GraphGAN: Graph Representation Learning with Generative Adversarial Nets

Most existing methods of graph representation learning can be classified into two categories:

- Generative
- Generative graph representation learning models assume that, for each vertex v_c, there exists an underlying true connectivity distribution $p_{ture}(v|v_c)$, which implies v_c's connectivity preference (or relevance distribution) over all other vertices in the graph. The edges in the graph can thus be viewed as observed samples generated by these conditional distributions, and these generative models learn vertex embeddings by maximizing the likelihood of edges in the graph.

- Discriminative
- Aim to learn a classifier for predicting the existence of edges directly. Discriminative models consider two vertices v_i and v_j, jointly as features, and predict the probability of an edge existing between the two vertices, $p(edge|(v_i, v_j))$

- Motivation
- Unifies generative and discriminative thinking for graph representation learning.

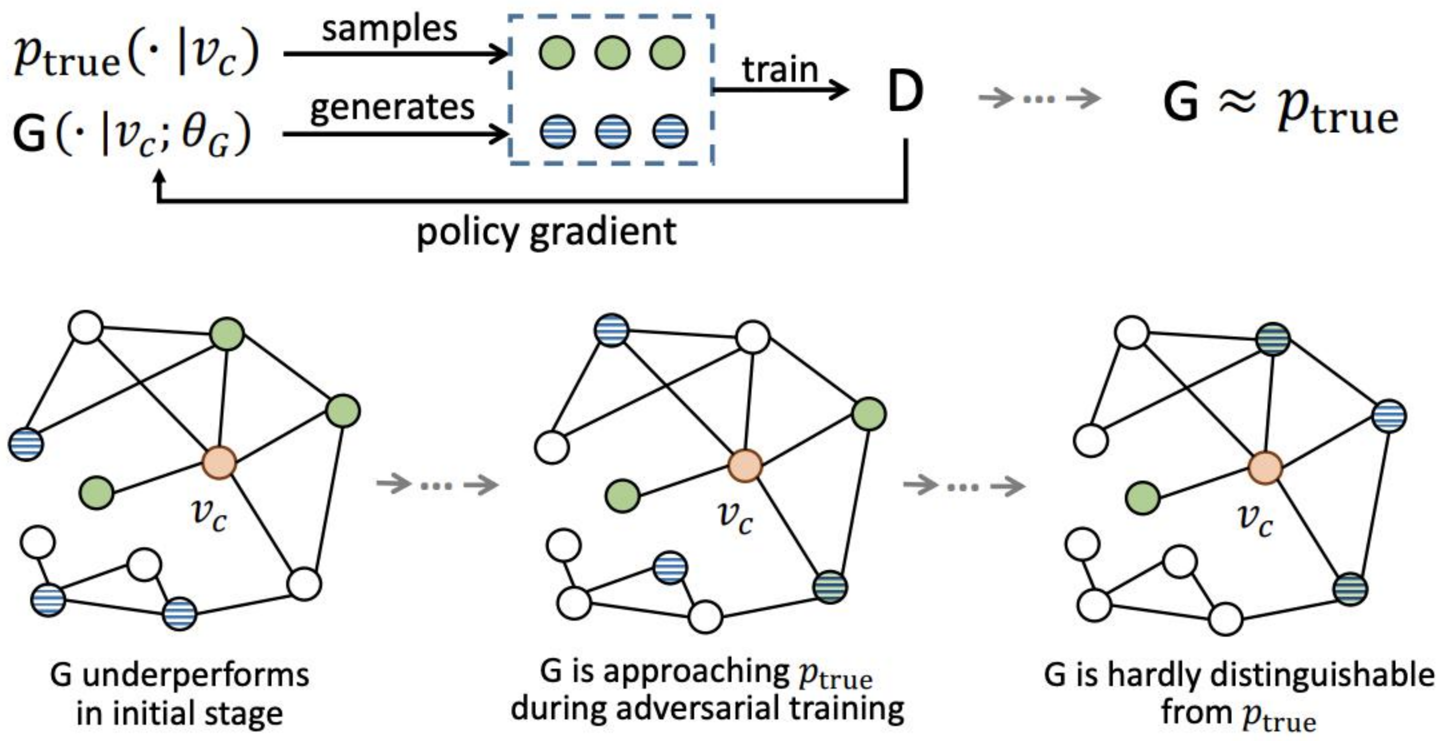
Framework

- $\mathcal{N}(v_c)$ denotes the set of vertices directly connected to v_c.
- Conditional probability $p_{ture}(v|v_c)$ reflects v_c's connectivity preference distribution over all other vertices. Thus, $\mathcal{N}(v_c)$ can be seen as a set of observed samples drawn from $p_{ture}(v|v_c)$.
- **Generator**

$G(v|v_c;\theta_G)$ tries to approximate the underlying true connectivity distribution and generates (or selects, if more precise) the most likely vertices to be connected with v_c from vertex set.

- **Discriminator**
- $D(v, v_c; \theta_D)$ aims to discriminate the connectivity for the vertex pair (v, v_c) and output a single scalar representing the probability of an edge existing between v and v_c.

$$\min_{\theta_G} \max_{\theta_D} V(G, D) = \sum_{c=1}^{V} \Big(\mathbb{E}_{v \sim p_{\text{true}}(\cdot|v_c)} \big[\log D(v, v_c; \theta_D) \big] + \mathbb{E}_{v \sim G(\cdot|v_c;\theta_G)} \big[\log \big(1 - D(v, v_c; \theta_D)\big) \big] \Big). \tag{1}$$



$p_{\mathrm{true}}(\cdot\,|v_c)$
samples
$\mathrm{G}(\cdot\,|v_c;\theta_G)$
generates
train
D
$\mathrm{G} \approx p_{\mathrm{true}}$
policy gradient
v_c
G underperforms in initial stage
G is approaching p_{true} during adversarial training
G is hardly distinguishable from p_{true}

Figure 1: Illustration of GraphGAN framework.

Discriminator Optimization

- $D(v, v_c) = \sigma({d_v}^T, d_{v_c}) = \frac{1}{1+\exp(-({d_v}^T d_{v_c}))}$(maximize)

$$\nabla_{\theta_D} V(G, D) = \begin{cases} \nabla_{\theta_D} \log D(v, v_c), \ if \ v \sim p_{\text{true}}; \\ \nabla_{\theta_D} \big(1 - \log D(v, v_c)\big), \ if \ v \sim G. \end{cases}$$

- Where $d_v, d_{v_c} \in R^k$, are the k-dimensional representation vectors of vertices v and v_c respectively for discriminator D,both of them can be update.

Generator Optimization

- The generator aims to minimize the log-probability that the discriminator correctly assigns negative labels to the samples generated by G. In other words, the generator shifts its approximated connectivity distribution (through its parameter θ_G) to increase the scores of its generated sample.
- Gradient $\nabla_{\theta_G} V(G, D)$ indicates that vertices with a higher probability of being negative samples will "tug" G stronger away from themselves.

$$
\begin{aligned}
&\nabla_{\theta_G} V(G, D) \\
=&\nabla_{\theta_G} \sum_{c=1}^{V} \mathbb{E}_{v \sim G(\cdot|v_c)} \big[\log \big(1 - D(v, v_c)\big) \big] \\
=&\sum_{c=1}^{V} \sum_{i=1}^{N} \nabla_{\theta_G} G(v_i|v_c) \log \big(1 - D(v_i, v_c)\big) \\
=&\sum_{c=1}^{V} \sum_{i=1}^{N} G(v_i|v_c) \nabla_{\theta_G} \log G(v_i|v_c) \log \big(1 - D(v_i, v_c)\big) \\
=&\sum_{c=1}^{V} \mathbb{E}_{v \sim G(\cdot|v_c)} \big[\nabla_{\theta_G} \log G(v|v_c) \log \big(1 - D(v, v_c)\big) \big].
\end{aligned}
$$

Limitation of Softmax in GE

- The calculation of softmax involves all vertices in the graph, which implies that for each generated sample v, we need to calculate gradients and update.
- The graph structure encodes rich information of proximity among vertices, but softmax completely ignores the utilization of structural information from graphs as it treats vertices without any discrimination

Graph Softmax for Generator

- The key idea of graph softmax is to define a new method of computing connectivity distribution in generator $G(.|v_c, \theta_G)$ that satisfies the following three desirable properties:
- *Normalized.* The generator should produce a valid probability distribution, i.e.,$\sum_{v \neq v_c} G(v|v_c; \theta_G) = 1$
- *Graph-structure-aware.* The generator should take advantage of the structural information of a graph to approximate the true connectivity distribution. Intuitively, for two vertices in a graph, their connectivity probability should decline with the increase of their shortest distance.
- *Computationally efficient.* Distinguishable from full softmax, the computation of $G(v|v_c; \theta_G)$ should only involve a small number of vertices in the graph.

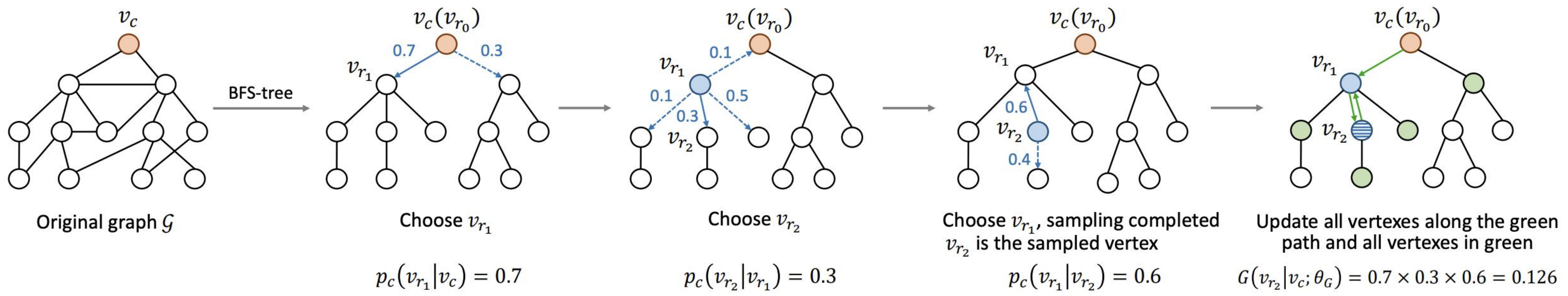
- 1. Performing Breadth First Search (BFS) on the original graph G starting from vertex v_c, which provides us with a BFS-tree T_c rooted at v_c. $\mathcal{N}_c(v)$ as the set of neighbors of v.
- 2. For a given vertex v and one of its neighbors $v_i \in \mathcal{N}_c(v)$. Define the relevance probability of v_i given v as:

$$p_c(v_i|v) = \frac{\exp({g_{v_i}}^T g_v)}{\sum_{v_j \in \mathcal{N}_c(v)} \exp({g_{v_j}}^T g_v)}$$

Each vertex v can be reached by a unique path from the root v_c in T_c. Denote the path as $P_{v_c \to v} = (v_{r_0}, v_{r_1}, \ldots, v_{r_m})$. Thus:

$$G(v|v_c) \triangleq \left(\prod_{j=1}^{m} p_c(v_{r_j}|v_{r_{j-1}})\right) \cdot p_c(v_{r_{m-1}}|v_{r_m}),$$

$p_c(.|.)$ is the relevance probability.


v_c
Original graph $\mathcal{G}$
BFS-tree
$v_c(v_{r_0})$
0.7
0.3
v_{r_1}
Choose v_{r_1}
$p_c(v_{r_1}|v_c) = 0.7$
$v_c(v_{r_0})$
v_{r_1}
0.1
0.1
0.3
0.5
v_{r_2}
Choose v_{r_2}
$p_c(v_{r_2}|v_{r_1}) = 0.3$
$v_c(v_{r_0})$
v_{r_1}
0.6
v_{r_2}
0.4
Choose v_{r_1}, sampling completed
v_{r_2} is the sampled vertex
$p_c(v_{r_1}|v_{r_2}) = 0.6$
$v_c(v_{r_0})$
v_{r_1}
v_{r_2}
Update all vertexes along the green path and all vertexes in green
$G(v_{r_2}|v_c;\theta_G) = 0.7 \times 0.3 \times 0.6 = 0.126$

Algorithm 1 Online generating strategy for the generator

```
Require: BFS-tree $T_c$, representation vectors $\{\mathbf{g}_i\}_{i \in \mathcal{V}}$
Ensure: generated sample $v_{gen}$
 1: $v_{pre} \leftarrow v_c, v_{cur} \leftarrow v_c$;
 2: while true do
 3:     Randomly select $v_i$ proportionally to $p_c(v_i|v_{cur})$ in Eq. (6);
 4:     if $v_i = v_{pre}$ then
 5:         $v_{gen} \leftarrow v_{cur}$;
 6:         return $v_{gen}$
 7:     else
 8:         $v_{pre} \leftarrow v_{cur}, v_{cur} \leftarrow v_i$;
 9:     end if
10: end while
```

Algorithm 2 GraphGAN framework

```
Require: dimension of embedding $k$, size of generating samples
    $s$, size of discriminating samples $t$
Ensure: generator $G(v|v_c;\theta_G)$, discriminator $D(v, v_c;\theta_D)$
 1: Initialize and pre-train $G(v|v_c;\theta_G)$ and $D(v, v_c;\theta_D)$;
 2: Construct BFS-tree $T_c$ for all $v_c \in \mathcal{V}$;
 3: while GraphGAN not converge do
 4:     for G-steps do
 5:         $G(v|v_c;\theta_G)$ generates $s$ vertices for each vertex $v_c$ ac-
            cording to Algorithm 1;
 6:         Update $\theta_G$ according to Eq. (4), (6) and (7);
 7:     end for
 8:     for D-steps do
 9:         Sample $t$ positive vertices from ground truth and $t$ nega-
            tive vertices from $G(v|v_c;\theta_G)$ for each vertex $v_c$;
10:         Update $\theta_D$ according to Eq. (2) and (3);
11:     end for
12: end while
13: return $G(v|v_c;\theta_G)$ and $D(v, v_c;\theta_D)$
```