
# Graph Embedding for Deep Learning

**Data Mining MTL782**
**Assignment 2**

**Aayush Somani** 2017MT10722
**Harsh Kumar** 2017MT10729
**Vivek Muskan** 2017MT10755

# Introduction

There is widespread availability of relational data, representing different relationships between the involved entities, both on the Internet as well as in the physical world, like the social network or biological protein interaction networks. Such data can be represented very well with the graph data structure, with the nodes of the graph representing the entities involved, and the edges of the graph representing various relationships and interactions they might have amongst themselves. This makes graph analysis and algorithms on graphs play a fundamental role in Machine Learning, with applications involving, but not limited to, link prediction, recommendations, anomaly detection, clustering, etc. The interest in applying Machine Learning on graphs is snowballing, but the mathematical and statistical operations which can be performed on graphs are limited, and thus it becomes challenging to perform Machine Learning methods to graphs directly. The principle problem in machine learning on graphs is to find a way to integrate the structure of the graph into the machine learning model. We need to learn some representation of graphs which is acceptable to the Machine Learning algorithms.

Traditional machine learning approaches often rely on summary graph statistics like kernel functions, degree coefficients, or engineered features to measure the neighbourhood structures. But these approaches are limited because they are inflexible, designing them is a computationally expensive process which is time-consuming, and they are not able to adapt and grow during the learning process, and this is where graph embeddings come into the picture.

# Graph Embeddings

Graph embedding is an approach to transform the properties of the graph, i.e. nodes, edges and their features into a lower-dimensional vector space. The embedding learnt should capture various properties of the graph, like the graph topology, vertex-vertex relationships and other relevant information like subgraphs, etc. More properties the embedding is able to encode, the better results it would be able to provide when utilised in any model. The task of generating graph embeddings is particularly tricky because graphs can vary in terms of their scale, specificity and subject.

We can roughly divide embeddings into two groups:

- **Node Embeddings:** In this, we encode each node with it's own vector representation. This is useful in performing predictions and visualizations on the node level, for eg visualization of the nodes on a 2D plane, or prediction of new connections (recommendations) based on node similarities.
- **Graph Embeddings:** In this, we represent the whole graph with a single embedding vector. This is useful when we want to perform predictions and visualizations on the graph level, i.e. compare multiple graphs with each other, for eg comparison of protein graphs of different compounds.

# Need for Graph Embeddings

- **Irregular structures of graphs:** Unlike images, audio or text which have a clear grid structure, graphs have very irregular structures, which makes it very difficult to generalize operations on graphs.
- **Limited availability of ML-DL algorithms on Graphs:** Graphs are composed of edges and nodes, and these network relationships can only use a specific subset of mathematical and statistical algorithms, while vector spaces have a very rich set of approaches.
- **Embeddings are compressed representations:** Graphs are usually represented by an adjacency matrix, which is a |V| x |V| matrix, where |V| is the number of nodes in the graph. For a fairly large graph (for e.g. 1M nodes), using this matrix as a feature space becomes practically impossible. On the other hand, embeddings are more compact and more practical as they compress the properties in a smaller dimensional space.
- **Vector operations are efficient:** Due to the arrival of parallel computing and, vector operations have become much simpler and faster to evaluate than the equivalent operations on graphs.

Embedding methods are a Machine Learning model by themselves, instead, they are a type of algorithm which is used in pre-processing, with the target to turn the graph into a computationally efficient format. Machine Learning algorithms are tuned for continuous data, while a graph is inherently discrete in nature, thus we map the graph into a continuous vector space embedding. There are a variety of ways to go about embedding graphs, but a good approach should satisfy the following conditions:

- The embeddings should well describe the properties of the graph, as the performance of the actual model highly depends on the quality of embeddings used.
- The algorithm which learns the embeddings should be efficient and it should be able to handle large graphs. The size of the network should not drastically slow down the learning process for embeddings.
- The embedding dimensions should be an adequate amount, not too large, not too small. Longer embeddings can preserve more information but in turn, they lead to higher time and space complexity. A trade-off must be made based on the requirements.

There exist many algorithms for generating graph embeddings, we have presented the following:

- DeepWalk
- Node2vec
- SDNE
- LINE

# DeepWalk

**DeepWalk** is the primary algorithm that proposes node embedding learned in an unsupervised manner. It belongs to the group of graph embedding techniques that uses Random walks. **Random Walk** is a technique for obtaining sequences from a graph. It enables the traversal of a graph by going from one node to another, as long as they are attached to a common edge. Fig. 1 shows an example of a random walk of length = 5. We can use these sequences to train a Skip-Gram model to study node embeddings. This entire process is known as a deep walk.

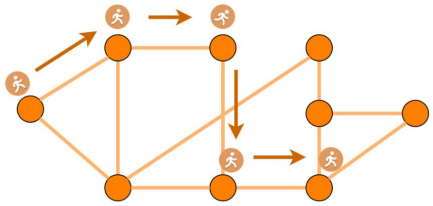


Fig.1: An example of a random walk of length 5 in a network

## The Skip-Gram Model

The goal of training the Skip-Gram model is to find those node representations that will help in predicting the surrounding nodes in the given graph. We are given a corpus and a window size, this model aims to maximize the resemblance of node embeddings of the nodes that occur in the same window. With a sequence of training nodes $w_1, w_2, w_3, \ldots, w_T$, the objective of the Skip-gram model for embeddings is to maximize the average log probability i.e.,

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t)$$

where c is the size of the given training context. The more the value of c will result in more training examples and thus eventually will lead to more accuracy at the expense of our training time. The basic Skip-gram formulation explains $p(wt+j\,|wt)$ using the given softmax function:

$$p(w_O|w_I) = \frac{\exp\left({v'_{w_O}}^{\top} v_{w_I}\right)}{\sum_{w=1}^{W}\exp\left({v'_w}^{\top} v_{w_I}\right)}$$

where $v_w$ and $v'_w$ are the "input" and "output" vector representations of w, and W is the number of nodes in the graph.

If each node in a graph is depicted with an arbitrary representation vector then transversal of the graph is probable. The steps of that traversal could be aggregated by adjusting the node representation vectors next to each other in a matrix. We can then feed that matrix representing the graph to a recurrent neural network.

**Stages of DeepWalk:** DeepWalk is mainly a two-stage method.

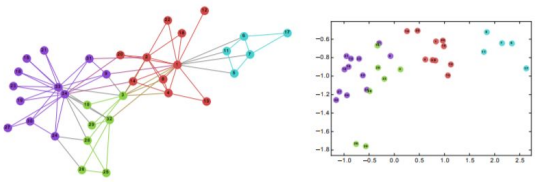
- In the first stage, it traverses the given network with random walks to indicate the regional structures by its neighbourhood relations and in the second stage, it uses an algorithm called Skip-Gram to learn embeddings that are improved by the suggested structures.
- In the next stage, our goal is to discover neighbourhoods in the network. To do so, we fix the number of random walks starting at every node let's say that number to be ***k***. Also, the length of each walk ***(l)*** is pre-assumed. Thus, when this stage is completed, we obtain ***k*** node sequences of length ***l***.

Since the adjacent nodes are similar so they should have similar embeddings. Based on this, those nodes that co-occur in a path as the same nodes will be accepted. This means that the number of co-occurrence in the random walks is a symbol of node similarity. This is a fair hypothesis since edges, by design, are mostly similar or interacting nodes in the network.

## Effect of *k* and *l*:

It is apparent that the effect of ***k*** and ***l*** is significant. The more the value of ***k*** is increased, the more the network is explored since more random walks are generated. On the other hand, when the length ***l*** is increased, paths become larger, and more different nodes are accepted as similar nodes.

Let's show the DeepWalk Graph Neural Network with an example. Suppose we want to train Karate Graph. Different colours indicate different labels in the input graph. We can see that in the output graph (Fig. 2), the nodes that have the same labels are clustered together and most of the nodes with different labels are clustered or separated properly (here the embedding is with 2 dimensions)



(a) Input: Karate Graph (b) Output: Representation

Fig. 2: Karate Graph and the output representation

## Drawback of DeepWalk:

The main drawback of DeepWalk is that it lacks the ability of centralization. Whenever a new node is allowed to come in, it again has to train the model in order to represent this node. Thus, such Graph Neural Network is not suitable for dynamic graphs where the nodes in the graphs come in regularly or those in which the graph is ever-changing.

# Node2Vec

Node2vec is just a modification of DeepWalk and has a small difference in random walks. It also takes the latent embedding of the walks and uses them as an input to a neural network which classifies the nodes. It has two parameters P and Q. Parameter P defines how probable is that the random walk would return to the given previous node, while parameter Q defines how probable it is that the random walk would go to the undiscovered part of the graph. It infers both the communities and complex dependencies. In other words, the parameter P prioritizes a breadth-first-search (BFS) procedure, while the parameter Q prioritizes a depth-first-search (DFS) procedure.

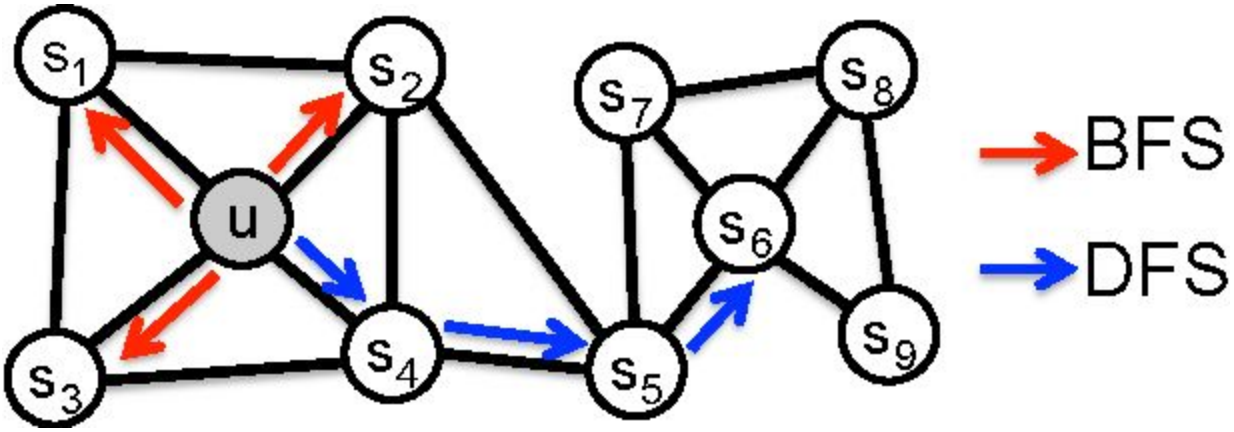



Fig. 3: Walks based on BFS and DFS

As we can see from Fig. 3, BFS is ideal to learn about the local neighbours, while DFS is better in learning about the global variables. Node2vec is able to do both depending on the task. This implies that given a single graph, Node2vec can return different embeddings based on the values of the parameters.

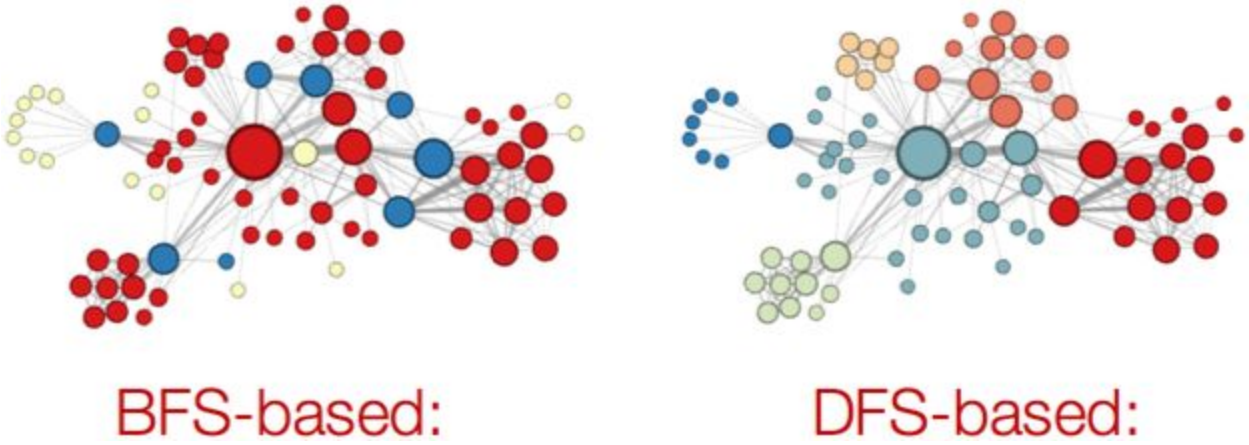



Fig. 4: The output embeddings for a BFS and DFS based walk

# SDNE (Structural Deep Network Embedding)

Unlike DeepWalk and Node2Vec, SDNE does not use random walks. Instead, it uses the metrics First-Order and Second-Order Proximity.

## First-order Proximity

It denotes the local pairwise similarity between two vertices. For every pair of vertices linked by an edge (u, v), the weight on that edge, $w_{uv}$, indicates the first order proximity between u and v. If there isn't an edge between u and v, then their first-order proximity is taken to be 0.

## Second-order Proximity

It denotes the similarity between the neighbourhood network structures of the two vertices (u,v). Mathematically, let us define $p_u = (w_{u,1}, \ldots, w_{u,|V|})$ denote the first-order proximity of the vertex u with all its neighbouring vertices, then the second-order proximity between u and v can be computed using the similarity between $p_u$ and $p_v$. If there is no vertex linked from/to both u and v, the second-order proximity is taken to be 0.

SDNE proposes a semi-supervised deep model to perform network embedding, whose framework is shown below.

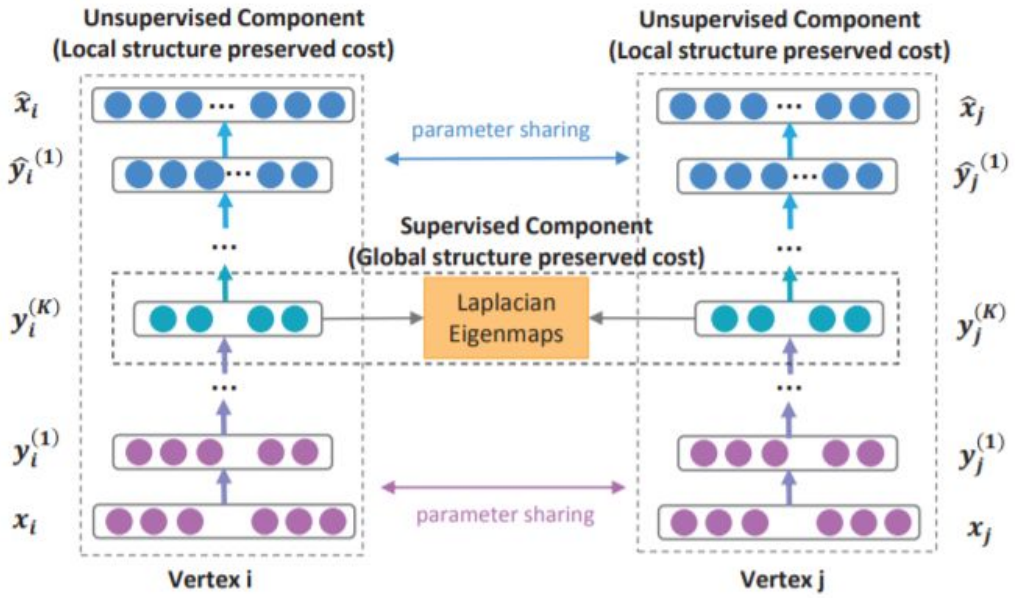



Fig. 5: Framework of SDNE

In detail, to capture the highly non-linear network structure, it proposes a deep architecture, which is composed of multiple nonlinear mapping functions to map the input data to a highly nonlinear latent space to capture the network structure. Furthermore, in order to address the structure-preserving and sparsity problems, it proposes a semi-supervised model to exploit both the second-order and first-order proximity.

It designs the supervised component to exploit the first-order proximity as the supervised information to refine the representations in the latent space. By jointly optimizing them in the proposed semi-supervised deep model, SDNE can preserve the highly-nonlinear local-global network structure well and is robust to sparse networks.

SDNE uses autoencoders to map the high dimensional information into low dimension vector space. It is composed of two parts, i.e. the encoder and decoder. The encoder consists of multiple non-linear functions that map the input data to the representation space. The decoder also consists of multiple non-linear functions mapping the representations in representation space to reconstruction space. Then given the input $\mathbf{x_i}$, the hidden representations for each layer are shown as follows :

**Algorithm 1** Training Algorithm for the semi-supervised deep model of *SDNE*

**Input:** the network $G = (V, E)$ with adjacency matrix $S$, the parameters $\alpha$ and $\nu$

**Output:** Network representations Y and updated Parameters: $\theta$

1: Pretrain the model through deep belief network to obtain the initialized parameters $\theta = \{\theta^{(1)}, ..., \theta^{(K)}\}$
2: $X = S$
3: **repeat**
4: Based on $X$ and $\theta$, apply Eq. 1 to obtain $\hat{X}$ and $Y = Y^K$.
5: $\mathcal{L}_{mix}(X;\theta) = \|(\hat{X} - X) \odot B\|_F^2 + 2\alpha tr(Y^T LY) + \nu \mathcal{L}_{reg}$.
6: Based on Eq. 6, use $\partial L_{mix}/\partial\theta$ to back-propagate through the entire network to get updated parameters $\theta$.
7: **until** converge
8: Obtain the network representations $Y = Y^{(K)}$

$$y^{(1)}_i = \sigma(W^{(1)} x_i + b^{(1)})$$

$$y^{(k)}_i = \sigma(W^{(k)} y^{(k-1)}_i + b^{(k)})$$

SDNE defines its loss function in three parts:

$$\mathcal{L}_{2nd} = || (\widehat{X} - X) \odot B ||^2_F$$

$$\mathcal{L}_{1st} = \sum_{i,j=1}^{n} s_{i,j} || (\widehat{y}^{(K)}_i - y^{(K)}_j) ||^2_2$$

$$\mathcal{L}_{reg} = \frac{1}{2} \sum_{k=1}^{K} ( || W^{(K)} ||^2_F - || \widehat{W}^{(K)} ||^2_F )$$

The final loss function is the sum of the above three. $\mathcal{L}_{2nd}$ corresponds to 2nd order proximity, $\mathcal{L}_{1st}$ for 1st order proximity and $\mathcal{L}_{reg}$ is just regularization factor. After formulation of the above loss function, it uses a given algorithm to train this semi-supervised deep model.

# LINE (Large-scale Information Network Embedding)

LINE model also works using the first-order and second-order proximity, in a way that it both preserves them separately, and then it provides a simple way to combine both of them.

## First-Order proximity

To model the first-order proximity, for each undirected edge (i, j), we define the joint probability between vertex $v_i$ and $v_j$ as follows:

$$p_1(v_i , v_j) = 1/( 1+ exp(-u_i^T . u_j))$$

To preserve the first-order proximity, a straightforward way is to minimize the following objective function:

$$O_1 = d(\hat{p}_1(. , .), p_1(. , .)) \quad \Rightarrow O_1 = - \sum_{(i,j) \varepsilon E} w_{ij} \log p_1(v_i , v_j)$$

Where $d(. , .)$ is KL-divergence.

## Second-Order proximity

The second-order proximity is applicable for both directed and undirected graphs. The second-order proximity assumes that vertices sharing many connections to other vertices are similar to each other. Let us introduce two vectors $u_i$ and $u_i^T$ , where $u_i$ is the representation of vi when it is treated as a vertex while $u_i^T$ is the representation of $v_i$ when it is treated as a specific "context". For each directed edge (i, j), we first define the probability of "context" $v_j$ generated by vertex $v_i$ as:

$$p_2(v_j \mid v_i) = exp(u_j^T . u_i) / \sum_{k=1}^{|V|} exp(u_j^T . u_i)$$

To preserve the second-order proximity, it makes the conditional distribution of the contexts $p_2(\cdot|v_i)$ specified by the low-dimensional representation be close to the empirical distribution $\hat{p}_2(\cdot|v_i)$. Therefore, we minimize the following objective function:

$$O_2 = - \sum_{i \varepsilon V} \lambda_i d( \hat{p}_2(\cdot|v_i) , p_2(\cdot|v_i) ) \quad \Rightarrow \quad O_2 = - \sum_{(i,j) \varepsilon E} w_{ij} \log p_2(v_i \mid v_j)$$

The objective function for LINE is the sum of above two objective functions defined for first and second-order proximity respectively.

Although, both SDNE & LINE preserves local & global network structures of graphs using first and second-order proximity respectively, the key difference between them is that:

- SDNE primarily focuses on capturing the highly non-linear network structure
- LINE proposes a graph embedding method that is scalable to real-world information

# References


- Structural Deep Network Embedding: Daixin Wang, Peng Cui, Wenwu Zhu, Tsinghua National Laboratory for Information Science and Technology
- Distributed Representations of Words and Phrases and their Compositionality: Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean
- A neural probabilistic language model: Yoshua Bengio, R´ejean Ducharme, Pascal Vincent, and Christian Janvin, The Journal of Machine Learning Research, 3:1137–1155, 2003.
- DeepWalk, Online Learning of Social Representations: Bryan Perozzi, Rami Al-Rfou, Steven Skiena, 27 Jun 2014
- Representation Learning on Graphs: Methods and Applications: William L. Hamilton, Rex Ying, Jure Leskovec, Department of Computer Science Stanford University
- Deep Learning on Graphs: A Survey Ziwei Zhang, Peng Cui and Wenwu Zhu
- Representation Learning on Graphs: Methods and Applications: William L. Hamilton, Rex Ying, Jure Leskovec, Department of Computer Science, Stanford University
- Learning graph representations with global structural information: S. Cao, W. Lu, and Q. Xu. Grarep
- Innovations in Graph Representation Learning: Alessandro Epasto, Senior Research Scientist and Bryan Perozzi, Senior Research Scientist, Graph Mining Team https://ai.googleblog.com/2019/06/innovations-in-graph-representation.html
- Graph Learning and Geometric Deep Learning: Falwnson Tong https://towardsdatascience.com/overview-of-deep-learning-on-graph-embeddings-4305c10ad4a4
- Graph Embeddings: Primož Godec https://towardsdatascience.com/graph-embeddings-the-summary-cc6075aba007