Learning Node Embeddings for Machine Learning Tasks

Shivank Goel, NetID : sg2359

1 INTRODUCTION

This reaction paper explores the different ways to generate low dimensional representations called embeddings for the nodes of a graph that can capture the graph structure, so that machine learning models can effectively exploit them. Various kinds of techniques have been used for this task, such as deep learning, non linear dimensionality reduction, matrix factorization, random walk algorithms, convolutional networks etc. These representations can have relevant applications in various domains. They can help visualize graphs on a 2D interface (for discovering communities and other hidden structures). We can do this by using dimensionality reduction (PCA, t-SNE) on the node embeddings. Further, we can also use various clustering methods such as K-means on the node embeddings to discover communities. We can also effectively classify nodes using embeddings as feature vectors, like classification of proteins based on their biological functions. These embeddings are also very useful for link prediction and recommender systems.

Earlier approaches to achieve these applications relied on hand crafted features, such as graph statistics (degrees or clustering coefficients), kernel functions or features to measure local neighborhood structures. However, these approaches were not flexible as they cannot adapt during the learning process and designing such features is time-consuming. Node embeddings automatically capture these features and important graph structures such that we can use them effectively for downstream ML tasks.

Hamilton et. al[1] discusses various approaches that are used for obtaining these node embeddings. All these approaches (and their limitations) have been summarized in Section 2. In Section 3 we will look into the details of the original GCN (graph convolutional network) architecture, introduced by Kipf et. al[2] and discuss its limitations. Section 4 and 5 will introduce two approaches namely GraphSAGE[3] and EvolveGCN[4] that address some of the limitations with the original GCNs.

2 NODE EMBEDDINGS

Node embeddings are the encodings for every node in a graph as a low dimensional vector, such that the relative positioning of the nodes in the original graph is maintained in the embedding vector space. Geometric relations in this embedding space correspond to the edges of the original graph.

2.1 *Encoder Decoder Framework*

Encoders encode nodes into low dimensional vectors and the decoder tries to decode the structural information about the graph. For example, given two vectors, the decoder may predict if an edge exists between them (or how close they are) in the original graph, or the decoder can predict the community (classification task) to which a given node belongs to in the original graph. If the decoder can do that successfully, then we can assume that we have all the graph structure information in the form of low dimensional embeddings.

The decoder outputs the proximity of two given vertices in the original graph and this output is compared with actual proximity value (which can be either binary indicating an edge between nodes, or a real valued number such as the probability of the nodes co-occurring on a random walk of fixed length).

2.2 *Direct Encoding Approaches*

These approaches work like an embedding lookup, where encoding of the node v_i is given by $Enc(v_i) = z_i = ZV_i$ where V_i is one-hot indicator vector, indicating the column of Z corresponding to the encoding of node v_i. The goal is to optimize the encoding matrix Z. Various direct encoding approaches differ in terms of decoder function (measure of

closeness of the two nodes in the original graph) and loss function, i.e, the comparison of $Dec(z_i, z_j)$ with $S_G(v_i, v_j)$, where $S_G(v_i, v_j)$ refers to the actual proximity value. There exists two types of direct encoding approaches:

1. Factorization Based Approaches : These consist of approaches like Laplacian Eigenmaps which defines decoder as $DEC(z_i, z_j) = \|z_i - z_j\|_2^2$ and the loss function weights the decoder outputs depending on the actual proximity $L = \sum_{(v_i,v_j)} DEC(z_i, z_j) \cdot S_G(v_i, v_j)$. Some methods define decoder as a pairwise inner product of node embeddings $DEC(z_i, z_j) = z_i^T z_j$ and use the MSE loss function $L = \sum_{(v_i,v_j)} \|DEC(z_i, z_j) - S_G(v_i, v_j)\|_2^2$. The function $S_G(v_i, v_j)$ captures first order similarity using the adjacency matrix directly, i.e., $S_G(v_i, v_j) = A_{i,j}$ or one can model higher order similarities by considering various powers of the adjacency matrix, i.e, $S_G(v_i, v_j) = A_{i,j}^2$. These approaches are called factorization based approaches because, on average (over all the nodes), they optimize the loss function given by $L = \|Z^T Z - S\|_2^2$.

2. Random Walk Approaches : These consist of approaches like DeepWalk and node2vec. The idea is to optimize the embeddings such that the nodes which tend to co-occur on short random walks have similar embeddings. Instead of using a deterministic decoder, these approaches use a decoder that predicts the probability of reaching the node v_j on a fixed length-T (usually in range 2 to 10) random walk starting from the node v_i. Formally the decoder function is given by $D(z_i, z_j) = e^{z_i^T z_j} / (\sum_{v_k} e^{z_i^T z_k}) = p_{G,T}(v_j / v_i)$. These approaches minimize the cross entropy loss which is given by $L = \sum_{(v_i,v_j) \in D} -log(DEC(z_i, z_j))$ where the set D is generated by sampling N pairs for each node v_i using the random walks starting from the node v_i. However, in practice we use an approximation of this loss because computing the denominator of decoder function is time consuming. For that we can randomly sample the v_k's for each v_i, instead of considering the complete set V (node2vec). We may also reduce computation time by using binary tree structure (DeepWalk- hierarchical softmax). Further node2vec introduces two random walk parameters p and q indicating likelihood of immediately revisiting a node and and visiting a one-hop neighborhood respectively. They can help us in deciding the trade-off between local and community structures learned by the embeddings. LINE is another method that optimizes the first-order and second-order proximity explicitly. For second order proximity, $S_G(v_i, v_j) = A_{i,j}^2$, it uses the same decoder as above, while for first-order proximity it uses the sigmoid decoder function given by $D(z_i, z_j) = 1/(1 + e^{-z_i^T z_j})$. Sometimes it is good to collapse similar nodes in a graph into super-nodes and learn their embeddings first, which can then be used to initialize the embeddings for the constituent nodes for further fine tuning (HARP). Other variations include using random walks that hop over few nodes, and instead of using dot product ($z_i^T z_j$) in decoder, one may use an euclidean distance.

2.3 *General Encoder Decoder Approaches*

In direct encoding we learn embeddings for each node independently, and there is no parameter sharing which means number of parameters grows as $O(|V|)$. Direct encoding also does not leverage any kind of node attributes (e.g. user profiles in social network) and they cannot generate embeddings for newly added nodes without additional rounds of optimization. General approaches instead use complex encoders which leverage the node attributes and graph structure.

1. Neighborhood Auto-encoder : Each node v_i is associated with a neighborhood vector s_i, i.e., the corresponding row in the proximity matrix S. The s_i vectors are passed through the multi-layer neural auto encoders which try to encode s_i into a low dimensional representation, z_i, such that it is easier to reconstruct s_i from that. The loss function for the neural auto-encoder network is given by $L = \sum_{v_i \in V} \|DEC(z_i) - s_i\|_2^2$. Since the number of nodes in a network can be very large (millions), the auto encoder with input size of $|V|$ can become intractable. Also this technique cannot adapt to evolving graphs since the size of auto encoders is fixed.

2. Neighborhood aggregation and convolutional encoders: The node embeddings are initialized as the input node attributes. At each iteration the nodes aggregates the embedding of the neighbors and combine it with its previous

embedding to generate a new embedding by passing all this information through a dense neural network. This aggregation can be done in different ways like element wise weighted average, max pooling neural network, or LSTMs. The parameters used for the aggregate function is shared by all the nodes. This approach leverage node attributes, graph structure, and can adapt to evolving graphs (Inductive Learning - GraphSAGE) since they can generate embeddings for nodes that were not available during training. These embeddings can further be combined with previously discussed decoders and loss functions and trained via Stochastic Gradient Descent. Since these approaches (GCNs, GraphSAGE, EvolveGCNs) are the state of the art approaches for various graph ML tasks, I will be discussing these models in more detail in the subsequent sections.

We can adapt these techniques for various special cases. We can fine tune embeddings for a specific task, for example, for node classification we can define loss function to be $\sigma(z_i^T\theta)$, and then we can update all the aggregation parameters and θ together using SGD. In cases where the given graph contains more than one type of edges and links, we may restrict random walks to only edges of the same type and/or can use type specific encoders and decoders. For graphs that contain multiple copies of the same node in different layers, we can add an additional regularization penalty to the loss function to share information across layers, i.e., if v_i belongs to layers G_1 and G_2, then we can modify loss function as $L(v_i)^{'} = L(v_i) + \lambda\|z_i^{G_1} - z_i^{G_2}\|$.

3 GRAPH CONVOLUTIONAL NETWORKS

GCNs were originally introduced by Kipf et. al[2] for semi-supervised classification. Previous work often assumes that the connected nodes in a graph are likely to share the same label, but graph edges may not necessarily encode similarity, and might contain some additional information. GCNs encode graph structure using neural network model where the hidden representations encode both the local graph structure and node features.

3.1 *Methodological Details*

First, we create an initial embedding for each node using the node features $H^{(0)} = X$. This embedding of nodes is passed through multiple layers (assume L layers) to get the final embedding. To get the $H^{(l+1)}$ layer from the $H^{(l)}$ layer we use the update $H^{(l+1)} = \sigma(\hat{A}H^{(l)}W^{(l)})$, where $W^{(l)}$ is a layer specific parametric matrix and $\hat{A}$ is a normalized self-connected adjacency matrix given by $\hat{A} = D^{-1/2}(A + I)D^{1/2}$. Here, $D = diag(\sum_j (A + I)_{ij})$ is the diagonal matrix containing row sums. $\sigma(\cdot)$ is an activation function which can be *Relu* for the hidden layers and either *Softmax*, *Sigmoid* or *Identity* for final layer depending on the task. We note that the multiplication of $\hat{A}$ with $H^{(l)}$ plays the role of the aggregation of the node embeddings of the neighbor nodes. The normalised version of adjacency matrix ($\hat{A}$) is used to prevent numerical instabilities (vanishing and exploding gradients). The authors call this the *Renormalization Trick*. Since the goal of this paper is to perform a semi-supervised classification task, let $Y_L \subseteq Y$ be the subset of nodes for which the node label is available. We use a softmax function in the final layer, and use a cross entropy loss to update parameters, given by $L = -\sum_{l \in Y_L} \sum_{f=1}^{F} Y_{lf} lnZ_{lf}$, where l refers to the node and f refers to the class of that node. The paper performs the gradient descent using the full training data set for every iteration. Dropout (which acts as an ensemble and regularization strategy) is used during training to introduce stochasticity. Using a sparse representation for A, the memory requirement for training is linear in the number of edges, i.e., $O(|E|)$.

3.2 *Relevant Contributions*

The layer wise linear formulation enables us to build deeper networks. The paper provides a way to incorporate information present both in the data (node features) X, and the adjacency matrix (graph structure) A into the node embeddings. It provides an efficient way (fewer parameters and operations) to achieve a high predictive classification performance (compared to previous approaches at time of paper) on a number of data sets like citation networks (Citeseer, Cora, Pubmed) and knowledge graphs (NELL). The paper also claims that even an untrained GCN model with randomly initialized weight matrices $W^{(l)}$ can serve as a powerful feature extractor for nodes in graph.

3.3 *Limitations*

The training time memory requirement grows linearly with the size of dataset. Instead of using full-batch we need to adopt mini-batch stochastic gradient descent. This is challenging since we need to store all K^{th} order neighborhood connections in the minibatch for a GCN with K layers. Also, though the approach takes into account the node specific features X but don't incorporate edge specific features (like edge weights). I believe this can be included by adjusting the $\hat{A}$ matrix to have weighted normalization (depending on edge weights). Further, instead of giving equal importance to self and neighborhood edges, introducing a trade-off parameter λ, i.e., $\hat{A} = D^{-1/2}(A + \lambda I)D^{1/2}$, might further improve the model performance.

Also, the approach requires that all the nodes in the graph be present during training time, and hence transductive in nature that don't generalize to unseen nodes. GraphSAGE[3] aims to mitigate this problem by sampling the neighborhood nodes and aggregating their information to generate embedding of a given node (or new node). Further the paper does not mention how to update the weight matrices $W^{(l)}$ if the adjacency matrix A or data features X changes (deletion or addition of new edges, changes in node attributes) and evolves with time. A simple approach might be to update the weight matrices by running more epochs of the GCN algorithm with the newly updated A and X (or suitable sub graph that includes new changes), however it might be a time consuming and computationally expensive process for the large networks. EvolveGCNs[4] solve this problem by updating weights using RNN architectures.

4 GRAPH SAGE (SAMPLE & AGGREGATE)

GraphSAGE[3] extends the GCN architecture (which uses transductive learning) to the task of inductive unsupervised learning, so that embeddings can be quickly generated for unseen nodes or entirely new sub graphs. Further, the paper leverages the approach of aggregating node embeddings of the neighbor nodes using trainable functions rather than simple convolutions used in GCNs and achieve significant performance gains. The paper also extends the GCN approach so that it can be used in a minibatch setting.

An inductive approach helps to generalize and extend the learned parameters across graphs with same kind of features, for example embedding parameters derived for proteins within one organism can be used for data collected on new organisms. It is tricky to align newly observed subgraphs to embeddings that have already been optimized. The inductive framework should be able to generate embeddings that capture both node's local role and its global position.

The idea is to leverage node features (text attributes, node profile, node degrees) and learn a set of *aggregator functions* to aggregate local neighborhood information (from a different number of hops or search depth) which can then be used to predict the embeddings of entirely unseen nodes. Using this approach we can learn the node embeddings in an unsupervised or fully supervised manner.

4.1 *Methodological Details*

The model aims to learn K aggregator functions denoted by $AGGREGATE_k$ and corresponding weight matrices W^k, $\forall k \in \{1, 2, ..., K\}$ where index k refers to the depth/layer of the neural network. At any given layer k of the neural network we update the hidden representations of every node h_v^k by first aggregating the the representations of its immediate neighbors into $h_{N(v)}^k$, and then passing this aggregated information along with the previous node representation h_v^{k-1} through a fully connected layer. The *Algorithm 1* describes the computations in further detail. To extend this algorithm for minibatch setting, given a set of nodes, the paper first samples the required neighborhood set (upto depth K) and then run the inner loop on the sampled nodes. Instead of considering the complete neighborhood set $N(v)$ for a given node, the authors uniformly sample (different uniform samples at every iteration k) a fixed size neighborhood that helps to keep computational requirements of each minibatch to be fixed. The embeddings can be learned in an fully-unsupervised way using the loss function that promotes nearby nodes to have similar embeddings and disparate nodes to have distinct ones. Specifically, the loss function is given by choosing a node v that co-occurs with u on a fixed-length random walk and defining : $J_G(z_u) = -log(\sigma(z_u^T z_v)) - Q \cdot E_{v_n \sim P_n(v)} log((-z_u^T z_{v_n}))$, where P_n is a neg-

ative sampling distribution, and Q refers to the number of negative samples. For supervised settings this loss function can be replaced or augmented by the task-specific objective.

for *k = 1....K* **do**
  for $\forall v \in V$ **do**
    $h^k_{N(v)} \leftarrow AGGREGATE_k(h^{k-1}_u, \forall u \in N(v))$
    $h^k_v \leftarrow \sigma(W^k \cdot CONCAT(h^{k-1}_v, h^k_{N(v)}))$
    $h^k_v \leftarrow h^k_v / \|h^k_v\|_2$
  end
end

Algorithm 1: GraphSAGE Algorithm

4.2 *Relevant Contributions*

The paper does a good job of suggesting an inductive way to generate embeddings for unseen nodes, incorporating mini batch training and improving aggregation methods using aggregator functions. The paper explores various aggregator functions like the *Mean* which basically takes the mean of neighborhood vectors, *LSTMs* that creates a condensed representation by passing neighborhood vectors via a LSTM model, *Pooling* which pools (both max or avg pool can be used) the neighborhood vectors after passing each of them through a fully connected layer. Max-pooling operator helps to effectively capture different aspects of local neighborhood. Ideally aggregator functions should be symmetric i.e., independent of the sampling order of neighborhood nodes, however LSTMs are not inherently symmetric. To adapt LSTMs to operate on unordered set they were applied to a random permutation of neighbors. The GraphSAGE-pool/LSTM in general performs better than GraphSAGE-GCNs, however I would have liked if the authors provided a comparison with the original GCN[2] architecture.

4.3 *Limitations*

Similar to GCNs, the paper does not focus on incorporating edge specific features like weighted or directed edges. One idea is that instead of taking a normal aggregate one can do weighted aggregate depending on the edge weights. Also, similar to GCNs, the paper does not mention how to update the parameters of $AGGREGATE_k$ (in case of LSTMS and pool) and $W^{(k)}$ if the adjacency matrix A or data features X changes as the graph evolves over time. One might use a strategy similar to EvolveGCN[4] for this, which I will discuss in the next section. Another extension of this work can be to look at other distributions for neighborhood sampling (non-uniform) or try to learn the parameters for the sampling distribution.

5 EVOLVE GCN

EvolveGCNs[4] adapts the GCN approach to focus on dynamically evolving graphs rather than static ones, and update the node embeddings to reflect variations that come in the network. It uses RNNs to update the graph model itself overtime. There have been methods that incorporate dynamism of evolving graphs using RNNs on top of features extracted by the GCNs. However, this method directly injects the dynamism to the parameters of the GCN. I will not go into complete details of this approach, but will briefly describe the key components.

5.1 *Methodological Details*

Given a sequence of graphs at different time steps, the paper tries to effectively calculate node embeddings for the next time step given embeddings of current time step. The paper assumes that all the graphs across time steps are built on a common node set of cardinality n. Let X_t contains the node embeddings for all the nodes at time step $t \in \{1, 2, ..., T\}$ (where X_1 was obtained using the exact GCN procedure described in section 3). The goal is to obtain the embeddings for the next time step (X_{t+1}). Let, $H^{(l)}_t$ be the hidden states of GCN at time t. Let the learned weight matrices at time step t for the L layers of GCN be represented by $W^{(l)}_t$ $\forall l \in \{1, 2...L\}$. First for each layer $l \in \{1, 2..., L\}$, we update the weight parameters $W^{(l)}_t$ to obtain weight parameters for next time step $W^{(l)}_{t+1}$ by passing the corresponding hidden

state from previous time step into the RNN (specifically GRU), i.e., $W_{t+1}^{(l)} = GRU(H_t^{(l)}, W_t^{(l)})$. Briefly, first we have to summarize $H_t^{(l)}$ so that it has same number of columns as of $W_t^{(l)}$ (using a parametrized vector p), and then we apply GRU to each column of $W_t^{(l)}$ independently using summarized $H_t^{(l)}$ as input. After obtaining weights for $W_{t+1}^{(l)}$, one can obtain the hidden states for time (t+1) iteratively layer by layer using standard GCN architecture (which the paper refers to as *GCONV*), i.e., $H_{t+1}^{(l+1)} = \sigma(\hat{A}H_{t+1}^{(l)}W_{t+1}^{(l)})$, where $H_{t+1}^{(0)} = X_t$ and $H_{t+1}^{(L)} = X_{t+1}$, i.e., the node embeddings for the next time step. This approach performs better over single GCN model for all time steps in some of the edge classification and link prediction tasks.

5.2 *Limitations*

The paper only extends the GCN architecture to incorporate temporal modeling instead of using one single GCN model across all the time steps. It faces the problems that were there with the original GCNs, such as the huge training time memory requirement, incorporating edge specific features, equal importance to self and neighborhood edges, and the generalization to unseen nodes.

6 CONCLUSIONS

Node embeddings are state of the art architectures to capture node features and graph structural information so that they can be leveraged for downstream machine learning tasks. GCNs provide an efficient and powerful approach to generate these embeddings, but they do not adapt to new nodes or temporal changes. GraphSAGE provides a way to adapt GCNs to new nodes and subgraphs. It also solves other problems with GCNs like incorporating mini batch gradient descent and effective ways to aggregate neighborhood information. EvolveGCNs further provide methodologies to get better embeddings for graphs that evolve with time. An interesting direction for future work would be to look at the approaches which merge ideas both from the GraphSAGE to increase adaptation to new nodes and EvolveGCNs to update model parameters when new changes are made in the network.

REFERENCES

1. W. L. Hamilton, R. Ying, and J. Leskovec, "Representation learning on graphs: Methods and applications," *arXiv preprint arXiv:1709.05584*, 2017.
2. T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
3. W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Advances in Neural Information Processing Systems*, pp. 1024–1034, 2017.
4. A. Pareja, G. Domeniconi, J. Chen, T. Ma, T. Suzumura, H. Kanezashi, T. Kaler, and C. E. Leisersen, "Evolvegcn: Evolving graph convolutional networks for dynamic graphs," *arXiv preprint arXiv:1902.10191*, 2019.