Hashing for Fast and Efficient Node Embedding

Xinfeng Xu
Virginia Tech
Blacksburg, VA, USA
xinfeng@vt.edu

B. Aditya Prakash
Virginia Tech
Blacksburg, VA, USA
badityap@vt.edu

ABSTRACT

Network features are important and are interesting for researchers to produce all sorts of methods to learn them. However, as the network grows larger and larger, the parameter spaces of network features can easily become overwhelming. We present hash node embedding, an efficient method for representing nodes of networks in vector form while still preserve the network neighborhood relations of nodes. Instead of fitting the embedding vectors for each node, these are chosen by the hashing trick from a shared pool of B embedding vectors. Our approach combines two ideas, a 2^{nd} order random walk procedure, which efficiently explores interesting neighborhoods with controlled biases, and hash embedding method, which uses Locality Sensitive Hashing(LSH) to preserve the neighborhood relation while drastically reduced the total parameters needed in the learning step.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

KEYWORDS

Network Embedding, Hashing, Random Walk, Neighborhood Relation

ACM Reference Format:
Xinfeng Xu and B. Aditya Prakash. 1997. Hashing for Fast and Efficient Node Embedding. In *Proceedings of ACM template,* Jennifer B. Sartor, Theo D'Hondt, and Wolfgang De Meuter (Eds.). ACM, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Recent advances in deep learning and feature representation learning in natural language processing have triggered a lot of interesting researches on these fields. Most of the former algorithms are focusing on the accuracy of embedding the feature of graph[2], [4], [9]. For large graph, the nodes to be learnt will be in the order of hundreds of thousands, adding more than millions of parameters to the model. The question then comes out that do we really need all of these parameters to keep the accuracy where the learning time for large graph suffers.
There could be several ways to reduce the feature learning cost when doing embedding. One is to dropout when learning the projection function. This could use the Adaptive Dropout[1] or the Hash Dropout method[5]. The other is instead of learning the embedding vectors for each node, we only learn a total of B embedding vectors for the hash function, then hashing nodes to these vectors. One obvious disadvantage of this kind of hashing is that important nodes may collide. Given a node $u \in V$, hashing this node to one of the "bucket" {1,2,...B} of hash function. This colliding happens if two nodes are assigned to the same bucket, they will get the same vector representation which prevent the model from distinguishing these important nodes.
The solution will be using multiple hash functions instead of one and [7] already shows that the collision can be drastically reduced even when the number of hashing functions increases to just 2.
In this paper, we propose our method based on hashing embedding[7] and when learning the model, and we can use the LSH to preserve the neighborhood information from the graph.

1.1 Points for milestone

1. Difficult: The learning space dimension is very high (See section 3.5). According to [7], they used Keras with Tensorflow backend. I also do the optimization using Keras and lack the time to use LSH to improve the hash functions. But I rpoposed the idea and possible way to do it in section 4.1.2.

2. Reply and changed to milestone: First, I updated section 3.3 to have 2 sets of problem, one is to use random hash functions and the other is to use LSH hash functions. Second, I updated section 4.1, where I detailly described how I want to use random hash functions or LSH in my learning process. Then I added the datasets for the experiments. And updated the second experiement, which is on the community classifications. (section 5.2.2). Finally, I updated the references.

2 RELATED WORK

The related work could be divided into two parts, the network embedding part and the combination of using hashing on embeddings.

Network Embedding: Network embedding is a very important method to learn low-dimensional representations of vertexes in networks. Various methods of network embedding have been proposed in the literature recently, including these following topics: 1. DeepWalk[4]: uses local information obtained from truncated random walks to learn latent representations. 2. Node2vec[2]: get a mapping of nodes to a low-dimensional space of features that maximizes the likelihood of preserving network neighborhoods of

ACM template, Oct 17, CS6604 VT

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

nodes 3. Structural Deep Network Embedding method(SDNE[9]): a deep model which can effectively capture the highly non-linear network structure and preserve the global and local structure of network. 4. Large-scale Information Network Embedding(LINE[8]): aims at producing a scalable network embedding method for large networks.

Dropout: 1. Dropout[6] is a regularization technique that aims at solving the issue of over-fitting by randomly dropping a proportion p($0 < p < 1$) of the nodes in a hidden layer while training the network. Generally, the network's performance becomes worse when too many nodes are dropped. Usually only 50% of the nodes in the network are dropped when training the network. At test time the weights are scaled as $W = pW$ and the resulting neural network is used without dropout. 2. Adaptive dropout[1] is an improvement to the dropout method described above. The methodology samples a small subset of nodes according to the activations of the nodes. The result shows that adoptive dropout demonstrate better performance than random dropout[6]. An important feature of Adaptive Dropout is that you can drop significantly more nodes that random dropout method while still keep a decent performance. 3. Winner-Take-All([3]): This is an extreme form of Adaptive Dropouts that uses mini-batch statistics to enforce a sparsity constraint. Then only the nodes with k% largest, non-zero activations are used during the forward and back-propagation phases of training. For each gradient update, WTA needs to first perform O(n logn) work to sort the activations to find the Active Set(AS) and then update their weights.

Hashing Embedding for Deep Learning[5]: The author presents a novel hashing-based technique to drastically reduce the amount of computations needed to train and test neural networks. All variations of Dropout method,[1], [3], [6] can work to reduce the computation time for the learning process. However, these method require full computations of the activations to sample nodes selectively, which means it is intended for better performance and not for reducing computational cost. Hashing Embedding for Deep Learning [5] combines two ideas, Adaptive Dropout and Randomized Hashing for Maximum Inner Product Search(MIPS), to select the nodes with the highest activation efficiently. They use the insight that selecting a very sparse set of hidden nodes with the highest activations can be reformulated as dynamic approximate query processing problem, which can be solved efficiently using LSH(Locality Sensitive Hashing). As a result, it only requires sub-linear time to determine the active set of nodes.

Hashing Embedding for Word Representation[7]: This can be seen as an interpolation between a standard word embedding and a word embedding created using a random hash function. Rather than fitting the embedding vectors for each token, the embedding vectors are selected by the hashing trick from the hashing function buckets. They used multiple hash functions to avoid frequent collision of important words. They showed that their embedding method exhibit at least the same level of performance as models trained using regular embedding but the number of parameters needed by hashing embedding is only a fraction of what is required by a regular embedding.

Bag of Tricks for Efficient Text Classification[10]: This paper explores a simple and efficient way for text classification. They use a bag of n-grams as additional features to capture some partial information about the local word order. This is very efficient in practice while achieving comparable results to methods that explicitly use the order.

3 PROBLEM FORMULATION

3.1 Random Walk

3.1.1 1^{st} *order Random Walk* : . When embedding nodes of graph, we need to consider the relations between nodes, which are edges or paths in the graph. A straightforward but efficient method to gather edges & paths relationship is to perform random walks on graph to generate sentences. The random walk could start from each node $u \in V$, s times. The probability walking from one node to another will be just dependent on the weight of the edges:

$$P(c_i|c_{i-1}) = \begin{cases} \frac{W(c_{i-1},c_i)}{Z} & if\ (c_{i-1},c_i) \in E \\ 0 & otherwise \end{cases} \quad (1)$$

However, this way is pretty limited method and cares local relations more. (But I will use this method as the first step on the experiments section.)

3.1.2 2^{nd} *order Random Walk* : . In order to generalize the 1^{st} order random walk and capture more information from the graph. We could formalize as follows. Given a source node u, we simulate a random walk of fixed length l. Let c_i be the ith node in the random walk, while c_0=u. The walk will be generated by the following probability distribution:

$$P(c_i|c_{i-1}) = \begin{cases} \frac{\pi_{i-1,i}}{Z} & if\ (c_{i-1},c_i) \in E \\ 0 & otherwise \end{cases} \quad (2)$$

where $\pi_{i-1,i}$ is the walk probability between nodes c_{i-1} and c_i, and Z is the normalization factor. If we do random walk based on the edge weights $w_{i-1,i}$, i.e., $\pi_{i-1,i} = w_{i-1,i}$, this becomes the 1^{st} order random walk mentioned above. Then we adopt the 2^{nd} order random walk with two parameter p and q to get the 2-step neighborhood relations[2]. The walk probability will be set as $\pi_{i-1,i} = \alpha_{pq}(i-2,i) \times w_{i-1,i}$ as equation 3 .

$$\alpha_{pq}(i-2,i) = \begin{cases} \frac{1}{p} & if\ d_{i-2,i} = 0 \\ 1 & if\ d_{i-2,i} = 1 \\ \frac{1}{q} & if\ d_{i-2,i} = 2 \end{cases} \quad (3)$$

where $d_{i-2,i}$ is the shortest path distance between node c_{i-2} and c_i. As in random walk these two nodes are just two steps away, so the shortest path distance will be in the range of [0,2]. Parameter P accounts for the probability that return back to a node already visited and parameter q denotes if the random walk is biased to nodes close to the starting nodes or away to it. Therefore, we can easily adjust the parameters p and q here to set the "favor" of our random walks to be focusing on different information of graph. The notations used in this proposal and their descriptions are summarized in Table 1.

3.2 Locality Sensitive Hashing

Locality Sensitive Hashing(LSH) [6,7,10] is a popular, sub-linear time algorithm for approximately implement the nearest-neighbor search. The main point is to project similar inputs into the same

bucket of a hash table with high probability. An LSH hash function maps an input data vector to an integer key -h(u): $\mathbb{R}^D \rightarrow [0, 1, 2, ..., N]$. A collision happens when the hash result for two elements are equal: h(u) = h(v). The collision probability for an LSH hash function is proportional to the similarity metric between the two elements:

$$Pr[h(u) = h(v)] \propto sim(u, v) \tag{4}$$

In our case, we want to use LSH to let nodes which are neighbors to be projected to same bucket. So the collision probability will be written as:

$$Pr[h(u) = h(v)] \propto 1/dist(u, v) \tag{5}$$

where dist(u,v) means the distance between two nodes from the random walks mentioned in the last section. In this way, nodes which are closer in random walk will be projected to the same bucket with high probability. We call it later as LSH-dist. For the first step, I will use:

$$dist(u, v) = \sum_{L=1,\infty} f(u, v) \times L \tag{6}$$

where f(u,v) is the weighted frequency of node u to v from the random walk sentences through a path length of L. This comes from the intuition that one path length will be considered closer to the actual dist(u,v) if it appears much frequently than other path lengths.

3.3 Learn Hash Functions

Now that we are familiar with the random walks and LSH, we can formally state the first part of problem that we are intended to solve. There are two ways to choose hash functions:
1. Random hash functions with classifications, which preserves the relations given by either ground-truth classifications or other interesting classifications.
2. Hash functions based on LSH, which preserves the neighborhood relations.
Here are the problem fomulation for them:

Problem 1. *Given a graph $G(V, E)$, the feature representation dimension d, and random hash functions H with classification / a hashing method H based on LSH-dist, learn a function from nodes to feature representations $f_H : V \rightarrow \{1, 2, ..., B\}$ such that the classification relations / neighborhood relations are preserved.*

So the first step is to learn the hash functions themselves and we will discuss about constructing and learning the embedding vectors below.

3.4 Learn Hash Embeddings

After we hashed each node in graph to one of the bucket {1,2,...,B} and follows the classification/neighborhood relation. We still need to learn the embedding vector for each bucket. The second part of the problem will be:

Problem 2. *Given a graph $G(V, E)$, the feature representation dimension d, and a hash function from nodes to feature representations $f_H : V \rightarrow \{1, 2, ..., B\}$ which preserves the classification realtions / neighborhood relations, learn embedding vectors for each bucket of hash functions' results.*

Table 1: Summary of symbols and descriptions

Symbol	Description
G	Graph G
V, $\|V\|$	node-set, # of nodes
E, $\|E\|$	edge-set, # of edges
u	$node \in V$
$W(u, v)$	edge weight between node u and v
s, p, q, L	Random Walk parameters
$\mathcal{H}, k$	hash functions, total # of hash functions
B	# of buckets in hash functions' result
d	dimensions for the B shared embedding vectors
E	trainable embedding matrix consisting embedding vectors
P	trainable embedding matrix consisting embedding weights

After solving these two problems, we can successfully embed each nodes in graph to a d-dimensional vector, which preserves either the classification relations or neighborhood relations.

4 PROPOSED METHOD

In this section, we propose our method for solving the problem stated in the previous section.

4.1 For Hash Functions

4.1.1 Random Hash Functions with Classifications. In this first step, we can randomly choose hash functions to project each node $u \in V$ to a bucket $b \in B$, where B is the total number of buckets in the hashing results. In the learning process, we tried to force the hash functions to learn the classifications given to them. The classifictions could be ground-truth categories, i.g. graph node labels. Other classifications can include random walk probabilities, cascade models, etc.

4.1.2 Hash Functions based on LSH. If we have time, we would also try to use LSH to build the hash functions directly, the favored hash functions based on LSH should minimize the loss function as:

$$-\sum_{(u,v)\in E} \{Ph[h(u) = h(v)] - \frac{const}{dist(u, v)}\}, const < 1 \tag{7}$$

where Ph[h(u)=h(v)] is 1 when h(u) = h(v) and 0 if not. The intuition is: if const is closer to 1 and two nodes have dist(u,v) = 1, two nodes projected to same bucket with almost zero loss, while pairs projected to different bucket will gain almost 1 loss. This can force the hash function to preserve the neighborhood relations based on LSH.

4.2 For Hash Embedding

One of the disadvantages of using hashing is that important nodes may collide into one bucket and we can distinguish them. In [7], the authors proved that even use only one more hash functions, the

Algorithm 1 Hash Embedding using Random Hash Functions

Require: Network G, with cascading model IC
Ensure: The node embeddings from hash method
1: Run random walks on G using IC model to generate sentences to Sents.
2: Label each sentence in Sents according to their groud-truth/other classifications.
3: **for** i=0, k **do**
4: **for** onesent in Sents **do**
5: Hash onesent to a bucket according to the Labels
6: Update the loss function using cross entropy
7: Minimize the loss function using Keras

collision problem will be much reduced. Here according to their method, we propose the steps for learning hash embeddings.

- Use k (k=2 for first step) different hash functions $\mathcal{H}_1, ...\mathcal{H}_k$ to choose k vectors for the node $u \in V$ from the hash functions' share pool of B embedding vectors.
- Combine the chosen embedding vectors from step 1 by the weighted sum: $e_u = \sum_{i=1}^{k} p_u^i \mathcal{H}_i(u)$, where $p_u = p_u^1, ...p_u^k$ are the weights for node u for each of the embedding functions.
- Calculate the cross entropy and use stochastic gradient descent method to minimize the cross entropy.

According to the steps above, the total trainable parameters including:

- A trainable embedding matrix E of size B × d, where each row of E is a embedding vector of length d.
- A trainable matrix P of the weight, where each row of P is the weight of a node to all of the hash functions. The size of P is $|V| \times k$.
- k different hash functions $\mathcal{H}_1, ...\mathcal{H}_k$ that they will assign one of the B embedding vectors to each node $u \in V$, and adapt LSH-dist such that close nodes(defined by the 2^{nd} order random walk) will be mapped to same "bucket" with high probability.

As a result, the total trainable parameters in the hashing embedding is equal to $B \times d + |V| \times k$, which should be compared with the standard embedding method where the number of trainable parameters is $|V| \times d$. The number of hash functions k and hash function buckets B can be relatively small while does not influence the performance much. It is already shown that even with k=2, the performance is still good.[7]

5 EXPERIMENT

5.1 Dataset

We use different datasets for different experiments.

Table 2: Datasets for Community Classifications

dataset	nodes	edges	categories
polblogs	1224	16718	2
cornell.cites	195	286	5
washington.cites	230	417	5
wisconsin.cites	265	479	5

Table 3: Datasets for Parameters Classifications

dataset	nodes	edges
$Dutch - College$	32	3062
$High - School$	70	366
$Air - Traffic - Control$	1226	2615

5.1.1 Datasets for Community Classifications. For community classifications, we use networks which contain ground-truth labels for each node. Polblogs dataset contains blogs during 2004 presidential election. Each blog is classified as either Republic or Democracy. Therefore, there are only two categories in this network. The other three datesets are webpage to webpage networks of famous universites, where each webpage belongs to one of the five categories: {project, course, staff, faculty, student} The statistics of these networks are shown in table 2.

5.1.2 Datasets for Parameters Classifications. For parameter classifications, we use the KONECT networks from: http://ko nect.uni-koblenz.de/networks/. The # of nodes are ranging from tens to thousands. We select some of them to show their statistics in table 3.

5.2 Performance Test

We evaluate hash embeddings on the datasets described above for various classification tasks, including model parameter classifications, community classifications. All the models are trained on 50% of the the sentences get from the random walk with the labels provided such as parameters used in this random walk, or community IDs. When testing, we use the entire document as input. We compare our hash embedding method on classifications with a baseline classification method [10].

5.2.1 Parameter Classifictions. In this experiment we compare the standard hashing trick embedding method [Joulin et al 2016] with the hash embedding. We first generate random walks from three different parameters (P=0.1, 0.5, 0.9) using the Independent Cascade(IC) model. Each of these sentences have the label which is the cascading probability it used. The hashing embedding use |V| different importance vectors, k=2 hash functions, B=50 buckets in the hash functions' result, d=20 embedding vector dimensions. This adds up to 2|V|+1000 parameters for the hash embeddings.

We show the test accuracy in Table 4. We can see our hash embedding method outperforms the standard embedding method, especially in samll networks. The accuracy dropped when going to large networks, but hash embedding still gave acceptable results.

Table 4: Test accuracy (in %) for Parameter Classifictions

Datasets Accuracy	Hash Embedding	Standard Embedding
Dutch – College	**97.7**	89.3
High – School	92.3	73.0
Air – Traffic – Control	72.9	68.1

Table 5: Test accuracy (in %) for Community Classifictions

Datasets Accuracy	Hash Embedding	Standard Embedding
polblogs	97.0	98.1
cornell.cites	75.2	86.8
washington.cites	80.0	74.0
wisconsin.cites	81.2	87.2

Table 6: Running time (in seconds) for Parameter Classifictions

Datasets Time	Hash Embedding	Standard Embedding
Air – Traffic – Control	**22.2**	49.7
Dutch – College	**7.00**	11.9
High – School	**10.4**	18.4

5.2.2 Community Classifications. In this experiment we compare our hash embedding with the standard hashing trick embedding method on the community classifications of given datasets. We have the ground-truth community for each of the node in the network. We run hash embedding and standard embedding on the datasets to get embedding of each node. After that, we can test other cascades' community from our embedding vectors. We show the test accuracy in Table 5. Basically, the hash embedding method gives comparable accuracy whilethe runnning time drops at most around 50%. Therefore, hash embedding is an efficient way compared with standard embedding method.

5.3 Time Complexity

The theoretical time complexity for standard embedding method is $|V| \times d$, where d is the dimension of vector space, for the hash embedding we proposed, it will be $B \times d + |V| \times k$. For a typical dataset with |V|=1000 nodes, k=2 hash functions and B=100 buckets of dimension d = 20. The standard embedding method will need 20K parameters while for hash embedding there will be 4K parameters. As a result, from theoretical view, our method of hash embedding will drastically decrease the time complexity especially for large graph.

We show the running time of parameter classifications and community classifications for selected datasets in Table 6 and 7. The hash embedding did show a shorter time compared with the hashing embedding method. As a result, hash embedding saves time while gives comparable accurate results.

6 TIME-LINE

Not needed here.

Table 7: Running time (in seconds) for Community Classifictions

Datasets Time	Hash Embedding	Standard Embedding
polblogs	**4.6**	4.8
cornell.cites	**5.5**	11.1
washington.cites	**6.4**	7.3
wisconsin.cites	**5.8**	11.4

7 IMPORTANT ALGORITHMS OR TOOLS

As the paramester space dimension is very high, even in our method it could be 4k parameters for |V| = 1000 nodes network. (See section 5.3), I decided to use the Keras to do the learning in section 4.2, where I need to calculate the cross entrypy of the whole tranning embedding matrix E and P. This tool has been used in [7] and get a good performance.

8 CONCLUSIONS & FUTURE WORK

In this paper, we present the method of hash embedding on networks. Hash embedding is an efficient way (shown in section 5.2), while still keeps the comparable accuracy to standard embedding method. There are some incomplete tasks, includes:

1. We show the 2^{nd} order random walk in section 3.1.2, while could be a improvement to current 1^{st} order random walk. 2^{nd} order random walk can capture the relations that are two-steps away and can adjust the parameter p and q to let the model focusing on different structures of the network (See section 3.1.2 for more details).

2. We lack the time to adopt the LSH to build the hashing function (section 4.1.2), but this seems to be a good way to use the structure of network itself to get the hash fucntions instead of learn random hashing functions. For all the experiments we presented, we used random hash functions and then learn it through the optimizing process to preserve the classifications we wanted.

3. The experiemtns are running on comparably small networks. We can test the efficiency on larger graphs in future and compare with standard embedding to see the difference.

9 TEAM WORK PROPORTION

I work alone as a group, so I will do all of this project and may get help from Prof. Aditya and Bijaya.

REFERENCES

[1] Jimmy Ba and Brendan Frey. 2013. Adaptive dropout for training deep neural networks. In Advances in Neural Information Processing Systems 26, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.). Curran Associates, Inc., 3084âĂŞ3092. http://papers.nips.cc/paper/5032-adaptive-dropout-for-training-deep-neural-networks.pdf

[2] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. InProceedingsof the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2016.

[3] Alireza Makhzani and Brendan J. Frey. 2014. A Winner-Take-All Method for Training Sparse Convolutional Autoencoders. CoRR abs/1409.2752 (2014). In NIPS, 2015.

[4] B. Perozzi, R. Al-Rfou, and S. Skiena. Deepwalk:Online learning of social

representations. InProceedings of the 20th ACM SIGKDD internationalconference on Knowledge discovery and data mining, pages 701 −710. ACM, 2014.
[5] R. Spring and A. Shrivastava. Scalable andsustainable deep learning via randomized hashing. InProceedings of the 23rd ACM SIGKDD InternationalConference on Knowledge Discovery and Data Mining,pages 445–454. ACM, 2017.
[6] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov,R., 2014. Dropout: A simple way to prevent neural networks from over-fitting. Journal of Machine Learning Research 15, 1929–1958.
[7] D. Svenstrup, J. Meinertz Hansen, and O. Winther. 2017. Hash Embeddings for EfficientWord Representations. In NIPS, 2017
[8] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei.LINE: Large-scale Information Network Embedding. In WWW, 2015.
[9] Daixin Wang, Peng Cui, and Wenwu Zhu. 2016. Structural Deep NetworkEmbedding. In ACM SIGKDD.
[10] Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. (2016b). Bag of tricks for efficient text classification. CoRR, abs/1607.01759.