JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

An Overview on Graph Network along with Deep Walk algorithm

[1]V.Karthick, [2]Mohamed Kassim, [2]V.Vidhya Lakshmi, [2]G.Aishwarya

[1.2]Asst Prof, Department of Mathematics, Agni College of Technology, Chennai - , Tamil Nadu.
[1]karthicklazer@gmail.com,
[2]mohamedkassim1236@gmail.com, [2]kalia.vidhya30@gmail.com, [2]aishusridevi23@gmail.com

ABSTRACT:

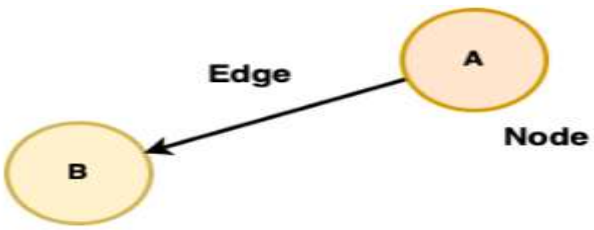
Networks provide an effective mechanism for exchanging data between processors in a parallel computing system. Embedding problems have gained importance in the field of interconnection networks for parallel computer architectures. The Research in this area is concentrating on discovering efficient heuristic that find solution in most cases. I will provide a high-level overview of graph networks, Word2Vec / Skip-Gram, and the Deep Walk process. To help with this, I'll present a multi-class classification example to walk through the algorithm. After that, I will consider different parameter configurations and show their impact on the performance of the algorithm. Lastly, I'll outline some considerations for deployment and handling unseen data within the system.

Keywords – ***Network, Embedding, Deepwalk algorithm.***

PRELIMINARY:

1.1 A Graph is a pair of sets. G = (V,E). V is the set of vertices. E is a set of edges. E is made up of pairs of elements from V (unordered pair).A Digraph is also a pair of sets. D = (V,A). V is the set of vertices. A is the set of arcs. A is made up of pairs of elements from V (ordered pair).

In the case of digraphs, there is a distinction between `(u,v)` and `(v,u)`. Usually the edges are called arcs in such cases to indicate a notion of direction.



TERMINOLOGY:

- The vertices u and v are called the end vertices of the edge (u,v).
- If two edges have the same end vertices they are Parallel.
- An edge of the form (v,v) is a loop.
- A Graph is simple if it has no parallel edges and loops.

- A Graph is said to be Empty if it has no edges. Meaning E is empty.
- A Graph is a Null Graph if it has no vertices. Meaning V and E is empty.
- A Graph with only 1 Vertex is a Trivial graph.
- Edges are Adjacent if they have a common vertex. Vertices are Adjacent if they have a common edge.
- The degree of the vertex v, written as d(v), is the number of edges with v as an end vertex. By convention, we count a loop twice and parallel edges contribute separately.
- Isolated Vertices are vertices with degree 1. d(1) vertices are isolated.
- A Graph is Complete if its edge set contains every possible edge between ALL of the vertices.
- A Walk in a Graph G = (V,E) is a finite, alternating sequence of the form ViEiViEi consisting of vertices and edges of the graph G.
- A Walk is Open if the initial and final vertices are different. A Walk is Closed if the initial and final vertices are the same.
- A Walk is a Trail if ANY edge is traversed at most once.
- A Trail is a Path if ANY vertex is traversed at most once (Except for a closed walk).
- A Closed Path is a Circuit – Analogous to electrical circuits.

1.2 Network graph (force directed graph) is a mathematical structure (graph) to show relations between points in an aesthetically-pleasing way. The graph visualizes how subjects are interconnected with each other. Entities are displayed as nodes and the relationship between them are displayed with lines. The graph is force directed by assigning a weight (force) from the node edges and the other interconnected nodes get assigned a weighted factor. The graph simulates the weight as forces in a physical system, where the forces have impact on the nodes and find the best position on the chart's plotting area.

1.3 An **embedding** of a guest graph G = (V, E), onto a host graph H = (W, E), is a one to-one mapping η: V → W in conjunction with a map μ which assigns each edge (u, v) ∈ E to path in H from η(u) to η(v). When | V |>| W | we allow many-to-one mappings, but limit the mapping so that each host vertex is the image of no more than |V |/ |W| guest vertices. The quantity |V |/ |W| is called the load of the embedding.

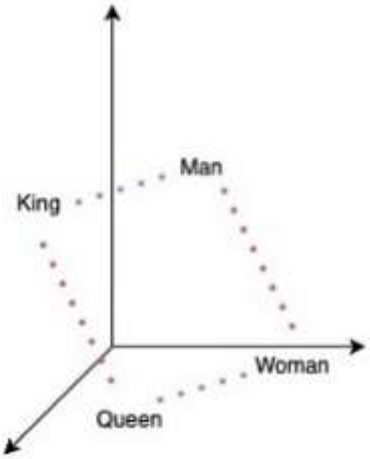
Main Results

THE DEEPWALK ALGORITHM:

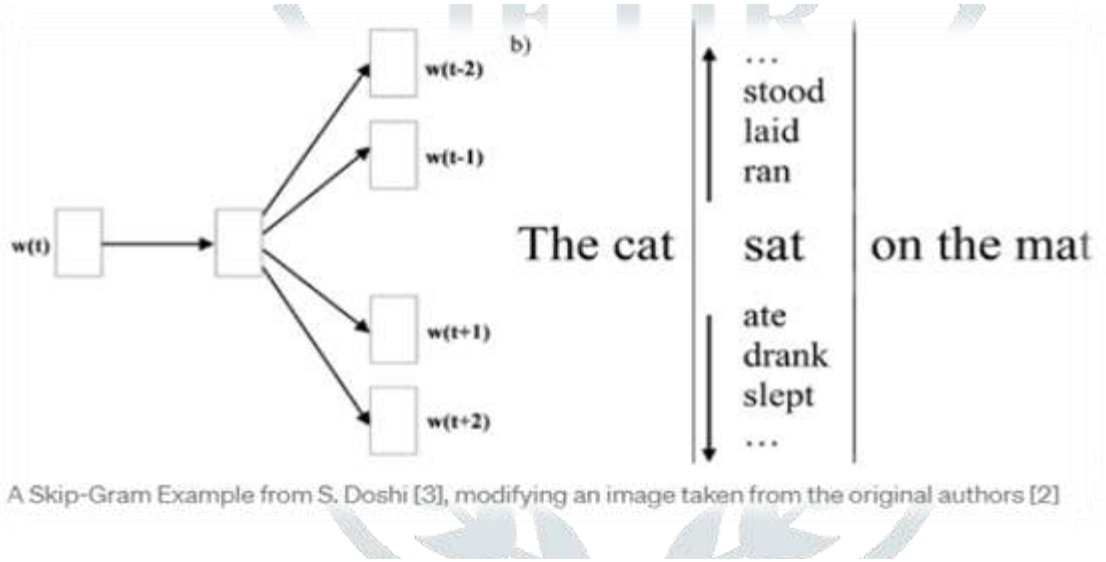
DeepWalk is a type of graph neural network *[1]* — a type of neural network that operates directly on the target graph structure. It uses a randomized path traversing technique to provide insights into localized structures within networks. It does so by utilizing these random paths as sequences that are then used to train a Skip-Gram Language Model. For simplicity in this article we will use the Gensim package Word2Vec to train our Skip-Gram model.

Word2Vec:

This simple implementation of the DeepWalk algorithm relies heavily on the Word2Vec language model [2]. Presented by Google in 2013, Word2Vec allowed for words to be embedded into n-dimensional space, with similar words being locally situated near each-other. This means that words that are often used together / used in similar situations would have smaller cosine distances.



Word2Vec does this by using the **Skip-Gram** algorithm to compare the target words with its context. At a high level, Skip-Gram operates using a sliding window technique — where it tries to predict the surrounding words given the target word in the middle. For our use-case of trying to encode similar nodes within the graph to be close to each-other in n-dimensional space, this means that we are effectively trying to guess the neighbors around the target node within our network.



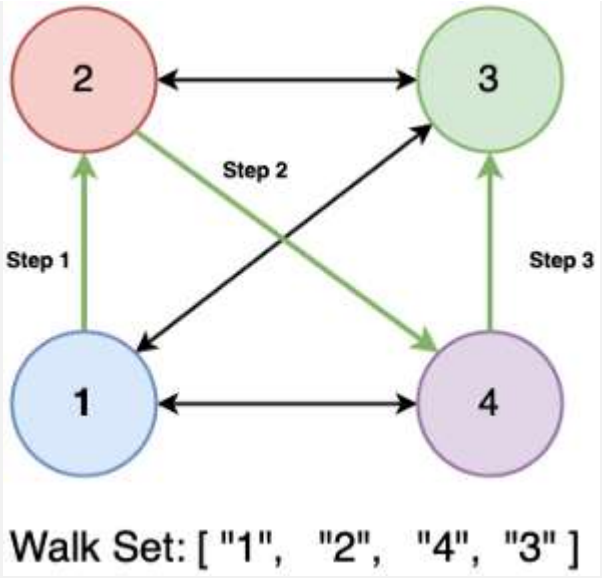
A Skip-Gram Example from S. Doshi [3], modifying an image taken from the original authors [2]

DeepWalk

DeepWalk utilizes random path-making through graphs to reveal latent patterns in the network; these patterns are then learned and encoded by neural networks to yield our final embeddings. These random paths are generated in an extremely simple manner: Starting from the target root, randomly select a neighbor of *that* node, and add it to the path, next you randomly choose a neighbor of *that* node and continue through the walk until the desired number of steps has been taken. Using the e-commerce example, this repeated sampling of network paths yields a list of product-ids. These ID's are then treated as if they were tokens in a sentence, and the state-space is learned from them using a Word2Vec model. More succinctly, the DeepWalk process follows the following steps:

The DeepWalk process operates in a few steps:

1. For each node, perform N "random steps" starting from that node
2. Treat each walk as a sequence of node-id strings
3. Given a list of these sequences, train a word2vec model using the Skip-Gram algorithm on these string sequences



What does this look like in code?

We start with the core network data structure defining a series of products given by their ID. Vertices between two products are products that were co-purchased together within the e-commerce ecosystem. Let us first start by defining a function **get_random_walk** (Graph, Node_Id):

```
# Instantiate a undirected Networkx graph
G = nx.Graph()
G.add_edges_from(list_of_product_copurchase_edges)
def get_random_walk(graph:nx.Graph, node:int, n_steps:int = 4)->List[str]:
    """ Given a graph and a node,
        return a random walk starting from the node
    """
         local_path = [str(node),]
        target_node = node

    for _ in range(n_steps):
        neighbors = list(nx.all_neighbors(graph, target_node))
        target_node = random.choice(neighbors)
        local_path.append(str(target_node))

        return local_path

walk_paths = []for node in G.nodes():
    for _ in range(10):
        walk_paths.append(get_random_walk(G, node))

walk_paths[0]
>>> ['10001', '10205', '11845', '10205', '10059']
```

What these random walks provide to us is a series of strings that act as a path from the start node — randomly walking from one node to the next down the list. What we do next is we treat this list of strings as a sentence, then utilize these series of strings to train a Word2Vec model

```
# Instantiate word2vec model
embedder = Word2Vec(
    window=4, sg=1, hs=0, negative=10, alpha=0.03, min_alpha=0.0001,
    seed=42
)
```

```
# Build Vocabulary
embedder.build_vocab(walk_paths, progress_per=2)

# Train
embedder.train(
   walk_paths, total_examples=embedder.corpus_count,
epochs=20,
   report_delay=1
)
```

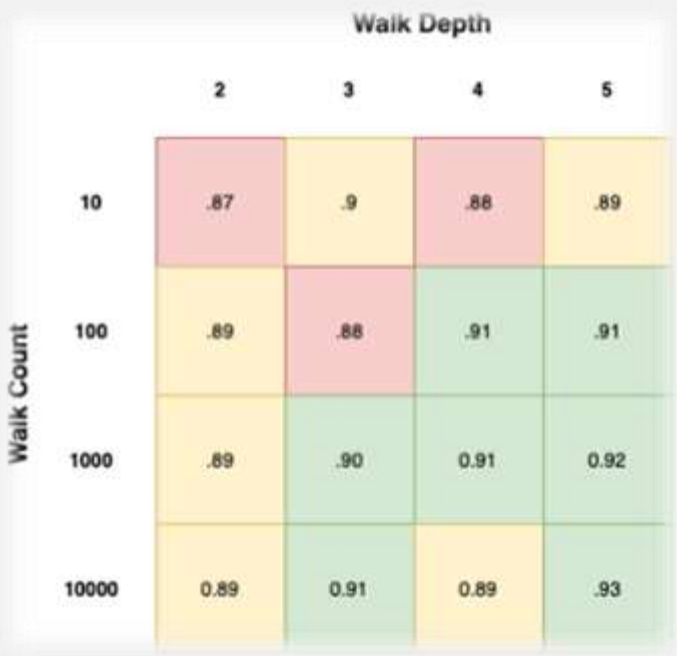
Tuning and Parameters

Now that we have the basic structure of our DeepWalk, there are lots of aspects that are parametrizable outside of the general model params from our Word2Vec model. These may include:

1. Number of random walks performed for the W2V training data

2. Depth of each walk taken from the node

I'll utilize a general classification approach on a sample dataset to show how these parameters can affect the performance of your model. In the graph described above — utilizing a series of products, with a graph defining co-purchased products — we seek to classify the products into their 10 respective categories.



Above shows the classification performance (in accuracy) of the classifier trained on the node vectors from our Word2Vec model using increasing numbers of random walks on the y-axis, and increasing random walk depth on the x-axis. **What we see is that accuracy increases as both parameters increase, but show a diminishing rate of returns as they both increase upwards**. One thing to note is as the walk count increased, the training time **increased linearly,** so training times can explode as the number of walks increase. For example, the difference in training time from the top left corner took only 15 seconds, while the bottom right corner took well over an hour.

Deploying the System: Warm-Start Retraining

So now that we know how it behaves, how do we make it practical? In most graph systems the core issue is updating and maintaining systems without having to retrain the whole model at once. Thankfully, due to Deep

Walks' relationship with NLP, we can rely on similar update procedures. When utilizing genism, the update algorithm is even simpler and follows the process:

- Add the target node to the graph
- Perform random walks from that node
- Using those sequences, to update the Word2Vec Model

```
# Add new nodes to graph from their edges with current nodes
G.add_edges_from([new_edges])

# From a list of the new nodes' product_ids (unknown_nodes) get rw's
sequences = [get_random_walks(G, node) for node in unknown_nodes]

model.build_vocab(new_nodes, update=True)
model.train(sequences,total_examples=model.corpus_count,
epochs=model.epochs)
```

Alternatively, there is an even easier way to handle new nodes in the system. You can utilize the known embeddings of the model to extrapolate the embedding of the unknown node. Following a similar pattern to the previous implementation:

- Add the target node to the graph
- Perform random walks from that node
- Aggregate the embeddings from the random walks, then use that aggregation to stand-in as the unknown nodes embedding

```
# Add new nodes to the graph from their edges with current nodes
G.add_edges_from([new_edges])

sequences = [get_random_walk(G, node) for node in unknown_nodes]

for walk in sequences:
   nodes_in_corpus = [
        node for node in walk if node in word2vec
   ]
   node_embedding = [ # here we take the average of known embeddings
        np.mean(embedder[nodes_in_corpus])
   ]
```

CONCLUSION:

Thus, our embedding is an average of the known nodes in the random walks starting from the unknown node. The benefit to this method is its speed, we don't need to retrain anything, and are performing several $O(1)$ calls to a dictionary-like data structure within Gensim. The drawbacks are its inexactness, without retraining the model, the interactions between the new node and its neighbors are approximated, and are only as good as your aggregation function. For more insight into these methods, you can check out papers that discuss the efficacy of such aggregations, like TF-IDF averaging, etc.

In the past 10 minutes, we've walked through the core components of DeepWalk, how to implement it, and some considerations for implementing it into your own work. There are many possibilities to consider for evaluation of graph networks, and given its simplicity and scalability, DeepWalk should certainly be

considered amongst other algorithms available. Below you can find some references to the algorithms outlined above, as well as some further reading regarding word embeddings.

REFERENCES

[1]Perozzi et Al **Deep Walk – Online Learning of social Representations:** https://arxiv.org/pdf/1403.6652.pdf
[2]Mikolov et Al **Efficient Estimation of world representation in vector space:**https://arxiv.org/pdf/1301.3781.pdf
[3]S.Doshi-**Skip Gram NLP context words prediction algorithm:** https://towardsdatascience.com/skip-gram-nlp-context-words-prediction-algorithm-5bbf34f84e0c?gi=8ff2aeada829
[4] Levy et Al. **Improving Distributional Similarity with Lessons Learned from Word Embedding's:** https://www.aclweb.org/anthology/Q15-1016/
[5] V. Chaudhary and J. K. Aggarwal. **Generalized mapping of parallel algorithms onto parallel architectures.** In D. A. Padua, editor, Proceedings of the 1990 International Conference on Parallel Processing. Volume 2: Software. Pages 137{141. Pennsylvania State University Press, 1990.
[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson**. Introduction to Algorithms. McGraw-Hill Higher Education**, 2nd edition, 2001.
[7] J.Xu.**Topological Structure and Analysis of Interconnection Networks**. Kluwer Academic Publishers, 2001.
[8] Abdulaziz Alashaikh, Teresa Gomes, David Tipper**, "The Spine concept for improving network availability"**, Elsevier, Computer Networks, Volume 82, 8 May 2015, Pages 4-19.https://doi.org/10.1016/j.comnet.2015.02.020.