Online Social Networks and Media

Graph ML II

Graph Embeddings

Graph Machine Learning

Outline

Part I: Introduction, Traditional ML

Part II: Graph Embeddings

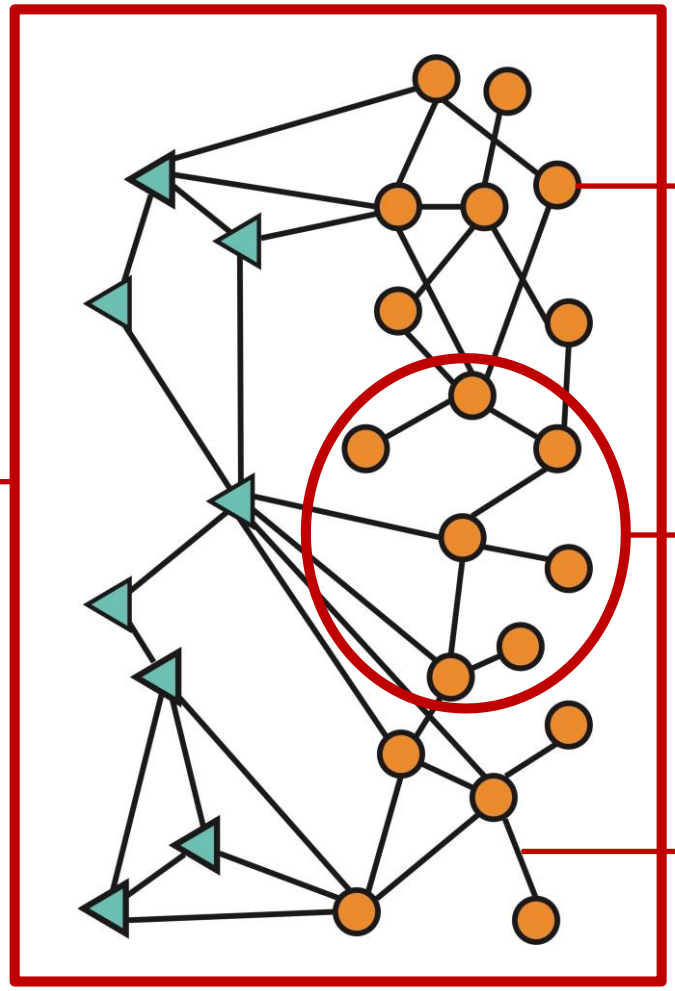
Part III: GNNs

Part IV (if time permits): Knowledge Graphs

Slides used based on:

CS224W: Machine Learning with Graphs
Jure Leskovec, Stanford University
http://cs224w.stanford.edu

Types of ML tasks in graphs



Node level

Community (subgraph) level

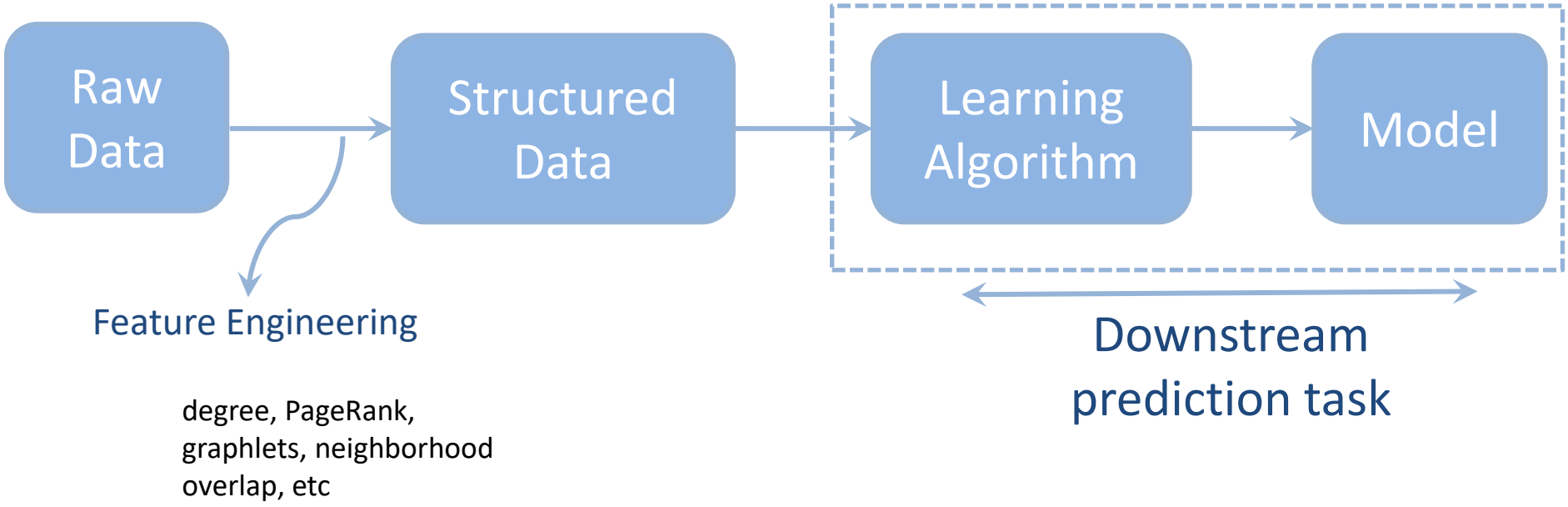
Edge (link) level

Graph-level prediction,

Graph generation

Graph embeddings: why?

Machine learning lifecycle



Part II:

Introduction to embeddings

Node embeddings on

matrix decomposition

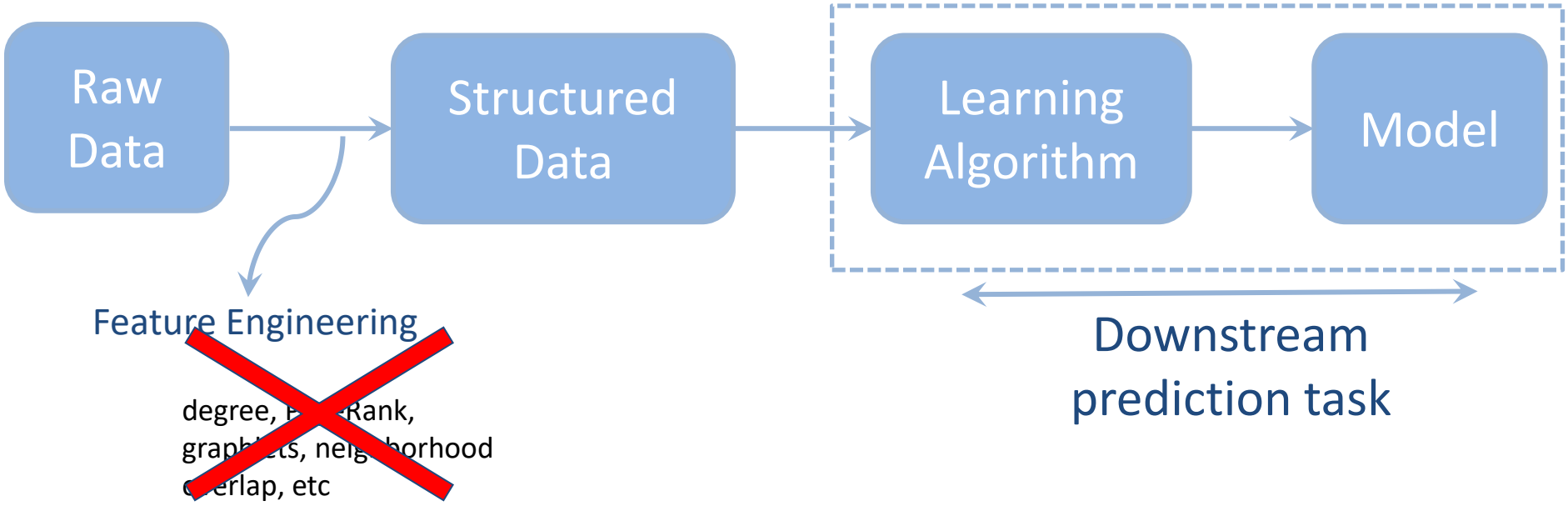
random-walks

Quick overview of word embedding

Link and subgraph embeddings

Graph embeddings: why?

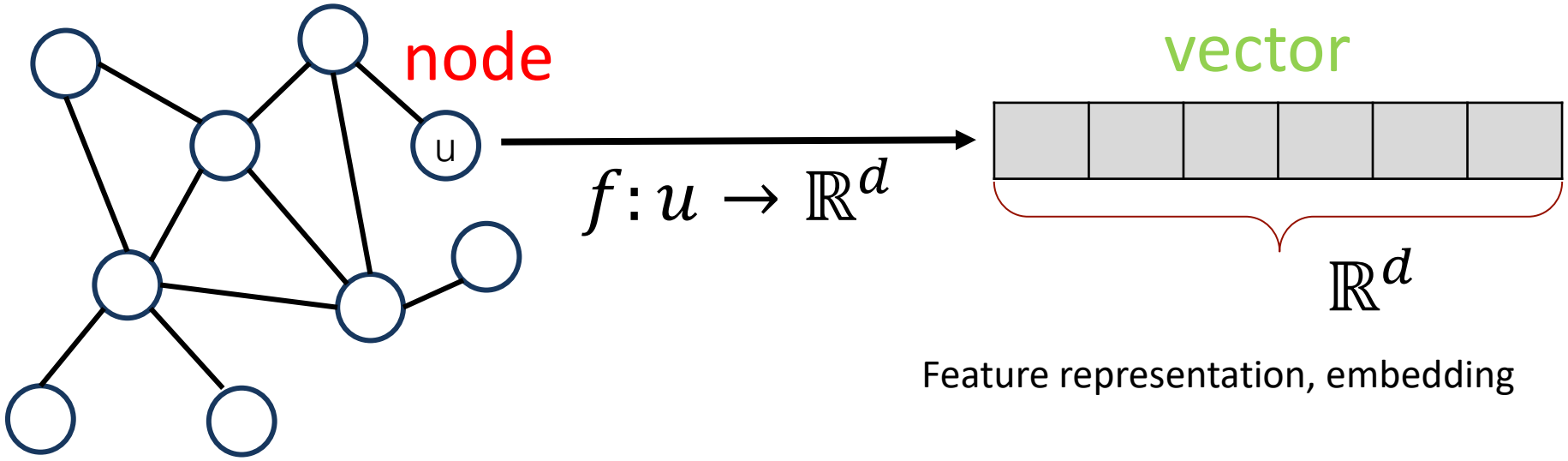
Machine learning lifecycle



Representation Learning

Automatically learn the features

Node embeddings: what are they?



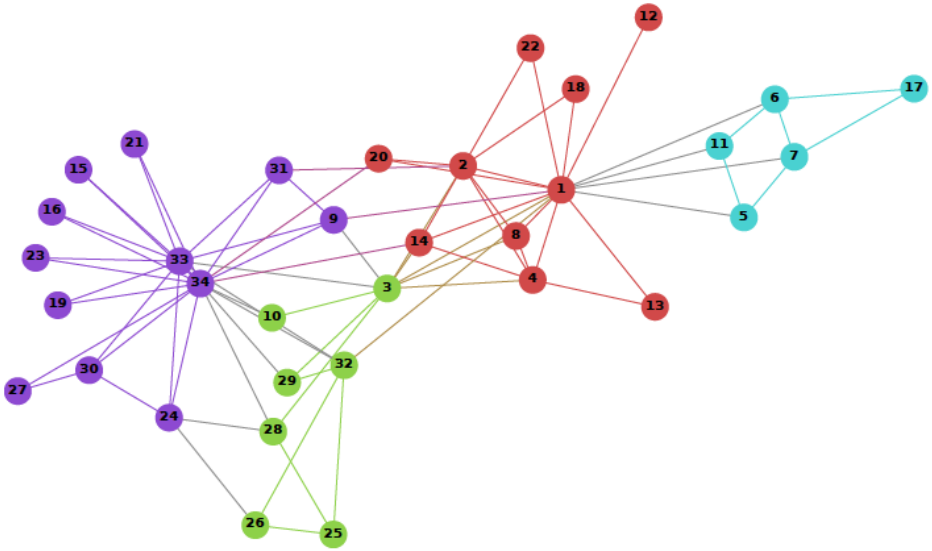
Map nodes to *d*-dimensional vectors so that:
"similar" nodes in the graph have embeddings that *are close together*.

- Encode network information
- Potentially used for many downstream predictions

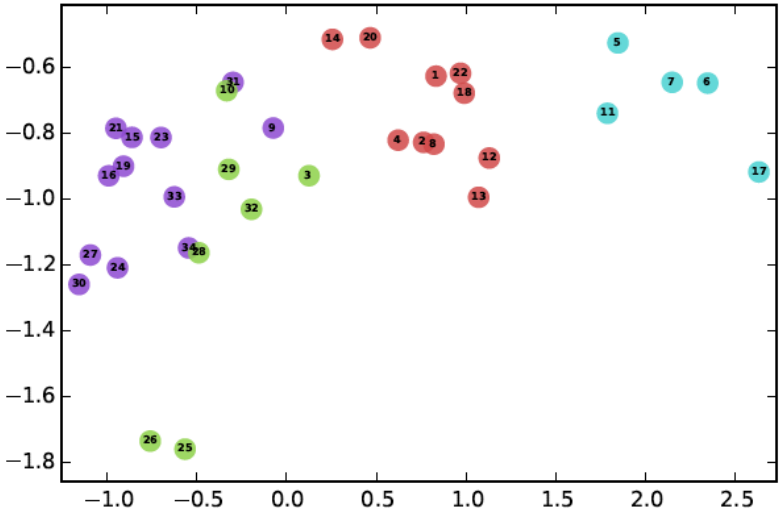
Example

Zachary's Karate Club Network:

Input



Output

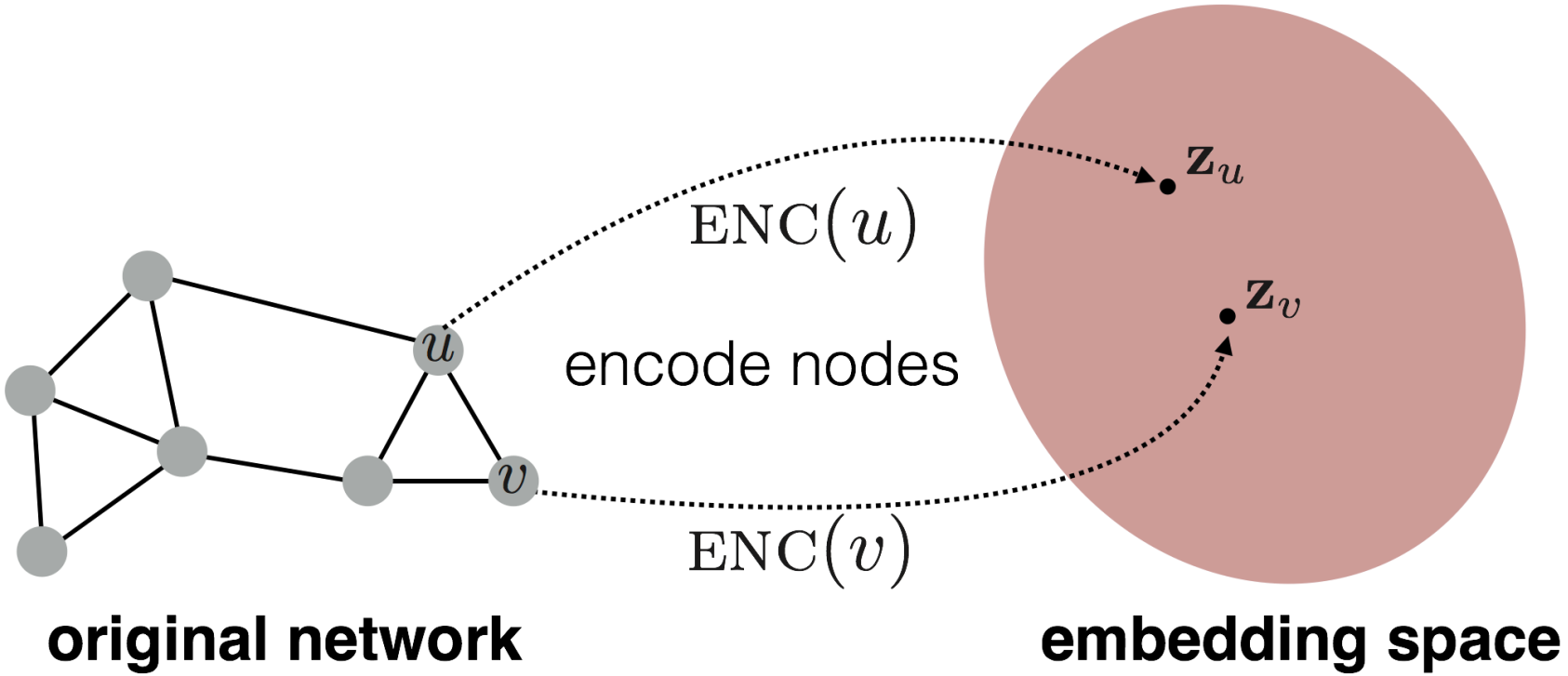


using t-SNE

Image from: Perozzi et al.. DeepWalk: Online Learning of Social Representations. KDD 2014.

Graph embeddings

Goal is to encode nodes so that similarity in the embedding space (e.g., dot product) approximates similarity in the graph



Two key components

- **Encoder:** maps each node to a low-dimensional vector

$$\text{ENC}(v) = z_v$$

d-dimensional embedding

node in the input graph

- **Similarity function:** specifies how the relationships in vector space map to the relationships in the original network

$$\text{similarity}(u, v) \approx z_u{}^T \cdot z_v$$

Decoder

Similarity of u and v in the original network

dot product between node embeddings

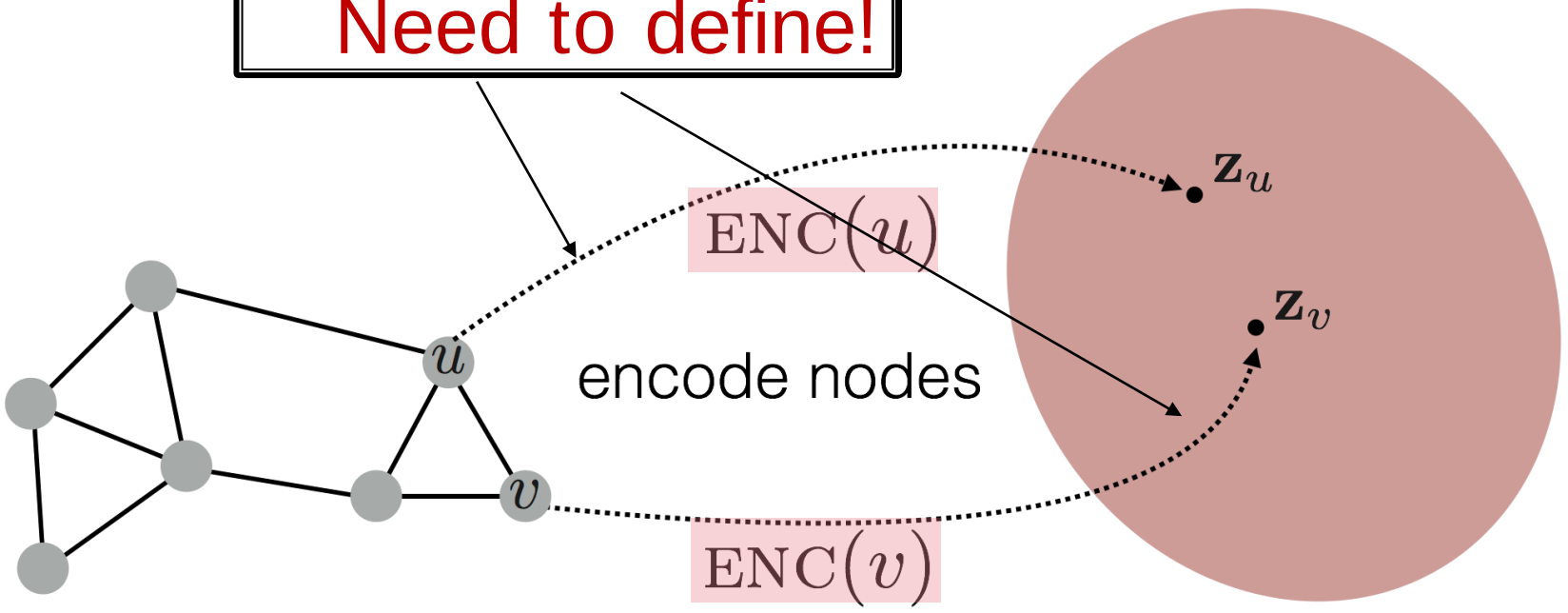
Embedding nodes

Goal: similarity$(u, v) \approx z_u{}^T \cdot z_v$

Similarity of the embedding

in the original network

Need to define!



original network

embedding space

Learning node embeddings

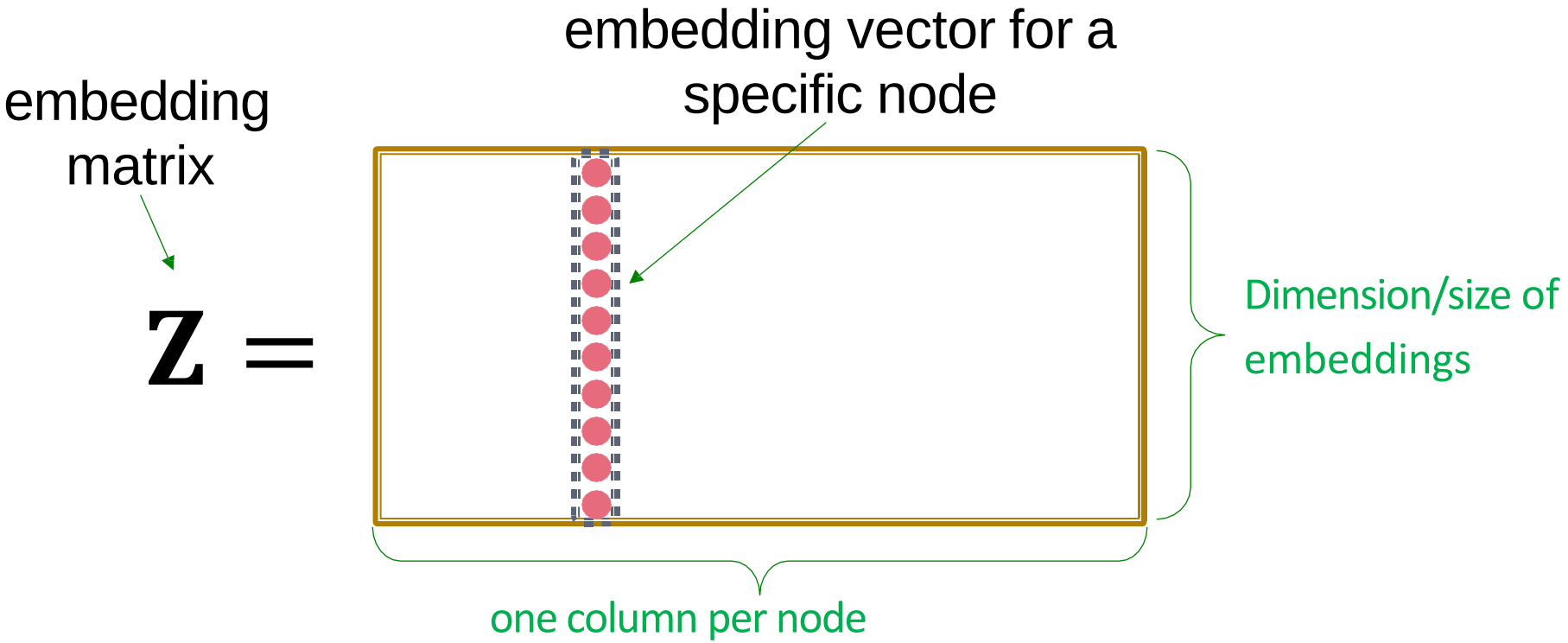
1. Define an encoder $\mathbf{ENC}$ that maps nodes to low dimensional spaces
2. Define *a node similarity function* (i.e., a measure of similarity in the original network).
3. Decoder $\mathbf{DEC}$ maps from embeddings to the similarity score
4. Optimize the parameters of the encoder so that we minimize *a loss function* $\boldsymbol{L}$ that looks (roughly) like:

$$\boldsymbol{L} = \sum_{u,v \in V} (similarity(u,v) - z_u^T \cdot z_v)^2$$

Shallow embeddings(*)

Each node is assigned a single d-dimensional vector

Learn $|V| \times d$ embedding matrix $\boldsymbol{Z}$: each column *i* is the embedding z_i of node *i*

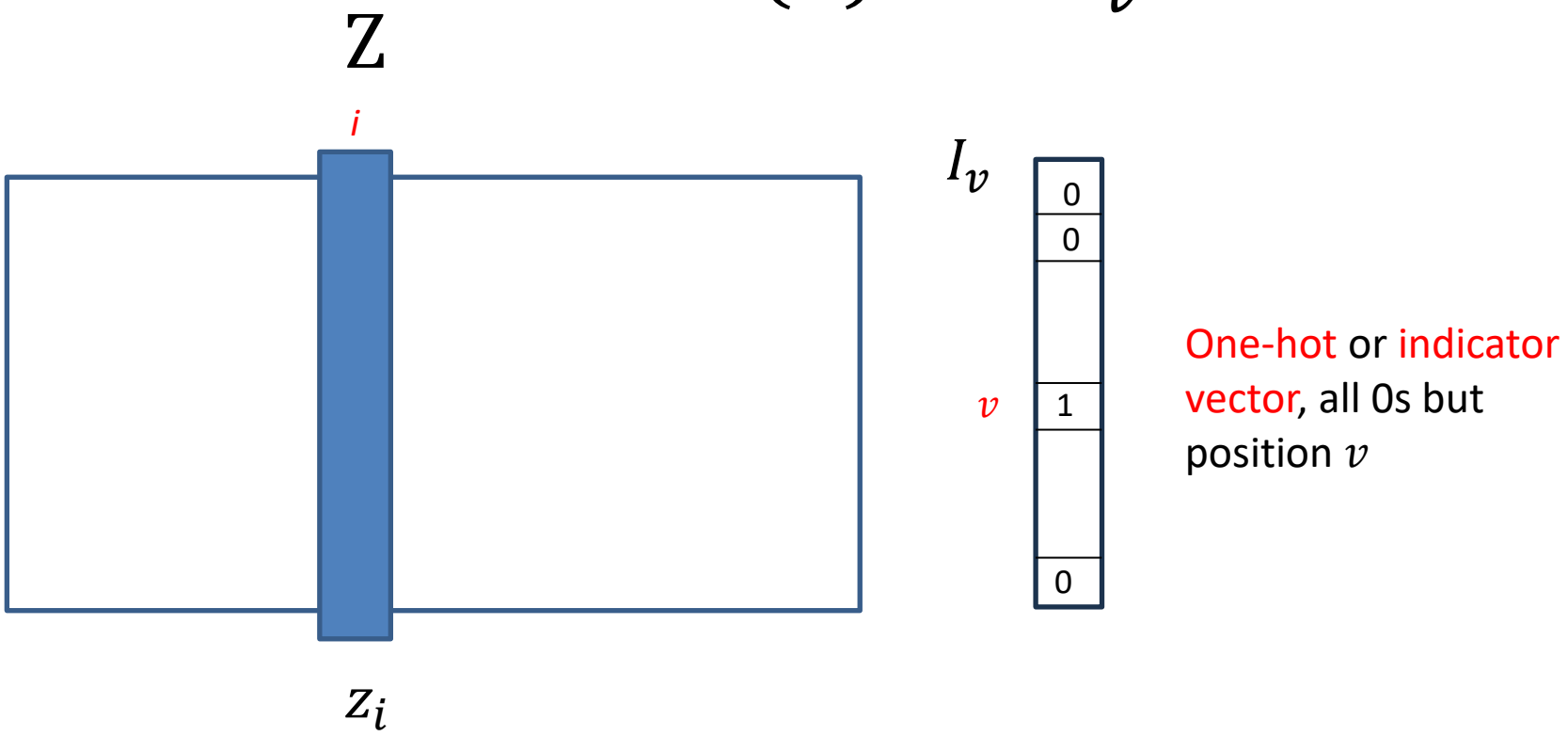


(*) As opposed to deep learning in graphs (GNN embeddings)

Shallow embeddings

Encoder is just an embedding lookup

$$ENC(v) = Z\ I_v$$



One-hot or indicator vector, all 0s but position v

Framework Summary

Encoder + Decoder Framework

- Shallow encoder: Embedding lookup
- Parameters to optimize: $\mathbf{Z}$ which contains node embeddings for all nodes $u \in V$
- We will cover deep encoders in the GNNs

- **Decoder:** based on node similarity.
- **Objective:** maximize $z_u^T \cdot z_v$ for node pairs (u, v) that are **similar**

How to define node similarity

- Key choice of methods is **how they define node similarity.**
- Should two nodes have a similar embedding if they…
  - are linked?
  - share neighbors?
  - have similar “structural roles”?

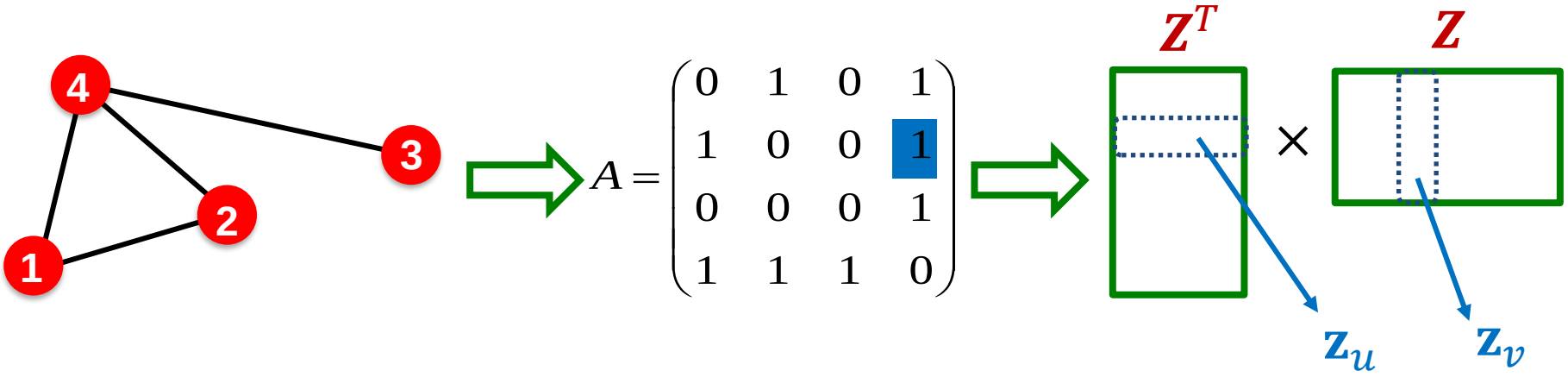
Note on node embeddings

- This is **unsupervised/self-supervised** way of learning node embeddings.
  - We are **not** utilizing node labels
  - We are **not** utilizing node features
  - The goal is to directly estimate a set of coordinates (i.e., the embedding) of a node so that some aspect of the network structure (captured by DEC) is preserved.
- These embeddings are **task independent:**
  - They are not trained for a specific task but can be used for any task.

ADJACENCY-BASED

Adjacency Matrix

- Simplest **node similarity**: Nodes u, v are similar if they are connected by an edge
- This means: $\mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u = A_{u,v}$ which is the (u, v) entry of the graph adjacency matrix A
- Therefore, $\boldsymbol{Z}^T \boldsymbol{Z} = A$



Adjacency-based approach

- The embedding dimension d (number of rows in $\boldsymbol{Z}$) is much smaller than number of nodes n.
- Inner product decoder with node similarity defined by edge connectivity is equivalent to matrix factorization of A.

- Exact factorization $A = \boldsymbol{Z}^{\boldsymbol{T}}\boldsymbol{Z}$ is generally not possible
- Matrix decomposition (for example, SVD decomposition)
  1. Scalability issues
  2. Produced matrices that are very dense

Adjacency-based approach

- However, we can learn $\boldsymbol{Z}$ approximately
- **Objective**: $\min_{\mathbf{Z}} \| \mathrm{A} - \boldsymbol{Z}^T \boldsymbol{Z} \|_2$
  - We optimize $\boldsymbol{Z}$ such that it minimizes the L2 norm (Frobenius norm) of $\mathrm{A} - \boldsymbol{Z}^T \boldsymbol{Z}$
  - We used softmax instead of L2. But the goal to approximate A with $\boldsymbol{Z}^T \boldsymbol{Z}$ is the same.

How: stochastic gradient descent

Adjacency-based approach

The loss that what we want to minimize

embedding similarity

$$L = \sum_{u,v \in V \times V} ||A_{u,v} - Z_u^T \cdot Z_v||^2$$

sum over all node pairs

(possibly weighted) adjacency matrix for the graph

A. Ahmed, N. Shervashidze, S. M. Narayanamurthy, V. Josifovski, A. J. Smola: *Distributed large-scale natural graph factorization*. WWW 2013

Adjacency-based approach – stochastic gradient descent

A few manipulations

$$L = \sum_{u,v \in V \times V} ||A_{uv} - Z_u^T \cdot Z_v||^2$$

sum over all node pairs

$$L = \sum_{(u,v) \in E} (A_{uv} - Z_u^T \cdot Z_v)^2$$

sum over all edges

$$L = \frac{1}{2} \sum_{(u,v) \in E} (A_{uv} - Z_u^T \cdot Z_v)^2 + \frac{\lambda}{2} \sum_u ||Z_u||^2$$

regularization factor

Stochastic Gradient Descent

After we obtain the objective function, how do we optimize (minimize) it?

$$L = \frac{1}{2}\sum_{(u,v)\in E}(A_{uv} - Z_u^T \cdot Z_v)^2 + \frac{\lambda}{2}\sum_{u\in V}||Z_u||^2$$

Gradient Descent: a simple way to minimize $\mathcal{L}$:

- Initialize z_u at some randomized value for all nodes u.
- Iterate until convergence:
  - For all u, compute the derivative $\frac{\partial \mathcal{L}}{\partial z_u}$.
  - For all u, make a step in reverse direction of derivative: $z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}$.

η: learning rate

Adjacency-based approach

$$L = \frac{1}{2}\sum_{(u,v)\ \in\ E}(A_{uv} - Z_u^T \cdot Z_v)^2 + \frac{\lambda}{2}\sum_{u\in V}||Z_u||^2$$

Taking the gradient

Gradient of L with respect to each row (column) of Z (learn one vector per node)

$$\frac{\partial L}{\partial Z_u} = -\sum_{v \in N(u)}(A_{uv} - Z_v \cdot Z_u^T)\, Z_v + \lambda Z_u$$

For each edge $(u, v) \in E$ this amounts for

$$\frac{\partial L}{\partial Z_u} = -(A_{uv} - Z_v \cdot Z_v)\, Z_v + \lambda Z_u$$

Adjacency-based approach

Requires: Adjacency matrix *A*, rank *d*, accuracy *ε*
Ensures: Local minimum
1: Initialize Z' at random
2: t ← 1
3; repeat
4: Z ← Z'
5: for all edges (i, j) ∈ E do
6: $\eta \leftarrow 1/\sqrt{t}$
7: t ← t +1
8: Zi ← Zi + η ((Aij – <Zi · Zj>Zj) + λ Zi)
9: end for
10: until $||Z- Z'||^2 <= \varepsilon$
11: return Z

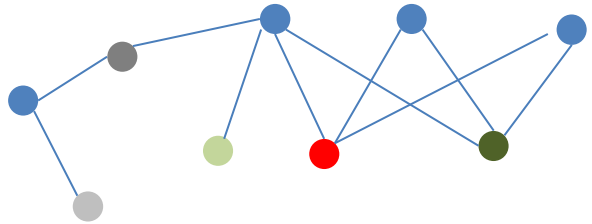
η: learning rate, captures the extent at which newly acquired information overrides old

- Complexity O(|E|)
- Can be parallelized

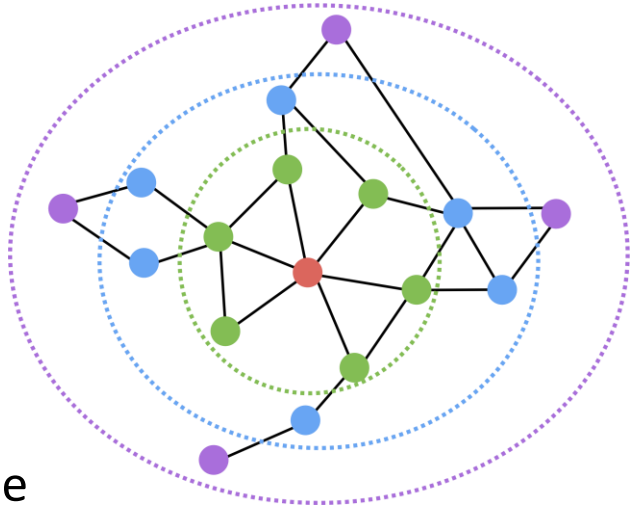
Multi-hop approaches

Only considers direct connections

What about further neighbors?



Look further than the 1-step neighbors and learn by using information from/for *k*-step neighbors



We will see two approaches

- GraRep: looks at probabilities of reaching a node
- HOPE: various metrics of similarity based on neighbors and paths

High-order Proximity Preserved Embeddings (HOPE)

Based on a high order proximity matrix S,

$$S_{uv} = proximity(u, v)$$

For directed graphs, learn two embedding vectors

$$Z = |Z^s, Z^t|$$

$$L = \sum_{(u,v) \in V \times V} ||S_{uv} - Z_u^s \cdot Z_v^t||^2$$

M. Ou, P. Cui, J. Pei, Z.i Zhang, W. Zhu: *Asymmetric Transitivity Preserving Graph Embedding*. KDD 2016

HOPE

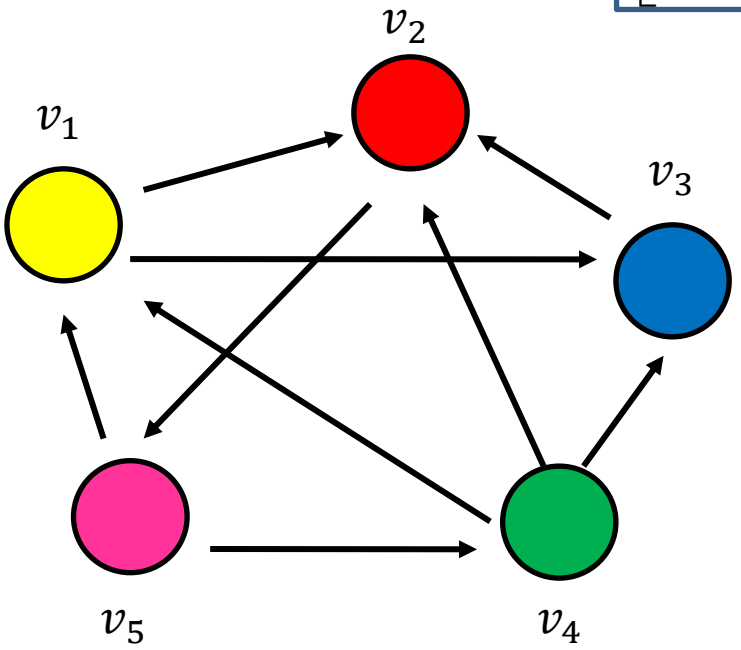
Local High Order Proximity

Common Neighbors (for directed graphs, source-target)

$$S^{CN} = A^2$$

$A^2 =$

$$\begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 2 & 1 & 0 & 1 \\ 1 & 2 & 2 & 0 & 0 \end{bmatrix}$$



Adamic-Adar

$$S^{AA} = A\ D\ A$$

Similar but assigns a weight to the neighbor reciprocal of its degree

HOPE

Global High Order Proximity

Katz

Sum over all paths of length l, using a decay parameter

$$S^{Katz} = \sum_{l=1}^{\infty} \beta^l A^l$$

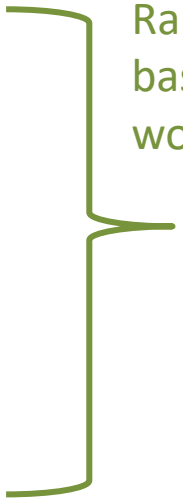
Rooted Pagerank

SVD with some tricks to save computations

Node embeddings

Approaches based on:

- Adjacency-like matrices
  - Adjacency matrix
  - Multi-hop neighborhoods
    - HOPE
    - GraRep (random walks)
- Random-walks
  - DeepWalk
  - Node2Vec



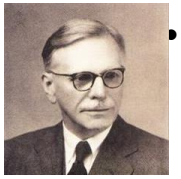
Random walks based on word embeddings

WORD EMBEDDINGS

(Some material from Chris Manning course)

Basic Idea

- You can get a lot of value by representing a word by means of its neighbors (distributional semantics)
- "You shall know a word by the company it keeps"
  - (J. R. Firth 1957: 11)

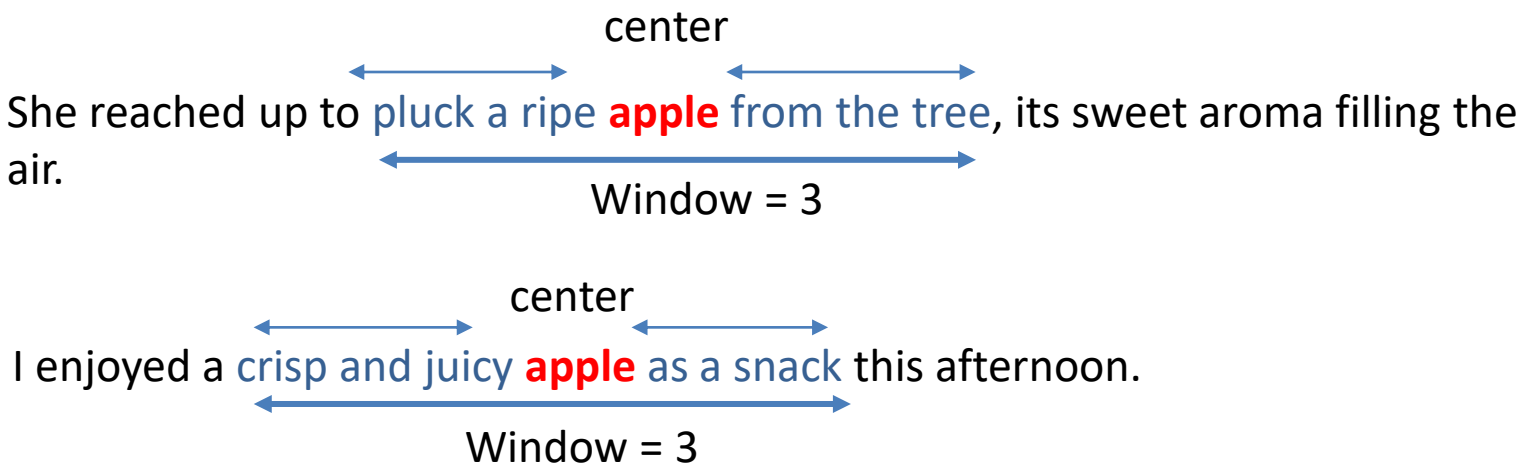


- One of the most successful ideas of modern statistical NLP

Basic Idea

A word is defined by its context

Context: words that appear in a fixed length window around the word



Use the many contexts of w to represent a word

Basic idea

Learn two embeddings per word: (1) as context (2) as center

word

$$\begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ . \\ . \\ . \\ w_d \end{bmatrix}$$

Center

$$\begin{bmatrix} w'_1 \\ w'_2 \\ w'_3 \\ . \\ . \\ . \\ w'_d \end{bmatrix}$$

Context

- Center-embedding of a center word similar with the context embeddings of its context words
- And vice-versa

Use text to learn these embeddings

Word2Vec

Predict between every word and its context words

Two **algorithms**

1. Skip-grams (SG)

   Predict context words given the center word

2. Continuous Bag of Words (CBOW)

   Predict center word from a bag-of-words context

Position independent (do not account for distance from center)

Two **training methods**

1. Hierarchical softmax
2. Negative sampling

Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, Jeffrey Dean: *Distributed Representations of Words and Phrases and their Compositionality.* NIPS 2013: 3111-3119

Hierarchical softmax

Instead of learning O(|V|) vectors, learn O(log(|V|) vectors

How?

- Build a binary tree with leaves the words *and learn one vector for each internal node*.
- The value for each word w is the product of the values of the internal nodes in the path from the root to w.

Basic Idea

She reached up to pluck a ripe **apple** from the tree, its sweet aroma filling the air.

Window = 3

CBOW

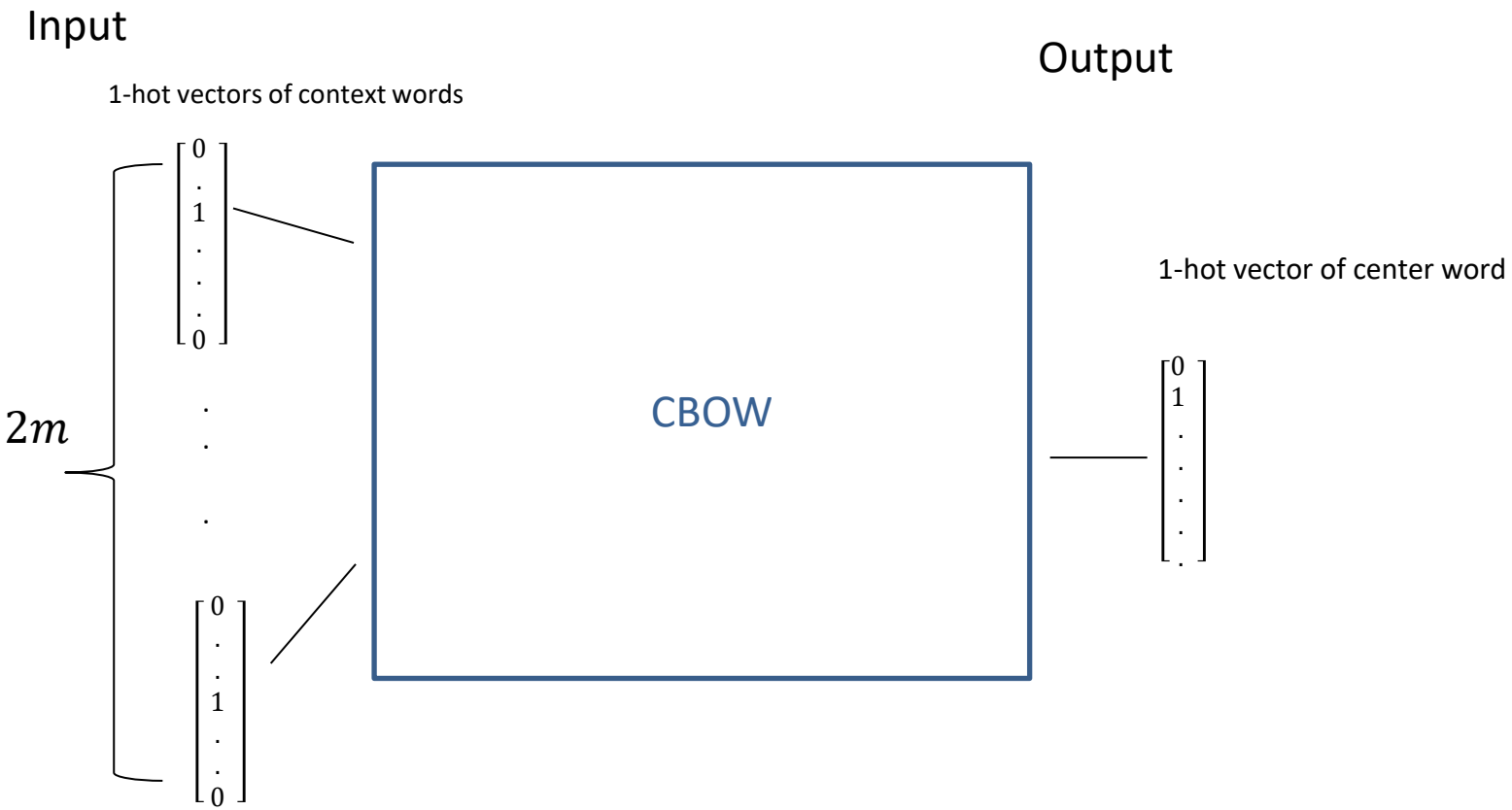
pluck a ripe _____ from the tree

Skipgram

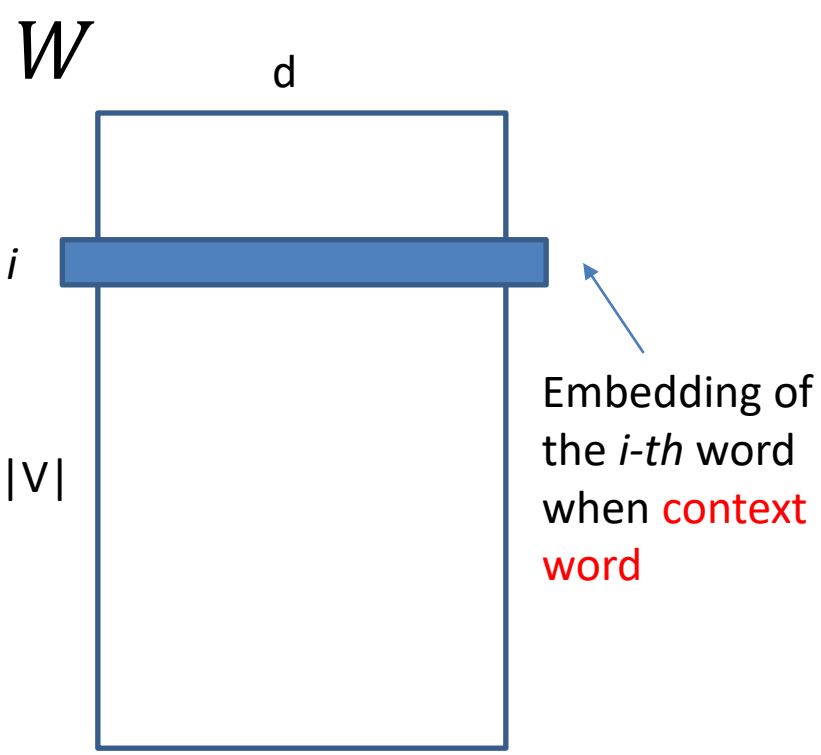
____ ____ ____ apple ____ ____ ____

CBOW

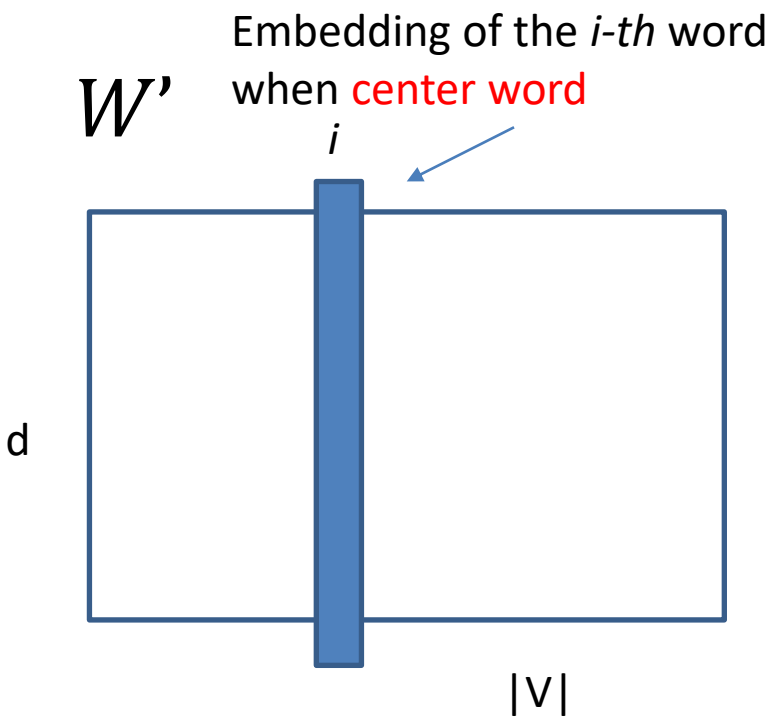
Use a window of context words to predict the center word



CBOW



$|V| \times d$ context embeddings when *input*



$d \times |V|$ center embeddings when *output*

CBOW

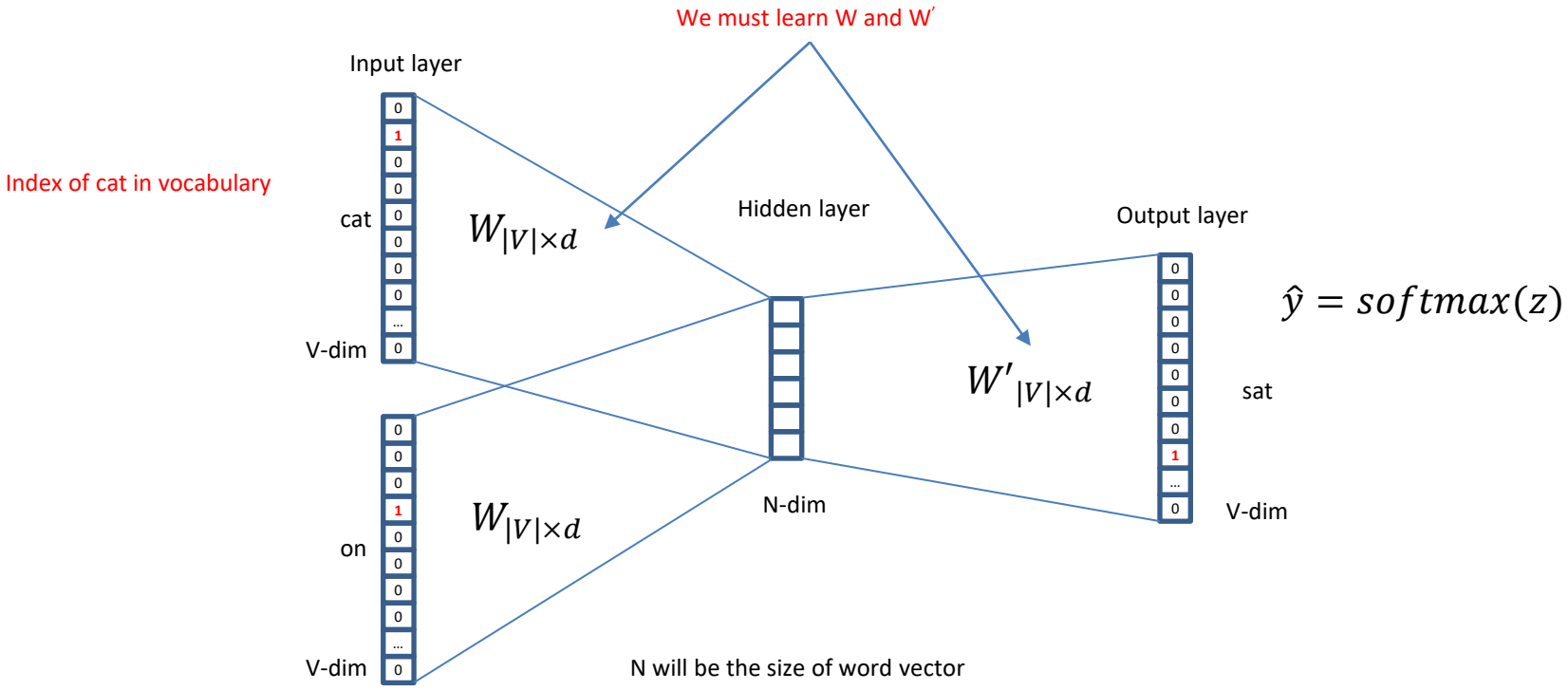
Intuition

The W'-embedding of the *center word* should be *similar* to the (average of) the W-embeddings of its *context words*

- For similarity, we will use cosine (dot product)
- We will take the average of the W-embeddings of the context word

We want similarity close to one for the center word and close to 0 for all other words

cat sat on
window size = 1



CBOW

Given window size *m*

$x^{(c)}$ one hot vector for context words, *y* one hot vector for the center word

1. **INPUT:** the *one hot vectors* for the *2m* context words

$x^{(c-m)}, \ldots, x^{(c-1)}, x^{(c+1)}, \ldots, x^{(c+m)}$

2. *GET THE EMBEDDINGS of the context words*

$v_{c-m} = Wx^{(c-m)}, \ldots, v_{c-1} = Wx^{(c-1)}, v_{c+1} = Wx^{(c+1)}, \ldots, v_{c+m} = Wx^{(c+m)}$

3. *TAKE THE SUM* these vectors (average)

$\hat{v} = \frac{v_{c-m} + v_{c-m+1} + \cdots v_{c+m}}{2m}$, $\hat{v} \in R^N$

4. *COMPUTE SIMILARITY: dot produce W' (all center vectors) and context $\hat{v}$ (generate score vector z)*

z = W' $\hat{v}$

5. Turn the *score vector to probabilities*

$\hat{y}$ = *softmax(z)*

We want this to be close to 1 for the center word

Softmax

From values to probability distributions

Exponentiate to make positive

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

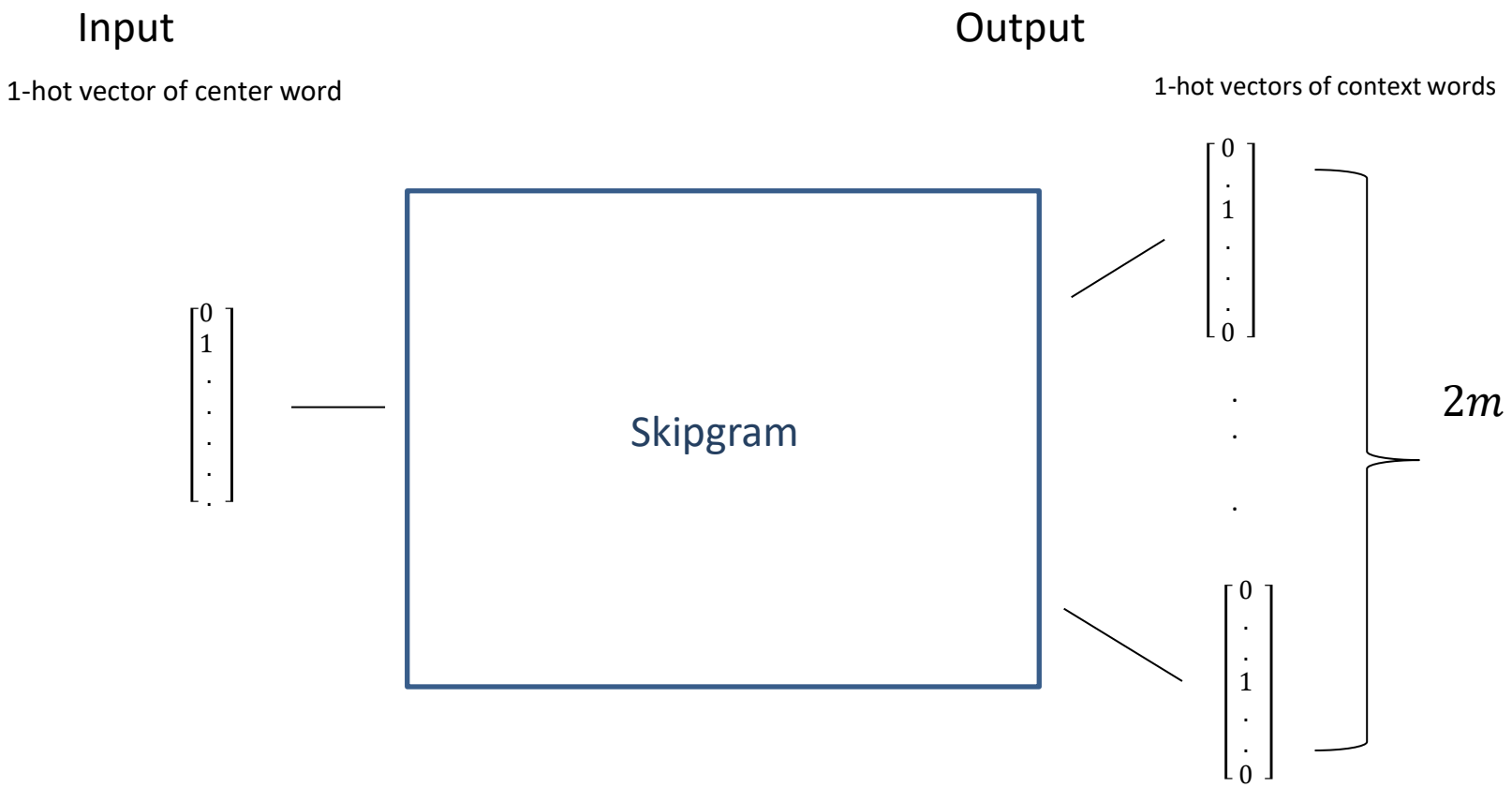
Normalize to get probabilities

- "Most" probability to the largest value (max)
- "Some" probability to the other values (soft)

```
>>> import numpy as np
>>> a = [1.0, 2.0, 3.0, 4.0, 1.0, 2.0, 3.0]
>>> np.exp(a) / np.sum(np.exp(a))
array([0.02364054, 0.06426166, 0.1746813, 0.474833,
0.02364054, 0.06426166, 0.1746813])
```

Skipgram

Given the center word, predict (or, generate) the context words



Learn two matrices

W: $d \times |V|$, input matrix, word representation as center word

W': $|V| \times d$, output matrix, word representation as context word

Skipgram

Given the center word, predict (or, generate) the context words

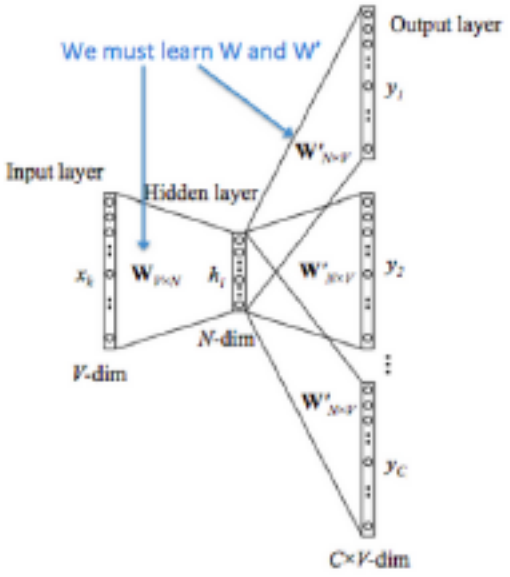
$y^{(j)}$ one hot vector for context words

1. Input: *one hot vector* of the center word
x

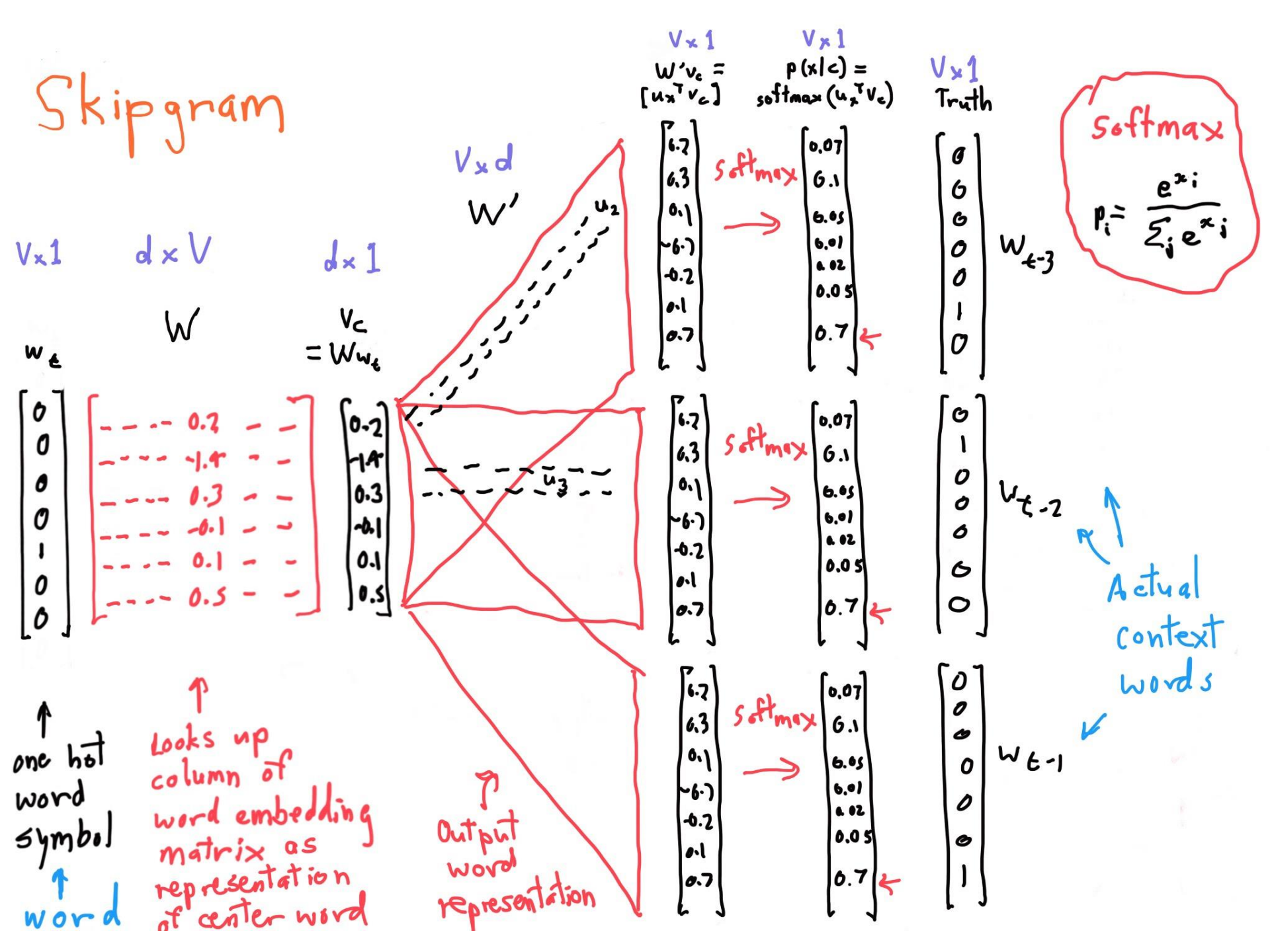
2. Get the *embedding of the center word*
$v_c = W\ x$

3. Generate a *score vector for each context word*
z = *W' v_c*

5. Turn the *score vector into probabilities*
$\hat{y}$ = *softmax(z)*



We want this to be close to 1 for the context words


Skipgram
V×1
d×V
d×1
V×d
W'
W
$v_c = Ww_t$
w_t
u_2
u_3
V×1
$W'v_c = [u_x^T v_c]$
$p(x|c) = \text{softmax}(u_x^T v_c)$
V×1
Truth
Softmax
$p_i = \frac{e^{x_i}}{\sum_i e^{x_i}}$
W_{t-3}
W_{t-2}
W_{t-1}
Actual context words
one hot word symbol
word
Looks up column of word embedding matrix as representation of center word
Output word representation

BACK TO GRAPHS

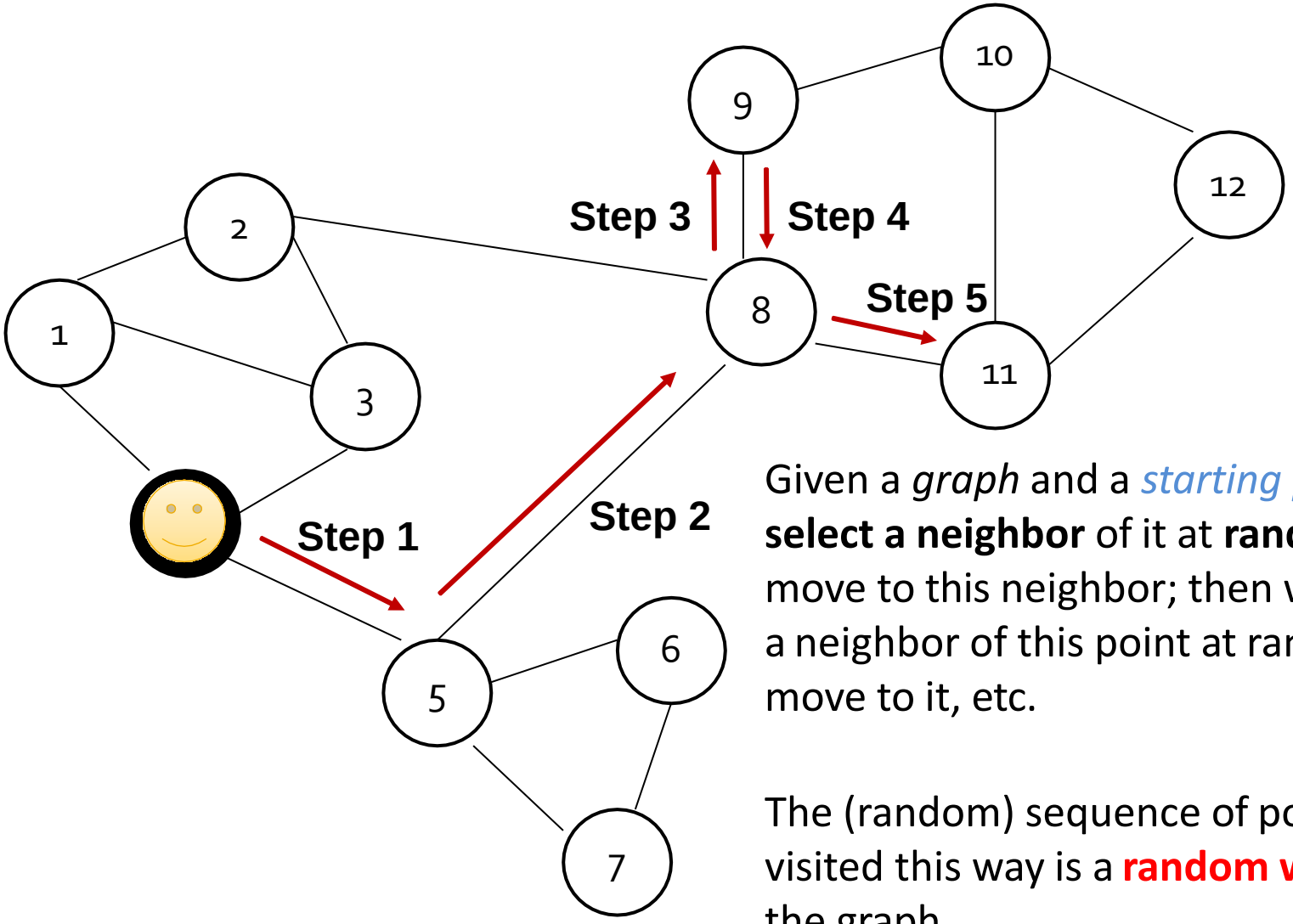
RANDOM -WALK BASED EMBEDDINGS

How?

Words = Nodes

Sentences = Paths, Random walks

Random Walk



Given a *graph* and a *starting point*, we **select a neighbor** of it at **random**, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc.

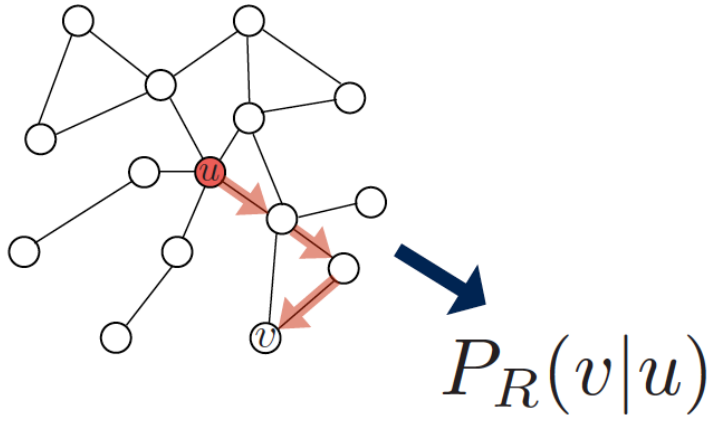
The (random) sequence of points visited this way is a **random walk** on the graph.

Random-walk embeddings

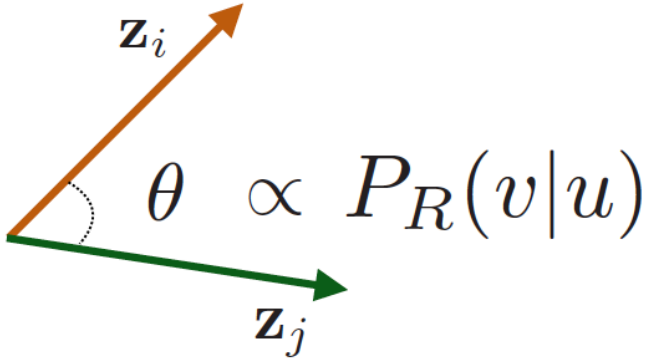
$$z_i \cdot z_j \approx$$ probability that *i* and *j* co-occur on a random walk over the network

Random-walk Embeddings

1. Estimate 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.



Why Random Walks?

1. **Expressivity:** Flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information.
   Idea: if random walk starting from node u visits v with high probability, u and v are similar (high-order multi-hop information)
2. **Efficiency:** Do not need to consider all node pairs when training; only need to consider pairs that co-occur on random walks.

Unsupervised Feature Learning

- Intuition: Find embedding of nodes in d-dimensional space that preserves similarity
- Idea: Learn node embedding such that nearby nodes are close together in the network
- Given a node u, how do we define nearby nodes?
  - $N_R(u)$: neighbourhood of u obtained by some random walk strategy R

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 $N_R(u)$, the multiset* of nodes visited on random walks starting from u.
3. Optimize embeddings according to: Given node u, predict its neighbors $N_{\mathrm{R}}(u)$.

$$\arg\max_{z} \sum_{u \in V} \log \mathrm{P}(N_{\mathrm{R}}(u) | \mathbf{z}_u) \Rightarrow$$

Maximum likelihood objective

*$N_R(u)$ can have repeat elements since nodes can be visited multiple times on random walks

Random Walk Optimization

Equivalently,

$$\arg\min_z \mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

Intuition: Optimize embeddings $\boldsymbol{z}_u$ to minimize the negative log-likelihood of random walk neighborhoods $N(u)$.

Parameterize $P(v|\mathbf{z}_u)$ using softmax:

$$P(v|\mathbf{z}_u) = \frac{\exp(\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_n)}$$

Why softmax?
We want node v to be most similar to node u (out of all nodes n).
Intuition: $\sum_i \exp(x_i) \approx \max_i \exp(x_i)$

Random Walk Optimization

Putting it all together:

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

sum over all nodes u

sum over nodes v seen on random walks starting from u

predicted probability of u and v co-occuring on random walk

Optimizing random walk embeddings = Finding embeddings z_u that minimize $\mathcal{L}$

Random Walk Optimization

But doing this naively is too expensive!

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

Nested sum over nodes gives $O(|V|^2)$ complexity!

Random Walk Optimization

But doing this naively is too expensive!

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

The normalization term from the softmax is the culprit… can we approximate it?

Negative Sampling

- **Solution:** Negative sampling

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

Why is the approximation valid?
Technically, this is a different objective. But Negative Sampling is a form of Noise Contrastive Estimation (NCE) which approx. maximizes the log probability of softmax.

New formulation corresponds to using a logistic regression (sigmoid func.) to distinguish the target node v from nodes n_i sampled from background distribution P_v.

More at https://arxiv.org/pdf/1402.3722.pdf

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

sigmoid function
(makes each term a "probability" between 0 and 1)

random distribution over nodes

Instead of normalizing w.r.t. all nodes, just normalize against k random "**negative samples**" n_i

- Negative sampling allows for quick likelihood calculation.

Negative Sampling

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

random distribution over nodes

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

- Sample k negative nodes n_i each with prob. proportional to its degree.
- Two considerations for k (# negative samples):
  1. Higher k gives more robust estimates
  2. Higher k corresponds to higher bias on negative events

  In practice k =5-20.

Can negative sample be any node or only the nodes not on the walk? People often sample any node (for efficiency).

Stochastic Gradient Descent

- After we obtained the objective function, how do we optimize (minimize) it?

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

- **Gradient Descent**: a simple way to minimize $\mathcal{L}$:
  - Initialize z_u at some randomized value for all nodes u.
  - Iterate until convergence:
    - For all u, compute the derivative $\frac{\partial \mathcal{L}}{\partial z_u}$.
    - For all u, make a step in reverse direction of derivative: $z_u \leftarrow z_u - \eta \frac{\partial \mathcal{L}}{\partial z_u}$.

η: learning rate

Stochastic Gradient Descent

- **Stochastic Gradient Descent**: Instead of evaluating gradients over all examples, evaluate it for each **individual** training example.
  - Initialize z_u at some randomized value for all nodes u.
  - Iterate until convergence: $\mathcal{L}^{(u)} = \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$
    - Sample a node u, for all v calculate the gradient $\frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$.
    - For all v, update: $z_v \leftarrow z_v - \eta \frac{\partial \mathcal{L}^{(u)}}{\partial z_v}$.

Random Walks: Summary

1. Run **short fixed-length** random walks starting from each node on the graph

2. For each node u collect $N_R(u)$, the multiset of nodes visited on random walks starting from u.

3. Optimize embeddings Z using Stochastic Gradient Descent:

$$\mathcal{L} = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(P(v|\mathbf{z}_u))$$

We can efficiently approximate this using negative sampling!

How should we randomly walk?

- **DeepWalk** just runs fixed-length, unbiased random walks starting from each node

- **Node2vec:** biased random walks that can trade-off between **local** and **global** views of the network

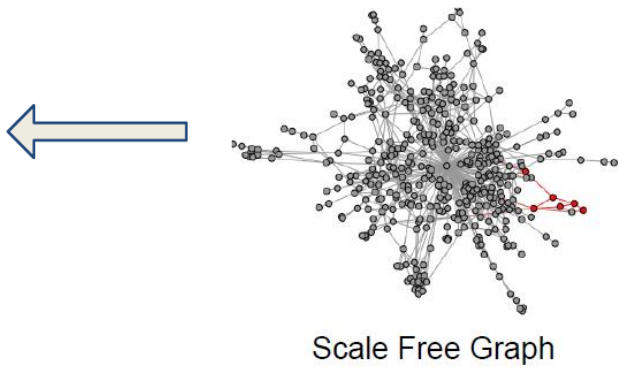
B. Perozzi, R. Al-Rfou, S. Skiena: DeepWalk: online learning of social representations. KDD 2014

A. Grover, J. Leskovec: *node2vec: Scalable Feature Learning for Networks*. KDD 2016

DeepWalk

Short random walks = sentences

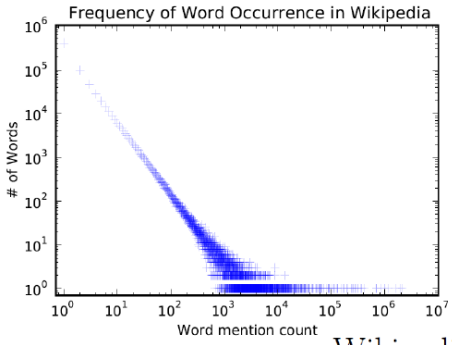
$$
\begin{array}{cccccc}
v_{71} \rightarrow & v_{24} \rightarrow & v_5 \rightarrow & v_1 \rightarrow & v_{17} \rightarrow & v_{80} \rightarrow \\
v_{92} \rightarrow & v_2 \rightarrow & v_3 \rightarrow & v_1 \rightarrow & v_{12} \rightarrow & v_{73} \rightarrow \\
v_{37} \rightarrow & v_{34} \rightarrow & v_9 \rightarrow & v_1 \rightarrow & v_{10} \rightarrow & v_{94} \rightarrow \\
v_{73} \rightarrow & v_{64} \rightarrow & v_5 \rightarrow & v_1 \rightarrow & v_{12} \rightarrow & v_1 \rightarrow \\
v_{75} \rightarrow & v_{14} \rightarrow & v_6 \rightarrow & v_1 \rightarrow & v_{13} \rightarrow & v_{61} \rightarrow
\end{array}
$$



Short truncated random walks are sentences in an artificial language

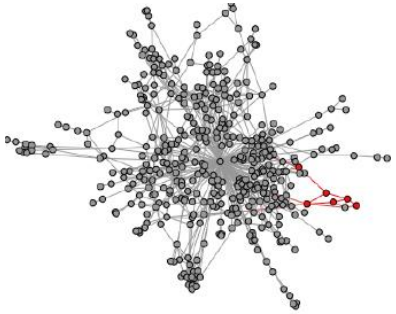
DeepWalk

Words frequency in a natural language corpus follows a power law.



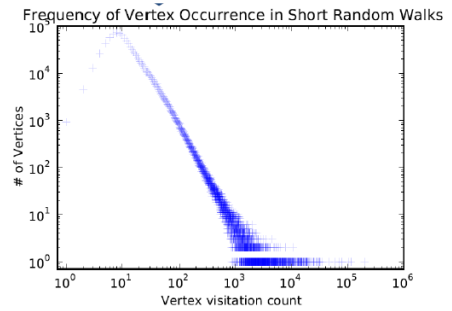
Wikipedia Article Text

$$
\begin{array}{llllll}
v_{71} \rightarrow & v_{24} \rightarrow & v_5 \rightarrow & v_1 \rightarrow & v_{17} \rightarrow & v_{80} \rightarrow \\
v_{92} \rightarrow & v_2 \rightarrow & v_3 \rightarrow & v_1 \rightarrow & v_{12} \rightarrow & v_{73} \rightarrow \\
v_{37} \rightarrow & v_{34} \rightarrow & v_9 \rightarrow & v_1 \rightarrow & v_{10} \rightarrow & v_{94} \rightarrow \\
v_{73} \rightarrow & v_{64} \rightarrow & v_5 \rightarrow & v_1 \rightarrow & v_{12} \rightarrow & v_1 \rightarrow \\
v_{75} \rightarrow & v_{14} \rightarrow & v_6 \rightarrow & v_1 \rightarrow & v_{13} \rightarrow & v_{61} \rightarrow
\end{array}
$$



Scale Free Graph

Node frequency in random walks on scale free graphs also follows a power law.



YouTube Social Graph

Representation mapping

$\mathcal{W}_{v_4} = v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_1 \rightarrow v_{46} \rightarrow v_{51} \rightarrow v_{89}$

$\mathcal{W}_{v_4} = 4$

u_k [3, 1, 5] v_j → Φ (d × j)

1

⋮

- Map the vertex under focus (v_1) to its representation.
- Define a window of size w
- If w = 1 and v = v_1

Maximize: $\Pr(v_3|\Phi(v_1))$

$\Pr(v_5|\Phi(v_1))$

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**

DeepWalk

The algorithm consists of two main components; first a random walk generator and second an update procedure

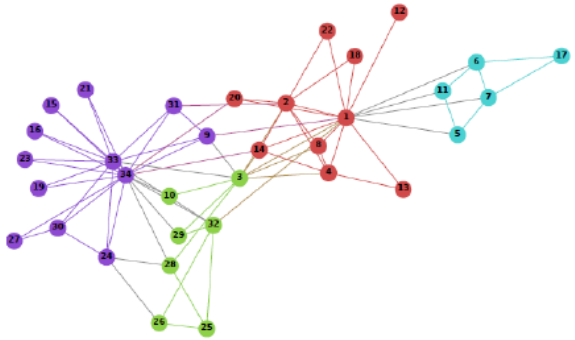
- Window *w*
- Generate *γ* random walks for *each vertex* in the graph
- Each short random walk has length *t (intuitively, sentence length)*
- Pick the next step *uniformly* from the node neighbors

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**
    $\mathcal{W}_{v_i} = RandomWalk(G, v_i, t)$
    SkipGram(Φ, $\mathcal{W}_{v_i}$, w)
  end for
end for

DeepWalk



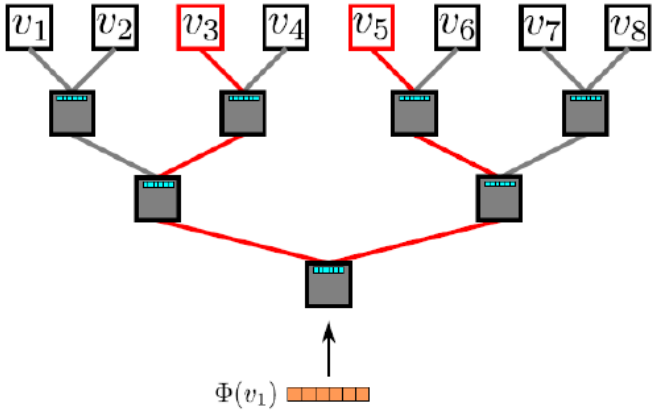
2 Random Walks

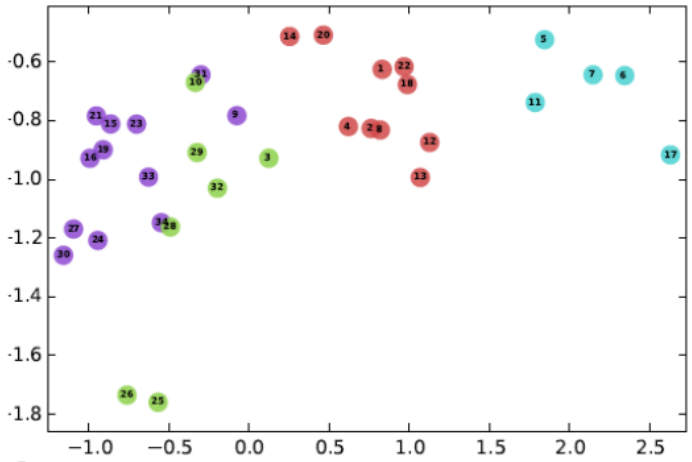
$$\mathcal{W}_{v_4} = 4$$

$$u_k \begin{bmatrix} 3 \\ 1 \\ 5 \\ 1 \\ \vdots \end{bmatrix} v_j \rightarrow \Phi \quad (d \times j)$$

1 Input: Graph

3 Representation Mapping



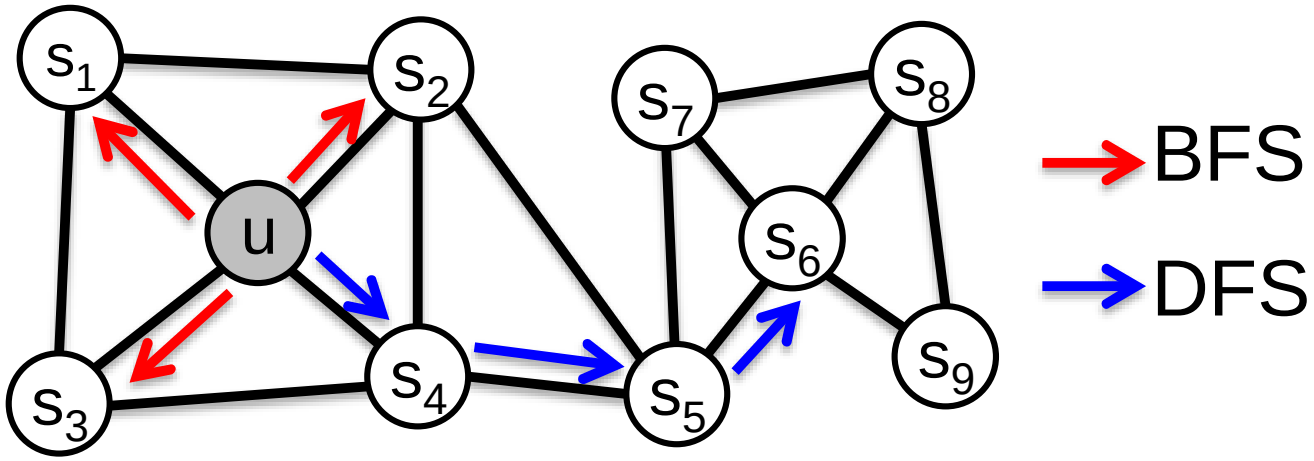


4 Hierarchical Softmax

5 Output: Representation

node2vec: Biased Walks

Two classic strategies to define a neighborhood $N_R(u)$ of a given node u:

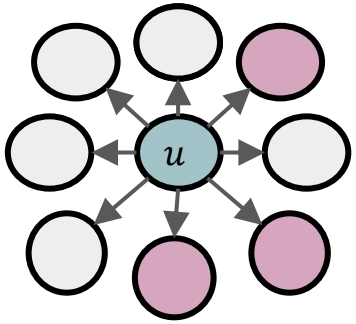


Walk of length 3 ($N_R(u)$ of size 3):

$N_{BFS}(u) = \{ s_1, s_2, s_3 \}$ Local microscopic view (BFS)

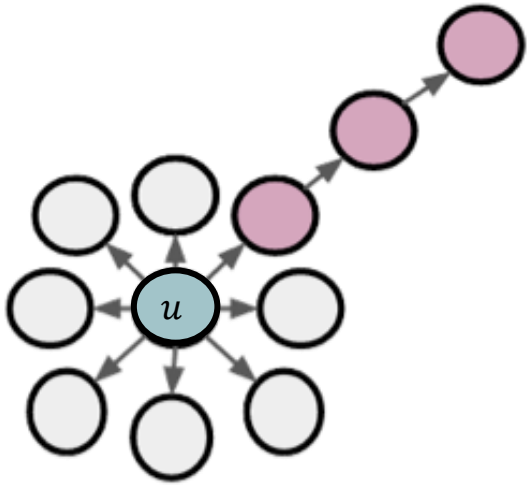
$N_{DFS}(u) = \{ s_4, s_5, s_6 \}$ Global macroscopic view (DFS)

BFS vs DFS



BFS:

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

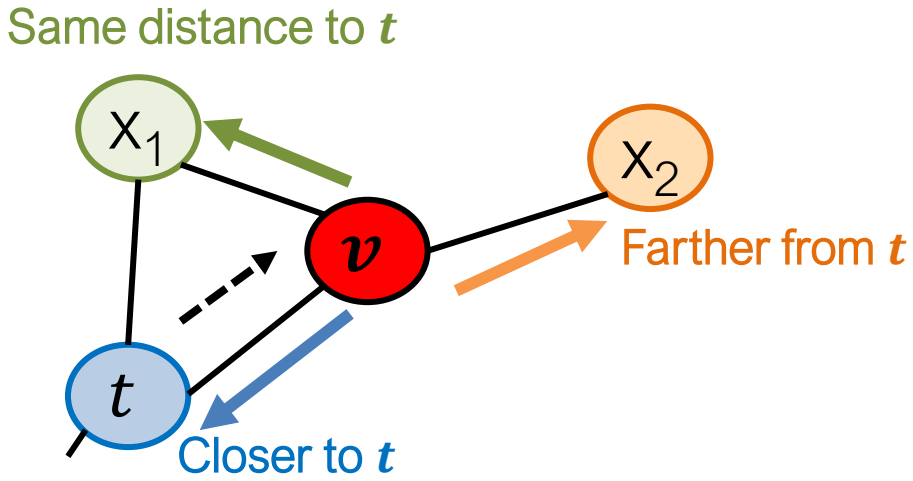


DFS:

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

Biased 2nd Order Random Walks

Walker from t, traversed (t, v) and is now in v, where to go next?



How much far away from t? Only three possible choices:

- Farther distance (distance =2)
- Same distance (distance = 1)
- Back to t (distance = 0)

Interpolating BFS and DFS

Biased random walk R that given a node u generates neighborhood $N_R(u)$

- Two parameters:
  - Return parameter p:
    - Return to the previous node
  - In-out parameter q:
    - Moving outwards (DFS) vs. inwards (BFS)
    - Intuitively, q is the "ratio" of BFS vs. DFS
- Specify how a single step of biased random walk is performed
  - Random walk is then just a sequence of these steps.

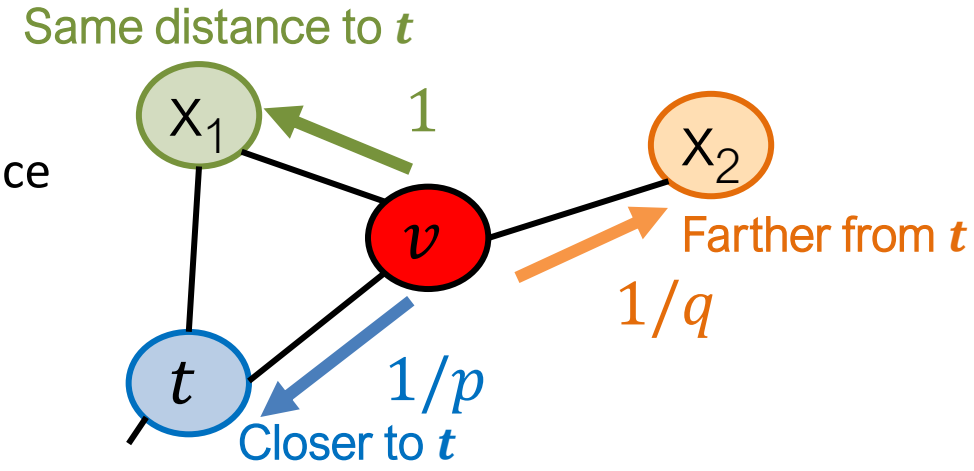
One step of the biased random walk

At v from t, where to go next?

Define the random walk by specifying the walk transition probabilities on edges adjacent to the current node v:

- 1 to node with same distance
- 1/q node further apart
- 1/p back to t

(unnormalized probabilities)

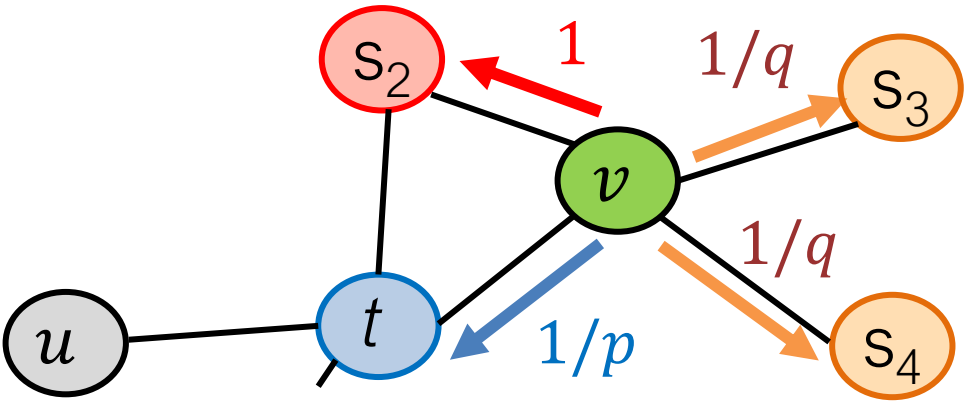


BFS-like walk: Low value of p

DFS-like walk: Low value of q

One step of the biased random walk

At v from S_1



$v \rightarrow$

Target	Prob.	Dist. (S_i, t)
t	$1/p$	0
S_2	1	1
S_3	$1/q$	2
S_4	$1/q$	2

Unnormalized transition prob. segmented based on distance from t

$N_R(v)$ are the nodes visited by the biased walk

node2vec algorithm

1) Compute edge transition probabilities:
   - For each edge (s_1, w) we compute edge walk probabilities (based on p, q) of edges $(w, \cdot)$

2) Simulate r random walks of length l starting from each node u

3) Optimize the node2vec objective using Stochastic Gradient Descent

Linear-time complexity

All 3 steps are individually parallelizable

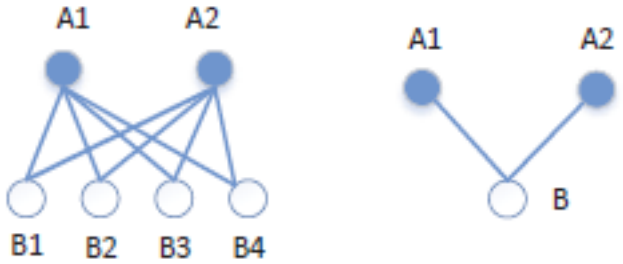
Other Random Walk Ideas

- **Different kinds of biased random walks:**
  - Based on node attributes (Dong et al., 2017).
  - Based on learned weights (Abu-El-Haija et al., 2017)
- **Alternative optimization schemes:**
  - Directly optimize based on 1-hop and 2-hop random walk probabilities (as in LINE from Tang et al. 2015).
- **Network preprocessing techniques:**
  - Run random walks on modified versions of the original network (e.g., Ribeiro et al. 2017 struct2vec, Chen et al. 2016 HARP).

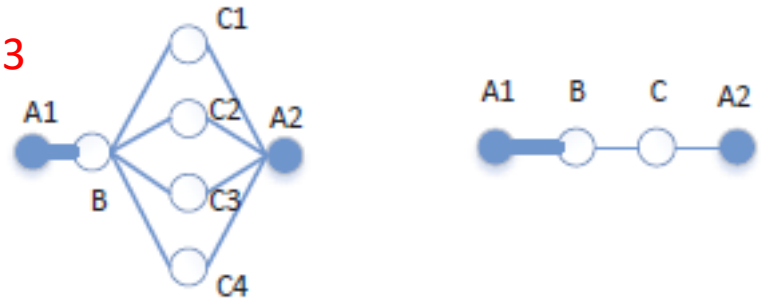
GraRep

Path of length k =1

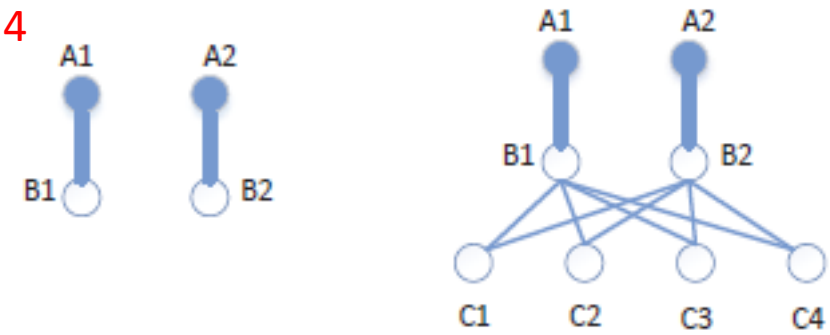
Path of length k = 2



Path of length k = 3



Path of length k = 4

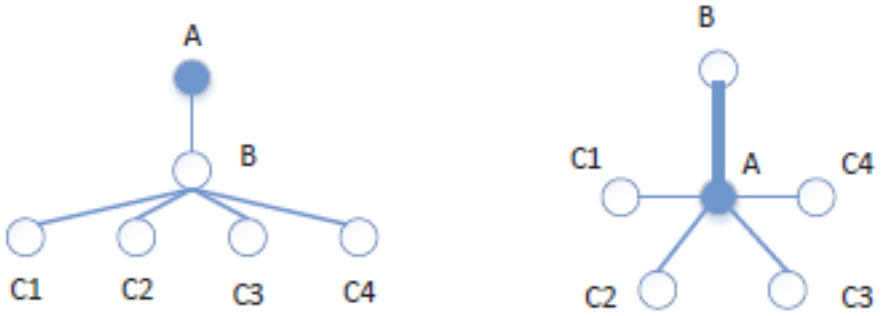


- Look at the paths that connect the nodes
- More paths -- more similar
  - Probability from a node to reach the other
- Considers paths of different lengths

S. Cao, W. Lu, Q.i Xu: *GraRep: Learning Graph Representations with Global Structural Information*. CIKM 2015

GraRep

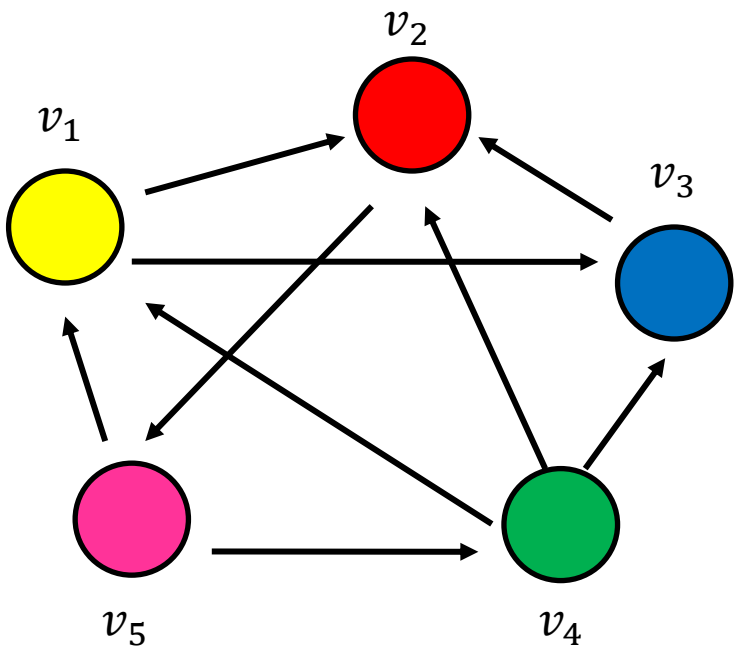
But not all k-neighbors equally important



Nearest neighborhoods more important

Maintain different k-step information differently in the graph representation

GraRep

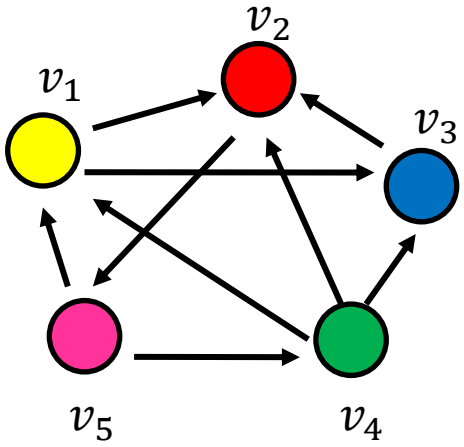


$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

$$P = D^{-1}A = \begin{bmatrix} 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \end{bmatrix}$$

Probabilistic adjacency matrix P_{ij} the probability of transition from node *i* to node *j* where the transition has *length exactly 1*

GraRep



$$P^2 = P * P = \begin{bmatrix} 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \end{bmatrix} * \begin{bmatrix} 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 & 0 \\ 1/2 & 0 & 0 & 1/2 & 0 \end{bmatrix}$$

Nodes reachable in 1-step **from node 2**

Nodes that reach **node 4** in one step

$$P^2 = \begin{bmatrix} 0 & 1/2 & 0 & 0 & 1/2 \\ 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1/2 & 1/6 & 0 & 1/3 \\ 1/6 & 5/12 & 5/12 & 0 & 0 \end{bmatrix}$$

P^2_{ij} the probability of transition form node *i* from node *j* when the transition has *length exactly 2*

GraRep

P_{ij}^{k}: Transition probability from node *i* to node *j* where the transition consists of **exactly** *k* steps

1. Minimize the loss for *a specific k*

$$L_k = \sum_{(i,j) \in V \times V} \| P_{ij}^{k} - z_i \cdot z_j \|^2$$

2. *Concatenate* the embeddings for the different *k*

Basic idea:

- Train embeddings to predict *k*-hop neighbors.
- Approach based on skipgrams

GraRep

Transition probability from node *i* (current node) to node *j (context node)* where the transition consists of *exactly k steps*

$$P_{ij}^{k} = p_k(j \mid i)$$

Skip-gram model

Given a center word w, predict the context words *c*, i.e., the words that appear within distance *k* from w

$$P_{cw}^{k} = p_k(c \mid w)$$

Learn two representations:

- One for node *i* as the source node (i.e., center word)
- One for node *i* as the destination node (i.e., context word)

GraRep

Use negative sampling (*) and maximum likelihood

Assume for a given *k*, the collection of all paths from G that start from *i* and end at *j*.

Maximize

(1) Probability that these pairs came from the graph, and

(2) Probability that all other pairs did not come form the graph

probability that pair (i, j) came from the graph

probability that pair (i, j) did not come from the graph

$$L_k(i) = \sum_{j \in V} \left(p_k(j|i) \log \sigma(z_i \cdot z_j)\right) + \lambda \; E_{j' \sim p_k(V)}[\log \sigma(-z_i \cdot z_{j'})]$$

σ: sigmoid function

hyper parameter indicating the number of negative samples

Sampled vertices drawn according to the vertex distribution over the graph (p_k(V))

GraRep

$$L_k(i) = \sum_{j \in V} (p_k(j|i) \log \sigma(z_i \cdot z_j)) + \lambda \; E_{j' \sim p_k(V)}[\log \sigma(-z_i \cdot z_{j'})]$$

Local objective for a specific pair of nodes

$$L_k(i,j) = P_{ij}^k \log \sigma(z_i \cdot z_j) + \frac{\lambda}{N} \sum_{j' \in V} P_{ij'}^k \log \sigma(-z_i \cdot z_{j'})]$$

As before, compute the gradient and use stochastic gradient descent

Or solve by setting = 0 and get

$$z_i z_j = \log\left(\frac{S_{i,jk}}{\sum_{i'} A_{i',jkk}}\right) - \log(\beta), \; \beta = \frac{\lambda}{N}$$

Summary

- **Basic idea:** Embed nodes so that distances in embedding space reflect node similarities in the original network.
- Different notions of node similarity:
  - Adjacency-based (i.e., similar if connected)
  - Multi-hop similarity definitions (HOPE, GraRep)
  - Random walk approaches (DeepWalk, node2vec)
- No one method wins in all cases
  - e.g., node2vec performs better on node classification while multi-hop methods performs better on link prediction

LINK ANG SUBGRAPH EMBEDDINGS

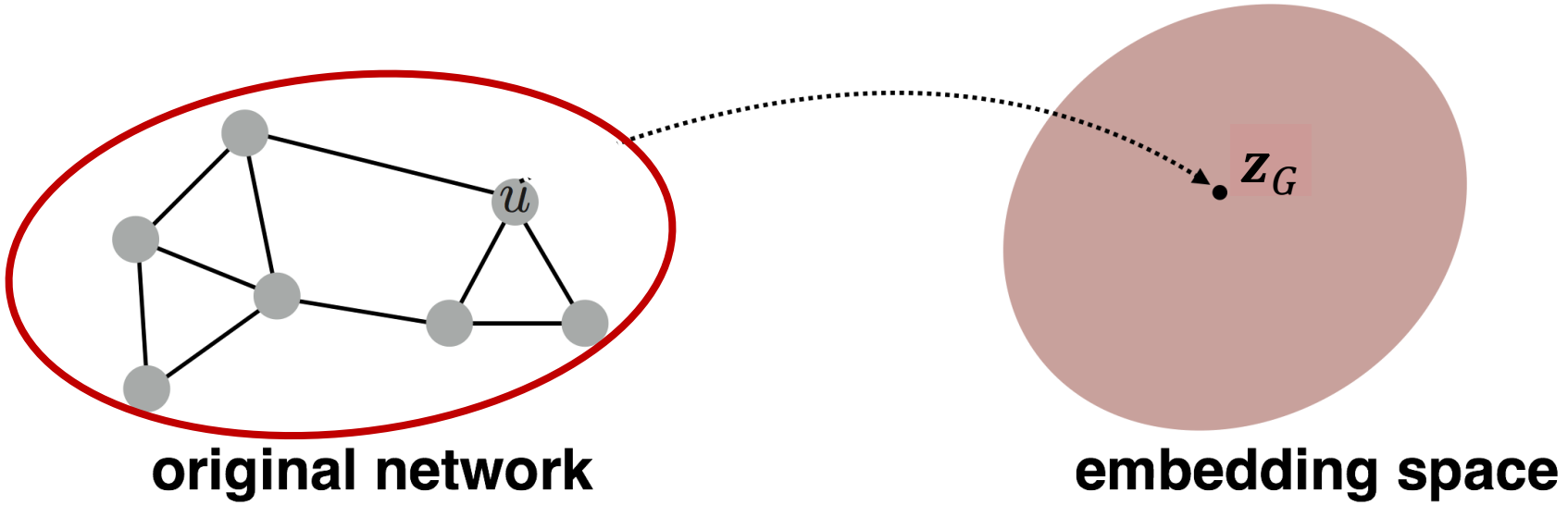
From node to link embeddings

Also learns edge vectors based on the vectors of their endpoints

Operator	Symbol	Definition
Average	$\boxplus$	$[f(u) \boxplus f(v)]_i = \frac{f_i(u)+f_i(v)}{2}$
Hadamard	$\boxdot$	$[f(u) \boxdot f(v)]_i = f_i(u) * f_i(v)$
Weighted-L1	$\|\cdot\|_{\bar{1}}$	$\|f(u) \cdot f(v)\|_{\bar{1}i} = \|f_i(u) - f_i(v)\|$
Weighted-L2	$\|\cdot\|_{\bar{2}}$	$\|f(u) \cdot f(v)\|_{\bar{2}i} = \|f_i(u) - f_i(v)\|^2$

Embedding Entire Graphs

- **Goal:** Want to embed a subgraph or an entire graph G. Graph embedding: $\mathbf{z}_G$.



- **Tasks:**
  - Classifying toxic vs. non-toxic molecules
  - Identifying anomalous graphs

Approach 1

Simple (but effective) approach 1:

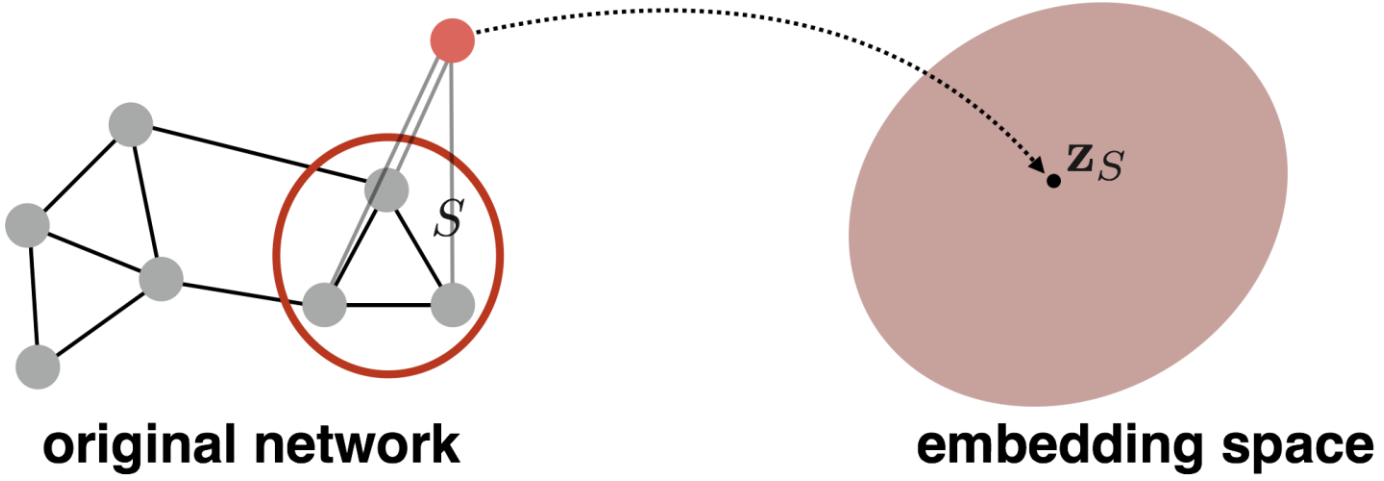
- Run a standard graph embedding technique *on* the (sub)graph G.
- Then just sum (or average) the node embeddings in the (sub)graph G.

$$\mathbf{z}_G = \sum_{v \in G} \mathbf{z}_v$$

Used by Duvenaud et al., 2016 to classify molecules based on their graph structure

Approach 2

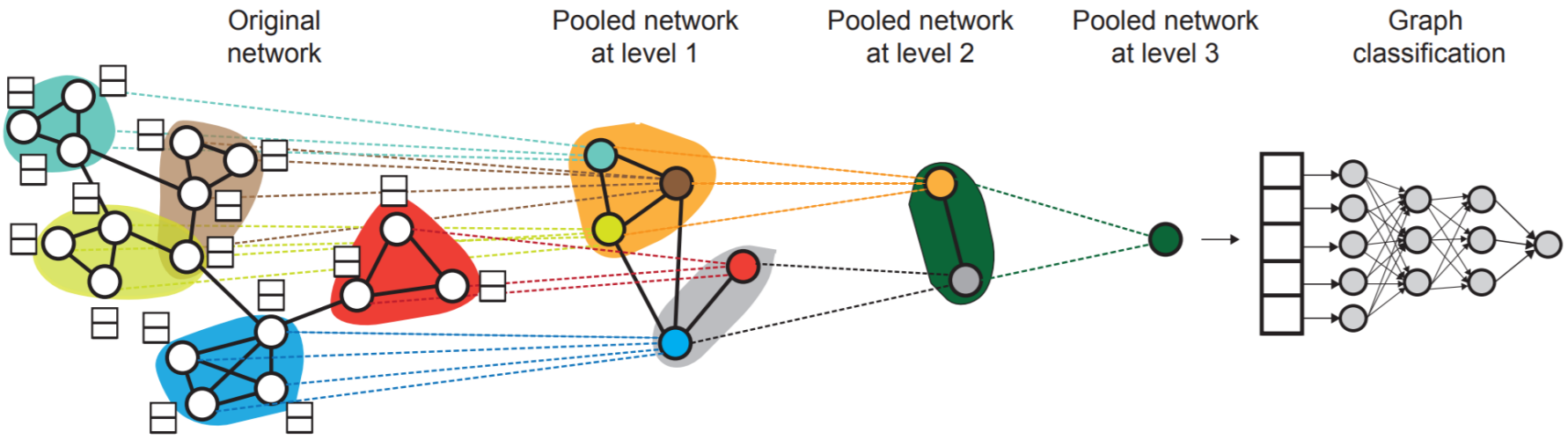
- **Approach 2:** Introduce a **"virtual node"** to represent the (sub)graph and run a standard graph embedding technique



Proposed by Li et al., 2016 as a general technique for subgraph embedding

Preview: Hierarchical Embeddings

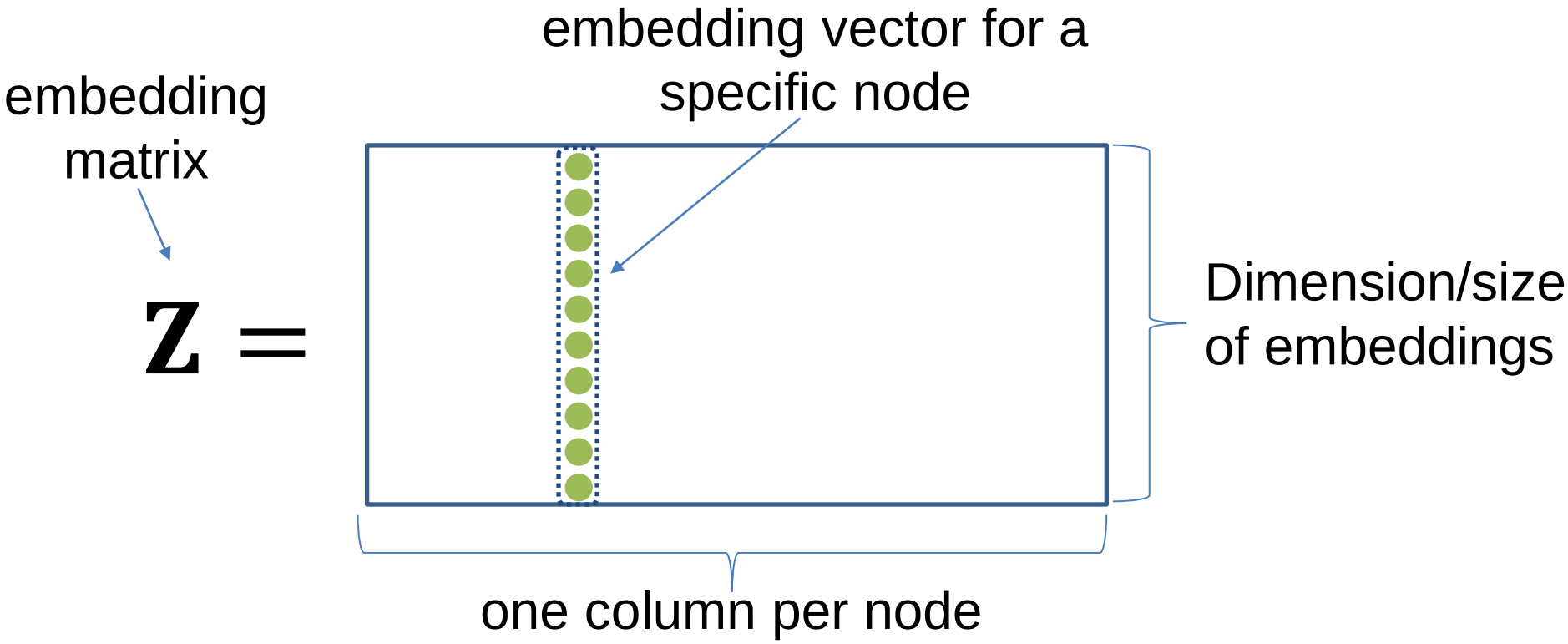
- **DiffPool:** We can also **hierarchically** cluster nodes in graphs, and **sum/avg** the node embeddings according to these clusters.



EMBEDDINGS AND FACTORIZATION

Embeddings & Matrix Factorization

Recall: encoder as an embedding lookup



Objective: maximize $\mathbf{z}_v^{\mathrm{T}}\mathbf{z}_u$ for node pairs (u, v) that are **similar**

Matrix Factorization

Simplest nodes similarity, two nodes are similar if connected by an edge.

Exact factorization $A = \boldsymbol{Z}^{\boldsymbol{T}}\boldsymbol{Z}$ is generally not possible

However, we can learn $\boldsymbol{Z}$ approximately

- **Objective**: $\min_{\mathbf{Z}} \| \mathrm{A} - \boldsymbol{Z}^{T}\boldsymbol{Z} \|_2$
  - We optimize $\boldsymbol{Z}$ such that it minimizes the L2 norm (Frobenius norm) of $\mathrm{A} - \boldsymbol{Z}^{T}\boldsymbol{Z}$
  - Note today we used softmax instead of L2. But the goal to approximate A with $\boldsymbol{Z}^{T}\boldsymbol{Z}$ is the same.

Conclusion: **Inner product decoder with node similarity defined by edge connectivity is equivalent to matrix factorization of A.**

Random Walk-based Similarity

- **DeepWalk** and **node2vec** have a more complex **node similarity** definition based on random walks
- **DeepWalk** is equivalent to matrix factorization of the following complex matrix expression:

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

  - Explanation of this equation is on the next slide.

Random Walk-based Similarity

Volume of graph

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

Diagonal matrix D

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

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

context window size

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

Power of normalized adjacency matrix

Number of negative samples

- **Node2vec** can also be formulated as a matrix factorization (albeit a more complex matrix)
- Refer to the paper for more details:

Network Embedding as Matrix Factorization: Unifying DeepWalk, LINE, PTE, and node2vec, WSDM 18

SUMMARY

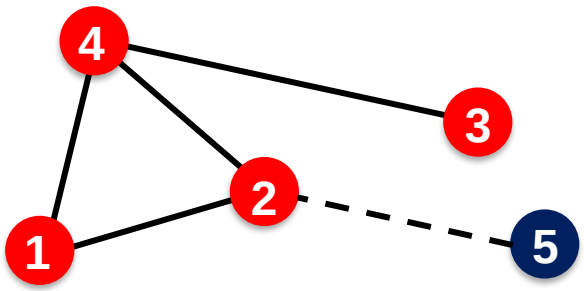
How to Use Embeddings

- **How to use embeddings z_i of nodes:**
  - **Clustering/community detection:** Cluster points $\boldsymbol{z_i}$
  - **Node classification:** Predict label of node i based on $\boldsymbol{z_i}$
  - **Link prediction:** Predict edge (i, j) based on $(\boldsymbol{z_i}, \boldsymbol{z_j})$
    - Where we can: concatenate, avg, product, or take a difference between the embeddings:
      - Concatenate: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g([\boldsymbol{z}_i, \boldsymbol{z}_j])$
      - Hadamard: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g(\boldsymbol{z}_i * \boldsymbol{z}_j)$ (per coordinate product)
      - Sum/Avg: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g(\boldsymbol{z}_i + \boldsymbol{z}_j)$
      - Distance: $f(\boldsymbol{z}_i, \boldsymbol{z}_j) = g(||\boldsymbol{z}_i - \boldsymbol{z}_j||_2)$
  - **Graph classification**: Graph embedding $\boldsymbol{z_G}$ via aggregating node embeddings or virtual-node. Predict label based on graph embedding $\boldsymbol{z}_G$.

Limitations (1)

Limitations of node embeddings via matrix factorization and random walks

- **Transductive (not inductive) method:** Cannot obtain embeddings for nodes not in the training set. Cannot apply to new graphs, evolving graphs.



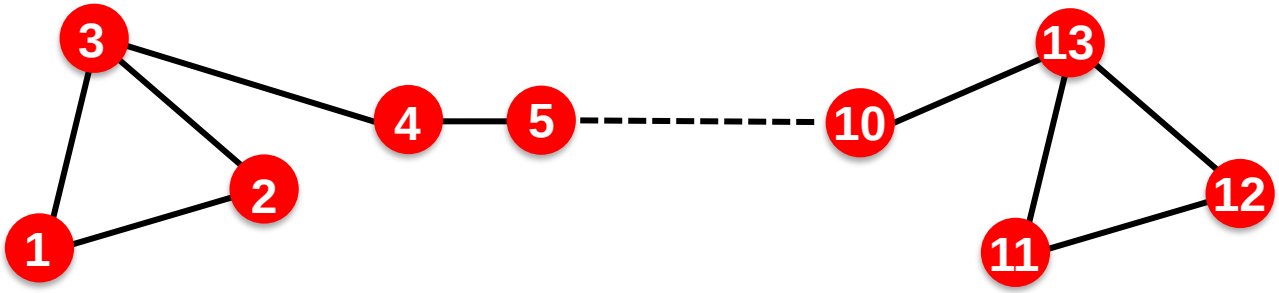
Training set

A newly added node 5 at test time (e.g., new user in a social network)

Cannot compute its embedding with DeepWalk / node2vec. Need to recompute all node embeddings.

Limitation (2)

Cannot capture **structural similarity**:



- Node 1 and 11 are **structurally similar** – part of one triangle, degree 2, …
- However, they have very **different** embeddings.
  - It is unlikely that a random walk will reach node 11 from node 1.

DeepWalk and node2vec do not capture structural similarity.

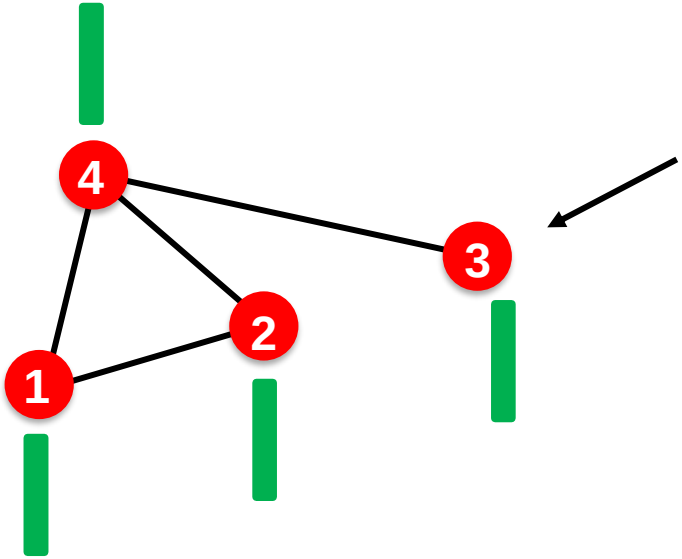
struct2vec

- $similarity_r(u, v)$: based on the difference of the degree sequence of nodes at radius r of u and v
- Builds a multilayer weighted graph, weights set based on similarity
- Perform a random walk: change layer, or do a weight-biased walk in the same layer

LFR Ribeiro et. al., struc2vec Learning Node Representations from Structural Identity, KDD 2017

Limitations (3)

- Cannot utilize node, edge and graph features



Feature vector
(e.g. protein properties in a protein-protein interaction graph)

DeepWalk / node2vec embeddings do not incorporate such node features

Summary

We discussed **graph representation learning**, a way to learn **node and graph embeddings** for downstream tasks, **without feature engineering**.

- **Encoder-decoder framework:**
  - **Encoder: embedding lookup**
  - **Decoder: predict score based on embedding to match node similarity**
- **Node similarity measure: (biased) random walk**
  - **Examples: DeepWalk, Node2Vec**
- **Extension to Graph embedding: Node embedding aggregation**

Acknowledgement

Most slides from

CS224W: Machine Learning with Graphs, Jure Leskovec, Stanford University, http://cs224w.stanford.edu