Università della Svizzera italiana

Graph Deep Learning

Academic Year 2020/2021 D. Raffo, F. Boelter, J. Schulte

Due date: Thursday, May 17, 2021, 10.30am

Non-markovian Graph Edit Networks

Abstract

The goal of this short paper is to extend the Graph Edit Networks proposed by Paassen et al. [2] to account for non-Markovian systems. To accomplish this, we apply the EvolveGCN framework by Pareja et al. [3], a developement of Graph Convolutional Networks (GCNs) which infers the parameters of the network by using a recurrent model. We test this on time series of graphs, which we call graph dynamical systems, and compare the results against the markovian model in [2].

Introduction

The prediction of networks has many different applications that make it a field of increasing importance. The Graph Edit Networks proposed by Paassen et al. represent one approach to predict both the edges and the nodes of a network in the next time step [2]. However, this method works only in the context of Markovian data. In the paper at hand, we extended this method to work in the context of non-Markovian data. The main aim of [2] was to explore the connection between Graph Neural Networks (GNNs) and Graph Edit Distances (GED) to expand the possibility of prediction of a graph G_{t+1} using the graph at the previous time step G_t as an input. Since currently available methods were capable only of predictions on node attributes but not on changes in the graph structure (i.e. insertion/deletion/relabeling of nodes/edges), GED can provide for a meaningful measure of how the graph should evolve at the next time step. The output layer of the presented model is called Graph Edit Network (GEN), which computes a sequence of graph edits from node embeddings.

We summarize here some key concepts necessary to understand the work. Most of the following definitions are taken from [2].

A **Graph neural network** computes representations of nodes in a graph by aggregating information of neighboring nodes. In particular, the representation $\phi^l(v) \in \mathbb{R}$ of node v in layer l is computed as follows:

$$\phi^l(v) = f^l_{merge}\big(\phi^{l-1}(v), f^l_{aggr}(\{\phi^{l-1}(u)|u \in N(V)\})\big) \tag{1}$$

where $\mathcal{N}(v)$ is some neighborhood of v in the graph and f^l_{merge} as well as f^l_{aggr} are functions that aggregate the information of their arguments, returning a single vector.

An **attributed, directed graph** is a triple $G = (V, E, \mathbf{X})$ where $V = \{1, \ldots, N\}$ is a finite set of node indices, $E \subseteq V \times V$ is a set of edges, and $\mathbf{X} \in \mathbb{R}^{N \times n}$ is a matrix of node attributes for some $n \in \mathbb{N}$. We define the nodes as indexes for notational simplicity, but we do not assume any specific order, i.e. we treat isomorphic graphs the same. In addition, define the adjacency matrix of G as $A \in \{0, 1\}^{N \times N}$, where $A_{i,j} = 1$ if the edge $e(i, j)$ exists, and 0 otherwise.

We define a **graph edit** as some function $\delta : \mathcal{G} \to \mathcal{G}$, which can contain any sequence of the following graph edits: *node deletions* del_i, which delete the ith node from a graph, *node replacements* $\text{rep}_{i,x}$, which set the node attribute of node i to x, *node insertions* ins_x, which add a new node with attribute x to a graph, edge deletions $\text{edel}_{i,j}$, which delete the edge (i, j) from a graph, and edge insertions $\text{eins}_{i,j}$, which insert the edge (i, j) into a graph. We then define an edit script $\overline{\delta}$ as a finite sequence $\overline{\delta} = \delta_1, \ldots, \delta_T$ of graph edits and we define the application of $\overline{\delta}$ as the composition of all edits, i.e. $\overline{\delta}(G) := \delta_T(G) \circ \cdots \circ \delta_1(G)$.

Finally, we define the **graph edit distance** $d_{GED}(G, G')$ between two graphs G and G' as the length of the shortest script $\overline{\delta}$ such that $\overline{\delta}(G) \cong G'$, where $\cong$ means isomorphic.

Graph Edit Networks

Let $G_1, G_2, \ldots G_T$ be a time series of graph. Differently from the markovian setting in [2], our goal is now to devise a GNN which takes a time series of graph $G_{t-k}, G_{t-k+1}, \ldots, G_t$ as input and outputs graph edits that transform G_t to G_{t+1}.

Now, let $G_t = (V, E, \mathbf{X})$ be an attributed graph with N nodes. The processing pipeline has three steps. First, we use some graph neural network (refer to Equation 1) to compute a matrix of node embeddings $\Phi \in \mathbb{R}^{N \times n}$. Second, we use a linear layer to compute numerical edit scores that express which nodes and edges should be deleted, inserted, and relabeled, respectively: these are node edit scores $\vec{\nu} \in \mathbb{R}^N$, edge filter scores $\vec{e}^{+} \in \mathbb{R}^{N \times n}$ as well as $\vec{e}^{-} \in \mathbb{R}^{N \times n}$, and new node attributes $\mathbf{Y}$ via linear maps from Φ. Third, we translate these scores to an edit script and apply this script to the input graph to obtain the output graph $\overline{\delta}(G_t)$.

The key challenge in training GENs is to identify which scores the network should produce such that the GEN transforms the input graph G_t into its desired successor G_{t+1}. In other words, we require a teaching signal consisting of ground truth scores $(\hat{\nu}, \hat{Y}, \hat{e}^{+}, \hat{e}^{-}, \hat{\mathcal{E}})$, such that the edit script yields $\overline{\delta}(G_t) \cong G_{t+1}$.

Unfortunately, such a one-step teaching signal is sometimes insufficient. That is why we swapped model's core neural network from a classic GCN to a non-markovian EvolveGCN, as suggested in [2].

EvolveGCN

EvolveGCN addresses the problems that changes in the node set across the graph time series generate in the prediction process. For this purpose, it adapts the graph convolutional network (GCN) model along the temporal dimension, incorporating non-markovianity into the model.

The following definitions are taken directy from [3] and adapted to the non-markovian setting. A **GCN** consists of multiple layers of graph convolution, with 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 the weight matrix $W_t^{(l)}$ to update the node embedding matrix to $H_t^{(l+1)}$ as output. Mathematically, we write:

$$H_t^{(l+1)} = \text{GCONV}(A_t, H_t^{(l)}, W_t^{(l)}) := \sigma(\hat{A}_t, H_t^{(l)}, W_t^{(l)}) \tag{2}$$

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

$$\hat{A} = \widetilde{D}^{-\frac{1}{2}}\widetilde{A}\widetilde{D}^{-\frac{1}{2}}, \qquad \widetilde{A} = A + I, \qquad \widetilde{D} = \text{diag}\Big(\sum_j \widetilde{A}_{ij}\Big),$$

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.

The parameters of the GCN are the weight matrices $W_t^{(l)}$, for different time steps t and layers l. Here is where the non-markovianity is inserted. EvolveGCN utilizes two different options, based on a recurrent architecture, to update $W_t^{(l)}$ at time-t. The first, denoted by "-H", also utilizes the node embedding matrix H_t for the prediction of W_t; the second, denoted by "-O", only utilizes the past weight matrix, W_{t-1}. For our implementation, we decided to only use the second version. Here are the details.

EvolveGCN-O treats $W_t^{(l)}$ as the output of the dynamical system (which becomes the input at the subsequent time step). A long short-term memory (LSTM) cell is used to model this input-output relationship. The LSTM itself maintains the system information by using a cell context. In this version, node embeddings are not used at all. Abstractly, we write:

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

Combining the graph convolution unit GCONV and the recurrent architectures above, we reach the **evolving graph convolution unit (EGCU)**:

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

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)**.

Datasets

We tested the model both on synthetic graph dynamical systems on which the GCN-based GEN was tested in [2], and real-data graph dynamical system which presumably have some non-markovian pattern that EvolveGCN can detect and exploit in its predictions.

Synthetic graph dynamical systems

Note that these datasets are strictly Markovian, therefore we were not expecting any significant improvement with the implementation of EGCN-O.

- **Edit Cycles**. A manually defined dataset of cycles in the set of undirected graphs with up to four nodes. The teaching protocol is hand-crafted to perform optimal edits between each graph and its successor. To sample a time series we let the cycle run for 4-12 time steps at random. The node features X_0 were set to zero.

- **Degree Rules**. A dynamical system on undirected graphs of arbitrary size with the following rules. First, delete every node with a degree larger than 3. Second, connect nodes that share at least one common neighbor. Third, insert a new node at any node with a degree lower than 3. We used the node index in one-hot coding as node features X_0. For every connected component in the input graph, this dynamical system provably converges to a 4-clique. We started with a random undirected adjacency matrix of size 8×8 and let the system run until convergence.

- **Game of life**. We simulated one of five oscillatory shapes in Conway's game of life [1], namely blinker, glider, beacon, toad, and clock, for 10 time steps, placed on a random location on a 10×10 grid and additionally activated 10% of the grid cells at random. We represented the grid as a graph with 100 nodes and represented the 8-neighborhood via the adjacency matrix. The desired edits were node deletions for all nodes that should switch from alive to dead and node insertions for all nodes that should switch from dead to alive. As node features we used the alive-state of the node.

Real-data graph dynamical systems

This datasets are directed, temporal, labeled graph datasets publicly available from download from the SNAP library mantained by the University of Stanford. The following descriptions are directly taken from the website.

- **High-energy physics theory citation network.** Arxiv HEP-TH (high energy physics theory) citation graph is from the e-print arXiv and covers all the citations within a dataset of 27,770 papers with 352,807 edges. If a paper i cites paper j, the graph contains a directed edge from i to j. If a paper cites, or is cited by, a paper outside the dataset, the graph does not contain any information about this. The data covers papers in the period from January 1993 to April 2003 (124 months). It begins within a few months of the inception of the arXiv, and thus represents essentially the complete history of its HEP-TH section. The data was originally released as a part of 2003 KDD Cup.

- **CollegeMsg temporal network**. This dataset is comprised of private messages sent on an online social network at the University of California, Irvine. Users could search the network for others and then initiate conversation based on profile information. An edge (u, v, t) means that user u sent a private message to user v at time t. The dataset here is derived from the one hosted by Tore Opsahl, but we have parsed it so that it can be loaded directly into SNAP as a temporal network.

Implementation and results

We compare the results of the GEN implemented through EvolveGCN-O with the baseline provided by a standard GCN implementation without any temporal modeling.

We added a final linear layer to the EvolveGCN-O architecture to make it comply to the GEN structure. The structure of our EvolveGCN includes 3 recurrent layers (LSTM), followed by a GCNConv layer, and then k layers of standard GCN (5 in our case). Instead, the standard GCN implementation only includes those k layers.

We train all networks with an Adam optimizer in PyTorch using a learning rate of 10^{-3} and stopping training after 2,000 time series or if the loss dropped below 10^{-3}. After training, we evaluated the predictive performance on 10 additional time series. We repeated each experiment five times.

Synthetic graph dynamical systems

The recall and precision for all all models, and all datasets is shown in Table 1, while the validation loss through the epochs is given in Figure 1.

	node insertion		node deletion		edge insertion		edge deletion	
model	recall	precision	recall	precision	recall	precision	recall	precision
				edit cycles				
GCN	1.0	0.983	0.971	0.965	1.0	1.0	1.0	1.0
EGCN-O	0.920	0.939	0.788	0.931	0.9956	0.965	1.0	1.0
				degree rules				
GCN	0.913	0.908	0.801	0.989	0.357	0.938	1.0	0.994
EGCN-O	0.975	0.976	0.953	0.989	0.453	0.964	1.0	1.0
				game of life				
GCN	0.190	1.0	0.211	0.942	1.0	0.572	1.0	0.692
EGCN-O	0.222	1.0	0.218	1.0	1.0	0.728	1.0	0.65

Table 1. The average precision and recall values across five repeats on the synthetic graph dynamical systems.

Contrary to what we expected, we notice that notwithstanding the Markovian nature of this synthetic graph dynamical systems, EvolveGCN-O reaches better performances in the degree rules and game of life datasets, with a significance reduction in loss and better precision-recall measures for the majority of edit operations. The standard GCN implementation is still better over the edit cycles dataset, with a loss trend which suggests that increasing epochs loss could have been reduced even more, while the loss of EGCN-O stabilizes after roughly 500 epochs. In general though, for all the datasets EGCN-O is significantly faster, in terms of number of epochs, to reach a low loss (i.e., converges faster).

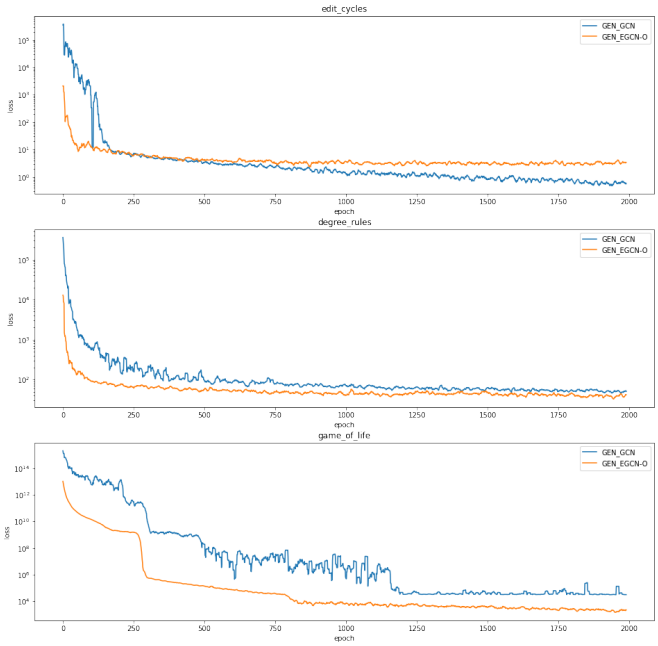

Figure 1. Validation loss over epochs for the different models.

Real-data graph dynamical systems

For our implementation on graph dynamical systems, we modified the code of the GEN and EvolveGCN to make it run on GPU, and run our experiments on the USI cluster which uses Nvidia GTX 1080 GPUs.

For the HEP-TH dataset, the loss over epochs is reported in Figure 3. Note that we are training and testing the model on the monthly data. We can see that the reduction in loss that the EGCN implementation is able to bring over the GCN implementation is quite significant: EGCN converges almost immediately, after a few epochs, to a value of the loss function around 2000, while GCN takes more to stabilize around a value around 3000. That means a 33.3% improvement over the validation loss.

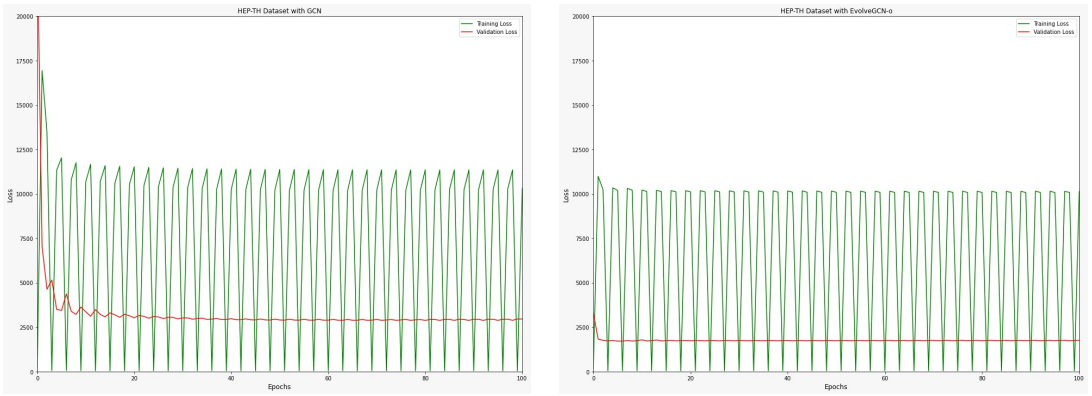

Figure 2. Training and validation loss over epochs for the different models on the HEP-TH dataset. Left: GCN based implementation. Right: EGCN-O based implementation.

For the CollegeMsg dataset, the loss over epochs is reported in Figure 3. Note that we are training and testing the model on the daily data. Here, we can say how the utilization of EGCN does not provide any visible benefit over GCN; the EGCN validation loss looks more "wobbly", but it looks like it is converging to the same value of the validation loss of GCN, which converges quicker. This might be due to a less Markovian nature of the dataset. Still, more epochs might give a better idea about the precise values around the two architectures are converging to.

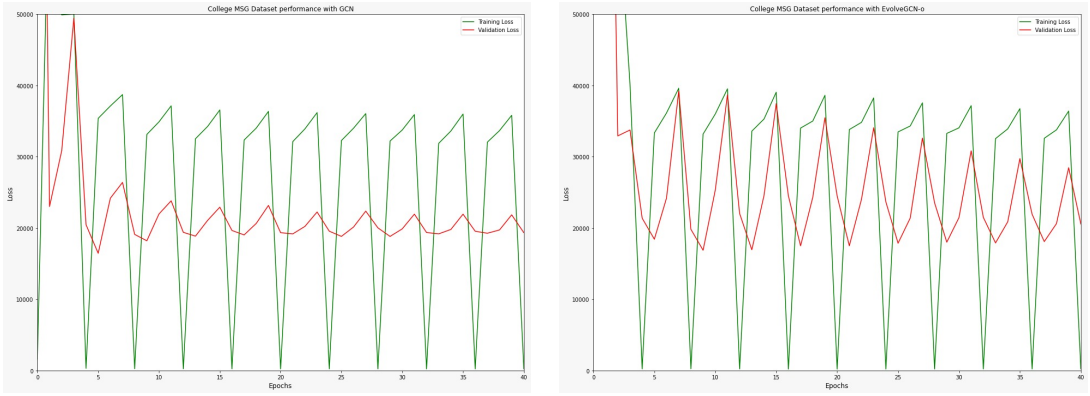

Figure 3. Training and validation loss over epochs for the different models on the collegeMsg dataset. Left: GCN based implementation. Right: EGCN-O based implementation.

Conclusion

We implemented Graph Edit Networks in a non-markovian setting, changing the core of GNN architecture from a classic Graph Convolutional Network to the EvolveGCN framework. Even on Markovian-by-design graph dynamical systems, EvolveGCN-O showed better performances than GCN.

EvolveGCN proved to significantly improve performance of the Graph Edit Network on one out of two of the temporal, real-world datasets that we tested it on against a standard GCN implementation.

Future research can be addressed toward the direction of implementing the -H version of EvolveGCN, where the GCN weights are treated as hidden states of the recurrent architecture, which in this case is a Gated Recurrent Unit (GRU) instead of an LSTM, and node embedding matrices are used in the prediction of the weight matrices. differently than in the -O version.

References

[1] Mathematical Games. The fantastic combinations of john conway's new solitaire game "life" by martin gardner. *Scientific American*, 223:120–123, 1970.

[2] Benjamin Paassen, Daniele Grattarola, Daniele Zambon, Cesare Alippi, and Barbara Eva Hammer. Graph edit networks. In *International Conference on Learning Representations*, 2021.

[3] Aldo Pareja, Giacomo Domeniconi, Jie Chen, Tengfei Ma, Toyotaro Suzumura, Hiroki Kanezashi, Tim Kaler, Tao B. Schardl, and Charles E. Leiserson. EvolveGCN: Evolving graph convolutional networks for dynamic graphs. In *Proceedings of the Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.