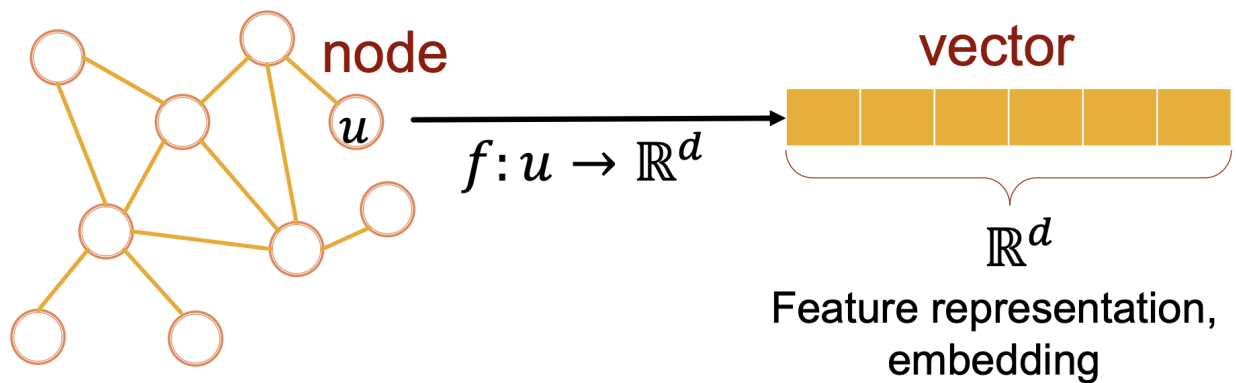


Lecture 2: Node Embeddings

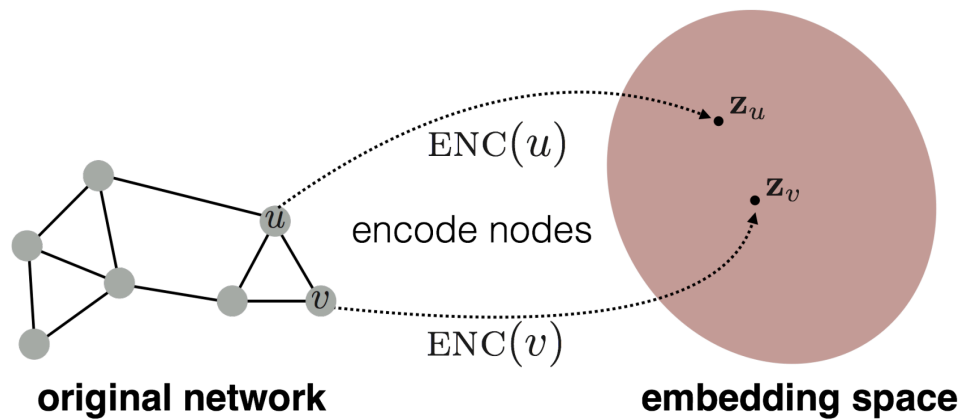
Introduction: In the past, we have used traditional ML techniques to complete downstream prediction tasks: Given an input graph, we extract node, link, and graph-level features and then learn a model that maps features to labels. Just like in the world of NLP, the recent focus of Graph representation learning has been to do efficient task-independent feature learning for machine learning with graphs.



With node embeddings, we can:

- Find the similarity between nodes
- Encode network information
- Complete many downstream tasks

Encoder & Decoder: When discussing node embeddings, we need to encode nodes according to their similarity in the graph.



Specifically, we want:

$$similarity(u, v) \approx z_v^T z_u$$

Where v and u are nodes in the map, the left side denotes the similarity in the original network and the right side gives the similarity of the embedding.

Encoder: maps nodes to embeddings

Decoder: maps from embeddings to the similarity score

Goal: Optimize the parameters of the encoder so that: $similarity(u, v) \approx z_v^T z_u$

Let us examine the simplest encoding approach: “**Shadow Encoding**”, where we have a matrix \mathbf{Z} , in which each column is a node embedding. We simply just look up the embedding vector for the specific node.

$$ENC(v) = z_v = \mathbf{Z} \cdot v$$

$$\mathbf{Z} \in \mathbb{R}^{d \times |V|}$$

$$v \in \mathbb{I}^{|V|}$$

How do we come up with these embedding vectors? How do we define node similarity?

Random Walks:

We want to create node embeddings that harness some sort of idea of similarity. We want these node embeddings to be independent of node labels, node features, and downstream tasks. We can think of similarity in many ways: are the nodes linked? Do the nodes share neighbors? Do the nodes have similar structural roles? This is where we get the idea of Random Walks.

Random Walk Algorithm: Given a graph and a starting point, we randomly move to a neighbor with an equal probability of it being any neighbor, and we continue this process until we have explored as much as we had set out to. The output of this algorithm is the sequence of points that were visited on this walk. This algorithm is expressive, gives low and high-level information, and is computationally efficient. We want $\mathbf{z}_u^T \mathbf{z}_v \approx$ the probability that u and v co-occur on a random walk together.

To get node embeddings using this strategy, we have two main steps:

- 1) Estimate the probability of visiting node v on a random walk starting from node u using some random walk strategy R
- 2) Optimize embeddings to encode these random walk statistics.

For the second step, given a node u, we want to learn feature representations that are predictive of the nodes in its random walk Neighborhood ($N_R(u)$)

Given graph G with Vertex set V and Edge set E, the goal is learn a function f, that gives the embedding of v, given u. We want to optimize such that we find the arguments that maximize the log-likelihood objective. Specifically:

$$\arg \max_{\mathbf{z}} \sum_{u \in V} \log P(N_R(u) | \mathbf{z}_u)$$

Random Walk Optimization:

- 1) Run short fixed-length random walks starting from each node u in the graph using some random walk strategy R .
- 2) For each node u , collect $N_R(u)$, which is the multiset of nodes visited on random walks starting from u .
- 3) Optimize embeddings according to the maximum likelihood objective.

We want to do a couple of things to our objective function: 1) take the argmin of the negative of the function and 2) parameterize $P(v|z_u)$ using softmax. Here is the overall equation:

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} - \log \left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)} \right)$$

Diagram illustrating the components of the equation:


- $\sum_{u \in V}$: sum over all nodes u (indicated by a blue arrow)
- $\sum_{v \in N_R(u)}$: sum over nodes v seen on random walks starting from u (indicated by a red arrow)
- $\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}$: predicted probability of u and v co-occurring on random walk (indicated by an orange arrow)

This computation is extremely expensive though due to the normalization constant in the denominator, we turn to negative sampling.

Negative Sampling: Instead of normalizing with respect to all nodes, just normalize against k random “negative samples” which allows for quick likelihood calculation. Two considerations for k (ie the number of negative samples) - 1) higher k gives more robust estimates 2) higher k corresponds to higher bias on negative events. Most researchers choose k to be between 5 and 20 (inclusive).

$$\log\left(\frac{\exp(\mathbf{z}_u^T \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^T \mathbf{z}_n)}\right)$$

random distribution
over nodes



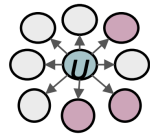
$$\approx \log\left(\sigma(\mathbf{z}_u^T \mathbf{z}_v)\right) + \sum_{i=1}^k \log\left(\sigma(-\mathbf{z}_u^T \mathbf{z}_{n_i})\right), n_i \sim P_V$$

How do we optimize, now that we have received our objective function?

Objective function: $L = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|z_u))$

We want to use Gradient Descent to minimize L, specifically stochastic gradient descent, which evaluates gradients over individual training examples. In SGD, we initialize z_u at some randomized value for all the nodes. And then iterate the following till convergence: sample a node u, and for all other nodes v calculate the gradient of the Loss function with respect to z_v and then update z_v .

NODE2VEC: We next discuss another way of getting node embeddings, which incorporates a flexible, biased random walk that can trade off between local and global views of the network. As there are two ways of exploring a neighborhood, Depth First Search and Breadth First Search, we need to understand how much we want to use either of them in our random walks. Whereas BFS gives us a local microscopic view, DFS gives a global macroscopic view.



BFS:

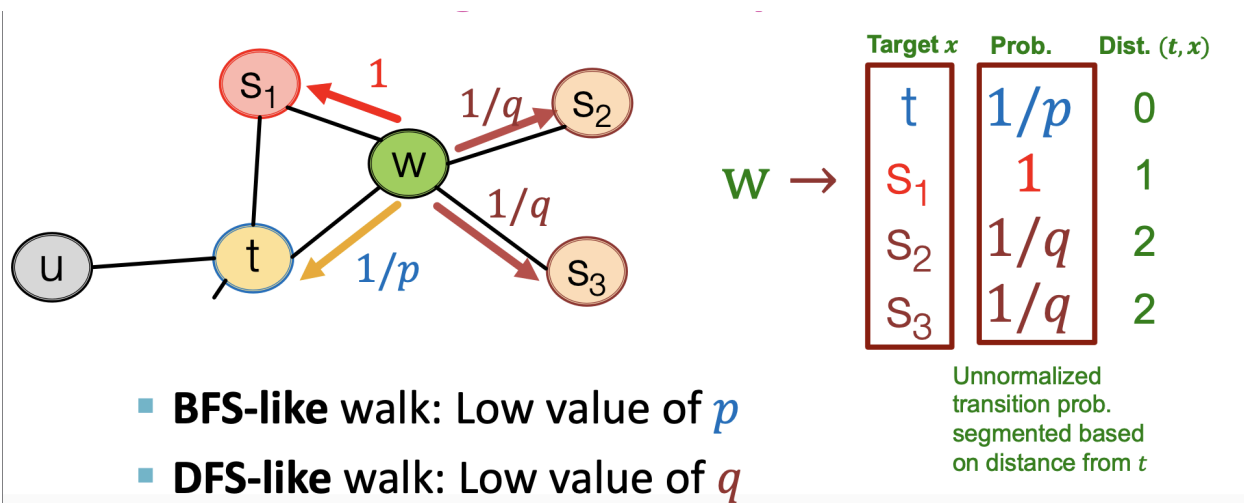
$N_R(\cdot)$ will provide a micro-view of neighbourhood



DFS:

$N_R(\cdot)$ will provide a macro-view of neighbourhood

In this new biased random walk, we use two new parameters p and q . p is the return parameter - it gives the probability that the walk returns to the previous node. q is the in-out parameter which gives the BFS vs DFS ratio. When there is a low value of p , that means the walk biases more towards a BFS approach. When there is a low value of q , that means the walk biases more towards a DFS approach. This is called a second order random walk because it has to keep track of where the node came from.



Overall Node2Vec algorithm: Compute edge transition probabilities, simulate r random walks of length l starting from each node u , and optimize the node2vec objective using SGD. This algorithm completes in linear-time and all three steps are parallelizable.

There are other random walk ideas that exist: biased random walks on different attributes like node attributes or learned weights, alternative optimization schemes, and network preprocessing techniques.

When choosing a random walk strategy, one must take the situation/problem into account.

Embedding Entire Graphs: What if we want to embed an entire subgraph or graph? For example, to classify the toxicity of a molecule or identify anomalous graphs. There are two approaches: 1) embed the nodes and sum/avg their embeddings 2) create a super-node that spans the entire sub or regular graph and then embed the node. Also, we can hierarchically cluster nodes in graphs, and sum/avg the node embeddings, in an approach called DiffPool.

Matrix Factorization and Node Embeddings:

Recall that we want to maximize $z_v^T z_u$ for node pairs (u, v) that are similar. What if we define that two nodes are similar if they are connected by an edge. This would mean that

$z_v^T z_u = A_{u,v}$ where A is the adjacency matrix. Therefore, $Z^T Z = A$. Whereas this is generally

not possible, we can create an objective function to minimize the distance between A and $Z^T Z$.

This objective function would be $\min_z ||A - Z^T Z||$. This gives us the conclusion that inner product decoder with node similarity defined by edge connectivity is equivalent to matrix factorization for A.

Random Walk-based Similarity: DeepWalk and node2vec have more complex similarity notions than random walks. DeepWalk is equivalent to matrix factorization of the following complex matrix expression:

Volume of graph

$$\text{vol}(G) = \sum_i \sum_j A_{i,j}$$

Diagonal matrix D

$$D_{u,u} = \text{deg}(u)$$

$$\log \left(\text{vol}(G) \left(\frac{1}{T} \sum_{r=1}^T (D^{-1}A)^r \right) D^{-1} \right) - \log b$$

context window size

See Lec 3 slide 30:

$$T = |N_R(u)|$$

**Power of normalized
adjacency matrix**

**Number of
negative samples**

How do we use embeddings?

- clustering/community detection
- Node classification
- Link prediction
- Graph classification

Limitations:

- 1) Limitations of node embeddings via matrix factorization and random walks
 - a) Transductive method (need to do a new run of DeepWalk or node2vec for new graphs and unseen nodes)
- 2) DeepWalk and node2vec do not capture structural similarity
- 3) Cannot utilize node, edge, and graph features