DATA-FREE CONTINUAL GRAPH LEARNING

Anonymous authors
Paper under double-blind review

ABSTRACT

Graph Neural Networks (GNNs), which effectively learn from static graph-structured data become ineffective when directly applied to streaming data in a continual learning (CL) scenario. A few recent works study this so-called "catastrophic forgetting" problem in GNNs, where historical data are not available during the training stage. However, they make a strong assumption that full access of historical data is provided during the *inference* stage. This assumption could make the graph learning system impractical to deploy due to a number of reasons, such as limited storage, GDPR[1] data retention policy, to name a few. In this work, we study continual graph learning without this strong assumption. Moreover, in practical continual learning, models are sometimes trained with accumulated batch data but required to do on-the-fly inference with a stream of test samples. In this case, without being re-inserted into previous training graphs for inference, streaming test nodes are often very sparsely connected. It makes the inference more difficult as the model is trained on a much more dense graph while required to infer on a sparse graph with insufficient neighborhood information. We propose a simple Replay GNN (ReGNN) to jointly solve the above two challenges without memory buffers (i.e., data-free): catastrophic forgetting and poor neighbour information during inference. Extensive experiments demonstrate the effectiveness of our model over baseline models, including competitive baselines with memory buffers.

1 INTRODUCTION

Graph Neural Networks (GNNs) (Kipf & Welling, 2016; Hamilton et al., 2017) have been recognized as a valid tool for graph learning, showing promising performance on a variety of tasks. In a practical scenario, the graphs are often evolving over time, and meanwhile, previous data (e.g., some nodes and edges) are inaccessible sometimes. Traditional graph learning models (Hamilton et al., 2017; Kipf & Welling, 2016) often have limited performance in this setting because the unavailability of previous data leads to catastrophic forgetting (McCloskey & Cohen, 1989).

Continual learning (Thrun, 1995) aims to develop an intelligent system that can continuously learn from new tasks without forgetting learnt knowledge in the absence of previous data. Common continual learning scenarios can be roughly divided into two categories (Van de Ven & Tolias, 2019): task-incremental learning (TI) and class incremental-learning (CI) (Rebuffi et al., 2017). Recently, a few works (Liu et al., 2021; Zhou & Cao, 2021; Wang et al., 2020; 2022; Galke et al., 2021) begin to study Continual Graph Leaning (CGL), which is helpful when accumulating all history data over time is not feasible due to storage pressure or customer data storage consent, for instance, the 30-days right-of-erasure in General Data Protection Regulation (GDPR) [2].

However, an unique challenge in continual graph learning is overlooked: in addition to the catastrophic forgetting problem, the unavailability of previous data further poses challenges during inference. When existing works evaluate their models under the continual graph learning setting with citation graphs, social networks, or co-purchase graphs, during the inference, they re-insert test nodes into the previous training graph and then aggregate the neighborhood information of test nodes to make a prediction (Figure 2). Such evaluation method is common in a traditional graph learning setting. Nevertheless, in continual graph learning, it is not always practical. This is because

[1]https://en.wikipedia.org/wiki/General_Data_Protection_Regulation
[2]https://gdpr-info.eu/

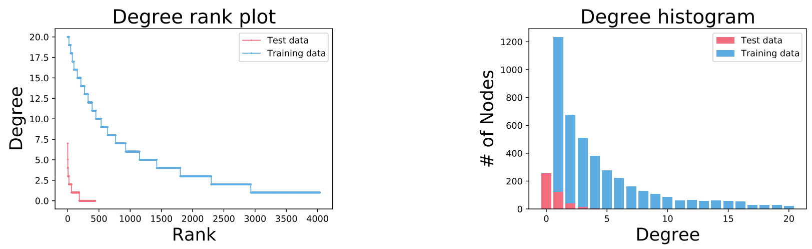

Figure 1: Node degree statistics of training and test data from a graph that records user interactions in Stack Exchange (an online platform). We extract data accumulated for a month as the training graph and data from the following few days as the test data. Obvious difference between training and test data *density* can be observed under continual graph learning setting (previous training nodes are unavailable).

in continual learning, training data (previous nodes) from previous tasks are no longer available at the current stage, including both training and *inference* stage. In a typical continual graph learning scenario, on-the-fly inference is sometimes required. For instance, in node classification, the model needs to infer the labels of streaming test nodes once it receives them, instead of accumulating a large number of test nodes to do a inference. Without re-inserting back to the training graph, the streaming test nodes of a relatively smaller size are often very sparsely connected. It means while the model is trained on a dense graph, it is required to infer on a much more sparse graph. Figure 1 showcases the above point in a real-world scenario. We study with the user interaction graph from the Stack Exchange platform [3] and observe notable difference between training and test data density under the continual graph learning setting. The above facts lead to a key challenge in addition to catastrophic forgetting in continual graph learning: *the neighborhood information available during the inference is very limited.* Different from existing works, we consider this practical yet ignored case in our paper and solve this challenge.

Besides, different form existing works that either focus on task-incremental and data-incremental, or focus class-incremental but use a buffer to store previous data for replay to prevent forgetting, in this paper, we focus on CI graph learning problem without memory buffers, i.e., data-free class-incremental graph learning. We further propose a generative replay based model, Replay GNN (ReGNN), to solve the above challenges. It outperforms baseline models including competitive baselines with memory buffers on several benchmark graph datasets.

The main contributions of this paper are as follows. We find an interesting and important case overlooked by existing works in continual graph learning in addition to catastrophic forgetting: the unavailability of previous training nodes and edges further lead to poor neighborhood information during the *inference* stage. Then we study continual graph learning with this ignored case under data-free class-incremental learning setting, which is a more difficult continual graph learning scenario and not studied by existing works. We further propose Replay GNN to address the problems and the experiments demonstrate its effectiveness.

2 Related Work

2.1 Graph Neural Networks

Graph neural networks are popular models for graph representation learning. GNNs can be roughly categorized into two groups: spatial methods and spectral methods. GraphSAGE (Hamilton et al., 2017) is a typical spatial method, which directly aggregates node representations from a node's neighborhoods to obtain its representation, while graph convolutional network (GCN) (Kipf & Welling, 2016), a typical spectral method, learns graph representation in the spectral domain, which alleviates over-fitting on local neighbors via the Chebyshev expansion. Both classic spatial and spectral methods need the entire graph during model training. To learn more effectively when scaling to large evolving graphs. Sampling strategies (Hamilton et al., 2017; Chen et al., 2018) are developed so that only part of the graph is required for each iteration during training.

[3] https://stackexchange.com/

2.2 Continual Learning

Continual learning studies the problem of learning from streaming data, with the aim of continuously acquiring new knowledge while maintaining its learnt knowledge. Current most studied continual learning settings can be roughly categorized into two categories (Van de Ven & Tolias, 2019) :Task-Incremental Learning (TI), and Class-Incremental Learning (CI). Class-incremental learning (CI) (Rebuffi et al., 2017) is a harder continual learning problem due to the unavailability of task-IDs. Representative continual learning models, such as (Kirkpatrick et al., 2017) and (Li & Hoiem, 2017), achieve promising performance on TI benchmarks, but suffer from notably forgetting on simple CI benchmarks. Existing approaches to solve the CI problem can be divided into three categories (Ebrahimi et al., 2020), including the replay-based methods (Rolnick et al., 2019; Chaudhry et al., 2019), structure-based methods (Yoon et al., 2017), and regularization-based methods (Kirkpatrick et al., 2017; Aljundi et al., 2018). Progress has been made in continual learning in recent years, however, only a few of them study continual learning with GNNs.

2.3 Continual Graph Learning

Different continual graph learning settings are explored in recent studies, including Data Incremental Learning (DI) (Wang et al., 2020; Xu et al., 2020; Cai et al., 2022; Han et al., 2020; Wang et al., 2022), Task-Incremental Learning (TI) (Liu et al., 2021; Zhou & Cao, 2021; Zhang et al., 2021) and Class-Incremental Learning (CI)(Wang et al., 2022). DI is not a typical continual learning setting, but it is studied in the context of graph learning. In DI, all samples are streamed randomly, while in CI and TI, all samples from a group of classes are streamed before switching to the next group. TWP (Liu et al., 2021) is a regularization-based method without storing any data or prototypes while (Zhou & Cao, 2021), (Zhang et al., 2021) and (Wang et al., 2022) need a buffer to store either raw data or prototypes to prevent from forgetting.

However, existing works overlook an interesting and important fact that is unique in the context of continual graph learning compared to regular continual learning in image classification: the unavailability of previous training nodes and edges lead to limited neighborhood information during *inference*, in addition to the catastrophic forgetting. Our work differs from existing works in that 1. we consider this ignored fact and 2. we focus on CI setting without the use of a memory buffer to store raw data or prototypes.

3 Methodology

In this paper, we focus on node classification. In this section, we first introduce a practical yet ignored uniqueness of continual learning in the context of graph leaning. Then we discuss how it poses new challenges for continual graph learning during the inference stage. Taking this case into account, we further propose our ReGNN model that jointly solve the catastrophic forgetting problem and the new challenge.

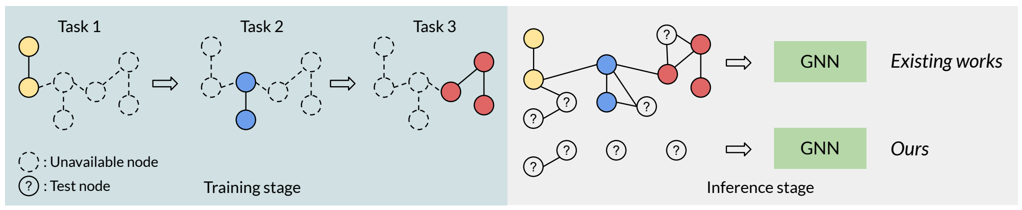

Figure 2: Different cases during the *inference* stage. Compared with existing works (Liu et al., 2021; Zhou & Cao, 2021; Wang et al., 2022), ours is the practical case for continual graph learning, where previous training nodes and edges are no longer available during current stage, which includes both the training stage and the *inference* stage.

3.1 Problem formulation

Data-Free Class-Incremental Graph Learning is defined as follows. Denote a graph as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$. A model learns from a sequence of data $D = \{\mathcal{D}^1, \mathcal{D}^2, ..., \mathcal{D}^m\}$, where $\mathcal{D}^i = \{\mathcal{V}^i, \mathcal{E}^i\}$.

Each $\mathcal{D}^i$ is the data distribution of the corresponding task $\mathcal{T}^i$, and the label space is $\mathcal{Y}^i$. During $\mathcal{T}^t$, only $\mathcal{D}^t$ is available and all data $\{\mathcal{D}^i | i < t\}$ is unavailable. The goal is to effectively learn from $\mathcal{D}^t$, while maintaining the model performance on learnt tasks. At the end of $\mathcal{T}^m$ (i.e., all m tasks are learnt), the model is required to map (test) samples from all seen data distributions to $\mathcal{Y}^1 \cup \mathcal{Y}^2 ... \cup \mathcal{Y}^m$ without task-IDs. Buffers to store raw data or prototypes are not allowed.

3.2 An ignored case during the inference stage

Different from typical image classification scenarios, where training data are not required for inference, in graph learning, they are often required for better inference when learning with large networks. This is because in image classification tasks, test samples are independent of training samples. However, in graph learning, test nodes are often dependent of training nodes. For instance, in many cases, test nodes are re-inserted into the training graph to exploit the dependency by neighbor aggregation for better inference. Thus in continual graph learning, the unavailability of previous data further poses a challenge during the inference stage in addition to the catastrophic forgetting problem.

However, existing works Wang et al. (2022); Liu et al. (2021); Zhou & Cao (2021) on TI and CI graph learning overlook this case. Fig 2 illustrates different inference cases in inductive node classification task. We do not consider transductive learning because it requires all test samples to appear in the graph for training from the beginning, which is not practical in CI graph learning, where streaming test samples often appear over time. The scenario adopted by existing works is not practical because in continual learning, data (i.e. nodes and edges) from previous tasks are no longer available once learned. But in existing works, test nodes with labels from all learnt classes are re-inserted into the graph and connected to previous nodes for inference.

Considering the unavailability of previous training data, we introduce ours as the practical inference scenario in continual graph learning, where previous training nodes are unavailable and only connections among test nodes are kept. In this case, as discussed in section 1, the connections among test nodes are very sparse. It leads to an unique challenge in graph continual learning: the neighborhood information available for inference is very poor.

3.3 ReGNN

To solve the major challenges in continual graph learning: (1) catastrophic forgetting and (2) limited neighborhood information during the inference stage, we propose our model ReplayGNN (ReGNN). Fig 4 illustrates the overview of ReGNN and Fig 3 illustrates how replay data is generated and used. The idea of our model is that: to address the first problem, a generative model (GraphCVAE) is maintained to generate old data for replay to prevent forgetting. To solve the second problem, the NodeAE module effectively learns from node attributes with little or no neighborhood information to adapt to our introduced inference case in Fig 2.

Note that We use a single-layer GNN in our model for simplicity. More advanced graph generation techniques can be integrated into our graph encoder and decoder if multiple layers are required. Experiments show our lightweight model is already effective. More details are discussed in Sec. 5

3.3.1 GraphCVAE Module

The Graph Conditional Variational Autoencoder (GraphCVAE) module consists of the graph decoder and graph encoder. During $\mathcal{T}^k$, GraphCVAE is trained with real data $\mathcal{D}^t$ and generated replay data $\mathcal{D}_g$, which is generated by the graph decoder D^{t-1}_{graph} from task $\mathcal{T}^{t-1}$. The generation is conditioned on the class label $\boldsymbol{c}$.

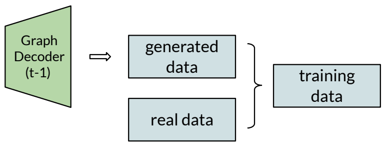

Figure 3: Composition of training data for current task t. It consists of replay data generated by the copy of the graph encoder from the last task $t-1$ and the training data from the current task t.

Graph Encoder. Given input nodes $\mathcal{D}^t$ with sampled neighbors, the graph encoder first obtains the node embeddings with the forward propagation steps. Denote the input node as

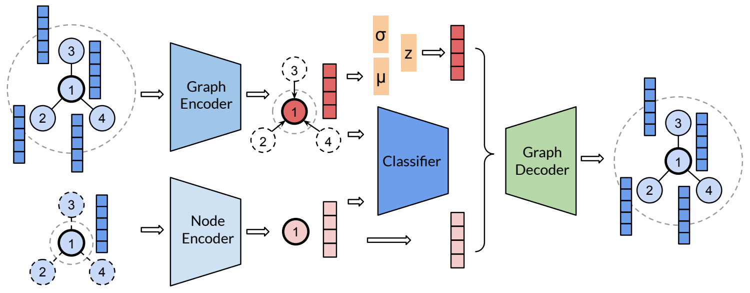

Figure 4: Overview of our ReplayGNN (ReGNN) model. ReGNN consists of three modules with shared components: a GraphCVAE, a classifier and a NodeAE. The input in the figure is a node ($node_1$) in a mini-batch with its sampled neighbors. The graph encoder consists of the convolution layer of GNN, which outputs low-dimensional embeddings (red) of $node_1$. The **GraphCVAE** module consists of the graph encoder and graph decoder. The **classifier** is a single classification layer. The **NodeAE**, which consists of the node encoder and the graph decoder (shared with GraphCVAE), takes a single node embedding as input and (1) tries to reconstruct the structure information and (2) yield node embedding (pink) that is distinguishable for classifier.

v and $\boldsymbol{h}_u^{in}$ as the raw feature of a node u. The output embedding $\boldsymbol{h}_v^{out}$ is calculated by:

$$\boldsymbol{h}_{\mathcal{N}(v)}^{out} \leftarrow \text{AGGREGATE}\left(\left\{\boldsymbol{h}_u^{in}, \forall u \in \mathcal{N}(v)\right\}\right), \tag{1}$$

$$\boldsymbol{h}_v^{out} \leftarrow \varphi\left(\boldsymbol{W} \cdot \text{CONCAT}\left(\boldsymbol{h}_v^{in}, \boldsymbol{h}_{\mathcal{N}(v)}^{out}\right)\right), \tag{2}$$

where $\boldsymbol{W}$ is the learnable parameter and φ is the activation function. Then the mean $\mathbf{z}_\mu$ and standard deviation $\mathbf{z}_\sigma$ of the distribution are calculated from the embedding $\boldsymbol{h}_v^{out}$ by:

$$\boldsymbol{z}_\mu = \text{ReLU}(\boldsymbol{W}_\mu \cdot \boldsymbol{h}_v^{out}),\ \boldsymbol{z}_\sigma = \text{ReLU}(\boldsymbol{W}_\sigma \cdot \boldsymbol{h}_v^{out}), \tag{3}$$

where Rectified Linear Unit (ReLU) (Nair & Hinton, 2010) is the activation function. $\boldsymbol{W}_\mu$ and $\boldsymbol{W}_\sigma$ are learnable parameters. With mean $\boldsymbol{z}_\mu$ and standard deviation $\boldsymbol{z}_\sigma$, latent variable $\mathbf{z}$ is sampled from $\mathcal{N}(\boldsymbol{z}_\mu,\ \boldsymbol{z}_\sigma)$.

Graph Decoder. The decoder tries to reconstruct the input data from the latent variable $\mathbf{z}$. It first generates embedding $\boldsymbol{f}_v$ from latent variable $\mathbf{z}$ with learnable parameter $\boldsymbol{B}$, then the input embedding is reconstructed:

$$\boldsymbol{f}_v = \varphi(\boldsymbol{B} \cdot \mathbf{z}). \tag{4}$$

$$[\boldsymbol{f}_v^{out}; \boldsymbol{f}_{\mathcal{N}(v)}^{out}] \leftarrow \varphi(\boldsymbol{M} \cdot \boldsymbol{f}_v), \tag{5}$$

where $\boldsymbol{M}$ is the learnable parameter and $[\mathbf{a}; \mathbf{b}]$ is the concatenation of embedding $\mathbf{a}$ and $\mathbf{b}$. The decoded embeddings $[\boldsymbol{f}_v^{out}; \boldsymbol{f}_{\mathcal{N}(v)}^{out}]$ try to match the input embedding $[\boldsymbol{h}_v^{in}, \boldsymbol{h}_{\mathcal{N}(v)}^{out}]$, which is the return value of the CONCAT function in Eq. (1). This embedding contains the neighbor information of v as well as the feature of the node v itself. We do ***not*** further decode it to obtain $\{\boldsymbol{h}_u^{in}, \forall u \in \mathcal{N}(v)\}$ because the current decoded value already can be the input of graph encoder and be helpful for replay. In other words, we replay the input embedding $[\boldsymbol{h}_v^{in}, \boldsymbol{h}_{\mathcal{N}(v)}^{out}]$ instead of $\{\boldsymbol{h}_u^{in}, \forall u \in \mathcal{N}(v)\}$. Then the loss functions in GraphCVAE are:

$$\mathcal{L}^{Recon} = \text{DIST}([\boldsymbol{f}_v^{out}; \boldsymbol{f}_{\mathcal{N}(v)}^{out}], [\boldsymbol{h}_v^{in}; \boldsymbol{h}_{\mathcal{N}(v)}^{out})]),\ \mathcal{L}^{KLD} = D_{KL}[\mathcal{N}(\boldsymbol{z}_\mu, \boldsymbol{z}_\sigma)||\mathcal{N}(\boldsymbol{c}, 1)], \tag{6}$$

where DIST measures the distance of the two embeddings. and D_{KL} is the Kullback–Leibler divergence. $\boldsymbol{c}$ is the class label of input data. The overall loss function then is:

$$\mathcal{L}^{cvae} = \mathcal{L}^{Recon} + \mathcal{L}^{KLD}. \tag{7}$$

3.4 NodeAE Module

Node Autoencoder (NodeAE) consists of the node encoder and the graph decoder. During the training, given a node v, as graphSAGE aggregates the neighbors of v to obtain its representation, the classifier learns to classify $\boldsymbol{h}_v^{out}$ in (1), i.e., $\varphi(\boldsymbol{W} \cdot [\boldsymbol{h}_v^{in}; \boldsymbol{h}_{\mathcal{N}(v)}^{out}])$. However, during the inference stage, the neighbors of v are often unavailable as illustrated in Figure 2. Thus, for most test nodes v, $\boldsymbol{h}_v^{out} = \varphi(\boldsymbol{W} \cdot [\boldsymbol{h}_v^{in}; \mathbf{0}])$ during inference, where $\mathbf{0}$ means this test node has no available neighbor test nodes. In this case, the classifier fails to fully exploit its learnt knowledge on training graph for inference because it only learns to classify nodes with rich neighbors. Thus given an isolated node v, the feature, which is sent to classifier for classification, is also expected to contain the neighbor information of the node. In this way, the node can be better classified by the classifier. We use an autoencoder to achieve it.

Given a node v, while the graph encoder takes node v and its sampled neighbors as input, node encoder ***only*** takes the node attribute of node v. But the decoder is required to reconstruct the embeddings of the aggregation of its neighbors as well as its own embedding. The forward propagation of the encoder is:

$$\boldsymbol{x}_v^{out} \leftarrow \varphi(\boldsymbol{H} \cdot [\boldsymbol{h}_v^{in}; \mathbf{0}]), \tag{8}$$

where $\boldsymbol{H}$ is the learnable parameter, whose size is the same as $\boldsymbol{W}$. The loss function is:

$$\mathcal{L}^{ae} = \text{DIST}(\varphi(\boldsymbol{M} \cdot \boldsymbol{x}_v^{out}), [\boldsymbol{h}_v^{in}; \boldsymbol{h}_{\mathcal{N}(v)}^{out}]). \tag{9}$$

When trained properly, given an isolated node, the node encoder can map the node attribute into a compressed embedding $\boldsymbol{x}_v^{out}$ that contains its neighbor information and is then used for prediction.

3.5 Classifier

The classifier is a single classification layer and takes the output of the graph encoder and the node encoder as input. The loss function for classification is:

$$\mathcal{L}^{cls} = L_{ce}(g(\mathbf{x}_v^{out}, \boldsymbol{h}_v^{out}), \boldsymbol{y}), \tag{10}$$

where L_{ce} is the cross entropy loss function, and $g(\cdot)$ is a function that combines two embeddings, and we simply add them together. Note that the current training data are a mixture of real data and replay data as Figure 3 illustrates. $\boldsymbol{y}$ are their corresponding labels.

By summarizing the three modules together, the overall loss function for our ReGNN model is:

$$\mathcal{L}^{ReGNN} = \mathcal{L}^{cls} + \mathcal{L}^{cvae} + \mathcal{L}^{ae}. \tag{11}$$

4 Experiments

4.1 Datasets

To evaluate the performance of ReGNN in our continual graph learning scenario, we conduct experiments on three benchmark datasets that are also used to build datasets for CI or TI settings in related works (Wang et al., 2022; Liu et al., 2021; Zhou & Cao, 2021): **Cora** (Sen et al., 2008), **CiteSeer** (Sen et al., 2008), and we use **Amazon** (McAuley et al., 2015) to refer to AmazonCoBuyPhoto, a segment of Amazon co-purchasing graph. We split each dataset into several tasks, following Liu et al. (2021); Zhou & Cao (2021); Wang et al. (2022). Details are provided in Appendix A.

4.2 Baselines

We adopt the following baselines in our experiments. **GraphSAGE** (Hamilton et al., 2017) is a representative GNN model, which is also used as the backbone for some other baselines for fair comparison. **LwF** (Li & Hoiem, 2017) is a representative data-free continual learning model, which uses the history state of itself as the teacher for knowledge distillation to avoid forgetting. **ER-GNN** (Zhou & Cao, 2021) uses a buffer to store data for experience replay, in order to prevent from forgetting. **ContinualGNN** (Wang et al., 2020) alleviates forgetting by storing data for replay and adopts a model regularization similar to EWC (Kirkpatrick et al., 2017). Although in data-free continual learning setting, no data can be stored for replay, we show that ReGNN outperforms ER-GNN (Zhou & Cao, 2021) and ContinualGNN(Wang et al., 2020), which use memory buffers for replay.

Table 1: Accuracy on different datasets under class-incremental learning scenario.

	Scenario		Dataset		
Model	Data-free	Memory size	Citeseer	Cora	Amazon
ER-GNN (Zhou & Cao, 2021)	✗	M = 1%	34.13 ± 0.14	41.78 ± 0.62	50.53 ± 2.94
		M = 3%	41.46 ± 2.32	51.96 ± 1.65	60.53 ± 2.85
		M = 5%	45.48 ± 2.35	57.96 ± 2.58	67.82 ± 2.23
ContinualGNN (Wang et al., 2020)	✗	M = 1%	35.94 ± 0.28	38.48 ± 0.17	49.53 ± 1.74
		M = 3%	46.08 ± 0.45	46.49 ± 0.51	56.78 ± 2.04
		M = 5%	50.60 ± 0.49	60.41 ± 1.91	60.79 ± 2.23
GraphSAGE (Hamilton et al., 2017)	✓	-	31.02 ± 0.12	25.00 ± 0.14	37.01 ± 0.43
LwF (Li & Hoiem, 2017)	✓	-	36.74 ± 1.15	42.27 ± 1.54	50.01 ± 1.83
ReGNN (Ours)	✓	-	**55.02** ± 2.01	**64.33** ± 1.82	**69.32** ± 1.48

4.3 Experimental Setup

We use the neighborhood sampling strategy in GraphSAGE for all models when sampling is available. SGD optimizer is used and the initial learning rate is set to 0.01 for Cora and Citeseer and 0.005 for Amazon. The batch size it set to 128 and run 100 epochs for each dataset. For baselines, the number of layers in GNN is set to the default value according to their papers. A standard two layer GNN is used if the information is not provided. We run each experiment five times and report the results with mean and standard deviation.

4.4 Results and Discussions

We design experiments to answer the following questions: **Q1**: Can ReGNN outperform related approaches in our class-incremental graph learning setting? **Q2**: How does each module contribute to the performance of ReGNN? **Q3**: How does ReGNN perform when the neighborhood information is given at different levels during inference? **Q4**: Are the generative replay data really effective in preserving neighbor information?

Q1. Can ReGNN outperform other approaches in our class-incremental graph learning setting? Table 1 shows the main results of our model and baselines. Our ReGNN outperforms LwF (Li & Hoiem, 2017), which does not require a memory buffer, by a large margin. Experience Replay, a typical memory based method is a competitive baseline because it is able to reach old data during training by maintaining a memory buffer, and it often notably outperforms the regularization based methods in class-incremental learning scenario (Buzzega et al., 2020; 2021) with a small buffer size (e.g., 2% in (Buzzega et al., 2021)). The shortage of memory based methods is that they need a memory buffer to store data, which is not always feasible. We experiment with memory size M = $\{1\%, 3\%, 5\%\}$ for memory based models. Experiment results show the effectiveness of ReGNN in class-incremental setting. Our model still outperforms ER-GNN (Zhou & Cao, 2021) and ContinualGNN (Wang et al., 2020) when they have a buffer with memory size M =5%, which is already a relatively large buffer in CL.

Q2. How does each module contribute to the performance of ReGNN? Table 2 shows the ablation study results on Citeseer. Full method refers to our ReGNN. Note that when ablating NodeAE, only the node encoder is removed. A clear performance drop is observed after removing NodeAE. It manifests the effectiveness of learning to reconstruct the neighbors from single node attribute embeddings with node AE module.

Table 2: Ablation study results.

Method	Accuracy
Full Method	**55.02 $\pm$ 2.01**
Ablate NodeAE	52.40 ± 2.14
Use separate GraphCVAE	51.21 ± 1.79
Ablate GraphCVAE (No replay)	31.02 ± 0.12

To simplify the model structure, we use the graph convolution layer in GNN as the graph encoder for the GraphCVAE. We are curious about whether sharing parameters can further improve the performance. To study this point, we train a separate GraphCVAE that owns an separate graph encoder. In this way, the GraphCVAE becomes a separate model without any shared parameters

Table 3: Accuracy on Citeseer dataset at different levels of neighborhood information. Best results in bold.

Model	Default	3-hop	5-hop	Full
ER-GNN (M=1%)	34.13 ± 0.14	36.44 ± 0.42	36.64 ± 0.25	38.65 ± 0.93
ER-GNN (M=3%)	41.46 ± 2.32	49.29 ± 2.76	50.20 ± 2.23	58.82 ± 3.32
ER-GNN (M=5%)	45.48 ± 2.35	55.82 ± 0.56	57.53 ± 1.27	61.44 ± 2.18
ContinualGNN (M=1%)	35.94 ± 0.28	40.06 ± 0.85	41.67 ± 0.99	41.46 ± 1.50
ContinualGNN (M=3%)	46.08 ± 0.45	52.71 ± 1.22	53.81 ± 1.44	55.02 ± 1.26
ContinualGNN (M=5%)	50.60 ± 0.49	57.22 ± 1.22	58.63 ± 1.35	60.14 ± 1.35
GraphSAGE	31.02 ± 0.12	31.02 ± 0.19	30.72 ± 0.39	31.92 ± 0.46
LwF	36.74 ± 1.15	40.65 ± 1.06	42.85 ± 1.34	48.97 ± 1.03
ReGNN (Ours)	$\mathbf{55.02} \pm 2.01$	$\mathbf{62.57} \pm 2.12$	$\mathbf{64.57} \pm 2.34$	$\mathbf{67.74} \pm 2.12$

with others. It follows the representative generative replay based framework (Shin et al., 2017), where an separate generative model is trained for replay without any interactions, such as parameter sharing, with the classification module.

Through the ablation experiment, We find that sharing part of the parameters (i.e., the graph encoder) not only simplifies the model structure, but also improves the performance of the model. It indicates proper parameter sharing can yield better results than using two separate modules in some cases. Finally we ablate the GraphCVAE module to stop generative replay. The forgetting phenomenon becomes obvious.

Q3. How does ReGNN perform when the neighborhood information is given at different levels during inference? Table 1 shows the effectiveness of ReGNN in our practical scenario for continual graph learning illustrated in Figure 1 and Figure 2, where the neighborhood information of test nodes is very poor or none. A natural question is, whether ReGNN can also show good performance in different cases, where the neighborhood information is given at different levels. This is practical when additional information is available to help build richer connections between test nodes for more neighborhood information and better inference. We further investigate the performance of ReGNN in different cases.

Because we randomly split the data into training and test sets and detach the test nodes from the *original graph*, we cannot control the level of neighborhood information among test nodes with the random partition process. Instead, we change the level of neighborhood information by manually linking test nodes according to their distance to each other in the *original graph*.

Specifically, given a pair of nodes (v, u) from detached test nodes, we denote the distance from v to u as $Dist(v, u)$, which is calculated by the number of hops from v to u in the *original graph*. *Original graph* refers to the complete graph before training/test partition. In this way, we can increase the neighborhood information of test nodes by manually linking pairs of test nodes. For instance, (v, u) can be linked if $Dist(v, u)$ is smaller than a certain threshold k. After linking all node pairs $\{(v, u)|Dist(v, u) < k, v \in \mathcal{V}_{test}, u \in \mathcal{V}_{test}\}$ in the test nodes, we denote the evaluation on the test data with this additional neighborhood information as *k-hop* evaluation. Note that the way we increase neighborhood information can involve noise if k is set as a very large number, which leads to a performance drop. In our experiments, we choose proper k that is smaller than a threshold, values larger than which fails to further improve the performance.

Experiments show that the manually added links are helpful for node classification. Because in *k-hop* evaluations, we manually select a proper k and add these links according to the original complete graph, an interesting topic is to train an additional task, for instance, link prediction, to predict these links and then link them for better node classification. We leave it for future work. In this work, we just manually add these links and focus on analyzing the ReGNN model with $k - hop$ evaluation. Table 3 shows the results of this study. *Default* means no additional links are added. k refers to *k-hop* evaluation, and $Full$ is the evaluation method adopted by existing works in Figure 2, i.e., linking the detached test nodes back to the graph from which they are detached for inference. Intuitively, $Full$ should yield best results. Compared to the results of the *default* evaluation, increasing neighborhood information can indeed improve our model performance. ReGNN constantly outperforms other baselines under different settings, which manifests the reliability and effectiveness of our model

Table 4: Results of replay based models on Citeseer when neighbors of the stored old nodes are not available in the memory buffer.

Model	**Default**	**3-hop**	**5-hop**	**Full**
GraphSAGE (Hamilton et al., 2017)	31.02	31.02	30.72	31.92
ContinualGNN (Wang et al., 2020)	41.86	45.28	47.89	47.59
ER-GNN (Zhou & Cao, 2021)	43.37	41.56	40.69	42.21
ReGNN (Ours)	55.02	62.57	64.57	67.74

and indicates that our model can work well in different cases with either rich or poor neighborhood information.

Q4. Are the generative replay data really effective in preserving neighbor information? Because each node is associated with a node attribute, simply replaying and using node attributes can also make predictions. Although ReGNN outperforms other baselines, we are curious to know whether generated replay data can really help ReGNN remember the neighbor information and effectively exploit them for inference.

We start with analyzing how ER-GNN remembers the neighborhood information. ER-GNN requires a memory buffer to store old training nodes from previous tasks for replay to prevent forgetting. When replaying stored old nodes from the buffer, the ***neighbor nodes*** of the old nodes need to be sampled via the sampling strategy in graphSAGE for aggregation to compute the embeddings of stored old nodes. Thus in earlier experiments, we ***allow*** ER-GNN to store the neighbors of the stored old nodes to fulfill the model's potential. In this case, the memory size of the buffet is in fact larger than our reported 1% 3% 5%, because their neighbors are also stored for the neighbor aggregation process in GraphSAGE. In this way, when ER-GNN uses old data for training, the neighborhood information of the stored old nodes is also replayed. Thus ER-GNN can remember and exploit the neighbor information during inference, which is proved by results in Table 3: as $k - hop$ increases, ER-GNN achieves better performance by using the richer neighbor information.

Here we experiment with another version of ER-GNN and ContinualGNN, where the neighbors of stored old nodes are ***not*** available, which means only their own node attributes are used for replay. Table 4 shows, in this case, ContinualGNN (Wang et al., 2020) still has improvements as k increases because it also uses model regularization besides data replay. However, the performance of ER-GNN, which fully relies on data replay, fails to be improved as k increases (i.e., richer neighborhood information is provided), which means ER-GNN cannot exploit neighborhood information anymore. This is because only isolated node attributes are replayed without neighbor aggregation thus model forgets how to exploit given neighbors.

Our model performance consistently grows with richer neighborhood information, indicating ReGNN remembers how to exploit given neighbors for better inference by replaying our generated data. It further manifests our generated data contains helpful neighborhood information, which prevents ReGNN from forgetting it during generative replay.

5 Conclusion and future work

In this paper, we find an important and practical case yet ignored by existing works in continual graph learning, which leads to poor neighborhood information during inference, in additional to catastrophic forgetting. We further propose ReGNN to jointly solve the challenges, whose effectiveness is proved by experiments.

More advanced graph generation techniques can be integrated into our graph encoder and decoder module for better generation with multi-layer GNNs. We leave this for future work because generating large graphs with different node attributes is rarely studied by existing works, most of which focus on molecular graph generation. It is another under-explored topic and not our major focus but could be interesting if combined. $k - hop$ evaluations have better performance but we manually set k and build links with ground truth. It is also interesting to automate this process and let model infer the connections by itself to improve model performance.

REFERENCES

Rahaf Aljundi, Francesca Babiloni, Mohamed Elhoseiny, Marcus Rohrbach, and Tinne Tuytelaars. Memory aware synapses: Learning what (not) to forget. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 139–154, 2018.

Pietro Buzzega, Matteo Boschini, Angelo Porrello, Davide Abati, and Simone Calderara. Dark experience for general continual learning: a strong, simple baseline. *Advances in neural information processing systems*, 33:15920–15930, 2020.

Pietro Buzzega, Matteo Boschini, Angelo Porrello, and Simone Calderara. Rethinking experience replay: a bag of tricks for continual learning. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pp. 2180–2187. IEEE, 2021.

Jie Cai, Xin Wang, Chaoyu Guan, Yateng Tang, Jin Xu, Bin Zhong, and Wenwu Zhu. Multimodal continual graph learning with neural architecture search. In *Proceedings of the ACM Web Conference 2022*, pp. 1292–1300, 2022.

Arslan Chaudhry, Marcus Rohrbach, Mohamed Elhoseiny, Thalaiyasingam Ajanthan, Puneet K Dokania, Philip HS Torr, and Marc'Aurelio Ranzato. On tiny episodic memories in continual learning. *arXiv preprint arXiv:1902.10486*, 2019.

Jie Chen, Tengfei Ma, and Cao Xiao. Fastgcn: fast learning with graph convolutional networks via importance sampling. *arXiv preprint arXiv:1801.10247*, 2018.

Sayna Ebrahimi, Franziska Meier, Roberto Calandra, Trevor Darrell, and Marcus Rohrbach. Adversarial continual learning. In *European Conference on Computer Vision*, pp. 386–402. Springer, 2020.

Lukas Galke, Benedikt Franke, Tobias Zielke, and Ansgar Scherp. Lifelong learning of graph neural networks for open-world node classification. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2021.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Yi Han, Shanika Karunasekera, and Christopher Leckie. Graph neural networks with continual learning for fake news detection from social media. *arXiv preprint arXiv:2007.03316*, 2020.

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

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526, 2017.

Zhizhong Li and Derek Hoiem. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947, 2017.

Huihui Liu, Yiding Yang, and Xinchao Wang. Overcoming catastrophic forgetting in graph neural networks. *AAAI*, 2021.

Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton Van Den Hengel. Image-based recommendations on styles and substitutes. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval*, pp. 43–52, 2015.

Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pp. 109–165. Elsevier, 1989.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Icml*, 2010.

Sylvestre-Alvise Rebuffi, Alexander Kolesnikov, Georg Sperl, and Christoph H Lampert. icarl: Incremental classifier and representation learning. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 2001–2010, 2017.

David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. *Advances in Neural Information Processing Systems*, 32, 2019.

Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3):93–93, 2008.

Hanul Shin, Jung Kwon Lee, Jaehong Kim, and Jiwon Kim. Continual learning with deep generative replay. *Advances in neural information processing systems*, 30, 2017.

Sebastian Thrun. Is learning the n-th thing any easier than learning the first? *Advances in neural information processing systems*, 8, 1995.

Gido M Van de Ven and Andreas S Tolias. Three scenarios for continual learning. *arXiv preprint arXiv:1904.07734*, 2019.

Chen Wang, Yuheng Qiu, Dasong Gao, and Sebastian Scherer. Lifelong graph learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 13719–13728, 2022.

Junshan Wang, Guojie Song, Yi Wu, and Liang Wang. Streaming graph neural networks via continual learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pp. 1515–1524, 2020.

Yishi Xu, Yingxue Zhang, Wei Guo, Huifeng Guo, Ruiming Tang, and Mark Coates. Graphsail: Graph structure aware incremental learning for recommender systems. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, pp. 2861–2868, 2020.

Jaehong Yoon, Eunho Yang, Jeongtae Lee, and Sung Ju Hwang. Lifelong learning with dynamically expandable networks. *arXiv preprint arXiv:1708.01547*, 2017.

Xikun Zhang, Dongjin Song, and Dacheng Tao. Hierarchical prototype networks for continual graph representation learning. *arXiv preprint arXiv:2111.15422*, 2021.

Fan Zhou and Chengtai Cao. Overcoming catastrophic forgetting in graph neural networks with experience replay. *AAAI*, 2021.

A DATASET

To evaluate the performance of ReGNN in our continual graph learning scenario, we conduct experiments on three benchmark datasets that are also used to build datasets for CI or TI settings in related works (Wang et al., 2022; Liu et al., 2021; Zhou & Cao, 2021).

Cora (Sen et al., 2008) is a citation network consisting of and 5429 links and 2708 nodes classified into one of seven classes. Each node has a vector of size 1433 as its attribute. **CiteSeer** (Sen et al., 2008) consists of 3312 publications classified into one of six classes. The citation network consists of 4732 links. Each node has a vector of size 3073 as its attribute. We use **Amazon** (McAuley et al., 2015) to refer to AmazonCoBuyPhoto, a segment of Amazon co-purchasing graph. It consists of 119,043 links with 7,650 nodes of 8 classes.

To construct datasets for class-incremental graph learning, we follow (Liu et al., 2021; Zhou & Cao, 2021; Wang et al., 2022) and divide each dataset into several tasks, where each task contains several non-overlapping classes.

For Cora, we divide it into three tasks. The first and second tasks consist of two classes, and the last task contains three classes. Citeseer are split into three tasks with two classes in each task. The first task of Amazon contains two classes. The second and third tasks on Amazon contain three classes.