiments. Our framework is flexible and allows other definitions of utilities. In all, a utility value is essentially a score with a higher value indicating a more desirable state in online learning.

2) Temporal Utility: We give the following definition.

Definition IV.1. The *temporal utility at a node* v at a step during continuous learning is defined as a function $u : V \to R$. It is a function of the next node v to be selected for training, and the returned utility $u(v)$ is measured right after the node v's training partition is performed.

Note that the temporal utility at a node is conditioned on the past training (which also follows $\mathcal{D}$). As in our algorithm (to be presented in Section IV-B), we choose the next node randomly based on a weight distribution $\mathcal{D}$, the temporal utility U at a time step is a random variable, and it has an *expected temporal utility* over the distribution $\mathcal{D}$.

Intuitively, if the temporal utility at a node v is high, we should probably increase v's weight to gain more on the overall expected utility. However, this does not go on forever (hence the term "temporal" utility). Recall that the temporal utility is conditioned on the previous training profile (also determined by $\mathcal{D}$). If v were trained too frequently already, additional selection of v would not be as beneficial to the utility as selecting a different node. In other words, the temporal utility at node v will likely decrease after some time (when it is trained sufficiently often), while the temporal utility at other nodes will increase.

B. The Algorithm

Our algorithm for learning the node weight distribution $\mathcal{D}$ will be based on the notion of temporal utility discussed in Section IV-A. As we aim for online continuous training and query answering, we devise a simple, efficient randomized algorithm that has little overhead, but that has a solid theoretical base. It is presented in Algorithm 1.

The main idea of the algorithm is as follows. Like a chip-firing game on graphs [26], our algorithm places *chips* at each node where the number of chips at a node is proportional to its weight. Let $\mathcal{D}[v]$ be the number of chips at node v. For a pair of nodes sampled based on $\mathcal{D}$, we perform each node's partition of incremental training (Section III-C). We compare the utilities from training each of the two nodes, and move one chip from the "loser" node (with lower utility) to the "winner" node with a higher probability than moving a chip in the opposite direction. The intuition is that, if training at a node is more beneficial now, we more likely increment its weight by a small amount.

Line 1 initially places k chips at each node (empirically, a small number, e.g., 5, gives good results). Line 2 iterates over every pair of nodes (to be sampled from $\mathcal{G}$ below in line 4) at each training step. In GetSampleNode, $\mathcal{U}$ is the set of nodes that have (or whose incident edges have) new data updates since the previous training step. $\mathcal{D}|\mathcal{U}$ in line 19 is the distribution (chips) from $\mathcal{D}$ when the node choices are limited to set $\mathcal{U}$. Lines 18-21 basically give more probability to nodes with new updates (e.g., we empirically set $p_u = 0.5$). $\mathcal{U}$ gets updated between two training steps (as part of the input graph stream $\mathcal{G}$) and reflects the current data changes.

Lines 5-6 perform v_i's partition of training work using data in a window up to the current time step t. The utility in line 7 is as described in Section IV-A. Finally, lines 8-16 handle chip moving between the "winner" node (index w) and "loser" node (index l). Lines 12 and 15 ensure that every node at least has 1 chip so it still has some chance to be selected.

Complexity. If the frequency of online learning is f steps/sec, then the time cost of Algorithm 1 is $O(fd^L)$ per second, where d is the average degree of sampled nodes and L is the number of layers of DGNN. The space complexity is $O(d^L)$. This is because the incremental training per step (lines 5-6) is performed over the chosen node v_i's partition—the induced subgraph L hops around v_i (illustrated as the subgraph inside the dashed circle in Figure 2).

Algorithm 1: ONLINEADAPTIVELEARNING

```
Input: $\mathcal{G}$: the graph stream
       $\mathcal{Q}$: continuous analytics workload
1  $\mathcal{D}[v] \leftarrow k$ for all $n$ nodes in $\mathcal{G}$
2  for each pair at each training step $t$ do
3      for $i \leftarrow 1, 2$ do
4          $v_i \leftarrow$ GETSAMPLENODE ($\mathcal{D}$,$\mathcal{U}$)
5          do self-supervised training at $v_i$
6          do supervised training on $\mathcal{Q}$ for $v_i$'s partition
7          $u_i \leftarrow$ utility measured for the training above
8      $w \leftarrow 2$, $l \leftarrow 1$ // winner/loser index
9      if $u_1 > u_2$ then
10         $w \leftarrow 1$, $l \leftarrow 2$
11     with probability $1/2$
12         if $\mathcal{D}[v_l] > 1$ then // minimum 1 chip
13             $\mathcal{D}[v_w]++$; $\mathcal{D}[v_l]--$
14     else with probability $e^{-\frac{u_w - u_l}{kn}}$
15         if $\mathcal{D}[v_w] > 1$ then // minimum 1 chip
16             $\mathcal{D}[v_l]++$; $\mathcal{D}[v_w]--$
17 Function GETSAMPLENODE ($\mathcal{D}$,$\mathcal{U}$)
18     with probability $p_u$
19         $v \leftarrow$ randomly chosen from $\mathcal{D}|\mathcal{U}$
20     else
21         $v \leftarrow$ randomly chosen from $\mathcal{D}$
22     return $v$
```

C. The Analysis

Now we perform a novel analysis of Algorithm 1. We first draw a connection between our algorithm and the *influence functions* in *robust statistics* [27]. Then we prove the correctness/optimality of the algorithm in terms of the learned distribution.

1) Preliminaries on Influence Functions and Markov Chains: An influence function tells us the effect of a change in one observation on an estimator of a statistic.

Definition IV.2. [27] Given some distribution F, the *influence function* of statistic $\hat{\theta}$ at F is the *Gâteaux derivative* of $\hat{\theta}$ at F in the direction δ_x:

$$\mathrm{IF}_{\hat{\theta},F}(x) = \lim_{\epsilon \to 0} \frac{\hat{\theta}((1-\epsilon)F + \epsilon\delta_x) - \hat{\theta}(F)}{\epsilon}$$

where δ_x is a probability distribution with all its mass at point x.

The influence function thus quantifies how statistic $\hat{\theta}(F)$ changes if distribution F is contaminated by a small amount of data mass at point x. Stated differently, it quantifies the influence of data point x on $\hat{\theta}$.

In our analysis of Algorithm 1 below, the estimated statistic $\hat{\theta}$ is the expected temporal utility of $\mathcal{D}$, while point x is a sampled node where chips may be moved in or out.

A *Markov chain* $\mathcal{M}$ is a discrete time stochastic process defined over a set of states S in terms of a matrix P of transition probabilities [28]. Let the number of states be n. Then a *stationary distribution* of the Markov chain is a probability distribution (over the states) $\pi = (\pi[1], \ldots, \pi[n])$ such that $\pi = \pi P$. Intuitively, if the Markov chain is in the stationary distribution at step t, it remains so at step $t+1$. A *finite*, *irreducible* (i.e., there is a non-zero probability to go between any two states), and *aperiodic* (i.e., the length of path from a state to itself does not have to be divisible by a period $\Delta > 1$) Markov chain must have a unique stationary distribution.

2) Analysis with Influence Functions: We first analyze the connection between the influence function and our algorithm—the influence of a chosen node on the expected temporal utility of $\mathcal{D}$. We also show that the update of node weight distribution $\mathcal{D}$ through moving chips is guided by the influence function values at chosen nodes. We prove the following.

Theorem IV.3. *Let the node-weight distribution be $\mathcal{D}$, and let the expected temporal utility of $\mathcal{D}$ be $\hat{u}(\mathcal{D})$. Then the influence function of $\hat{u}$ at $\mathcal{D}$ is $\mathrm{IF}_{\hat{u},\mathcal{D}}(x) = u(x) - \hat{u}(\mathcal{D})$, where $u(x)$ is the temporal utility of node x. Moreover, for a pair of nodes v_1 and v_2 chosen in lines 3-4 of Algorithm 1, the ratio of the probabilities of moving one chip in the direction $v_1 \to v_2$ vs. $v_2 \to v_1$ (lines 13 and 16) is an exponential function of $\mathrm{IF}_{\hat{u},\mathcal{D}}(v_2) - \mathrm{IF}_{\hat{u},\mathcal{D}}(v_1)$.*

Proof. Based on Definition IV.2, we have

$$\begin{aligned}\mathrm{IF}_{\hat{u},\mathcal{D}}(x) &= \lim_{\epsilon \to 0} \frac{\hat{u}((1-\epsilon)\mathcal{D} + \epsilon\delta_x) - \hat{u}(\mathcal{D})}{\epsilon} \\ &= \lim_{\epsilon \to 0} \frac{(1-\epsilon)\hat{u}(\mathcal{D}) + \epsilon \cdot u(x) - \hat{u}(\mathcal{D})}{\epsilon} \\ &= u(x) - \hat{u}(\mathcal{D})\end{aligned}$$

From the above equation, for two nodes v_1 and v_2 chosen in lines 3-4 of Algorithm 1, we must have $\mathrm{IF}_{\hat{u},\mathcal{D}}(v_2) - \mathrm{IF}_{\hat{u},\mathcal{D}}(v_1) = u(v_2) - u(v_1)$. Consider the case $u(v_2) \geq u(v_1)$. The ratio of the probabilities, from lines 11-16, is $\frac{p_{v_1 \to v_2}}{p_{v_2 \to v_1}} = \frac{1/2}{(1/2) \cdot e^{-(u(v_2)-u(v_1))/(kn)}} = e^{(u(v_2)-u(v_1)) \cdot \frac{1}{kn}}$, which is an exponential function of $\mathrm{IF}_{\hat{u},\mathcal{D}}(v_2) - \mathrm{IF}_{\hat{u},\mathcal{D}}(v_1)$ from the equation above. One can verify that we get the same result for the case $u(v_1) > u(v_2)$. □

Theorem IV.3 shows that the influence function of expected utility from a given node is the difference between the utility of the node and that of the whole distribution. Moreover, the theorem shows that Algorithm 1 examines the IF difference of nodes, and uses it to guide the movement of chips and hence to change the node weight distribution $\mathcal{D}$.

3) Correctness Analysis of Our Algorithm: The following theorem shows that the learned distribution $\mathcal{D}$ is as desired. In particular, adaptive to the dynamic data and pattern/workload changes, the execution of Algorithm 1 tends to stay at a state with a probability being an exponential function of the expected temporal utility of the state (i.e., a higher utility has a much higher probability).

Theorem IV.4. *Let the numbers of chips at each node of $\mathcal{G}$ in $\mathcal{D}$ be the state during the execution of Algorithm 1. Then the state follows a Markov chain with a unique stationary distribution, in which the probability of state s is proportional to e^{u_s}, where u_s is the expected temporal utility of s.*

Proof. The total number of chips stays constant and there is a finite number of nodes, so there is a finite number of states. The next state only depends on the current state (lines 11-16), making it a finite Markov chain. The state transition graph is strongly connected, so the Markov chain is *irreducible* [28]. Also, there is a positive probability that a state will stay put in the next step (no chip is moved), so the chain is *aperiodic.* A finite, irreducible, and aperiodic Markov chain must have a unique stationary distribution [28]. At a discrete state s of $\mathcal{D}$, let the corresponding probability at node v_i be p_i (normalized distribution based on the number of chips). From the definition of temporal utility U_s at state s, we have

$$u_s \equiv \mathbf{E}[U_s] = \sum_{i=1}^{n} p_i u_i \tag{1}$$

where u_i is the temporal utility at node v_i $(1 \leq i \leq n)$. Now let $\Delta p \equiv \frac{1}{kn}$. Since there are kn chips in total (line 1), moving 1 chip in line 13 or 16 translates to moving Δp probability from one node to another. Let the two vertices in the sample of a step be v_w and v_l with $u_w \geq u_l$. There are two possible state transitions from state s—one is in line 13 to state s^+ while the other is in line 16 to state s^-. The expected temporal utility of state s^+ is

$$\begin{aligned} u^+ &= \sum_{\substack{i=1\dots n,\\ i\neq w,l}} p_i u_i + (p_w + \Delta p) u_w + (p_l - \Delta p) u_l \\ &= \sum_{i=1}^{n} p_i u_i + \Delta p (u_w - u_l) = u_s + \frac{u_w - u_l}{kn} \end{aligned} \tag{2}$$

where the last equality is based on Eq 1 and the definition of Δp. Likewise, the expected temporal utility of state s^- is

$$\begin{aligned} u^- &= \sum_{\substack{i=1\dots n,\\ i\neq w,l}} p_i u_i + (p_w - \Delta p) u_w + (p_l + \Delta p) u_l \\ &= \sum_{i=1}^{n} p_i u_i - \Delta p (u_w - u_l) = u_s - \frac{u_w - u_l}{kn} \end{aligned} \tag{3}$$

We now show that, the probability of state s in the stationary distribution must be $\pi_s = e^{u_s}/Z$, where u_s is the expected temporal utility of s and Z is a normalization constant. Without loss of generality, consider two valid neighboring states s^- and s whose expected temporal utilities are in Eq 3 and Eq 1, respectively. The transition probability from s^- to s is $p_{s^- \to s} = 1/2$ (line 11), while $p_{s \to s^-} = \frac{1}{2} e^{-\frac{u_w - u_l}{kn}}$ (line 14). We now have

$$\pi_s \cdot p_{s \to s^-} = \frac{e^{u_s}}{Z} \cdot \frac{1}{2} \cdot e^{-\frac{u_w - u_l}{kn}} = \frac{e^{u^-}}{Z} \cdot \frac{1}{2} = \pi_{s^-} \cdot p_{s^- \to s} \tag{4}$$

where the second to last equality is due to Eq 3. Eq 4 shows $\pi_s = e^{u_s}/Z$. □

Theorem IV.4 shows that Algorithm 1 has a desired property—a distribution $\mathcal{D}$ that has a higher utility will have a much larger probability than one with a lower utility. Furthermore, the distribution learning is adaptive to data/workload shifts. In turn, the training is adaptive to $\mathcal{D}$.

V. Sampling Nodes from a Graph-KDE Smoothed Distribution

In this section, we begin with some background of kernel density estimation (KDE), and how it could fit in our problem. Then we propose a novel node sampling algorithm that is equivalent to sampling from a sum of graph KDE kernel functions. The advantage is that the sampling distribution ends up being a smoothed version of the node-weight distribution learned in Algorithm 1. This improves online training efficiency and reduces training resource usage.

A. Graph KDE

1) Preliminaries: In statistics, *kernel density estimation* (KDE) is the application of kernel smoothing for probability density estimation [14], [15]. It is used to model the distribution of a variable based on a random sample. Let $x_1, x_2, \dots, x_n$ be a sample drawn from an unknown probability density function f. Then the kernel density estimate of f is given by

$$\tilde{f}(x) = \frac{1}{n} \sum_{i=1}^{n} K_h(x - x_i) \tag{5}$$

where K_h is the kernel function, $K_h(t) = \frac{1}{h} K(\frac{t}{h})$, and h is the *bandwidth* parameter (for smoothing). Intuitively, the true value of $f(x)$ is estimated as the average "distance" from x to the sample data points x_i. The "distance" between x and x_i is calculated via a kernel function $K(t)$. Popular kernel functions include Gaussian, Epanechnikov, and exponential functions. So, for Gaussian kernels, $\tilde{f}(x)$ is a mixture of Gaussian functions, each of which is centered at sample data point x_i.

2) Random-walk KDE: For a large dynamic network, adaptively learning and revising the weights of each node to its up-to-date value may have a significant time lag. On the other hand, since message passing of DGNN is between each node and its neighborhood, the weights of two adjacent nodes should not be far apart. In other words, the weight distribution should be "smooth" along the edges.

A reasonable assumption widely used in the kernel density estimation [14], [15] is that a smooth distribution can be approximated by the kernel mixture of a data sample. However, there are at least two challenges to adopt KDE in our problem:

- KDE is typically for continuous distributions, and in our context, we have discrete nodes with a graph topology as we intend to smooth the node-weight distribution.
- The optimal node-weight distribution is dynamically changing as the data changes over time.

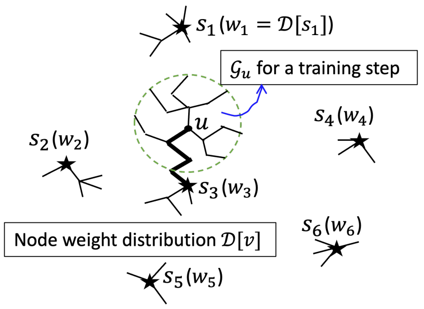
For the first challenge, fortunately, in our problem, we do not need to get the exact form of KDE function directly, but only need to sample nodes from a smoothed node-weight distribution, replacing the GetSampleNode function in Algorithm 1. Thus, our goal is to *devise a graph KDE sampling algorithm* that is equivalent to sampling from a sum of a number of KDE kernels.

For the second challenge, we essentially sample from a sum of a *dynamic* set of kernel functions, based on a dynamic sample (set of nodes). Next, we present our random-walk based sampling algorithm in conjunction with Algorithm 1 to achieve a smoothed node-weight distribution and a more effective model update.

B. Graph KDE Based Node Sampling and the Analysis

1) Intuition and Basic Idea: As mentioned earlier, we only need to replace the sampling function GetSampleNode in Algorithm 1 (while moving chips at nodes is still the same). The basic idea is as follows. First consider how one would get a random sample from a classical KDE function as in Equation 5. It's simple—we first pick one of the n kernels $K_h(x - x_i)$ uniformly at random, and then draw a sample data point from that kernel function (e.g., a Gaussian function). In our context, it is analogous. Each kernel function corresponds to the neighborhood of a *center node* (which corresponds to a data point in the dynamic sample). For a classical Gaussian kernel (or another type), a data point that is closer to the kernel center has a higher probability (and a farther one has a lower probability). We perform a random walk from a center node (of a kernel), which will also achieve the negative correlation between the hop-distance to the center and the probability being sampled.

More specifically, we maintain a set (sliding window) of w *seed nodes* $\mathcal{S}_w$ which consists of recently sampled nodes and randomly chosen nodes. *$\mathcal{S}_w$ is the dynamic sample for KDE that creates w kernels.* To get a new sample node, we first



$s_1 (w_1 = \mathcal{D}[s_1])$
$\mathcal{G}_u$ for a training step
u
$s_2(w_2)$
$s_4(w_4)$
$s_3(w_3)$
Node weight distribution $\mathcal{D}[v]$
$s_6(w_6)$
$s_5(w_5)$

Fig. 3: Illustrating graph-KDE sampling for online training. A number of seeds (e.g., s_1 to s_6) are selected based on their weights ($w_i = \mathcal{D}[s_i]$). A short random walk is performed from a chosen seed s_3 and reaches node u, where a step of SGD training is performed in the induced subgraph $\mathcal{G}_u$ at u's 3-hop neighborhood.

pick a seed s from $\mathcal{S}_w$ based on the chip distribution. Then we perform a random walk from s and stop with a certain probability at any node during the walk, which will be the new sample node. This is illustrated in Figure 3.

```
Algorithm 2: GraphKDESampling ($\mathcal{G}, \mathcal{D}$)
Input: $\mathcal{G}$: a graph stream
       $\mathcal{D}$: chip distribution
Output: a node sample continuously over time
1  choose $w$ nodes $\mathcal{S}_w$ uniformly at random from $\mathcal{G}$ as
     the seeds
2  while true do
3      $s \leftarrow$ randomly chosen from $\mathcal{S}_w$ based on $\mathcal{D}$
4      while true do
5          with probability $q$
6              add $s$ to the continuous sample; break
7          else
8              $s \leftarrow$ choose one neighbor of $s$ uniformly
                 at random
9      with probability $p$
10         replace oldest node in $\mathcal{S}_w$ by $s$
11     else
12         replace oldest node in $\mathcal{S}_w$ by a node chosen
             uniformly at random from $\mathcal{G}$
```

2) Algorithm Details and Analysis: The details are presented in Algorithm 2. Lines 4-8 perform a random walk starting from s, and lines 9-12 update the seed window set $\mathcal{S}_w$. The *stop probability* q and parameter p will be studied in Section VI. Line 12 ("teleport") is to prevent the seed set from being confined to a densely connected but relatively isolated area of the graph.

In a nutshell, the graph KDE function is embodied in our novel sampling method that is equivalent to sampling from a sum of graph KDE kernels. Note that the loop in lines 2-12 is over continuous sampling calls—each sampling is only one iteration of the loop in lines 2-12, which has an expected time cost of $O(\frac{1}{q}) = O(1)$ and a space complexity of $O(w) = O(1)$. We perform an analysis as follows.

Theorem V.1. *The effective sampling density at node v, denoted as $p_{kde}(v)$, is the sum of at least two terms, one of which is positively correlated with $\mathcal{D}(v)$, while the other one is a linear combination of the $\mathcal{D}(u) \cdot p_{kde}(u)$ of all nodes u in v's neighborhood, whose coefficients decrease as u's hop distance from v increases along a path.*

Proof. For v to be a sample, it must be from one of the two events: (E_1) v is one of the seeds in $\mathcal{S}_w$ that is subsequently chosen to begin a random walk (line 3) and the walk stops right at v (line 6). (E_2) A random walk started from one of v's neighbors stops at v (line 6). As these two events are disjoint, the probability $p_{kde}(v) = Pr(E_1) + Pr(E_2)$. The first term is $Pr(E_1) = Pr(v \in \mathcal{S}_w) \cdot \frac{\mathcal{D}(v)}{\sum_{u \in \mathcal{S}_w} \mathcal{D}(u)} \cdot q$. Consider $Pr(v \in \mathcal{S}_w)$. Initially each node may join $\mathcal{S}_w$ with a constant probability $\frac{w}{n}$ (line 1). Line 12 (with probability $1-p$) also chooses a node uniformly at random to include it in $\mathcal{S}_w$. Otherwise a node may be included in $\mathcal{S}_w$ after it is selected as a sample (line 10), which has a probability $p_{kde}(v)$. So the first term is positively correlated with $\mathcal{D}(v)$.

The second term is $Pr(E_2) = \sum_{u \in \mathcal{N}_h(v), h \geq 1} [(1-q)^h q \cdot \prod_{u_i \in path(u,v)} \frac{1}{deg(u_i)} \cdot Pr(u\, starts\, a\, walk)]$, where $\mathcal{N}_h(v)$ denotes the set of nodes that are h-hop distance to v. From the previous analysis, the probability that u starts a walk is proportional to $\mathcal{D}(u) \cdot p_{kde}(u)$ after some initial time. It is also clear that the coefficient $(1-q)^h q \cdot \prod_{u_i \in path(u,v)} \frac{1}{deg(u_i)}$ decreases as h (i.e., u's hop distance from v) increases along a path to v. □

Same as classical KDE kernels (Equation 5), the probability of a data point (node) within a kernel component decreases as it is farther away from the center (seed). The distance metric between two data points (nodes) here is the hop-distance (i.e., the shortest path distance in the graph). The parameter q determines the *smoothness*—a smaller q gives a smoother distribution, which is analogous to the role of the bandwidth parameter h in Equation 5 of classical KDE.

Furthermore, it is a form of *weighted* KDE [29] where each data-point in $\mathcal{S}_w$ carries a weight. The chip counts $c_1, \ldots, c_w$ of the seeds in $\mathcal{S}_w$ are the weights of the w kernels. In general, for a node v that is h_1 hops away from v_1 in $\mathcal{S}_w$, h_2 hops away from $v_2, \ldots,$ and h_w hops away from v_w, the probability that it is the new sample node is the sum of its corresponding probabilities in each of those w kernels. This is consistent with the classical KDE.

Consider a specific situation where a kernel center node has a high degree (or is in a dense area of the graph)—each of its neighbors will have a smaller probability to be chosen in the random walk, but that is desired for our sampling purpose—as long as some node in that area is chosen as a sample node, the GNN based training will perform message passing in that area of the graph. Finally, by using a dynamic sample ($\mathcal{S}_w$ for w kernels), graph KDE is adaptive to data distribution shifts.

Intuition for why KDE sampling helps. Some regions of the graph are more profitable for training, i.e., they are the peaks of a relatively smooth node-weight distribution $\mathcal{D}$ in Algorithm 1. The vanilla version of Algorithm 1 performs point-by-point incremental update of the vertex probabilities in $\mathcal{D}$. By contrast, KDE sampling "plants" a number of seeds in the graph; once a seed is close to a peak region, it learns the high probability mass and the random-walk KDE grows from the seed in the peak region, as seed-selection tends to stick there due to the learned high probability-mass that spans many vertices reached in the region. In other words, the seeds and the smooth probability mass function from Theorem V.1 enable Algorithm 1 to spend more time at high-probability vertices and hence reach the true distribution $\mathcal{D}$ faster.

VI. Experiments

We have performed a comprehensive empirical study to evaluate our work, from the motivation to the effectiveness of our methodologies. We use three real-world graph stream datasets and compare against six baseline state-of-the-art dynamic graph neural network models. We aim to answer the following research questions (RQ):

- **RQ1:** Regarding motivation, is continuous, online, adaptive training actually needed for *graph streams*?
- **RQ2:** How does our weighted adaptive training compare to full/uniform training for graph streams in terms of training time and memory usage? In addition, how does our graph-KDE sampling version perform? How do they compare against the six baseline models?
- **RQ3:** How do the choices of the parameter values of our method affect training time, memory consumption, and accuracy?

A. Dataset Description and Machine Setup

We use five real-world graph stream datasets as follows.

- **Bitcoin:** This dataset [30], [31] maps Bitcoin transactions to real entities belonging to licit categories (exchanges, wallet providers, etc.) versus illicit ones (scams, malware, etc.). A major task is to classify the illicit and licit nodes. We treat a transaction as a node and a flow of Bitcoins between two transactions as a dynamic edge. Each node has 166 features. There are 203,769 nodes and 234,355 edges (700 MB).
- **Reddit:** This network represents the directed connections between subreddits (a subreddit is a community) [32]. It is extracted from Reddit data of 2.5 years. Each ink is annotated with the sentiment of the source post towards the target post, and the text property vector of the source post. There are 55,863 nodes and 858,490 edges (4 GB).
- **Taxi:** It contains the information of all taxi trips in the New York City in 2013 [33]. It has 14 attributes including taxi and trip information. We partition the geographic area into 20×20 grids. There are two types of nodes—the grid nodes and 14.5M trip nodes. Each trip node has 2

temporal edges connecting 2 grid nodes; thus there are 29M edges (total 30 GB).
- **Stack Overflow:** This common link prediction data [34] describes the interactions on the Stack Overflow platform. The nodes are users and the edges are answering and commenting activities. There are 2,601,977 nodes and 63,497,050 edges (1.5 GB) over one week.
- **UCI Messages:** This is another common link prediction dataset [35] consisting of private messages sent on an online social network system among students, where nodes are users and edges are messages. It has 1,899 nodes and 59,835 edges (1.1 MB) over a one-week period.

All the algorithms described in this paper are implemented in Python. All the experiments are run on a machine with Intel i7-8750H CPU and GeForce RTX 2080 GPU.

B. Continuous Predictive Queries and Self-supervision Tasks

Each dataset is divided into time *steps*. Each step corresponds to a graph snapshot. For the Bitcoin dataset, the self-supervised learning targets are to predict whether a transaction in the next time step is illicit or licit. This is called self-supervision as the information being predicted is readily a node label within the dataset. The supervised analytics workload is to monitor predictive events that there will be bitcoin flows between licit and illicit transactions.

For the Reddit dataset, the self-supervised targets are to predict the sentiment (positive or negative) of edge posts; this is self-supervision as it is about labels directly provided by the dataset. The supervised workload is to monitor subreddits that will have negative-post ratio above a threshold. For the Taxi dataset, the self-supervised targets are to predict the trip distance labels, while the supervised ones are to monitor whether certain grid nodes (i.e., locations) have a fraction of slow incoming/outgoing trips above a threshold. For each of the three graph streams, a number of such continuous predictive queries based at different nodes/edges are being monitored. We also use common link prediction datasets (e.g., in [20], [21]) Stack Overflow and UCI Messages to perform continuous link predictions in the next snapshots.

C. Baseline Models

As our architecture allows to plug in a given DGNN model and reduce its resource consumption, we have tried a number of DGNN models, and our methods can reduce both training time and memory usage on all of them. We present the experimental results comparing against seven DGNN baseline models: (1) TGCN [16], (2) DCRNN [17], (3) GCLSTM [18], (4) DyGrEncoder [19], (5) ROLAND [20], (6) WinGNN [21], and (7) EvolveGCN [22].

All these baseline models characterize message passing at graph snapshot level, as well as handle the sequential aspect over time. These DGNN models are representative state-of-the-art ones published in recent years. We note that our approach can be applied to many other DGNN models as well and reduce their resource usage.

D. Research Question 1: Need for Continuous Model Updating

We first verify the motivation of online continuous model updating. While online learning has been studied, to our knowledge, little has been done to investigate the need and extent of online adaptive learning for dynamic graphs and graph streams.

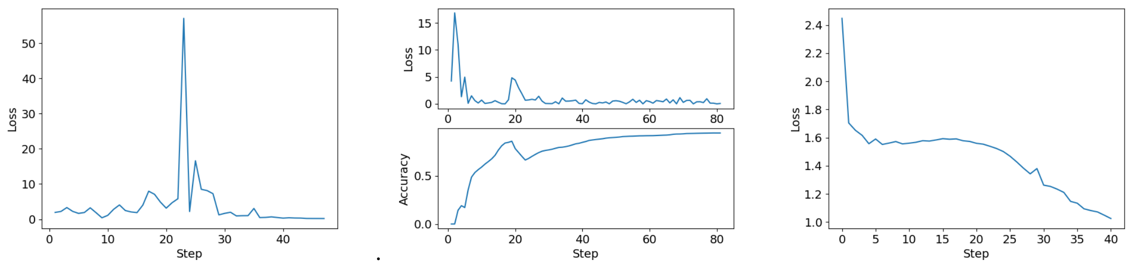
The results are shown in Figure 4. For each dataset, we compare online continuous learning at each time step using a recent sliding window of graph snapshots (Figure 4a) and only continuous training for the first 1/4 of the steps in the experiments (Figure 4b). We display the continuous target evaluation loss at each step (predicting the targets of the next step).

The result for the Bitcoin data shows that continuous learning (first plot in Figure 4a) overall achieves low prediction errors at nearly all time steps, except for occasional steps (step 22) where there is a sudden data/pattern drift. Yet online learning is able to quickly adapt the change in the model and bring the error down again for subsequent steps. By contrast, in the first plot of Figure 4b, we only continuously train the model using sliding windows at each step up to step 10. After that, the training stops and we only evaluate the model (to predict the next step targets) for the remaining steps. While the model still performs well for another 10 steps or so, it quickly deteriorates significantly—the MSE loss goes all the way to over 2000, while the maximum error in the top-left plot is less than 60. This clearly demonstrates the need for online adaptive training and model updating.

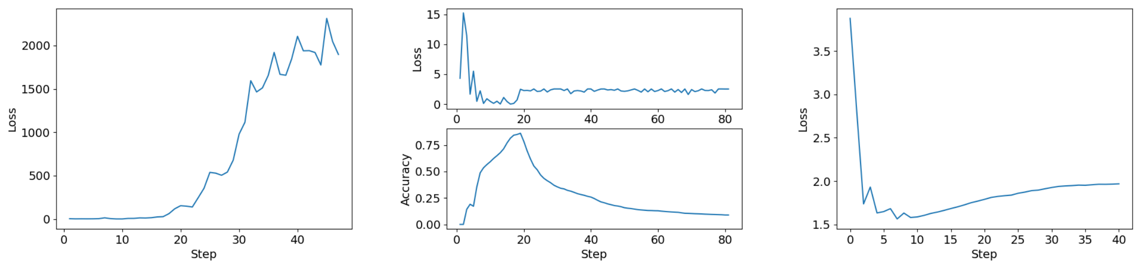
We observe similar results for the Taxi data, where, in the Figure 4b right plot, the model is only continuously trained for the first 10 steps, and it deteriorates increasingly over the subsequent steps. Likewise, for the Reddit data, the comparison between the middle one of Figure 4a and middle one of Figure 4b shows that, when the continuous training stops at step 20 (middle-top one, Figure 4b), the error stays at a higher level than continuous training all the way. As the loss difference is not as significant as the other two datasets, we further compare the accuracy of continuous predictions in the two middle-bottom plots. The accuracy drops sharply over time after the continuous training stops at step 20 in the middle-bottom plot of Figure 4b, while it stays at a high level in Figure 4a.

E. Research Question 2: Resource Consumption Comparisons with Baselines

Next we compare our weighted adaptive training (Algorithm 1) with the default full/uniform training using a baseline DGNN model, as well as the weighted training augmented by our graph-KDE method (Algorithm 2). We use the default parameter values as determined in Research Question 3 below. We measure the total training time (in seconds) over a fixed number of steps, maximum memory consumption during training, and the average prediction MSE (error). For monitoring binary events and link predictions, as in [20], [21], we also use commonly-used metrics AUC, mean reciprocal rank (MRR), and accuracy. The error bars



(a) Continuous training on Bitcoin (left), Reddit (middle) and Taxi (right) data



(b) Partial training on Bitcoin (left), Reddit (middle) and Taxi (right) data

Fig. 4: The need for online continuous training for datasets.

TABLE I: Experiments with event monitoring workloads

Dataset	Model	Method	Training Time	Memory	Error	AUC	MRR
Bitcoin	TGCN	Full/Uniform	107.51±9.27	31MB	4.25±0.07	0.779±0.006	0.696±0.011
		Weighted	12.92±1.05	**12MB**	4.24±0.05	0.780±0.008	0.693±0.009
		KDE	**11.85±0.78**	**12MB**	4.12±0.05	0.780±0.005	0.697±0.009
	WinGNN	Full/Uniform	126.63±10.72	31MB	4.51±0.06	0.733±0.009	0.615±0.013
		Weighted	14.24±1.29	**12MB**	4.50±0.07	0.731±0.006	0.619±0.012
		KDE	**12.51±1.25**	**12MB**	4.52±0.06	0.735±0.010	0.616±0.010
Reddit	GCLSTM	Full/Uniform	19.22±1.93	4.5MB	1.08±0.09	0.712±0.007	0.815±0.012
		Weighted	8.24±0.41	**272KB**	1.09±0.11	0.709±0.009	0.819±0.008
		KDE	**7.33±0.57**	**272KB**	1.07±0.09	0.711±0.008	0.816±0.011
	DyGrEncoder	Full/Uniform	17.75±1.26	4.5MB	0.99±0.07	0.719±0.011	0.826±0.006
		Weighted	5.43±0.68	267KB	0.98±0.09	0.726±0.006	0.829±0.008
		KDE	**5.14±0.65**	**265KB**	0.98±0.11	0.725±0.006	0.830±0.009
Taxi	DCRNN	Full/Uniform	387.96±27.21	95MB	4.19±0.08	0.687±0.013	0.715±0.012
		Weighted	7.14±0.77	**15MB**	4.17±0.07	0.685±0.012	0.721±0.008
		KDE	**3.11±0.59**	**15MB**	3.85±0.05	0.690±0.008	0.719±0.011
	ROLAND	Full/Uniform	78.73±6.59	153MB	4.28±0.07	0.679±0.012	0.706±0.015
		Weighted	**2.05±0.37**	**34MB**	4.23±0.09	0.678±0.010	0.709±0.011
		KDE	2.06±0.32	**34MB**	4.17±0.06	0.681±0.007	0.710±0.009

are over 10 runs. The results are shown in Table I for event-monitoring workloads and in Table II for the two continuous link prediction datasets.

We adjust each method's training interval so that they give similar errors/accuracy, and then we can fairly compare each method's time spent on training and its maximum memory usage. For each dataset, we display the results of one or two DGNN baseline models (the trends are similar for other combinations). For all the datasets, our weighted adaptive learning and our graph-KDE variant consume up to 2 orders of magnitude less training time, significantly less memory, when the prediction accuracy is similar.

Table I and Table II also show that the graph-KDE variant achieves even less training times in general than the plain weighted training with the same accuracy. This verifies the reason and intuition stated at the end of Section V. In contrast to the vanilla version of Algorithm 1 that does point-by-point incremental probability update, graph-KDE enables Algorithm 1 to sample and grow seeds in smooth high-probability regions. Overall, the training using learned weights and its graph-KDE variant consume significantly less memory and computation because they judiciously perform weighted training on fewer and more important nodes. Thus, with less training, it can achieve the same level of accuracy.

F. *Research Question 3: Impact of Parameter Value Choices*

We have studied how the choices of various parameters of our ultimate method, the graph-KDE enhanced weighted

TABLE II: Experiments with common link prediction datasets

Dataset/Model	Method	Training Time	Memory	Accuracy	AUC	MRR
Stack Overflow (EvolveGCN)	Full/Uniform	220.15±10.74	585MB	0.667±0.021	0.739±0.018	0.788±0.015
	Weighted	32.57±3.18	**6MB**	0.692±0.017	0.745±0.016	0.795±0.010
	KDE	**32.41±4.65**	**6MB**	0.660±0.018	0.747±0.015	0.771±0.012
UCI Messages (ROLAND)	Full/Uniform	98.76±7.52	35MB	0.779±0.008	0.875±0.019	0.907±0.011
	Weighted	25.62±3.19	**2MB**	0.781±0.013	0.874±0.008	0.905±0.009
	KDE	**25.15±2.86**	**2MB**	0.783±0.010	0.875±0.011	0.907±0.007

TABLE III: Experiments with various parameters and datasets

Dataset/Model	Parameter	Training Time	Memory	Error	AUC	MRR
Bitcoin (TGCN)	$Interval = 1$	11.85±0.78	12MB	4.12±0.05	0.780±0.005	0.697±0.009
	$Interval = 2$	6.51±0.57	12MB	4.15±0.03	0.774±0.009	0.691±0.011
	$Interval = 5$	2.37±0.41	11MB	4.19±0.05	0.772±0.010	0.692±0.009
	$Interval = 10$	1.24±0.22	10MB	25.86±3.31	0.759±0.012	0.634±0.015
Reddit (DCRNN)	$\#pairs = 1$	5.88±0.74	257KB	0.99±0.10	0.723±0.007	0.829±0.011
	$\#pairs = 3$	18.12±2.78	278KB	0.98±0.12	0.723±0.006	0.831±0.008
	$\#pairs = 7$	42.47±4.65	271KB	0.95±0.11	0.727±0.006	0.835±0.007
Taxi (GCLSTM)	$\#seeds = 5$	2.47±0.29	16MB	7.17±0.11	0.669±0.013	0.702±0.008
	$\#seeds = 15$	39.76±4.77	38MB	3.76±0.08	0.695±0.009	0.721±0.011
	$\#seeds = 50$	14.51±3.14	27MB	3.84±0.09	0.683±0.010	0.714±0.006
Bitcoin (DyGrEncoder)	$q = 0.1$	11.79±0.91	12MB	3.58±0.11	0.769±0.009	0.690±0.007
	$q = 0.5$	11.23±0.78	11MB	3.61±0.08	0.762±0.007	0.674±0.012
	$q = 0.9$	11.19±0.65	11MB	8.72±0.12	0.721±0.009	0.618±0.006
Reddit (WinGNN)	$p = 0.1$	6.46±0.67	270KB	1.09±0.12	0.718±0.011	0.821±0.009
	$p = 0.5$	6.71±0.45	268KB	0.99±0.11	0.713±0.007	0.827±0.012
	$p = 0.8$	6.69±0.48	265KB	1.07±0.14	0.717±0.012	0.823±0.009

training, affect the training time, memory usage, and predictive query accuracy. The results are shown in Table III. Under all the datasets and DGNN baseline models, we observe similar patterns as we vary the parameter choices; thus, for succinctness, we only display one dataset/model combination for each parameter. We first vary the interval of continuous training, ranging from 1 time step to 10 time steps. While the total training time decreases proportionally as we increase the training interval, the average prediction error does not increase significantly when the interval is no more than 5. This indicates that one can find a suitable training interval to cut down resource consumption without sacrificing much on the accuracy.

Next we vary the number of pairs of sample nodes per step used for adjusting the weight distribution. While the total training time increases proportionally with the number of pairs, the accuracy only slightly increases. This indicates that we may not need many pairs of sample nodes per step. Our default is 1.

We then vary the number-of-seeds parameter of graph KDE sampling (i.e., w in Algorithm 2). Our default is 15. When we set it to a much smaller value, 5, the error significantly increases because fewer seeds imply a larger variance on the quality of the sampled nodes. On the other hand, when the number of seeds is a much larger value 50, some seeds will be at sparser areas of the graph, which makes training faster (recall from Algorithm 1 that the training time/space cost is proportional to d^L where d is node degree and L is the number of DGNN layers), but the error increases as it explores areas of the graph that are less "profitable" for training.

Recall that the graph-KDE sampling has two probability parameters q (probability to stop the random walk, default 0.5) and p (probability to make the current sample a new seed, default 0.8). When q is smaller (0.1), the random walk goes farther, and the distribution is smoother. This slightly increases the training cost, but results in slightly better accuracy in this case. On the other hand, we find that parameter p does not significantly change the accuracy nor the training cost.

G. *Summary of Experimental Results*

Our comprehensive experiments using five real-world graph stream datasets and seven baseline models show that continuously learning model updates while answering and updating predictive queries is much needed to adapt to the data changes in graph streams. Our experiments with various parameter choices help us empirically pick the parameter values, some of which (such as the training interval) provide a tradeoff between resource consumption and query result accuracy. With all six baseline dynamic graph neural network models, our weighted adaptive model updates and query answering take up to 2 orders of magnitude less training time and significantly less memory when the continuous predictive query results have the same accuracy. The experiments also show that our graph-KDE-sampling variant is effective and, in most cases, further reduces the continuous training resource usage slightly.

VII. Related Work

Dynamic Graph Neural Networks (DGNN). We refer the readers to two excellent surveys on DGNN [9], [10]. We focus on discrete-time dynamic graphs (DTDG), for which typically a DGNN consists of a GNN model (as in static graphs) and a sequence model. We can plug in a given DGNN model as a component of our work (Figure 2) and reduce its training time and memory usage, as shown in our experiments in Section VI. We have presented the results with six representative recent DGNN models, and they all show a very similar pattern in the reduction of training time and maximum memory usage under the same accuracy. Our work is general and applies to any models for DGNN, as we find induced subgraphs for node-level training partitioning, which is independent of the specific DGNN model.

Online Learning and Lifelong Learning. Online learning has a long history. It learns to update models from data streams sequentially [36], [37], with algorithms including Perceptron [38], Online Gradient Descent [39], and Passive Aggressive [40], all for learning linear models. The work on online Learning with kernels includes [41]. Recently online learning is studied for deep learning [11]–[13]. Lifelong learning considers systems that can continually learn many tasks over a life time [42]. It is a concept related to online learning, but it is not concerned with the *online continuous* and *real-time* aspect. None of such previous work studies online continuous learning under DGNN for *continuous predictive queries* as we do, where we aim to reduce the training time and memory usage.

Influence Functions and Subsampling. Influence functions originated in *robust statistics* [27]. Intuitively, a statistic is robust if arbitrary changes to a small part of data, or small changes to all data, result in only small changes to the value of the statistic. This notion of sensitivity is captured in *influence functions* (IFs). In the pioneering work of Koh and Liang [43], IFs are used to interpret machine learning predictions. However, IFs are in general well-defined and studied for models such as logistic regression [43], where the underlying loss-function is convex. Basu et al. [44] provide a comprehensive study when the convexity assumption is violated, which is the case in deep learning, and find a pessimistic answer: influence estimation is quite fragile for a variety of deep networks. For the same reason, subsampling using IFs as weights [45] is only limited to asymptotically linear estimators.

Thus, we do not directly use influence functions for the node weights. Our novel weight-learning and sampling are adaptive and applicable to DGNN, tailored to online learning, while our analysis in Theorem IV.3 draws a connection to the influence functions of the learning *utility* from any given node.

Kernel Density Estimation (KDE). KDE is a nonparametric approach that estimate the density distribution directly from the data [46], [47]. It is a well-established technique in both statistics and machine learning [14], [15]. KDE has been successfully applied in many applications. Just to show but a few examples, Zheng et al. propose randomized, parallel, and distributed implementations of KDE on very large data [48]. Qahtan et al. study the estimation of the density of spatiotemporal data streams [49]. Their method can efficiently estimate the density function with linear time complexity using interpolation on a kernel model.

Our KDE part of work is drastically different. We devise a novel *graph KDE* to smooth a discrete distribution over the vertices of a graph. For the purpose of our work, we do not need to get the exact form of the KDE function (which is a sum of kernel functions); but rather, the graph KDE function is embodied in a sampling method that is equivalent to sampling from a sum of graph KDE kernels, which is based on random walks and a dynamic sample.

Curriculum Learning and Boosting. Our online learning of a node-weight distribution for model updates is also remotely related to curriculum learning and boosting. The basic idea of curriculum learning (CL) [50] is to sort examples into a sequence ordered by easiness. Starting with the easiest examples, simpler concepts are learned first, and then introducing gradually more difficult examples speeds up the training. Likewise, in boosting algorithms [51], difficult examples are gradually emphasized.

By contrast, graph vertices are not the same as "examples" and we do not sort them by easiness. Instead, we adaptively learn a distribution over them, and use the vertices as guidance for training both supervised and self-supervised targets, as well as for back-propagation into the induced subgraphs.

VIII. Conclusions and Discussions

Monitoring predictive events and queries in graph streams has an important role in practical applications. We find that continuous learning and model updating is crucial to keeping the predictive query results current and correct. We aim to reduce the time and memory usage in training given the same prediction accuracy. Our approach partitions the incremental training at the graph node level, and we propose a randomized algorithm to continuously learn a weight distribution over nodes, which is simultaneously used to do model updates, which in turn give feedback to the node-weight learning. We also devise a novel graph KDE embodied in its sampling algorithm to smooth the node-weight distribution, to make the model updates more effective, and to further reduce the resource usage. Our experiments demonstrate the advantage of our approach—to achieve the same prediction accuracy, the training time ranges from several times to two orders of magnitude shorter, and the maximum memory consumption is several times to 20 times smaller.

One assumption of our approach is that dynamic online training is more profitable at certain regions of the graph and that the distribution is relatively smooth. In other words, the node-weight distribution that we learn has the *homophily* property in the graph, which is the base of our graph-KDE sampling design.

References

[1] A. Johnson, T. Pollard, L. Shen, L. Lehman, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L. Celi, and R. Mark, "MIMIC-III, a freely accessible critical care database," *Scientific Data*, 2016.

[2] A. McGregor, "Graph stream algorithms: A survey," *SIGMOD Rec.*, vol. 43, no. 1, p. 9–20, 2014.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, ser. PODS '02, 2002, p. 1–16.

[4] T. Healy, "The value of predictive analytical applications as digital health tools," *Pharmaceutical Outsourcing*, pp. 32–34, 2022.

[5] SAP, "Predictive analytics: The future of data analysis," *SAP Business Technology Platform*, 2023.

[6] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *IEEE Transactions on Neural Networks*, vol. 5, no. 1, pp. 61–80, 2016.

[7] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," *arXiv:1710.10903*, 2018.

[8] M. M. Bronstein, J. Bruna, T. Cohen, and P. Veličković, "Geometric deep learning: Grids, groups, graphs geodesics and gauges," *arXiv:2104.13478*, 2021.

[9] J. Skarding, B. Gabrys, and K. Musial, "Foundations and modelling of dynamic networks using dynamic graph neural networks: A survey," *IEEE Acces*, vol. 9, 2021.

[10] M. S. Kazemi, "Dynamic graph neural networks," in *Graph Neural Networks: Foundations, Frontiers, and Applications*, L. Wu, P. Cui, J. Pei, and L. Zhao, Eds. Singapore: Springer Singapore, 2022, pp. 323–349.

[11] G. Zhou, K. Sohn, and H. Lee, "Online incremental feature learning with denoising autoencoders," *AISTATS*, 2012.

[12] J. Lee, J. Yun, S. Hwang, and E. Yang, "Lifelong learning with dynamically expandable networks," *arXiv:1708.01547*, 2017.

[13] D. Sahoo, Q. Pham, J. Lu, and S. C. H. Hoi, "Online deep learning: Learning deep neural networks on the fly," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, 7 2018, pp. 2660–2666.

[14] Z. I. Botev, J. F. Grotowski, and D. P. Kroese, "Kernel density estimation via diffusion," *Ann. Stat.*, vol. 38, no. 5, pp. 2916–2957, 2010.

[15] J. Kim and C. Scott, "Robust kernel density estimation," *Journal of Machine Learning Research*, pp. 2529–2565, 2012.

[16] L. Zhao, Y. Song, C. Zhang, Y. Liu, P. Wang, T. Lin, M. Deng, and H. Li, "T-GCN: A temporal graph convolutional network for traffic prediction," *IEEE Transactions on Intelligent Transportation Systems*, vol. 21, no. 9, pp. 3848–3858, 2020.

[17] Y. Li, R. Yu, C. Shahabi, and Y. Liu, "Diffusion convolutional recurrent neural network: Data-driven traffic forecasting," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2018.

[18] J. Chen, X. Wang, and X. Xu, "GC-LSTM: graph convolution embedded lstm for dynamic network link prediction," *Applied Intelligence*, vol. 52, p. 7513–7528, 2022.

[19] A. Taheri and T. Berger-Wolf, "Predictive temporal embedding of dynamic graphs," in *2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, 2019, pp. 57–64.

[20] J. You, T. Du, and J. Leskovec, "ROLAND: Graph learning framework for dynamic graphs," in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '22. Association for Computing Machinery, 2022, pp. 2358–2366.

[21] Y. Zhu, F. Cong, D. Zhang, W. Gong, Q. Lin, W. Feng, Y. Dong, and J. Tang, "WinGNN: Dynamic graph neural networks with random gradient aggregation window," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '23. Association for Computing Machinery, 2023, pp. 3650–3662.

[22] A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, T. B. Schardl, and C. E. Leiserson, "EvolveGCN: Evolving graph convolutional networks for dynamic graphs," in *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[23] Y. Wang, W. Jin, and T. Derr, "Graph neural networks: Self-supervised learning," in *Graph Neural Networks: Foundations, Frontiers, and Applications*. Springer, Singapore, 2022.

[24] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 2, pp. 318–327, 2020.

[25] J. Moon, J. Kim, Y. Shin, and S. Hwang, "Confidence-aware learning for deep neural networks," in *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[26] A. Björner, L. Lovász, and P. W. Shor, "Chip-firing games on graphs," *European Journal of Combinatorics*, vol. 12, no. 4, pp. 283–291, 1991.

[27] F. Hampel, P. Rousseeuw, E. Ronchetti, and W. Strahel, *Robust Statistics: The approach based on influence functions*. Wiley, NY, 1986.

[28] S. Meyn and R. Tweedie, *Markov Chains and Stochastic Stability*, ser. Communications and Control Engineering. Springer London, 2012.

[29] M. D. Porter and B. J. Reich, "Evaluating temporally weighted kernel density methods for predicting the next event location in a series," *Annals of GIS*, vol. 18, no. 3, pp. 225–240, 2012.

[30] (2023) Elliptic. At www.elliptic.co.

[31] M. Weber, G. Domeniconi, J. Chen, D. K. I. Weidele, C. Bellei, T. Robinson, and C. E. Leiserson, "Anti-money laundering in bitcoin: Experimenting with graph convolutional networks for financial forensics," in *KDD'19 Workshop on Anomaly Detection in Finance*, Anchorage, AK, USA, 2019.

[32] S. Kumar, W. L. Hamilton, J. Leskovec, and D. Jurafsky, "Community interaction and conflict on the web," in *Proceedings of the 2018 World Wide Web Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2018, pp. 933–943.

[33] (2023) New york taxi data. Available at https://chriswhong.com/open-data/foil_nyc_taxi/.

[34] A. Paranjape, A. R. Benson, and J. Leskovec, "Motifs in temporal networks," in *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, 2017.

[35] P. Panzarasa, T. Opsahl, and K. M. Carley, "Patterns and dynamics of users' behavior and interaction: Network analysis of an online community," *Journal of the American Society for Information Science and Technology*, vol. 60, no. 5, pp. 911–932, 2009.

[36] N. Cesa-Bianchi and G. Lugosi, *Prediction, learning, and games*. Cambridge University Press, 2006.

[37] S. C. H. Hoi, D. Sahoo, J. Lu, and P. Zhao, "Online learning: A comprehensive survey," *arXiv preprint arXiv:1802.02871*, 2018.

[38] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain," *Psychological review*, vol. 65, no. 6, 1958.

[39] M. Zinkevich, "Online convex programming and generalized infinitesimal gradient ascent," in *ICML*, 2003.

[40] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer, "Online passive-aggressive algorithms," *JMLR*, 2006.

[41] D. Sahoo, S. C. H. Hoi, and P. Zhao, "Cost sensitive online multiple kernel classification," *ACML*, 2016.

[42] S. Thrun, *Explanation-Based Neural Network Learning: A Lifelong Learning Approach*. USA: Kluwer Academic Publishers, 1996.

[43] P. W. Koh and P. Liang, "Understanding black-box predictions via influence functions," in *Proceedings of the 34th International Conference on Machine Learning*, Sydney, Australia, 2017.

[44] S. Basu, P. Pope, and S. Feizi, "Influence functions in deep learning are fragile," in *The Ninth International Conference on Learning Representations*, 2021.

[45] D. Ting and E. Brochu, "Optimal subsampling with influence functions," in *32nd Conference on Neural Information Processing Systems (NeurIPS 2018)*, Montréal, Canada, 2018.

[46] B. Silverman, *Density Estimation for Statistics and Data Analysis*. Routledge, 2018.

[47] J. Simonoff, *Smoothing Methods in Statistics*. Springer, New York, 1996.

[48] Y. Zheng, J. Jestes, J. M. Phillips, and F. Li, "Quality and efficiency for kernel density estimates in large data," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 433–444. [Online]. Available: https://doi.org/10.1145/2463676.2465319

[49] A. Qahtan, S. Wang, and X. Zhang, "Kde-track: An efficient dynamic density estimator for data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 3, pp. 642–655, 2017.

[50] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09. Association for Computing Machinery, 2009, p. 41–48.

[51] Y. Freund and R. E. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Computational Learning*

Theory, P. Vitányi, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 23–37.