Technische Universität Berlin
Fakulät IV
Mentor: Serafeim Papadias
WiSe 2020/2021

Dynamic Network Embedding :An Extended Approach

Project summary

Haroon Rashid 452834
Ali Arous 452833
Amit Nautiyal 408787

2022-08-17

Contents

1 Abstract

Machine Learning on networks requires learning representations from the raw network that are suitable for the down-stream machine learning algorithms. The representations are learnt by embedding the nodes into the feature space based on the node content and the neighbourhood. A lot of methods have been proposed in the literature to learn the embeddings on the static networks. However, real-world networks are evolving continuously and there is a need to learn the representations that adapt to the continuous changes in the network. Most dynamic embedding methods apply static embedding methods on the new snapshot of the network. These methods, though perform well in practice, are very in-efficient. Recently, Dynamic Network Embedding (DNE) has been proposed to learn the network embeddings for the dynamic networks efficiently. This work designs a set of experiments to understand the strength and limitations of different embedding algorithms in the dynamic setting. Further experiments have also been performed to understand the performance of DNE for machine learning tasks on networks such as link prediction and node classification. Experiments are done to test the limitations of DNE for different snapshot sizes as well as its performance on large-scale datasets. The results from experiments demonstrate that the DNE is a very good choice in terms of efficiency with a minimum compromise on performance for large-scale networks in the dynamic setting.

2 Introduction

Network analysis has become very critical in large-scale applications with the advent of big data. The evolution of social networks and the worldwide web has resulted in large networks with billions of nodes and their interconnections. It is very desirable to perform analysis on these large networks to improve the products and services for the end-user. Recently, machine learning has become a very popular tool in many domains including network analysis. Network analysis on social networks, product-user interaction networks as well as other real-world networks uses machine learning (ML) solutions to learn usage patterns.

A pre-requisite to apply machine learning on networks is to transform the network into a representation that is useful for machine learning algorithms. Most machine learning algorithms operate on some features of data. Extracting features from the networks is very critical to apply machine learning to them. A popular technique is to embed the raw nodes or edges in a network to common feature space and learn models over that space i.e. each node is embedded into a d-dimension space $\mathbb{R}^d$. The most common algorithms, found in the literature, to learn embedding space operate on static networks. i.e. they assume that the network is fixed and learn the embeddings by taking into account the neighbourhood of the nodes in the graph. Most common of these algorithms are node2vec [1], LINE [6] among others. However, these algorithms do not consider the temporal evolution of the network. Real-world networks are very dynamic with frequent addition and removal of nodes and addition and removal of connections. It is very desirable to develop an algorithm that learns the embeddings by taking into account the temporal evolution of the networks. A common approach to do so is to apply static embedding methods like node2vec[1] and LINE[6] on the whole network after a certain period. This approach tends to be very inefficient as learning embeddings on new snapshots of the networks requires re-learning the embeddings on the whole network.

Recently, a new approach called Dynamic Network Embedding (DNE) [4] has been developed that learns the embeddings on dynamic networks by learning the embeddings on new nodes that are added and updates the embeddings of nodes that are affected by the addition or removal of the nodes from the previous snapshots. In this way DNE provides an efficient way to learn the embeddings on the dynamic networks. In this work, we have done a study on DNE and explored its effectiveness and performance on machine learning tasks. The machine learning tasks chosen are node classification and link prediction, two of the most common machine learning tasks in the network analysis. We have devised and implemented different sets of experiments to test the performance of DNE compared with static embeddings methods both in terms of time efficiency and performance on ML tasks. We have also explored the impact of different parameters on the performance of DNE as well as tested its effectiveness in terms of scalability by learning embeddings on a larger network.

3 Background and Related Work

Network embedding is a mapping of the network into a feature space. Network embedding can be categorized into two types: 1) Static Network Embedding and Dynamic Network Embedding. This categorization is based on whether the network is evolving or is static. This work deals with the networks that are evolving as in the case of social networks or the worldwide web. These dynamic networks are more common in a real-world setting and learning representations for continuously evolving dynamic networks is a challenging task.

The literature on Dynamic Network Embedding builds on top of static network embedding methods. Static methods are based on matrix factorization [2], deep autoencoders[3] as well as skip-gram [1] inspired methods. Skip gram based methods are in general more efficient and have good performance. The following sections introduce the most popular static embedding methods and will be used as a background to introduce the Dynamic Embedding Network [4] method.

3.1 node2vec

node2vec [1] defines a notion of neighbourhood network of the node and designs a second order random walk strategy to sample the neighbouring node. node2vec helps in mapping the similar nodes closer to each other in the embeddings space. It does so by identifying the similarity within the community using breadth-first-search (BFS) and exploring out to find similar communities using depth-first-search (DFS). Thus, node2vec learn the mapping of the nodes to a low-dimensional space of features in a robust manner.

3.2 LINE: Large Scale Information Network Embedding

LINE [6] proposes an edge-sampling algorithm which addresses the limitation of the classical stochastic gradient descent and improves both the effectiveness and the efficiency of the inference. LINE preserves the first-order and second-order proximity of the network.

For two nodes v_i and v_j the joint probability distribution between these nodes helps in generating the first order proximity as:

$$p_i(v_i, v_j) = \frac{1}{1 + exp(-\mathbf{u}_i^T \mathbf{u}_j)}$$

And, then the second order proximity is calculated by probability if v_j generated by v_i, which is given by :

$$p_i(v_j, v_i) = \frac{exp(-\mathbf{u}_j^T \mathbf{u}_i)}{\sum_k exp(-\mathbf{u}_k^T \mathbf{u}_i)}$$

3.3 Dynamic Network Embedding

A simple and naive method to extend static embedding methods to the dynamic setting is to apply the static embedding algorithms to each snapshot of the dynamic network. However, this approach may result in drifting of the embedding space and is time inefficient. In most cases, a network may not change much during a short period in dynamic situations, which means that the embedding space should not change much, and retraining for the whole network is inefficient.

Dynamic Network Embedding (DNE) [4] is an efficient and stable embedding framework for dynamic networks. It is an extension of **Skip-gram network embedding** (SGNE) methods in a dynamic setting. All SGNE methods are supported by the framework to learn the initial embeddings. The DNE framework only updates the embeddings for the newly added nodes and the affected nodes. The framework is formally introduced below:

Dynamic Network embeddings is a sequence of functions:

$\mathbf{\Phi} = (\mathbf{\Phi_1}, ...\mathbf{\Phi_T})$ for a dynamic graph $\mathbf{G}$,

where each function $\mathbf{\Phi_T} \subset \mathbf{\Phi}$is

$\mathbf{V_T} \to \mathrm{R}^d, (d << min_T|V_T|)$

is learnt such that it preserves the structural property of G_T.

The dynamic embedding process is divided into two parts, the representational learning of new nodes and adjustment of the nodes from previous snapshots. In this way, DNE provides an efficient and accurate method for learning the network embeddings in a dynamic setting. In the following sections.

So, the DNE algorithm is tested compared to the static embedding methods for different machine learning tasks such as node classification and link prediction applied to the dynamic networks.

3.4 Node Classification

In the node classification problem, it is assumed that a subset of the nodes in the network are labelled. It is desirable to use these labelled nodes in conjunction with the structure and content for the classification of nodes that are not currently labelled.

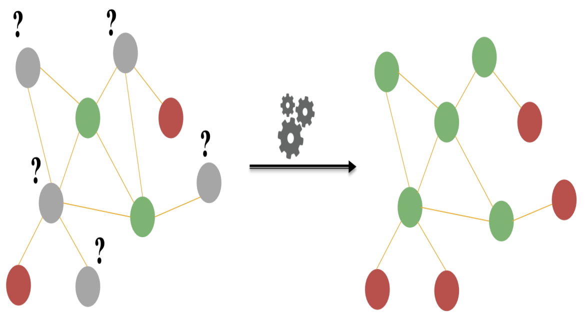

Figure 1: The grey nodes on the left are unlabeled i.e '?' and are not classified as 'green' or 'brown'. Node classification task label these nodes as 'green' or 'brown'(on right side) by understanding the structure and content of the neighbouring nodes.

For example, many blogs or other network documents may naturally belong to specific topics based on their content and linkage patterns. It is very desirable to classify new and unknown nodes to their topics and respective classes.

3.5 Link Prediction

Link Prediction in dynamic networks aims to model the patterns of relationship formed between any two nodes in a multi-node network for predicting future links. In our case we predict links between two connected components in a dynamic network, using intuitive network topological features, a novel feature processing technique especially when time is involved, and two different ways of learning a classifier based on the amount of data collected. We, therefore, can predict the link between the two nodes based on the snapshot of a given network.

4 Implementation

4.1 Node Classification with Node2vec

The node classification script expects the existence of a true labels file (e.g., *dataset_flag.dat*) organised in lines of two space-separated integers, the left number representing a node id, and the right one representing its label. labels could be binary or multi-class. However, to bound the scope of the experiments, the original paper considered only binary labels. We also restricted to binary labels to make the node classification and link prediction problems comparable to each other. The node classification script takes the labels file and loads it to memory in the form of a two-column pandas DataFrame with the schema: (node, label).

The script also reads another file representing the dataset (e.g., dataset.dat), and constructs a similar DataFrame, but with the schema (node1, node2). This is because the dataset file is expected to be organised in an incidence list representation of a graph, with each line representing a link between two nodes. Having the dataset reflected in a DataFrame object makes it easy to build a graph object using *Networkx* library by calling the *from_pandas_edgelist()* builder function on the constructed DataFrame.

The next step is to transform the node representation into a numerical vector reflecting the current position of the node in the graph and its proximity to the neighbouring nodes. This implementation employs *node2vec* [1] to achieve this and it initializes a *node2vec* instance with the following parameters: dimensions=100, walk_length=16, num_walks=50. The choice of these parameters is based on their recommended values. We also tested different other configurations and found these values to be giving optimal results for the node classification task. The *node2vec* object takes the *Networkx* graph object as a first parameter, and it trains on it using the random walk strategy by calling the *.fit()* function. We pass this function the parameters (window = 7, and min_count = 1). The result of the training is a key-value dictionary, which for a given node id, maps to its embedding vector.

Given the list of all nodes in the graph, we construct the dataset of node embeddings which represents the set of data samples X for our node classification pipeline. We use *sklearn* library to run a basic classification task, given X (the data samples' set), and y (the node labels which we have ready in our first constructed DataFrame). We do a train-test split of 70-30, and then we run a *LogisticRegression* classifier on the training set, and record its AUC score on the test set. The previously described process is organized in loosely coupled python functions, forming the overall procedure of the node classification task over node2vec embeddings.

For testing the performance of the generated embeddings under the dynamic network condition, we need to simulate a network that increases in size after each time interval. To achieve this, we load the *snapshots repository* file (e.g., dataset.nw_dynamic) which contains future snapshots to be added to the initial graph. The file is generated from the *Data Partitioning* script (section 4.5) which splits the original dataset file into an initial dataset (seen at time t=0) and a list of future nodes to be added in subsequent time steps, stored in the aforementioned *snapshots repository* file. The file contains for each node a line with the node id along with the number of its links (edges) followed by a set of lines representing all the links belonging to the node, in an incidence list format (i.e., [node_id][space][neighbor_id]).

After loading the file in memory, the script loops through this file, taking *snapshot_size* nodes at each iteration, and adding them to the previous graph structure, which results in a new augmented graph for each time step (t=1,2,3,..). Given the new graph, we call our procedure for node classification on node2vec embedding to re-train node2vec on the new graph structure and generate new embeddings that will feed the node classification pipeline. These recorded AUC scores throughout the loop are plotted against the number of increased nodes at each iteration.

4.2 Node Classification with DNE

The implementation for this experiment is taken from the provided Github repo in the original paper, which includes the python script files used by the DNE method. We, however, re-implement this experiment in our codebase to make it aligned with the scripts written for the other experiments. Mainly, we follow the same logic as in 4.1, but instead of applying *node2vec* on the whole network to re-learn the embeddings in each iteration, we use DNE embeddings that are calculated only for the newly added portion of nodes, as well as, the affected ones from the previous state of the graph.

4.3 Link Prediction with node2vec

We provide here our implementation for the Link Prediction problem, and how we applied it over node2vec embeddings of the links in the graph.

For the Link Prediction problem, we need to prepare the dataset from the given graph. This dataset will have features of node pairs and the label would be a binary value indicating the presence or absence of a link. The negative labels will be for links that are absent in the original graph whereas the positive labels will be for links that existed in the original graph, but we will remove them from our picturing of the graph at time (t-1). To find out the negative links we build an adjacency matrix representation of the original graph by calling *to_numpy_matrix()* function from *Networkx* library, passing it the graph instance as the parameter. Since the resulting matrix represents an undirected graph, it is going to be symmetric, and it suffices to loop through the elements above or below the diagonal collecting the node pairs where there is a 0 at their cross-junction cell. We store all collected pairs in a DataFrame assigning for each pair a label of 0 (negative example). Next, We need to add the set of positive examples which represent existing links in the original graph that we are going to drop from the graph state at time t-1. We need to be careful here to not drop links that will disconnect the graph, and this will be our criteria for deciding whether a link is omissible (can serve as a positive example in our dataset) or not. We use the function *number_connected_components()* from *Networkx* graph to tell if after dropping a link we end up with a disconnected graph (i.e., number of connected components > 1). If that was the case we undo the dropping. We continue the described action iteratively until we visit all existing links in the original graph, and at the end of the iterations we end up with a collected list of dropped links that will be added as positive examples to our dataset under construction. The dataset at this point will become a DataFrame of the union of negative and positive examples DataFrames, along with their labels.

The newly constructed dataset will represent the input for the link prediction task (binary classification of links), but before we feed it to the classification pipeline, we need to encode its data examples as numeric feature vectors, and here we come back to the node2vec library, which we train on the graph structure at time (t-1) (after dropping the omissible links) and use it to embed the graph nodes as we did in the previous implementation for node classification in section 4.1. While node2vec embeds nodes in a network to numerical vectors, we still need to find an embedding for the link between every two nodes. We employ several options for link embedding in our experiments, including summing up the two nodes embedding, concatenating the two embeddings or the inner product between the two embedding vectors. We decompose our script for link prediction task in loosely coupled functions, that form together an overall procedure. We invoke this procedure as we did before in section 4.1, iteratively, inside a loop where each iteration represents a new snapshot addition to the original graph. Hence the procedure will redo the calculations from scratch for each snapshot including the node2vec embedding on the new graph structure.

4.4 Link Prediction with DNE

In our implementation for this task, we depend on the DNE implementation provided in the paper's Github repo to generate the dynamic embeddings. The DNE generates the embeddings of different snapshots with a minor adjustment to the previously computed embeddings. This minor adjustment addresses the potentially affected nodes from the previous graph structure. The code generates a dynamic embedding file that contains for each snapshot, the embeddings of all nodes in the new resulting graph. To take advantage of the generated embeddings from DNE, we design a multi-layer embedding dictionary that is constructed as follows:
For each snapshot, we loop through its nodes' embeddings, and for each given embedding, we check if its node already existed in the dictionary keys. If so, we add the embedding to the node's list of *historical* embeddings as the last up-to-date embedding for the current snapshot, if the node didn't exist in the dictionary keys, we add it with a new entry to the dictionary, where the key represents the node id, and the value represents the list of historical embeddings. Since the node is newly added (haven't been embedded before) and if the current snapshot is not the initial one, we need to pad the beginning of the list with 0s for the previous snapshots when the node didn't yet exist.

By maintaining this multi-layer dictionary structure, we accommodate the historical values of the embeddings of graph nodes across different snapshots, and hence it becomes easy to query in O(1) the embedding of a node (i.e., embed[NODE_ID][SNAPSHOT_NUM]), and as a result the embedding of a link (being an aggregate of its connected nodes). Here we add the time spent for generating the embeddings at each snapshot to the dictionary lookup time and to the link embedding time (achieved by summing up its two nodes embeddings or any of the previously discussed variant), to get the overall embedding time for the link prediction on DNE. The Link Prediction task is conducted here for each snapshot as in section 4.3 and the corresponding AUC values are recorded.

4.5 Dataset Partitioning

The data partitioning script follows a systematic approach to split the dataset into an initial dataset file, and a file containing the remaining nodes along with their connections (i.e., the *snapshots repository* file). Given as input a dataset file named, for example, *dataset.dat*, the script will output a first file named *dataset.nw_init* and a second one named *dataset.nw_dynamic*. The former represents the initial dataset (graph at time t=0 in the dynamic scenario), and the latter file represents a repository of nodes to be added in subsequent snapshots (t=1,2,3,..) to the initial graph.

The partitioning script is compatible with scripts written for previous tasks. Thus, the dataset files generated by this script work flawlessly on the same code that we implemented before and run on the Amherst dataset (already partitioned). The script, however, requires the input dataset file to start with a line containing a single number representing the size of the dataset, followed by a set of lines each containing two space-separated numbers representing a link between the source node, to the left, and the sink, to the right. The key requirement here is that the source nodes are listed in ascending order and that all the links belonging to a source node are listed in one block of lines with no interference with other source nodes links. This is necessary to construct a *Networkx* graph in memory in an incremental fashion. We point here that all the sink nodes related to a given source node should have less node id value (lexicographical or integer) than the source node (i.e., all the sink nodes have already been listed before as source nodes with their corresponding connections in their line blocks).

Given the raw dataset file structured as above, the data partitioning script will loop through the list of nodes with id less than *initial_dataset_size* parameter, and query all their links from the *Networkx* graph object. Then, it will write these links (in a [source][space][sink] format) in the first output file (representing the initial dataset). While processing each source node's link, a check is made on whether the sink also satisfies being less than *initial_dataset_size*. If not, it is skipped (and added to the graph in a reversed direction). Here we note that while the graphs used in our experiments throughout the project are undirected, the directed graph is used in this script for implementation purposes only and it does not affect the consideration of the input and output files being undirected *incidence list* representation of graphs. Indeed, the first output file generated from this script will contain no duplicate edges (even under undirected consideration).

A similar approach is followed for the second output file *(Snapshots repository)* that stores all remaining nodes from the original graph. These nodes, along with their links, are to be added to the graph in the subsequent snapshots, and they are listed in the *Snapshots repository file* in blocks of lines, where each block starts with a line containing a node n and the number of its links k, followed by k lines representing links information of the node.

4.6 Node Deletion

By node deletion, we refer to the additional modification we introduced to the implementation of incremental graph construction, from snapshots. Here we are interested in testing the effect of both additions and deletions on the overall performance of DNE. We first delete a small ratio of nodes from the existing graph, before adding the set of new nodes in each subsequent snapshot. After we delete the nodes, we also delete a very small ratio of edges from between the remaining nodes. We believe this will be more representative of the real-word changes occurring to a dynamic graph structure. We apply node deletion logic, inside the snapshots addition loop, and just before each snapshot begins. We take the *Networkx* Graph, say G, we get the list of its nodes ids using a *.nodes()* call, and we randomly sample this list with a specified number of nodes to be deleted. After that, we remove the sampled nodes with *.remove_nodes_from()* passing it the sample list.

Several problems appear here due to the inherent DNE dependence on datasets with incremented and non-discontinued numbers of node ids (i.e., for a dataset of size N, the nodes should be strictly listed with ids from 0 to N-1).

This property is used by DNE itself to calculate the affected nodes after each snapshot iteration. Snapshots addition script also depends on it, since then the new nodes in the *snapshots repository file* are arranged in a way that every new node, comes along with a set of links that will connect it to existing nodes in the graph. The expectation here is that all nodes, with id's, before the one's being currently added, already exist (which does not hold in case of random nodes deletion). The snapshot addition script adds new nodes to the existing graph in the ascending order of their ids. To mitigate these problems, we run the nodes renaming procedure, by measuring the size of the graph after random deletion and snapshot addition step. We generate a list of increasing numbers from 0 to SIZE-1, and map it as new labels to the graph nodes. By doing this, we compress labels space to fill the gaps that resulted from random nodes deletion. Another step in this regard is to modify the snapshot addition script to only add a new node's link if it connected the new node to an exiting node id in the previous graph. This can result in reducing the number of newly added nodes by a small margin (i.e., less than 1% of the snapshot size)

5 Experiments

Multiple experiments have been conducted in the project that give us hints and insights about the process of Dynamic network embeddings(DNE) in terms of efficiency and scalability. This section covers the two most important properties that lead to the development of our conclusive remarks about the performance of DNE.

5.1 Data

Before we begin with the experiment, to validate our assumptions on the DNE efficiency estimations, it is crucial to decide on the datasets which are going to be used. For the sake of simplicity, we have focused on binary classification and decided to use opensource datasets that are also well researched by the community. Considering the above-mentioned aspects we have decided to use the social networks in Amherst College(2235 nodes, 90954 edges) and NCI Dataset [5] (122.3K nodes, 265.5K edges).

5.2 Experiment 1: Node classification on node2vec embeddings

In the first experiment, we have studied the impact of node classification(3.4).We perform the experiment on Amherst dataset (5.1) on node2vec embedding(3.1). Here, 1 or 0 label assigned to each node (node classification) where, 1 is for omissible edges, and 0 is for absent edges (link prediction). Our Experiments settings for this experiment consist of an initial network snapshot of size with 578 nodes (including 5110 edges), new snapshots of 200 nodes are added subsequently making a total no of snapshots as 10.

We use Area Under the Curve (AUC) metric for calculating the score and after the first snapshot in fig 9 the AUC scores increase and doesn't show a linear growth but after reaching snapshot 10 the AUC score is higher than the previous snapshot.

5.3 Experiment 2: Node classification on DNE embeddings

In the second experiment, node classification on the DNE framework is studied. Here node2vec[1] helps in learning about the nodes having similar properties. figure(9) in the appendix shows us that after the first snapshot, the AUC score improves but is non-linear like in the section (5.2).However, after a few snapshots, the performance improves.

ML Task	Emb. Method	Init	S1	S2	S3	S4	S5	S6	S7	S8	S9
Node Classification	node2vec	0.88	0.93	0.94	0.94	0.96	0.95	0.97	0.96	0.96	**0.96**
	DNE	0.89	0.91	0.92	0.89	0.90	0.91	0.92	0.92	0.93	0.91
Link Prediction	node2vec	0.66	0.61	0.59	0.59	0.58	0.57	0.57	0.58	0.59	0.57
	DNE	0.71	0.63	0.63	0.61	0.60	0.60	0.61	0.61	0.61	**0.60**

Table 1: AUC scores for node2vec and DNE for different snapshots in the dataset. The scores are reported for both node classification and link prediction

Results:

We observe the following.

1. Figure(9) shows that node2vec(3.1) performs better than DNE for node classification(3.4) by around 5%.

2. We also notice that the DNE maintains its performance despite learning and updating embeddings for only the relevant nodes.

5.4 Experiment 3: Link prediction on node2vec embeddings

We perform the experiment on the similar Amherst dataset (5.1) on node2vec embedding(3.1). Here, 1 or 0 label assigned to each node (node classification) where, 1 is for omissible edges, and 0 is for absent edges (link prediction). Our Experiments settings for this experiment consist of an initial data-set (snapshot) of size with 578 nodes (including 5110 edges), then the Snapshot size consisting of 200 nodes and total no of snapshots as 10.

Through this experiment, we want to observe how the link prediction algorithm performs with the node2vec and DNE. We use AUC metrics for calculating the score. And we can see in figure(10) the DNE for Link Prediction does not show much growth with the increase in the number of snapshots. Or we can conclude that the DNE operates kind of linearly over the period for this experimental setup.

5.5 Experiment 4: Link prediction on DNE embeddings

On performing the fourth experiment for Link prediction(3.5) on DNE embeddings, we learn that the AUC scores decrease in the subsequent snapshots or are not stable for both node2vec as well as the DNE operation.
Results:

The graph embeddings for link prediction on node2vec(3.1)and DNE shows that DNE performance is around 10% more than node2vec for link prediction. DNE maintains its performance despite learning and updating embeddings for only the relevant nodes.

To substantiate this refer fig (10), where we plot a graph for AUC scores for the 10 snapshots. Here we keep the 2000 nodes for the snapshots. The overall DNE's AUC score for the link prediction remains significantly higher for each snapshot and follows the same pattern across the link prediction experiment. Another interesting pattern that is notable is that the AUC score drops down for DNE as well as node2vec after the initial snapshot and thus continues for other snapshots as well.

Also, refer(1) where we can notice that for Link Prediction, for the initial snapshot the AUC score for node2vec and DNE are 0.66 and 0.71 respectively. And it continues to almost decline for the rest of the snapshots.

5.6 Experiment 5: Testing different functions for generating Link Embeddings from Node Embed

In our previous experiments, we have studied about the graph embeddings for link prediction(3.5) and node classification (3.4)on node2vec(3.1) and DNE.We observed that the node classification performed better as compared to the link prediction in terms of AUC scores.

We are now interested in understanding the time-cost for node2vec and the DNE.So, we experimented and observed in fig(2) that the node2vec running time increases linearly almost with the data size. The last two snapshots having the least time cost difference. It is also worth noting that DNE has almost a steady running time across the different snapshots, with even a boost of performance towards the end.

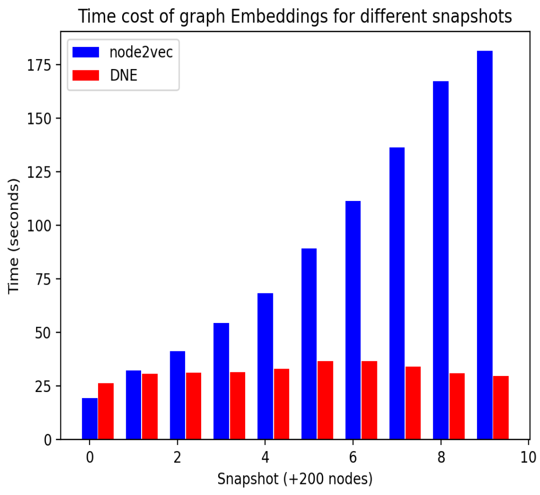

Figure 2: **Experiment result for Time Cost - node2vec vs DNE**

As we can see that DNE performs faster than the node2vec we further see in fig(3) that average time cost for DNE is almost 4 times less than node2vec when the embedding is calculated over the same 10 snapshots.

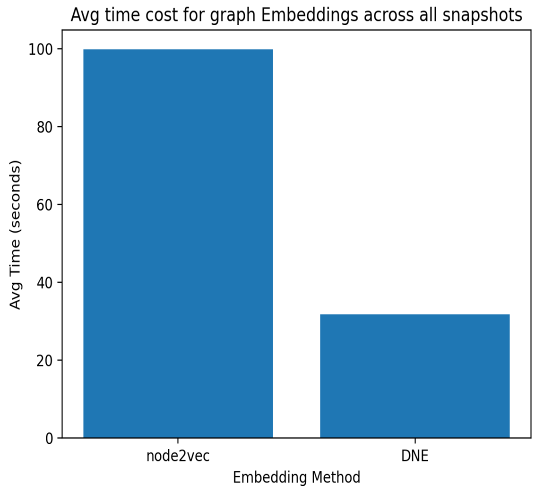

Figure 3: **Experiment result for average time cost- node2vec vs DNE**

In our next experiment, we calculate the Node to link embeddings. Here we experiment with four embedding functions for link embeddings i.e. Add function, Hamdard product function, mean function and concatenate function. for calculation of score, we consider the same AUC scores as used in previous experiments.

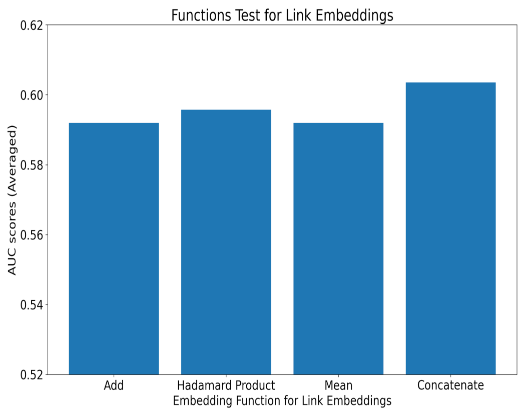

Figure 4: **Experiment result for Node to Link Embedding**

We observed through figure(4) that the Concatenating embeddings give the best results. But it has its drawback i.e. due to concatenation of the embeddings the dimensions becomes double the original.

In contrast to this, if the dimensions are kept the same throughout the experiment then the Hadamard Product function performs best out of the four functions.

5.7 Experiment 6: Performance and Time cost analysis for different snapshot sizes

Until now we have experimented on a similar no of snapshot sizes. But to see the correlation between node2vec and DNE in an extensive way we need to check the impact of varying snapshot sizes on node2vec(3.1) and DNE. So, for our next experiment, we experiment with varying snapshots of sizes:50, 100, 200, 400, 600, 800 and 1000.

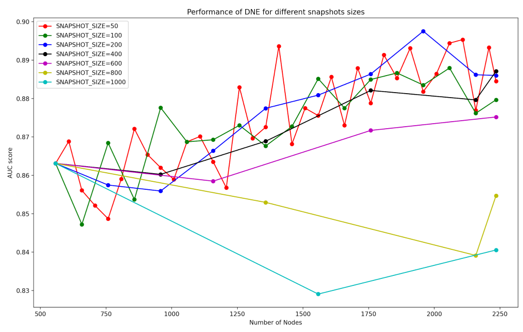

Figure 5: **Experiment result for Different snapshot sizes for DNE**

In fig(5)we see that the accuracy for smaller snapshot sizes i.e 50, 200 is better as compared to the larger snapshots. This is due to the Dynamic nature of DNE embeddings.

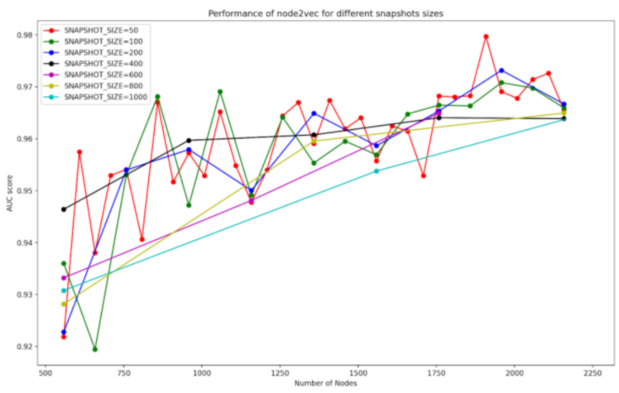

Figure 6: **Experiment result for Different snapshot sizes for node2vec**

However, if we perform the similar experiment for node2vec(3.1) we observe that in fig(6), the node2vec embeddings have no impact with the varying snapshot sizes. This is because node2vec is static embedding.

In terms of time cost for node2vec and DNE in fig(7) we observe that DNE scales much better with the increasing number of snapshots as compared to the node2vec.

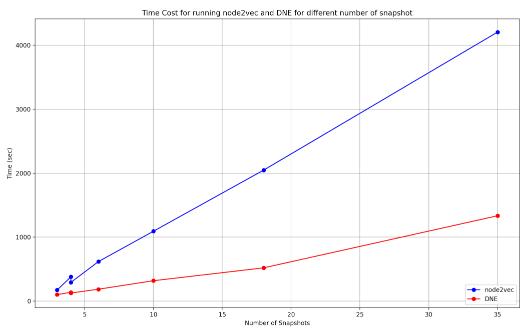

Figure 7: **Time cost for Different snapshot sizes**

5.8 Experiment 7: Scaling the network

After studying the Amherst dataset rigorously, it was decided to push the network scale. In this set of experiments, both node2vec and DNE were tested on the NCI dataset [5].

Emb. Method	Init	S1	S3	S5	S7	S9	S11	S13	S15	S17
node2vec	0.652	0.673	0.678	0.685	0.683	0.678	0.673	0.679	0.682	**0.680**
DNE	0.667	0.683	0.657	0.651	0.649	0.651	0.660	0.651	0.661	0.655

Table 2: AUC scores for node2vec and DNE for different snapshots in the NCI dataset. The scores are reported for both node classification and link prediction

NCI dataset has 122.3K nodes and 265.5K edges. Moreover, it has 3 labels compared to 2 in the previous datasets.

For this dataset, 20,000 nodes were chosen to learn the initial embeddings and 5000 new nodes were added in each snapshot. This way the dataset serves as a good benchmark to test the performance and scalability of DNE and node2vec.

The AUC scores can be seen in the fig(11). The performance for both algorithms drops because of an increased number of data points and classes. However, the comparative performance of both algorithms is similar to the Amherst dataset. As we can see in the table (2), node2vec performs slightly better than DNE in terms of AUC scores.

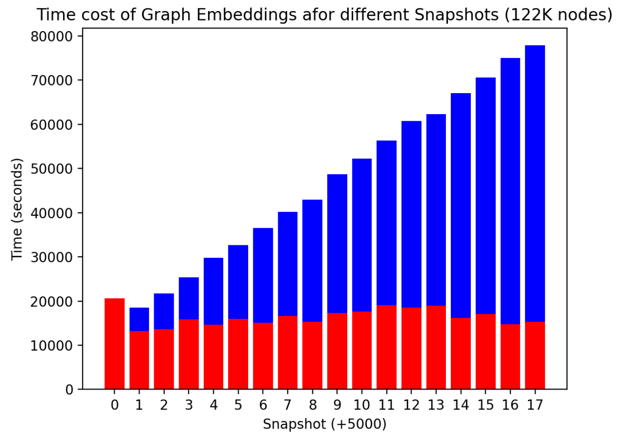

Figure 8: **Time cost for node classification on bigger dataset (100K nodes) - node2vec (blue) vs DNE (red)**

Although node2vec performs slightly better than DNE in terms of AUC score, it does not scale as well in time as compared to DNE fig(8). Since DNE updates only the newly added nodes and the affected nodes, it takes almost constant time for each newly added snapshot. However, node2vec scales linearly as we add new nodes to the graph. This makes DNE a very attractive choice for learning embeddings for large scale dynamic networks.

5.9 Experiment 8: Node classification and Link prediction on DNE embeddings with and without Node deletion

In this experiment, we have studied the impact of node deletion on DNE embeddings for both node classification and link prediction tasks.

Our settings for this experiment consist of an initial data-set of size 578 nodes from Amherst dataset, and then each subsequent snapshot is of size 200 nodes. Before adding each snapshot, we delete randomly a ratio of 5% of the existing nodes in the previous snapshot graph and 2% of edges between remaining nodes. This should represent a realistic scenario of changes that could occur to the graph in the dynamic settings. We plot the AUC scores of DNE in the two settings (DNE: no deletion, DNE_WITH_DEL: with node deletion) for the 10 snapshots available for Amherst dataset. The results shown in figures 12 and 13 show that despite a slight drop in AUC values in the

DNE_WITH_DEL setting, the AUC curve does not drift much from the original case (without applying deletion) and we conclude that DNE is able to handle node deletion well under a small threshold ($\sim$5% of total nodes) for both node classification and link prediction tasks.

6 Conclusion

In this work, we have explored the performance and efficiency of different network embedding methods in the dynamic setting. Several experiments were designed to test the static and dynamic embedding methods for different machine learning tasks (i.e. node classification and link prediction), both in terms of performance and efficiency. It was shown that DNE scales very well while preserving good performance on the downstream machine learning tasks. We also explored the effect of snapshot-size on the performance of DNE which demonstrated that DNE performs well if the snapshot size is kept small (i.e. embeddings are updated frequently). Results from the experiments demonstrate that DNE is a very good choice in terms of efficiency with a minimum compromise on performance for large-scale networks in the dynamic setting.

References

[1] Jure Leskovec Aditya Grover. “node2vec: Scalable Feature Learning for Networks”. In: *In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2016* (2016). ISSN: ISBN 978-1-4503-4232-2/16/08. URL: https://arxiv.org/pdf/1607.00653.pdf.

[2] Mikhail Belkin and Partha Niyogi. “Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering”. In: *Advances in Neural Information Processing Systems*. Ed. by T. Dietterich, S. Becker, and Z. Ghahramani. Vol. 14. MIT Press, 2002.

[3] Shaosheng Cao, Wei Lu, and Qiongkai Xu. “Deep Neural Networks for Learning Graph Representations”. In: *AAAI*. 2016.

[4] Lun Du et al. “Dynamic Network Embedding :An Extended Approach for Skip-gram based Network Embedding”. In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)* (2018). ISSN: JCAI-18. URL: https://www.ijcai.org/Proceedings/2018/0288.pdf.

[5] Ryan A. Rossi and Nesreen K. Ahmed. “The Network Data Repository with Interactive Graph Analytics and Visualization”. In: *AAAI*. 2015. URL: http://networkrepository.com.

[6] Jian Tang et al. “LINE”. In: *Proceedings of the 24th International Conference on World Wide Web* (May 2015). DOI: 10.1145/2736277.2741093. URL: http://dx.doi.org/10.1145/2736277.2741093.

6.1 Appendix

6.1.1 Experiment 1: Node Classification on Amherst dataset with 2.3k nodes (node2vec vs DNE)

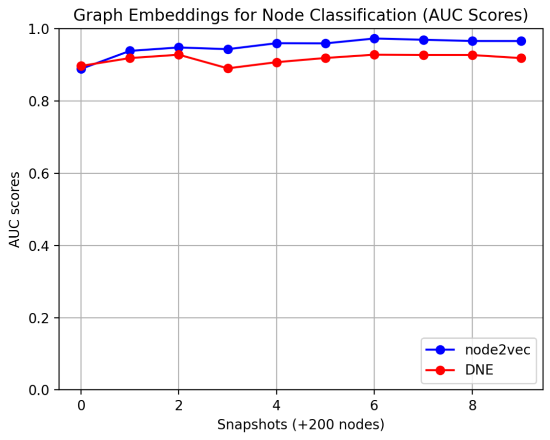

Figure 9: **Node classification on node2vec embeddings vs DNE embeddings**

6.1.2 Experiment 2: Link Prediction on Amherst dataset with 2.3k nodes (node2vec bvs DNE)

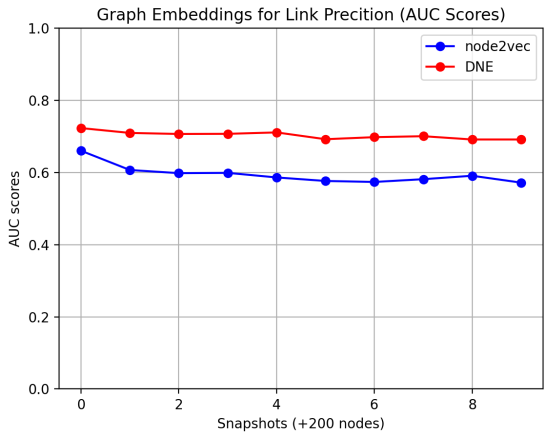

Figure 10: **Link prediction for node2vec embeddings vs Link prediction for DNE embeddings**

6.1.3 Experiment 7: Node Classification on NCI Dataset (120K nodes)

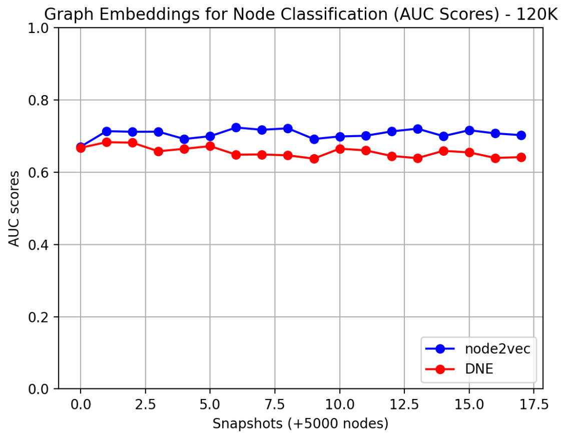

Figure 11: **AUC score for node classification on bigger dataset (100K nodes) - node2vec vs DNE**

6.1.4 Experiment 8a: Node Classification with Deletion

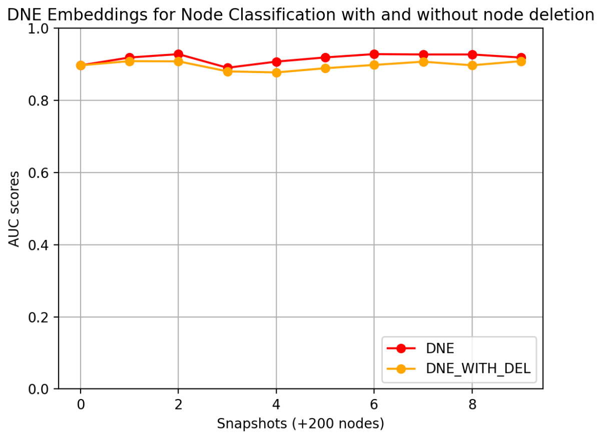

Figure 12: **DNE Embeddings for Node Classification with and without node deletion**

6.1.5 Experiment 8b: Link Prediction with Deletion

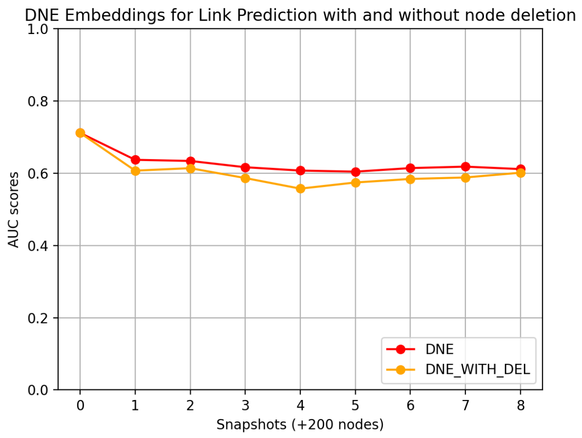

Figure 13: **DNE Embeddings for Link Prediction with and without node deletion**