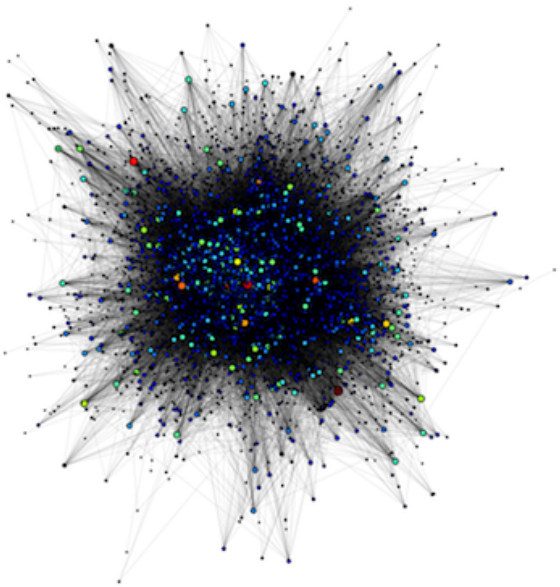
Deep Learning for Graphs

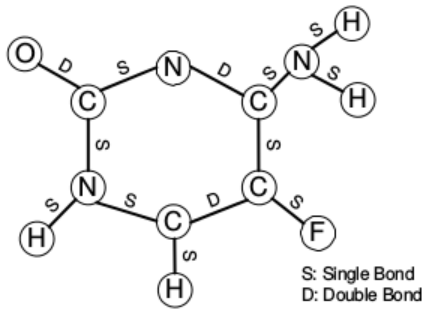
Michalis Vazirgiannis and Giannis Nikolentzos

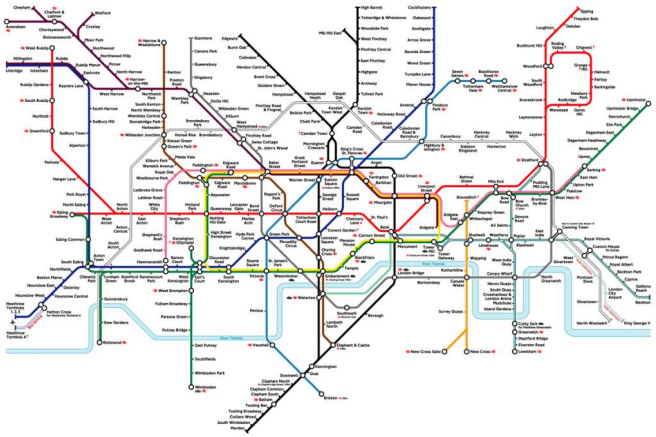
LIX
École Polytechnique

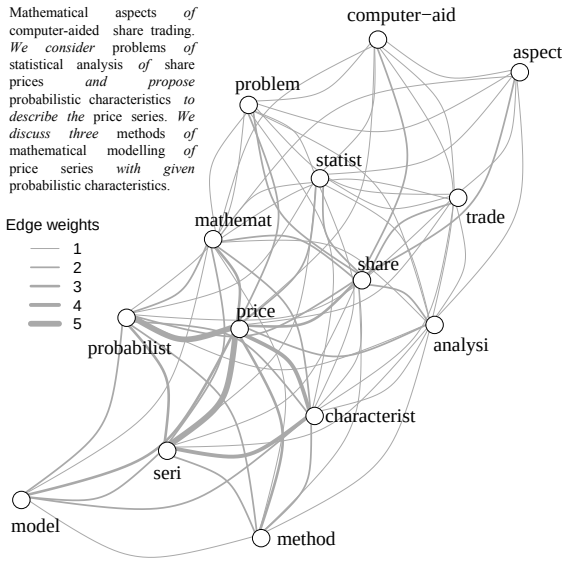
January 8, 2019

Graphs Are Everywhere









A flexible and powerful data structure

Outline

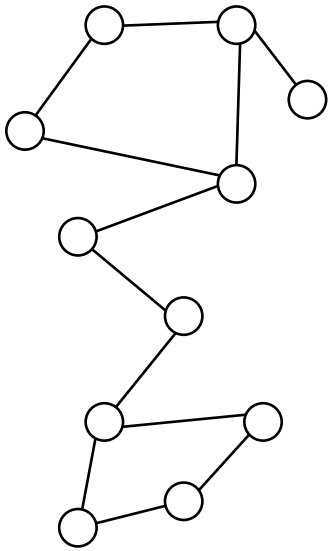
1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

Traditional Node Representation

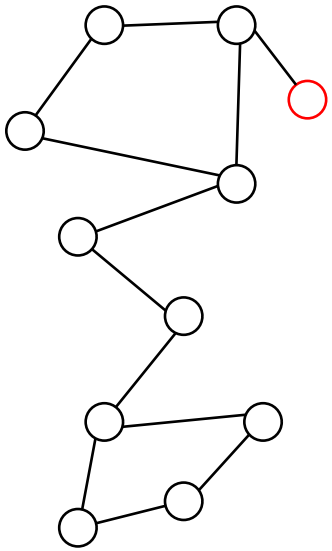
Representation: row of adjacency matrix



$$\rightarrow \quad \begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 0 \end{pmatrix}$$

Traditional Node Representation

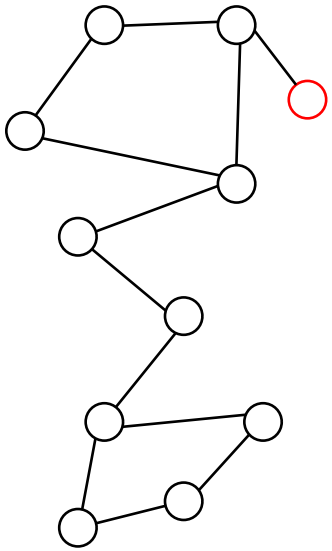
Representation: row of adjacency matrix



$$\rightarrow \quad \begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 0 \end{pmatrix}$$

Traditional Node Representation

Representation: row of adjacency matrix



$$\rightarrow \quad \begin{pmatrix} 0 & 1 & \dots & 0 \\ 1 & 0 & \dots & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & \dots & 0 \end{pmatrix}$$

However, such a representation suffers from:

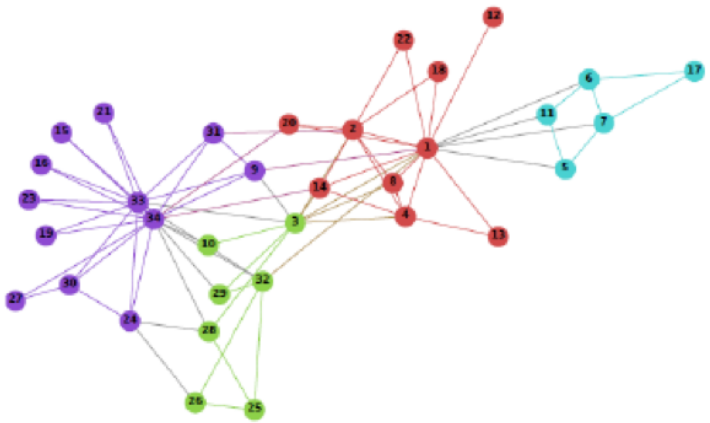
- data sparsity
- high dimensionality

$$\vdots$$

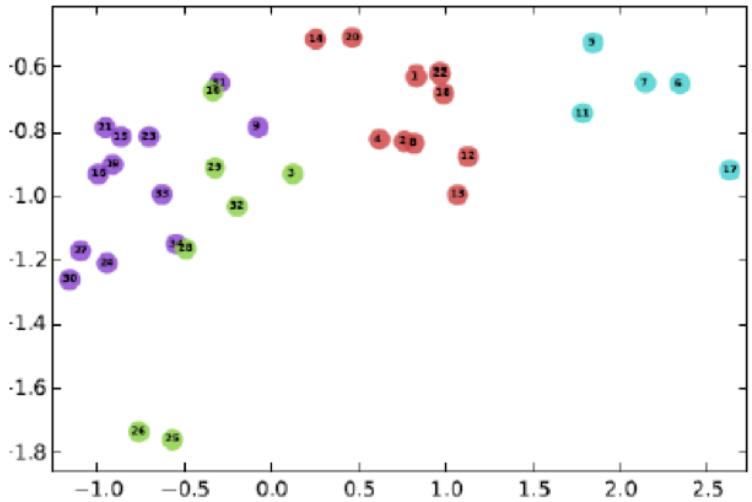
Node Embedding Methods

Map vertices of a graph into a low-dimensional space:

- dimensionality $d \ll |V|$
- similar vertices are embedded close to each other in the low-dimensional space

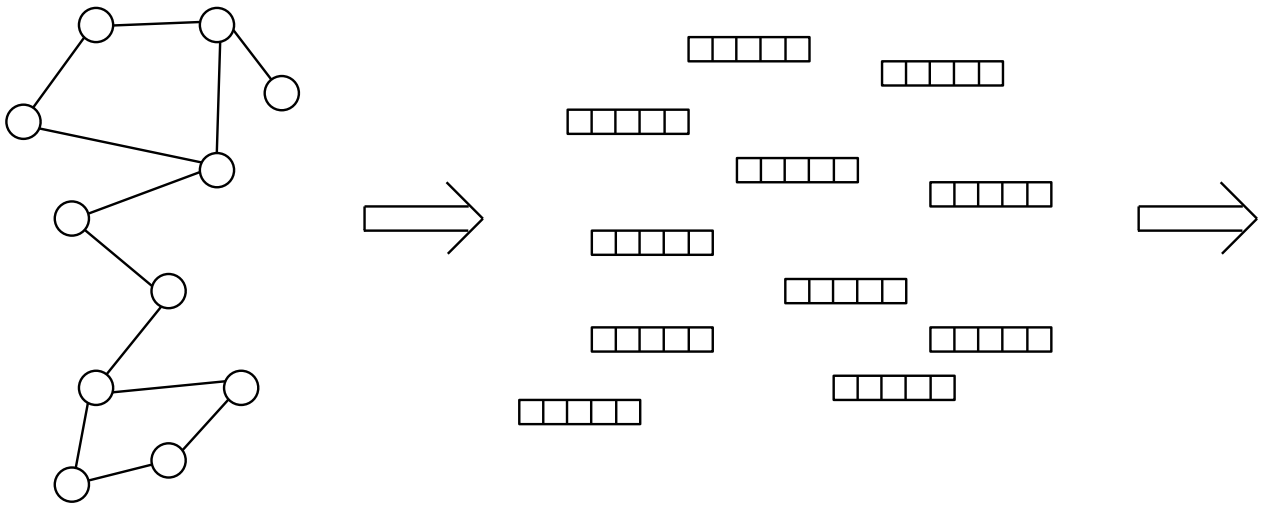


Input



Output

Why Learning Node Representations?



Node Classification
Anomaly Detection
Link Prediction
Clustering
Recommendation
⋮

Examples:

- Recommend friends
- Detect malicious users

Early Methods

- Focused mainly on matrix-factorization approaches (e. g., Laplacian eigenmaps)
- Laplacian eigenmaps projects two nodes i and j close to each other when the weight of the edge between the two nodes A_{ij} is high
- Embeddings are obtained by the following objective function:

$$y^* = \arg\min \sum_{i \neq j} (y_i - y_j)^2 A_{ij} = \arg\min y^T L y$$

  where L is the graph Laplacian

- The solution is obtained by taking the eigenvectors corresponding to the d smallest eigenvalues of the normalized Laplacian matrix

[1] Belkin and Niyogi. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In NIPS'02

Recent Methods

Most methods belong to the following groups:

1. Random walk based methods: employ random walks to capture structural relationships between nodes
2. Edge modeling methods: directly learn node embeddings using structural information from the graph
3. Matrix factorization methods: generate a matrix that represents the relationships between vertices and use matrix factorization to obtain embeddings
4. Deep learning methods: apply deep learning techniques to learn highly non-linear node representations

Node Embedding Methods

Map vertices of a graph into a low-dimensional space:

- dimensionality $d \ll |V|$
- **similar vertices** are embedded close to each other in the low-dimensional space

When two vertices are **similar** to each other?

- -> first-order proxmity
- -> second-order proxmity
- -> third-order proxmity

$\vdots$

Node Embedding Methods

Map vertices of a graph into a low-dimensional space:

- dimensionality $d \ll |V|$
- **similar vertices** are embedded close to each other in the low-dimensional space

When two vertices are **similar** to each other?

$->$ first-order proxmity

$->$ second-order proxmity

$->$ third-order proxmity

$\vdots$

Proximities

Definition (First-order proximity)

The first-order proximity captures the direct neighboring relationships between vertices. If two vertices v and u are linked by an edge, the first-order proximity between them is determined by their edge weight, otherwise is equal to 0.

Definition (Second-order proximity)

The second-order proximity captures the 2-step relations between two vertices v and u. It describes the proximity of the neighborhood structures of v and u, and is thus determined by the number of common neighbors shared by the two vertices.

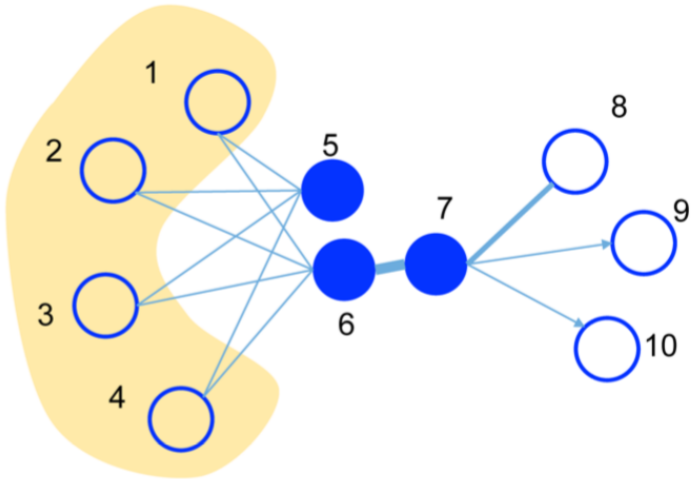
Definition (High-order proximity)

The high-order proximity captures the k-step relations ($k \geq 3$) between two vertices v and u. It is determined by the number of k-step paths from v to u.

Proximities

First-order proximity: observed links in the network

Second-order proximity: shared neighborhood structures

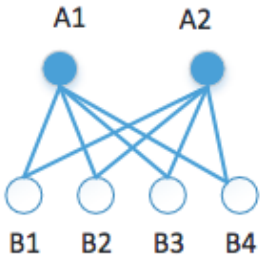


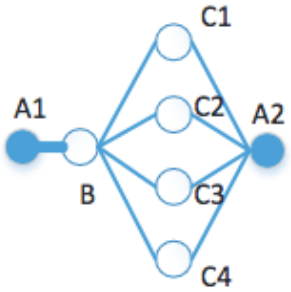
- Vertices 6 and 7 have a high *first-order proximity* since they are connected through a strong tie $\rightarrow$ they should be placed closely in the embedding space

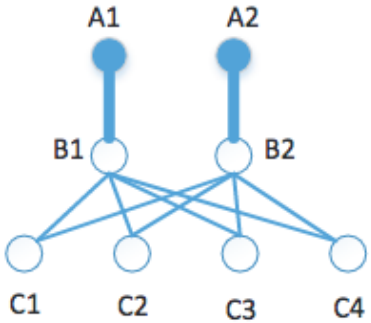
- Vertices 5 and 6 have a high *second-order proximity* since they share similar neighbors $\rightarrow$ they should also be placed closely

Proximities

k-order proximities for $k = 1, \ldots, 4$



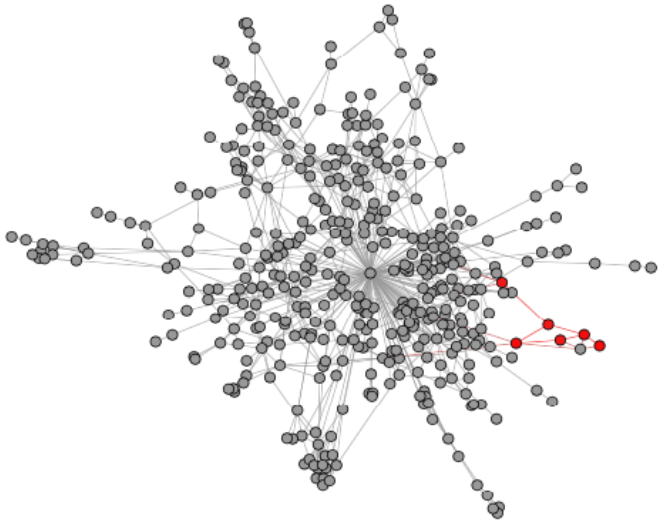




- Second-order and high-order proximities capture similarity between vertices with similar structural roles
- Higher-order proximities capture more global structure

DeepWalk

Inspired by recent advances in language modeling [1]



$\rightarrow$

$v_5 \rightarrow v_8 \rightarrow v_{32} \rightarrow v_{28} \rightarrow v_6 \rightarrow v_{10} \rightarrow v_9$

$v_3 \rightarrow v_5 \rightarrow v_{28} \rightarrow v_8 \rightarrow v_9 \rightarrow v_{10} \rightarrow v_{25}$

$v_{20} \rightarrow v_{10} \rightarrow v_{12} \rightarrow v_6 \rightarrow v_8 \rightarrow v_4 \rightarrow v_5$

$v_{23} \rightarrow v_5 \rightarrow v_{32} \rightarrow v_{10} \rightarrow v_8 \rightarrow v_3 \rightarrow v_1$

$v_4 \rightarrow v_3 \rightarrow v_1 \rightarrow v_5 \rightarrow v_1 \rightarrow v_{12} \rightarrow v_{10}$

$\vdots$

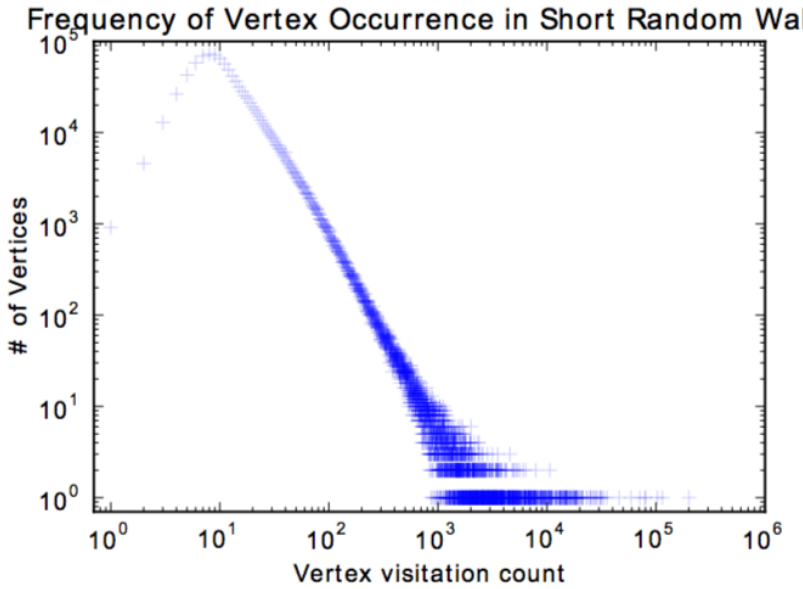
- Simulates a series of short random walks

[1] Mikolov et al. Distributed Representations of Words and Phrases and their Compositionality. In NIPS'13

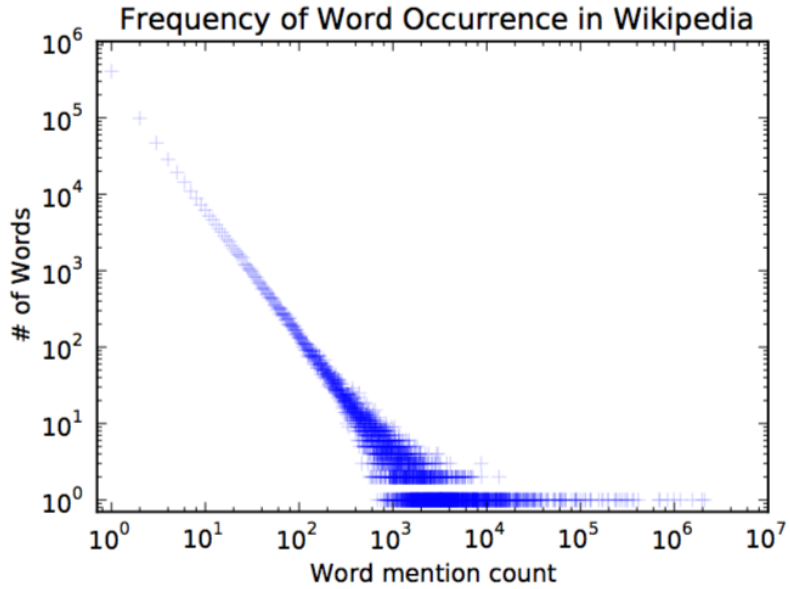
[2] Perozzi et al. DeepWalk: Online Learning of Social Representations. In KDD'14

DeepWalk

Inspired by recent advances in language modeling [1]



(a) YouTube Social Graph



(b) Wikipedia Article Text

- Simulates a series of short random walks
- **Main Idea**: Short random walks = Sentences

[1] Mikolov et al. Distributed Representations of Words and Phrases and their Compositionality. In NIPS'13

[2] Perozzi et al. DeepWalk: Online Learning of Social Representations. In KDD'14

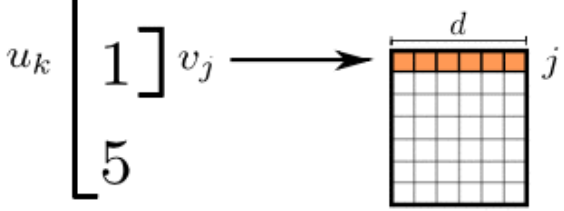
Skipgram

Skipgram is a recently-proposed language model that:

- uses one word to predict the context
- context is composed of words appearing to both the right and left of the given word
- removes the ordering constraint on the problem (i. e. does not take into account the offset of context words from the given word)

In our setting:

$$\mathcal{W}_{v_4} = \begin{matrix} 4 \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix} \begin{bmatrix} 3 \\ 1 \\ 5 \\ 1 \\ \vdots \end{bmatrix} \begin{matrix} \\ \\ \\ \\ \\ \\ \end{matrix}$$



- Slide a window of length $2w + 1$ over the random walk
- Use the representation of central vertex to predict its neighbors

Skipgram

This yields the optimization problem:

$$\underset{f}{\text{minimize}} \quad -\frac{1}{T}\sum_{i=1}^{T} \log P(\{v_{i-w}, \ldots, v_{i+w}\} \setminus v_i | f(v_i))$$

v_i: central vertex
$v_{i-w}, \ldots, v_{i+w}$: neighbors of central vertex
$f(v)$: embedding of vertex v

Skipgram approximates the above conditional probability using the following independence assumption:

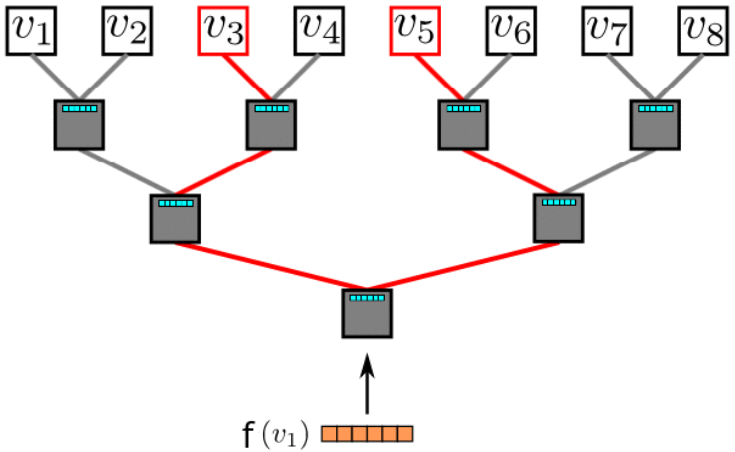
$$\underset{f}{\text{minimize}} \quad -\frac{1}{T}\sum_{i=1}^{T} \sum_{\substack{j=i-w \\ j\neq i}}^{i+w} \log P(v_j | f(v_i))$$

- We can learn such a posterior distribution using several choices of classifiers
- **However**, most of them (e. g., logistic regression) would produce a huge number of labels (i. e. $|V|$ labels)

Hierarchical Softmax

Reduces complexity from $\mathcal{O}(|V|)$ to $\mathcal{O}(\log|V|)$ using a binary tree

- Assigns the vertices to the leaves of a binary tree
- New problem: Maximizing the probability of a specific path in the hierarchy



If the path to vertex v_j is identified by a sequence of tree nodes $(b_0, b_1, \ldots, b_{\lceil \log|V| \rceil})$ then

$$P(v_j|f(v_i)) = \prod_{l=1}^{\lceil \log|V| \rceil} P(b_l|f(v_i))$$

where

$$P(b_l|f(v_i)) = 1/(1+e^{-f(v_i)^\top f'(b_l)}) = \sigma(f(v_i)^\top f'(b_l))$$

and $f'(b_l) \in \mathbb{R}^d$ is the representation assigned to tree node b_l's parent

node2vec

Like DeepWalk, node2vec is also a random walk based method

DeepWalk uses a *rigid* search strategy

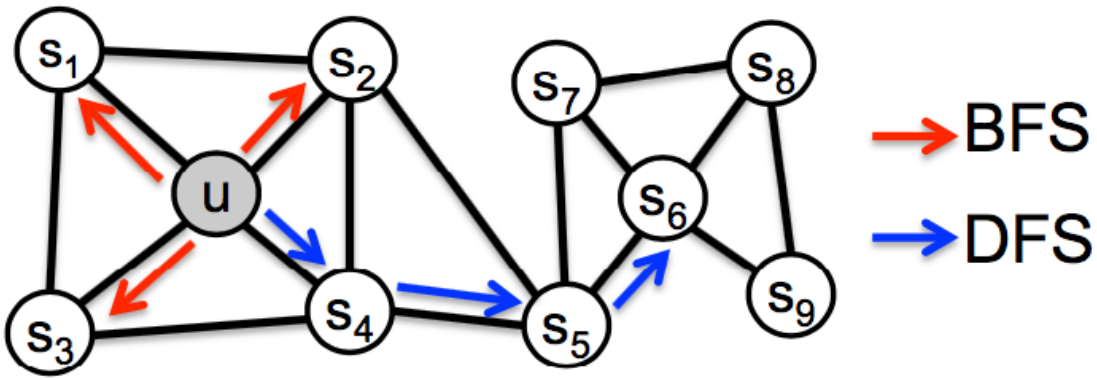
Conversely, node2vec simulates a family of biased random walks which

- explore diverse neighborhoods of a given vertex
- allow it to learn representations that organize vertices based on
  - their network roles
  - the communities they belong to

[1] Grover and Leskovec. node2vec: Scalable Feature Learning for Networks. In KDD'16

Two Extreme Sampling Strategies

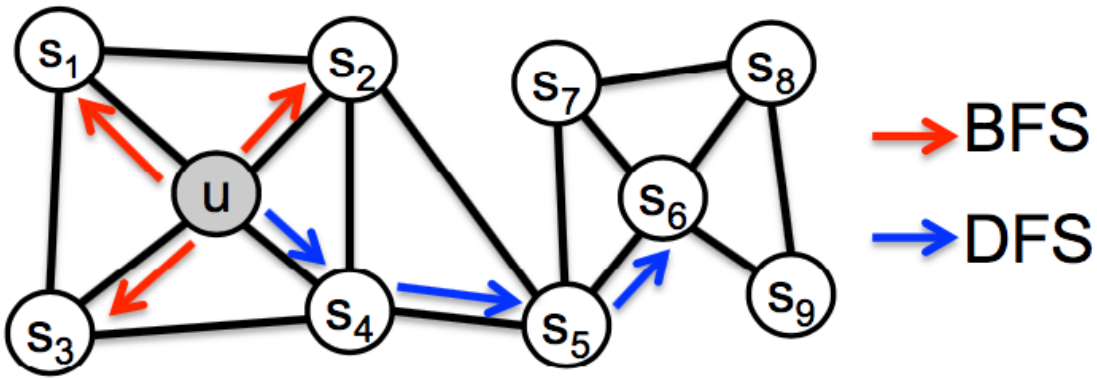
The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space



Goal: Given a source node u, sample its neighborhood $\mathcal{N}(u)$ where $|\mathcal{N}(u)| = k$

Two Extreme Sampling Strategies

The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space

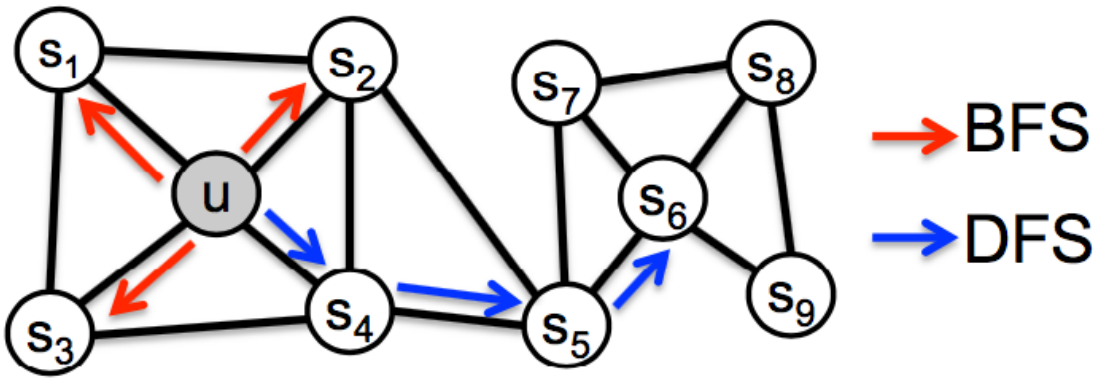


In most applications, we are interested in two kinds of similarities between vertices:

1. homophily: nodes that are highly interconnected and belong to similar communities should be embedded closely together (e. g., s_1 and u)
2. structural equivalence: nodes that have similar structural roles should be embedded closely together (e. g., u and s_6)

Two Extreme Sampling Strategies

The *breadth-first sampling* (BFS) and *depth-first sampling* (DFS) represent extreme scenarios in terms of the search space



BFS and DFS strategies play a key role in producing representations that reflect these two properties:

- The neighborhoods sampled by BFS lead to embeddings that correspond closely to structural equivalence
- The neighborhoods sampled by DFS reflect a macro-view of the neighborhood which is essential in inferring communities based on homophily

Random Walks of node2vec

Given a source node, node2vec simulates a random walk of fixed length l

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_l$$

The i^{th} node in the walk is generated as follows:

$$P(c_i = x | c_{i-1} = v) = \begin{cases} \frac{\pi_{vx}}{Z}, & \text{if } (v, x) \in E \\ 0, & \text{otherwise} \end{cases}$$

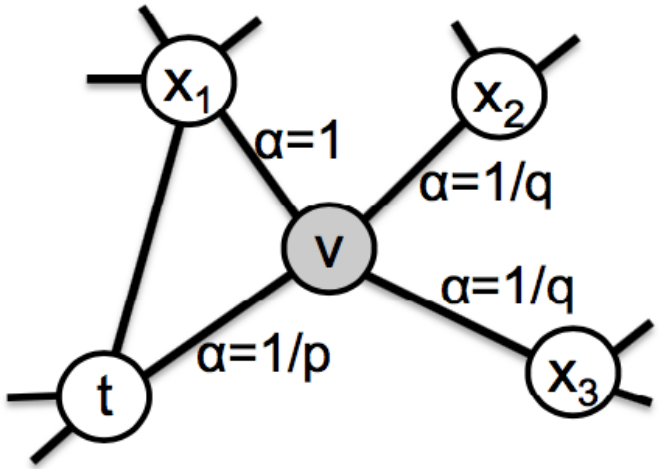
where π_{vx} is the unnormalized transition probability between v and x, and Z is a normalizing factor

To capture both structural equivalence and homophily, node2vec uses a neighborhood sampling strategy which

- is based on a flexible biased random walk procedure
- allows it to smoothly interpolate between BFS and DFS

Random Walks of node2vec

The random walk shown below just traversed edge (t, v) and now resides at node v



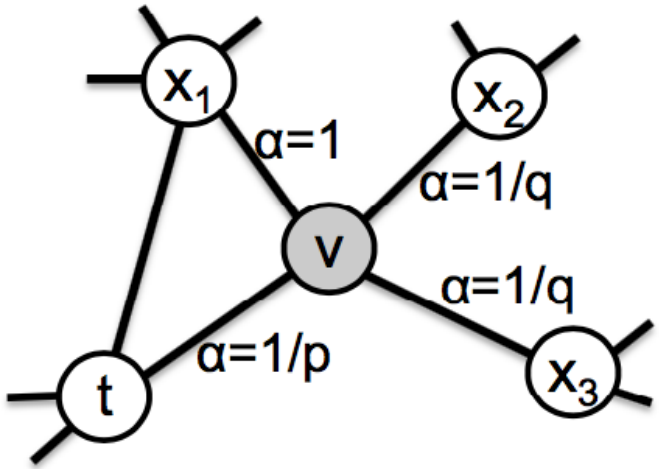
The unnormalized transition probability is $\pi_{vx} = w_{vx}\alpha_{pq}(t, x)$, where:

$$\alpha_{pq}(t,x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

where d_{tx} denotes the shortest path distance between t and x

Random Walks of node2vec

The random walk shown below just traversed edge (t, v) and now resides at node v

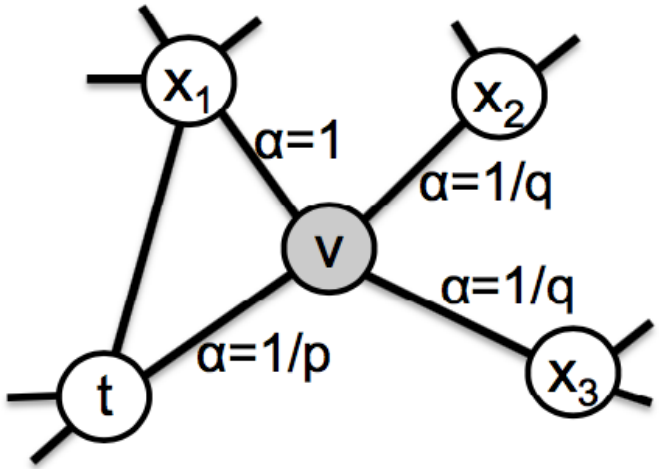


The *return parameter* p controls the likelihood of immediately revisiting a node in the walk

- if p is high, we are less likely to sample an already-visited node in the following two steps
- if p is low, it would keep the walk in the local neighborhood of the starting node

Random Walks of node2vec

The random walk shown below just traversed edge (t, v) and now resides at node v



The *in-out parameter* q allows the search to differentiate between "inward" and "outward" nodes.

- if q is high, the random walk is biased towards nodes close to node t
- if q is low, the walk is more inclined to visit nodes which are further away from the node t

Optimization

After defining the neighborhood $\mathcal{N}(v) \subset V$ of each node v, node2vec uses the Skipgram architecture:

$$\underset{f}{\text{minimize}} \quad -\sum_{v \in V} \log \prod_{u \in \mathcal{N}(v)} P(u|f(v))$$

where conditional likelihood is modelled as a softmax unit parametrized by a dot product of their features:

$$P(u|f(v)) = \frac{e^{f'(u)^\top f(v)}}{\sum_{k=1}^{|V|} e^{f'(v_k)^\top f(v)}}$$

and $f'(u) \in \mathbb{R}^d$ is the representation of node u when considered as context

The objective function thus becomes:

$$\underset{f,f'}{\text{minimize}} \quad -\sum_{v \in V} \left(-\log \sum_{u \in V} e^{f'(u)^\top f(v)} + \sum_{u \in \mathcal{N}(v)} f'(u)^\top f(v) \right)$$

Since learning the above posterior distribution is very expensive, node2vec approximates it using negative sampling

GraRep

GraRep

- constructs transition matrices
- applies matrix factorization to generate node embeddings

k-step Transition Probabilities

Let S be the adjacency matrix of a graph, and D the diagonal degree matrix:

$$D_{ij} = \begin{cases} \sum_p S_{ip} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$$

Then, the 1-step probability transition matrix is defined as:

$$A = D^{-1}S$$

and then the k-step probability transition matrix is defined as:

$$A^k = \underbrace{A \cdots A}_{k}$$

Let $P_k(v_j|v_i)$ denote the probability for a transition from v_i to v_j in exactly k steps. Then,

$$P_k(v_j|v_i) = A^k_{ij}$$

[1] Cao et al. GraRep: Learning Graph Representations with Global Structural Information. In CIKM'15

GraRep

For a given k, the loss function of GraRep is:

$$\underset{f,f'}{\text{minimize}} \quad -\sum_{v_i \in V} \Big(\sum_{v_j \in V} P_k(v_j|v_i) \log \sigma(f'(v_j)^\top f(v_i)) + \lambda \mathbb{E}_{v_c \sim P_k(V)} [\log \sigma(-f'(v_c)^\top f(v_i))] \Big)$$

λ: a hyper-parameter indicating the number of negative samples
$P_k(V)$: distribution over the vertices

Given a specific starting vertex v_i and ending vertex v_j, the local loss over that pair is defined as:

$$L_k(v_i, v_j) = -P_k(v_j|v_i) \log \sigma(f'(v_j)^\top f(v_i)) - \lambda P_k(v_j) \log \sigma(-f'(v_j)^\top f(v_i))$$

and $P_k(v_j)$ can be computed as:

$$P_k(v_j) = \frac{1}{|V|} \sum_{v_l \in V} A_{lj}^k$$

This leads to:

$$L_k(v_i, v_j) = -A_{ij}^k \log \sigma(f'(v_j)^\top f(v_i)) - \frac{\lambda}{|V|} \sum_{v_l \in V} A_{lj}^k \log \sigma(-f'(v_j)^\top f(v_i))$$

By defining $e = f(v_i)^\top f'(v_j)$ and setting $\frac{\partial L_k}{\partial e} = 0$, we get:

$$Y_{ij}^k = f(v_i)^\top f'(v_j) = W_i^k C_j^k = \log\left(\frac{A_{ij}^k}{\sum_{v_l \in V} A_{lj}^k}\right) - \log\left(\frac{\lambda}{|V|}\right)$$

Hence, optimizing the proposed loss essentially involves a matrix factorization problem

GraRep

To reduce noise, GraRep replaces all negative entries in Y^k with 0:

$$X_{ij}^k = \max(Y_{ij}^k, 0)$$

And then decomposes X^k using SVD:

$$X^k = U^k \Sigma^k (V^k)^\top$$

Let X_d^k be a low-rank approximation of X^k (by keeping the top d singular values). Then,

$$X^k \approx X_d^k = U_d^k \Sigma_d^k (V_d^k)^\top = W^k C^k$$

where

$$W^k = U_d^k (\Sigma_d^k)^{\frac{1}{2}} \qquad C^k = (\Sigma_d^k)^{\frac{1}{2}} (V_d^k)^\top$$

rows of W^k: *representations of vertices*
columns of C^k: representations of vertices when considered as context

GraRep

To capture high-order proximities between vertices, GraRep:

- computes the k-step transition probability matrix A^k for each $k = 1, 2, \ldots, K$
- computes each k-step representation
- concatenates all k-step representations

Main disadvantage: by setting K to large values, GraRep fails to efficiently scale to large networks

SDNE

Most real-world networks are very complex

Shallow models

- cannot capture the highly non-linear network structure
- generate sub-optimal node representations

SDNE is a *deep* model which

- has multiple layers of non-linear functions
- preserves the first-order and second-order proximities

[1] Wang et al. Structural Deep Network Embedding. In KDD'16

SDNE

To preserve the second-order proximity, SDNE employs a deep autoencoder

Given an input $\mathbf{x}_i$ (i^{th} row of adjacency matrix), the hidden representations at layers $1, \ldots, k$ are:

$$\mathbf{y}_i^{(1)} = \sigma(\mathbf{W}^{(1)}\mathbf{x}_i + \mathbf{b}^{(1)})$$
$$\mathbf{y}_i^{(k)} = \sigma(\mathbf{W}^{(k)}\mathbf{y}_i^{(k-1)} + \mathbf{b}^{(k)})$$

where σ is a non-linear activation function (e. g., sigmoid function)

After obtaining $\mathbf{y}_i^{(k)}$ (node i's' embedding), we compute the reconstructed input $\hat{\mathbf{x}}_i$ by reversing the above calculation process

The objective function is then:

$$\mathcal{L}_{2nd} = \sum_{i=1}^{n} ||(\hat{\mathbf{x}}_i - \mathbf{x}_i) \odot \mathbf{b}_i||_2^2$$

where $\odot$ is the Hadamard product, $\mathbf{b}_{ij} = 1$ if nodes i and j are not connected by an edge, and $\mathbf{b}_{ij} > 1$ otherwise

Vertices that have similar neighborhoods are mapped close to each other in the embedding space

SDNE

To capture the first-order proximity, SDNE borrows the idea of Laplacian Eigenmaps:

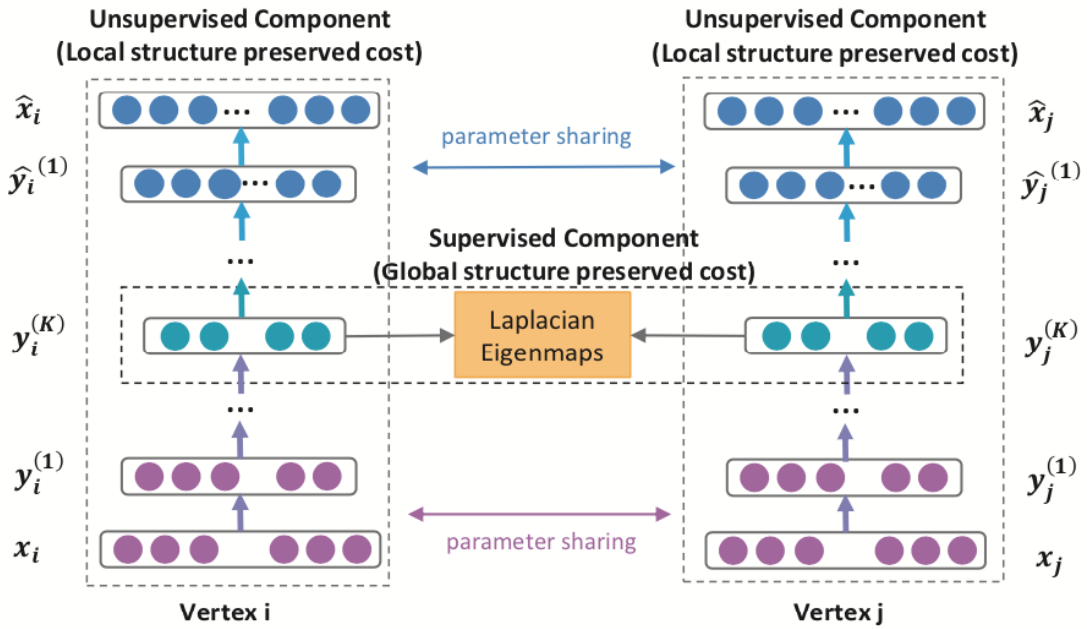
$$\mathcal{L}_{1st} = \sum_{i=1}^{n}\sum_{j=1}^{n} \mathbf{x}_{ij} ||\mathbf{y}_i^{(k)} - \mathbf{y}_j^{(k)}||_2^2$$

Vertices linked by edges with high weights are thus mapped close to each other

SDNE then jointly minimizes the following objective function:

$$\mathcal{L} = \mathcal{L}_{2nd} + \alpha\mathcal{L}_{1st} + \nu\mathcal{L}_{reg}$$

where $\mathcal{L}_{reg}$ is an l_2-norm regularizer term to prevent overfitting



LINE

LINE employs an objective function that explicitly uses structural information from the graph to learn node representations

Specifically, LINE

- preserves both the first-order and second-order proximities
- trains two models separately
- concatenates the two learned embeddings for each vertex

[1] Tang et al. LINE: Large-scale Information Network Embedding. In WWW'15

LINE with First-order Proximity

To model the first-order proximity, for each undirected edge (v_i, v_j), define the joint probability between v_i and v_j as follows:

$$P_1(v_i, v_j) = \frac{1}{1 + e^{-f(v_i)^\top f(v_j)}}$$

where $f(v_i) \in \mathbb{R}^d$ is the low-dimensional vector representation of vertex v_i

The empirical probability can be defined as:

$$\hat{P}_1(v_i, v_j) = \frac{w_{ij}}{W}$$

w_{ij}: weight of the edge between v_i, v_j
W: sum of weights of all edges

LINE minimizes the KL-divergence of the two probability distributions:

$$\underset{f}{\text{minimize}} \quad - \sum_{(v_i, v_j) \in E} w_{ij} \log P_1(v_i, v_j)$$

LINE with Second-order Proximity

To model the second-order proximity, for each edge (v_i, v_j), LINE defines the probability of context v_j generated by vertex v_i:

$$P_2(v_j|v_i) = \frac{e^{f'(v_j)^\top f(v_i)}}{\sum_{k=1}^{|V|} e^{f'(v_k)^\top f(v_i)}}$$

$f(v_i)$: representation of v_i when treated as a vertex
$f'(v_i)$: representation of v_i when treated as context

The empirical probability can be defined as:

$$\hat{P}_2(v_j|v_i) = \frac{w_{ij}}{d_i}$$

d_i: out-degree of v_i

LINE minimizes the KL-divergence of the two probability distributions:

$$\underset{f,f'}{\text{minimize}} \quad -\sum_{(v_i,v_j)\in E} w_{ij} \log P_2(v_j|v_i)$$

LINE with Second-order Proximity

Optimizing the objective of the second-order proximity is computationally very *expensive*

Instead, use negative sampling: for each edge, sample multiple negative edges according to some noisy distribution

Every $\log P_2(v_j|v_i)$ term in the objective is replaced with:

$$\log \sigma(f'(v_j)^\top f(v_i)) + \sum_{k=1}^{K} \mathbb{E}_{v_k \sim P_n(v)}[\log \sigma(-f'(v_k)^\top f(v_i))]$$

where $\sigma = 1/(1 + e^{-x})$ is the sigmoid function and K the number of negative edges

Experimental Evaluation

Experimental comparison conducted in [1]

Compared algorithms:

- DeepWalk
- GraRep
- SDNE
- LINE
- Laplacian Eigenmaps (LE)

[1] Wang et al. Structural Deep Network Embedding. In KDD'16

Datasets

Five datasets:

- three social networks
- one citation network
- one language network

Dataset	#(V)	#(E)
BLOGCATALOG	10312	667966
FLICKR	80513	11799764
YOUTUBE	1138499	5980886
ARXIV GR-QC	5242	28980
20-NEWSGROUP	1720	Full-connected

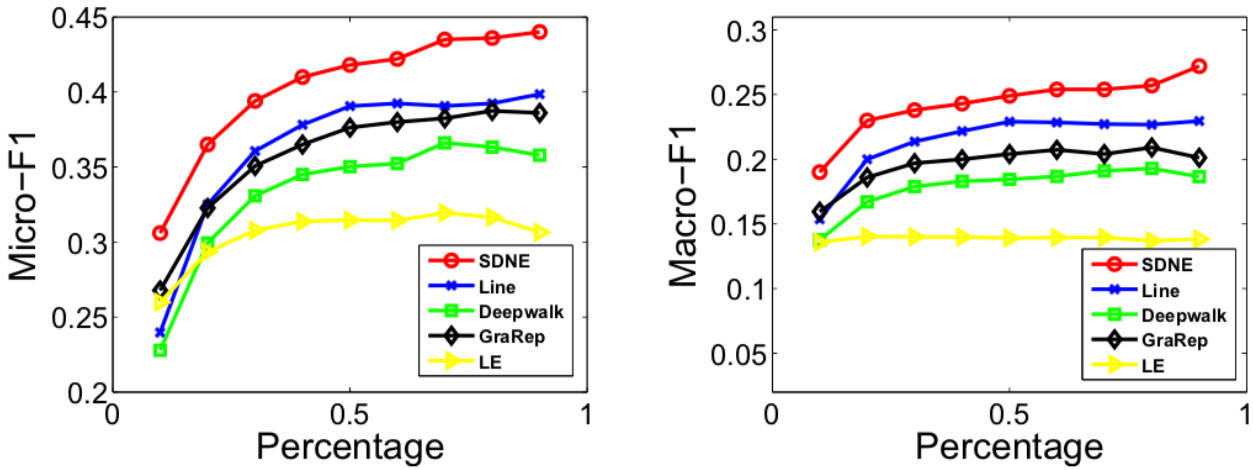
Three real-world applications

- node classification
- link prediction
- visualization

Node Classification

Vertex representations generated from node embedding methods and given as input to a logistic regression classifier to predict a set of labels for each vertex

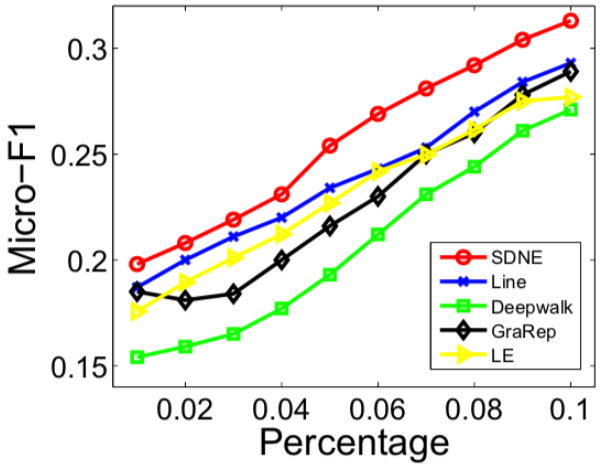
BLOGCATALOG

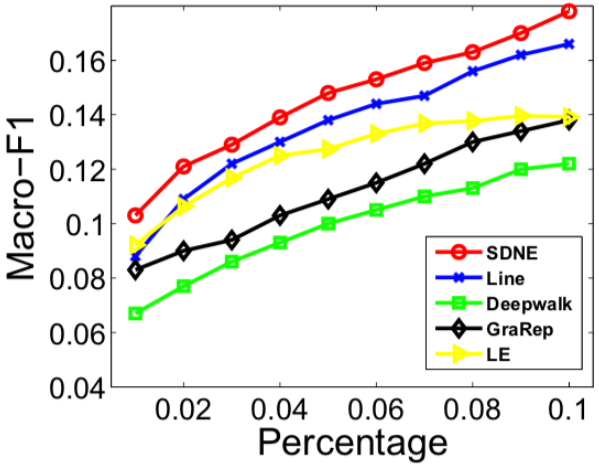


For BLOGCATALOG, the training/test ratio is increased from 10% to 90%

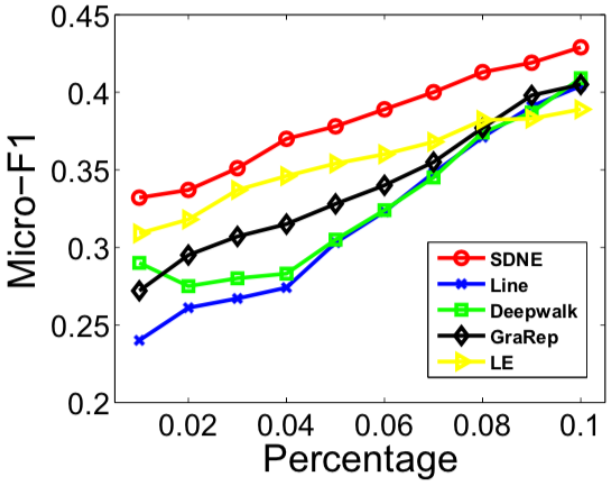
Node Classification

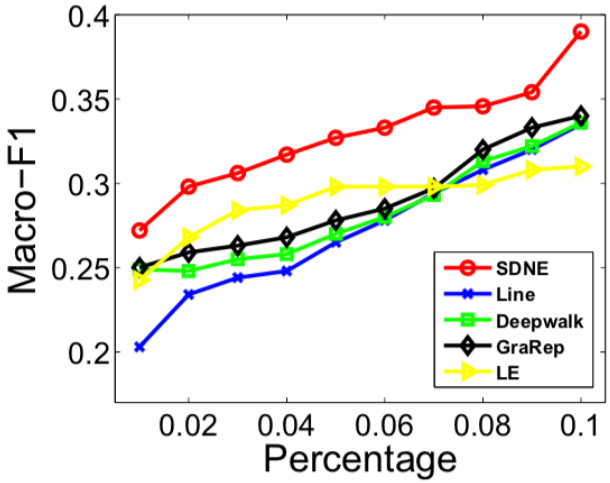
FLICKR





YOUTUBE



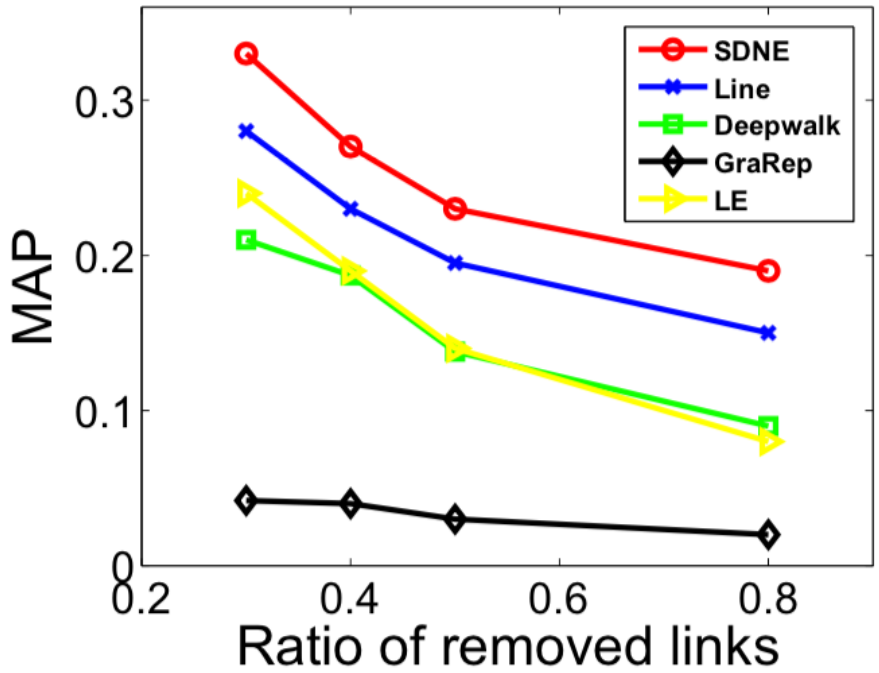


For FLICKR and YOUTUBE, the training/test ratio is increased from 1% to 10%

Link Prediction

Followed procedure:

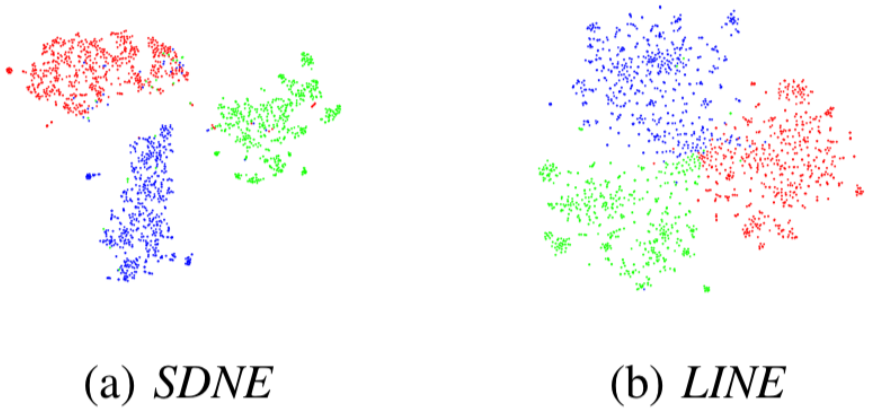
- Remove a portion of ARXIV GR-QC's edges
- Use the emerging network to learn node embeddings
- Predict missing links



Visualization

Visualization of 20-NEWSGROUP

- Each point indicates one document
- Color of a point indicates the category of the document



(a) *SDNE*

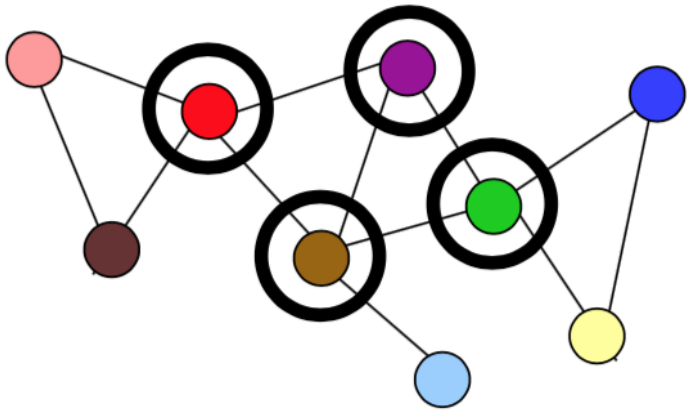
(b) *LINE*

(c) *DeepWalk*

(d) *GraRep*

Structural Identity

- Nodes in networks have specific roles
  - e. g., individuals, web pages, proteins, etc
- Structural identity
  - identification of nodes based on network structure (no other attribute)
  - often related to role played by node
- Automorphism: strong structural equivalence



Red, Green: structurally identical
Purple, Brown: structurally similar

struc2vec

- Learns node representations based on structural identity
  - structurally similar nodes close in space

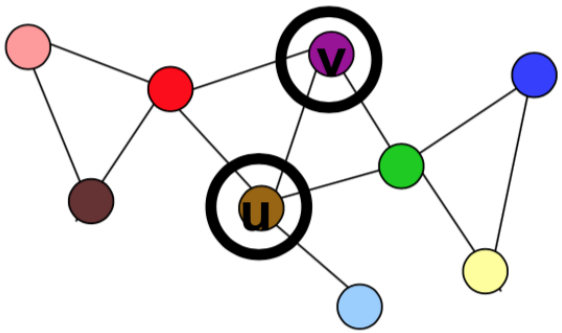
Key ideas:

- Structural similarity does not depend on hop distance
  - neighbor nodes can be different, far away nodes can be similar
- Structural identity as a hierarchical concept
  - depth of similarity varies
- Flexible four step procedure
  - operational aspect of steps are flexible

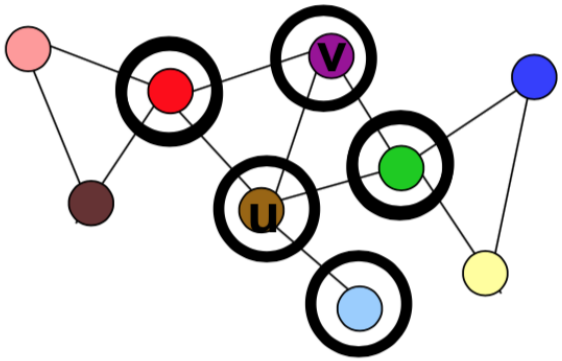
[1] Ribeiro et al. struc2vec: Learning Node Representations from Structural Identity. In KDD'17

Step 1: Structural Similarity

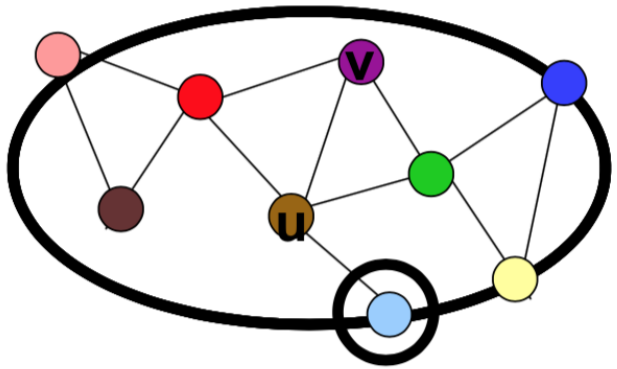
- Hierarchical measure for structural similarity between two nodes
- $R_k(v)$: set of nodes at distance k from v (ring)
- $s(S)$: ordered degree sequence of set S



$s(R_0(u)) = 4$
$s(R_0(v)) = 3$



$s(R_1(u)) = 1, 3, 4, 4$
$s(R_1(v)) = 4, 4, 4$



$s(R_2(u)) = 2, 2, 2, 2$
$s(R_2(v)) = 1, 2, 2, 2, 2$

Step 1: Structural Similarity

- $g(D_1, D_2)$: distance between two ordered sequences
  - cost of pairwise alignment: $\max(a,b)/\min(a,b) - 1$
  - optimal alignment by Dynamic Time Warping in our framework

$s(R_0(u)) = 4$ $\quad$ $s(R_1(u)) = 1, 3, 4, 4$ $\quad$ $s(R_2(u)) = 2, 2, 2, 2$

$s(R_0(v)) = 3$ $\quad$ $s(R_1(v)) = 4, 4, 4$ $\quad$ $s(R_2(v)) = 1, 2, 2, 2, 2$

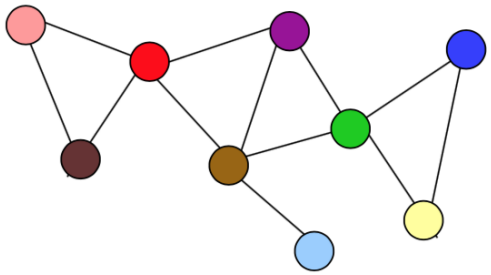
$g(\cdot, \cdot) = 0.33$ $\quad$ $g(\cdot, \cdot) = 3.33$ $\quad$ $g(\cdot, \cdot) = 1$

- $f_k(v, u)$: structural distance between nodes v and u considering first k rings
  - $f_k(v, u) = f_{k-1}(v, u) + g(s(R_k(v)), s(R_k(u)))$

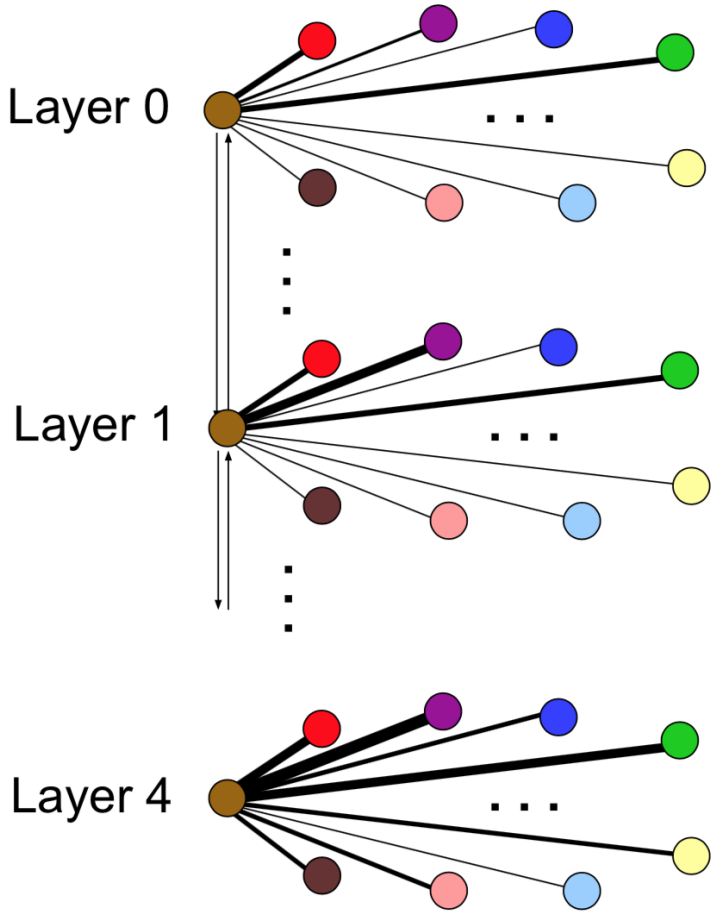
$f_0(v, u) = 0.33$ $\quad$ $f_1(v, u) = 3.66$ $\quad$ $f_2(v, u) = 4.66$

Step 2: Multi-layer graph

- Encodes structural similarity between all node pairs



- Each layer is a weighted complete graph
  - corresponds to similarity hierarchies
- Edge weights in layer k
  - $w_k(v, u) = e^{-f_k(v,u)}$
- Connect corresponding nodes in adjacent layers

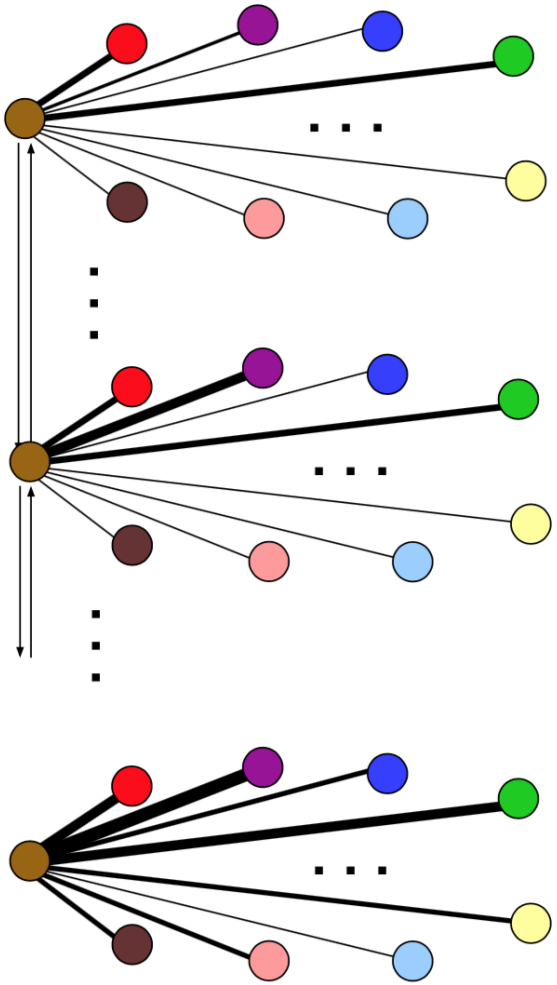


Step 3: Generate Context

- Context generated by biased random walk
  - walking on multi-layer graph
- Walk in current layer with probability p
  - choose neighbor according to edge weight
  - RW prefers more similar nodes
- Change layer with probability $1 - p$
  - choose up/down according to edge weight
  - RW prefers layer with less similar neighbors

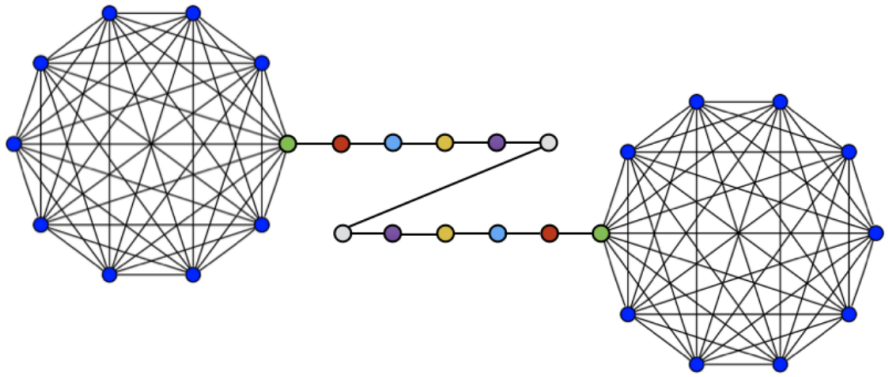
Step 3: Learn Representation

- For each node, generate set of independent and relative short random walks
  - context for node $\rightarrow$ sentences of a language

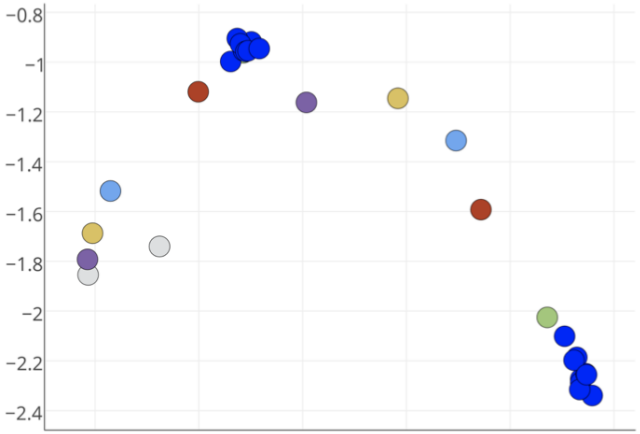


- Train a neural network to learn latent representation for nodes
  - maximize probability of nodes within context
  - Skip-gram (Hierarchical Softmax) adopted

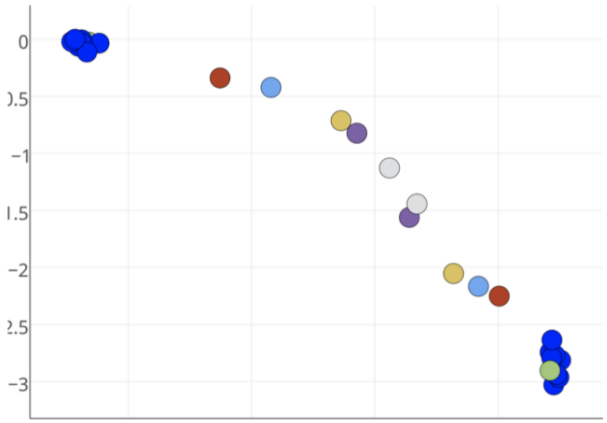
Barbell Network



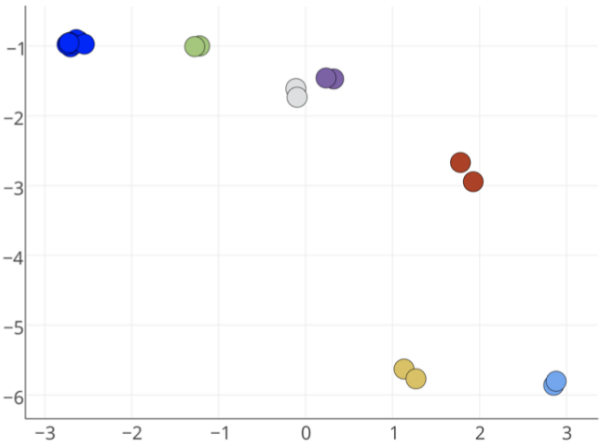
(a) Barbell Graph B(10, 10)



(c) *DeepWalk*



(d) *node2vec*



(e) *struc2vec*

- struc2vec embeds isomorphic nodes very close to each other in space

Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

Planetoid

Planetoid

- assumes node attributed graphs (e. g., a feature vector is associated with each vertex)
- takes into account both the class labels and the graph structure to learn node embeddings
- minimizes the following loss function: $\mathcal{L} = \mathcal{L}_s + \lambda \mathcal{L}_u$

  $\mathcal{L}_s$: a supervised loss of predicting the labels

  $\mathcal{L}_u$: an unsupervised loss of predicting the graph context

[1] Yang et al. Revisiting Semi-Supervised Learning with Graph Embeddings. In ICML'16

Planetoid

With regards to its unsupervised part, Planetoid uses a Skipgram-like architecture: samples triplets (i, c, γ) from a distribution where

i, c: instance and context

$\gamma = +1 \rightarrow (i, c)$ is a positive pair

$\gamma = -1 \rightarrow (i, c)$ is a negative pair

The unsupervised loss is then defined as:

$$\mathcal{L}_u = -\mathbb{E}_{(i,c,\gamma)} \log \sigma(\gamma \mathbf{w}_c^T f(i))$$

Two parameters:

1. $r_1 \in (0, 1)$ controls the ratio of positive and negative samples
2. $r_2 \in (0, 1)$ controls the ratio of two types of context:
   - based on the graph structure with probability r_2
   - based on class labels with probability $1 - r_2$

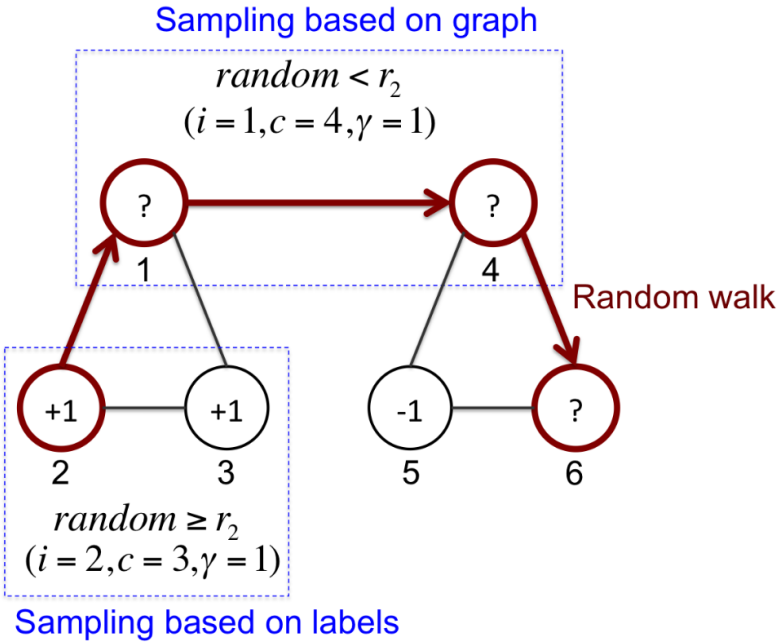
Planetoid

Context based on graph structure:

- samples a random walk S
- with probability r_1, samples a positive pair (i, c) from the set $\{(S_j, S_k) : |j - k| < d\}$
  d: the window size.
- with probability $(1 - r_1)$ corrupts context c to sample a negative pair

Context based on class labels:

- with probability r_1, samples a positive pair (i, c) (i. e. i, c belong to the same class)
- with probability $(1 - r_1)$ samples a negative pair (i, c) (i. e. i, c belong to different classes)



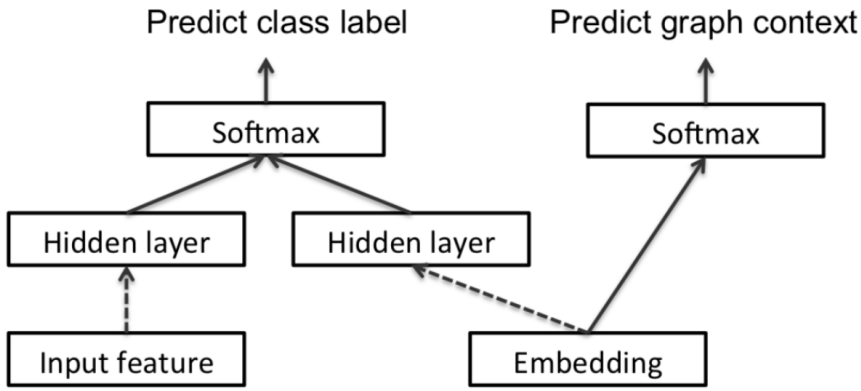
Planetoid

Given a vertex v, Planetoid

- applies k layers on the feature vector $\mathbf{x}_v$ of node v to obtain $h^k(\mathbf{x}_v)$
- applies l layers on the embedding $f(v)$ to obtain $h^l(f(v))$
- concatenates the output and feeds them to a softmax layer

The probability of predicting the label y is then:

$$p(y|\mathbf{x}_v, f(v)) = \frac{e^{[h^k(\mathbf{x}_v), h^l(f(v))]^T \mathbf{w}_y}}{\sum_{y'} e^{[h^k(\mathbf{x}_v), h^l(f(v))]^T \mathbf{w}_{y'}}}$$



The loss function combines the unsupervised with the supervised part and is defined as:

$$\mathcal{L} = -\frac{1}{|V|} \sum_{v \in V} \log p(y_v|\mathbf{x}_v, f(v)) - \lambda \mathbb{E}_{(i,c,\gamma)} \log \sigma(\gamma \mathbf{w}_c^T f(v))$$

where λ is a weighting factor

GCN

Given the adjacency matrix $\mathbf{A}$ of a graph, GCN first computes:

$$\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}} \, \tilde{\mathbf{A}} \, \tilde{\mathbf{D}}^{-\frac{1}{2}}$$

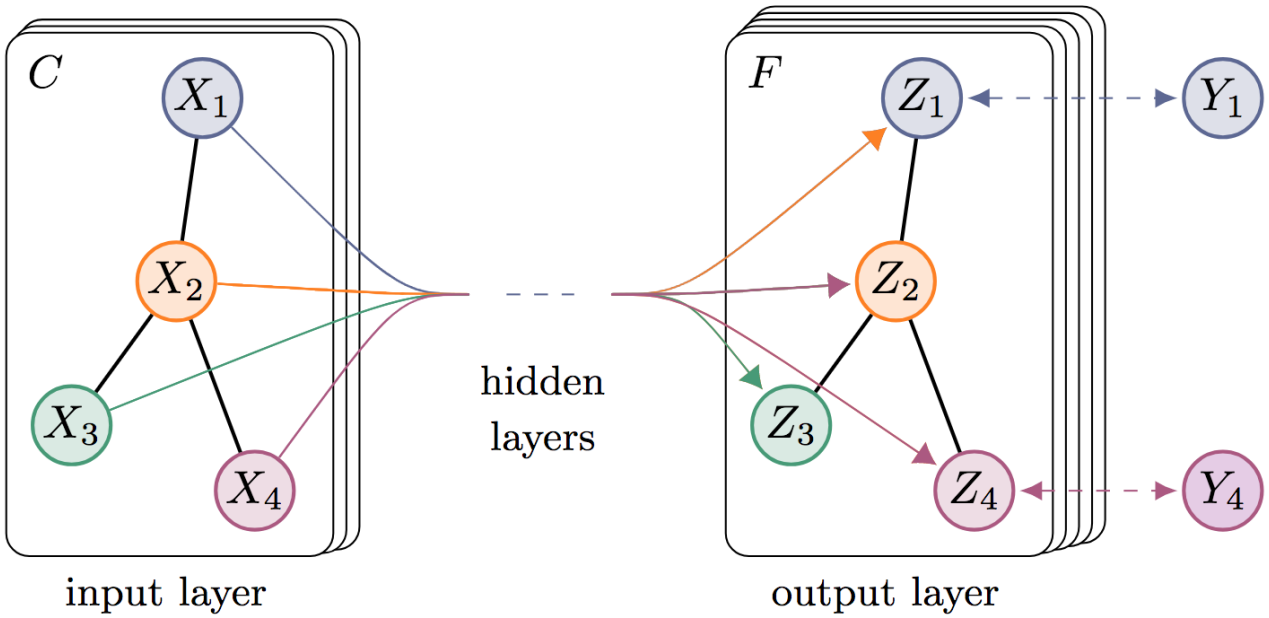
where
$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$
$\tilde{\mathbf{D}}$: a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$

Then, the output of the model is:

$$\mathbf{Z} = softmax(\hat{\mathbf{A}} \; ReLU(\hat{\mathbf{A}} \, \mathbf{X} \, \mathbf{W}^0) \, \mathbf{W}^1)$$

where
$\mathbf{X}$: matrix whose rows contain the attributes of the nodes
$\mathbf{W}^0, \mathbf{W}^1$: trainable weight matrices

[1] Kipf and Welling. Semi-supervised Classification with Graph Convolutional Networks. In ICLR'17



To learn node embeddings, GCN minimizes the following loss function:

$$\mathcal{L} = -\sum_{i \in I} \sum_{j=1}^{|\mathcal{C}|} \mathbf{Y}_{ij} \log \hat{\mathbf{Y}}_{ij}$$

I: indices of the nodes of the training set
$\mathcal{C}$: set of class labels

Experimental Evaluation

Experimental comparison conducted in [1]

Compared algorithms:

- DeepWalk
- ICA [2]
- Planetoid
- GCN

Task: node classification

[1] Kipf and Welling. Semi-supervised Classification with Graph Convolutional Networks. In ICLR'17

[2] Lu and Getoor. Link-based classification. In ICML'03

Datasets

Dataset	Type	Nodes	Edges	Classes	Features	Label rate
Citeseer	Citation network	3,327	4,732	6	3,703	0.036
Cora	Citation network	2,708	5,429	7	1,433	0.052
Pubmed	Citation network	19,717	44,338	3	500	0.003
NELL	Knowledge graph	65,755	266,144	210	5,414	0.001

Label rate: number of labeled nodes that are used for training divided by the total number of nodes

Citation network datasets:

- nodes are documents and edges are citation links
- each node has an attribute (the bag-of-words representation of its abstract)

NELL is a bipartite graph dataset extracted from a knowledge graph

Classification accuracies of the 4 methods

Method	Citeseer	Cora	Pubmed	NELL
DeepWalk	43.2	67.2	65.3	58.1
ICA	69.1	75.1	73.9	23.1
Planetoid	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)

Observation: DeepWalk $\rightarrow$ unsupervised learning of embeddings

$\hookrightarrow$ fails to compete against the supervised approaches

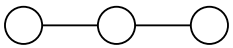
Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

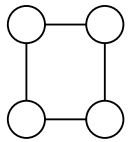
Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

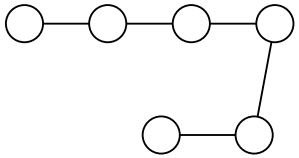
Graph Classification



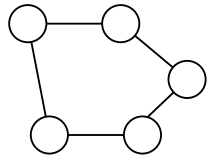
class -1



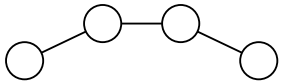
class 1



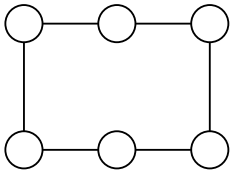
class -1



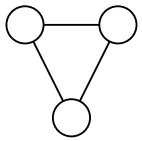
???



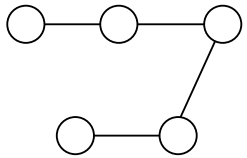
class -1



class 1



class 1



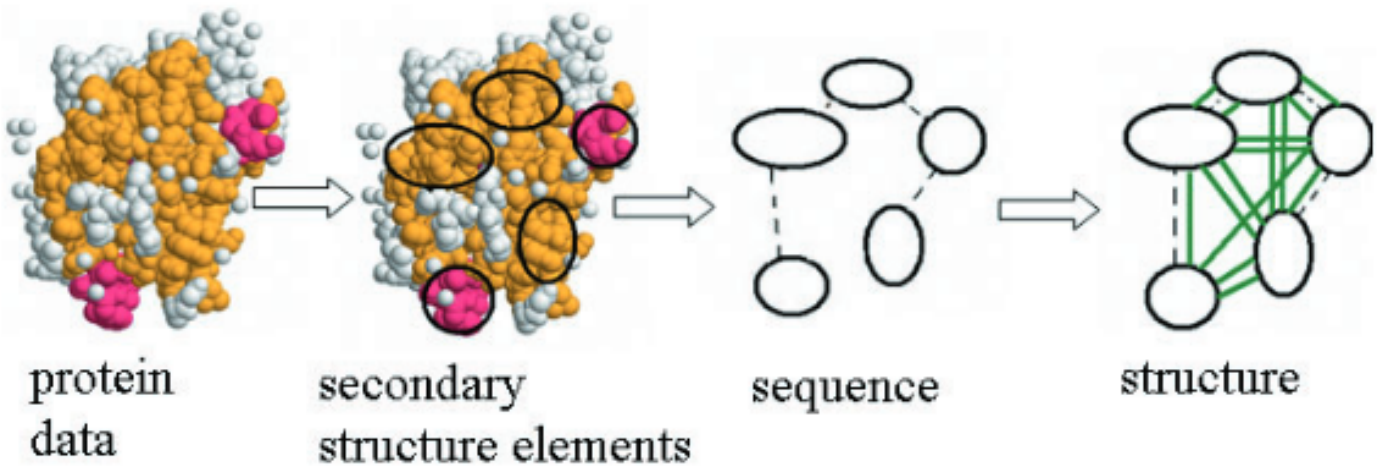
???

- Input data $x \in \mathcal{X}$
- Output $y \in \{-1, 1\}$
- Training set $\mathcal{S} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$
- Goal: estimate a function $f : \mathcal{X} \to \mathbb{R}$ to predict y from f(x)

Motivation - Protein Function Prediction

For each protein, create a graph that contains information about its

- structure
- sequence
- chemical properties



Perform **graph classification** to predict the function of proteins

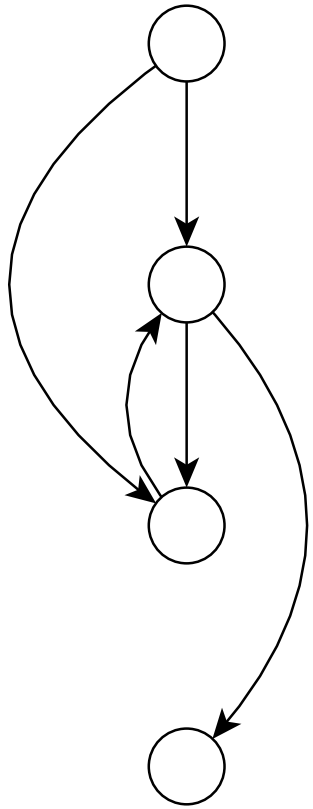
[1] Borgwardt et al. Protein function prediction via graph kernels. Bioinformatics 21, 2005

Motivation - Malware Detection

Given a computer program, create its control flow graph

```
            processed_pages.append(processed_page)
            visited += 1
            links = extract_links(html_code)
            for link in links:
                if link not in visited_links:
                    links_to_visit.append(link)

        return create_vocabulary(processed_pages)


def parse_page(html_code):
    punct = re.compile(r'([^A-Za-z0-9])')
    soup = BeautifulSoup(html_code, 'html.parser')
    text = soup.get_text()
    processed_text = punct.sub(" ", text)
    tokens = processed_text.split()
    tokens = [token.lower() for token in tokens]
    return tokens


def create_vocabulary(processed_pages):
    vocabulary = {}
    for processed_page in processed_pages:
        for token in processed_page:
            if token in vocabulary:
                vocabulary[token] += 1
            else:
                vocabulary[token] = 1

    return vocabulary
```

→



Perform **graph classification** to predict if there is malicious code inside the program or not

[1] Gascon et al. Structural Detection of Android Malware using Embedded Call Graphs. In AISec'13

Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

Message Passing Neural Networks

Consist of two steps:

1. a message passing step
2. a readout step

Step 1: The message passing phase runs for T time steps and updates the representation of each vertex h_v^t based on its previous representation and the representations of its neighbors:

$$m_v^{t+1} = \sum_{u \in \mathcal{N}(v)} M_t(h_v^t, h_u^t, e_{vu})$$

$$h_v^{t+1} = U_t(h_v^t, m_v^{t+1})$$

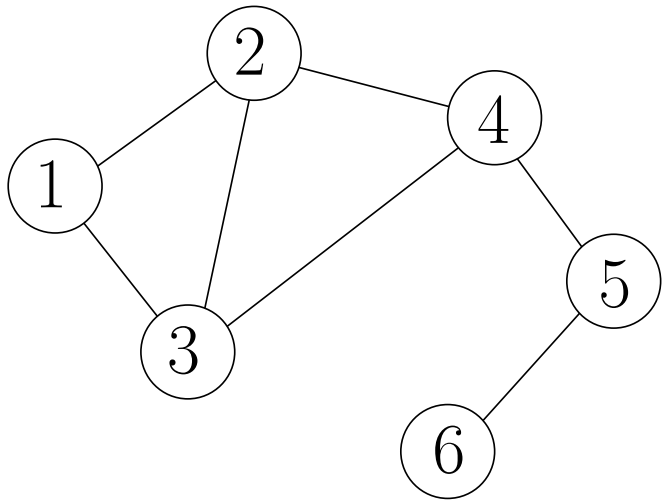
where $\mathcal{N}(v)$ is the set of neighbors of v and M_t, U_t are message functions and vertex update functions respectively

Step 2: The readout step computes a feature vector for the whole graph using some readout function R:

$$\hat{y} = R(\{h_v^T | v \in G\})$$

Message Passing Phase Example

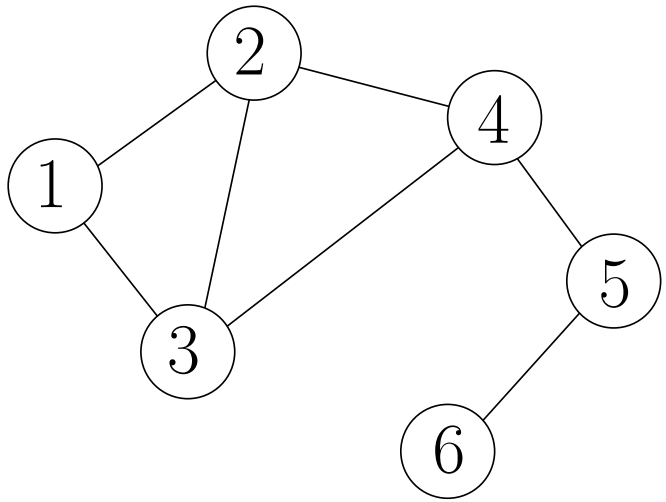
$$
\begin{aligned}
h_1^{t+1} &= W_0^t h_1^t + W_1^t h_2^t + W_1^t h_3^t \\
h_2^{t+1} &= W_0^t h_2^t + W_1^t h_1^t + W_1^t h_3^t + W_1^t h_4^t \\
h_3^{t+1} &= W_0^t h_3^t + W_1^t h_1^t + W_1^t h_2^t + W_1^t h_4^t \\
h_4^{t+1} &= W_0^t h_4^t + W_1^t h_2^t + W_1^t h_3^t + W_1^t h_5^t \\
h_5^{t+1} &= W_0^t h_5^t + W_1^t h_4^t + W_1^t h_6^t \\
h_6^{t+1} &= W_0^t h_6^t + W_1^t h_5^t
\end{aligned}
$$



Readout Step Example

Output of message passing phase:

$$\{h_1^{T_{max}}, h_2^{T_{max}}, h_3^{T_{max}}, h_4^{T_{max}}, h_5^{T_{max}}, h_6^{T_{max}}\}$$



Graph representation:

$$z_G = \frac{1}{6}\left(h_1^{T_{max}} + h_2^{T_{max}} + h_3^{T_{max}} + h_4^{T_{max}} + h_5^{T_{max}} + h_6^{T_{max}}\right)$$

Step 1: The network updates the states of the nodes as follows:

$$\mathbf{m}_v^{t+1} = \mathbf{h}_v^t + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^t$$

$$\mathbf{h}_v^{t+1} = \sigma(\mathbf{H}_t^{deg(v)} \mathbf{m}_v^{t+1})$$

$deg(v)$: degree of vertex v
$\mathbf{H}_t^{deg(v)}$: a learned matrix for each time step t and vertex degree $deg(v)$

Step 2: The network computes the graph representation as:

$$\mathbf{h}_G = \sum_{t=0}^{T} \sum_{v \in V} softmax(\mathbf{W}_t \mathbf{h}_v^t)$$

The output $\mathbf{h}_G$ is then fed to a fully-connected neural network

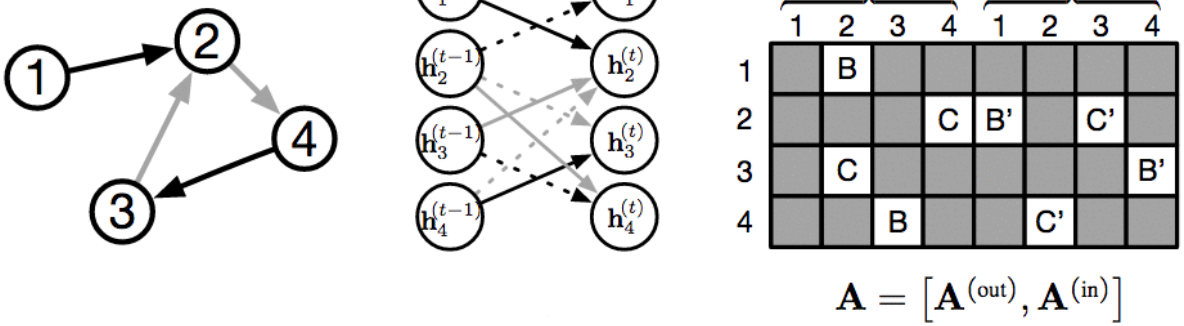
[1] Duvenaud et al. Convolutional Networks on Graphs for Learning Molecular Fingerprints. In NIPS'15

GG-NN

Step 1: Hidden states are initially set equal to node attributes: $\mathbf{h}_v^0 = \mathbf{x}_v$

Then, at each iteration

- the network transfers information between each vertex and its neighbors: $\mathbf{a}_v^t = \mathbf{A}_{v:}^T \left[\mathbf{h}_1^{t-1}, \ldots, \mathbf{h}_{|V|}^{t-1}\right]$



- then the following GRU-like updates take place:

$$\mathbf{z}_v^t = \sigma(\mathbf{W}^z \mathbf{a}_v^t + \mathbf{U}^z \mathbf{h}_v^{t-1})$$
$$\mathbf{r}_v^t = \sigma(\mathbf{W}^r \mathbf{a}_v^t + \mathbf{U}^r \mathbf{h}_v^{t-1})$$
$$\tilde{\mathbf{h}}_v^t = tanh(\mathbf{W}\mathbf{a}_v^t + \mathbf{U}(\mathbf{r}_v^t \odot \mathbf{h}_v^{t-1}))$$
$$\mathbf{h}_v^t = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{t-1} + \mathbf{z}_v^t \odot \tilde{\mathbf{h}}_v^t$$

[1] Li et al. Gated Graph Sequence Neural Networks. In ICLR'16

Step 2: Generates a a graph level representation vector as follows:

$$\mathbf{h}_G = tanh\Big(\sum_{v \in V} \sigma\big(i(\mathbf{h}_v^T, \mathbf{x}_v) \odot tanh\big(j(\mathbf{h}_v^T, \mathbf{x}_v)\big)\big)\Big)$$

$\sigma\big(i(\mathbf{h}_v^T, \mathbf{x}_v)$: a soft attention mechanism that decides which nodes are relevant to the current graph-level task

$i(\cdot), j(\cdot)$: neural networks that take the concatenation of $\mathbf{h}_v^T, \mathbf{x}_v$ as input and output real-valued vectors

Step 1: Aggregates node information in local neighborhoods to extract local substructure information:

$$\mathbf{Z}^{t+1} = f(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}\mathbf{Z}^{t}\mathbf{W}^{t})$$

$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$
$\mathbf{D}$: a diagonal matrix such that $\mathbf{D}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$
$\mathbf{W}^t$: a matrix of trainable parameters
$\mathbf{Z}^0$: a matrix containing the initial vertex features as rows
f: a nonlinear activation function

After T iterations, concatenates the outputs $\mathbf{Z}^t, t = 1, \ldots, T$ horizontally to form a concatenated output:

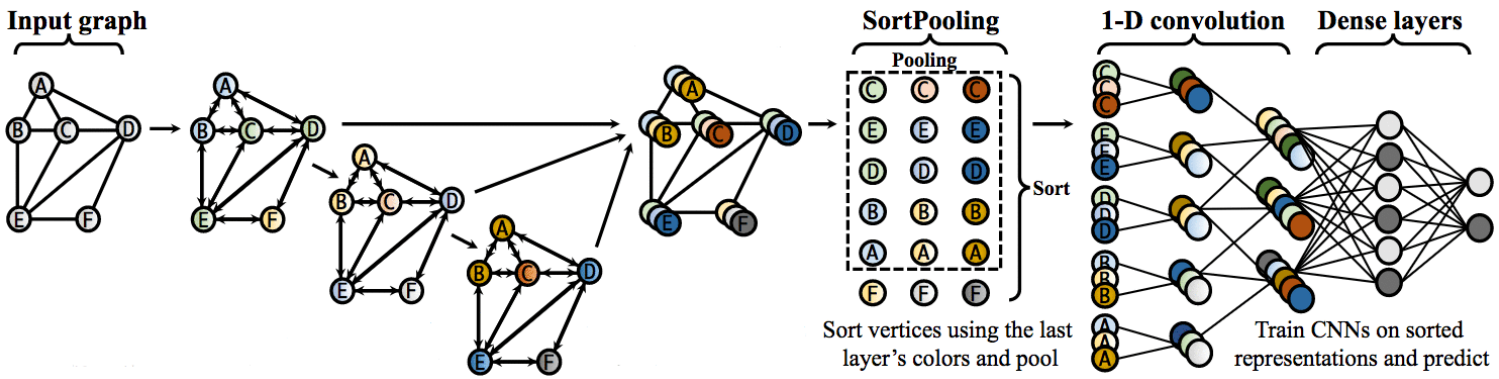
$$\mathbf{Z}^{1:T} = [\mathbf{Z}^1, \ldots, \mathbf{Z}^T]$$

[1] Zhang et al. An End-to-End Deep Learning Architecture for Graph Classification. In AAAI'18

Step 2: SortPooling layer:

- Sorts the output $\mathbf{Z}^{1:T}$ of step 1 row-wise:
  - vertices are sorted in a descending order based on the last component of $\mathbf{Z}^{T}$
  - vertices that have the same value in the last component are compared based on the second to last component and so on
- Unifies the sizes of the outputs to handle graphs with different numbers of vertices:
  - Truncates/extends the output tensor in the first dimension from n to k

Output is then passed to traditional CNN



DCNN

Step 1: Applies a diffusion process to compute new node representations:

$$\mathbf{Z} = \tilde{\mathbf{P}}\mathbf{X}$$

$\mathbf{P} = \mathbf{D}^{-1}\mathbf{A}$: a degree-normalized transition matrix
$\tilde{\mathbf{P}}$: a tensor that contains the T hops of graph diffusion $\mathbf{P}, \mathbf{P}^2, \mathbf{P}^3, \ldots, \mathbf{P}^T$
$\mathbf{X}$: matrix that contains the initial node representations

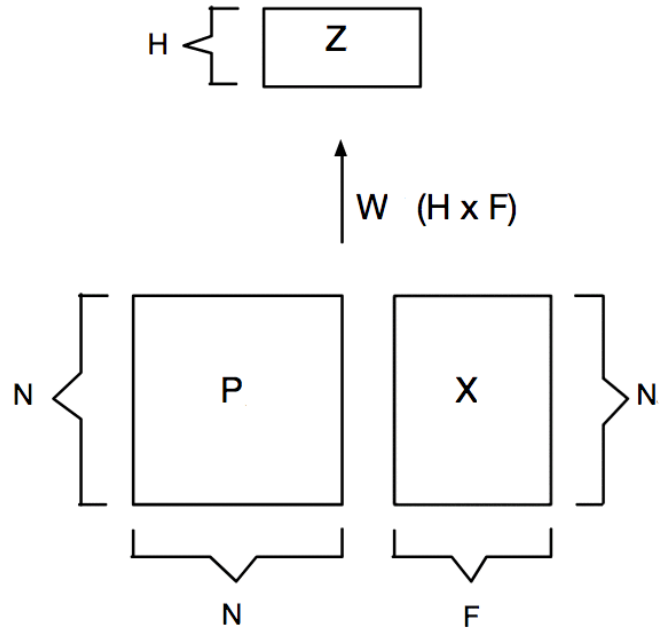
Step 2: DCNN first computes the mean activation over the nodes:

$$\bar{\mathbf{Z}} = \frac{\mathbf{1}^\top \mathbf{Z}}{N}$$

and then produces a graph representation as follows:

$$\mathbf{Z}_G = f(\mathbf{W} \odot \bar{\mathbf{Z}})$$

where f is an activation function



[1] Atwood and Towsley. Diffusion-Convolutional Neural Networks. In NIPS'16

Graph Pooling

Message passing neural networks:

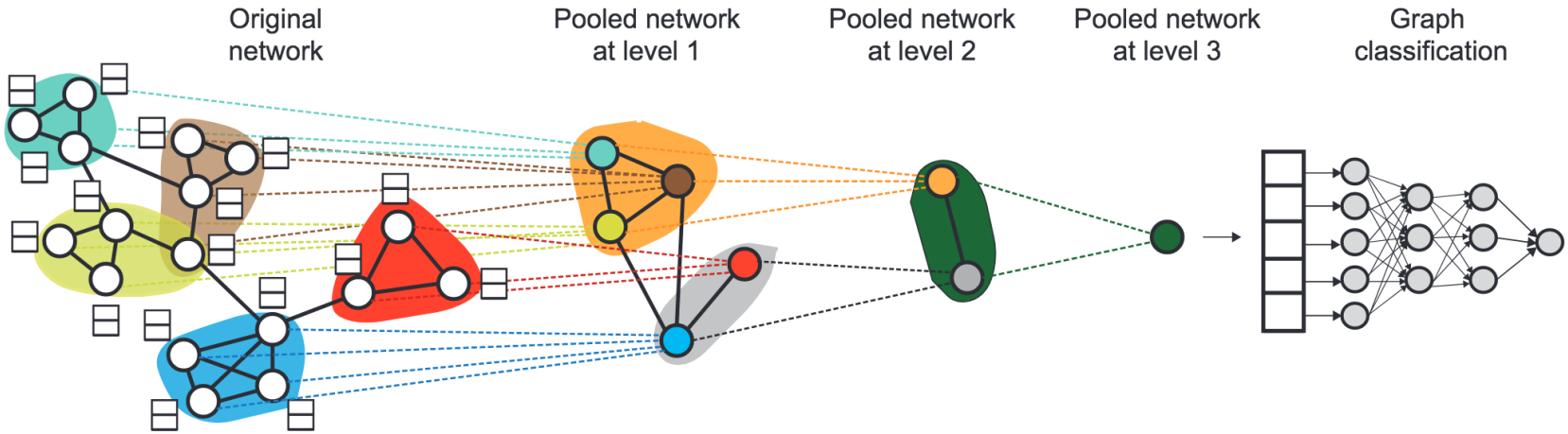
- **Step 1** learns individual node representations
- **Step 2** simply globally aggregates these representations
  - mean/max/sum of all node embeddings
  - pool by sorting (e. g., DGCNN)

Problem:
How to aggregate information in a hierarchical way to capture the entire graph?

DiffPool

- Learns hierarchical pooling analogous to CNNs
- Sets of nodes are pooled hierarchically
- Soft assignment of nodes to next-level nodes

A different GNN is learned at every level of abstraction



[1] Ying et al. Hierarchical Graph Representation Learning via Differentiable Pooling. In NIPS'18

DiffPool

The matrix $\mathbf{S}^{(l)} \in \mathbb{R}^{n_l \times n_{l+1}}$

- corresponds to the learned cluster assignment matrix at layer l
- each row corresponds to one of the n_l nodes (or clusters) at layer l and each column to one of the n_{l+1} clusters at the next layer $l+1$
- it provides a soft assignment of each node at layer l to a cluster in the next coarsened layer $l+1$

Each DiffPool layer coarsens the input graph:

$$\mathbf{X}^{(l+1)} = \mathbf{S}^{(l)^\top} \mathbf{Z}^{(l)}$$

$$\mathbf{A}^{(l+1)} = \mathbf{S}^{(l)^\top} \mathbf{A}^{(l)} \mathbf{S}^{(l)}$$

$\mathbf{A}^{l+1}$: coarsened adjacency matrix
$\mathbf{X}^{l+1}$: matrix of embeddings for each node/cluster

DiffPool

- DiffPool generates the assignment and embedding matrices using two separate message passing neural networks
- Both are applied to the input cluster node features $\mathbf{X}^{(l)}$ and coarsened adjacency matrix $\mathbf{A}^{(l)}$

$$\mathbf{Z}^{(l)} = MPNN_{l,embed}(\mathbf{A}^{(l)}\mathbf{X}^{(l)})$$
$$\mathbf{S}^{(l)} = softmax\big(MPNN_{l,embed}(\mathbf{A}^{(l)}\mathbf{X}^{(l)})\big)$$

where the softmax function is applied in a row-wise fashion

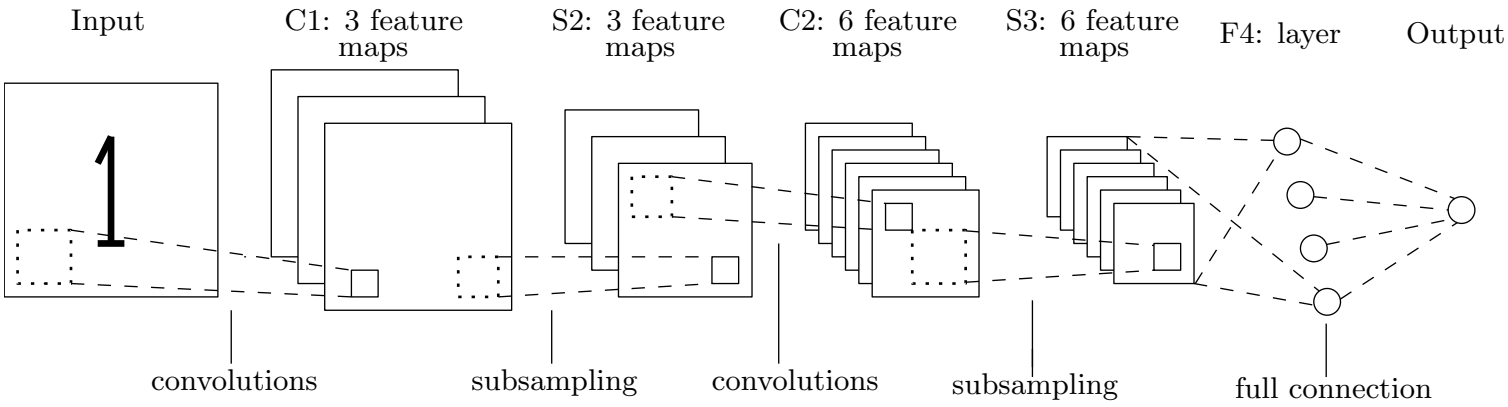
- $MPNN_{l,embed}$ generates new embeddings for the input nodes at this layer
- $MPNN_{l,pool}$ generates a probabilistic assignment of the input nodes to n_{l+1} clusters

Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

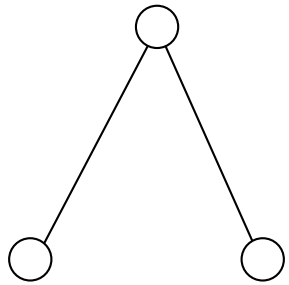
Convolutional Neural Networks

In recent years, CNNs have proven extremely successful on many domains (i. e. Computer Vision, Natural Language Processing)



Useful when we expect to find strong local clues regarding class membership, but these clues can appear in different places in the input

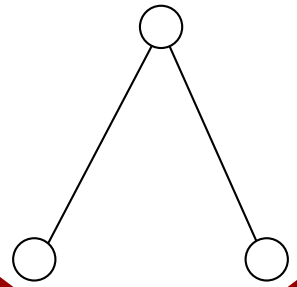
Use Adjacency Matrix as Image



$$A_1 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad A_3 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

- Isomorphic graphs can have totally different adjacency matrices

Use Adjacency Matrix as Image



$$A_1 = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \qquad A_3 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

- Isomorphic graphs can have totally different adjacency matrices

Patchy-San

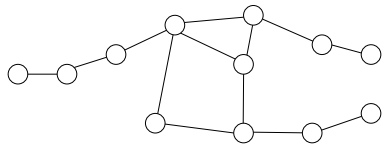
Patchy-San is a spatial approach which can be decomposed in three steps:

- select a node
- construct its neighborhood
- normalize the selected subgraph (i. e. order the neighboring nodes)

The extracted patches are then fed into a conventional CNN

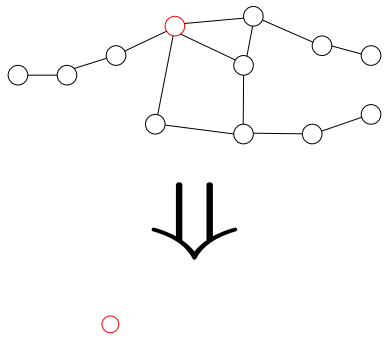
[1] Niepert et al. Learning Convolutional Neural Networks for Graphs. In ICML'16

Patchy-San



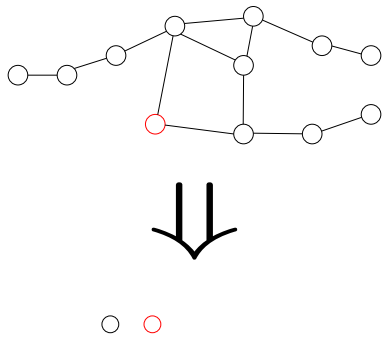
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)



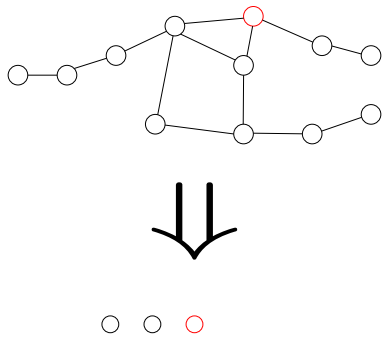
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)



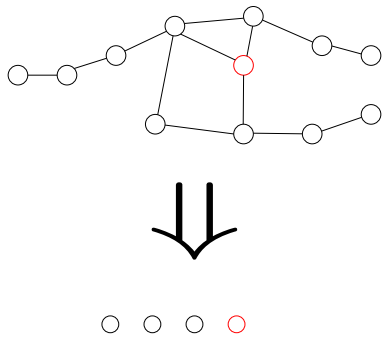
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)



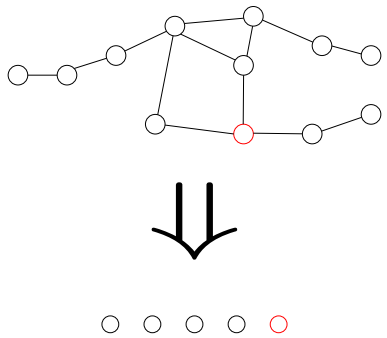
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)



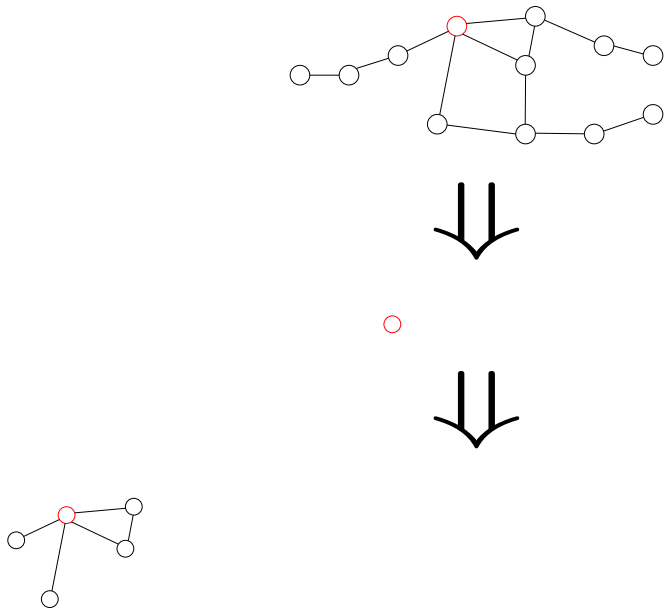
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)



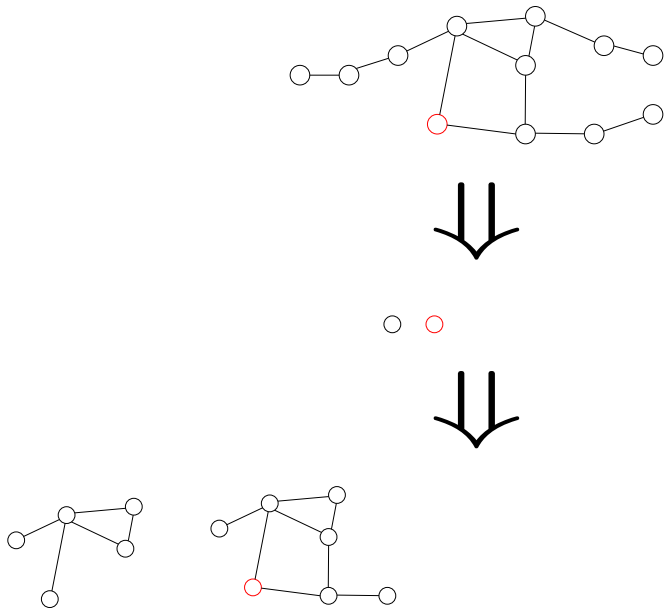
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e. g., using BFS)



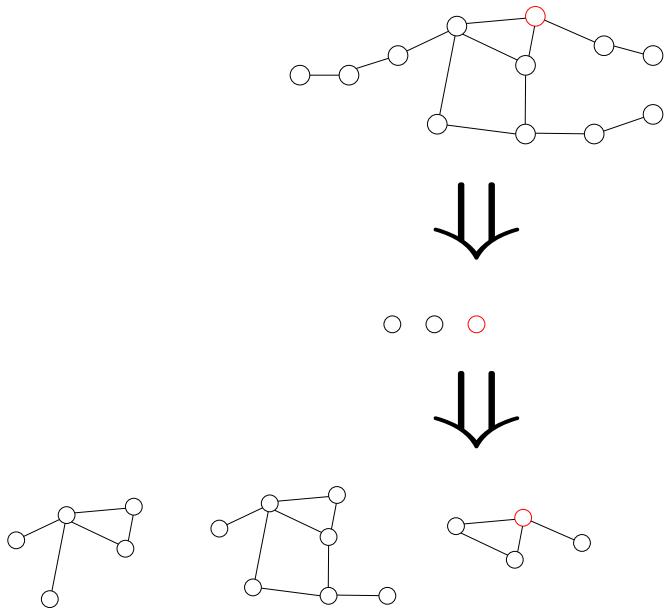
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e. g., using BFS)



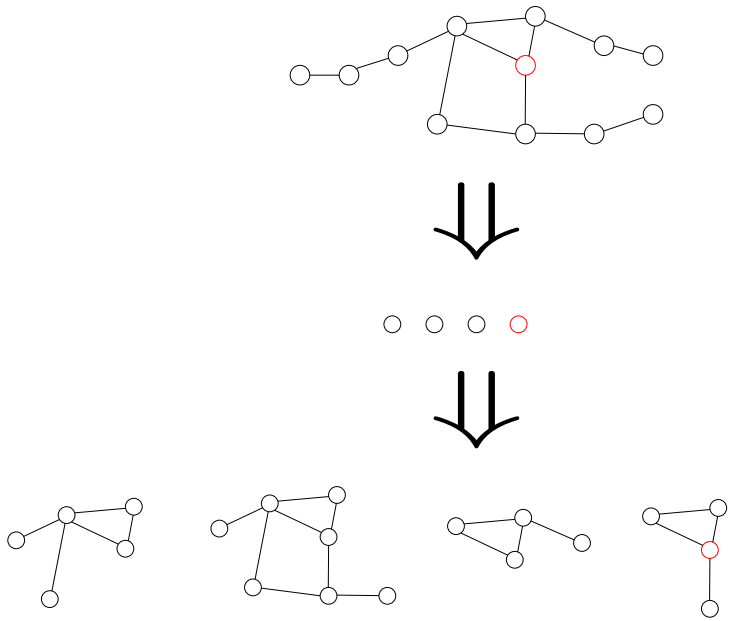
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e. g., using BFS)



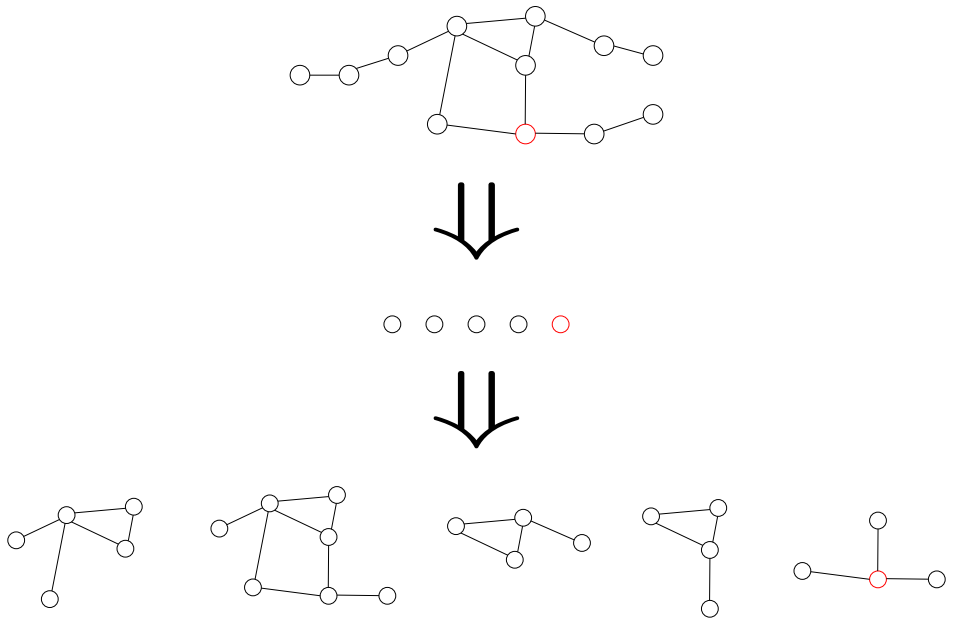
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e. g., using BFS)



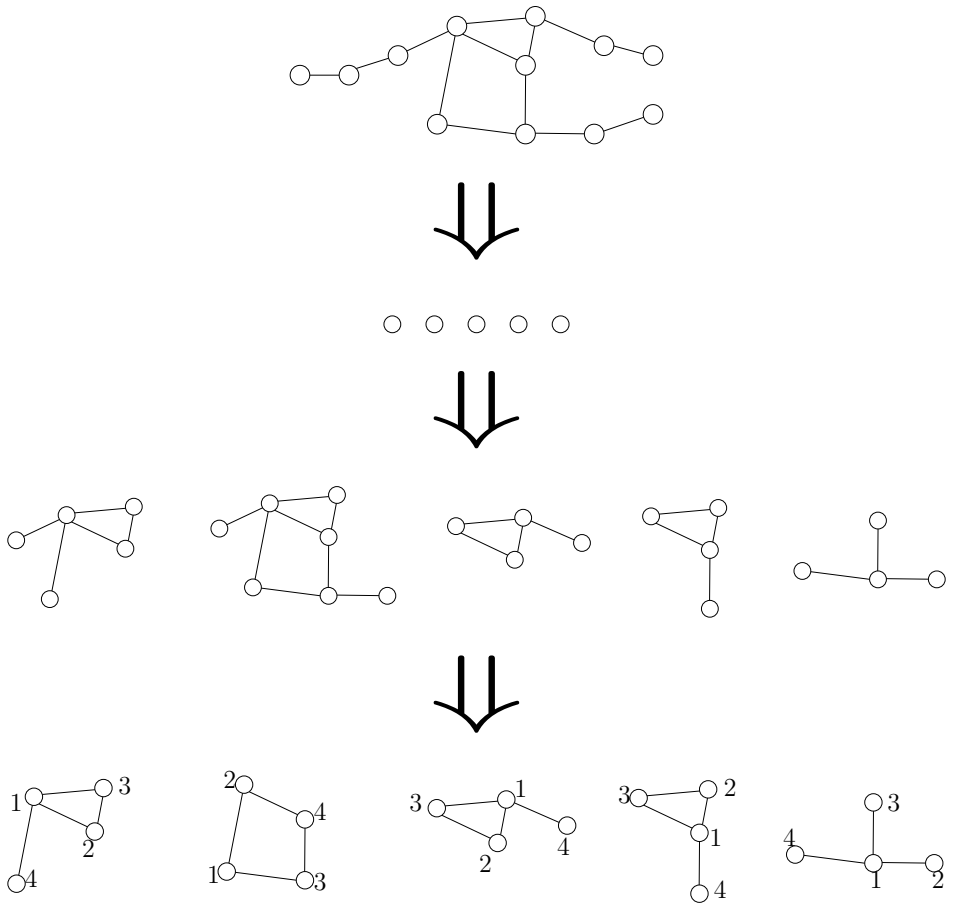
Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e. g., using BFS)

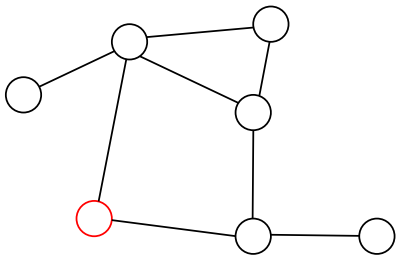


Patchy-San

- Select $w = 5$ nodes (e. g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e. g., using BFS)
- Keep exactly $k = 4$ nodes and arrange them in an order

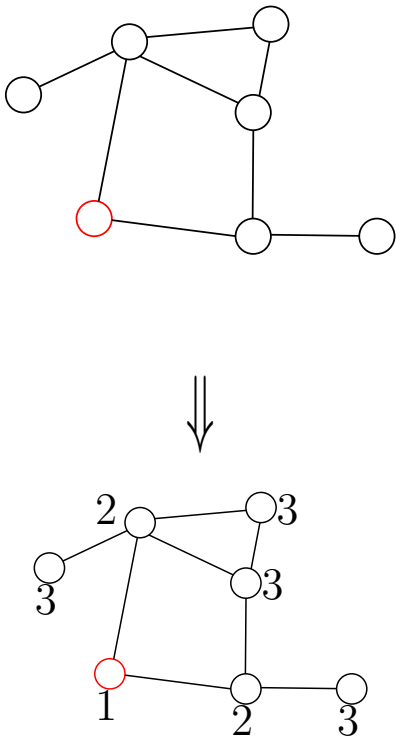


Neighborhood Normalization



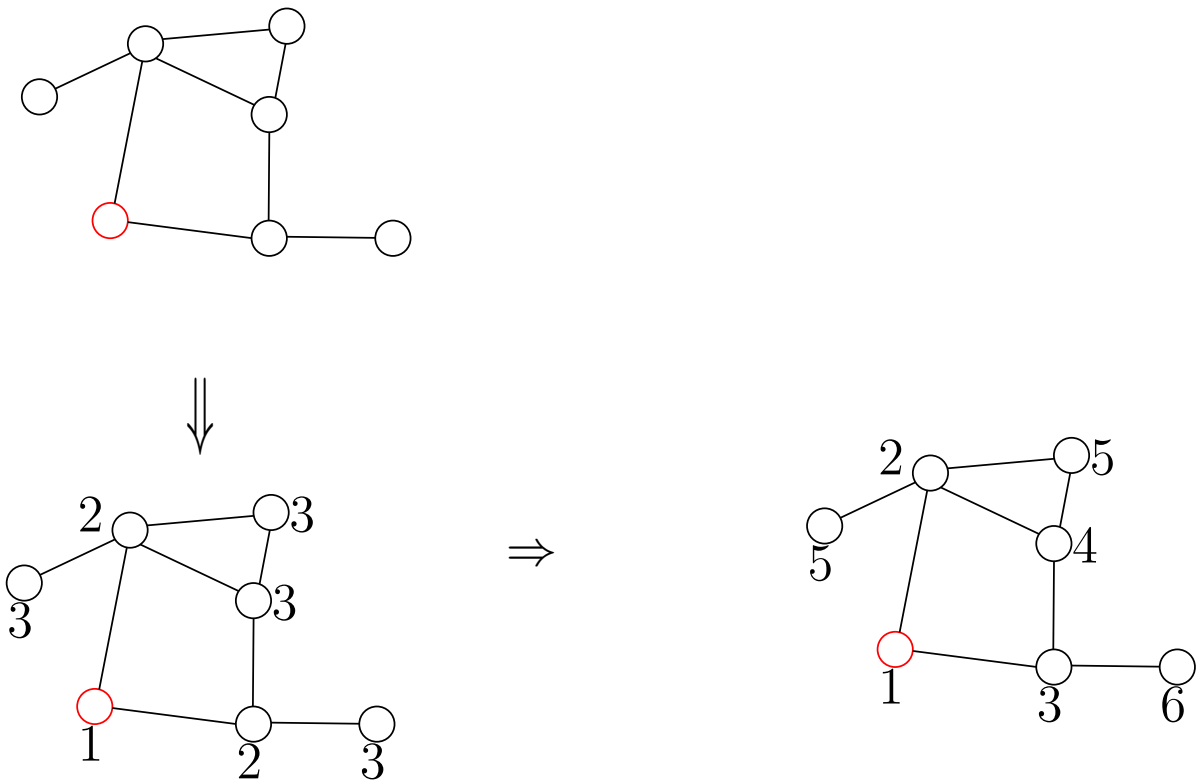
Neighborhood Normalization

- Distance to root



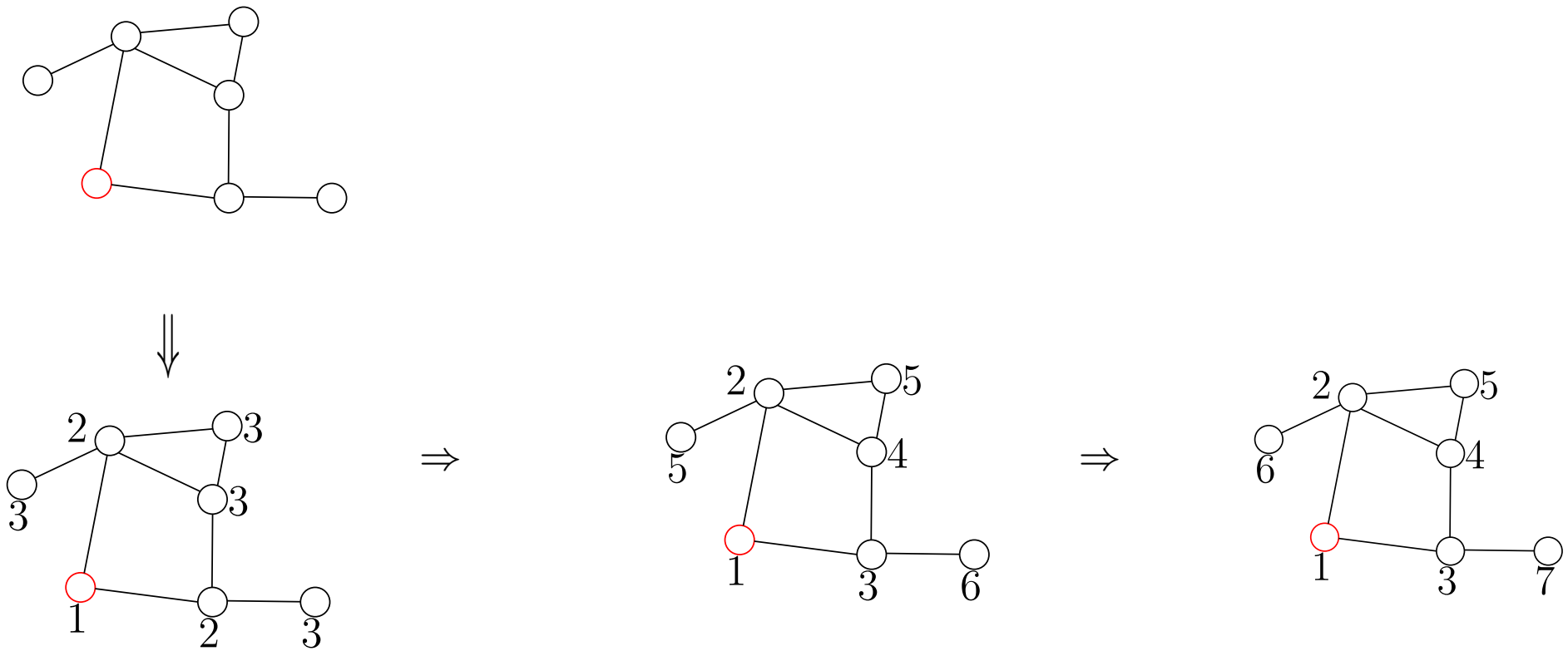
Neighborhood Normalization

- Distance to root
- Use centrality measures



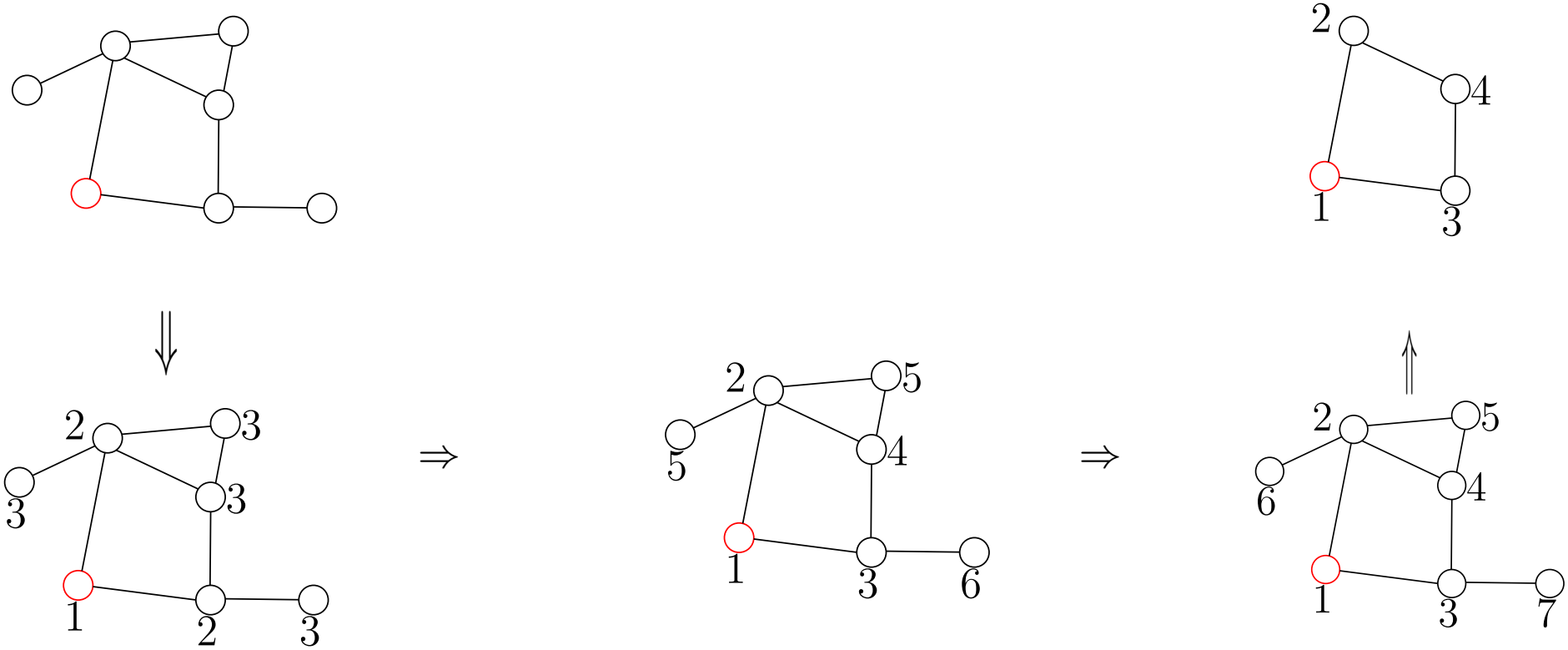
Neighborhood Normalization

- Distance to root
- Use centrality measures
- Canonicalization



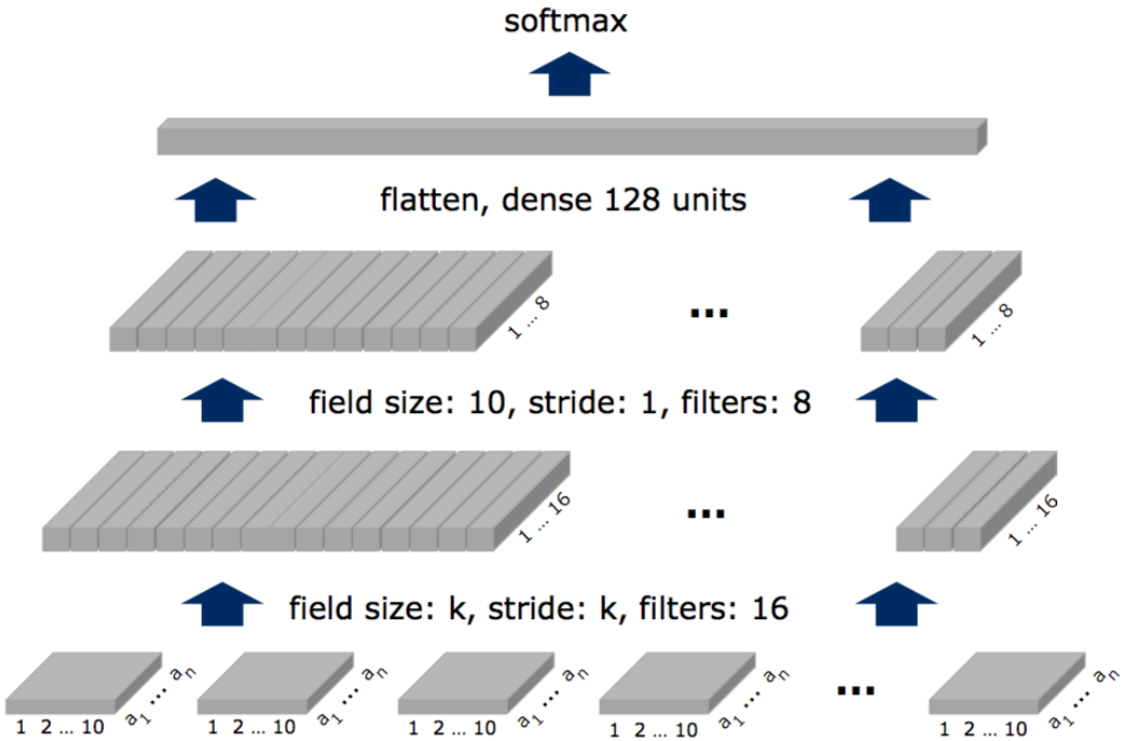
Neighborhood Normalization

- Distance to root
- Use centrality measures
- Canonicalization
- Remove extra nodes

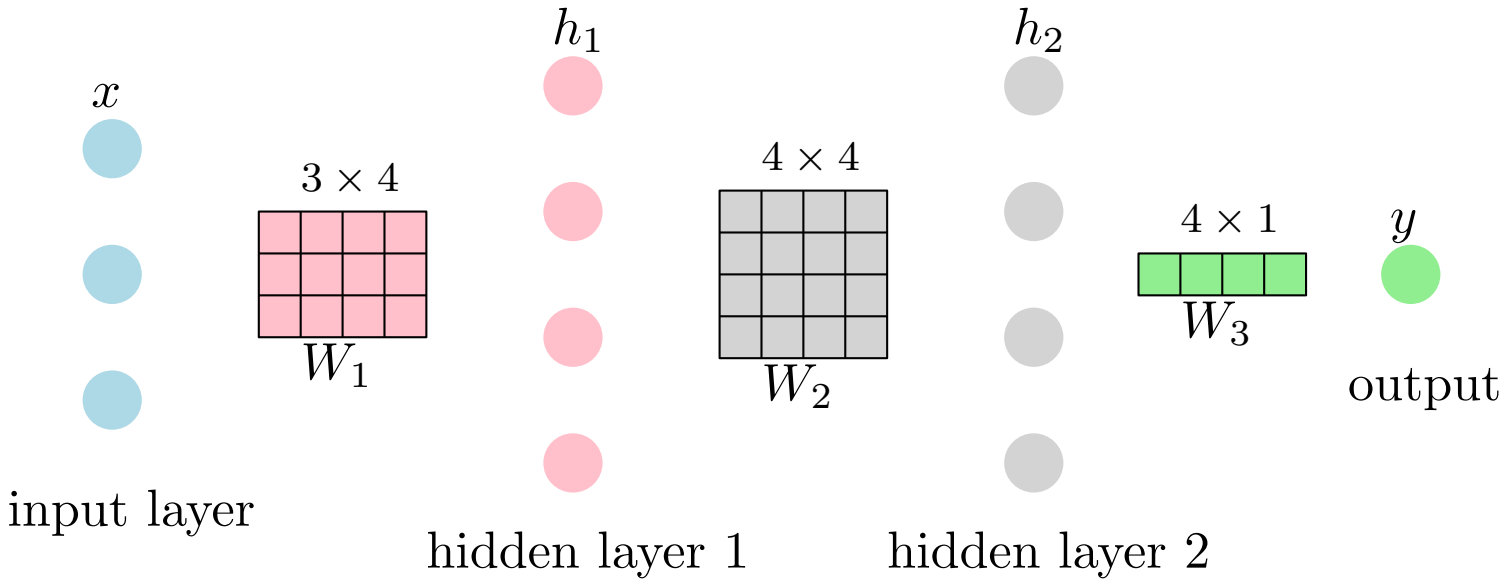


CNN architecture of Patchy-San

- The $k \times k$ adjacency matrices of the neighborhoods of the w vertices given as input to a CNN: Each adjacency matrix convolved with filters of size $k \times k$
- In case of node labels (n discrete labels) $\rightarrow$ create a $k \times n$ matrix to serve as an indicator matrix: Each indicator matrix convolved with filters of size $k \times n$
- These two types of matrices correspond to different channels of the CNN
- Two convolutional layers in total, followed by a fully-connected feedforward network

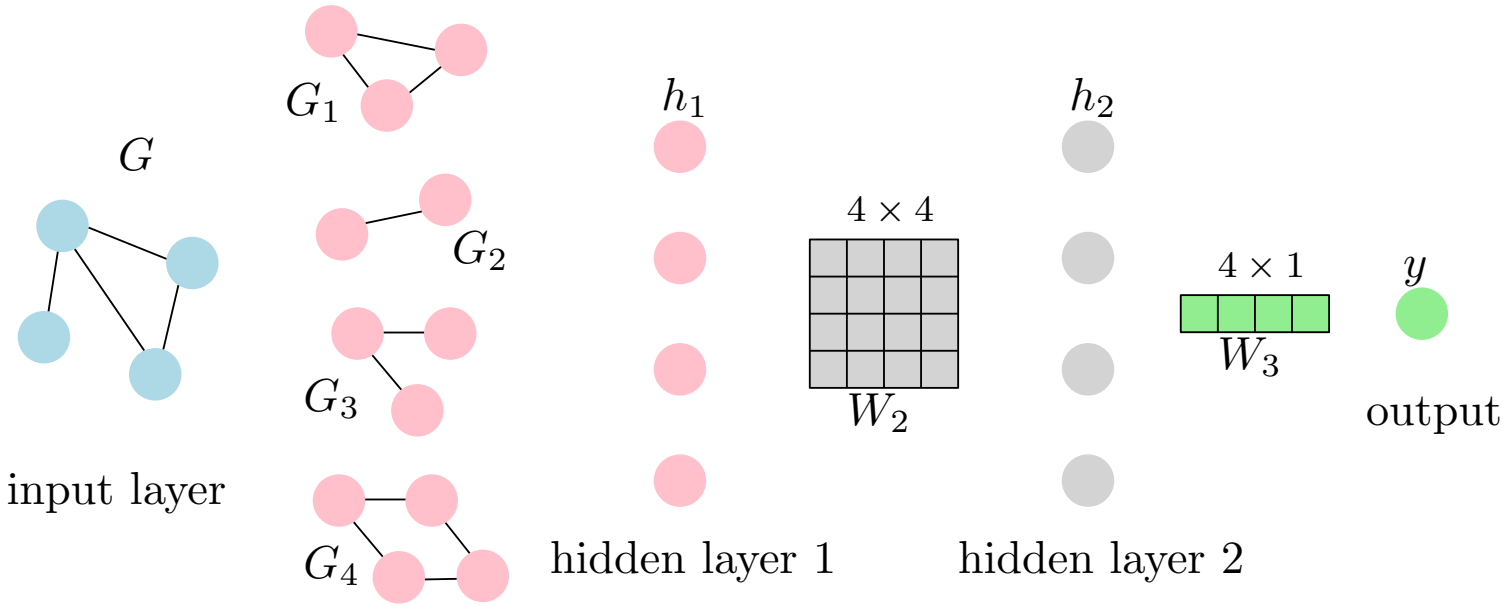


Multilayer Perceptron



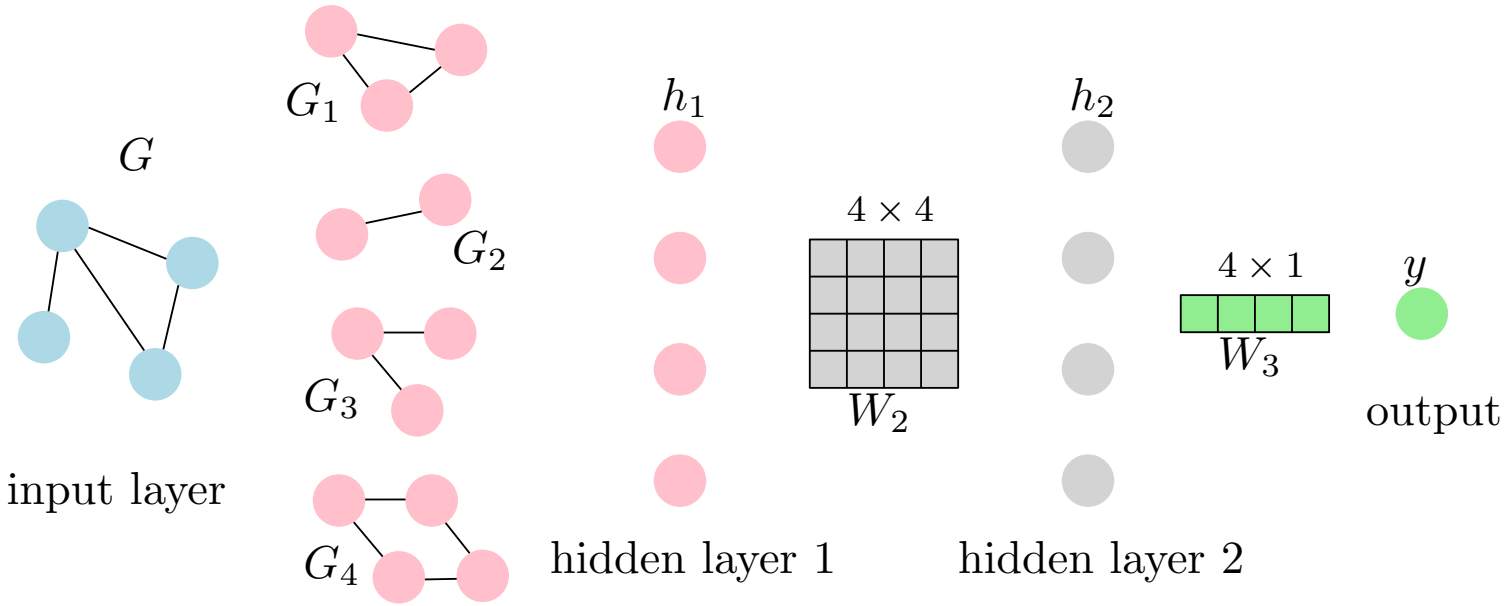
$$h_1 = f(x^\top W_1)$$
$$h_2 = f(h_1^\top W_2)$$
$$y = f(h_2^\top W_3)$$

Graphs As Hidden Units in MLP



$$h_1^{(1)} = f(G, G_1)$$
$$h_1^{(2)} = f(G, G_2)$$
$$h_1^{(3)} = f(G, G_3)$$
$$h_1^{(4)} = f(G, G_4)$$
$$h_2 = f(h_1^\top W_2)$$
$$y = f(h_2^\top W_3)$$

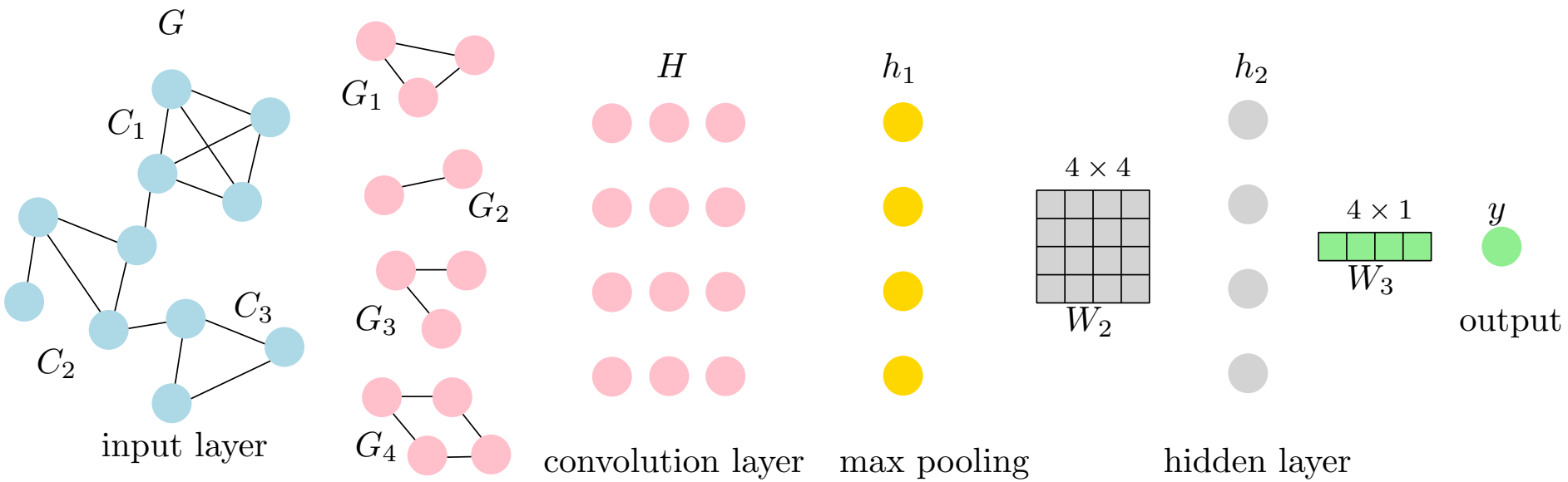
Graphs As Hidden Units in MLP



$$
\left.
\begin{aligned}
h_1^{(1)} &= f(G, G_1) \\
h_1^{(2)} &= f(G, G_2) \\
h_1^{(3)} &= f(G, G_3) \\
h_1^{(4)} &= f(G, G_4)
\end{aligned}
\right\} \text{graph kernel}
$$

$$
\begin{aligned}
h_2 &= f(h_1^\top W_2) \\
y &= f(h_2^\top W_3)
\end{aligned}
$$

Graphs As Hidden Units in CNN



$$H_{ij} = f(C_i, G_j)$$
$$h_1 = maxpool(H)$$
$$h_2 = f(h_1^\top W_2)$$
$$y = f(h_2^\top W_3)$$

Blending CNNs with Graph Kernels

Performing graph classification using both

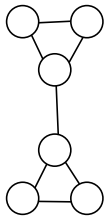
- graph kernels
- CNNs

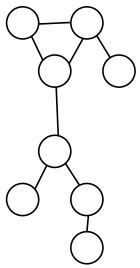
Thanks to its

- kernel component, it can be easily applied to structured data (e. g., graphs)
- CNN component, learns task-dependent features without the need for feature engineering

[1] Nikolentzos et al. Kernel Graph Convolutional Neural Networks. In ICANN'18

Overview

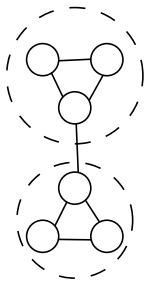
G_1 

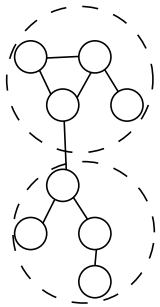
G_2 

$\vdots$

Overview

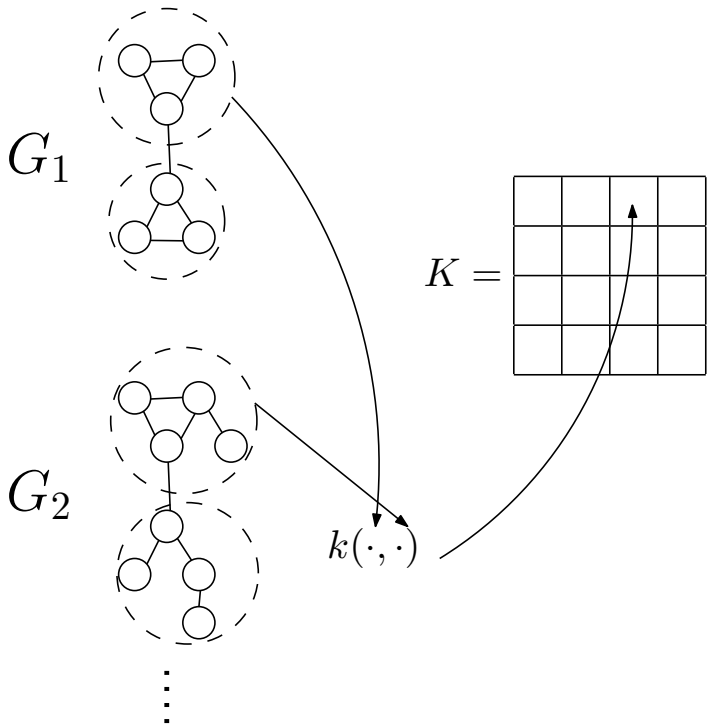
- Extract a set of subgraphs that will play the role of patches

G_1 

G_2 

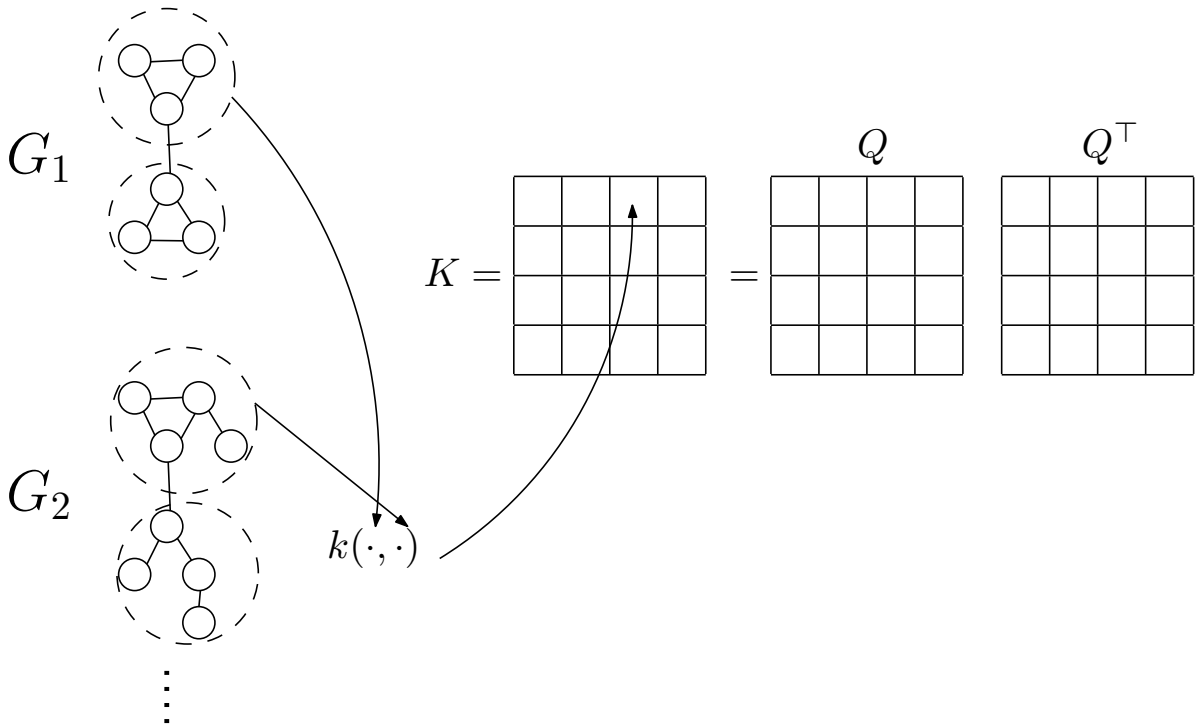
Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels



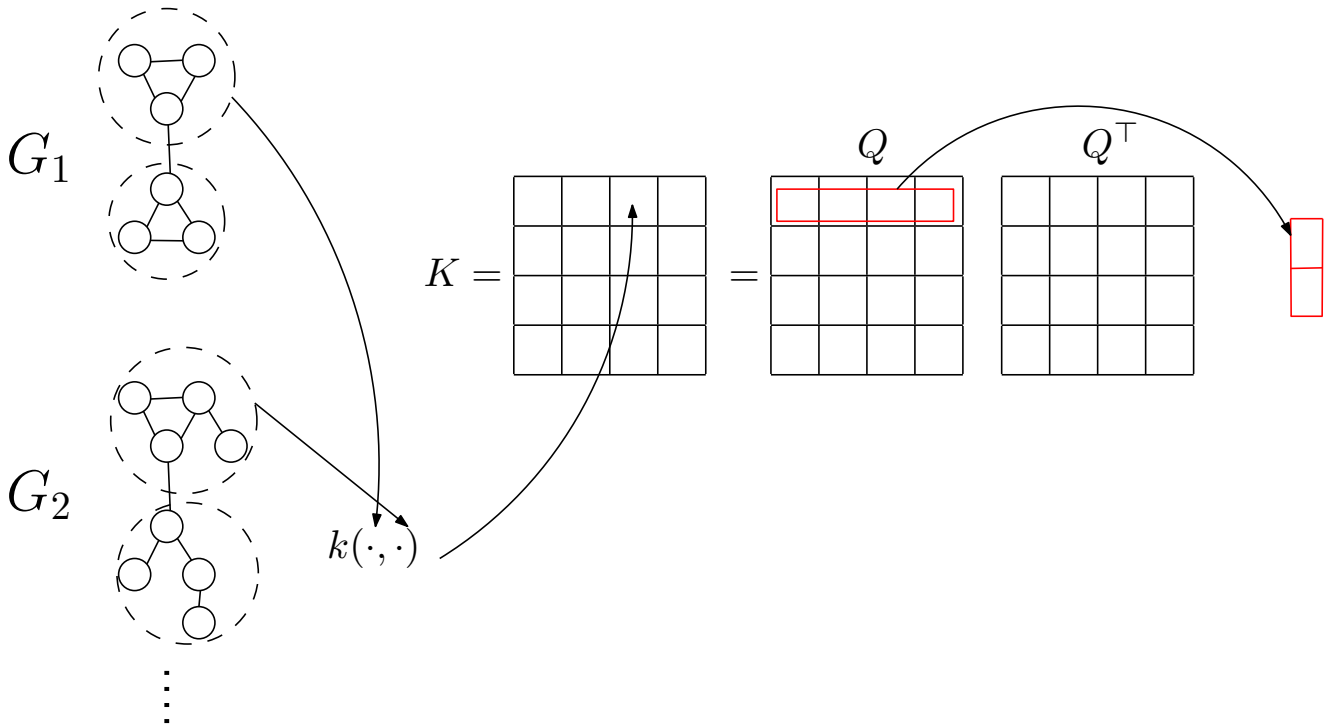
Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels
- Decompose kernel matrix to get subgraph representation (or approximate it using Nystrom)



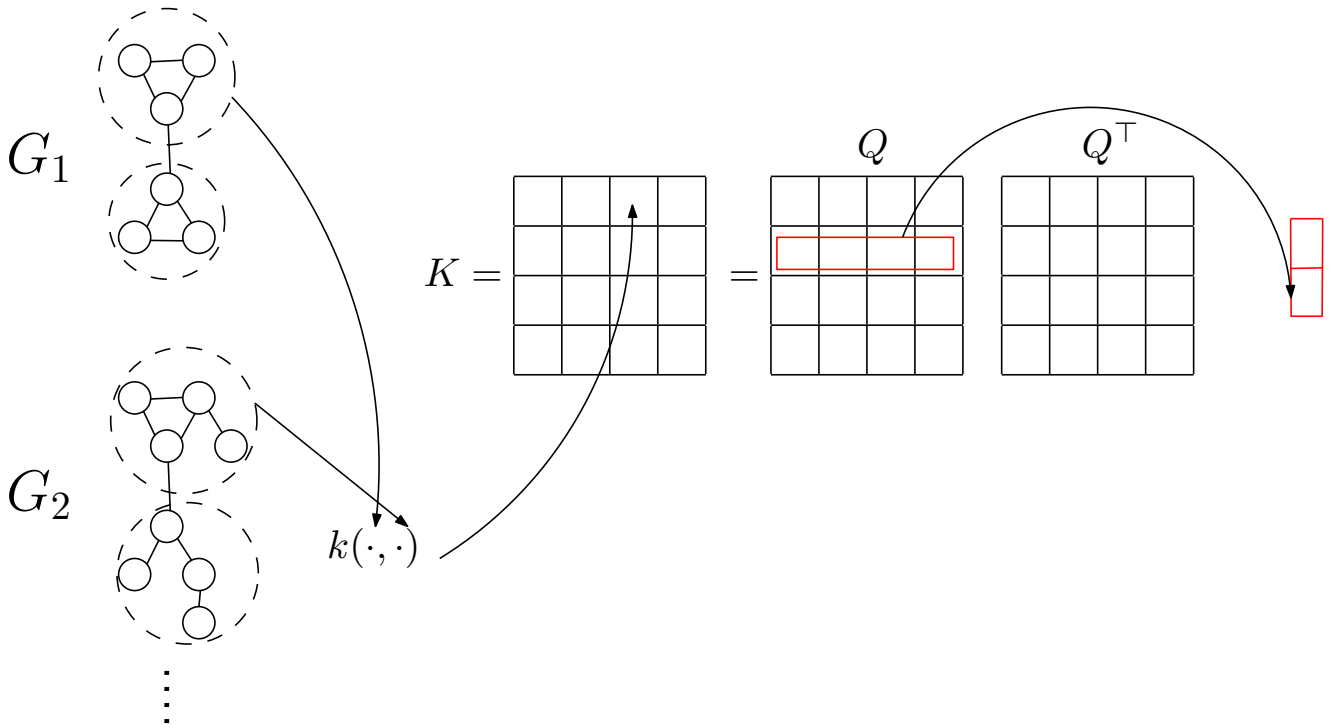
Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels
- Decompose kernel matrix to get subgraph representation (or approximate it using Nystrom)
- These vectors are then convolved with the filters of the CNN



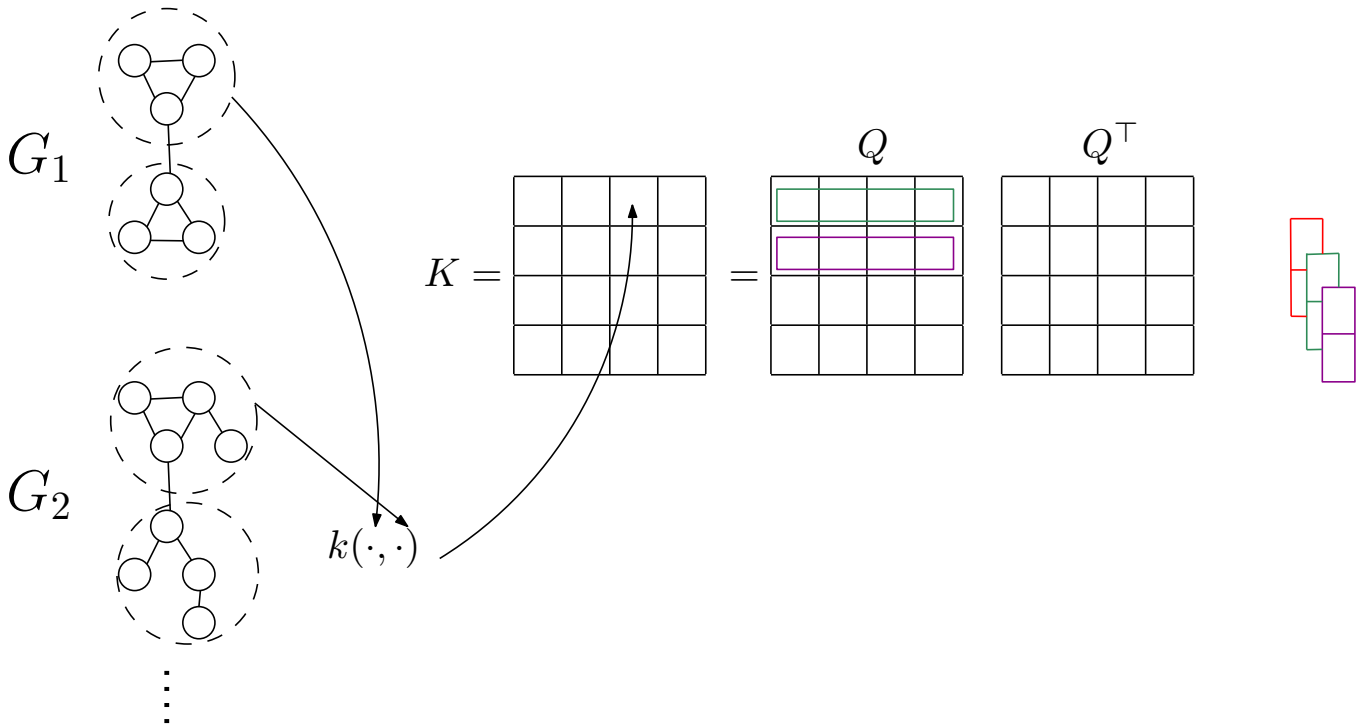
Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels
- Decompose kernel matrix to get subgraph representation (or approximate it using Nystrom)
- These vectors are then convolved with the filters of the CNN



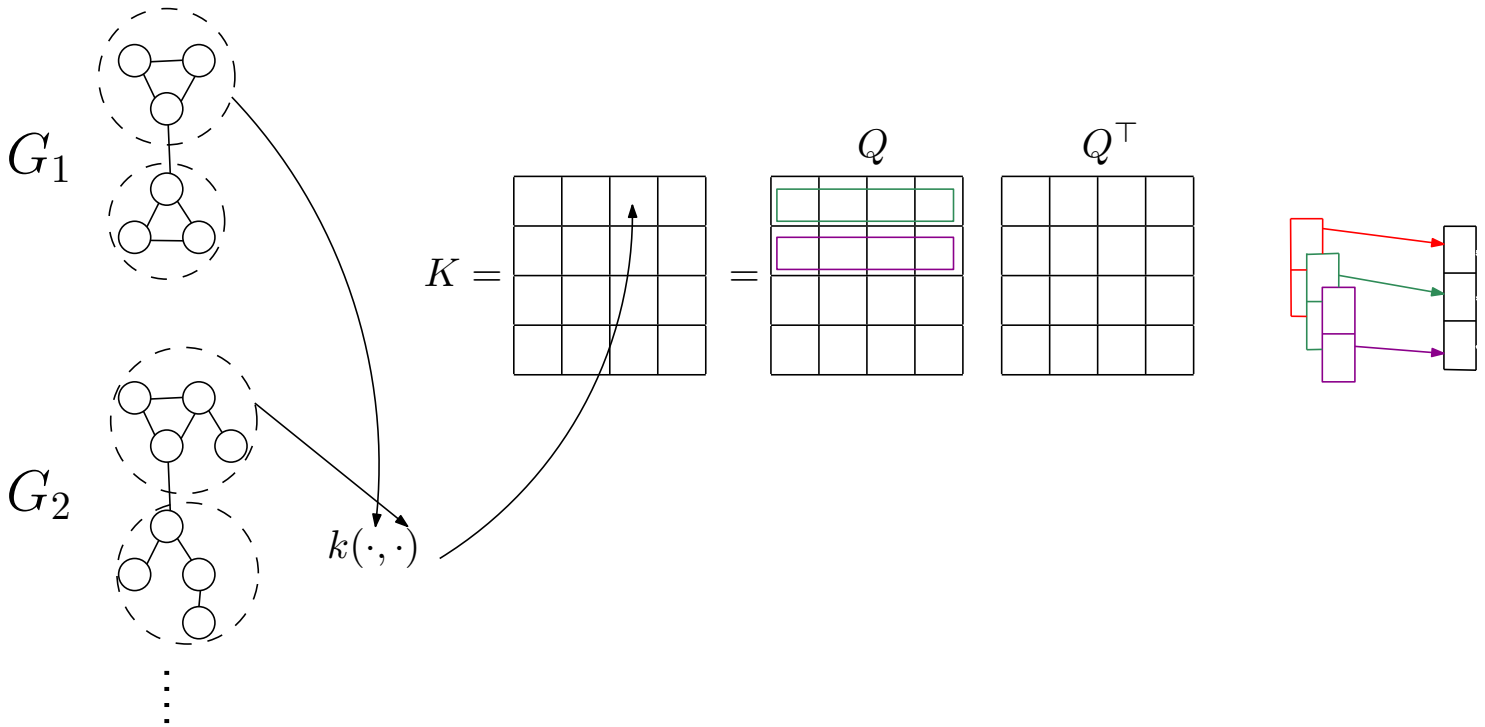
Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels
- Decompose kernel matrix to get subgraph representation (or approximate it using Nystrom)
- These vectors are then convolved with the filters of the CNN



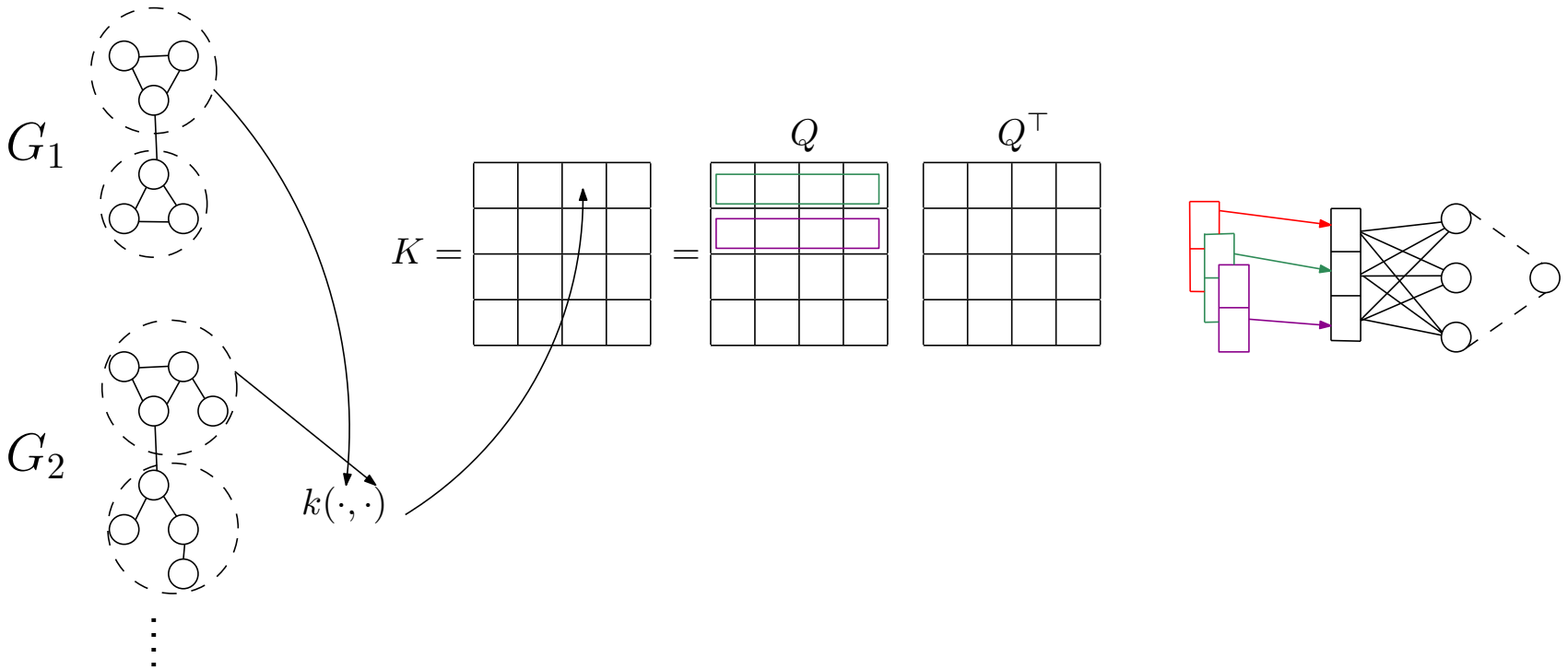
Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels
- Decompose kernel matrix to get subgraph representation (or approximate it using Nystrom)
- These vectors are then convolved with the filters of the CNN
- Then, a pooling layer is followed by a fully-connected one to output class probabilities for the input graph



Overview

- Extract a set of subgraphs that will play the role of patches
- To give each graph as input to the CNN, we employ graph kernels
- Decompose kernel matrix to get subgraph representation (or approximate it using Nystrom)
- These vectors are then convolved with the filters of the CNN
- Then, a pooling layer is followed by a fully-connected one to output class probabilities for the input graph



Employed Kernels

2 graph kernels giving rise to up to 2 channels:

1. Shortest path kernel (SP): counts pairs of shortest paths in two graphs having the same source and sink labels and identical length [Borgwardt and Kriege, ICDM '05]

2. Weisfeiler-Lehman subtree kernel (WL): for a number of iterations it performs an exact matching between the compressed multiset labels of two graphs, while at each iteration it updates these labels [Shervashidze et al., JMLR '11]

Graph Classification

Method \ Dataset	ENZYMES	NCI1	PROTEINS	PTC-MR	D&D
SP	40.10 (± 1.50)	73.00 (± 0.51)	75.07 (± 0.54)	58.24 (± 2.44)	> 3 days
GR	26.61 (± 0.99)	62.28 (± 0.29)	71.67 (± 0.55)	57.26 (± 1.41)	78.45 (± 0.26)
RW	24.16 (± 1.64)	> 3 days	74.22 (± 0.42)	57.85 (± 1.30)	> 3 days
WL	53.15 (± 1.14)	80.13 (± 0.50)	72.92 (± 0.56)	56.97 (± 2.01)	77.95 (± 0.70)
Deep Kernels	**53.43** (± 0.91)	**80.31** (± 0.46)	75.68 (± 0.54)	60.08 (± 2.55)	NA
PSCN $k = 10$	NA	76.34 (± 1.68)	75.00 (± 2.51)	62.29 (± 5.68)	76.27 (± 2.64)
KCNN SP	46.35 (± 0.39)	75.70 (± 0.31)	74.27 (± 0.22)	**62.94** (± 1.69)	76.63 (± 0.09)
KCNN WL	43.08 (± 0.68)	75.83 (± 0.25)	**75.76** (± 0.28)	61.52 (± 1.41)	75.80 (± 0.07)
KCNN SP + WL	48.12 (± 0.23)	77.21 (± 0.22)	73.79 (± 0.29)	62.05 (± 1.41)	**78.83** (± 0.29)

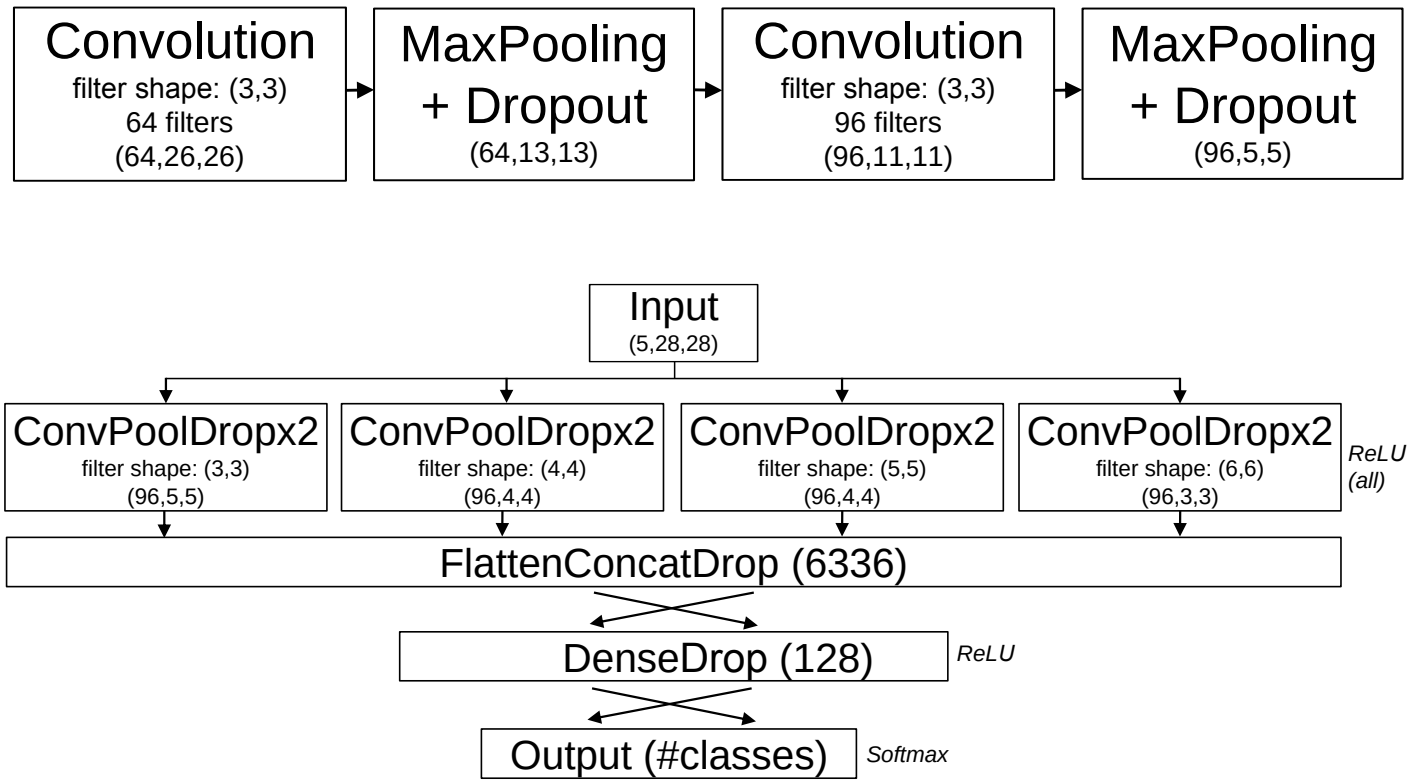
Method \ Dataset	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
GR	65.87 (± 0.98)	43.89 (± 0.38)	77.34 (± 0.18)	41.01 (± 0.17)	72.84 (± 0.28)
Deep GR	66.96 (± 0.56)	44.55 (± 0.52)	78.04 (± 0.39)	41.27 (± 0.18)	73.09 (± 0.25)
PSCN $k = 10$	71.00 (± 2.29)	45.23 (± 2.84)	**86.30** (± 1.58)	49.10 (± 0.70)	72.60 (± 2.15)
KCNN SP	69.60 (± 0.44)	45.99 (± 0.23)	77.23 (± 0.15)	44.86 (± 0.24)	70.78 (± 0.12)
KCNN WL	70.46 (± 0.45)	46.44 (± 0.24)	81.85 (± 0.12)	**50.04** (± 0.19)	**74.93** (± 0.14)
KCNN SP + WL	**71.45** (± 0.15)	**47.46** (± 0.21)	78.35 (± 0.11)	44.63 (± 0.18)	74.12 (± 0.17)

Outline

1. Learning Node Embeddings
   - Unsupervised Methods
   - Supervised Methods
2. Graph Classification
   - Introduction
   - Message Passing Neural Networks
   - Spatial Approaches
   - Graphs as Images

Classifying Graphs as Images: CNN Architecture

Example of `ConvPoolDrop`×2 for filters of shape (3,3), no zero-padding, and a stride of 1:



Reaches **99.45%** accuracy on the MNIST handwritten digit classification dataset

Code: `https://github.com/Tixierae/graph_2D_CNN`

How to Represent a Graph as an Image? (1/2)

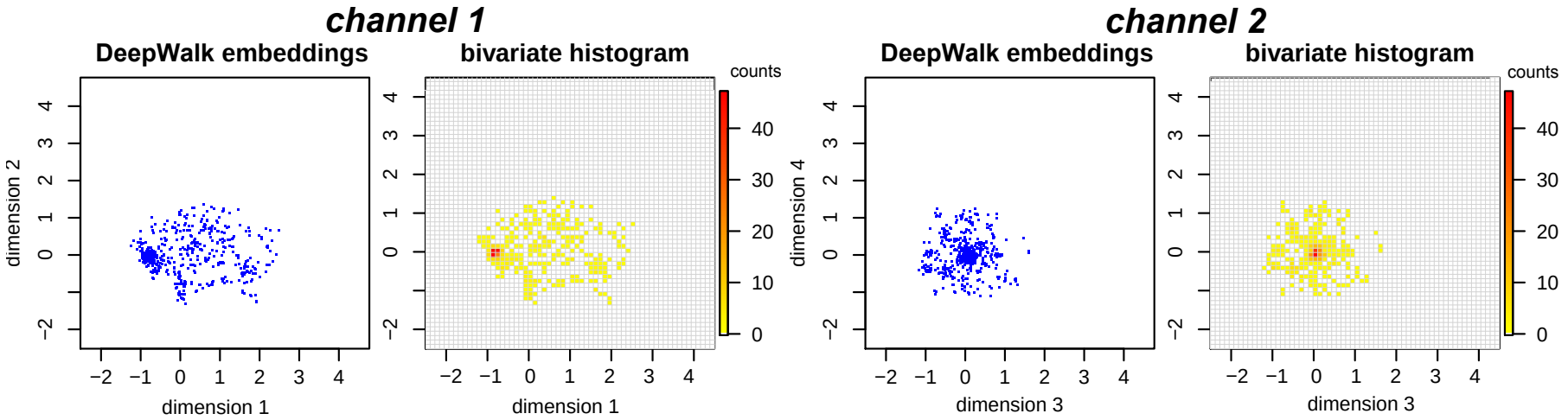
- the Adjacency matrix A or Laplacian matrix L do not satisfy the **spatial dependence** property
- on the other hand, the distance between two nodes in an Euclidean **node embedding space** is meaningful: inversely proportional to node similarity
- idea: represent a graph as a **stack of $d/2$ two-dimensional histograms** of its compressed node embeddings. We retain the first $d \ll D$ principal components, where $D \sim 100$ is the dimension of the embedding space
- PCA also serves an **alignment purpose**

Note: PCA computes the covariance matrix $\mathcal{O}(D^2|V|)$ and carry out its eigenvalue decomposition $\mathcal{O}(D^3)$. May seem complex, but fast in practice as $D \sim 100$

Code: `https://github.com/Tixierae/graph_2D_CNN`

How to Represent a Graph as an Image? (2/2)

Histogram representation of graph ID #10001 (577 nodes, 1320 edges) from the REDDIT-12K dataset:



- bins as **pixels**
- $d/2$ 2D slices of the embedding space as **channels**:
  each pixel is associated with a vector of size $d/2$, whose entries are the counts of the nodes falling into that bin in the corresponding channel
- **resolution** $\sim$ number of bins (fixed along dimensions, same for each channel)

Code: `https://github.com/Tixierae/graph_2D_CNN`

How to Learn Graph Node Embeddings?

- **eigenvectors** associated with the largest eigenvalues of the **adjacency matrix**
- `node2vec` (Grover et al. 2016) learns graph node embeddings via Skip-Gram (Mikolov et al. 2013) on **truncated biased random walks**. Return and in-out parameters p and q are used to bias walks towards exploring larger areas of the graph (**structural equivalence**) or staying in local neighborhoods (**homophily**)

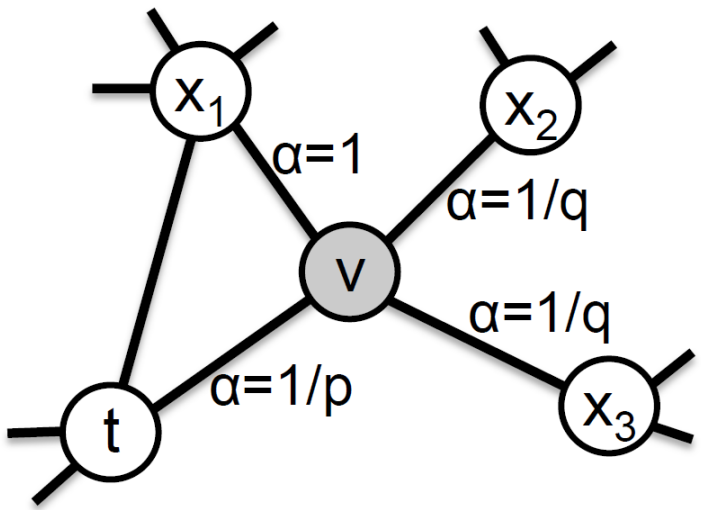


Figure 2: Illustration of our random walk procedure. The walk just transitioned from t to v and is now evaluating its next step out of node v. Edge labels indicate search biases α.

Experimental Evaluation

Experimental comparison conducted in [1]

Compared algorithms:

- graph kernels
  - graphlet [2]
  - WL [3]
  - Deep GK [4]
- neural networks
  - 2D CNN
  - DGCNN
  - Patchy-San (PSCN)

Task: graph classification

[1] Tixier et al. Graph Classification with 2D Convolutional Neural Networks. Submitted to KDD'18

[2] Shervashidze et al. Efficient graphlet kernels for large graph comparison. In AISTATS'09

[3] Shervashidze et al. Weisfeiler-Lehman Graph Kernels. JMLR, 2011

[4] Yanardag and Vishwanathan. Deep Graph Kernels. In KDD'15

Results

Social Interaction Datasets

Method \ Dataset	REDDIT-B (size=2,000;nclasses=2)	REDDIT-5K (4,999;5)	REDDIT-12K (11,929;11)	COLLAB (5,000;3)	IMDB-B (1,000;2)
Graphlet Shervashidze2009	77.26 (± 2.34)	39.75 (± 1.36)	25.98 (± 1.29)	73.42 (± 2.43)	65.40 (± 5.95)
WL Shervashidze2011	78.52 (± 2.01)	50.77 (± 2.02)	34.57 (± 1.32)	**77.82*** (± 1.45)	**71.60** (± 5.16)
Deep GK Yanardag2015	78.04 (± 0.39)	41.27 (± 0.18)	32.22 (± 0.10)	73.09 (± 0.25)	66.96 (± 0.56)
PSCN $k = 10$ Niepert2016	86.30 (± 1.58)	49.10 (± 0.70)	41.32 (± 0.42)	72.60 (± 2.15)	71.00 (± 2.29)
DGCNN Zhang2018	-	-	-	73.76 (± 0.49)	70.03 (± 0.86)
2D CNN (our method)	**89.12*** (± 1.70)	**52.11** (± 2.24)	**48.13*** (± 1.47)	71.33 (± 1.96)	70.40 (± 3.85)

PROTEINS dataset

	2D CNN our method	DGCNN Zhang18	PSCN $k = 10$ Niepert16	HGK-SP Morris16	HGK-WL Morris16	GIK Orsini15	GraphHopper Feragen13	PROP-diff Neumann12	PROP-WL Neumann12
Acc.	**77.12**	75.54	75.00	75.14	74.88	76.6	74.1	73.3	73.1
Std. dev.	2.79	0.94	2.51	0.47	0.64	0.6	0.5	0.4	0.8

- Classification repeated 10 times for each dataset and each model
- Tables report average classification accuracy and std over the 10 runs