EasyChair Preprint

№ 4583

Representation Learning on Graphs - A Survey

Ankur Sharma, Mehak Preet Dhaliwal and Kartikeya Sharma

November 16, 2020

Representation Learning on Graphs - A Survey

Ankur Sharma*
Indian Institute of Technology, Delhi
2015CS50278

Mehak Preet Dhaliwal*
Indian Institute of Technology, Delhi
2015CS10238

Kartikeya Sharma*
Indian Institute of Technology, Delhi
2015CS10234

ABSTRACT

Learning methods to represent graph nodes as feature vectors is a field that has recently seen a surge in research. Embedding graph nodes as vectors is useful to make graph datasets suitable for use in several downstream machine learning tasks. In this survey, we attempt to present an overview of the various methods found in the literature.

KEYWORDS

Graphs, Node Embeddings, Representation Learning, Survey

1 INTRODUCTION

Graphs have become a ubiquitous data structure not only in computer science but also in several other fields, since they can be used to model a plethora of real-world phenomena. From the textbook example of road networks with physical nodes and edges, to the more abstract social networks, and to recent applications such as modelling protein-protein interaction networks in biology, graphs have innumerable applications in countless systems. Graphs can be thought of as structured repositories of data, capturing relational knowledge between various entities and allowing it to be easily and efficiently accessed and modified. Searching in such graph databases is faster and more efficient in terms of number of lookups and memory usage.

Apart from being structured databases, graphs also capture several inherent relationships between the entities represented by their nodes, such as internode similarity and community structure. With the increasing amount of data available in today's world and with a large amount of it being stored in graph databases, graphs have also proven amenable to modern methods of data analysis using machine learning techniques. Such analysis is used in applications such as link prediction for friend suggestion in social networks, making product recommendations to customers on an e-commerce website and so on.

A longstanding challenge for using graph data in machine learning models has been to find a way to incorporate information about structure into the model. As an example, one might want a feature vector to encode relationships between a node and its neighbours, such as the number of common friends in the case of a social network. Until recently, these problems used to be dealt with by manually designing features such as summary graph statistics (number of 1-hop neighbours, degrees), kernel functions and so on.

A recent way to tackle this problem has been to learn these feature vector representations from data. That is, the step which was previously treated as part of pre-processing data before using it in a machine learning model, is being replaced by a machine learning problem in itself.

2 MOTIVATION

Formal network analysis is usually done by modelling the network as a graph and analysing its properties. As stated previously, problems such as link prediction, for example, predicting edges between users in a social network, require analysing the pairwise similarity between nodes in a graph. Node classification tasks such as predicting functional labels of proteins, require analysing node structures and properties.

The first challenge in performing any analysis on graphs is finding an effective representation. The goal of such an endeavour is to represent networks in a concise manner, so that analysis can be performed efficiently on them. Traditional methods of graph representation include the adjacency list or adjacency matrix representations. Such representations suffer from several issues such as high computational complexity, high dimensionality of representation, low parallelizability, and inability to apply machine learning methods, most of which require data to be in the form of feature vectors. Mapping nodes of the graph to a low dimensional vector space serves to eliminate these problems. Distance or similarity functions defined over node embeddings can directly be used for analysing node properties, thus reducing computation time substantially. These representations can also be directly fed into traditional machine learning and deep learning models to be trained on downstream tasks such as link prediction, classification and clustering, thus widening the domain of tasks that can be accomplished through graph analysis significantly.

Simple approaches such as hand engineering node features is a time consuming and tedious process and one cannot expect such features to generalise well or be scalable to large graphs. For example, one could consider the feature vector of a node to be the corresponding row in the graph's adjacency matrix. Thus, all nodes would be represented by a feature vector with dimensionality as high as the number of nodes in the graph. Clearly such an approach would not scale to modern graphs such as social networks or the internet. Moreover, the feature vectors would be very sparse, too large and contain redundant and noisy information making them unsuitable for learning tasks. Another problem with this representation is that it loses semantic information. One would expect each orthogonal dimension in a feature vector to be semantically very different from another, but a traditional representation obviously does not take this into account.

Due to such issues, it becomes crucial to define methodologies for learning representations of nodes of graphs in the form of low dimensional feature vectors, preserving structural and other task specific properties of the network. Figure 1 shows an example embedding of the Zachary club Karate network generated using the DeepWalk method.

*These authors have contributed equally to the paper.

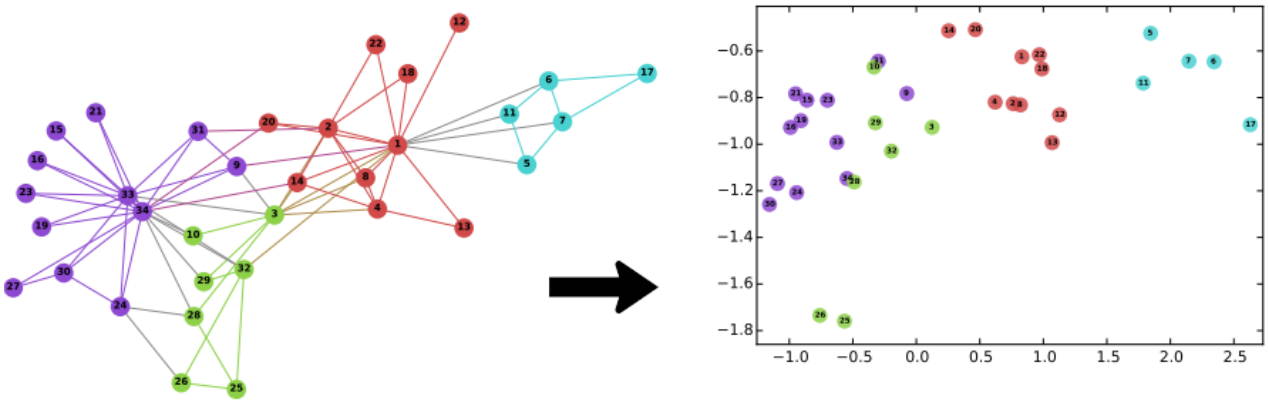

Figure 1: Example of a node embedding generated using the DeepWalk method

3 PROBLEM FORMULATION

The network embedding problem, takes as input a graph G represented by a vertex set V and edge set E, and optionally a matrix of node attributes X. It then outputs a mapping from the vertex set V to a $d-$dimensional Euclidean space $\mathbb{R}^d$

The goal of such an embedding is two-fold: firstly, the graph should be able to be reconstructed back from the embedding. An edge between two nodes should correspond to a relatively low Euclidean distance between the feature vectors of these nodes. On the other hand, the embedding should be amenable to inferring network information such as degrees and neighbours, and performing analysis such as link prediction, node classification and clustering. Finally, the embedding dimensionality$-d$ should be much lesser than the number of nodes in the graph G.

More formally, an embedding E is a mapping: $E : G, X \rightarrow \mathbb{R}^d$ where X and G are defined previously.

4 NODE EMBEDDING (RELATED WORK)

We begin by presenting an overview of techniques for node embedding. We classify methods according to the tree structure shown in Figure 2. Most methods review fall into the category of unsupervised embedding methods, where information regarding the downstream machine learning task for which the embedding is performed is not included during the embedding stage. We divide the embedding methods into shallow and deep embedding methods. Shallow methods are further subdivided into methods based on Matrix Factorization (MF) and those based on Random Walks (RW). Deep embedding methods follow either the Autoencoder approach, or the Aggregator approach.

There are lines of work which we do not review here for reasons of brevity, such as statistical relational learning embedding methods, geometric deep learning and manifold learning algorithms.

4.1 Shallow Embedding Methods

Shallow embedding methods are so-called because the embedding here simply involves a matrix *lookup*. In other words, the

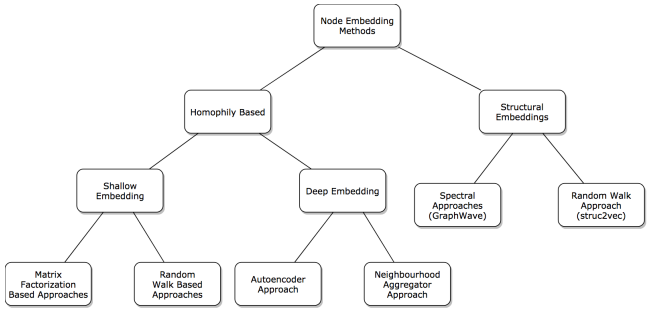

Figure 2: Classification of various Node Embedding Methods

embedding of a node is given by:

$$EMBED\,(V_i) = \mathbf{Z}\mathbf{v_i},$$

where Z is a matrix containing the embedding vectors for all nodes and v_i is an indicator vector used to select a particular column from Z. Here, the matrix Z is optimized during the embedding process. These algorithms are inspired by techniques for dimensionality reduction in high dimensional spaces, such as principal component analysis.

4.1.1 ***Matrix Factorization (MF) Approaches.*** Graphs are often represented using adjacency matrices, where a node is represented by each column and node. An edge between nodes is represented by the value at the intersection of the corresponding row and column. Matrix factorization approaches aim to embed graphs by performing standard dimensionality reduction methods on the adjacency matrix. Here, singular value decomposition (SVD) and non-negative matrix factorization (NMF) are often used.

Some algorithms based on this approach are presented below:

(1) **Laplacian Eigenmaps**[2]
This technique uses spectral graph methods to perform dimensionality reduction. It is based on assuming that the adjacency matrix lies in a low-dimensional manifold in a high-dimensional space. It begins by constructing a graph using

epsilon-neighborhoods or K nearest neighbors. Thus, the connectivity of the nodes depends on how close the neighbouring points are.

Then the heat kernel (Berline et al, ADDREF) is utilized to choose the weight W_{ij} of nodes i and j in the graph. Finally, the representation u_i of node i can be obtained by minimizing a loss function which depends on the squared distance between the embedding vectors. The aggregate loss function is the sum of these squared distances (which are further weighted), across all pairs of vertices. In other words, a cost function based on the graph is minimized, which ensures that points close to each other are mapped close together even in the low-dimensional space, thus preserving local distances.

The remaining three methods build on Laplacian eigenmaps, these build on a cost function that depends on a pairwise dot-product of the node embedding vectors $\mathbf{z_i^T z_j}$, and a pairwise node similarity function. These algorithms differ on the similarity function used.

(2) **Graph Factorization, GraREP, HOPE**[1][4][11]
These three algorithms are based on minimising a loss function:

$$\sum_{(v_i,v_j)\in V} ||z_i^T z_j - s_G(v_i, v_j)||_2^2,$$

where $s_G(v_i, v_j)$ is the similarity function. Graph Factorization defines the similarity function of two nodes as the corresponding element in the adjacency matrix, i.e., $s_G(v_i, v_j) = A_{ij}$. GraREP generalizes this notion to other powers of the adjacency matrix (for example, A_{ij}^2). The HOPE algorithm allows for more flexible definitions of the similarity function. For instance, it can use the Jaccard neighbourhood overlaps or another similarity measure. ADDREF:Survey

It is important to note that there is a fine balance between giving more weight to first-order similarity, such as in Graph Factorization, and higher order similarity such as in GraREP. First-order similarity would be more efficient at representing connections between nodes directly due to the use of the adjacency matrix, where as higher powers of this matrix would represent information from multiple-hop neighbours.

(3) **Isomap, Locally Linear Embedding and Other Methods**[17][15]
The IsoMap technique, similar to Laplacian Eigenmaps, first constructs a similarity graph using connectivity algorithms such as K-nearest neighbours. Then a matrix D_G of shortest path distances for every pair of vertices is calculated. Then, any dimensionality reduction technique, for example multidimensionality scaling (MDS) can be used to compute the coordinate vector by minimizing the squared loss of the difference between the shortest path distance and the L2 norm of the difference between the embedding vectors.

IsoMap suffers from the high complexity of computing all the inter-vertex distances. Locally linear embedding, eliminates the need to calculate these distances, and uses a different loss function.

(4) **LINE (Large-Scale Information Network Embedding**[16]
This algorithm is not exactly based on random walks, although due to its similarity with node2vec and DeepWalk in terms of its measure of similarity between nodes, it is often discussed in the same context.

LINE models two kinds of proximities between nodes explicitly, the so-called *first-order* and *second-order* proximities and minimises a loss function that takes into account both of these.

It determines the first-order proximity as

$$p_1(v_i, v_j) = \frac{1}{1 + exp(-\mathbf{v_i}^T \mathbf{v_j})}$$

Where $\mathbf{v_i}$ and $\mathbf{v_j}$ are the embedding vectors. The empirical probability of the same is defined as $\hat{p_1}(i, j) = \frac{w_{ij}}{\sum_{i,j} w_{ij}}$ where w represents edge-weight.

The way to optimise the first order proximity is the minimise the KL divergence between the two probability distributions p_1 and $\hat{p_1}$. Doing so reduces the loss function to a cross entropy loss

$$O_1 = -\sum_{(i,j)\in E} w_{ij} log\, p_1(\mathbf{v}_i, \mathbf{v}_j)$$

LINE also defines a *second-order* proximity using the skip-gram model similar to the word2vec algorithm in natural language processing. For those unfamiliar, this proximity appears as:

$$p_w(v_j|v_i) = \frac{exp(\mathbf{u'}_j^T.\mathbf{u}_i)}{\sum_{k=1}^{|V|} exp(\mathbf{u'}_k^T.\mathbf{u}_i)}$$

Here u_i is the embedding of a node, and u_i' is the embedding when it is treated as a *context*. Here also, the optimization is done by minimizing the KL-divergence between this and the empirical distribution (however, here it is done for the conditional distributions, and also weighted by a "prestige" value of the vertex measured by either degree or through some other methods such as PageRank). The empirical distribution in this case has similar structure to that of the first-order proximity, however the denominator is now the out-degree of the vertex i, i.e., $\sum_{k\in neighbours(i)} w_{ik}$. This also reduces to the cross entropy loss

$$O_2 = -\sum_{(i,j)\in E} w_{ij} log\, p_2(\mathbf{v}_j|\mathbf{v}_i)$$

By learning the vectors $\mathbf{u}_i$ and $\mathbf{u'}_i$ that minimize this objective function, the graph is reduced to a lower dimensional space.

4.1.2 ***Random Walk Approaches.*** All approaches based on Matrix Factorization, as defined in the previous section, used a similarity measure which was deterministic. This adds another hyperparameter to the embedding approach and reduces its flexibility. The advantage of random walk methods is to replace this deterministic similarity measure by a measure that can be obtained using the graph structure. These methods thus depend on approximating the similarity measure using random walk statistics. They resulted in significant gains in embedding quality, and spawned a whole host of algorithms such as node2vec. So these techniques make sure that nodes have alike embeddings if they all occur on random walks in the graph. One should note that this approach is more flexible and stochastic.

(1) **node2Vec, DeepWalk**[7][12]
Both these algorithms use a similarity measure which is equal

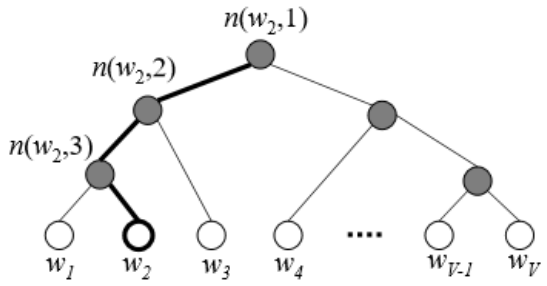

Figure 3: Binary Tree Data Structure used in Hierarchical Softmax Computation

to $P_{G,k}(v_j|v_i)$, the probability of visiting node v_j on a length k random-walk starting from node v_i. Both of them also use the softmax function on the inner product between two node embeddings as the definition of this probability, i.e

$$P_{G,k}(v_j|v_i) = \frac{e^{z_i^T z_j}}{\sum_{k \in V} e^{z_i^T z_k}}$$

However, computing this for every pair of nodes is too expensive, due to the denominator term. Both of these also try to minimize a cross entropy loss:

$$\sum_{(v_i,v_j) \in T} -log(P_{G,k}(v_j|v_i))$$

where the training set T is generated by simulating random walks starting from each node. However, they differ in the way they approximate this probability. node2vec uses the technique of negative sampling to approximate the denominator, whereas DeepWalk uses a hierarchical softmax function to do so. This results in a great improvement in computational complexity of these methods. For example, hierarchical softmax improves the order of computation from $O(|V|)$ to $O(log_2(|V|))$, where $|V|$ is the size of the vocabulary.

Hierarchical Softmax uses a binary tree structure to accelerate the computation of the softmax function. Each word can be traced by following a path down the root of the three, the nodes along which represent the probability mass contribution along that way. These masses are calculated by a simple sigmoid function. The idea is that the probability we are going to calculate (the final softmax) is the product of the probability masses down this path. This is represented in Figure 3, where where $n(w, j)$ is the j-th node on the path from the root to w.

Negative sampling is based on the idea of approximating the softmax using a few 'negative' samples from tha data. It is also similar to the optimization technique stochastic gradient descent - instead of changing all of the weights, a few of them are updated dramatically speeding up the training process. These negative samples are sampled from a ceratin probability distribution, for example the noise distribution. This distribution can be tuned as per the data to give the requisite results.

They also differ in that node2vec allows a less constrained definition of what constitutes a random walk. Whereas DeepWalk uses a simple random walk, node2vec can adjust the probability p of visiting a neighbour node and the probability q of revisiting the same node at any timestep. By doing so, it can focus more on the global structure, allowing less revisiting of the same node, or on the local structure, allowing more revisiting by tuning the parameters that represent this probability.

(2) **HARP**[5]
This method adds a preprocessing stage before peforming random-walks in approaches such as node2vec and DeepWalk. A procedure to coarsen the input graph is applied to collapse nodes into supernodes, then the random walk algorithm is used to learn embeddings, and the learned embedding of the supernode is used as the initial embedding for the random walk algorithm for its constituent nodes in the successive iteration. This can be repeated depending on the number of coarsening steps, and has been shown to improve performance in practice.

There also exist extensions to the idea of random walks, by learning encodings of nodes that, say, jump over multiple nodes. Sometimes they involve a change in the distance measure to a hyperbolic one.

4.2 Deep Embedding Methods

The methods described in the previous section involved the embedding vector being determined through a simple embedding lookup. This approach has a few limitations, such as:

- There is no sharing of information between the embedding vectors, This can be quite inefficient, as it has two drawbacks: the number of parameters grows linearly with the number of nodes, which clearly does not scale to large graphs. This is because parameters are not shared by the encoded vectors.
- Only the structural information in the graph is used for calculating the encodings, and the node attribute information is not used. Sometimes node attributes are extremely important in deciding similarity between nodes, for example in social networks the user's profile information can be used to measure similarity.
- Nodes which are not observed during the training phase cannot be generated embeddings for. This is not useful in scenarios where there is a separate test data set, the properties of which are supposed to be predicted by generalizing information learnt from a training data set. In situations such as temporally evolving graphs, graphs which are too large to fit inside main memory, these methods fail to provide much utility.

Thus, a new paradigm of methods have been proposed to address these issues, which we present here as deep embedding methods. These methods differ in that they use much more intricate encoding techniques, often involving deep autoencoders or neighbourhood aggregation techniques. We further subdivide deep embedding methods into methods that use autoencoders and those that use neighbourhood aggregation techniques. Both of these utilise node attributes along with the structure of the graph.

4.2.1 ***Neighbourhood Autoencoder Techniques.*** The reader would have noticed that the previously described techniques use a loss function that is a aggregation (sum) of terms, each of which uses information from a pair of nodes (for example, the term $\mathbf{z}_i^T \mathbf{z}_j$). The techniques discussed in this subsection utilize a loss function which

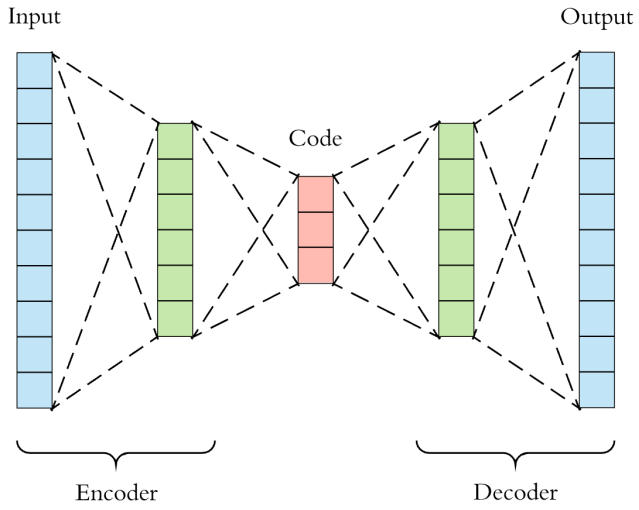

Figure 4: An Autoencoder network

aggregates terms that utilize information from a single embedding instead.

These techniques utilize deep autoencoders to generate the node embeddings. These are deep neural networks which are shown in Figure 4. An autoencoder is a type of artificial neural network that is used to reduce the dimensionality of its input, which generates a low dimensional vector in its middle layer, and then use this low-dimensional vector to reconstruct the original input. It then learns to efficiently compress data such that it can be reconstructed from its encoding. Here, $ENC(\mathbf{v}_i)$ represents a function that encodes a node v_i and $DEC(\mathbf{z}_i)$ decodes the corresponding embedding z_i

The loss function for these autoencoder methods takes the form $||DEC(ENC(\mathbf{s}_i)) - \mathbf{s}_i||_2^2$, where $\mathbf{s}_i$ is a similarity vector that replaces the similarity function described in earlier sections. This contains the similarity of a node with respect to all other nodes in the graph. Structural Deep Network Embeddings (SDNE)[18] and Deep Graph Neural Representations (DGNR)[3] both use the autoencoder technique, but differ in the similarity function that they implement. SDNE uses the corresponding adjacency matrix row as the similarity vector, whereas DGNR uses statistics derived from random walks similar to node2vec and DeepWalk.

The fact that the encoding utilizes the similarity vector of a node with respect to other nodes allows these methods to incorporate neighbourhood information into the embeddings, thus differing from the previously described shallow embedding methods. However, there are still drawbacks: they cannot still cope with evolving graphs. They also cannot generate embeddings for previously unseen nodes. Also the input dimension is fixed. They also do not take into account the node attributes. Thus the only improvement they offer (from a feature perspective) is to offer a more complex encoding scheme, using deep neural networks for the same.

4.2.2 ***Neighbourhood Aggregation Techniques.*** The techniques discussed in this subsection are the first ones to utilize node attributes in addition to the structural information present in the graph.

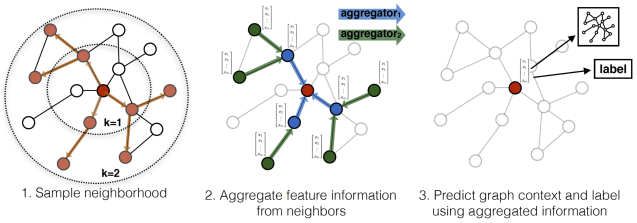

Figure 5: A diagrammatic view of how Neighbourhood Aggregation Techniques work

For example, in a protein-protein interaction network, there might be information about molecular markers within each node, or within a social network there is data about the user's profile within the user's node. These methods aggregate this information to inform their embeddings. In cases where this information is not available or is not sufficient, they can be artificially introduced - for example the degree of a node, the number of one-hop neighbours, etc., but they are more useful when this information is already present and needs utilization. They are also called graph convolutional methods, as they often use the neighbouring node information and aggregate it, similar to kernels in convolutional neural networks. This is shown in Figure 5

The GraphSAGE (Graph SAmple and aggreGAte)[8] algorithm, described in Figure 6, was the first algorithm that made an effort in this direction. The intuition behind this algorithm is to combine information from the local neighbours of a graph, and repeat this iteratively so that a node incrementally gains more and more information from further nodes in the graph. An aggregator function is provided which aggregates information from a node's local neighbours. This is followed by a concatenation step, in which information from the previous iteration's embedding is combined with the aggregator output, and this is passed through a single layer in a neural network. This is repeated for a number of iterations (equal to neural network layers) and the algorithm then outputs the learned embedding. The neural network weights are learned by stochastic gradient descent on a dataset generated by simulating random walks from the nodes. The algorithm, as described in the original paper, also uses the technique of negative sampling. The node representations fed into the loss function are generated from the node attributes themselves, rather than an embedding lookup. Interestingly, this approach can also be used in supervised learning settings, where the unsupervised loss function, can be replaced by a task specific objective. In this sense, GraphSAGE is more of a meta-algorithm or an algorithmic framework. Various aggregator functions, such as a simple mean aggregator, a more complicated max-pooling aggregator, or an even more complex LSTM aggregator were explored by the authors.

Other techniques such as graph convolutional networks (GCN)[10] and Column Networks[13] use the same framework as GraphSAGE, but they differ primarily in terms of what aggregation and concatenation function they use. GCNs and column networks use a weighted sum concatenation function and an element-wise mean aggregator function. In addition, column networks also add a scalar parameter in the aggregator function that allows control of how

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\{\mathbf{W}^k, \forall k \in [1, K]\}$; non-linearity σ; differentiable aggregator functions $\{\text{AGGREGATE}_k, \forall k \in [1, K]\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output: Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{COMBINE}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \text{NORMALIZE}(\mathbf{h}_v^k), \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Figure 6: The framework of all neighbourhood aggregation algorithms

much information to retain from the previous iteration in the next iteration's aggregated embedding. This allows more fine-grained control to retain more/less local information about a node.

These neighborhood aggregator methods have shown consistent performance gains in applications such as link predictions over their shallow embedding counterparts. They also address the issues discussed at the start of this section - they utilize neighbor attributes, they share information across embeddings via the aggregator and concatenation functions and the use of complex neural encoders, the number of parameters does not grow with the number of vertices and they can generate embeddings for previously unseen nodes This comes at the cost of much more tunable hyper-parameters in terms of the neural network itself, and the aggregator and concatenation functions.

5 SUPERVISED EMBEDDING METHODS

All of the methods discussed earlier are unsupervised methods, in the sense that they do not utilize information about the downstream machine learning task while generating the embeddings. In particular, the neighbourhood aggregation techniques discussed earlier often make use of a loss function tailored to the specific downstream machine learning task at hand.

For example, if the downstream task is a binary classification task with node labels given per node, we could use the generated embedding vectors in a logistic regression classifier, and then compute the cross entropy loss on the training samples with the predicted and actual class labels.

The gradient of this cross entropy loss can be used to optimise the encoder by means of backpropagation. This can also completely replace the loss function used in the decoder part of the autoencoder.

6 STRUCTURAL EMBEDDINGS

All the methods discussed until now rely on the basic assumption of *homophily*: If two nodes have similar contexts, they are likely to be similar and thus must have similar embeddings. All embedding methods, whether shallow or deep, have relied on this assumption. In practice, this often holds true, similar nodes are clustered together. However, there are instances where structurally identical nodes are present far apart, e.g. Figure 7. For example, consider an air traffic network, where different nodes representing cities might have similar functions though far apart in a graph, for example as air traffic hubs. In this scenario, methods such as DeepWalk and node2vec, or for that matter even neighbourhood aggregation methods will not generate similar embeddings for these nodes, as they are far apart in the network. An embedding which generates similar encodings for structurally similar nodes is called a structural embedding. The RolX algorithm uses hand designed features to measure structural similarity, whereas the struc2vec and GraphWave algorithms use representation learning techniques to learn the embeddings. The result of the GraphWave algorithm is shown in Figure 7.

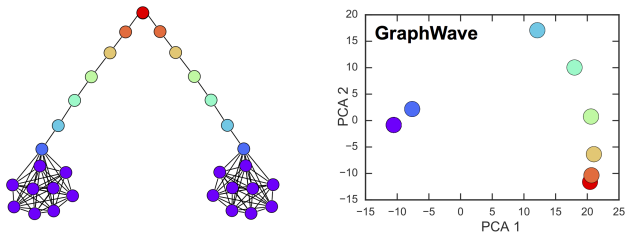

Figure 7: Artificial Barbell Graph. The purple coloured nodes are structurally similar though far apart in the graph. On the right is the result of running the GraphWave spectral embedding algorithm on this graph

Type	Method	Decoder	Similarity measure	Loss function (ℓ)
Matrix factorization	Laplacian Eigenmaps	$\|\mathbf{z}_i - \mathbf{z}_j\|_2^2$	general	$\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) \cdot s_{\mathcal{G}}(v_i, v_j)$
	Graph Factorization	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}$	$\|\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j)\|_2^2$
	GraRep	$\mathbf{z}_i^\top \mathbf{z}_j$	$\mathbf{A}_{i,j}, \mathbf{A}_{i,j}^2, ..., \mathbf{A}_{i,j}^k$	$\|\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j)\|_2^2$
	HOPE	$\mathbf{z}_i^\top \mathbf{z}_j$	general	$\|\text{DEC}(\mathbf{z}_i, \mathbf{z}_j) - s_{\mathcal{G}}(v_i, v_j)\|_2^2$
Random walk	DeepWalk	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v_j \mid v_i)$	$-s_{\mathcal{G}}(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$
	node2vec	$\frac{e^{\mathbf{z}_i^\top \mathbf{z}_j}}{\sum_{k \in V} e^{\mathbf{z}_i^\top \mathbf{z}_k}}$	$p_{\mathcal{G}}(v_j \mid v_i)$ (biased)	$-s_{\mathcal{G}}(v_i, v_j) \log(\text{DEC}(\mathbf{z}_i, \mathbf{z}_j))$

Figure 8: Shallow embedding methods as manifestations of the unified framework proposed by Hamilton et. al

struc2vec[14] was the first algorithm that aimed to embed nodes according to their structural identity. It aims to measure this similarity without taking into consideration node and edge attributes, or for that matter even node position in the network. This approach also does not require the network to be connected. It also proposes a hierarchical way to measure this similarity. The method also generates a random context for each node. Two nodes that frequently appear with similar contexts will have the same structure.

The algorithm first determines structural similarility between pairs of vertices for different sizes of the neighbourhood windows, to establish the hierarchy described in the previous paragraph. It then constructs a multilayer graph, which contains every node in each layer, with different layers corresponding to a different level in the hierarchy. It uses a biased random walk similar to node2vec to generate node sequences for the training data, and then applies a technique to learn latent representations such as the skip-gram model. It thus uses the outputs of these biased random walk sequences to optimize a loss function used in node2vec.

There are also spectral graph techniques such as GraphWave[6], which use the heat kernel of a graph to encode structural information effectively.

7 UNIFIED FRAMEWORK (ADAPTED FROM [9])

Hamilton et al. propose a unified encoder-decoder approach to summarize the various node embedding methods discussed until now. They state that there is a need to unify the various notations developed in the literature.

This framework has 4 key components:

(1) A pairwise similarity function $s_G : V \times V \mapsto \mathbb{R}^+$, defined over the graph G. This takes as input two nodes from the graph and outputs their similarity.
(2) A function *ENC* to generate the node embeddings.
(3) A function *DEC* to recover the similarity values fro mthe generated embeddings. This usually takes as input two embeddings and outputs a pairwise similarity score.
(4) A cost function c to measure how the reconstructions are evaluated, i.e., how close the output of the *DEC* function is to the ground truth similarity value, given by:

$$\mathcal{L} = \sum_{(v_i, v_j) \in V} c(DEC(\mathbf{z}_i, \mathbf{z}_j), s_G(v_i, v_j))$$

Figure 8 shows how a few shallow embedding approaches can be represented using this framework.

8 CONCLUSION

Throughout this paper, we have tried to present an overview of the various techniques available in the literature on representation learning in graphs. These techniques have tremendously improved performance of algorithms in tasks such as link prediction and node classification. However, there is a lot of future work to be done, in the direction of improving the performance of these existing methods, improving on their limitations, and also developing a unified theoretical framework that explains why these methods work the way they do.

REFERENCES

[1] Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy, Vanja Josifovski, and Alexander J. Smola. 2013. Distributed Large-Scale Natural Graph Factorization. In *Proceedings of the 22nd International Conference on World Wide Web (WWW '13)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/2488388.2488393

[2] Mikhail Belkin and Partha Niyogi. 2003. Laplacian Eigenmaps for Dimensionality Reduction and Data Representation. (2003), 1373–1396.

[3] Shaosheng Cao. 2016. deep neural network for learning graph representations.

[4] Shaosheng Cao, Wei Lu, and Qiongkai Xu. 2015. GraRep: Learning Graph Representations with Global Structural Information. In *Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (CIKM '15)*. Association for Computing Machinery, New York, NY, USA, 891–900. https://doi.org/10.1145/2806416.2806512

[5] Haochen Chen, Bryan Perozzi, Yifan Hu, and Steven Skiena. 2017. HARP: Hierarchical Representation Learning for Networks. (06 2017).

[6] Claire Donnat, Marinka Zitnik, David Hallac, and Jure Leskovec. 2017. Spectral Graph Wavelets for Structural Role Similarity in Networks. (10 2017).

[7] Aditya Grover and Jure Leskovec. 2016. Node2vec: Scalable Feature Learning for Networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 855–864. https://doi.org/10.1145/2939672.2939754

[8] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *CoRR* abs/1706.02216 (2017). arXiv:1706.02216 http://arxiv.org/abs/1706.02216

[9] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Representation Learning on Graphs: Methods and Applications. *CoRR* abs/1709.05584 (2017). arXiv:1709.05584 http://arxiv.org/abs/1709.05584

[10] Thomas Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. (09 2016).

[11] Mingdong Ou, Peng Cui, Jian Pei, Ziwei Zhang, and Wenwu Zhu. 2016. Asymmetric Transitivity Preserving Graph Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1105–1114. https://doi.org/10.1145/2939672.2939751

[12] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. 2014. DeepWalk: Online Learning of Social Representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '14)*. Association for Computing Machinery, New York, NY, USA, 701–710. https://doi.org/10.1145/2623330.2623732

[13] Trang Pham, Truyen Tran, Dinh Phung, and Svetha Venkatesh. 2017. Column Networks for Collective Classification. *AAAI* (01 2017).

[14] Leonardo Ribeiro, Pedro Saverese, and Daniel Figueiredo. 2017. struc2vec: Learning Node Representations from Structural Identity. 385–394. https://doi.org/10.1145/3097983.3098061

[15] Sam T. Roweis and Lawrence K. Saul. 2000. Nonlinear dimensionality reduction by locally linear embedding. *SCIENCE* 290 (2000), 2323–2326.

[16] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. 2015. LINE: Large-scale Information Network Embedding. *CoRR* abs/1503.03578 (2015). arXiv:1503.03578 http://arxiv.org/abs/1503.03578

[17] Joshua B. Tenenbaum, Vin de Silva, and John C. Langford. 2000. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science* 290, 5500 (2000), 2319.

[18] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural Deep Network Embedding. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16)*. Association for Computing Machinery, New York, NY, USA, 1225–1234. https://doi.org/10.1145/2939672.2939753