EvolveGCN: Evolving Graph Convolutional Networks for Dynamic Graphs

Aldo Pareja,[1,2*] **Giacomo Domeniconi,**[1,2*] **Jie Chen,**[1,2†] **Tengfei Ma,**[1,2] **Toyotaro Suzumura,**[1,2]
Hiroki Kanezashi,[1,2] **Tim Kaler,**[1,3] **Tao B. Schardl,**[1,3] **Charles E. Leiserson**[1,3]
[1]MIT-IBM Watson AI Lab, [2]IBM Research, [3]MIT CSAIL
{Aldo.Pareja, Giacomo.Domeniconi1}@ibm.com, chenjie@us.ibm.com
Tengfei.Ma1@ibm.com, {tsuzumura, hirokik}@us.ibm.com, {tfk, neboat, cel}@mit.edu

Abstract

Graph representation learning resurges as a trending research subject owing to the widespread use of deep learning for Euclidean data, which inspire various creative designs of neural networks in the non-Euclidean domain, particularly graphs. With the success of these graph neural networks (GNN) in the static setting, we approach further practical scenarios where the graph dynamically evolves. Existing approaches typically resort to node embeddings and use a recurrent neural network (RNN, broadly speaking) to regulate the embeddings and learn the temporal dynamics. These methods require the knowledge of a node in the full time span (including both training and testing) and are less applicable to the frequent change of the node set. In some extreme scenarios, the node sets at different time steps may completely differ. To resolve this challenge, we propose EvolveGCN, which adapts the graph convolutional network (GCN) model along the temporal dimension without resorting to node embeddings. The proposed approach captures the dynamism of the graph sequence through using an RNN to evolve the GCN parameters. Two architectures are considered for the parameter evolution. We evaluate the proposed approach on tasks including link prediction, edge classification, and node classification. The experimental results indicate a generally higher performance of EvolveGCN compared with related approaches. The code is available at https://github.com/IBM/EvolveGCN.

1 Introduction

Graphs are ubiquitous data structures that model the pairwise interactions between entities. Learning with graphs encounters unique challenges, including their combinatorial nature and the scalability bottleneck, compared with Euclidean data (e.g., images, videos, speech signals, and natural languages). With the remarkable success of deep learning for the latter data types, there exist renewed interests in the learning of graph representations (Perozzi, Al-Rfou, and Skiena 2014; Tang et al. 2015; Cao, Lu, and Xu 2015; Ou et al. 2016; Grover and Leskovec 2016) on both the node and the graph level, now parameterized by deep neural networks (Bruna et al. 2014; Duvenaud et al. 2015; Defferrard, Bresson, and Vandergheynst 2016; Li et al. 2016; Gilmer et al. 2017; Kipf and Welling 2017; Hamilton, Ying, and Leskovec 2017; Jin et al. 2017; Chen, Ma, and Xiao 2018; Veličković et al. 2018; Gao and Ji 2019).

These neural network models generally focus on a given, static graph. In real-life applications, however, often one encounters a dynamically evolving graph. For example, users of a social network develop friendship over time; hence, the vectorial representation of the users should be updated accordingly to reflect the temporal evolution of their social relationship. Similarly, a citation network of scientific articles is constantly enriched due to frequent publications of new work citing prior art. Thus, the influence, and even sometimes the categorization, of an article varies along time. Update of the node embeddings to reflect this variation is desired. In financial networks, transactions naturally come with time stamps. The nature of a user account may change owing to the characteristics of the involved transactions (e.g., an account participates money laundering or a user becomes a victim of credit card fraud). Early detection of the change is crucial to the effectiveness of law enforcement and the minimization of loss to a financial institute. These examples urge the development of dynamic graph methods that encode the temporal evolution of relational data.

Built on the recent success of graph neural networks (GNN) for static graphs, in this work we extend them to the dynamic setting through introducing a recurrent mechanism to update the network parameters, for capturing the dynamism of the graphs. A plethora of GNNs perform information fusion through aggregating node embeddings from one-hop neighborhoods recursively. A majority of the parameters of the networks is the linear transformation of the node embeddings in each layer. We specifically focus on the graph convolutional network (GCN) (Kipf and Welling 2017) because of its simplicity and effectiveness. Then, we propose to use a recurrent neural network (RNN) to inject the dynamism into the parameters of the GCN, which forms an evolving sequence.

Work along a similar direction includes (Seo et al. 2016; Manessia, Rozza, and Manzo 2017; Narayan and Roe 2018), among others, which are based on a combination of GNNs (typically GCN) and RNNs (typically LSTM). These meth-

[*] Equal contribution

[†] Contact author

ods use GNNs as a feature extractor and RNNs for sequence learning from the extracted features (node embeddings). As a result, one single GNN model is learned for all graphs on the temporal axis. A limitation of these methods is that they require the knowledge of the nodes over the whole time span and can hardly promise the performance on new nodes in the future.

In practice, in addition to the likelihood that new nodes may emerge after training, nodes may also frequently appear and disappear, which renders the node embedding approaches questionable, because it is challenging for RNNs to learn these irregular behaviors. To resolve these challenges, we propose instead to use the RNN to regulate the GCN model (i.e., network parameters) at every time step. This approach effectively performs model adaptation, which focuses on the model itself rather than the node embeddings. Hence, change of nodes poses no restriction. Further, for future graphs with new nodes without historical information, the evolved GCN is still sensible for them.

Note that in the proposed method, the GCN parameters are not trained anymore. They are computed from the RNN and hence only the RNN parameters are trained. In this manner, the number of parameters (model size) does not grow with the number of time steps and the model is as manageable as a typical RNN.

2 Related Work

Methods for dynamic graphs are often extensions of those for a static one, with an additional focus on the temporal dimension and update schemes. For example, in matrix factorization-based approaches (Roweis and Saul 2000; Belkin and Niyogi 2002), node embeddings come from the (generalized) eigenvectors of the graph Laplacian matrix. Hence, DANE (Li et al. 2017) updates the eigenvectors efficiently based on the prior ones, rather than computing them from scratch for each new graph. The dominant advantage of such methods is the computational efficiency.

For random walk-based approaches (Perozzi, Al-Rfou, and Skiena 2014; Grover and Leskovec 2016), transition probabilities conditioned on history are modeled as the normalized inner products of the corresponding node embeddings. These approaches maximize the probabilities of the sampled random walks. CTDANE (Nguyen et al. 2018) extends this idea by requiring the walks to obey the temporal order. Another work, NetWalk (Yu et al. 2018), does not use the probability as the objective function; rather, it observes that if the graph does not undergo substantial changes, one only needs to resample a few walks in the successive time step. Hence, this approach incrementally retrains the model with warm starts, substantially reducing the computational cost.

The wave of deep learning introduces a flourish of unsupervised and supervised approaches for parameterizing the quantities of interest with neural networks. DynGEM (Goyal et al. 2017) is an autoencoding approach that minimizes the reconstruction loss, together with the distance between connected nodes in the embedding space. A feature of DynGEM is that the depth of the architecture is adaptive to the size of the graph; and the autoencoder learned from the past time step is used to initialize the training of the one in the following time.

A popular category of approaches for dynamic graphs is point processes that are continuous in time. Know-Evolve (Trivedi et al. 2017) and DyRep (Trivedi et al. 2018) model the occurrence of an edge as a point process and parameterize the intensity function by using a neural network, taking node embeddings as the input. DynamicTriad (Zhou et al. 2018) uses a point process to model a more complex phenomenon—triadic closure—where a triad with three nodes is developed from an open one (a pair of nodes are not connected) to a closed one (all three pairs are connected). HTNE (Zuo et al. 2018) similarly models the dynamism by using the Hawkes process, with additionally an attention mechanism to determine the influence of historical neighbors on the current neighbors of a node. These methods are advantageous for event time prediction because of the continuous nature of the process.

A set of approaches most relevant to this work is combinations of GNNs and recurrent architectures (e.g., LSTM), whereby the former digest graph information and the latter handle dynamism. The most explored GNNs in this context are of the convolutional style and we call them graph convolutional networks (GCN), following the terminology of the related work, although in other settings GCN specifically refers to the architecture proposed by (Kipf and Welling 2017). GCRN (Seo et al. 2016) offers two combinations. The first one uses a GCN to obtain node embeddings, which are then fed into the LSTM that learns the dynamism. The second one is a modified LSTM that takes node features as input but replaces the fully connected layers therein by graph convolutions. The first idea is similarly explored in WD-GCN/CD-GCN (Manessia, Rozza, and Manzo 2017) and RgCNN (Narayan and Roe 2018). WD-GCN/CD-GCN modifies the graph convolution layers, most notably by adding a skip connection. In addition to such simple combinations, STGCN (Yu, Yin, and Zhu 2018) proposes a complex architecture that consists of so-called ST-Conv blocks. In this model, the node features must be evolving over time, since inside each ST-Conv block, a 1D convolution of the node features is first performed along the temporal dimension, followed by a graph convolution and another 1D convolution. This architecture was demonstrated for spatiotemporal traffic data (hence the names STGCN and ST-Conv), where the spatial information is handled by using graph convolutions.

3 Method

In this section we present a novel method, coined *evolving graph convolutional network* (EvolveGCN), that captures the dynamism underlying a graph sequence by using a recurrent model to evolve the GCN parameters. Throughout we will use subscript t to denote the time index and superscript l to denote the GCN layer index. To avoid notational cluttering, we assume that all graphs have n nodes; although we reiterate that the node sets, as well as the cardinality, may change over time. Then, at time step t, the input data consists of the pair $(A_t \in \mathbb{R}^{n\times n}, X_t \in \mathbb{R}^{n\times d})$, where the former is the graph (weighted) adjacency matrix and the latter is the

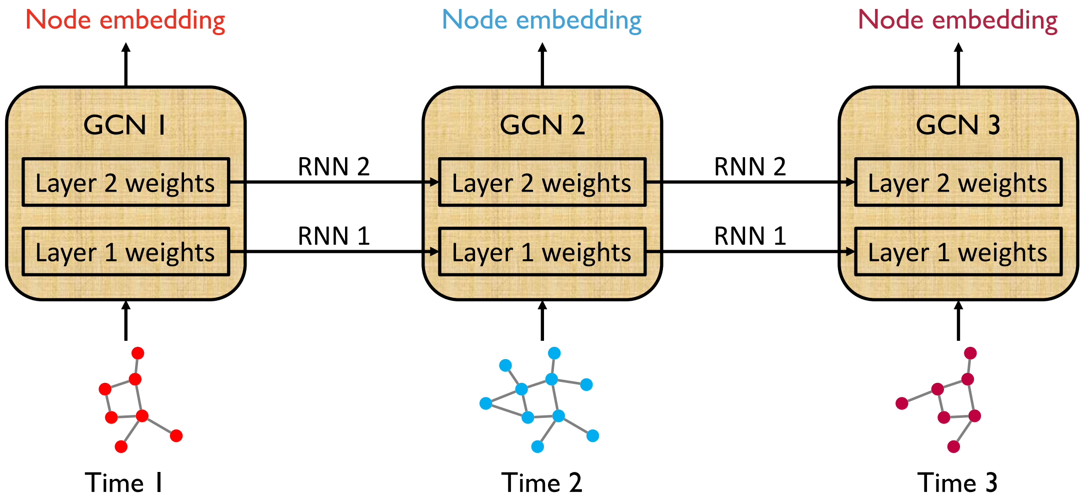


Figure 1: Schematic illustration of EvolveGCN. The RNN means a recurrent architecture in general (e.g., GRU, LSTM). We suggest two options to evolve the GCN weights, treating them with different roles in the RNN. See the EvolveGCN-H version and EvolveGCN-O version in Figure 2.

matrix of input node features. Specifically, each row of X_t is a d-dimensional feature vector of the corresponding node.

3.1 Graph Convolutional Network (GCN)

A GCN (Kipf and Welling 2017) consists of multiple layers of graph convolution, which is similar to a perceptron but additionally has a neighborhood aggregation step motivated by spectral convolution. At time t, the l-th layer takes the adjacency matrix A_t and the node embedding matrix $H_t^{(l)}$ as input, and uses a weight matrix $W_t^{(l)}$ to update the node embedding matrix to $H_t^{(l+1)}$ as output. Mathematically, we write

$$\begin{aligned} H_t^{(l+1)} &= \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)}) \\ &:= \sigma(\widehat{A}_t H_t^{(l)} W_t^{(l)}), \end{aligned} \tag{1}$$

where $\widehat{A}_t$ is a normalization of A_t defined as (omitting time index for clarity):

$$\widehat{A} = \widetilde{D}^{-\frac{1}{2}} \widetilde{A} \widetilde{D}^{-\frac{1}{2}}, \quad \widetilde{A} = A + I, \quad \widetilde{D} = \text{diag}\left(\sum_j \widetilde{A}_{ij} \right),$$

and σ is the activation function (typically ReLU) for all but the output layer. The initial embedding matrix comes from the node features; i.e., $H_t^{(0)} = X_t$. Let there be L layers of graph convolutions. For the output layer, the function σ may be considered the identity, in which case $H_t^{(L)}$ contains high-level representations of the graph nodes transformed from the initial features; or it may be the softmax for node classification, in which case $H_t^{(L)}$ consists of prediction probabilities.

Figure 1 is a schematic illustration of the proposed EvolveGCN, wherein each time step contains one GCN indexed by time. The parameters of the GCN are the weight matrices $W_t^{(l)}$, for different time steps t and layers l. Graph convolutions occur for a particular time but generate new information along the layers. Figure 2 illustrates the computation at each layer. The relationship between $H_t^{(l)}$, $W_t^{(l)}$, and $H_t^{(l+1)}$ is depicted in the middle part of the figure.

3.2 Weight Evolution

At the heart of the proposed method is the update of the weight matrix $W_t^{(l)}$ at time t based on current, as well as historical, information. This requirement can be naturally fulfilled by using a recurrent architecture, with two options.

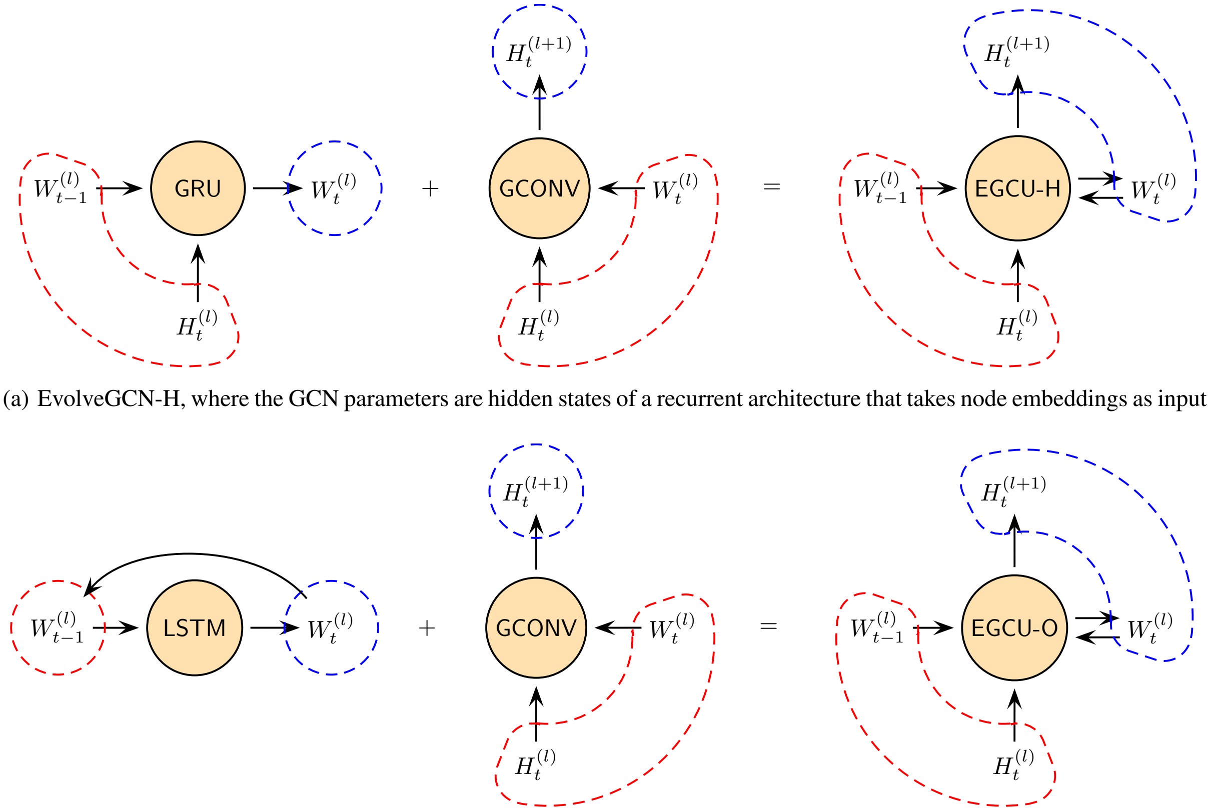
The first option is to treat $W_t^{(l)}$ as the hidden state of the dynamical system. We use a gated recurrent unit (GRU) to update the hidden state upon time-t input to the system. The input information naturally is the node embeddings $H_t^{(l)}$. Abstractly, we write

$$\underbrace{\overbrace{W_t^{(l)}}^{\text{GCN weights}}}_{\text{hidden state}} = \text{GRU}(\underbrace{\overbrace{H_t^{(l)}}^{\text{node embeddings}}}_{\text{input}}, \underbrace{\overbrace{W_{t-1}^{(l)}}^{\text{GCN weights}}}_{\text{hidden state}}),$$

with details deferred to a later subsection. The GRU may be replaced by other recurrent architectures, as long as the roles of $W_t^{(l)}$, $H_t^{(l)}$, and $W_{t-1}^{(l)}$ are clear. We use "-H" to denote this version; see the left part of Figure 2(a).

The second option is to treat $W_t^{(l)}$ as the output of the dynamical system (which becomes the input at the subsequent time step). We use a long short-term memory (LSTM) cell to model this input-output relationship. The LSTM itself maintains the system information by using a cell context, which acts like the hidden state of a GRU. In this version, node embeddings are not used at all. Abstractly, we write

$$\underbrace{\overbrace{W_t^{(l)}}^{\text{GCN weights}}}_{\text{output}} = \text{LSTM}(\underbrace{\overbrace{W_{t-1}^{(l)}}^{\text{GCN weights}}}_{\text{input}}),$$



(a) EvolveGCN-H, where the GCN parameters are hidden states of a recurrent architecture that takes node embeddings as input.

(b) EvolveGCN-O, where the GCN parameters are input/outputs of a recurrent architecture.

Figure 2: Two versions of EvolveGCN. In each version, the left is a recurrent architecture; the middle is the graph convolution unit; and the right is the evolving graph convolution unit. Red region denotes information input to the unit and blue region denotes output information. The mathematical notation W means GCN parameters and H means node embeddings. Time t progresses from left to right, whereas neural network layers l are built up from bottom to top.

with details deferred to a later subsection. The LSTM may be replaced by other recurrent architectures, as long as the roles of $W_t^{(l)}$ and $W_{t-1}^{(l)}$ are clear. We use "-O" to denote this version; see the left part of Figure 2(b).

3.3 Evolving Graph Convolution Unit (EGCU)

Combining the graph convolution unit GCONV presented in Section 3.1 and a recurrent architecture presented in Section 3.2, we reach the *evolving graph convolution unit* (EGCU). Depending on the way that GCN weights are evolved, we have two versions:

```
**function** $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-H}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$
    $W_t^{(l)} = \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)})$
    $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$
**end function**
```

```
**function** $[H_t^{(l+1)}, W_t^{(l)}] = \text{EGCU-O}(A_t, H_t^{(l)}, W_{t-1}^{(l)})$
    $W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)})$
    $H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)})$
**end function**
```

In the -H version, the GCN weights are treated as hidden states of the recurrent architecture; whereas in the -O version, these weights are treated as input/outputs. In both versions, the EGCU performs graph convolutions along layers and meanwhile evolves the weight matrices over time.

Chaining the units bottom-up, we obtain a GCN with multiple layers for one time step. Then, unrolling over time horizontally, the units form a lattice on which information ($H_t^{(l)}$ and $W_t^{(l)}$) flows. We call the overall model *evolving graph convolutional network* (EvolveGCN).

3.4 Implementation of the -H Version

The -H version can be implemented by using a standard GRU, with two extensions: (a) extending the inputs and hidden states from vectors to matrices (because the hidden state is now the GCN weight matrices); and (b) matching the column dimension of the input with that of the hidden state.

The matrix extension is straightforward: One simply places the column vectors side by side to form a matrix. In other words, one uses the same GRU to process each column of the GCN weight matrix. For completeness, we write the matrix version of GRU in the following, by noting that all named variables (such as X_t and H_t) are only local variables; they are not to be confused with the mathematical notations we have been using so far. We use these local variable names so that the reader easily recognizes the GRU functionality.

```
function $H_t = g(X_t, H_{t-1})$
    $Z_t = \text{sigmoid}(W_Z X_t + U_Z H_{t-1} + B_Z)$
    $R_t = \text{sigmoid}(W_R X_t + U_R H_{t-1} + B_R)$
    $\widetilde{H}_t = \tanh(W_H X_t + U_H (R_t \circ H_{t-1}) + B_H)$
    $H_t = (1 - Z_t) \circ H_{t-1} + Z_t \circ \widetilde{H}_t$
end function
```

The second requirement is that the number of columns of the GRU input must match that of the hidden state. Let the latter number be k. Our strategy is to summarize all the node embedding vectors into k representative ones (each used as a column vector). The following pseudocode gives one popular approach for this summarization. By convention, it takes a matrix X_t with many rows as input and produces a matrix Z_t with only k rows (see, e.g., (Cangea et al. 2018; Gao and Ji 2019)). The summarization requires a parameter vector p that is independent of the time index t (but may vary for different graph convolution layers). This vector is used to compute weights for the rows, among which the ones corresponding to the top k weights are selected and are weighted for output.

```
function $Z_t = summarize(X_t, k)$
    $y_t = X_t p / \|p\|$
    $i_t = \text{top-indices}(y_t, k)$
    $Z_t = [X_t \circ \tanh(y_t)]_{i_t}$
end function
```

With the above functions g and $summarize$, we now completely specify the recurrent architecture:

$$\begin{aligned} W_t^{(l)} &= \text{GRU}(H_t^{(l)}, W_{t-1}^{(l)}) \\ &:= g(summarize(H_t^{(l)}, \#col(W_{t-1}^{(l)}))^T, W_{t-1}^{(l)}), \end{aligned}$$

where $\#col$ denotes the number of columns of a matrix and the superscript T denotes matrix transpose. Effectively, it summarizes the node embedding matrix $H_t^{(l)}$ into one with appropriate dimensions and then evolves the weight matrix $W_{t-1}^{(l)}$ in the past time step to $W_t^{(l)}$ for the current time.

Note again that the recurrent hidden state may be realized by not only GRU, but also other RNN architectures as well.

3.5 Implementation of the -O Version

Implementing the -O version requires only a straightforward extension of the standard LSTM from the vector version to the matrix version. The following is the pseudocode, where note again that all named variables are only local variables and they are not to be confused with the mathematical notations we have been using so far. We use these local variable names so that the reader easily recognizes the LSTM functionality.

```
function $H_t = f(X_t)$
    Current input $X_t$ is the same as the past output $H_{t-1}$
    $F_t = \text{sigmoid}(W_F X_t + U_F H_{t-1} + B_F)$
    $I_t = \text{sigmoid}(W_I X_t + U_I H_{t-1} + B_I)$
    $O_t = \text{sigmoid}(W_O X_t + U_O H_{t-1} + B_O)$
    $\widetilde{C}_t = \tanh(W_C X_t + U_C H_{t-1} + B_C)$
    $C_t = F_t \circ C_{t-1} + I_t \circ \widetilde{C}_t$
    $H_t = O_t \circ \tanh(C_t)$
end function
```

With the above function f, we now completely specify the recurrent architecture:

$$W_t^{(l)} = \text{LSTM}(W_{t-1}^{(l)}) := f(W_{t-1}^{(l)}).$$

Note again that the recurrent input-output relationship may be realized by not only LSTM, but also other RNN architectures as well.

3.6 Which Version to Use

Choosing the right version is data set dependent. When node features are informative, the -H version may be more effective, because it incorporates additionally node embedding in the recurrent network. On the other hand, if the node features are not much informative but the graph structure plays a more vital role, the -O version focuses on the change of the structure and may be more effective.

4 Experiments

In this section, we present a comprehensive set of experiments to demonstrate the effectiveness of EvolveGCN. The setting includes a variety of data sets, tasks, compared methods, and evaluation metrics. Hyperparameters are tuned by using the validation set and test results are reported at the best validation epoch.

4.1 Data Sets

We use a combination of synthetic and publicly available benchmark data sets for experiments.

Stochastic Block Model. (SBM for short) SBM is a popularly used random graph model for simulating community structures and evolutions. We follow (Goyal et al. 2017) to generate synthetic data from the model.

Bitcoin OTC.[1] (BC-OTC for short) BC-OTC is a who-trusts-whom network of bitcoin users trading on the platform http://www.bitcoin-otc.com. The data set may be used for predicting the polarity of each rating and forecasting whether a user will rate another one in the next time step.

Bitcoin Alpha.[2] (BC-Alpha for short) BC-Alpha is created in the same manner as is BC-OTC, except that the users and ratings come from a different trading platform, http://www.btc-alpha.com.

UC Irvine messages.[3] (UCI for short) UCI is an online com-

[1] http://snap.stanford.edu/data/soc-sign-bitcoin-otc.html

[2] http://snap.stanford.edu/data/soc-sign-bitcoin-alpha.html

[3] http://konect.uni-koblenz.de/networks/opsahl-ucsocial

munity of students from the University of California, Irvine, wherein the links of this social network indicate sent messages between users. Link prediction is a standard task for this data set.

Autonomous systems.[4] (AS for short) AS is a communication network of routers that exchange traffic flows with peers. This data set may be used to forecast message exchanges in the future.

Reddit Hyperlink Network.[5] (Reddit for short) Reddit is a subreddit-to-subreddit hyperlink network, where each hyperlink originates from a post in the source community and links to a post in the target community. The hyperlinks are annotated with sentiment. The data set may be used for sentiment classification.

Elliptic.[6] Elliptic is a network of bitcoin transactions, wherein each node represents one transaction and the edges indicate payment flows. Approximately 20% of the transactions have been mapped to real entities belonging to licit categories versus illicit ones. The aim is to categorize the unlabeled transactions.

These data sets are summarized in Table 1. Training/validation/test splits are done along the temporal dimension. The temporal granularity is case dependent but we use all available information of the data sets, except AS for which we use only the first 100 days following (Goyal et al. 2017).

Table 1: Data sets.

	# Nodes	# Edges	# Time Steps
			(Train / Val / Test)
SBM	1,000	4,870,863	35 / 5 / 10
BC-OTC	5,881	35,588	95 / 14 / 28
BC-Alpha	3,777	24,173	95 / 13 / 28
UCI	1,899	59,835	62 / 9 / 17
AS	6,474	13,895	70 / 10 / 20
Reddit	55,863	858,490	122 / 18 / 34
Elliptic	203,769	234,355	31 / 5 / 13

4.2 Tasks

The proposed EvolveGCN supports three predictive tasks elaborated below. The model for producing the embeddings and the predictive model are trained end to end. The output embedding of a node u by GCN at time t is denoted by h_t^u.

Link Prediction. The task of link prediction is to leverage information up to time t and predict the existence of an edge (u, v) at time $t + 1$. Since historical information has been encoded in the GCN parameters, we base the prediction on h_t^u and h_t^v. To achieve so, we concatenate these two vectors and apply an MLP to obtain the link probability. As a standard practice, we perform negative sampling and optimize the cross-entropy loss function.

Five data sets are used for experimentation for this task. See the header of Table 2. Evaluation metrics include mean average precision (MAP) and mean reciprocal rank (MRR).

Edge Classification. Predicting the label of an edge (u, v) at time t is done in almost the same manner as link prediction: We concatenate h_t^u and h_t^v and apply an MLP to obtain the class probability.

Three data sets are used for experimentation for this task: BC-OTC, BC-Alpha, and Reddit. Evaluation metrics are precision, recall, and F1.

Node Classification. Predicting the label of a node u at time t follows the same practice of a standard GCN: The activation function of the last graph convolution layer is the softmax, so that h_t^u is a probability vector.

Publicly available data sets for node classification in the dynamic setting are rare. We use only one data set (Elliptic) for demonstration. This data set is the largest one in node count in Table 1. The evaluation metrics are the same as those for edge classification.

4.3 Compared Methods

We compare the two versions of the proposed method, EvolveGCN-H and EvolveGCN-O, with the following four baselines (two supervised and two unsupervised).

GCN. The first one is GCN without any temporal modeling. We use one single GCN model for all time steps and the loss is accumulated along the time axis.

GCN-GRU. The second one is also a single GCN model, but it is co-trained with a recurrent model (GRU) on node embeddings. We call this approach GCN-GRU, which is conceptually the same as Method 1 of (Seo et al. 2016), except that their GNN is the ChebNet (Defferrard, Bresson, and Vandergheynst 2016) and their recurrent model is the LSTM.

DynGEM. (Goyal et al. 2017) The third one is an unsupervised node embedding approach, based on the use of graph autoencoders. The autoencoder parameters learned at the past time step is used to initialize the ones of the current time for faster learning.

dyngraph2vec. (Goyal, Chhetri, and Canedo 2019) This method is also unsupervised. It has several variants: dyngraph2vecAE, dyngraph2vecRNN, and dyngraph2vecAERNN. The first one is similar to DynGEM, but additionally incorporates the past node information for autoencoding. The others use RNN to maintain the past node information.

4.4 Additional Details

The data set Elliptic is equipped with handcrafted node features; and Reddit contains computed feature vectors. For all other data sets, we use one-hot node-degree as the input feature. Following convention, GCN has two layers and MLP has one layer. The embedding size of both GCN layers is set the same, to reduce the effort of hyperparameter tuning. The time window for sequence learning is 10 time steps, except for SBM and Elliptic, where it is 5.

[4] http://snap.stanford.edu/data/as-733.html

[5] http://snap.stanford.edu/data/soc-RedditHyperlinks.html

[6] https://www.kaggle.com/ellipticco/elliptic-data-set

Table 2: Performance of link prediction. Each column is one data set.

	mean average precision					mean reciprocal rank				
	SBM	BC-OTC	BC-Alpha	UCI	AS	SBM	BC-OTC	BC-Alpha	UCI	AS
GCN	0.1987	0.0003	0.0003	0.0251	0.0003	0.0138	0.0025	0.0031	0.1141	0.0555
GCN-GRU	0.1898	0.0001	0.0001	0.0114	0.0713	0.0119	0.0003	0.0004	0.0985	0.3388
DynGEM	0.1680	0.0134	0.0525	0.0209	0.0529	0.0139	0.0921	0.1287	0.1055	0.1028
dyngraph2vecAE	0.0983	0.0090	0.0507	0.0044	0.0331	0.0079	0.0916	0.1478	0.0540	0.0698
dyngraph2vecAERNN	0.1593	**0.0220**	**0.1100**	0.0205	0.0711	0.0120	**0.1268**	**0.1945**	0.0713	0.0493
EvolveGCN-H	0.1947	0.0026	0.0049	0.0126	**0.1534**	**0.0141**	0.0690	0.1104	0.0899	**0.3632**
EvolveGCN-O	**0.1989**	0.0028	0.0036	**0.0270**	0.1139	0.0138	0.0968	0.1185	**0.1379**	0.2746

4.5 Results for Link Prediction

The MAP and MRR are reported in Table 2. At least one version of EvolveGCN achieves the best result for each of the data sets SBM, UCI, and AS. For BC-OTC and BC-Alpha, EvolveGCN also outperforms the two GCN related baselines, but it is inferior to DynGEM and dyngraph2vec. These latter methods differ from others in that node embeddings are obtained in an unsupervised manner. It is surprising that unsupervised approaches are particularly good on certain data sets, given that the link prediction model is trained separately from graph autoencoding. In such a case, graph convolution does not seem to be sufficiently powerful in capturing the intrinsic similarity of the nodes, rendering a much inferior starting point for dynamic models to catch up. Although EvolveGCN improves over GCN substantially, it still does not reach the bar set by graph autoencoding.

4.6 Results for Edge Classification

The F1 scores across different methods are compared in Figure 3, for the data sets BC-OTC, BC-Alpha, and Reddit. In all cases, the two EvolveGCN versions outperform GCN and GCN-GRU. Moreover, similar observations are made for the precision and the recall, which are omitted due to space limitation. These appealing results corroborate the effectiveness of the proposed method.

4.7 Results for Node Classification

The F1 scores for the data set Elliptic are plotted also in Figure 3. In this data set, the classes correspond to licit and illicit transactions respectively and they are highly skewed. For financial crime forensic, the illicit class (minority) is the main interest. Hence, we plot the minority F1. The micro averages are all higher than 0.95 and not as informative. One sees that EvolveGCN-O performs better than the static GCN, but not so much as GCN-GRU. Indeed, dynamic models are more effective.

For an interesting phenomenon, we plot the history of the F1 scores along time in Figure 4. All methods perform poorly starting at step 43. This time is when the dark market shutdown occurred. Such an emerging event causes performance degrade for all methods, with non-dynamic models suffering the most. Even dynamic models are not able to perform reliably, because the emerging event has not been learned.

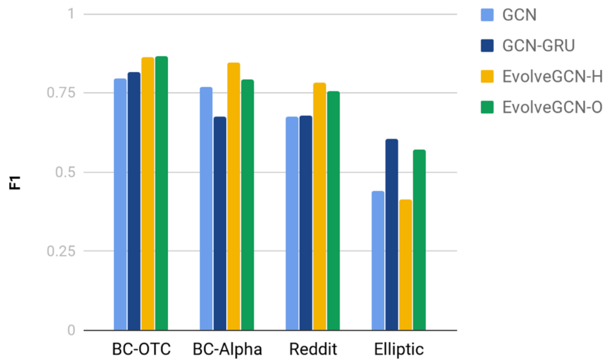


Figure 3: Performance of edge classification and node classification. For edge classification (BC-OTC, BC-Alpha, and Reddit), the F1 score is the micro average. For node classification (Elliptic), because of the exceedingly high class imbalance and strong interest in the minority class (illicit transactions), the minority F1 is plotted instead.

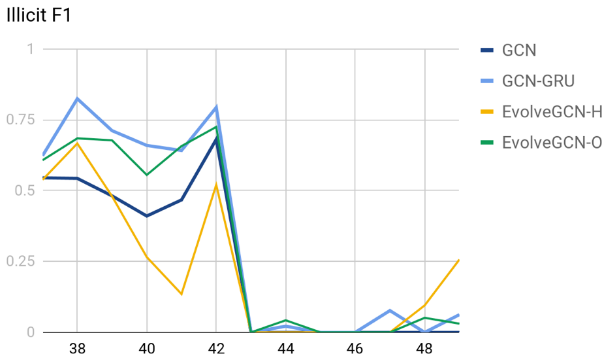


Figure 4: Performance of node classification over time. The F1 score is for the minority (illicit) class.

5 Conclusions

A plethora of neural network architectures were proposed recently for graph structured data and their effectiveness have been widely confirmed. In practical scenarios, however, we are often faced with graphs that are constantly evolving, rather than being conveniently static for a once-for-all investigation. The question is how neural networks handle such a dynamism. Combining GNN with RNN is a natural idea. Typical approaches use the GNN as a feature extractor and use an RNN to learn the dynamics from the extracted node features. We instead use the RNN to evolve the GNN, so that the dynamism is captured in the evolving network parameters. One advantage is that it handles more flexibly dynamic data, because a node does not need to be present all time around. Experimental results confirm that the proposed approach generally outperforms related ones for a variety of tasks, including link prediction, edge classification, and node classification.

References

[Belkin and Niyogi 2002] Belkin, M., and Niyogi, P. 2002. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS*.

[Bruna et al. 2014] Bruna, J.; Zaremba, W.; Szlam, A.; and LeCun, Y. 2014. Spectral networks and locally connected networks on graphs. In *ICLR*.

[Cangea et al. 2018] Cangea, C.; Veličković, P.; Jovanović, N.; and Thomas Kipf, P. L. 2018. Towards sparse hierarchical graph classifiers. In *NIPS Workshop on Relational Representation Learning*.

[Cao, Lu, and Xu 2015] Cao, S.; Lu, W.; and Xu, Q. 2015. GraRep: Learning graph representations with global structural information. In *CIKM*.

[Chen, Ma, and Xiao 2018] Chen, J.; Ma, T.; and Xiao, C. 2018. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *ICLR*.

[Defferrard, Bresson, and Vandergheynst 2016] Defferrard, M.; Bresson, X.; and Vandergheynst, P. 2016. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS*.

[Duvenaud et al. 2015] Duvenaud, D.; Maclaurin, D.; Aguilera-Iparraguirre, J.; Gómez-Bombarelli, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. P. 2015. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS*.

[Gao and Ji 2019] Gao, H., and Ji, S. 2019. Graph U-Nets. In *ICML*.

[Gilmer et al. 2017] Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for quantum chemistry. In *ICML*.

[Goyal et al. 2017] Goyal, P.; Kamra, N.; He, X.; and Liu, Y. 2017. DynGEM: Deep embedding method for dynamic graphs. In *IJCAI Workshop on Representation Learning for Graphs*.

[Goyal, Chhetri, and Canedo 2019] Goyal, P.; Chhetri, S. R.; and Canedo, A. 2019. dyngraph2vec: Capturing network dynamics using dynamic graph representation learning. *Knowledge-Based Systems*.

[Grover and Leskovec 2016] Grover, A., and Leskovec, J. 2016. node2vec: Scalable feature learning for networks. In *KDD*.

[Hamilton, Ying, and Leskovec 2017] Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Inductive representation learning on large graphs. In *NIPS*.

[Jin et al. 2017] Jin, W.; Coley, C. W.; Barzilay, R.; and Jaakkola, T. 2017. Predicting organic reaction outcomes with Weisfeiler-Lehman network. In *NIPS*.

[Kipf and Welling 2017] Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[Li et al. 2016] Li, Y.; Tarlow, D.; Brockschmidt, M.; and Zemel, R. 2016. Gated graph sequence neural networks. In *ICLR*.

[Li et al. 2017] Li, J.; Dani, H.; Hu, X.; Tang, J.; Chang, Y.; and Liu, H. 2017. Attributed network embedding for learning in a dynamic environment. In *CIKM*.

[Manessia, Rozza, and Manzo 2017] Manessia, F.; Rozza, A.; and Manzo, M. 2017. Dynamic graph convolutional networks. arXiv:1704.06199.

[Narayan and Roe 2018] Narayan, A., and Roe, P. H. O. 2018. Learning graph dynamics using deep neural networks. *IFAC-PapersOnLine* 51(2):433–438.

[Nguyen et al. 2018] Nguyen, G. H.; Lee, J. B.; Rossi, R. A.; Ahmed, N. K.; Koh, E.; and Kim, S. 2018. Continuous-time dynamic network embeddings. In *WWW*.

[Ou et al. 2016] Ou, M.; Cui, P.; Pei, J.; Zhang, Z.; and Zhu, W. 2016. Asymmetric transitivity preserving graph embedding. In *KDD*.

[Perozzi, Al-Rfou, and Skiena 2014] Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. DeepWalk: Online learning of social representations. In *KDD*.

[Roweis and Saul 2000] Roweis, S. T., and Saul, L. K. 2000. Nonlinear dimensionality reduction by locally linear embedding. *Science* 290(5500):2323–2326.

[Seo et al. 2016] Seo, Y.; Defferrard, M.; Vandergheynst, P.; and Bresson, X. 2016. Structured sequence modeling with graph convolutional recurrent networks. arXiv:1612.07659.

[Tang et al. 2015] Tang, J.; Qu, M.; Wang, M.; Zhang, M.; Yan, J.; and Mei, Q. 2015. LINE: Large-scale information network embedding. In *WWW*.

[Trivedi et al. 2017] Trivedi, R.; Dai, H.; Wang, Y.; and Song, L. 2017. Know-Evolve: Deep temporal reasoning for dynamic knowledge graphs. In *ICML*.

[Trivedi et al. 2018] Trivedi, R.; Farajtabar, M.; Biswal, P.; and Zha, H. 2018. Representation learning over dynamic graphs. arXiv:1803.04051.

[Veličković et al. 2018] Veličković, P.; Cucurull, G.; Casanova, A.; Romero, A.; Liò, P.; and Bengio, Y. 2018. Graph attention networks. In *ICLR*.

[Yu et al. 2018] Yu, W.; Cheng, W.; Aggarwal, C.; Zhang, K.; Chen, H.; and Wang, W. 2018. NetWalk: A flexible deep embedding approach for anomaly detection in dynamic networks. In *KDD*.

[Yu, Yin, and Zhu 2018] Yu, B.; Yin, H.; and Zhu, Z. 2018. Spatio-temporal graph convolutional networks: A deep learning framework for traffic forecasting. In *IJCAI*.

[Zhou et al. 2018] Zhou, L.; Yang, Y.; Ren, X.; Wu, F.; and Zhuang, Y. 2018. Dynamic network embedding by modeling triadic closure process. In *AAAI*.

[Zuo et al. 2018] Zuo, Y.; Liu, G.; Lin, H.; Guo, J.; Hu, X.; and Wu, J. 2018. Embedding temporal network via neighborhood formation. In *KDD*.