Loss Function for Skip-Gram methods

Yitao (Viola) Chen

Recall from last time

Algorithm 1 DEEPWALK(G, w, d, γ, t)

Input: graph $G(V, E)$
window size w
embedding size d
walks per vertex γ
walk length t

Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$

Initialization: Sample Φ from $\mathcal{U}^{|V| \times d}$
Build a binary Tree T from V
for $i = 0$ to γ **do**
    $\mathcal{O} = \text{Shuffle}(V)$
    for each $v_i \in \mathcal{O}$ **do**
        $\mathcal{W}_{v_i} = RandomWalk(G, v_i, \text{t})$
        SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
    end for
end for

Correction

- it was mentioned last time that this algorithm takes a graph G, and a high dimensional node feature matrix as input
- this was indeed very briefly mentioned in DeepWalk paper
- However, I was not able to fully understand how it works with the high dimensional feature space
- in natural language processing, words don't have a high dimensional feature
- other Skip-gram based graph models did not mention high dimensional node features
- For the rest of the discussion, I will correct myself that the goal is to <u>learn an embedding purely from a graph specified by an adjacency matrix</u> (there could be a way to work with features too, but I don't know)

DeepWalk Skip-gram update procedure

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

1: **for each** $v_j \in \mathcal{W}_{v_i}$ **do**
2:     **for each** $u_k \in \mathcal{W}_{v_i}[j-w : j+w]$ **do**
3:         $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
4:         $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
5:     **end for**
6: **end for**

- given current representation of vertex $v_j, \Phi(v_j) \in \mathbb{R}^d$, want to maximize the probability of seeing its neighbors in the walk

Objective

- Goal:

$$\max_{\Phi} \sum_{i-w \leq k \leq i//+w} \log \Pr(u_k | \Phi(v_i))$$

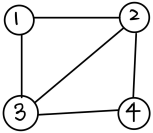
- Softmax:

$$\Pr(u_k | \Phi(v_i))$$

- Basic definition:

$$\Pr(u_k | \Phi(v_i)) = \frac{\exp(\Phi(u_k)^T \Phi(v_i))}{\sum_{1 \leq v_j \leq |V|} \exp(\Phi(v_j)^T \Phi(v_i))}$$

and yes this probably can be written in the matrix form, leading to the question of whether we really have to do one node at a time



Suppose we wants a 2-dimensional embedding, ie. $d = 2$

Initialize: $\Phi = \begin{bmatrix} 0.1 & 0.6 \\ 0.7 & 0.2 \\ 0.5 & 0.5 \\ 0.1 & 0.3 \end{bmatrix}$

walk: $W_{v_4} = [4, 2, 3]$

$$\begin{aligned} \Pr(v_3 | \Phi(v_4)) &= \frac{\exp(\Phi(v_3)^T \Phi(v_4))}{\sum_{k=1}^{W} \exp(\Phi(v_k)^T \Phi(v_4))} \\ &= \frac{\exp(0.2)}{\exp(0.19) + \exp(0.13) + \exp(0.2)} \end{aligned}$$

Computational challenge

- it really depends on size of $|V|$
- if $|V|$ is in the range of $\geq 10^5$. computing this at each round could be very expensive
- however, if your graph doesn't have as many nodes (say < 5000) perhaps simply computing like this is enough

There are ways to efficiently approximate the softmax:

- hierarchical softmax
- negative sampling / Noise contrastive estimation

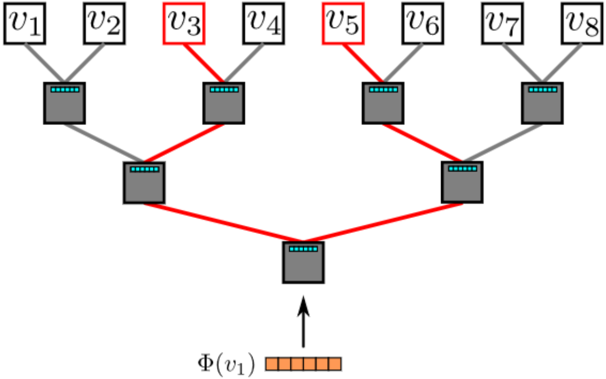
Hierarchical softmax

- in essence, a smart way to reduce computational cost using binary tree representation
- uses the W words as its leaves
- for each node, represent explicitly the relative probabilities of its child nodes (like a decision tree we try to train)
- tree structure is pre-built (line 2 of DeepWalk), trainable
- training: learn the branch "length"
- only need to evaluate about $\log_2(W)$ nodes to compute softmax

Hierarchical softmax in DeepWalk

Example:

- current word/vertex: v_1
- co-occurring words / neighbors: v_3, v_5



Hierarchical softmax in Deepwalk (II)

- $\Pr(v_3|\Phi(v_1)) = \Pr(b_1|\Phi(v_1)) \times \Pr(b_2|\Phi(v_1) \cap b_1) \times \Pr(v_3|\Phi(v_1) \cap b_1 \cap b_2)$
- Assumption: each branching point in the tree is conditional independent given a node embedding $\Phi(v_i)$

$$\Pr(v_3|\Phi(v_1)) = \prod_{l=1}^{\log|V|} \Pr(b_l|\Phi(v_1))$$

$$\Pr(v_5|\Phi(v_1)) = \prod_{l=1}^{\log|V|} \Pr(b_l|\Phi(v_1))$$

where b_l are those branching points (boxes) in the tree, $\Pr(b_l|\Phi(v_1))$ can be modeled by binary classifier assigned to its parent.

Noise Contrastive Estimation (NCE) / Negative sampling

- so far, only maximizing probability of word/node within context
- here, aims at maximizing the similarity of the words in the same context and minimizing it when they occur in different contexts
- idea: get a noise distribution by negative sampling from unrelated parts of the graph

Loss function for negative sampling

- k-step transition matrix: A^k
- Pr(go from node v_i to node v_j in k steps) $= A^k_{i,j}$,
  let this be denoted by $p_k(v_j|v_i)$
- let $p_k(V)$ be the distribution over vertices in the graph
- let c be a node obtained from negative sampling, and let $p_k(c) = \dfrac{1}{N}\sum_{v_i} A^k_{v_i,c}$
- define local loss over a specific pair (v_i, v_j):

$$
\begin{aligned}
L_k(v_i, v_j) &= p_k(v_j|v_i) \cdot \log(\sigma(\Phi(v_i) \cdot \Phi(v_j)) + \lambda \cdot p_k(c) \cdot \log \sigma(-\Phi(v_i) \cdot \Phi(v_j)) \\
&= A^k_{v_i,v_j} \cdot \log(\sigma(\Phi(v_i) \cdot \Phi(v_j)) + \lambda \cdot p_k(c) \cdot \log \sigma(-\Phi(v_i) \cdot \Phi(v_j))
\end{aligned}
$$

Loss function continue

- k-step loss function of a node:
  $L_k(v_i) = \sum_{v_j \neq v_i} L_k(v_i, v_j)$
- k-step loss function over whole graph
  $L_k = \sum_{v \in V} L_k(v)$
- let $e = \Phi(v_i) \cdot \Phi(v_j)$, and set $\dfrac{\delta L_k}{\delta e} = 0$
  we get
  $$\Phi_(v_i) \cdot \Phi(v_j) = \log\left(\frac{A^k_{v_i,v_j}}{\sum_w A^k_{w,v_j}}\right) - \log\frac{\lambda}{N}$$
- let the k-step log probability matrix be X^k, then $X^k_{i,j} \approx \Phi_(v_i) \cdot \Phi(v_j)$ after replacing the negative entries with 0
- X^k can then be factored with SVD or other methods to get $X^k \approx U\Sigma V$
- thus $\Phi \approx U(\Sigma)^{\frac{1}{2}}$

GraRep - Algorithm

GraRep Algorithm

Input
Adjacency matrix S on graph
Maximum transition step K
Log shifted factor β
Dimension of representation vector d

1. Get k-step transition probability matrix A^k
Compute $A = D^{-1}S$
Calculate $A^1, A^2, \ldots, A^K$, respectively
2. Get each k-step representations
For $k = 1$ to K
  2.1 Get positive log probability matrix
  calculate $\Gamma_1^k, \Gamma_2^k, \ldots, \Gamma_N^k$ ($\Gamma_j^k = \sum_p A_{p,j}^k$) respectively
  calculate $\{X_{i,j}^k\}$

$$X_{i,j}^k = \log\left(\frac{A_{i,j}^k}{\Gamma_j^k}\right) - \log(\beta)$$

  assign negative entries of X^k to 0
  2.2 Construct the representation vector W^k
  $[U^k, \Sigma^k, (V^k)^T] = SVD(X^k)$
  $W^k = U_d^k (\Sigma_d^k)^{\frac{1}{2}}$
End for
3. Concatenate all the k-step representations
$W = [W^1, W^2, \ldots, W^K]$

Output
Matrix of the graph representation W

References

Distributed Large-scale Natural Graph Factorization.
Amr Ahmed, Nino Shervashidze, Shravan Narayanamurthy.

GraRep: Learning Graph Representations with Global Structural information.
Shaosheng Cao, Wei Lu, Qiongkai Xu.

DeepWalk: Online Learning of Social Representations.
Bryan Perozzi, Rami Al-Rfou, Steven Skiena

Distributed Representations of Words and Phrases and their compositionality
Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, Jeffrey Dean

Thank you for listening!

Yitao Chen

chen_yitao@gis.a-star.edu.sg