Stanford CS224W: Node Embeddings

CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

Stanford CS224W: Announcements

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

Course Logistics: Colab 0

- **Colab 0 will be released today by 9PM on our course website**
- **Colab 0:**
  - Overview of NetworkX and PyTorch Geometric
  - Does not need to be handed in
  - TAs will hold a recitation session to walk you through Colab 0:
    - Time: Friday (09/27), 3-4pm PT
    - Location: Zoom, link is posted on Ed
    - Session will be recorded

Course Logistics: Colab 1

- **Colab 1 will be released today by 9PM on our course website**
- **Colab 1:**
  - Will cover material from Lectures 1-2, so you can get started right away!
  - Due on Thursday 10/10 (2 weeks from today)
  - Submit written answers and code on Gradescope

Stanford CS224W: Node Embeddings

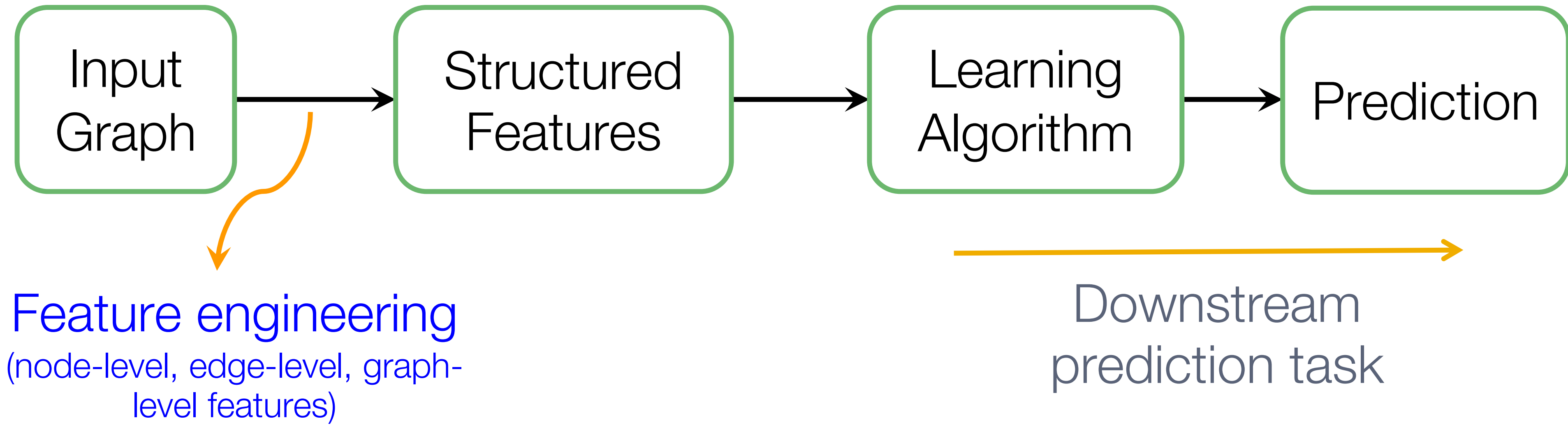
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

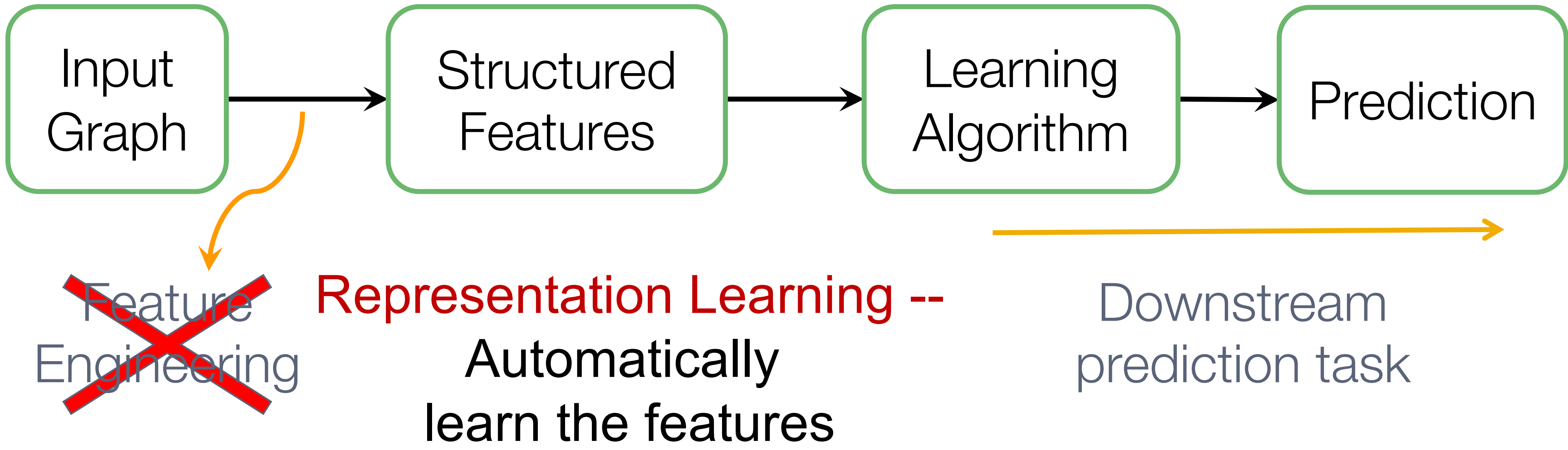
Recap: Traditional ML

Given an input graph, extract node, link and graph-level features, then learn a model (SVM, neural network, etc.) that maps features to labels.



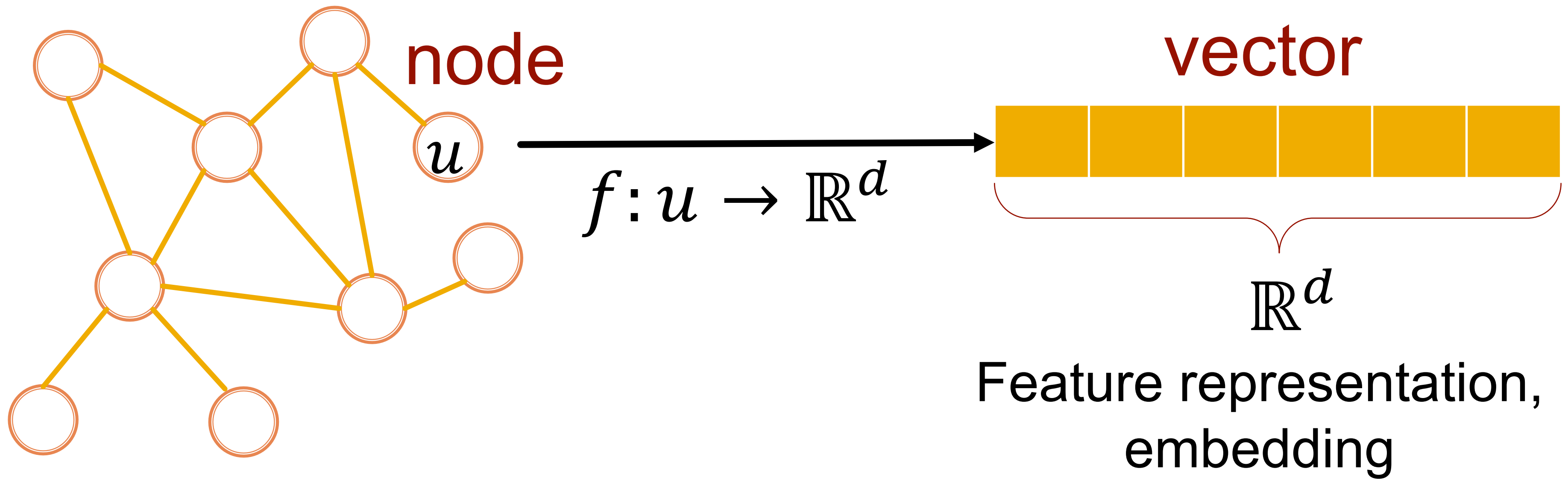
Graph Representation Learning

Graph Representation Learning alleviates the need to do feature engineering every single time.


Input Graph
Structured Features
Learning Algorithm
Prediction
~~Feature Engineering~~
Representation Learning --
Automatically learn the features
Downstream prediction task

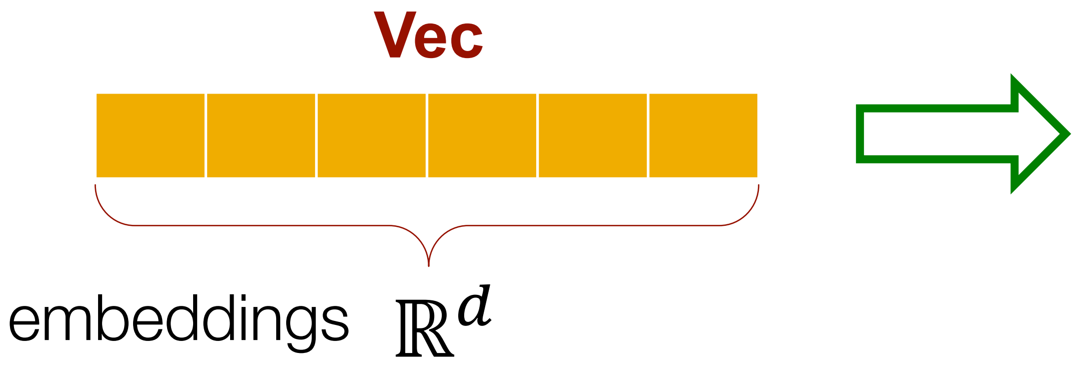
Graph Representation Learning

Goal: Efficient task-independent feature learning for machine learning with graphs!



Why Embedding?

- **Task: Map nodes into an embedding space**
  - Similarity of embeddings between nodes indicates their similarity in the network. For example:
    - Both nodes are close to each other (connected by an edge)
  - Encode network information
  - Potentially used for many downstream predictions

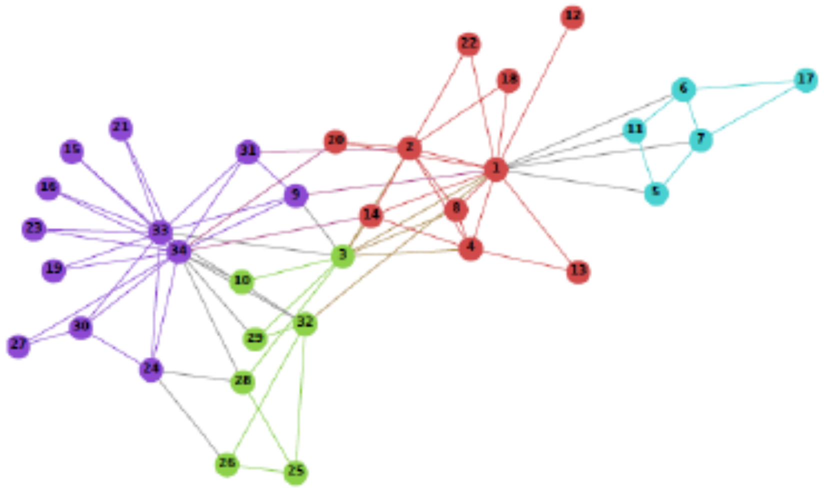


Tasks

- Node classification
- Link prediction
- Graph classification
- Anomalous node detection
- Clustering
- ….

Example Node Embedding

- **2D embedding of nodes of the Zachary's Karate Club network:**



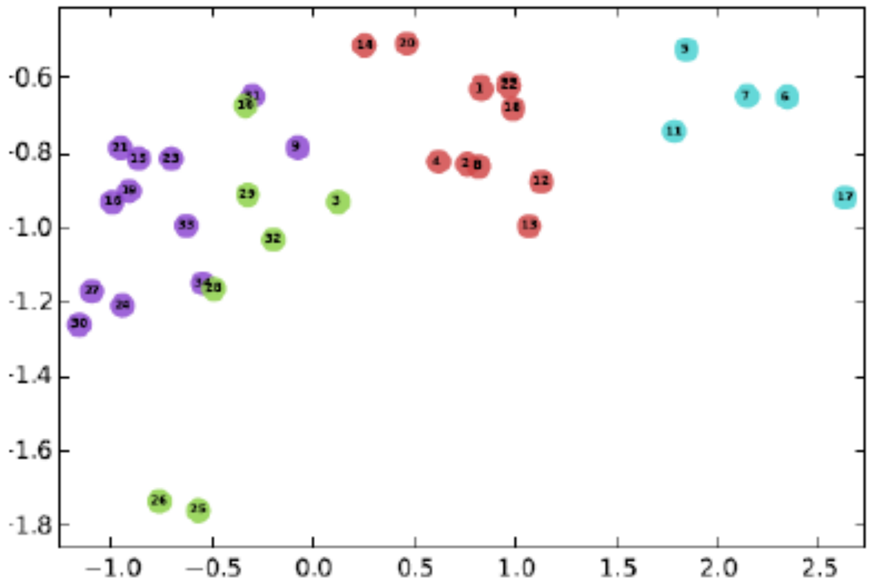


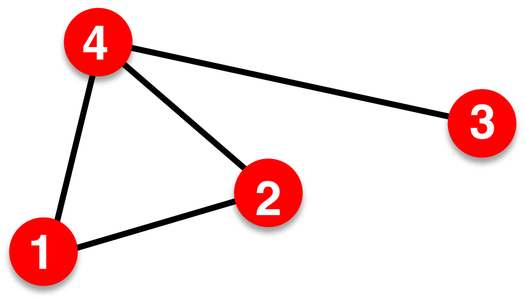
Image from: Perozzi et al. DeepWalk: Online Learning of Social Representations. *KDD 2014.*

Stanford CS224W: Node Embeddings: Encoder and Decoder

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

Setup

- **Assume we have an (undirected) graph G:**
  - V is the vertex set.
  - A is the adjacency matrix (assume binary).
  - **For simplicity: No node features or extra information is used**

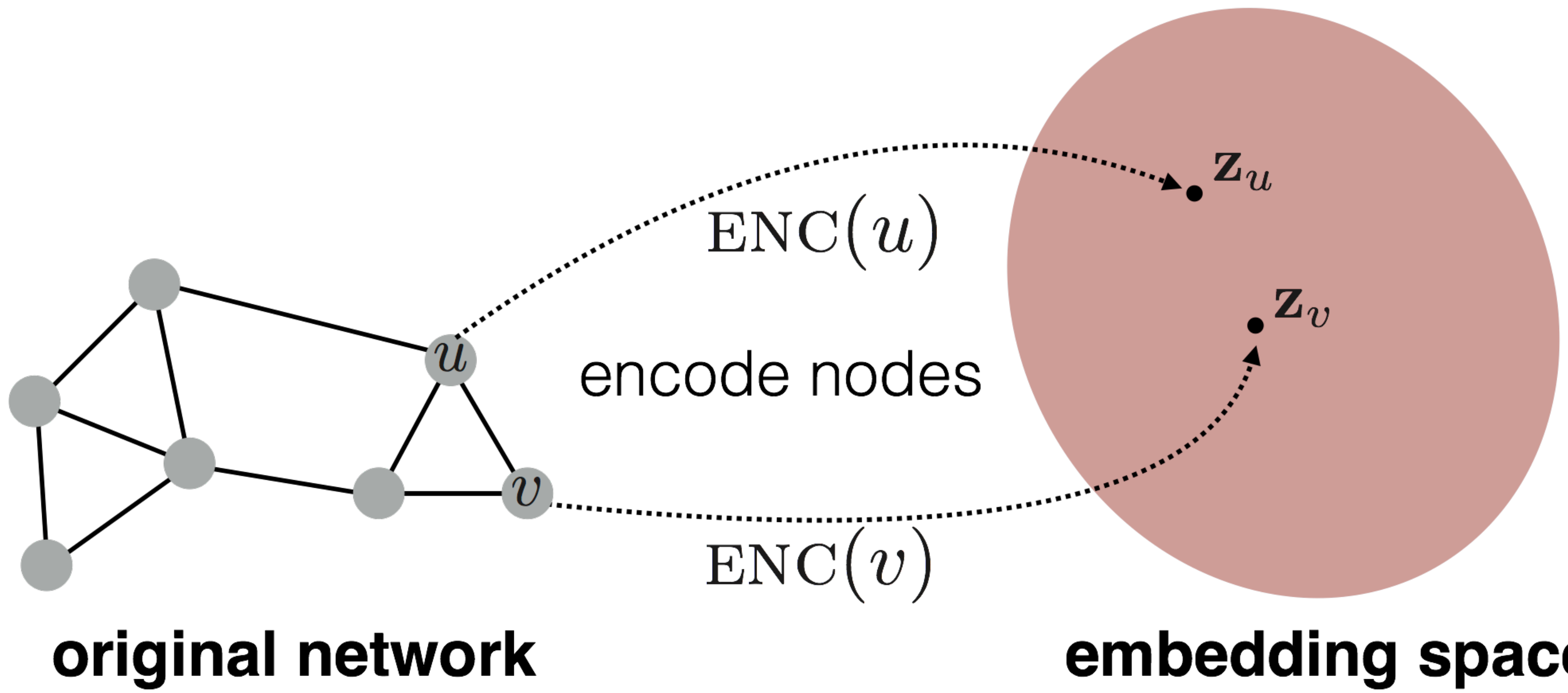


V: {1, 2, 3, 4}

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Embedding Nodes

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



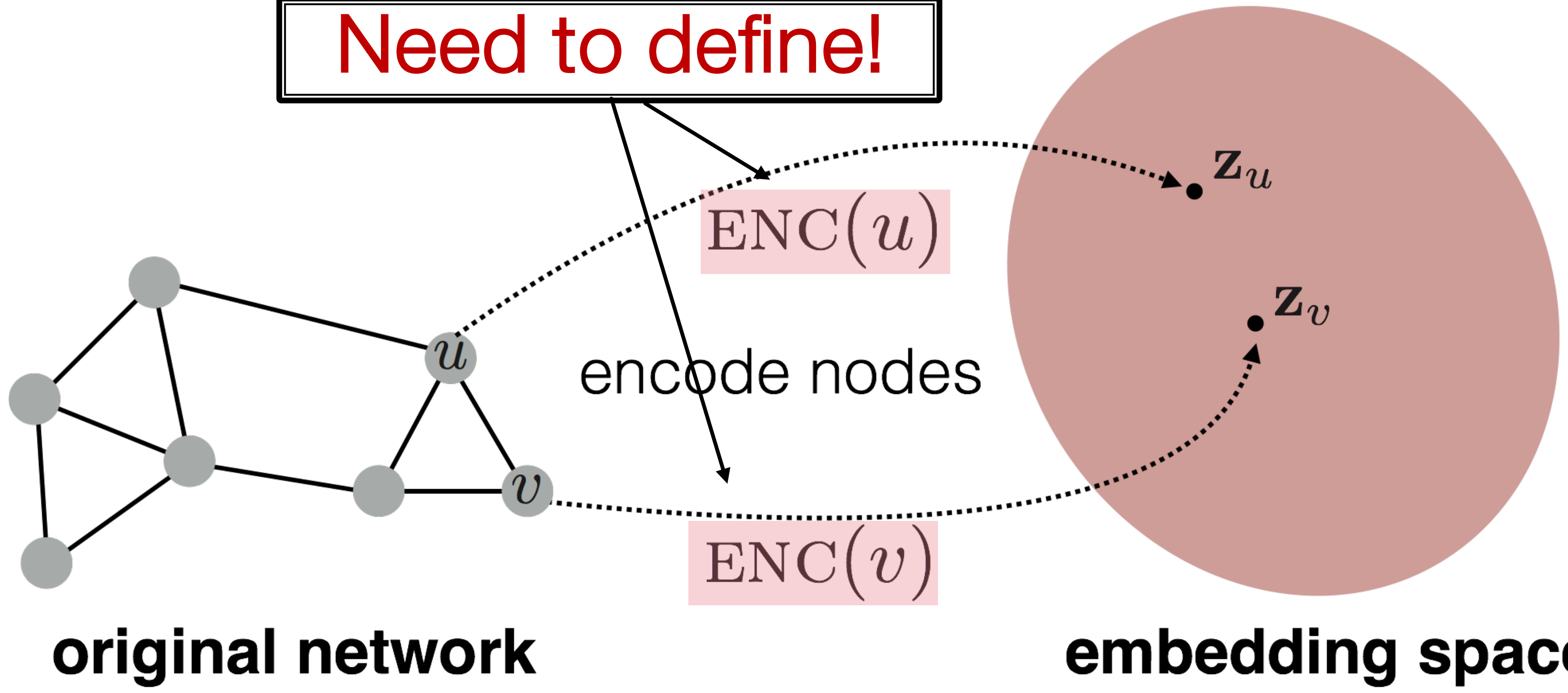
Embedding Nodes

Goal: $\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$

in the original network

Similarity of the embedding

Need to define!



Learning Node Embeddings

1. **Encoder** maps from nodes to embeddings
2. **Define a node similarity function** (i.e., a measure of similarity in the original network)
3. **Decoder** DEC maps from embeddings to the similarity score
4. **Optimize the parameters of the encoder so that:**

$\mathrm{DEC}(\mathbf{z}_v^{\mathrm{T}}\mathbf{z}_u)$

$$\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}}\mathbf{z}_u$$

in the original network Similarity of the embedding

Two Key Components

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

$$\text{ENC}(v) = \boxed{\mathbf{z}_v}$$

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

$$\text{similarity}(u, v) \approx \mathbf{z}_v^{\mathrm{T}} \mathbf{z}_u$$

Decoder

Similarity of u and v in the original network

dot product between node embeddings

“Shallow” Encoding

Simplest encoding approach: **Encoder is just an embedding-lookup**

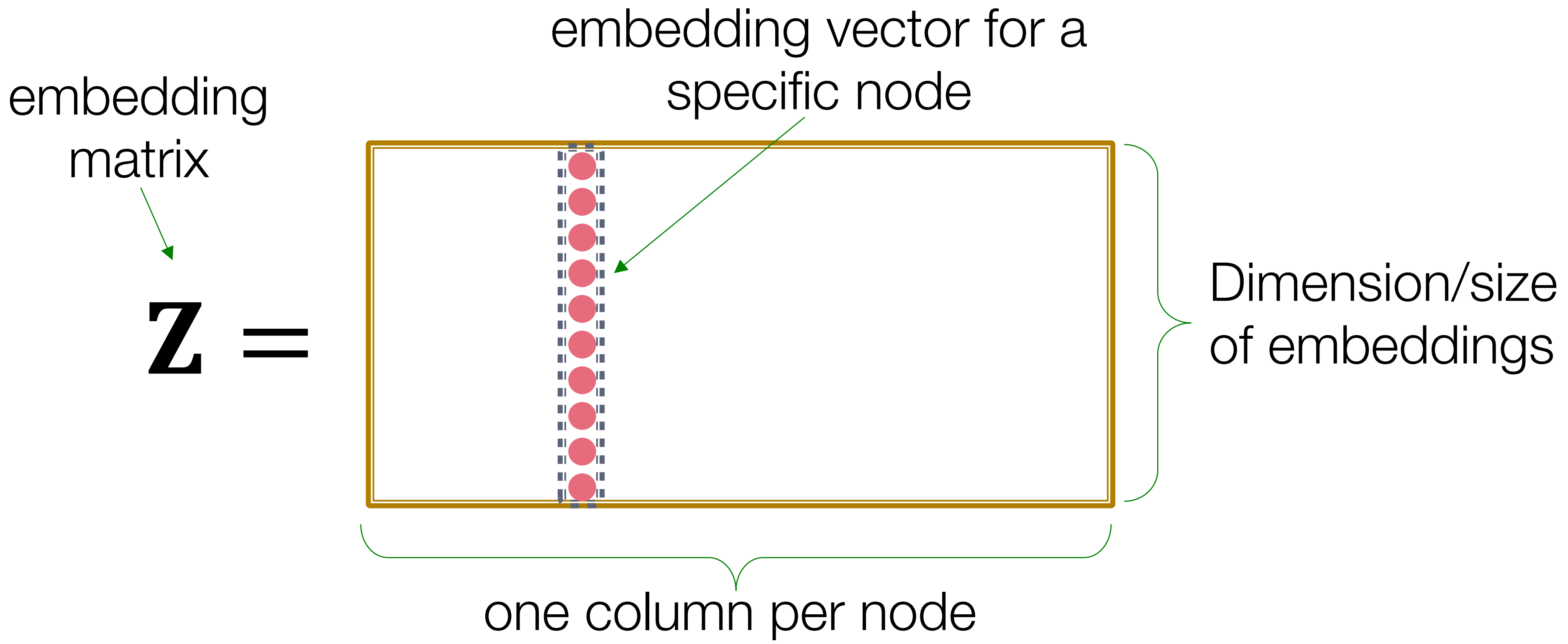
$$\mathrm{ENC}(v) = \mathbf{z}_v = \mathbf{Z} \cdot v$$

$\mathbf{Z} \in \mathbb{R}^{d \times |\mathcal{V}|}$ matrix, each column is a node embedding [what we learn / optimize]

$v \in \mathbb{I}^{|\mathcal{V}|}$ indicator vector, all zeroes except a one in column indicating node *v*

“Shallow” Encoding

Simplest encoding approach: **encoder is just an embedding-lookup**


embedding vector for a specific node
embedding matrix
$\mathbf{Z} =$
Dimension/size of embeddings
one column per node

"Shallow" Encoding

Simplest encoding approach: **Encoder is just an embedding-lookup**

Each node is assigned a unique embedding vector
(i.e., we directly optimize the embedding of each node)

Many methods: DeepWalk, node2vec

Framework Summary

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

  - **Decoder:** based on node similarity.
  - **Objective:** maximize $\mathbf{z}_v^{\mathrm{T}}\mathbf{z}_u$ 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”?
- We will now learn node similarity definition that uses **random walks**, and how to optimize embeddings for such a similarity measure.

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.

Stanford CS224W: Random Walk Approaches for Node Embeddings

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

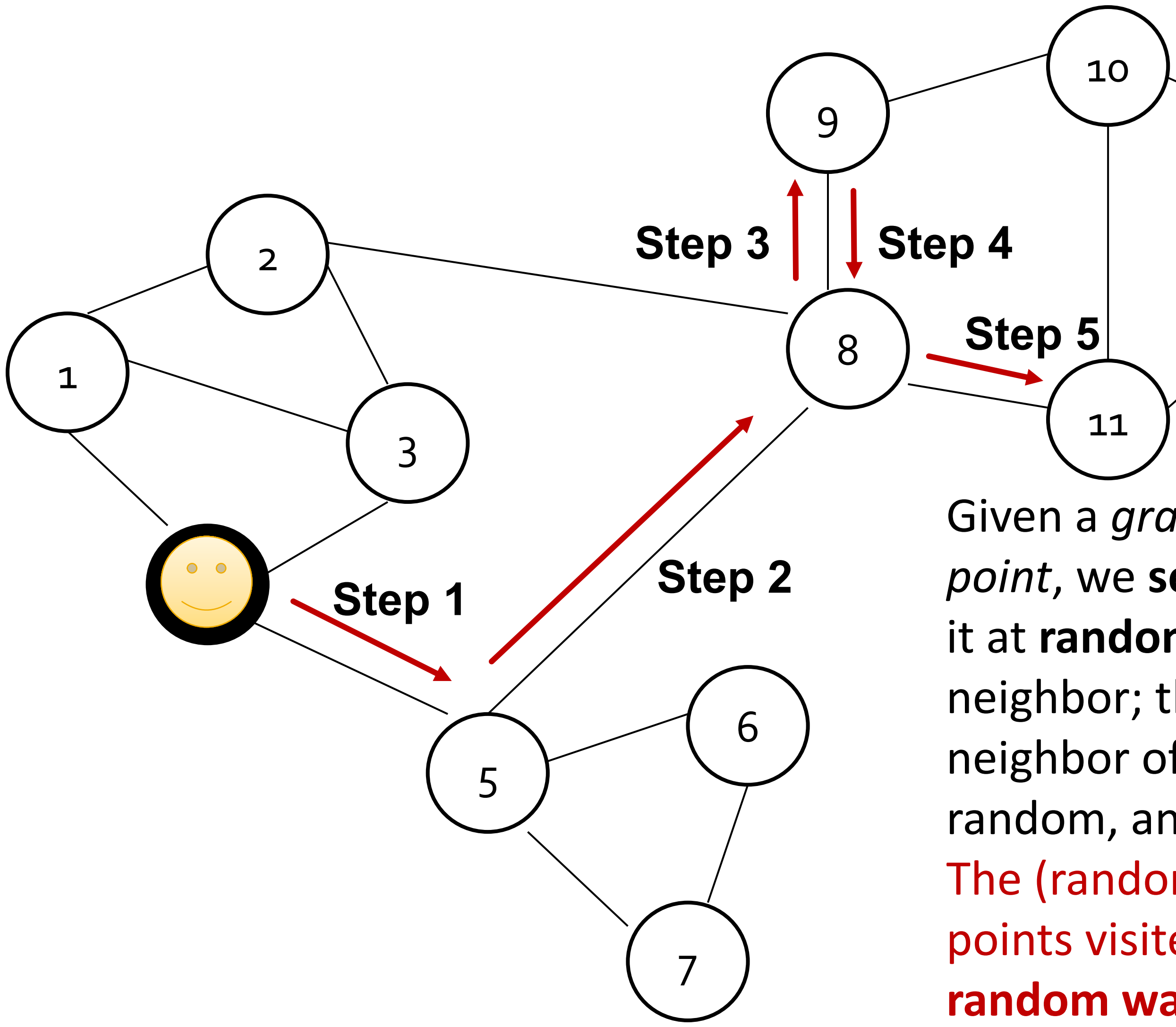
Notation

- **Vector** $\mathbf{z}_u$:
  - The embedding of node u (what we aim to find).
- **Probability** $P(v \,|\mathbf{z}_u)$: $\Longleftarrow$ Our model prediction based on $\mathbf{z}_u$
  - The **(predicted) probability** of visiting node v on random walks starting from node u.

Non-linear functions used to produce predicted **probabilities**

- **Softmax** function:
  - Turns vector of K real values (model predictions) into K probabilities that sum to 1: $S(\mathbf{z})[i] = \frac{e^{\mathbf{z}[i]}}{\sum_{j=1}^{K} e^{\mathbf{z}[j]}}$
- **Sigmoid** function:
  - S-shaped function that turns real values into the range of (0, 1). Written as $\sigma(x) = \frac{1}{1+e^{-x}}$.

Random Walk



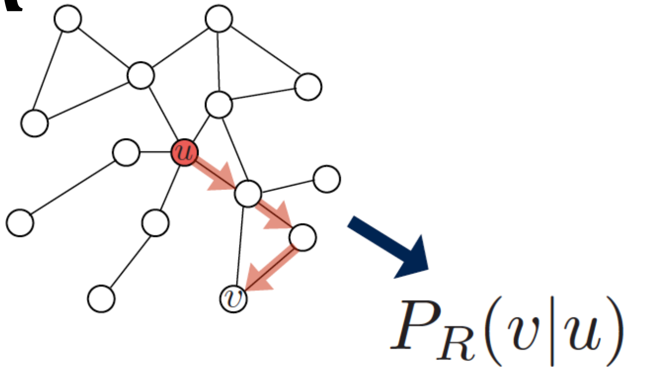
Given a *graph* and a *starting point*, we **select a neighbor** of it at **random**, and move to this neighbor; then we select a neighbor of this point at random, and move to it, etc. The (random) sequence of points visited this way is a **random walk on the graph**.

Random-Walk Embeddings

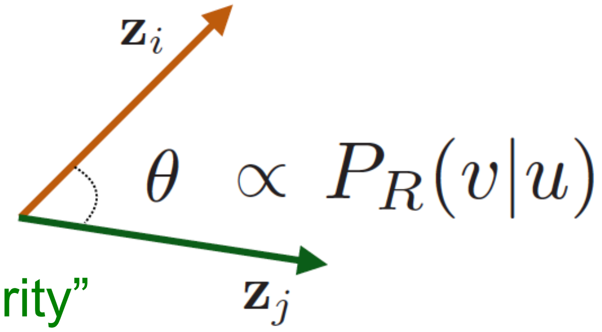
$$\mathbf{z}_u^{\mathrm{T}}\mathbf{z}_v \approx$$ probability that u and v co-occur on a random walk over the graph

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:**



Similarity in embedding space (Here: dot product=$\cos(\theta)$) encodes random walk "similarity"

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

Feature Learning as Optimization

- Given $G = (V, E)$,
- Our goal is to learn a mapping $f: u \to \mathbb{R}^d$:
  $f(u) = \mathbf{z}_u$
- Log-likelihood objective:

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

  - $N_R(u)$ is the neighborhood of node u by strategy R
- Given node u, we want to learn feature representations that are predictive of the nodes in its random walk neighborhood $N_R(u)$.

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

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

$$\mathcal{L} = \underbrace{\sum_{u \in V}}_{\text{sum over all nodes } u} \underbrace{\sum_{v \in N_R(u)}}_{\text{sum over nodes } v \text{ seen on random walks starting from } u} -\log\Bigg(\underbrace{\frac{\exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_v)}{\sum_{n \in V} \exp(\mathbf{z}_u^{\mathrm{T}} \mathbf{z}_n)}}_{\text{predicted probability of } u \text{ and } v \text{ co-occuring on random walk}}\Bigg)$$

Optimizing random walk embeddings =

Finding embeddings $\mathbf{z}_u$ 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
$\mathrm{O}(|\mathrm{V}|^2)$ 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

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

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

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

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

random distribution over nodes

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

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

- 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?

- So far we have described how to optimize embeddings given a random walk strategy R
- **What strategies should we use to run these random walks?**
  - Simplest idea: **Just run fixed-length, unbiased random walks starting from each node** (i.e., DeepWalk from Perozzi et al.)
    - The issue is that such notion of similarity is too constrained
- **How can we generalize this?**

Reference: Perozzi et al. DeepWalk: Online Learning of Social Representations. *KDD.*

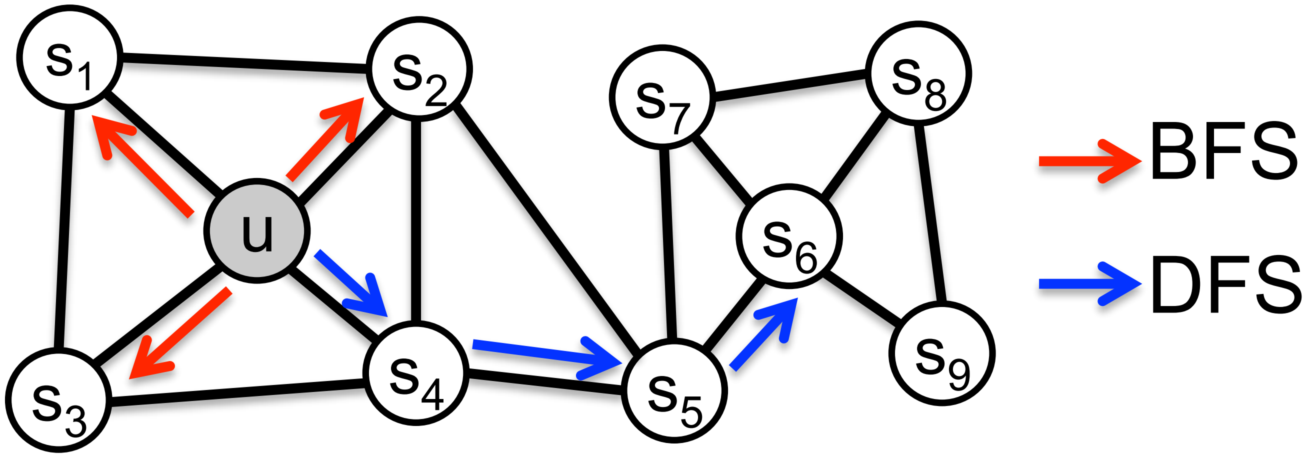
Overview of node2vec

- **Goal:** Embed nodes with similar network neighborhoods close in the feature space.
- We frame this goal as a maximum likelihood optimization problem, independent to the downstream prediction task.
- **Key observation:** Flexible notion of network neighborhood $N_R(u)$ of node u leads to rich node embeddings
- Develop biased 2nd order random walk R to generate network neighborhood $N_R(u)$ of node u

Reference: Grover et al. node2vec: Scalable Feature Learning for Networks. *KDD.*

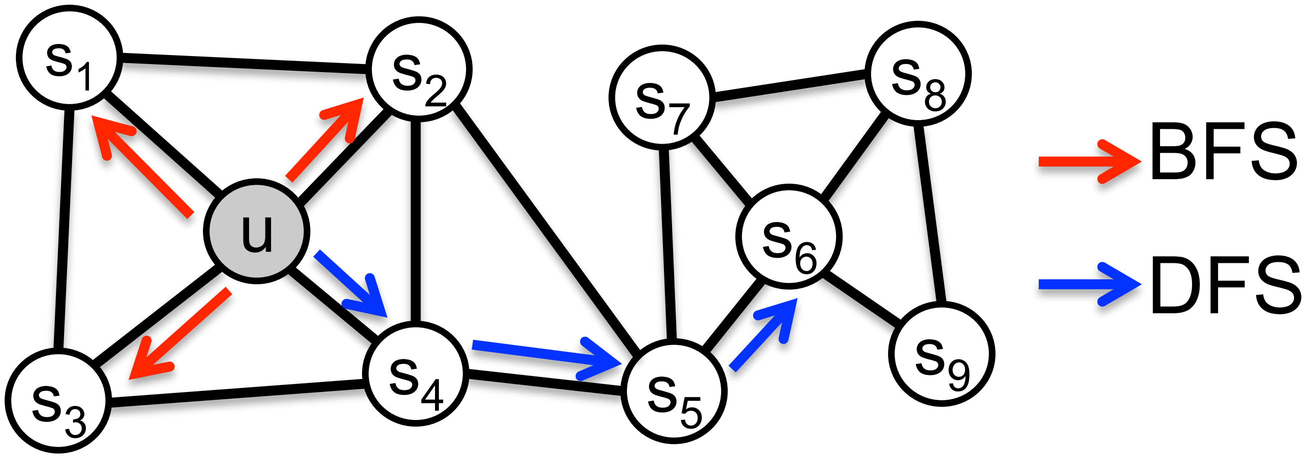
node2vec: Biased Walks

Idea: use flexible, biased random walks that can trade off between local and global views of the network (Grover and Leskovec).



node2vec: Biased Walks

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

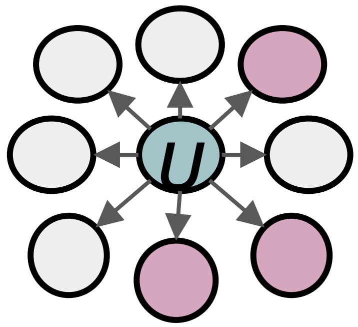


Walk of length 3 ($N_R(u)$ of size 3):

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

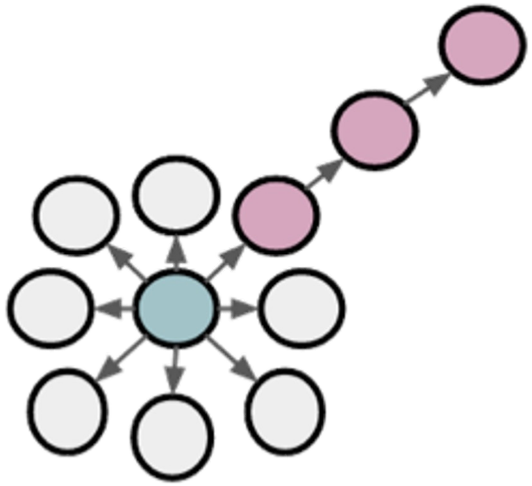
$N_{DFS}(u) = \{ s_4, s_5, s_6 \}$ Global macroscopic view

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

Interpolating BFS and DFS

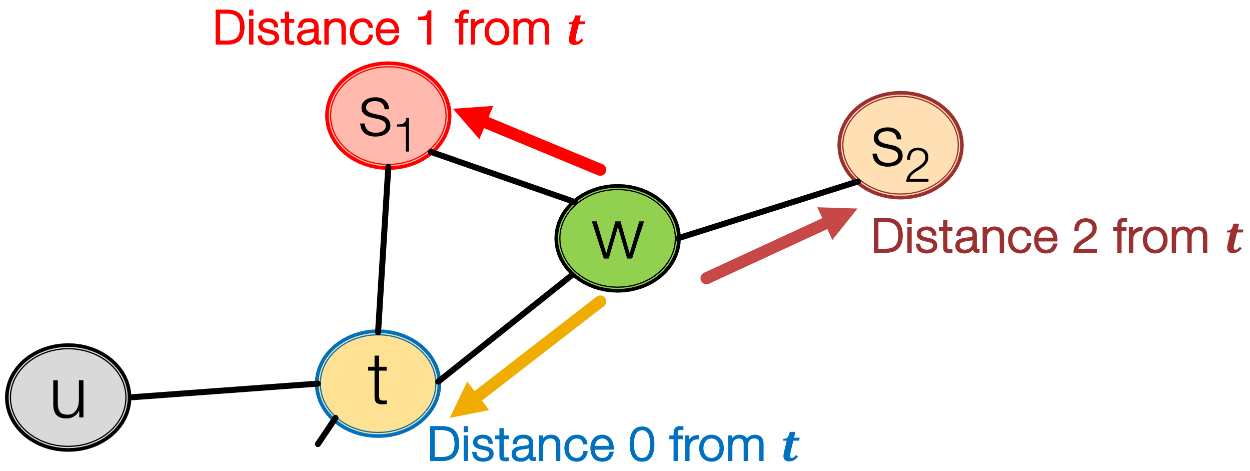
Biased fixed-length random walk R that given a node u generates neighborhood $N_R(u)$

- Random walk has two parameters:
  - **Return parameter p:**
    - Return back to the previous node
  - **In-out parameter q:**
    - Moving outwards (DFS) vs. inwards (BFS) from the previous node
    - Intuitively, q is the "ratio" of BFS vs. DFS
- Next, we 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

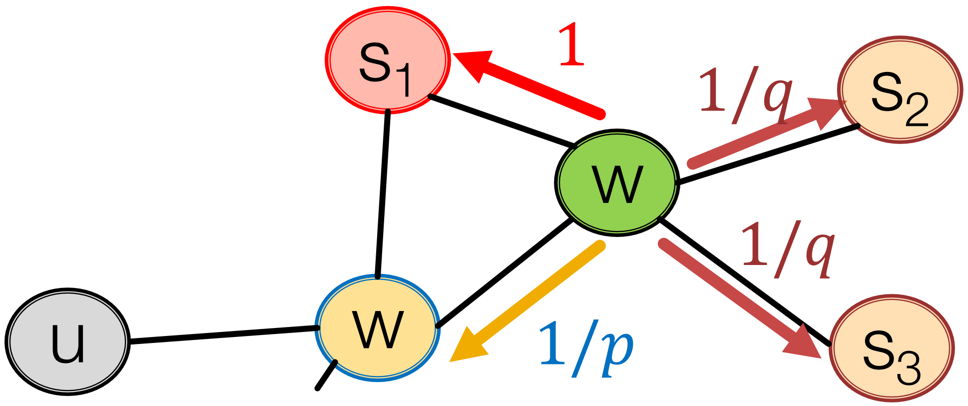
Define the random walk by specifying the walk transition probabilities on edges adjacent to the current node w:

- Rnd. walk **just traversed edge** (t, w) and **is now at** w
- We specify edge transition probs. out of node w
- **Insight:** Neighbors of w can only be:



One Step of the Biased Random Walk

- **Walker came over edge $(t, \mathbf{w})$ and is now at $\mathbf{w}$.**
  How to set edge transition probabilities?



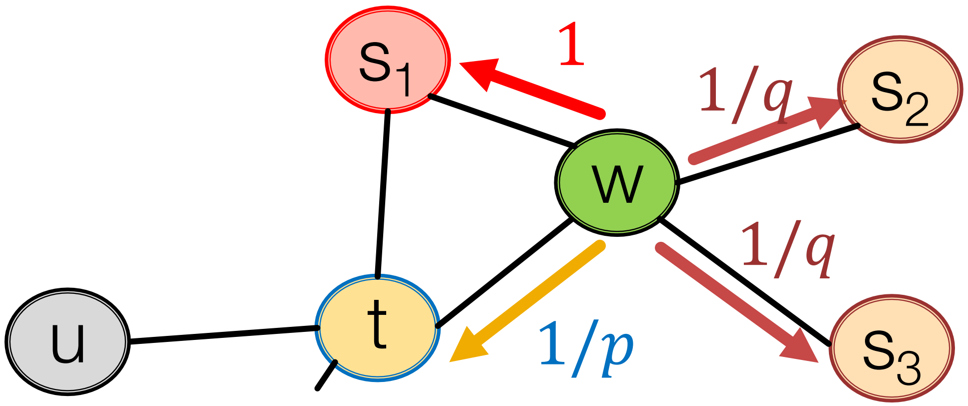
$1/p, 1/q, 1$ are unnormalized probabilities

- p, q model transition probabilities
  - p … return parameter
  - q … "walk away" parameter

One Step of the Biased Random Walk

- **Walker came over edge $(s_1, \mathbf{w})$ and is at $\mathbf{w}$.**

  How to set edge transition probabilities?



$\mathrm{w} \rightarrow$

Target x	Prob.	Dist. (t, x)
t	$1/p$	0
s_1	1	1
s_2	$1/q$	2
s_3	$1/q$	2

Unnormalized transition prob. segmented based on distance from t

- **BFS-like** walk: Low value of p
- **DFS-like** walk: Low value of q

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

node2vec algorithm

- 1) Compute edge transition probabilities:
  - For each edge (t, 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.).
  - Based on learned weights (Abu-El-Haija et al.)
- **Alternative optimization schemes:**
  - Directly optimize based on 1-hop and 2-hop random walk probabilities (as in LINE from Tang et al.).
- **Network preprocessing techniques:**
  - Run random walks on modified versions of the original network (e.g., Ribeiro et al. struct2vec, Chen et al. HARP).

Summary so far

- **Core idea:** Embed nodes so that distances in embedding space reflect node similarities in the original network.
- **Different notions of node similarity:**
  - Naïve: Similar if two nodes are connected
  - Random walk approaches (covered today)

Summary so far

- **So, what method should I use..?**
- No one method wins in all cases….
  - E.g., node2vec performs better on node classification while alternative methods perform better on link prediction (Goyal and Ferrara, survey).
- Random walk approaches are generally more efficient.
- **In general:** Must choose definition of node similarity that matches your application.

Stanford CS224W: Embedding Entire Graphs

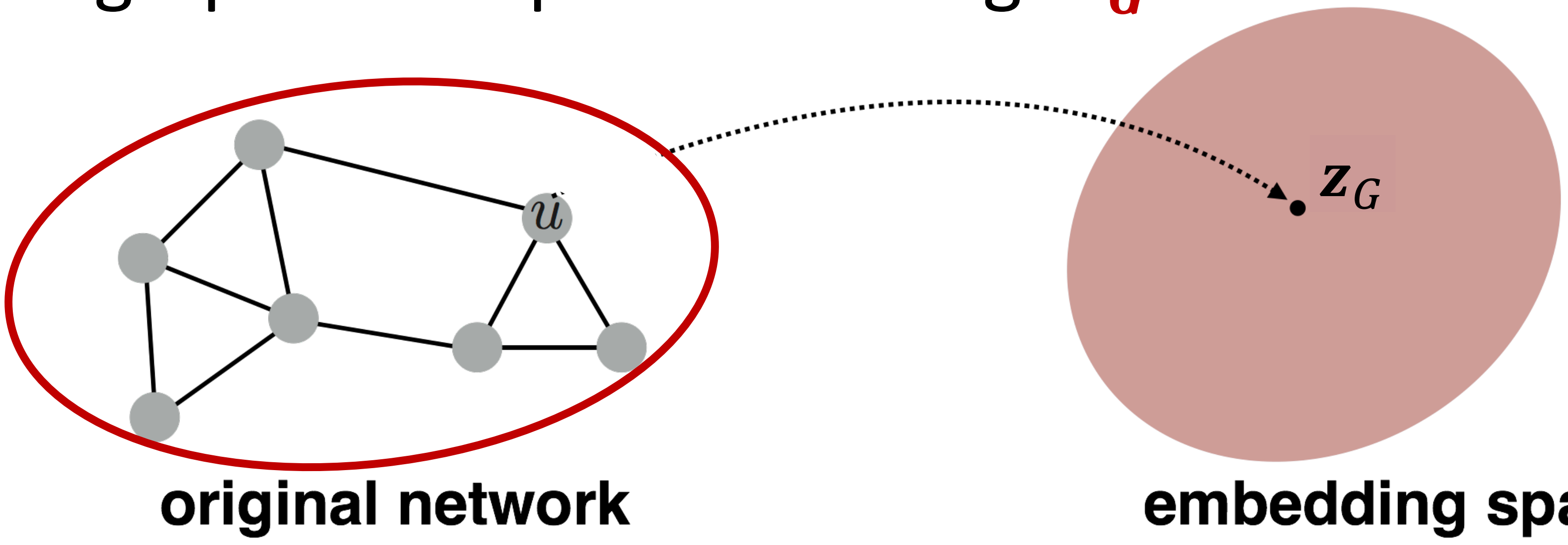
CS224W: Machine Learning with Graphs

Jure Leskovec, Stanford University

http://cs224w.stanford.edu

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:

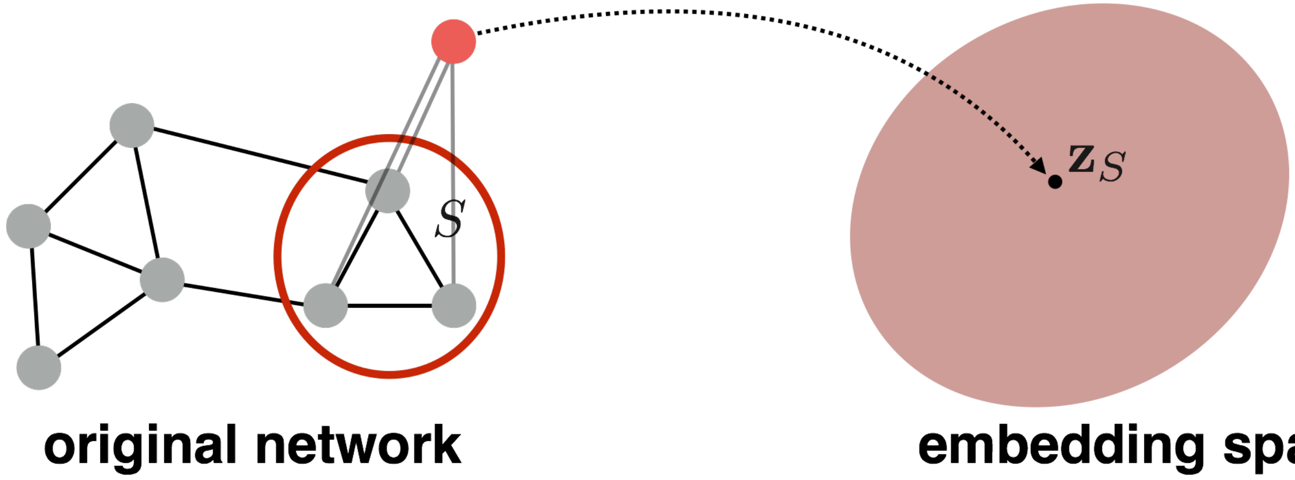
- Run a standard graph embedding technique *on* the (sub)graph G.
- Then just sum (or average) the node embeddings in the (sub)graph G.

$$\mathbf{z}_{G} = \sum_{v \in G} \mathbf{z}_{v}$$

- Used by Duvenaud et al. 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. as a general technique for subgraph embedding

Summary

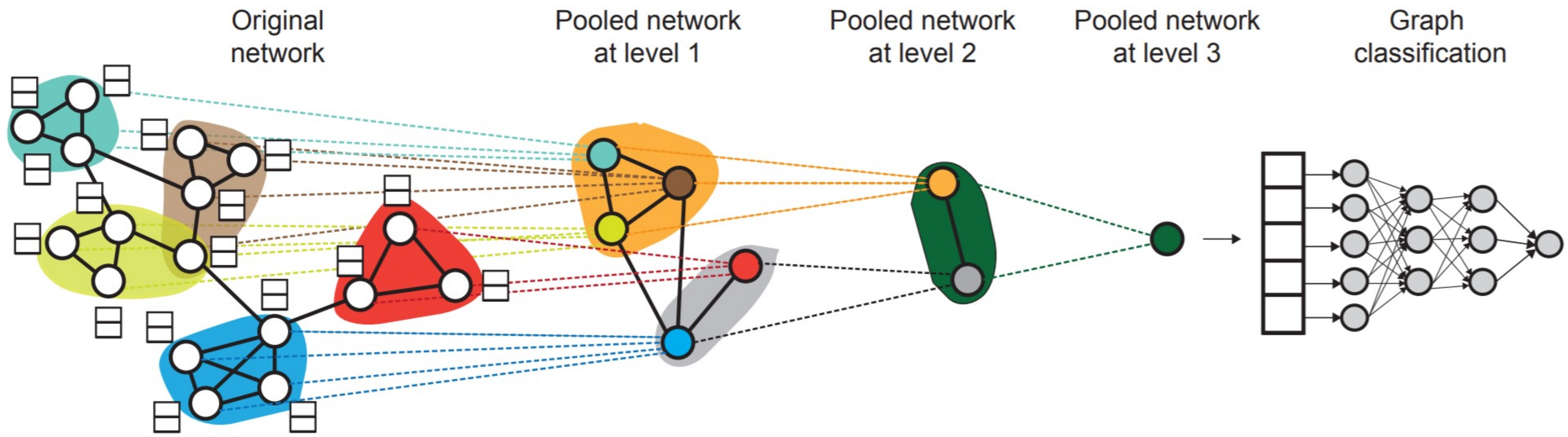
We discussed 2 ideas to graph embeddings:

- **Approach 1: Embed nodes and sum/avg them**

- **Approach 2: Create super-node that spans the (sub) graph and then embed that node.**

Preview: Hierarchical Embeddings

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

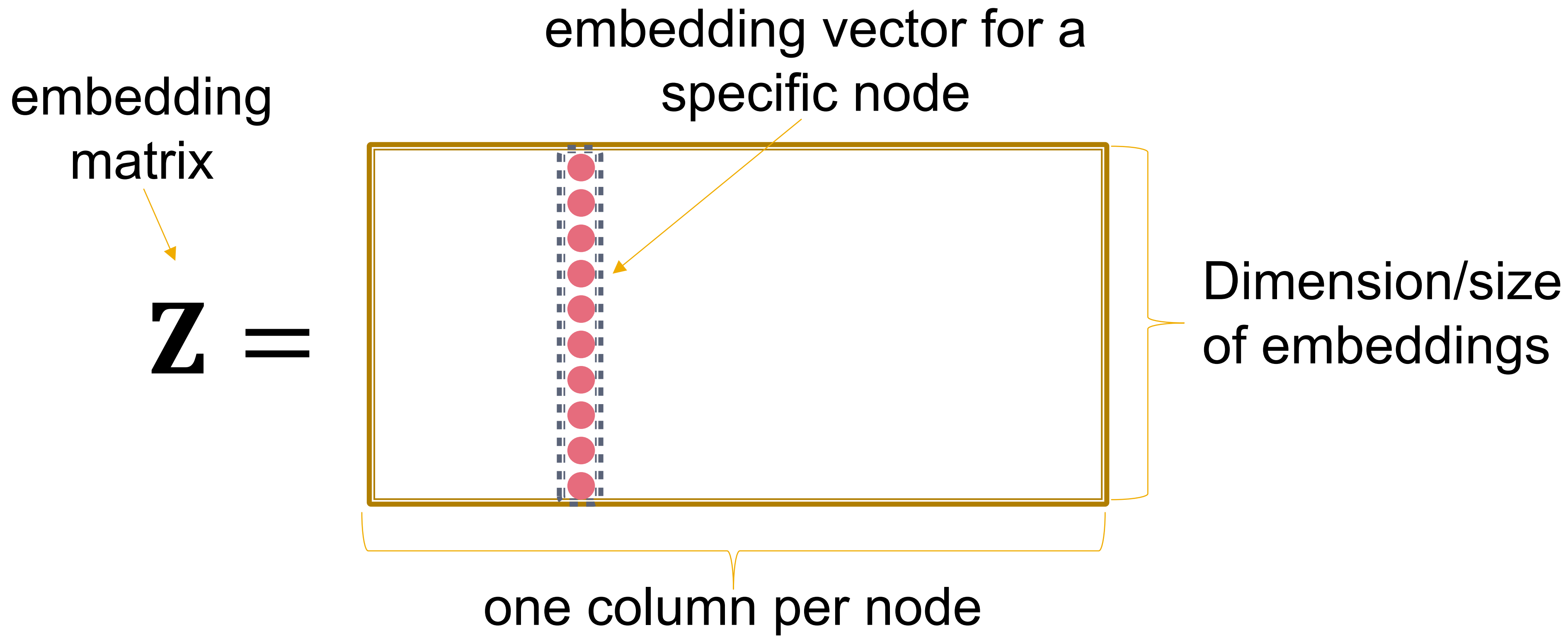

Original network
Pooled network at level 1
Pooled network at level 2
Pooled network at level 3
Graph classification

Stanford CS224W: Matrix Factorization and Node Embeddings

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

Embeddings & Matrix Factorization

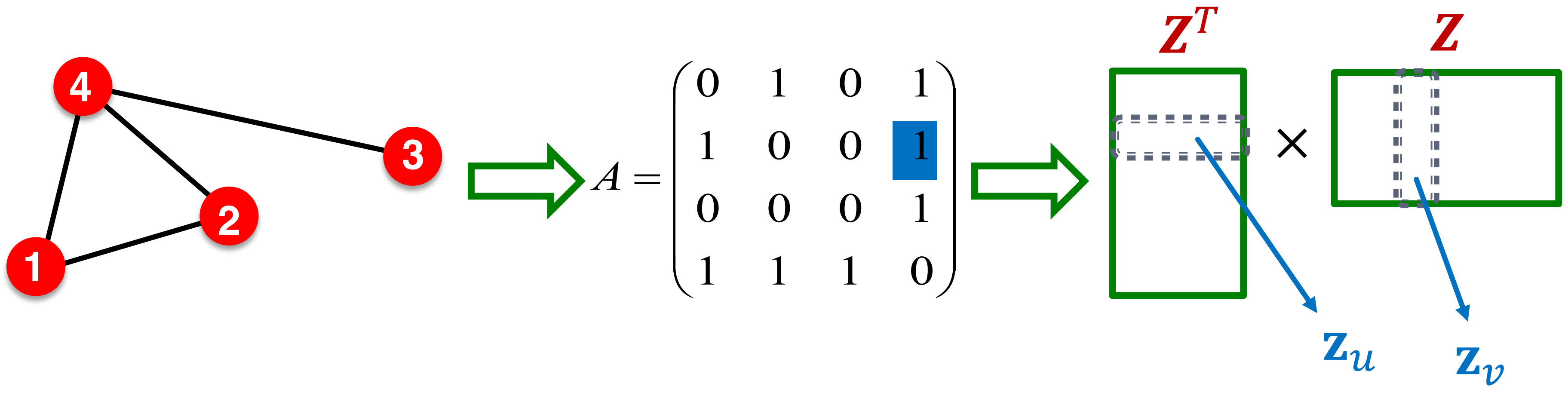
- **Recall:** encoder as an embedding lookup


embedding vector for a specific node
embedding matrix
$\mathbf{Z} =$
Dimension/size of embeddings
one column per node

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

Connection to Matrix Factorization

- 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, $\boldsymbol{Z}^T\boldsymbol{Z} = \boldsymbol{A}$



$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & \boxed{1} \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Matrix Factorization

- The embedding dimension d (number of rows in $\boldsymbol{Z}$) is much smaller than number of nodes n.
- Exact factorization $A = \boldsymbol{Z}^T\boldsymbol{Z}$ is generally not possible
- However, we can learn $\boldsymbol{Z}$ approximately
- **Objective:** $\min_{\mathbf{Z}} \parallel \mathrm{A} - \boldsymbol{Z}^T\boldsymbol{Z} \parallel_2$
  - 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.

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

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

See Lec 3 slide 30:

$T = |N_R(u)|$

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.

How to Use Embeddings

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

Today's Summary

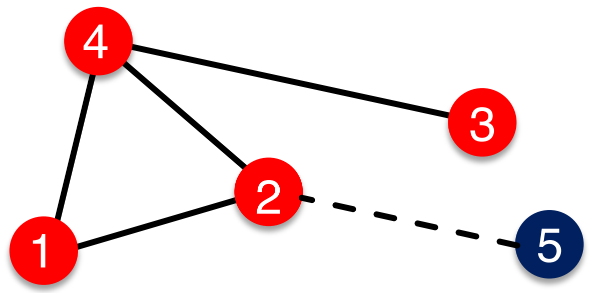
We discussed **graph representation learning**, a way to learn **node and graph embeddings** for downstream tasks, **without feature engineering**.

- **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**

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
    - If you apply DeepWalk to the same graph multiple times, each time you'll get a different embedding each time. Why?



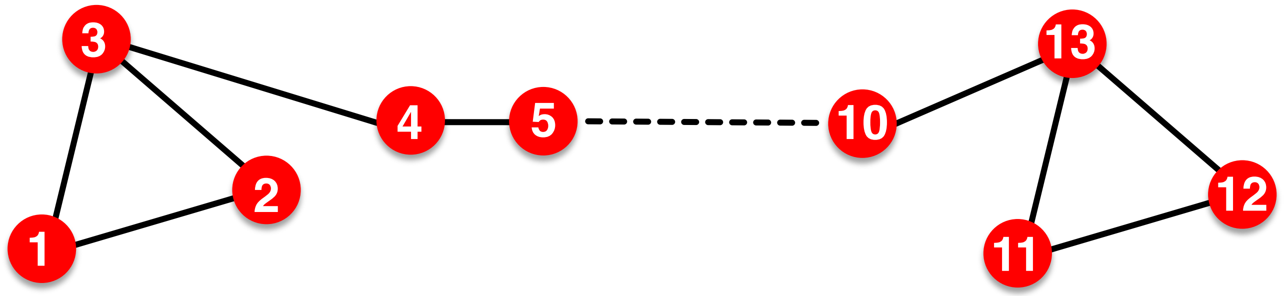
Training set

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**:

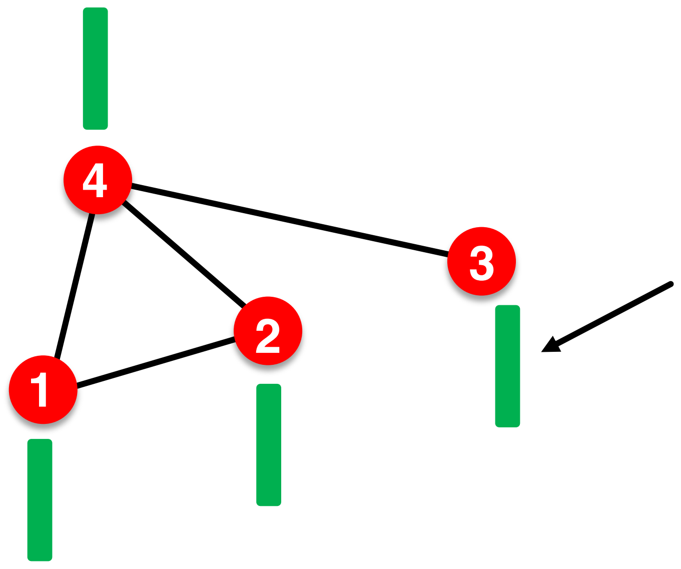


  - Node 1 and 11 are **structurally similar** – part of one triangle, degree 2, …
  - However, they have very **different** embeddings.
    - It’s unlikely that a random walk will reach node 11 from node 1.

- **DeepWalk and node2vec do not capture structural similarity.**

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

Solution to these limitations: Deep Representation Learning and Graph Neural Networks
(To be covered in depth next)