Photo by Fabio Bracht on Unsplash

Member-only story

DeepWalk: Its Behavior and How to Implement It

A cheat sheet for quickly analyzing and evaluating relationships in graph networks using Python, Networkx, and Gensim

Nick Hespe · Follow
Published in Towards Data Science
8 min read · Sep 11, 2020

Listen Share More

The ability of graph data structures to represent complex interactions has led to new ways to analyze and classify entities defined by their co-interactions. While these analyses are powerful at finding different structures within communities, they lack the ability to encode aspects of the graph for input into conventional machine learning algorithms. With the proposal of DeepWalk, [1] co-interactions within graphs can be captured and encoded by simple neural networks into embeddings consumable by the aforementioned ML algorithms. Yet while there are articles presenting simple introductions to the DeepWalk algorithm, few that I could

In this short article, I will provide a high-level overview of graph networks, Word2Vec / Skip-Gram, and the DeepWalk 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.

Graph Networks

Graphs are data structures that efficiently represent the interactions between two or more objects in an ecosystem. The structure is defined by two objects, a node *or* vertex that defines the entities within the system. For this article I'll use an example of a network of purchases on an e-commerce website, where the nodes in the graph are the products being sold.

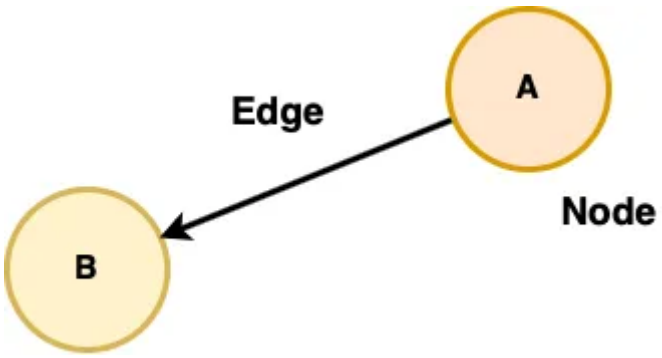


Image by author

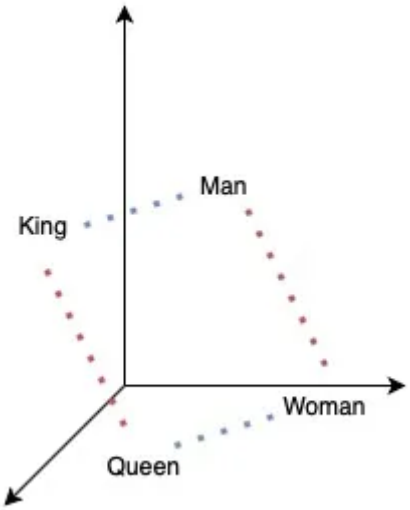
The other aspect of graphs is what links them: an *edge*, which defines the interaction that connects two nodes. An edge can be directed, showing a to-from relationship — Imagine person A sent an email to person B. An edge may also have a *weight* which defines their interaction. In our case, we could define an edge weight that represents the proportion of consumers who bought *both* of the products on our e-commerce website.

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.

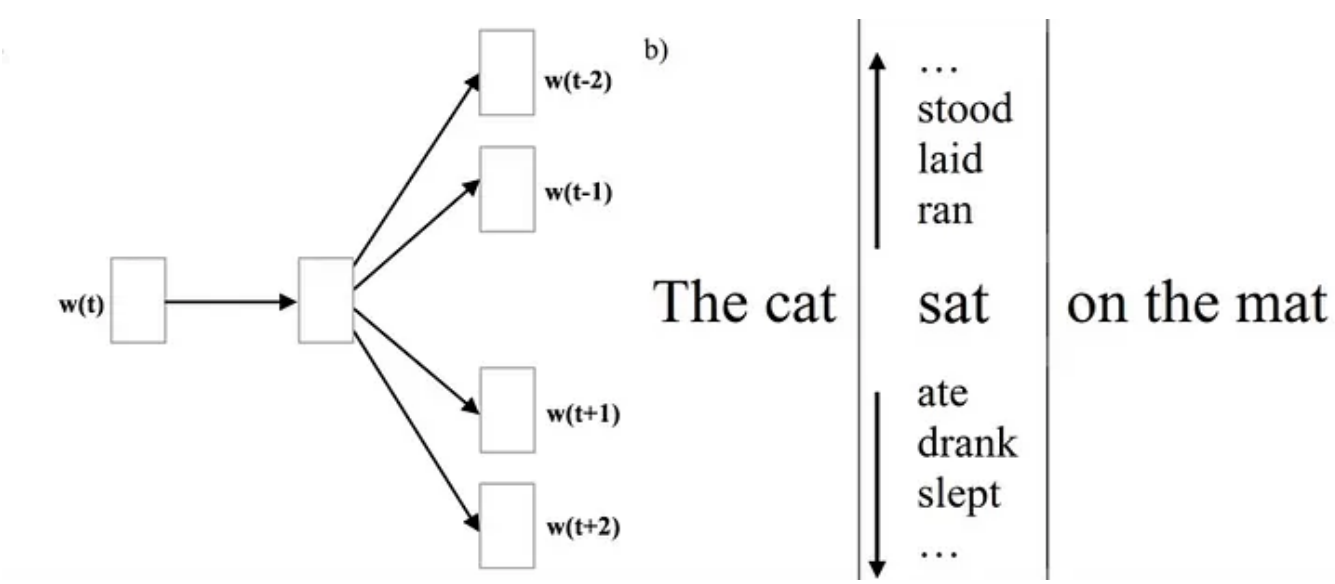
Word2Vec

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.



3-Dim Example of Word Embedding Behavior, Image by author

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

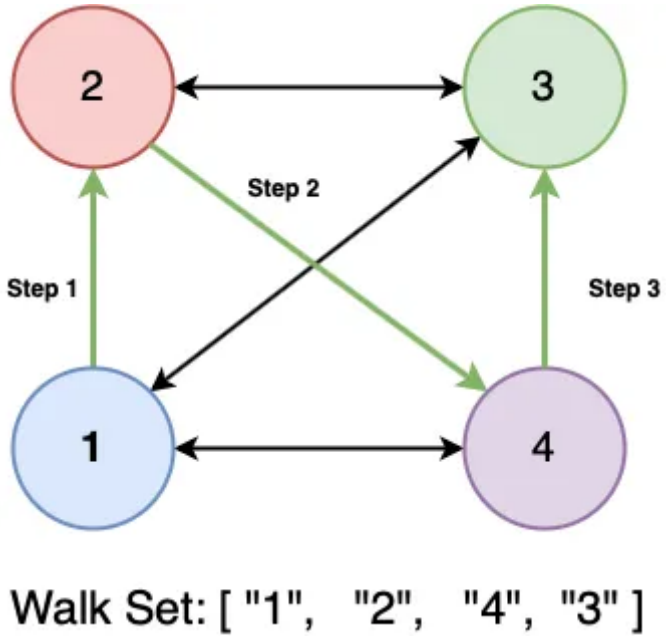


Image by author

What does this look like in code?

We start with the core networkx 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. First, we're going to define 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.

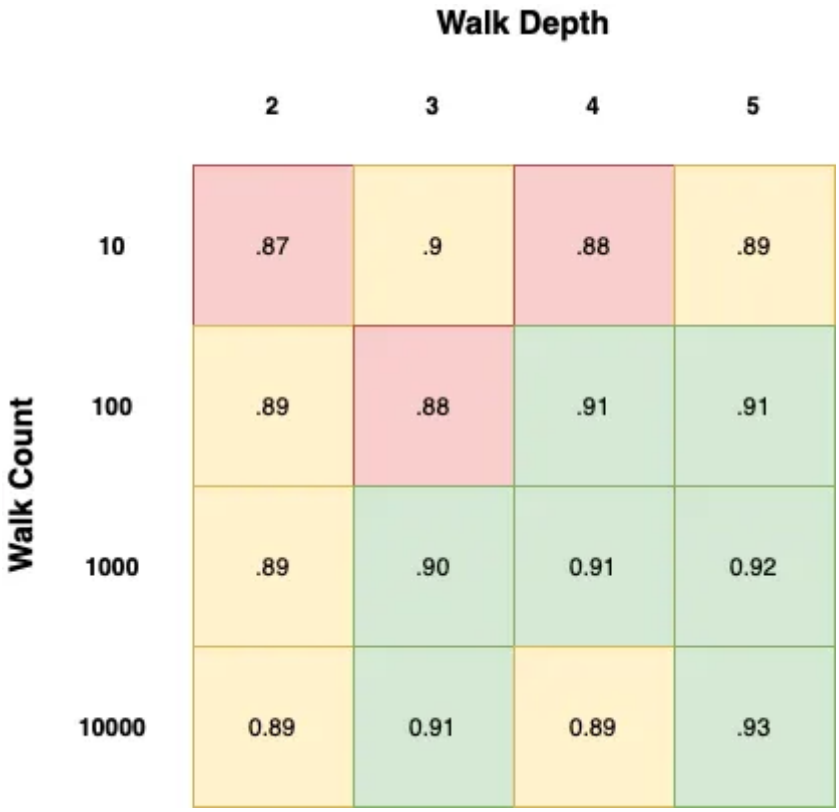


Image by author

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 DeepWalks' relationship with NLP, we can rely on similar update procedures. When utilizing gensim, the update algorithm is even more simple and follows the process:

1. Add the target node to the graph
2. Perform random walks from that node
3. 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)
```

No Retraining Necessary

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:

1. Add the target node to the graph
2. Perform random walks from that node
3. 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])
    ]
```

Here, 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. [4]

In the past 10 minutes, we've walked through [*hah.*] 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.

Sources:

[1] Perozzi et Al. **DeepWalk: Online Learning of Social Representations** https://arxiv.org/pdf/1403.6652.pdf

[2] Mikolov et Al. **Efficient Estimation of Word Representations 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 Embeddings** https://www.aclweb.org/anthology/Q15-1016/

Network | Gnn | Word2vec | Graph Neural Networks | Graph Embedding