E9 333 - Advanced Deep Representation Learning Term Paper

Kawin M
MTech CSE - 19723

Rankit Kachroo
MTech CSE - 19266

1) Learning Graph Embedding with Adversarial Training Methods

I. Introduction

Problem Statement: Given a graph $G = \{V, E, X\}$, map the nodes $v_i \in V$ to a low-dimensional vector (embeddings) $z_i \in R^d$. Formally, $f : (A, X) \to Z$, where z_i^T is the i^{th} row of the embedding matrix Z, A is the adjacency matrix of G and X is the feature matrix such that $x_i \in X$ encodes the textual features of each node v_i.
Previous works mainly focus on preserving the graph structure or minimizing the reconstruction error, while completely ignoring the latent data distribution. In practice, this learns a degenerate identity mapping and leads to poor representation. The paper tackles this problem by incorporating an adversarial training scheme to regularize the latent codes and enforce them to follow some prior data distribution to learn a robust graph representation. Additional challenge for graph data is that, both topological structure A and content information X are required to be represented into the latent space.
Two variants of adversarial approaches, adversarially regularized graph autoencoder (ARGA) and it's variational variant ARVGA are proposed to learn the graph embedding.

II. Overall Framework

The framework consists of two modules: 1) Graph convolutional auto-encoder and 2) Adversarial regularization.

A. Graph Convolutional Auto-Encoder

The autoencoder takes in the structure of graph A and the node content X as inputs to learn a latent representation Z, and then reconstructs the graph data A and/or X from Z. The encoder uses 2 layer GCN and has two variants - Graph Encoder and Variational Graph Encoder, which is constructed as follows:

$$Z^{(1)} = f_{Relu}(X, A|W^{(0)}) \tag{1}$$

$$Z^{(2)} = f_{linear}(Z^{(1)}, A|W^{(1)}) \tag{2}$$

where, $f(Z^{(l)}, A|W^{(l)}) = \phi(D^{-1/2}AD^{-1/2}Z^{(l)}W^{(l)})$ is the spectral convolution function. The variational variant is defined by an inference model:

$$q(Z|X, A) = \prod_{i=1}^{n} q(z_i|X, A), \tag{3}$$

$$q(z_i|X, A) = N(z_i|_i, diag(\sigma^2)) \tag{4}$$

Here, $\mu = Z^{(2)}$ is the matrix of mean vectors z_i; similarly $log\sigma = f_{linear}(Z^{(1)}, A|W'(1))$ which shares the weights $W^{(0)}$ with μ in the first layer in Eq. (1).

The decoder is used to reconstruct the graph data. There are two variants - i) simple inner product decoder and ii) GCN decoder and each of these variants can chose to reconstruct either A or X or both. The inner product decoder is constructed as:

$$p(A|Z) = \prod_{i=1}^{n}\prod_{j=1}^{n} p(A_{ij}|z_i, z_j); \tag{5}$$

$$p(A_{ij}|z_i, z_j) = \sigma(z_i^T \cdot z_j) \tag{6}$$

and the GCN decoder is constructed using two-layer GCN similar to that of the encoder, such that

$$Z^{(D)} = f_{linear}(X, A|W_D^{(0)}) \tag{7}$$

$$O = f_{linear}(Z^{(D)}, A|W_D^{(1)}) \tag{8}$$

Optimization: For the graph encoder, the reconstruction error of the graph data is minimized by:

$$\mathcal{L}_A = \mathbb{E}_{q(Z|(X,A))}[logp(A|Z)] \tag{9}$$

$$\mathcal{L}_X = \mathbb{E}_{q(Z|(X,A))}[logp(X|Z)] \tag{10}$$

$$\mathcal{L}_0 = \mathcal{L}_A + \mathcal{L}_X \tag{11}$$

For the variational graph encoder, the variational lower bound is optimized as follows:

$$\mathcal{L}_1 = \mathbb{E}_{q(Z|(X,A))}[logp(A|Z)] - KL[q(Z|X, A)|p(Z)] \tag{12}$$

where, prior distribution p(•) can be a uniform distribution or a Gaussian distribution.

B. Adversarial Model D(z)

The adversarial model acts as a discriminator to distinguish whether a latent code is from the prior p_z (positive) or graph encoder G(X, A) (negative). The equation for training the encoder model with Discriminator D(Z) can be written as follows:

$$\min_{\mathcal{G}} \max_{\mathcal{D}} \mathbb{E}_{z\sim p_z}[logD(Z)] + \mathbb{E}_{x\sim p(x)}[log(1 - D(G(X, A)))] \tag{13}$$

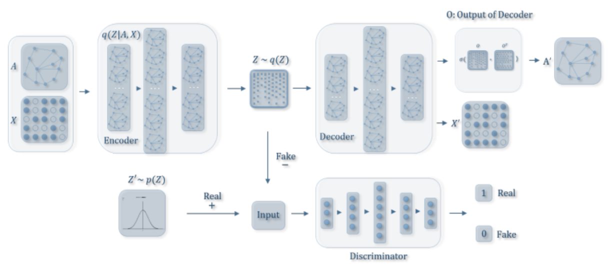

Fig. 1. The adversarially regularized graph autoencoder model with GCN decoder which simultaneously reconstructs graph data A and X.

C. *Unified loss*

The model jointly optimizes both the autoencoder and the adverserial model. Given a graph G, get the latent variables matrix Z from the graph convolutional encoder. Take same number of samples from the generated Z and the real data distribution p_z to update the discriminator with the cross-entropy loss computed. Discriminator is trained for K runs and generator updates itself with the generated gradient.

III. Observations

Gaussian prior vs. Uniform prior: Performance of the proposed models is not very sensitive to the prior distributions, for the node clustering task. For link prediction, ARGA with Gaussian distribution slightly outperforms the ones with Uniform distribution. The situation reversed with the variational ARGA models.

Embedding visualization: Adversarially regularized embedding shows better visualization with clear boundary line between two clusters. Considering the only difference between ARGA and the GAE is the adversarial training regularization scheme, it is claimed that adversarial regularization is helpful to enhance the quality of graph embedding.

Limitations: The limitations of this approach is that it is only well-defined when given a fixed-set of nodes, limiting its utility for the general graph generation problem, and are limited to learning from a single input graph.

2) GraphRNN: Generating Realistic Graphs with Deep Auto-regressive Models

I. Introduction

Problem Statement: To model and generate graphs by learning a distribution $p_{model}(G)$ over graphs, based on a set of observed graphs $G = G_1, ..., G_s$ sampled from data distribution $p(G)$, where each graph G_i may have a different number of nodes and edges. Modeling complex distributions over graphs and then efficiently sampling from these distributions is challenging due to the non-unique, high-dimensional nature of graphs and the complex, non-local dependencies that exist between edges in a given graph.
Previous works are limited to learning from a single graph or generating small graphs with 40 or fewer nodes. Unlike other modalities like text, images, the problem with graph data is that there are exponential number of ways to represent a given graph using adjacency matrix by different ordering of nodes. The paper addresses the above challenges by introducing an autoregressive model "GraphRNN" that approximates any distribution of graphs with minimal assumptions about their structure.

II. Overview

As it is complex to model the distribution p(G) over graphs, model it as distribution over the sequence $p(S^\pi)$ of adjacency vectors of nodes seen as of now, under a fixed node ordering π. This can be further decomposed into conditional probabilities $p(S^\pi) = \prod_{i=1}^{n+1} p(S_i^\pi|S_{<i}^\pi)$, which can be modelled as RNN. However, this can be further decomposed into $p(S_i^\pi|S_{<i}^\pi) = \prod_{j=1}^{i-1} p(S_{i,j}^\pi|S_{i,<j}^\pi, S_{<i}^\pi)$, which can be modelled as hierarchy of RNN. Still, there are n! node orderings and variable size of distribution $p(S_i^\pi)$ at each time i, which is solved by generating the sequences in the BFS ordering.

III. Overall Framework

A. Definitions

One common way to represent a graph $G = (V, E)$ is using an adjacency matrix A, which requires a node ordering π that maps nodes to rows/columns of the adjacency matrix. Here, π is a permutation function over V is a permutation of (v1, ..., vn)). Let $\prod$ be the set of all n! possible node permutations and let's assume a uniform probability over all node orderings.

B. Modelling Graphs as sequences

Transform modelling of p(G): Formally, for a fixed node ordering π for a graph with n nodes, a mapping f_S from graphs to sequences is defined as $S^\pi = f_S(G, \pi) = (S_1^\pi, ..., S_n^\pi)$, where $S_i^\pi \in 0, 1^{i-1}$ is the adjacency vector between node $\pi(i)$ and the previous nodes $\pi(j), j < i$.
For undirected graphs, S^π determines a unique graph G. Thus, instead of learning p(G), whose sample space cannot be easily characterized, sample the auxiliary π to get the observations of S^π and learn $p(S^\pi)$. This is modeled autoregressively due to the sequential nature of S^π. At inference time, G can be sampled without explicitly computing p(G) by sampling S^π,

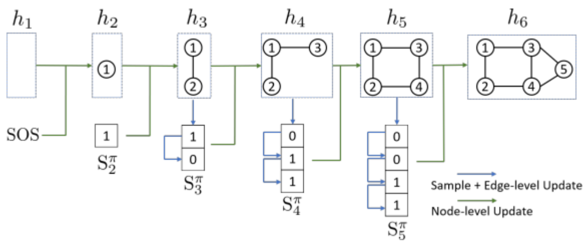

Fig. 2. GraphRNN at inference time. Green arrows denote the graph-level RNN and Blue arrows denote the edge-level RNN.

which maps to G via f_G.

Decomposing $p(S^\pi)$ into product of conditional probabilities of previous time steps: S^π can be further decomposed as the product of conditional distributions over the elements:

$$p(S^\pi) = \prod_{i=1}^{n+1} p(S_i^\pi|S_1^\pi, ..., S_{i-1}^\pi) = \prod_{i=1}^{n+1} p(S_i^\pi|S_{<i}^\pi) \qquad (14)$$

C. GraphRNN framework

A GRU is used to model the conditional probability $p(S^\pi)$ that consists of a state-transition function and an output function:

$$h_i = f_{trans}(h_{i-1}, S_{i-1}^\pi) \qquad (15)$$
$$\theta_i = f_{out}(h_i) \qquad (16)$$

where, $h_i \in \mathbb{R}^d$ is a vector that encodes the state of the graph generated so far and θ_i specifies the distribution of next node's adjacency vector (i.e., $S_i^\pi \sim P_{\theta_i}$). **GraphRNN variants:** Two variants of GraphRNN, both of which implement the transition function f_{trans} (i.e., the graph-level RNN) as a GRU but differ in the implementation of f_{out} (i.e., the edge-level model).

Variant 1: Multivariate Bernoulli $p(S_i^\pi|S_{<i}^\pi)$ is modelled as a multivariate Bernoulli distribution, parameterized by the $\theta_i \in \mathbb{R}^{i-1}$ vector that is output by f_{out}. In this, f_{out} is implemented as single layer MLP with sigmoid activation, that shares weights across all time steps. The output of f_{out} is a vector θ_i, whose element $\theta_i[j]$ can be interpreted as a probability of edge (i, j). Then, sample edges in S_i^π independently according to a multivariate Bernoulli distribution parameterized by θ_i.

Variant 2: Dependent Bernoulli Sequence $p(S_i^\pi|S_{<i}^\pi)$ is further decomposed to capture the complex edge dependencies, such that,

$$p(S_i^\pi|S_{<i}^\pi) = \prod_{j=1}^{i-1} p(S_{i,j}^\pi|S_{i,<j}^\pi, S_{<i}^\pi) \qquad (17)$$

where, $S_{i,j}^\pi$ denotes a binary scalar that is 1 if node $\pi(v_i + 1)$ is connected to node $\pi(v_j)$. Each distribution in the product is approximated by an another RNN. Conceptually, this is a

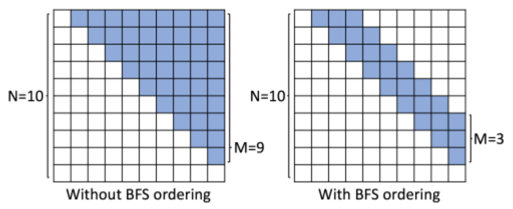

Fig. 3. Illustration of using BFS ordering to reduce M.

hierarchical RNN, where the first (i.e., the graph-level) RNN generates the nodes and maintains the state of the graph, and the second (i.e., the edge-level) RNN generates the edges of a given node. The edge-level RNN is also a GRU model, where the hidden state is initialized via the graph-level hidden state h_i and where the output at each step is mapped by a MLP to a scalar indicating the probability of having an edge. $S^{\pi}_{i,j}$ is sampled from this distribution specified by the jth output of the ith edge-level RNN, and is fed into the j+1th input of the same RNN. All edge-level RNNs share the same parameters.

IV. Tractability via Breath First Search:

There are two major problems to the approach proposed till this point:

1) There are n! possible orderings for a graph with n nodes to train the model.
2) The output vector θ_i is of varying size in each iteration, that is, it increments by one with every addition of a node to the adjacency vector, which is difficult to model using RNN.

Thus, rather than learning to generate graphs under any possible node permutation, learn to generate graphs using breadth-first-search (BFS) node orderings. This BFS function takes a random permutation π as input, picks $\pi(v1)$ as the starting node and appends the neighbors of a node into the BFS queue in the order defined by π. Note that the BFS function is many-to-one, i.e., multiple permutations can map to the same ordering after applying the BFS function.

As multiple node permutations map to the same BFS ordering, it provides a reduction in the overall number of sequences. Also, the BFS ordering makes learning easier by reducing the number of edge predictions we need to make in the edge-level RNN; in particular, when we are adding a new node under a BFS ordering, the only possible edges for this new node are those connecting to nodes that are are still in the BFS queue.

This insight allows to redefine the variable size S^{π}_{i} vector as a fixed M-dimensional vector, representing the connectivity between node $\pi(v_i)$ and nodes in the current BFS queue with maximum size M, which can be bounded by $O(max_{d=1}^{diam(G)}|\{v_i|dist(v_i, v1) = d\}|$, where, dist denotes the shortest-path-distance between vertices.

Thus, the overall time complexity of GraphRNN is $O(Mn)$.