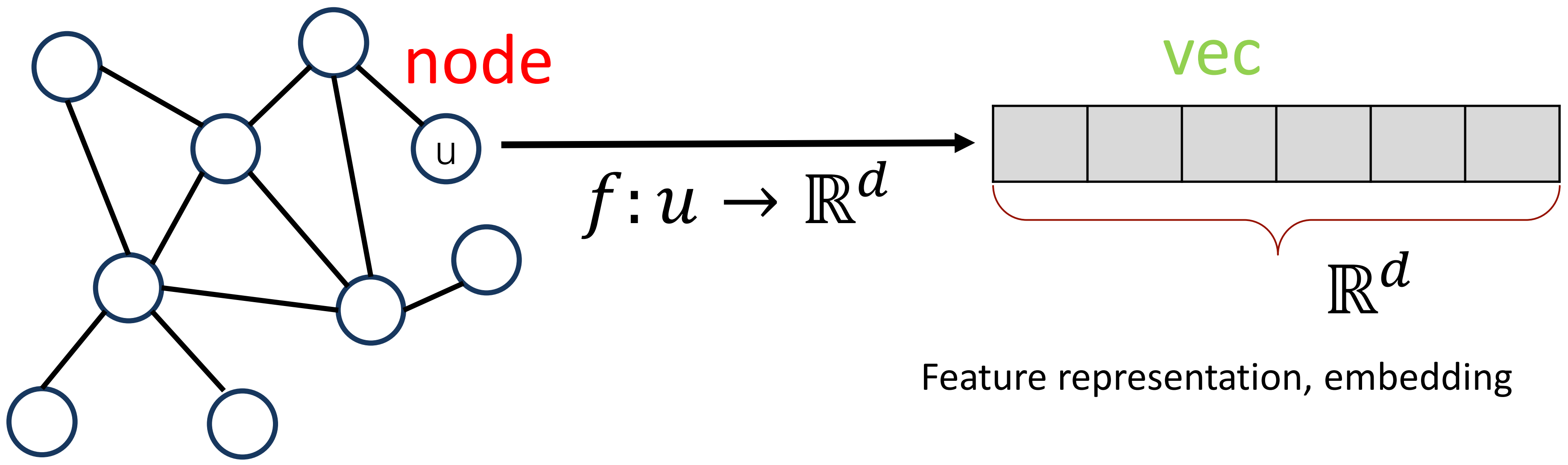
Online Social Networks and Media

Graph Embeddings

Graph embeddings: what are they?


node
u
vec
$f: u \rightarrow \mathbb{R}^d$
$\mathbb{R}^d$
Feature representation, embedding

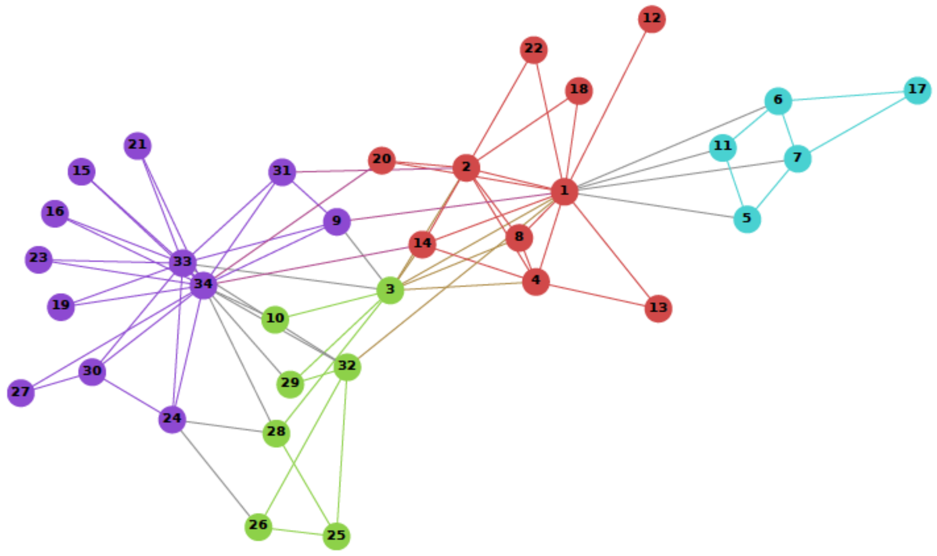
Map nodes to *d*-dimensional vectors so that:
"similar" nodes in the graph have embeddings that *are close together*.

Example

Zachary's Karate Club Network:

Input

Output



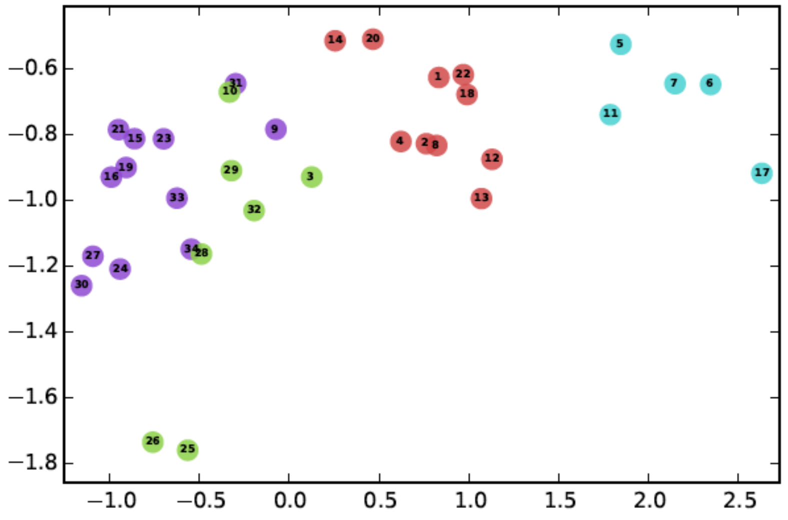
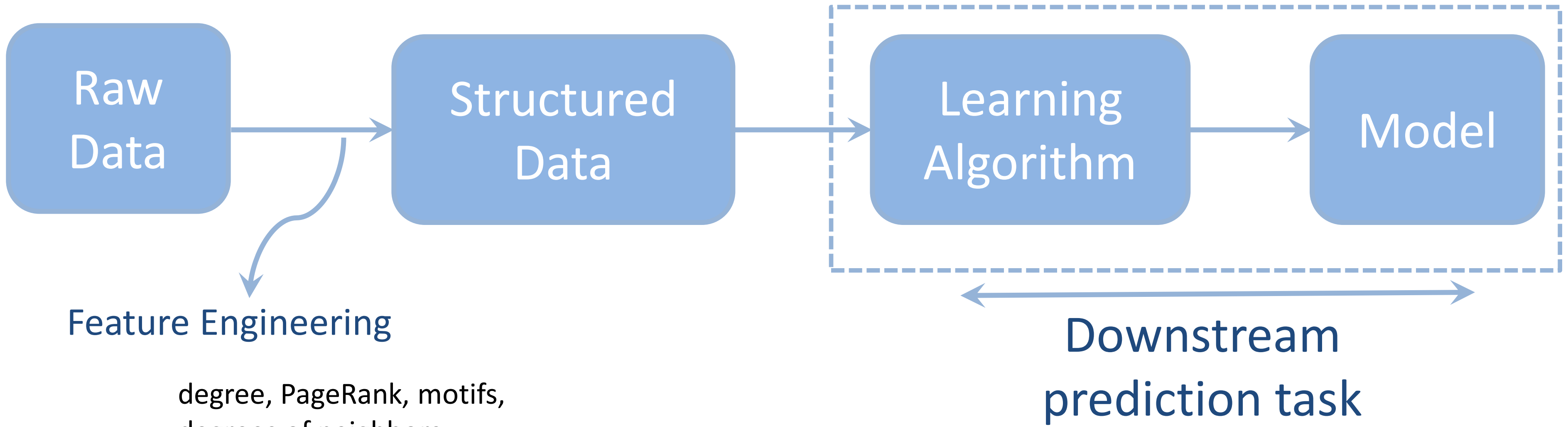


Image from: Perozzi et al.. DeepWalk: Online Learning of Social Representations. KDD 2014.

Graph embeddings: why?

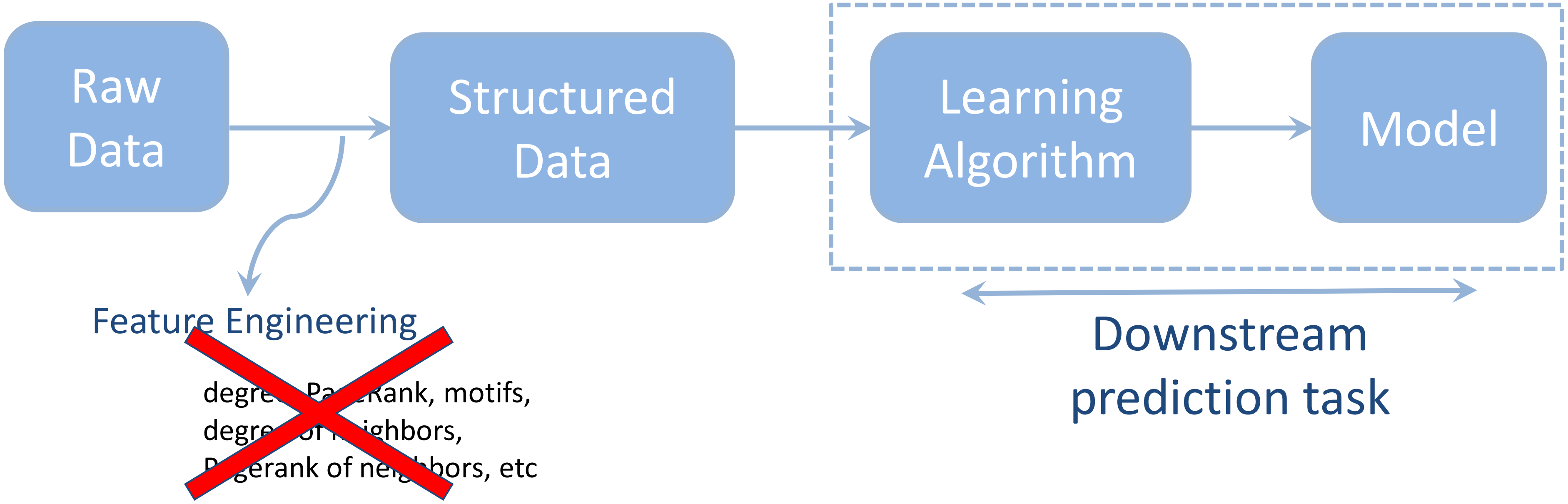
Machine learning lifecycle



degree, PageRank, motifs,
degrees of neighbors,
Pagerank of neighbors, etc

Graph embeddings: why?

Machine learning lifecycle

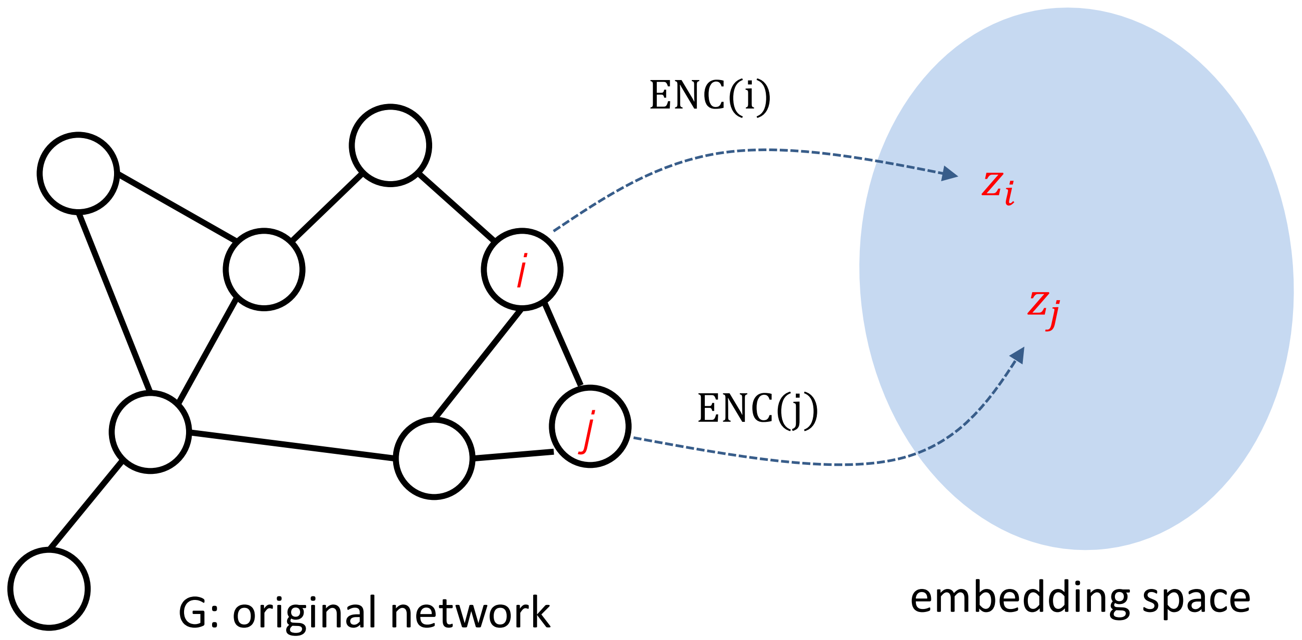

Raw Data
Structured Data
Learning Algorithm
Model
Feature Engineering
~~degree, PageRank, motifs, degree of neighbors, Pagerank of neighbors, etc~~
Downstream prediction task

Automatically learn the features (embeddings)

Embedding nodes

Input: Graph G(V, E)

Goal: encode nodes so that *similarity in the embedding space* approximates similarity in the original network.

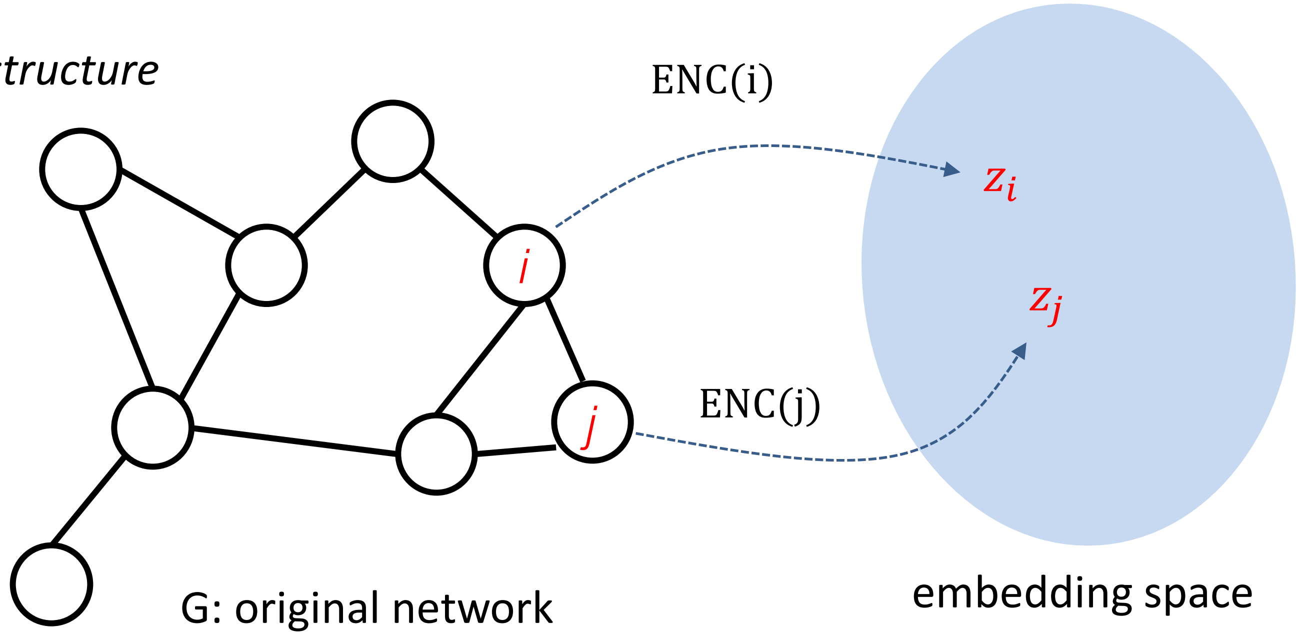


Embedding nodes

Goal: $\boxed{\text{similarity(i, j)}} \approx \boxed{z_i \cdot z_j}$

to be defined
how relationships in vector space map to relationships in the original network
encode structure

dot product (other definitions possible)



Learning node embeddings

1. Define an encoder that maps nodes to low dimensional spaces
2. Define *a node similarity function* in the original network.
3. Optimize the parameters of the encoder so that we minimize *a loss function* $\boldsymbol{L}$ that looks (roughly) like:

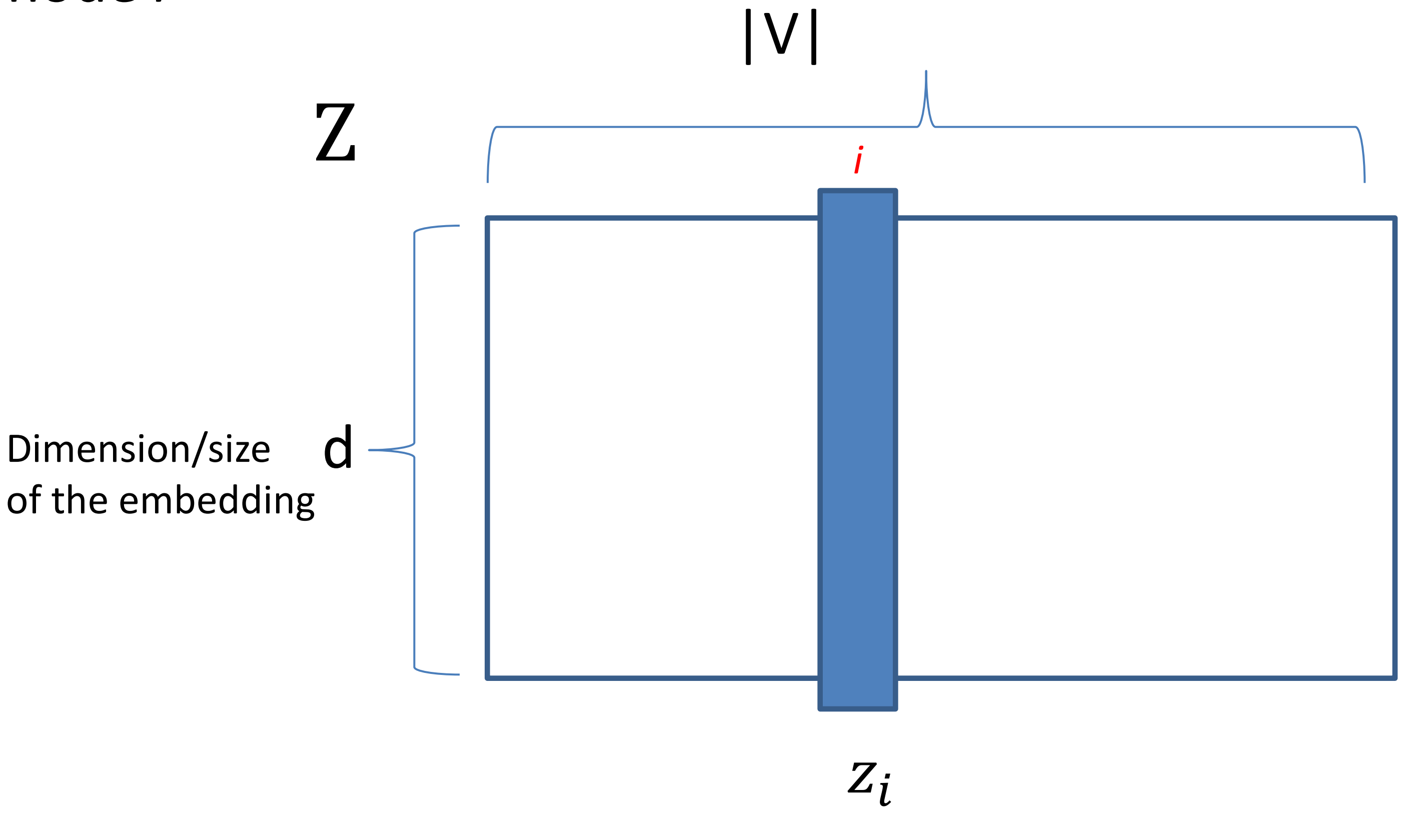
$$\boldsymbol{L} = \sum_{i,j \in V} (similarity(i,j) - z_i \cdot z_j)^2$$

When are two nodes similar? Any ideas?

Shallow embeddings(*)

Each node is assigned *a single d-dimensional vector*

Learn embedding matrix Z: each column *i* is the embedding z_i of node *i*

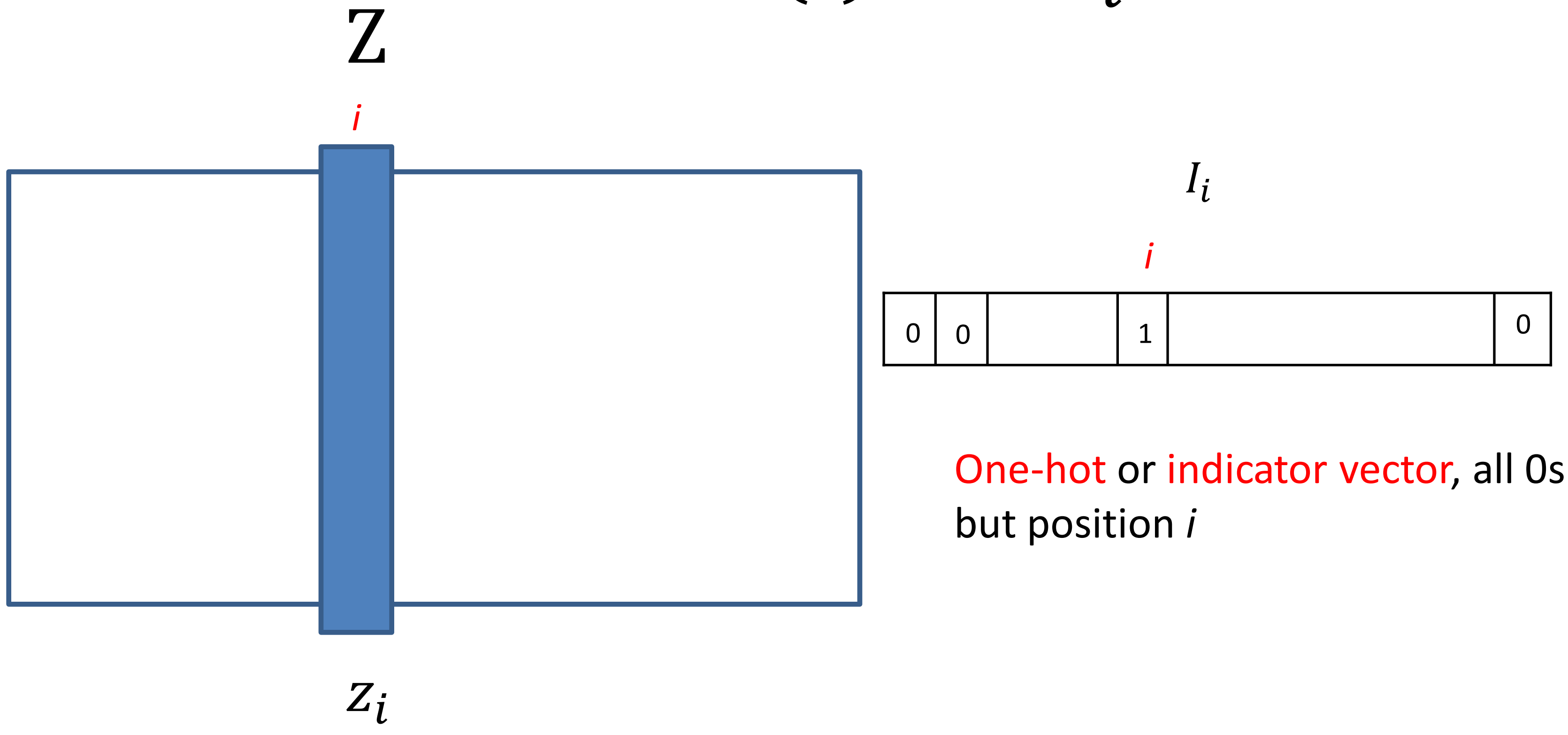


(*) As opposed to deep learning in graphs (neural networks embeddings)

Shallow embeddings

Encoder is an embedding lookup

$$ENC(i) = Z\, I_i$$



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

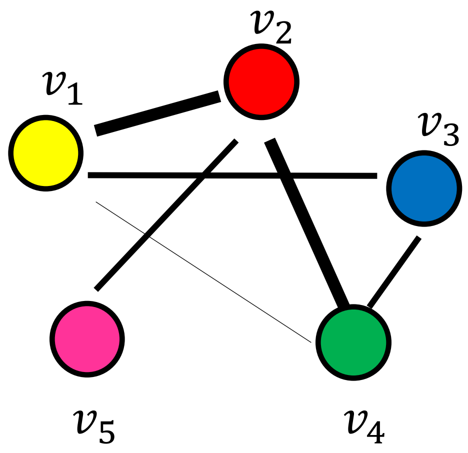
Node embeddings

Three approaches based on:

- Adjacency matrix
- Multi-hop neighborhoods
  - HOPE
  - GraRep

  Background on word2vec
- Random-walks
  - DeepWalk
  - Node2Vec

Adjacency-based approach



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

- *Similarity function* is just the edge (weight) between *u* and *v* in the original network.
- Dot products between node embeddings approximate *edge existence*.

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

Adjacency-based approach

The loss that what we want to minimize

embedding similarity

$$L = \sum_{i,j \in V \times V} \| A_{ij} - z_i \cdot z_j \|^2$$

sum over all node pairs

(weighted) adjacency matrix for the graph

How to minimize loss

1. Matrix decomposition (for example, SVD decomposition)
    1. Scalability issues
    2. Produced matrices that are very dense

2. Stochastic gradient descent

Singular Value Decomposition

$$A = U \quad \Sigma \quad V^T = \begin{bmatrix} \vec{u}_1 & \vec{u}_2 & \cdots & \vec{u}_r \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_r \end{bmatrix} \begin{bmatrix} \vec{v}_1 \\ \vec{v}_2 \\ \vdots \\ \vec{v}_r \end{bmatrix}$$

$$[n \times r]\,[r \times r]\,[r \times n]$$

- **r** : rank of matrix A
- $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$: singular values (square roots of eigenvals AA^T, A^TA)
- $\vec{u}_1, \vec{u}_2, \cdots, \vec{u}_r$: left singular vectors (eigenvectors of AA^T)
- $\vec{v}_1, \vec{v}_2, \cdots, \vec{v}_r$: right singular vectors (eigenvectors of A^TA)

$$A_r = \sigma_1 \vec{u}_1 \vec{v}_1^T + \sigma_2 \vec{u}_2 \vec{v}_2^T + \cdots + \sigma_r \vec{u}_r \vec{v}_r^T$$

Adjacency-based approach – stochastic gradient descent

A few manipulations

$$L = \sum_{i,j \in V \times V} ||A_{ij} - z_i \cdot z_j||^2$$

sum over all node pairs

$$L = \sum_{(i,j) \in E} (A_{ij} - z_i \cdot z_j)^2$$

sum over all edges

$$L = \frac{1}{2} \sum_{(i,j) \in E} (A_{ij} - z_i \cdot z_j)^2 + \frac{\lambda}{2} \sum_i ||z_i||^2$$

regularization factor

Adjacency-based approach

$$L = \frac{1}{2}\sum_{(i,j)\,\in\, E}(A_{ij} - z_i \cdot z_i)^2 + \frac{\lambda}{2}\sum_i ||z_i||^2$$

Taking the gradient

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

$$\frac{\partial L}{\partial z_i} = -\sum_{j\,\in\, N(i)}\left(A_{ij} - z_j \cdot z_i\right) z_j + \lambda\, z_i$$

For each edge (i, j) ∈ E this amounts for

$$\frac{\partial L}{\partial z_i} = -\ (A_{ij} - z_i \cdot z_j)\, z_j + \lambda\, z_i$$

Adjacency-based approach

```
Requires: Adjacency matrix $A$, rank $d$, accuracy $\varepsilon$
Ensures: Local minimum
1:  Initialize Z' at random
2:  t ← 1
3;  repeat
4:      Z ← Z'
5:      for all edges (i, j) ∈ E do
6:          $\eta \leftarrow 1/\sqrt{t}$
7:          t ← t +1
8:          $Z_i \leftarrow Z_i + \eta ((A_{ij} - \langle Z_i \cdot Z_j \rangle Z_j) + \lambda Z_i)$
9:      end for
10: until $||Z - Z'||^2 <= \varepsilon$
11: return Z
```

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

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

Node embeddings

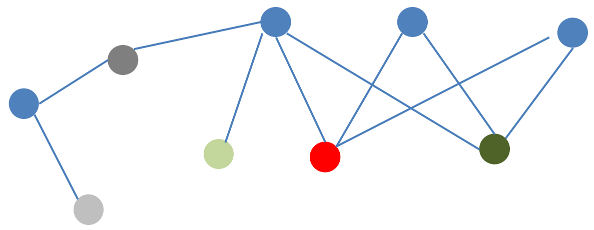
Three approaches based on:

- Adjacency matrix
- Multi-hop neighborhoods
  - HOPE
  - GraRep
- Random-walks
  - DeepWalk
  - Node2Vec

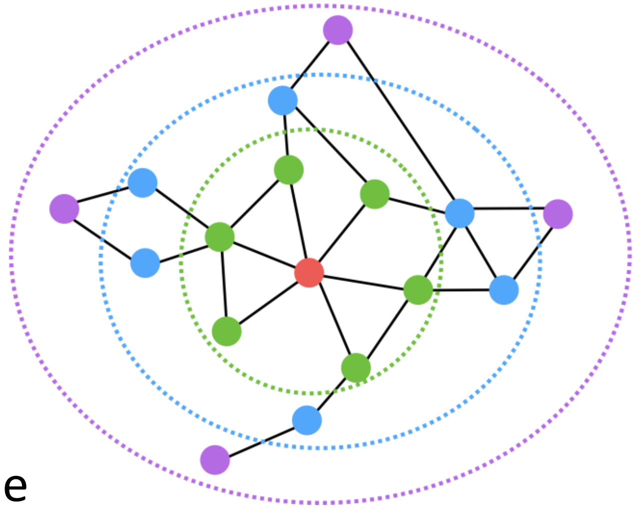
Multi-hop approaches

Only considers direct connections

What about further neighbors?



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



We will see two approaches

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

High-order Proximity Preserved Embeddings (HOPE)

Based on a high order proximity matrix S,

S_{ij} = proximity(i j)

Learn two embeddings vectors

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

$$L = \sum_{(i,j) \in V \times V} ||S_{ij} - z_i^s \cdot z_j^t||^2$$

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

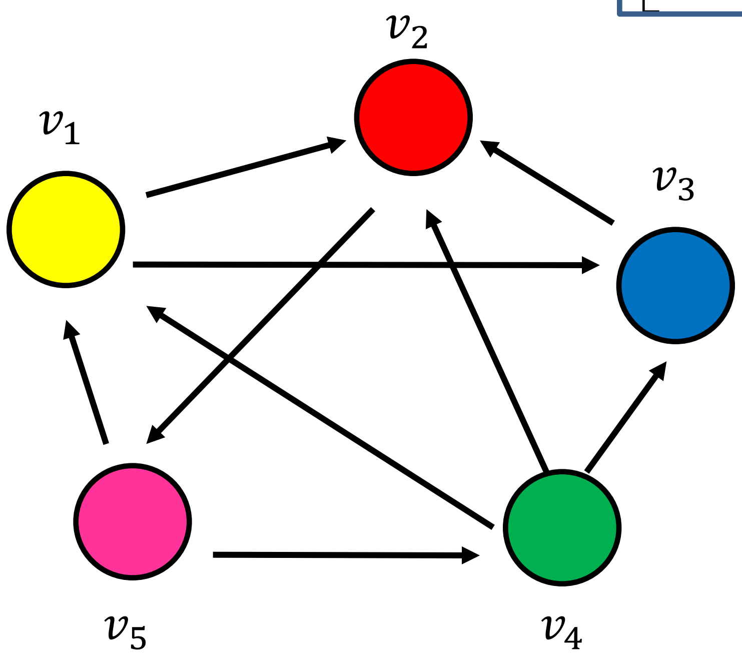
HOPE

Local High Order Proximity

Common Neighbors (for directed graphs, source-target)

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

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



Adamic-Adar

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

Similar but assigns a weight to the neighbor = reciprocal of its degree
The more vertexes a node is connected to, the less important it is on evaluating the proximity between a pair of nodes

HOPE

Global High Order Proximity

Katz

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

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

Rooted Pagerank

SVD with some tricks to save computations

(Thanks to Philipp Koehn for the material borrowed from his slides)

INTRODUCTION TO NEURAL NETWORKS

Classification

- **Classification** is the task of ***learning a target function* f** that maps attribute set **x** to one of the predefined class labels **y**

	categorical	categorical	continuous	class
Tid	**Refund**	**Marital Status**	**Taxable Income**	**Cheat**
1	Yes	Single	125K	**No**
2	No	Married	100K	**No**
3	No	Single	70K	**No**
4	Yes	Married	120K	**No**
5	No	Divorced	95K	**Yes**
6	No	Married	60K	**No**
7	Yes	Divorced	220K	**No**
8	No	Single	85K	**Yes**
9	No	Married	75K	**No**
10	No	Single	90K	**Yes**

One of the attributes is the class attribute
In this case: Cheat

Two class labels (or classes): Yes (1), No (0)

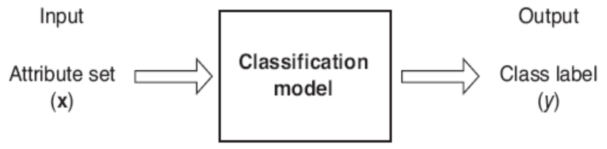
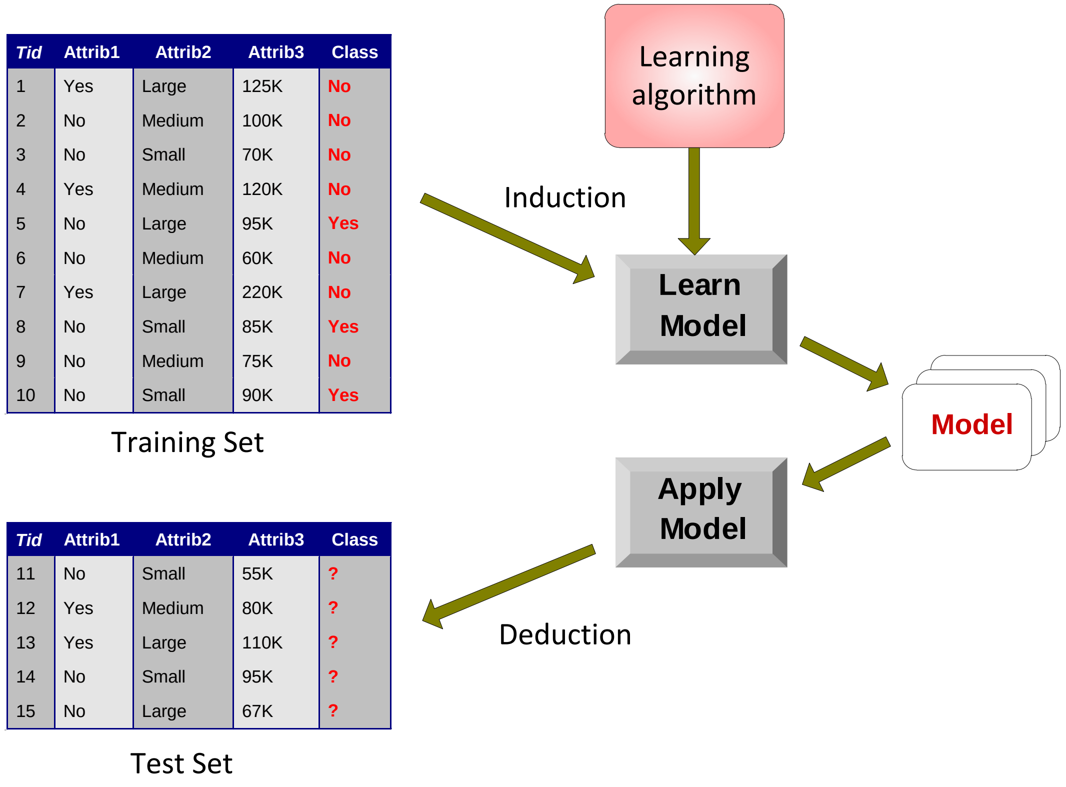


Figure 4.2. Classification as the task of mapping an input attribute set $\mathbf{x}$ into its class label y.

Illustrating Classification Task



Tid	Attrib1	Attrib2	Attrib3	Class
1	Yes	Large	125K	No
2	No	Medium	100K	No
3	No	Small	70K	No
4	Yes	Medium	120K	No
5	No	Large	95K	Yes
6	No	Medium	60K	No
7	Yes	Large	220K	No
8	No	Small	85K	Yes
9	No	Medium	75K	No
10	No	Small	90K	Yes

Training Set

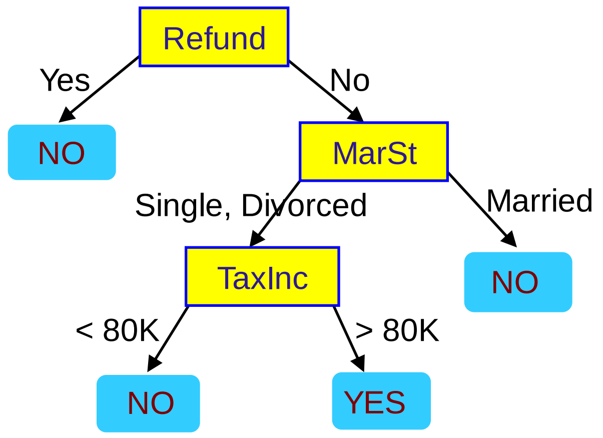
Tid	Attrib1	Attrib2	Attrib3	Class
11	No	Small	55K	?
12	Yes	Medium	80K	?
13	Yes	Large	110K	?
14	No	Small	95K	?
15	No	Large	67K	?

Test Set

Example of a Model

Tid	Refund categorical	Marital Status categorical	Taxable Income continuous	Cheat class
1	Yes	Single	125K	**No**
2	No	Married	100K	**No**
3	No	Single	70K	**No**
4	Yes	Married	120K	**No**
5	No	Divorced	95K	**Yes**
6	No	Married	60K	**No**
7	Yes	Divorced	220K	**No**
8	No	Single	85K	**Yes**
9	No	Married	75K	**No**
10	No	Single	90K	**Yes**

Training Data



Model: Decision Tree

Classification in Networks

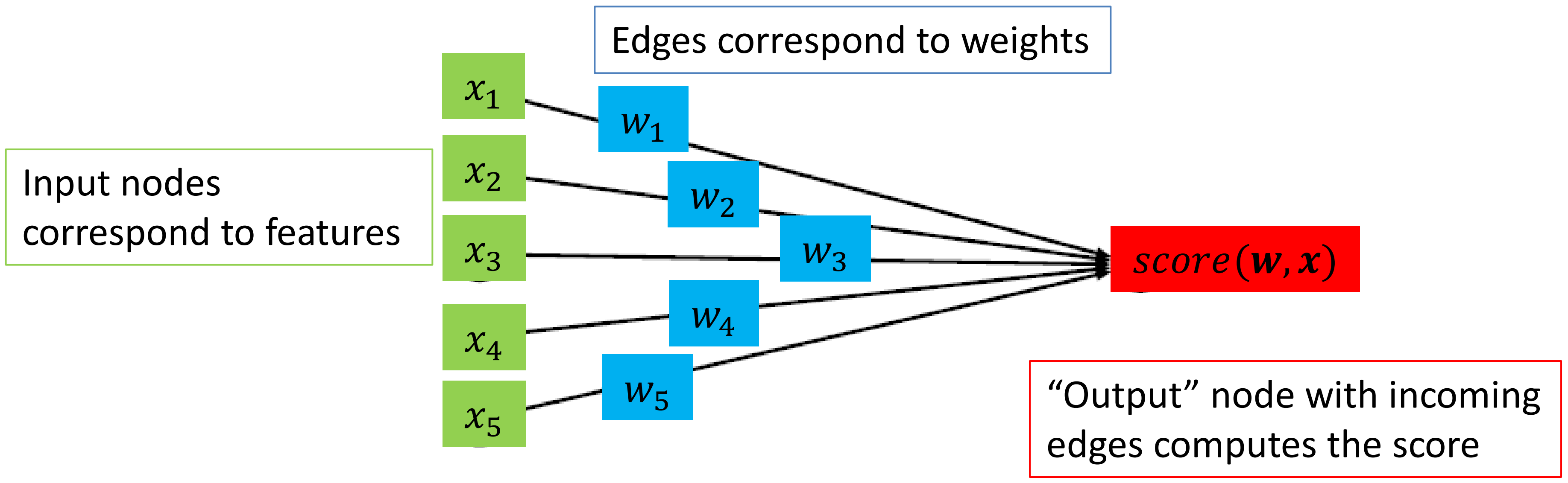
- There are various problems in network analysis that can be mapped to a classification problem:
  - Link prediction: Predict 0/1 for missing edges, whether they will appear or not in the future.
  - Node classification: Classify nodes as democrat-republican/spammers-legitimate/other categories
    - Use node features but also neighborhood and structural features
    - Label propagation
  - Edge classification: Classify edges according to type (professional/family relationships), or according to strength.
  - More…
- Recently all of this is done using Neural Networks.

Linear Classification

- A simple model for classification is to take a linear combination of the feature values and compute a score.
- Input: Feature vector $\boldsymbol{x} = (x_1, \ldots, x_n)$
- Model: Weights $\boldsymbol{w} = (w_1, \ldots, w_n)$
- Output: $score(\boldsymbol{w}, \boldsymbol{x}) = \sum_i w_i x_i$
- Make a decision depending on the output score.
  – E.g.: Decide "Yes" if $score(\boldsymbol{w}, \boldsymbol{x}) > 0$ and "No" if $score(\boldsymbol{w}, \boldsymbol{x}) < 0$

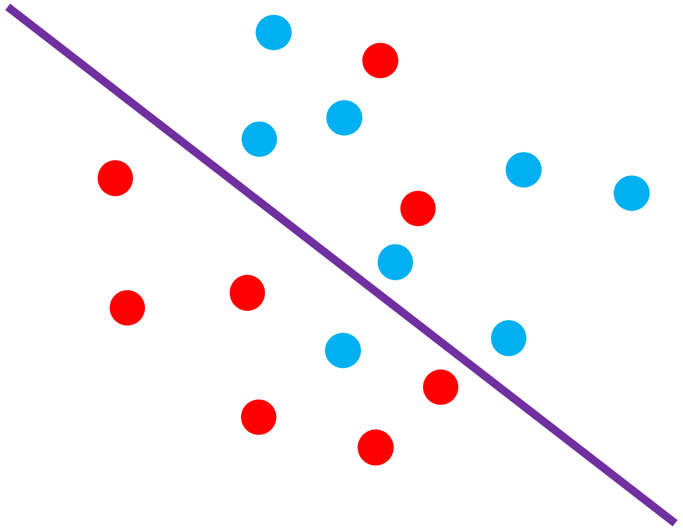
Linear Classification

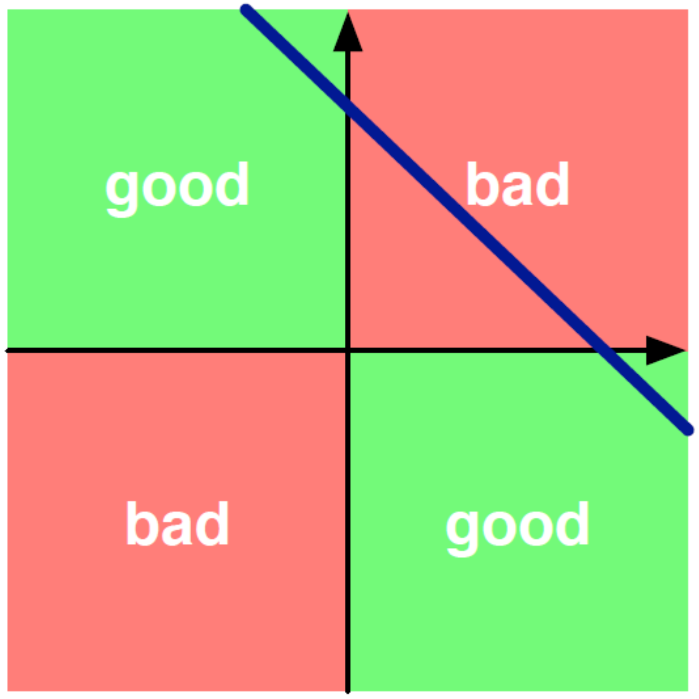
- We can represent this as a network


Edges correspond to weights
Input nodes correspond to features
x_1
x_2
x_3
x_4
x_5
w_1
w_2
w_3
w_4
w_5
$score(\boldsymbol{w}, \boldsymbol{x})$
"Output" node with incoming edges computes the score

Linear models

- Linear models partition the space according to a hyperplane

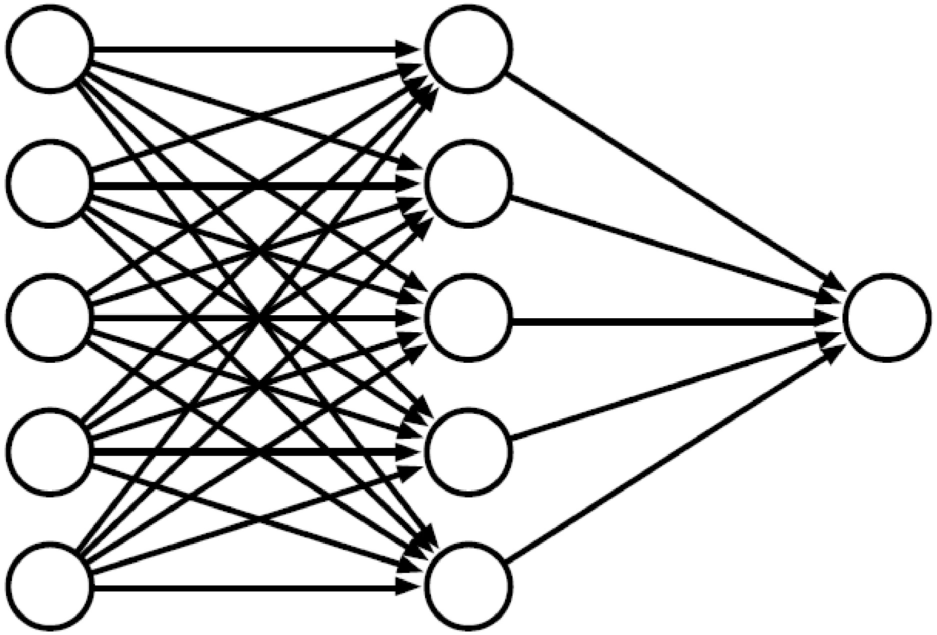




- But they cannot model everything

Multiple layers

- We can add more layers:
  - Each arrow has a weight
  - Nodes compute scores from incoming edges and give input to outgoing edges



Did we gain anything?

Non-linearity

- Instead of computing a linear combination

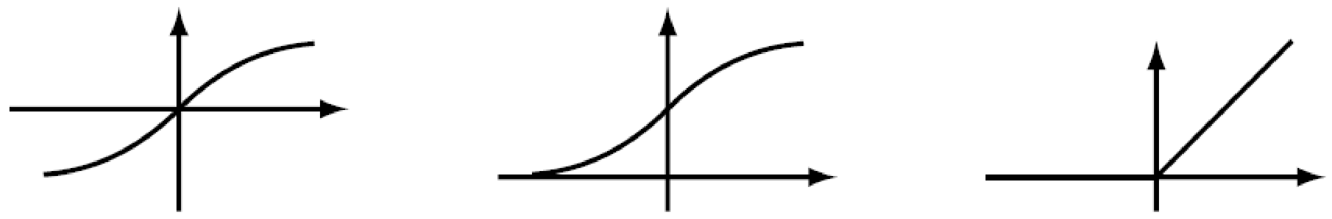
$$score(\boldsymbol{w}, \boldsymbol{x}) = \sum_i w_i x_i$$

- Apply a non-linear function on top:

$$score(\boldsymbol{w}, \boldsymbol{x}) = g\left(\sum_i w_i x_i\right)$$

- Popular functions:

$\tanh(x)$ $\quad$ $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$ $\quad$ $\text{relu}(x) = \max(0, x)$

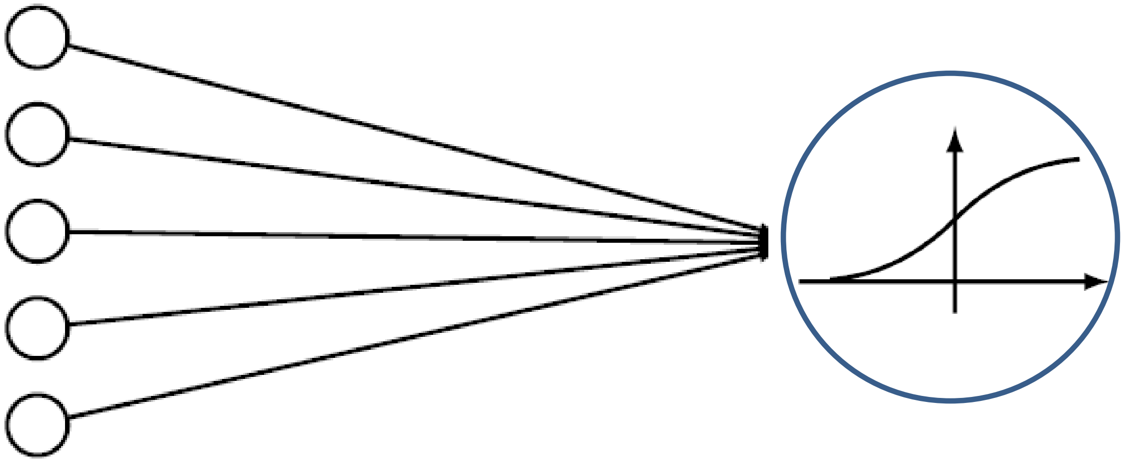


(sigmoid is also called the "logistic function")

These functions play the role of a soft "switch" (threshold function)

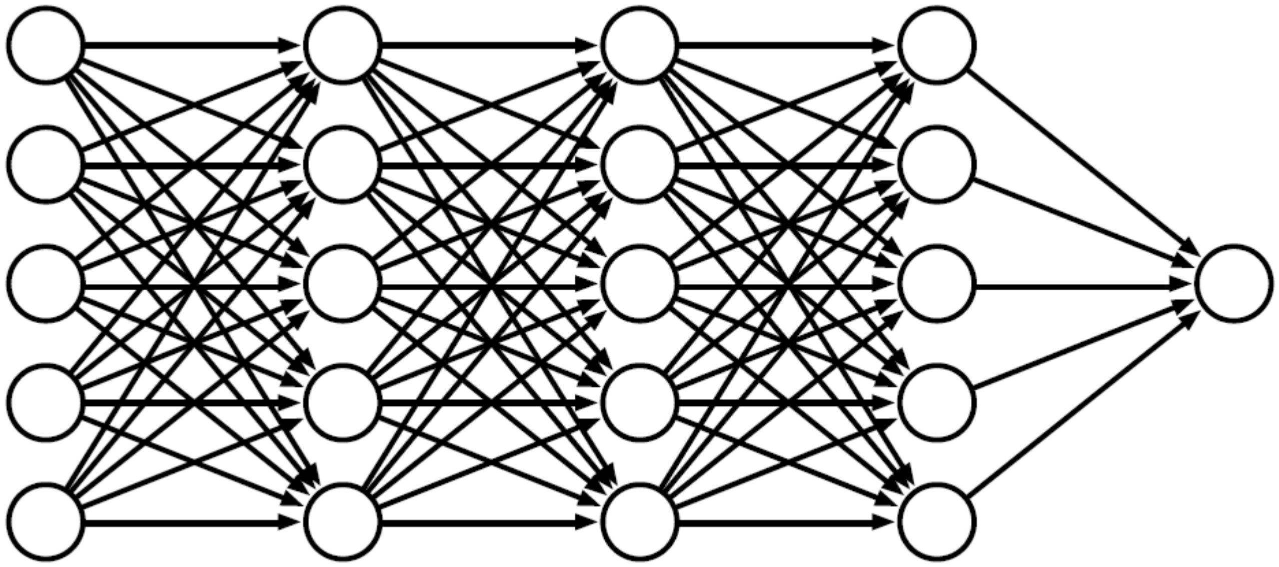
Side note

- Logistic regression classifier:
  - Single layer with a logistic function



Deep learning

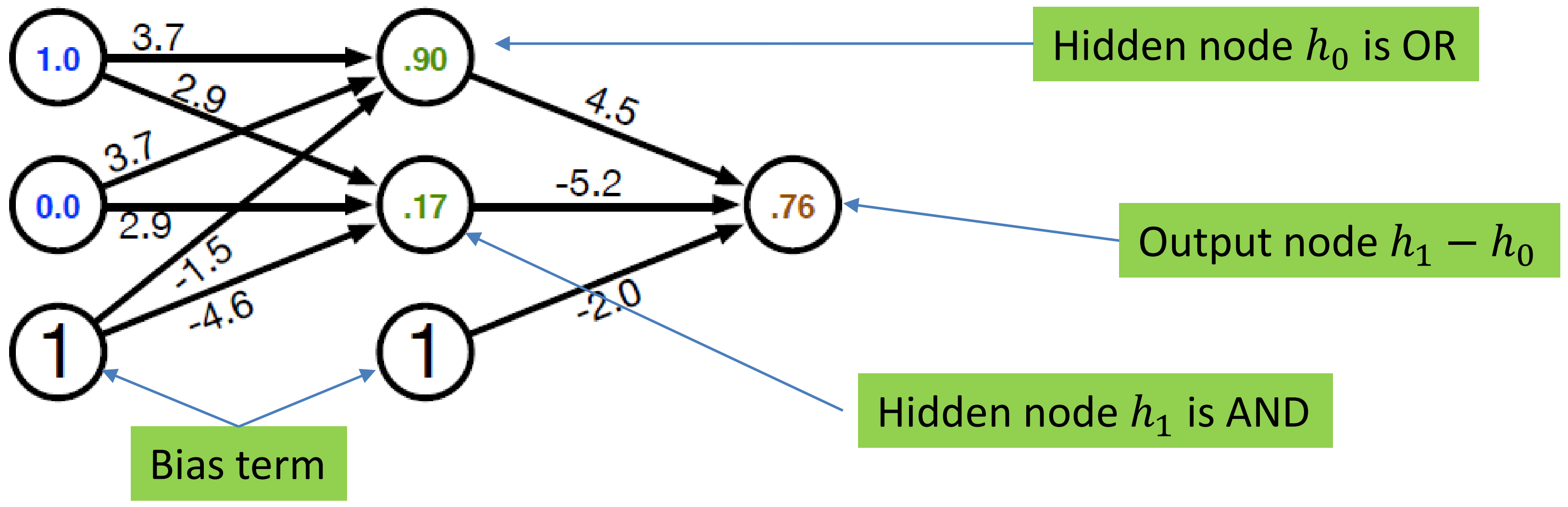
- Networks with multiple layers



- Each layer can be thought of as a processing step
- Multiple layers allow for the computation of more complex functions

Example

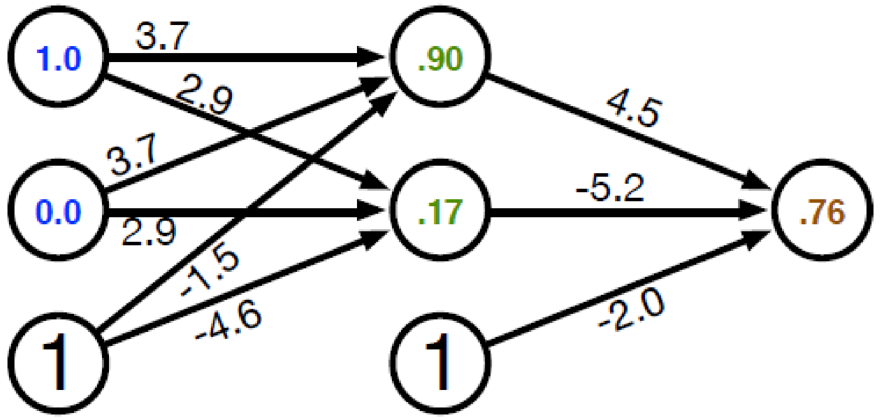
- A network that implements XOR



Input x_0	Input x_1	Hidden h_0	Hidden h_1	Output y_0
0	0	0.12	0.02	$0.18 \to 0$
0	1	0.88	0.27	$0.74 \to 1$
1	0	0.73	0.12	$0.74 \to 1$
1	1	0.99	0.73	$0.33 \to 0$

Error

- The computed value is 0.76 but the correct value is 1
  - There is an error in the computation

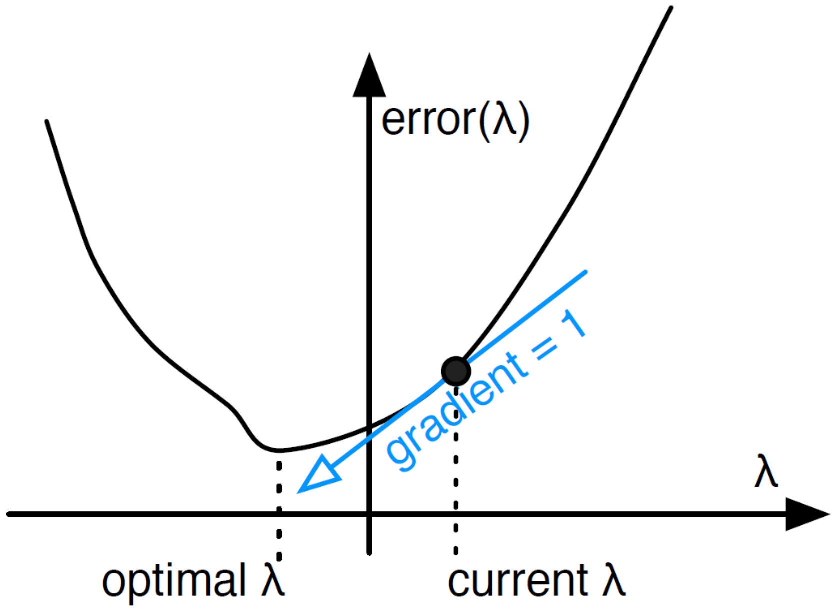


  - How do we set the weights so as to minimize this error?

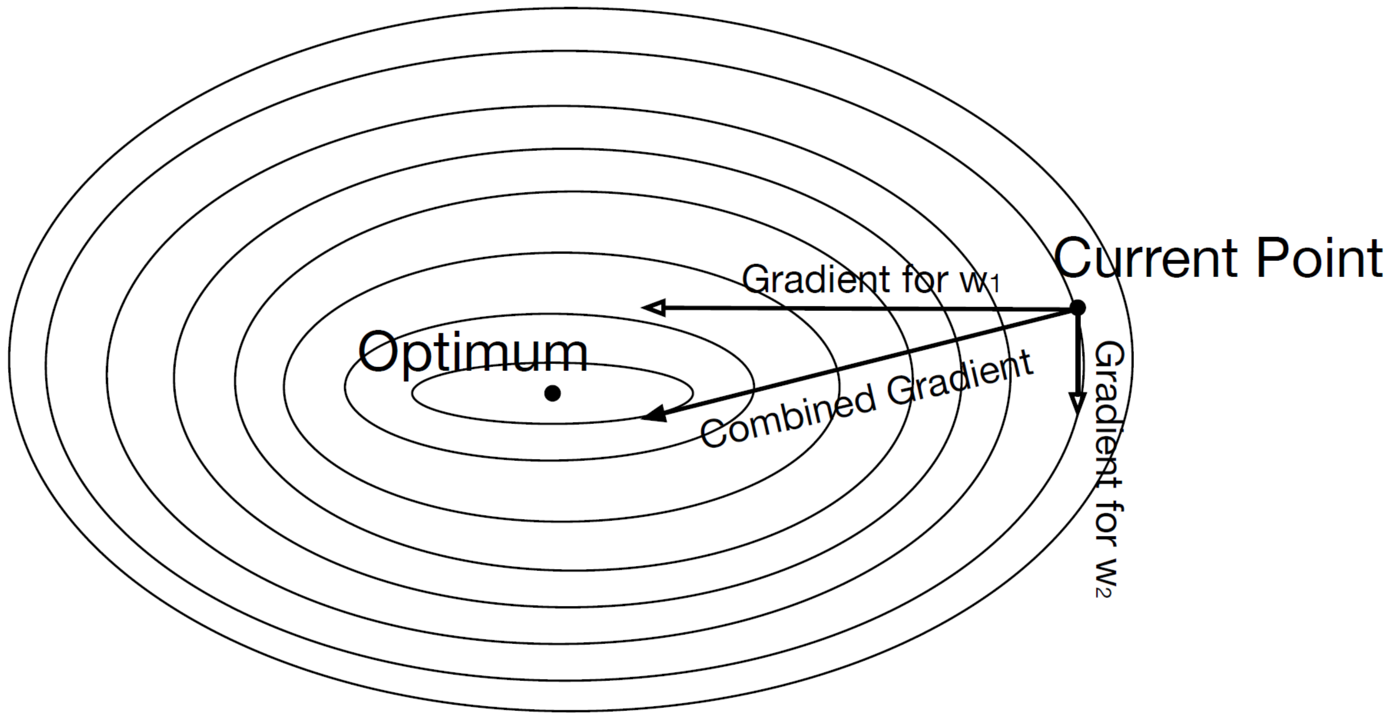
Gradient Descent

- The error is a function of the weights
- We want to find the weights that minimize the error
- Compute gradient: gives the direction to the minimum
- Adjust weights, moving at the direction of the gradient.

Gradient Descent

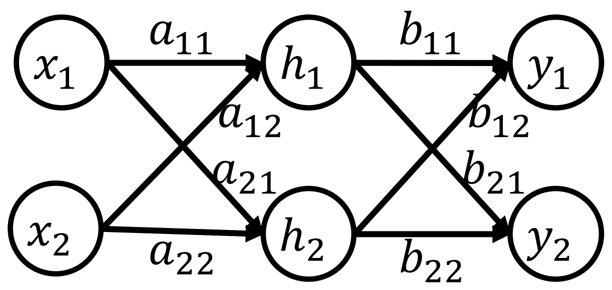


Gradient Descent



Backpropagation

- How can we compute the gradients? Backpropagation!
- Main idea:
  - Start from the final layer: compute the gradients for the weights of the final layer.
  - Use these gradients to compute the gradients of previous layers using the chain rule
  - Propagate the error backwards
- Backpropagation essentially is an application of the chain rule for differentiation.



Error: $E = \|y - t\|^2 = (y_1 - t_1)^2 + (y_2 - t_2)^2$

$$\frac{\partial E}{\partial b_{11}} = \frac{\partial E}{\partial s_{y_1}} \frac{\partial s_{y_1}}{\partial b_{11}} = \delta_{y_1} h_1$$

$$\boxed{\delta_{y_1} = \frac{\partial E}{\partial s_{y_1}} = \frac{\partial E}{\partial y_1} \frac{\partial y_1}{\partial s_{y_1}} = 2(y_1 - t_1) g'(s_{y_1})}$$

Notation:

Activation function: g

$s_{y_1} = b_{11} h_1 + b_{12} h_2$, $y_1 = g(s_{y_1})$

$s_{y_2} = b_{21} h_1 + b_{22} h_2$, $y_2 = g(s_{y_2})$

$s_{h_1} = a_{11} x_1 + a_{12} x_2$, $h_1 = g(s_{h_1})$

$s_{h_2} = a_{21} x_1 + a_{22} x_2$, $h_2 = g(s_{h_2})$

$$\frac{\partial E}{\partial b_{21}} = \delta_{y_2} h_1 \qquad \boxed{\delta_{y_2} = \frac{\partial E}{\partial s_{y_2}} = 2(y_2 - t_2) g'(s_{y_2})}$$

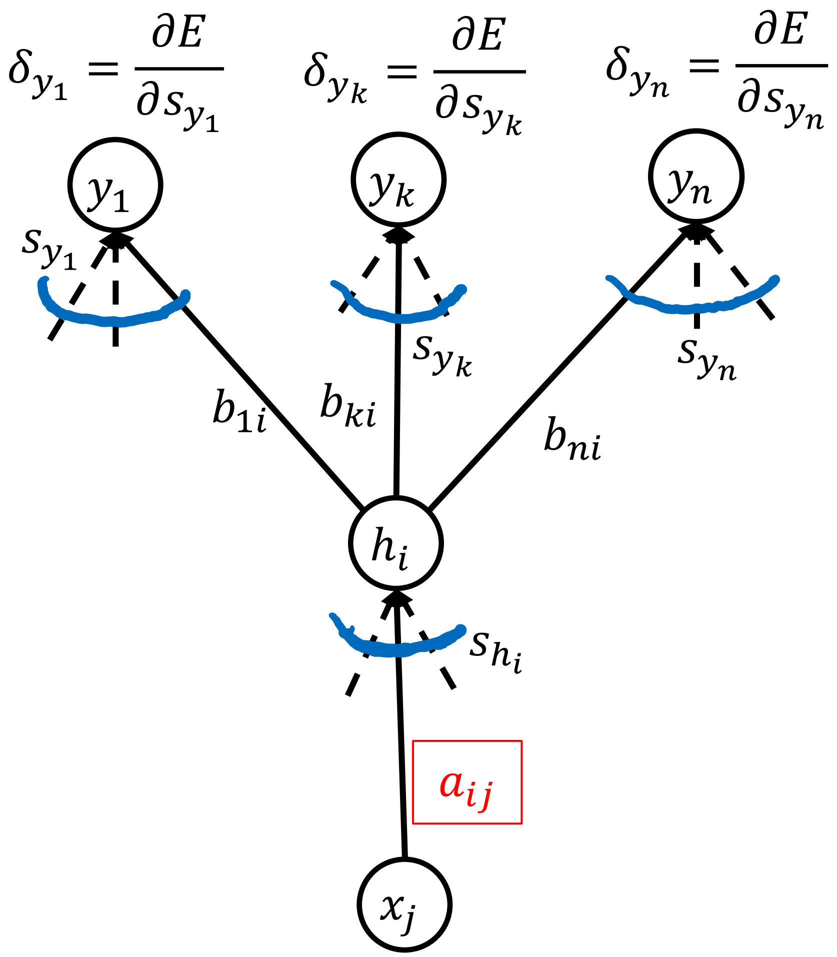
$$\frac{\partial E}{\partial b_{12}} = \delta_{y_1} h_2 \qquad \frac{\partial E}{\partial b_{22}} = \delta_{y_2} h_2$$

$$\frac{\partial E}{\partial a_{11}} = \frac{\partial E}{\partial s_{h_1}} \frac{\partial s_{h_1}}{\partial a_{11}} = \delta_{h_1} x_1 \qquad \frac{\partial E}{\partial a_{22}} = \frac{\partial E}{\partial s_{h_2}} \frac{\partial s_{h_2}}{\partial a_{22}} = \delta_{h_2} x_2 \qquad \frac{\partial E}{\partial a_{21}} = \delta_{h_1} x_2 \qquad \frac{\partial E}{\partial a_{12}} = \delta_{h_2} x_1$$

$$\delta_{h_1} = \frac{\partial E}{\partial s_{h_1}} = \frac{\partial E}{\partial h_1} \frac{\partial h_1}{\partial s_{h_1}} = \left(\frac{\partial E}{\partial s_{y_1}} \frac{\partial s_{y_1}}{\partial h_1} + \frac{\partial E}{\partial s_{y_2}} \frac{\partial s_{y_2}}{\partial h_1} \right) g'(s_{h_1}) = (\delta_{y_1} b_{11} + \delta_{y_2} b_{21}) g'(s_{h_1})$$

$$\delta_{h_2} = (\delta_{y_1} b_{12} + \delta_{y_2} b_{22}) g'(s_{h_2})$$

Backpropagation



$\delta_{y_1} = \frac{\partial E}{\partial s_{y_1}}$
$\delta_{y_k} = \frac{\partial E}{\partial s_{y_k}}$
$\delta_{y_n} = \frac{\partial E}{\partial s_{y_n}}$
y_1
y_k
y_n
s_{y_1}
s_{y_k}
s_{y_n}
b_{1i}
b_{ki}
b_{ni}
h_i
s_{h_i}
a_{ij}
x_j

$$\boxed{\frac{\partial E}{\partial a_{ij}} = \sum_{k=1}^{n} \delta_{y_k} b_{ki}\, g'(s_{h_i}) x_j}$$

For the sigmoid function:

$$g(x) = \frac{1}{1+e^{-x}}$$

The derivative is:

$$g'(x) = g(x)(1 - g(x))$$

This makes it easy to compute it. We have:

$$g'(s_{h_i}) = h_i(1 - h_i)$$

Stochastic gradient descent

- Ideally the loss should be the average loss over all training data.
- We would need to compute the loss for all training data every time we update the gradients.
  - However, this is expensive.
- Stochastic gradient descent: Consider one input point at the time. Each point is considered only once.
- Intermediate solution: Use mini-batches of data points.

WORD EMBEDDINGS

Basic Idea

- You can get a lot of value by representing a word by means of its neighbors
- "You shall know a word by the company it keeps"
  - (J. R. Firth 1957: 11)
- One of the most successful ideas of modern statistical NLP

government debt problems turning into banking crises as has happened in
saying that Europe needs unified banking regulation to replace the hodgepodge

↖ These words will represent *banking* ↗

Basic idea

Define a model that aims to predict between a center word w_c and context words in some window of length m in terms of word vectors

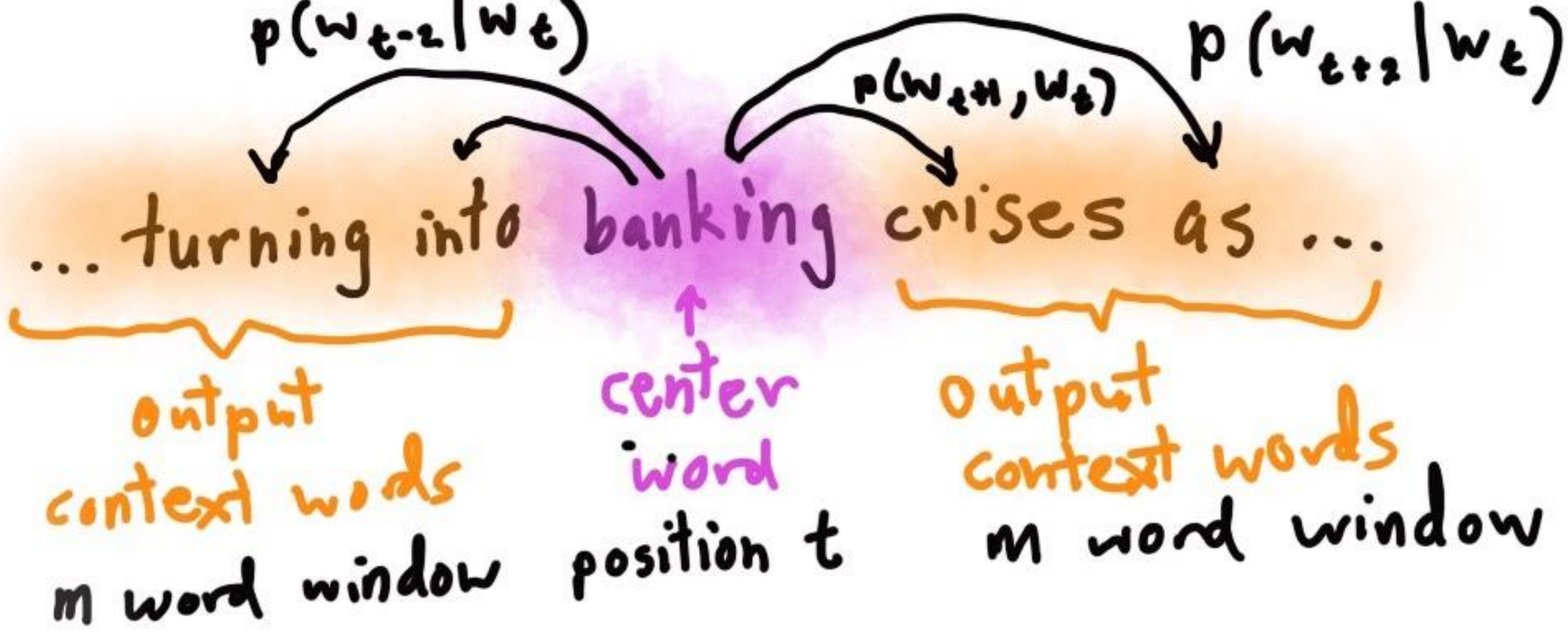
$$\mathrm{P}(w_c \mid w_{c-m,} \dots, w_{c-1,}\ w_{c+1} \dots, w_{c+m})$$

model)

Pairwise probabilities

Independence assumption (bigram model)

$$P(w_1, w_2, \dots, w_n) = \prod_{i=2}^{n} P(w_i \mid w_{i-1})$$


$p(w_{t-2} \mid w_t)$
$p(w_{t+2} \mid w_t)$
$p(w_{t+1}, w_t)$
… turning into banking crises as …
output context words
m word window
center word
position t
output context words
m word window

Word2Vec

Predict between every word and its context words

Two **algorithms**

1. Skip-grams (SG)

   Predict context words given the center word

2. Continuous Bag of Words (CBOW)

   Predict center word from a bag-of-words context

Position independent (do not account for distance from center)

Two **training methods**

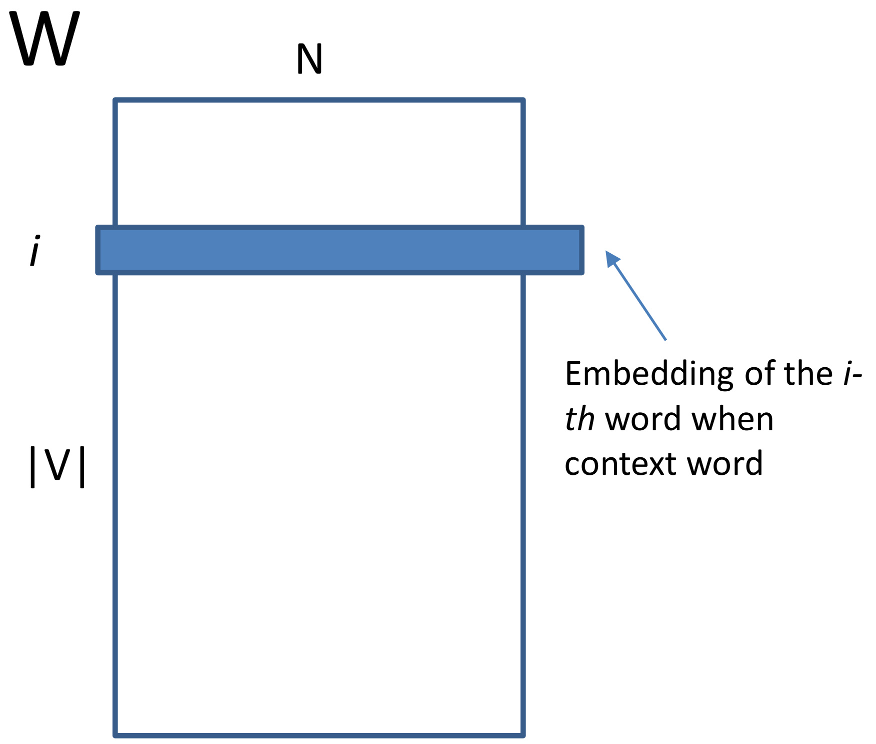
1. Hierarchical softmax
2. Negative sampling

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

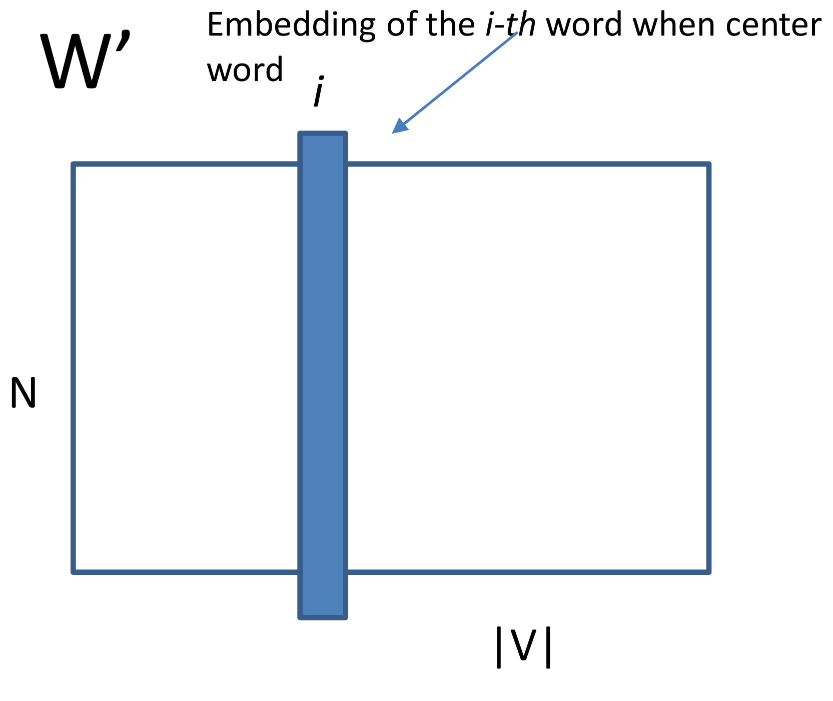
CBOW

Use a window of context words to predict the center word

Learn two matrices (N size of embedding, |V| number of words)



|V| x N context embeddings when *input*



N x |V| center embeddings when *output*

CBOW

Given window size *m*

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

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

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

2. Compute the *embeddings of the context words*

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

3. *Average* these vectors

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

4. Generate a *score vector*

z = W' $\hat{v}$

dot product, (embedding of center word) similar vectors close to each other

5. Turn the *score vector to probabilities*

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

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

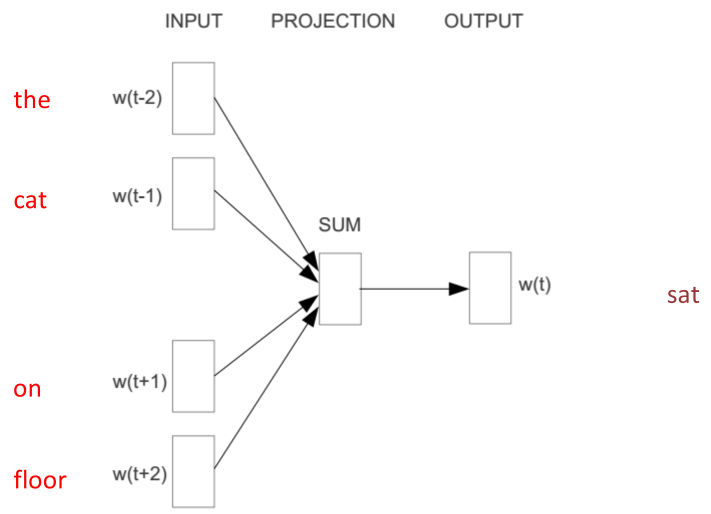
Softmax

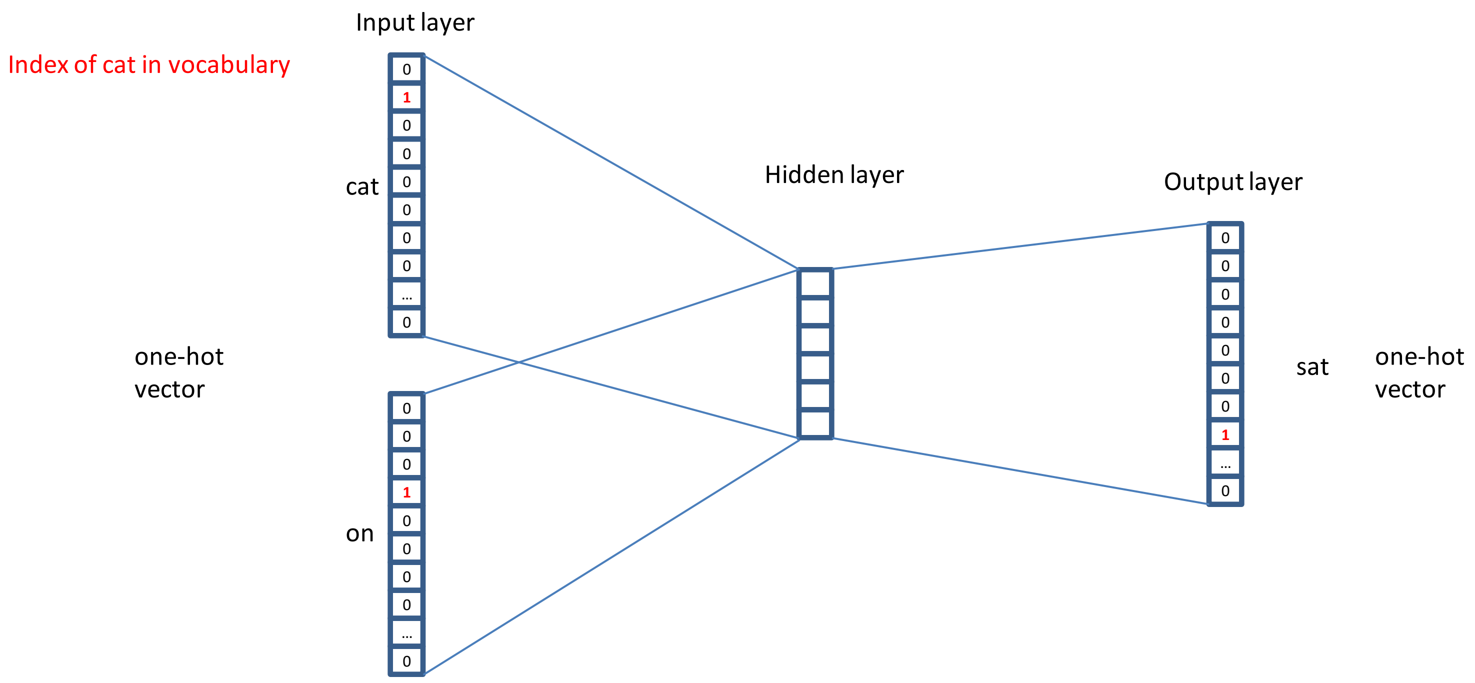
Exponentiate to make positive

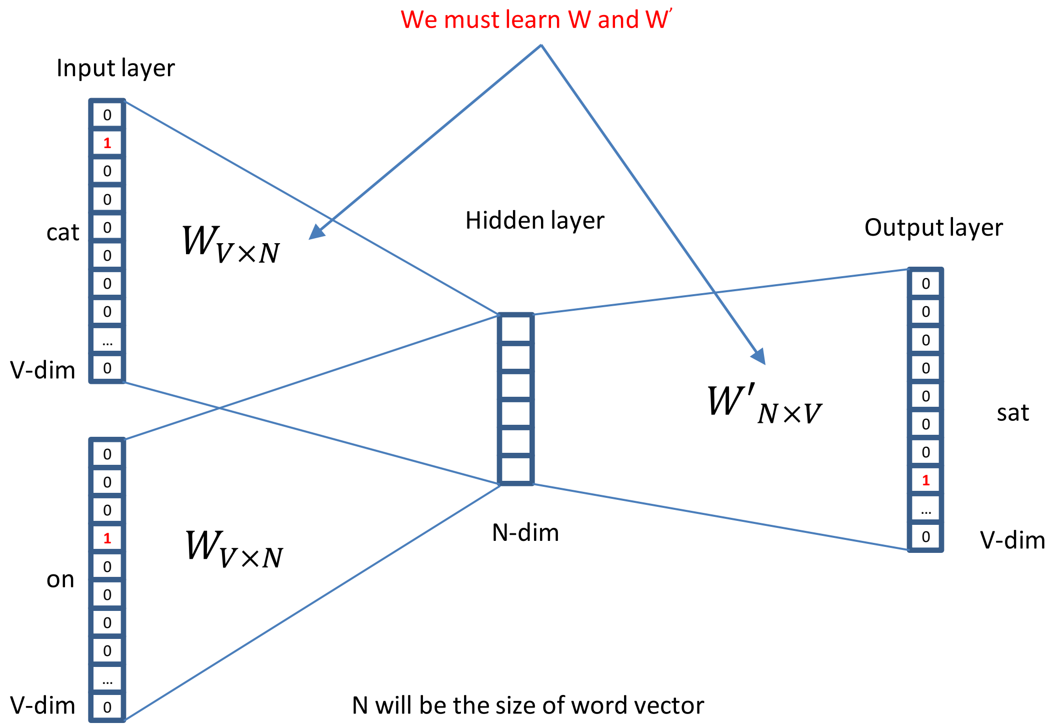
$$p_i = \frac{e^{u_i}}{\sum_j e^{u_j}}$$

Normalize to give probability

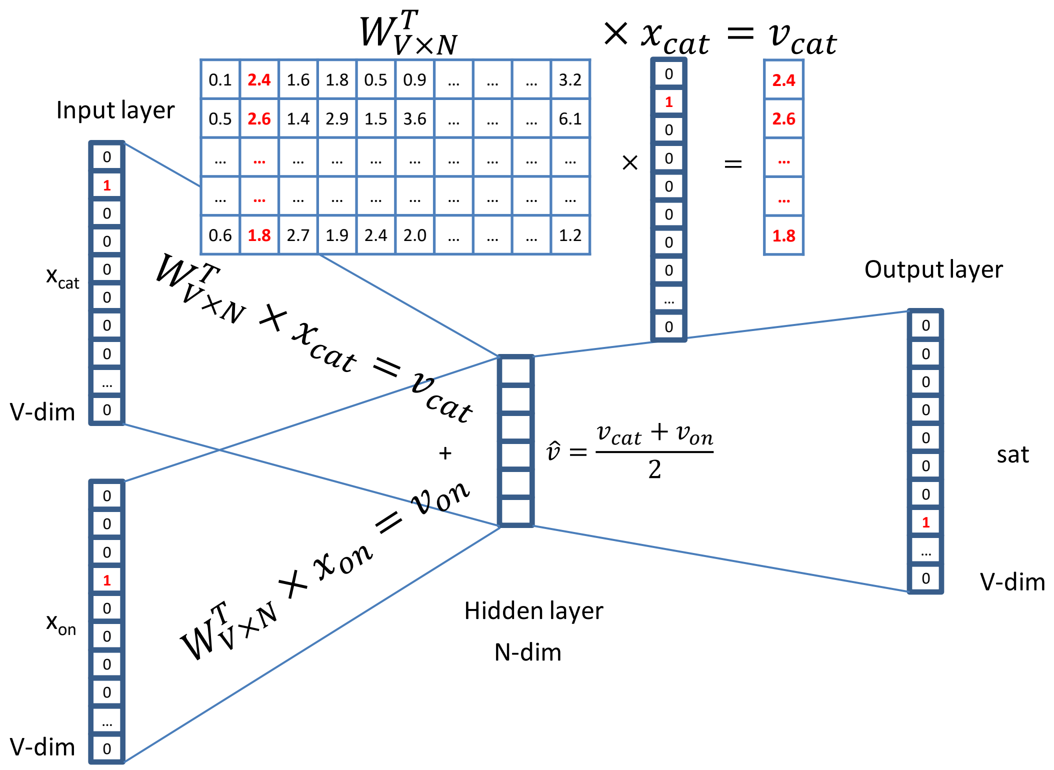
- E.g. “The cat sat on floor”
  - Window size = 2




Input layer
Index of cat in vocabulary
cat
0
1
0
0
0
0
0
0
...
0
Hidden layer
Output layer
one-hot
vector
on
0
0
0
1
0
0
0
0
...
0
sat
one-hot
vector
0
0
0
0
0
0
0
1
...
0



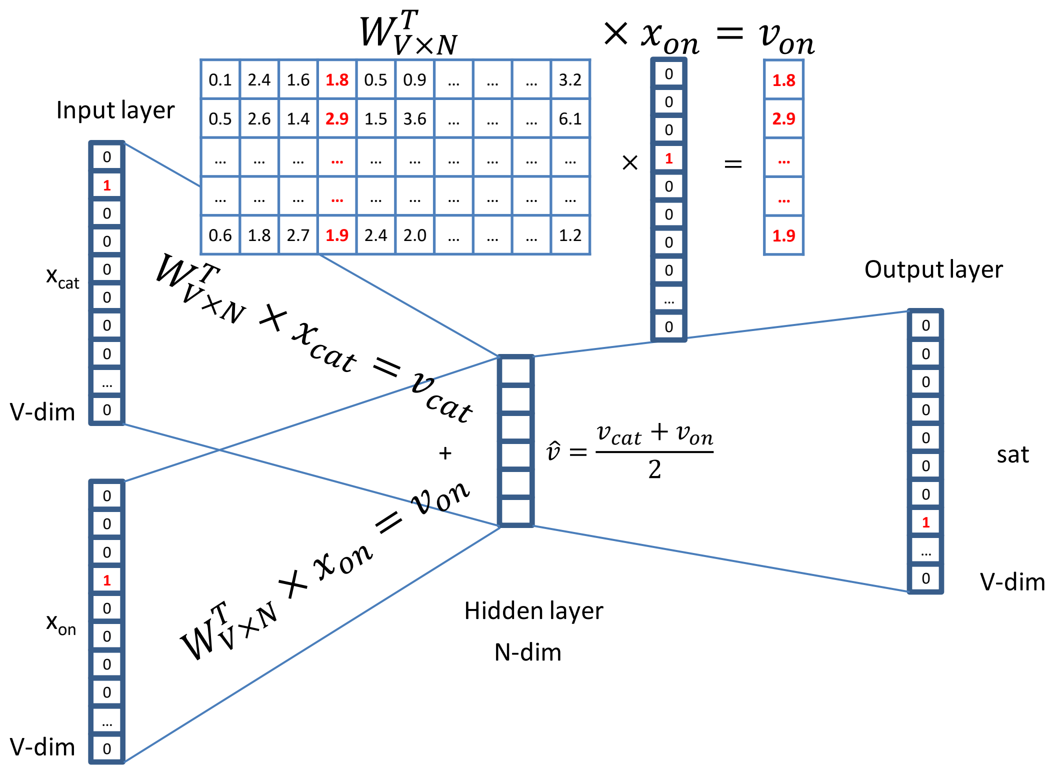
N will be the size of word vector



$W^T_{V \times N}$
$\times x_{cat} = v_{cat}$
Input layer
x_{cat}
V-dim
x_{on}
V-dim
$W^T_{V \times N} \times x_{cat} = v_{cat}$
+
$W^T_{V \times N} \times x_{on} = v_{on}$
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Hidden layer
N-dim
Output layer
sat
V-dim

0.1	2.4	1.6	1.8	0.5	0.9	…	…	…	3.2
0.5	2.6	1.4	2.9	1.5	3.6	…	…	…	6.1
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
0.6	1.8	2.7	1.9	2.4	2.0	…	…	…	1.2

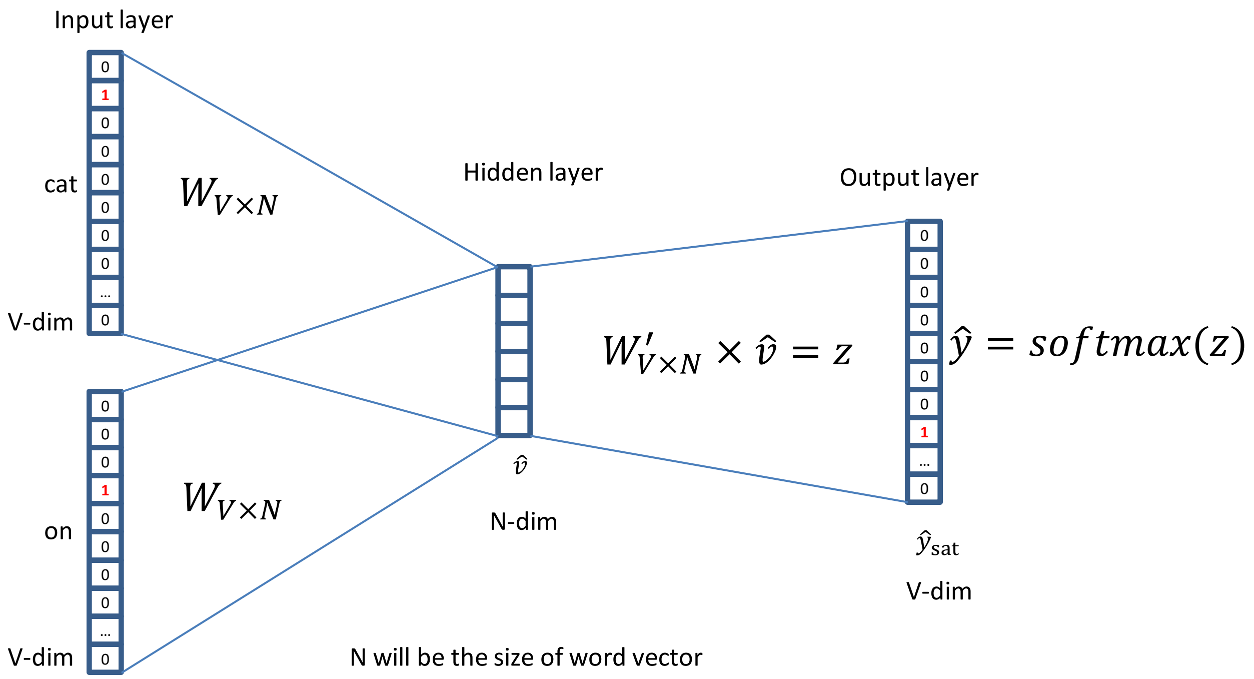
$\times$ (0, 1, 0, 0, 0, 0, 0, 0, …, 0) $=$ (2.4, 2.6, …, …, 1.8)



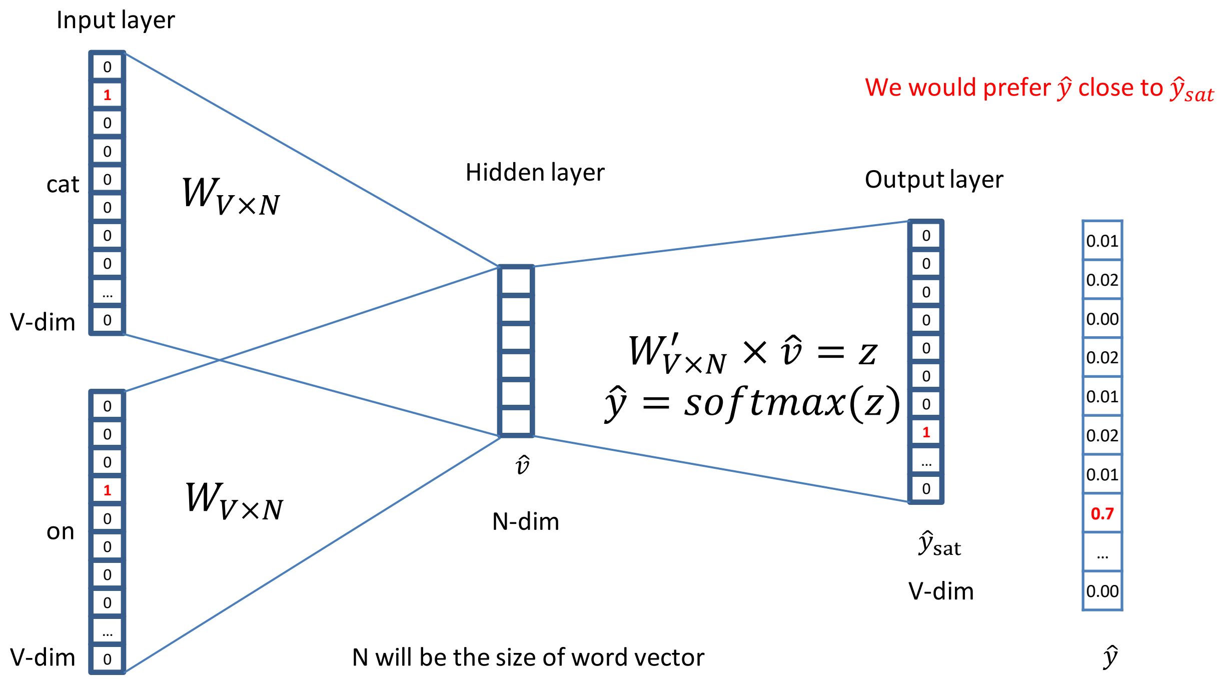
$W^T_{V\times N}$
$\times x_{on} = v_{on}$
Input layer
x_{cat}
V-dim
x_{on}
V-dim
$W^T_{V\times N} \times x_{cat} = v_{cat}$
$+$
$W^T_{V\times N} \times x_{on} = v_{on}$
$\hat{v} = \frac{v_{cat} + v_{on}}{2}$
Hidden layer
N-dim
Output layer
sat
V-dim

0.1	2.4	1.6	1.8	0.5	0.9	…	…	…	3.2
0.5	2.6	1.4	2.9	1.5	3.6	…	…	…	6.1
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
0.6	1.8	2.7	1.9	2.4	2.0	…	…	…	1.2

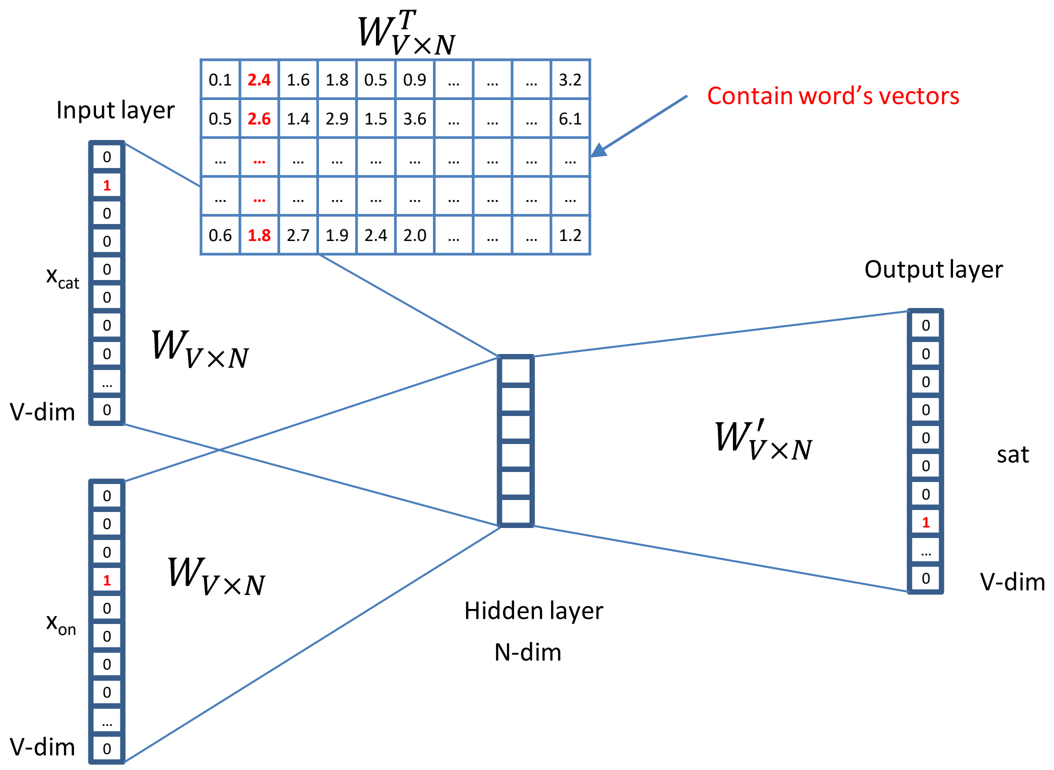
$\times$ [0, 0, 0, 1, 0, 0, 0, 0, …, 0] $=$ [1.8, 2.9, …, …, 1.9]



N will be the size of word vector



N will be the size of word vector



0.1	2.4	1.6	1.8	0.5	0.9	…	…	…	3.2
0.5	2.6	1.4	2.9	1.5	3.6	…	…	…	6.1
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
0.6	1.8	2.7	1.9	2.4	2.0	…	…	…	1.2

We can consider either W (context) or W' (center) as the word's representation.
Or even take the average.

Skipgram

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

W: N x |V|, input matrix, word representation as center word
W': |V| x N, output matrix, word representation as context word

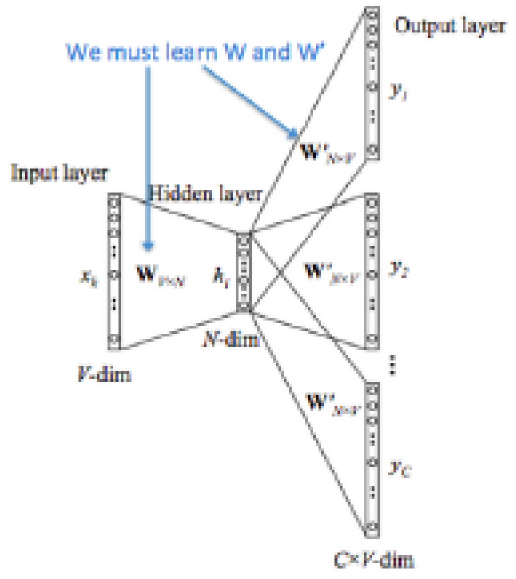
$y^{(j)}$ one hot vector for context words

1. Get *one hot vector* of the center word
x

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

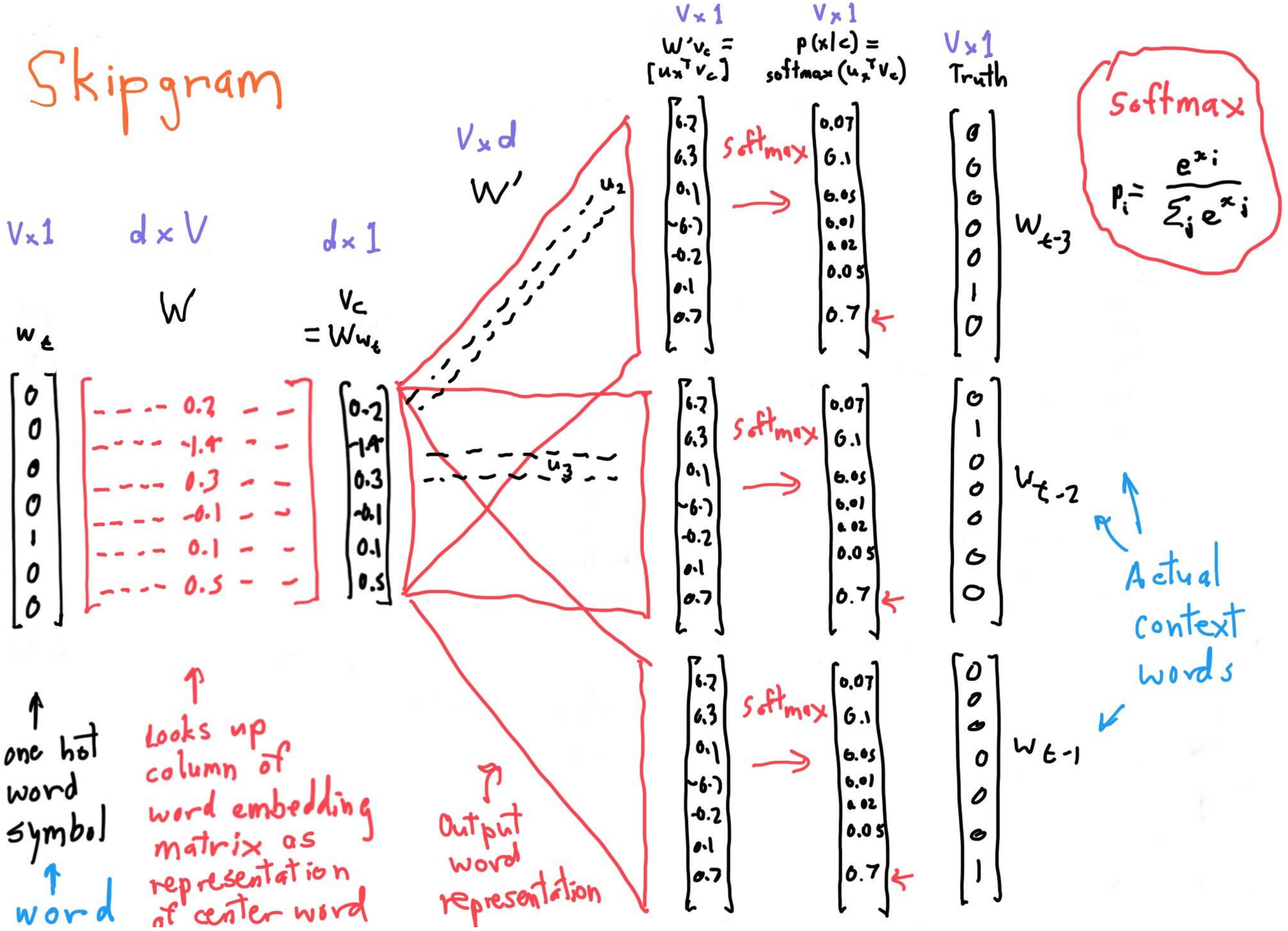
3. Generate a *score vector for each context word*
z = *W'* v_c

5. Turn the *score vector into probabilities*
ŷ = softmax(z)



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

Skipgram



Softmax

$$p_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

Skipgram

- For each word *t* = 1 … *T*, predict surrounding words in a window of "radius" *m* of every word.
- Objective function: Maximize the probability of any context word given the current center word:

$$J'(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \ne 0}} p(w_{t+j} \mid w_t ; \theta)$$

Negative Log Likelihood

$$J(\theta) = -\frac{1}{T} \sum_{t=1}^{T} \sum_{\substack{-m \le j \le m \\ j \ne 0}} \log p(w_{t+j} \mid w_t)$$

where θ represents all variables we will optimize

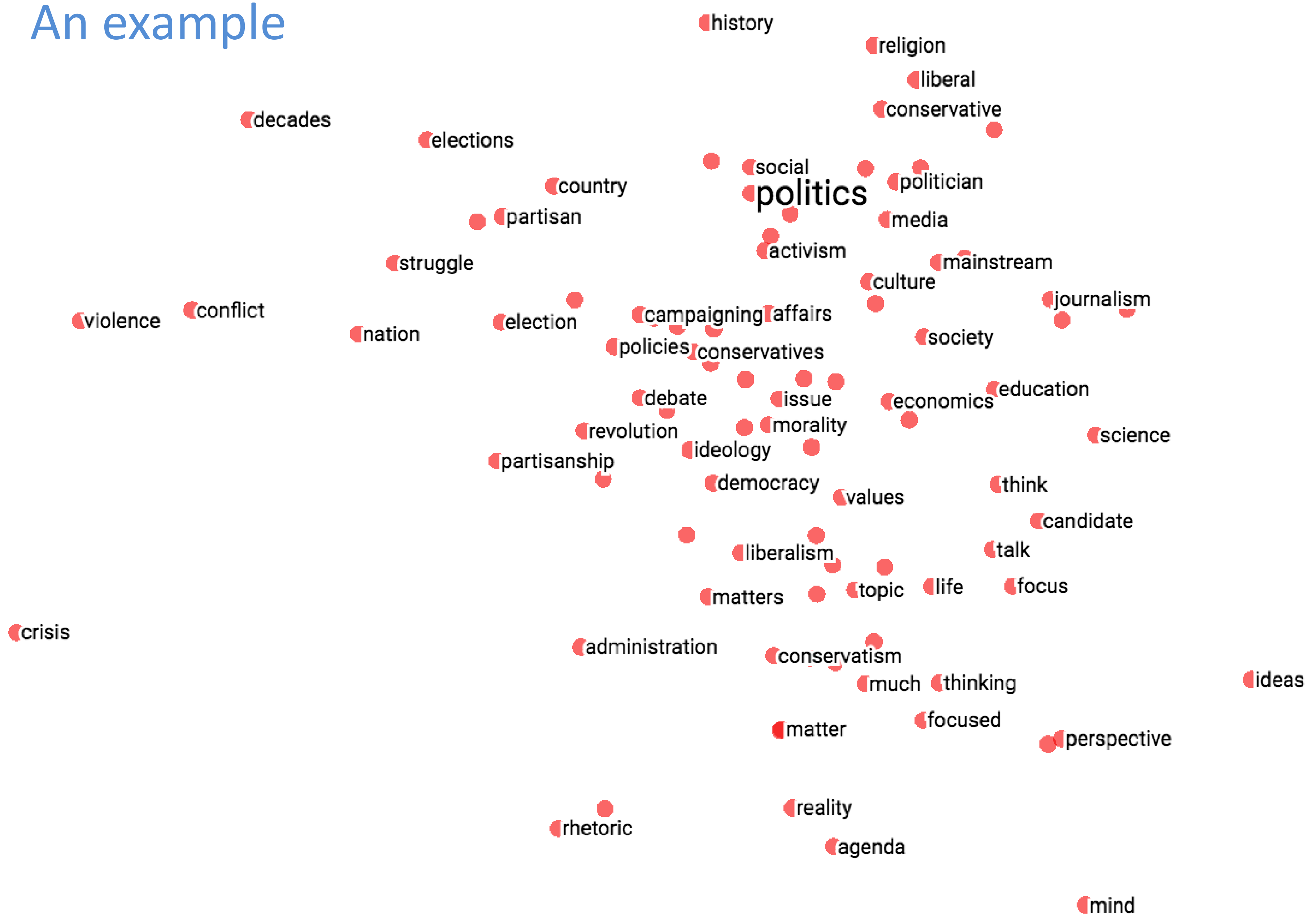
- The basic skipgram utilizes the softmax function:

$$p(c|w) = \frac{\exp\left({v'_c}^T v_w\right)}{\sum_{i=1}^{T} \exp({v'_i}^T v_w)}$$

- Where:
  - T – # of words in the corpus.
  - v_w - input vector of *w*.
  - v'_w - output vector of *w*.

Word	Input	Output
King	[0.2,0.9,0.1]	[0.5,0.4,0.5]
Queen	[0.2,0.8,0.2]	[0.4,0.5,0.5]
Apple	[0.9,0.5,0.8]	[0.3,0.9,0.1]
Orange	[0.9,0.4,0.9]	[0.1,0.7,0.2]

An example


history
religion
liberal
conservative
decades
elections
social
politics
politician
country
partisan
media
activism
struggle
mainstream
culture
journalism
violence
conflict
nation
election
campaigning
affairs
society
policies
conservatives
education
debate
issue
economics
revolution
morality
science
partisanship
ideology
democracy
values
think
candidate
liberalism
talk
matters
topic
life
focus
crisis
administration
conservatism
ideas
much
thinking
matter
focused
perspective
reality
rhetoric
agenda
mind

These representations are *very good* at encoding similarity and dimensions of similarity!

- Analogies testing dimensions of similarity can be solved quite well just by doing vector subtraction in the embedding space

  Syntactically

  – $x_{apple} - x_{apples} \approx x_{car} - x_{cars} \approx x_{family} - x_{families}$

  – Similarly for verb and adjective morphological forms

  Semantically (Semeval 2012 task 2)

  – $x_{shirt} - x_{clothing} \approx x_{chair} - x_{furniture}$

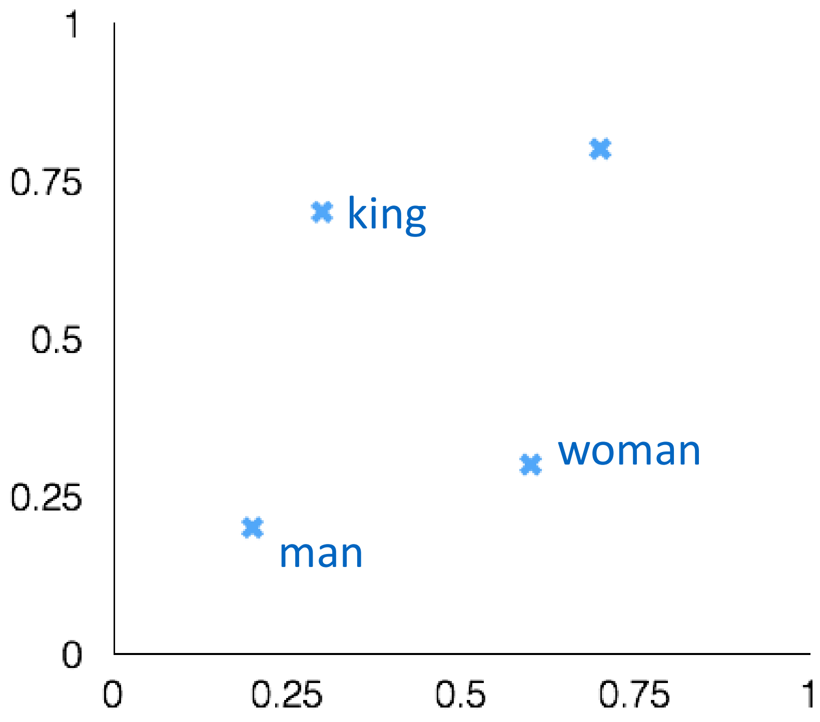
  – $x_{king} - x_{man} \approx x_{queen} - x_{woman}$

Test for linear relationships, examined by Mikolov et al.

$$\boxed{\text{a:b :: c:?}} \longrightarrow \boxed{d = \arg\max_x \frac{(w_b - w_a + w_c)^T w_x}{||w_b - w_a + w_c||}}$$

man:woman :: king:?

+	king	[0.30 0.70]
–	man	[0.20 0.20]
+	woman	[0.60 0.30]
	queen	[0.70 0.80]



Hierarchical softmax

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

How?

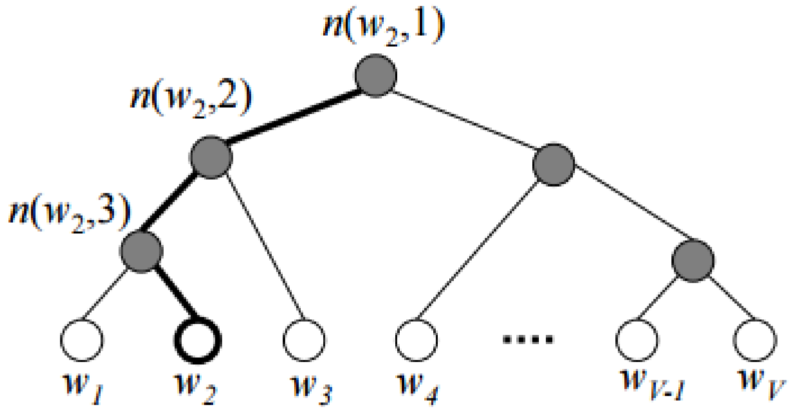
- Build a binary tree with leaves the words, *and learn one vector for each internal node*.
- The value for each word w is the product of the values of the internal nodes in the path from the root to w.

The probability of a word being the context word is defined as:

$$p(c|w) = \prod_{j=1}^{L(w)-1} \sigma(\underbrace{[\![n(c,j+1) = ch(n(w,j))]\!]}_{\text{returns 1 if the path goes left, - 1 if it goes right}} \cdot \overbrace{v_{n(c,j)}{}^{T} v_w}^{\text{compares the similarity of the } \textbf{\textit{input vector}}\ v_w \text{ to each internal node vector}})$$

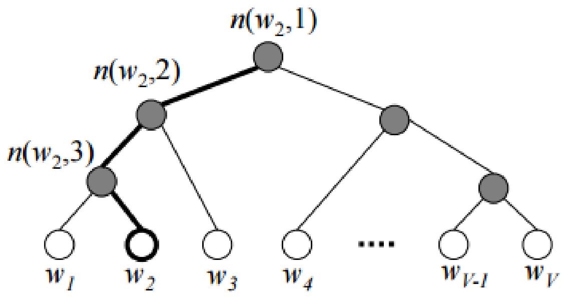
where:

- $n(w,j)$ – is the j-th node on the path from the root to *w*. $\mathrm{n(w,1) = root}$ $\mathrm{n\big(w, L(w)\big) = parent\ of\ w}$
- $L(w)$ – is the length of the path from root to *w*. $\mathrm{L(w_2) = 3}$
- $ch(n)$ – is the left child of node n.
- $[\![x]\!] = \begin{cases} 1 & if\ x\ is\ true \\ -1 & otherwise \end{cases}$
- $\sigma(x) = \frac{1}{1+e^{-x}}$



Suppose we want to compute the probability of w_2 being the output word.

- The probabilities of going right/left in a node *n* are:
  - $p(n, left) = \sigma(v_n{}^T v_w)$
  - $p(n, right) = 1 - \sigma(v_n{}^T v_w) = \sigma(-v_n{}^T v_w)$



$$p(w_2 = c) = p(n(w_2,1), left) \cdot p(n(w_2,2), left) \cdot\ p(n(w_2,3), right)$$

$$= \sigma(v_{n(w_2,1)}{}^T v_w) \cdot \sigma(v_{n(w_2,2)}{}^T v_w) \cdot \sigma(-v_{n(w_2,3)}{}^T v_w)$$

Complexity improved even further using a Huffman tree:

- Designed to compress binary code of a given text.
- A full binary suffix tree that guarantees a minimal average weighted path length when some words are frequently used.

Negative Sampling

- For each positive example we draw ***K*** *negative examples*.
- The negative examples are drawn according to the unigram distribution of the data

$$P_D(c) = \frac{\#(c)}{|D|}$$

$p(D = 1|w, c)$ is the probability that $(w, c) \in D$.

$p(D = 0|w, c) = 1 - p(D = 1|w, c)$ is the probability that $(w, c) \notin D$.

For negative samples: $p(D = 1|w, c)$ must be low $\Rightarrow$ $p(D = 0|w, c)$ will be high.

$$\arg\max_{\theta} \prod_{(w,c)\in D} p(D = 1|c, w; \theta) \boxed{\prod_{(w,c)\in D'} p(D = 0|c, w; \theta)}$$

$$= \arg\max_{\theta} \sum_{(w,c)\in D} \log \sigma(v_w \cdot v_c) + \sum_{(w,c)\in D'} \log \sigma(-v_w \cdot v_c)$$

For one sample:

$$\log \sigma(v_w \cdot v_c) + \sum_{i=1}^{k} \log \sigma(-v_w \cdot v_c)$$

BACK TO GRAPHS

How?

Words = Nodes

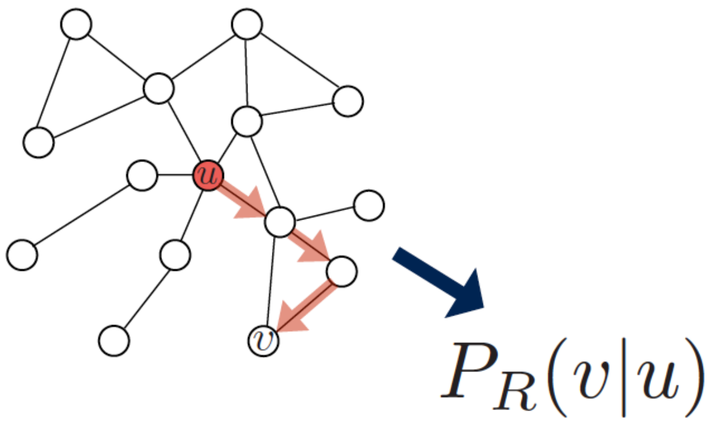
Sentences = Paths, Random walks

Random-walk embeddings

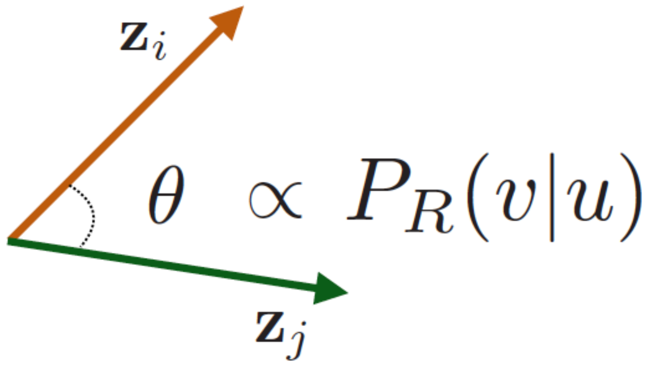
$z_i \cdot z_j \approx$ probability that *i* and *j* co-occur on a random walk over the network

Random-walk Embeddings

1. Estimate probability of visiting node v on a random walk starting from node u using some random walk strategy R.



2. Optimize embeddings to encode these random walk statistics.



Random Walk Optimization

1. Run short random walks starting from each node on the graph using some strategy R.
2. For each node u collect $\mathrm{N}_R(u)$, the multiset* of nodes visited on random walks starting from u.
3. Optimize embeddings according to (maximum likelihood):

$$L = \sum_{i \in V} \sum_{j \in N(i)} -\log(P(j|z_i))$$

* $\mathrm{N}_R(u)$ can have repeat elements since nodes can be visited multiple times on random walks.

Random Walk optimization

$$L = \sum_{i\,\in V} \sum_{j\,\in N(i)} -\log(P(j|z_i))$$

Intuition: Optimize embeddings to maximize likelihood of random walk co-occurrences.

Parameterize P(*v* | **z**$_u$) **using softmax:**

$$P(j|z_i) = \frac{\exp(z_i \cdot z_j)}{\sum_{m\in V} exp\,(z_i \cdot z_m)}$$

predicted probability of *i* and *j* co-occuring on random walk

$$L = -\log(\sum_{i\,\in V} \sum_{j\,\in N(i)} \frac{\exp(z_i \cdot z_j)}{\sum_{m\in V} exp\,(z_i \cdot z_m)})$$

sum over all nodes *i*

sum over nodes *m* seen on random walks starting from *i*

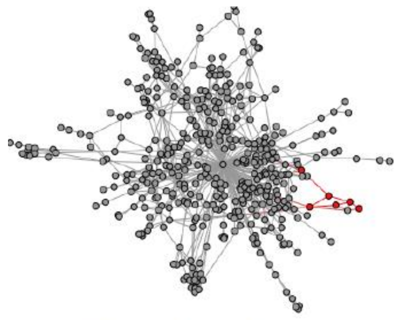
Why Random Walks?

1. **Expressivity:** Flexible stochastic definition of node similarity that incorporates both local and higher-order neighborhood information.
2. **Efficiency:** Do not need to consider all node pairs when training; only need to consider pairs that co-occur on random walks.

DeepWalk

Short random walks = sentences

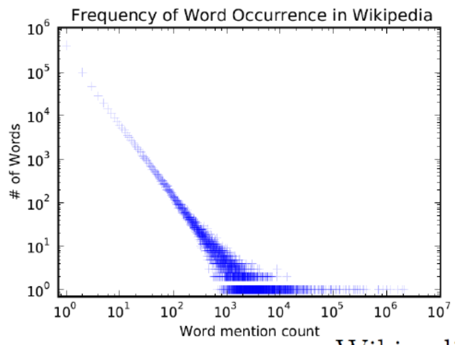
$$\begin{array}{cccccc}
v_{71} \rightarrow & v_{24} \rightarrow & v_{5} \rightarrow & v_{1} \rightarrow & v_{17} \rightarrow & v_{80} \rightarrow \\
v_{92} \rightarrow & v_{2} \rightarrow & v_{3} \rightarrow & v_{1} \rightarrow & v_{12} \rightarrow & v_{73} \rightarrow \\
v_{37} \rightarrow & v_{34} \rightarrow & v_{9} \rightarrow & v_{1} \rightarrow & v_{10} \rightarrow & v_{94} \rightarrow \\
v_{73} \rightarrow & v_{64} \rightarrow & v_{5} \rightarrow & v_{1} \rightarrow & v_{12} \rightarrow & v_{1} \rightarrow \\
v_{75} \rightarrow & v_{14} \rightarrow & v_{6} \rightarrow & v_{1} \rightarrow & v_{13} \rightarrow & v_{61} \rightarrow
\end{array}$$


Scale Free Graph

Short truncated random walks are sentences in an artificial language

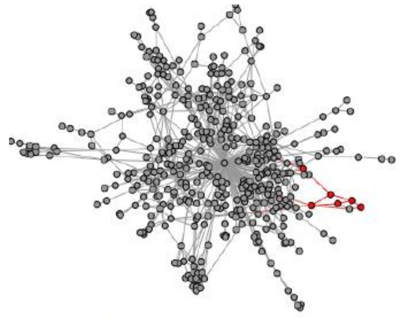
DeepWalk

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

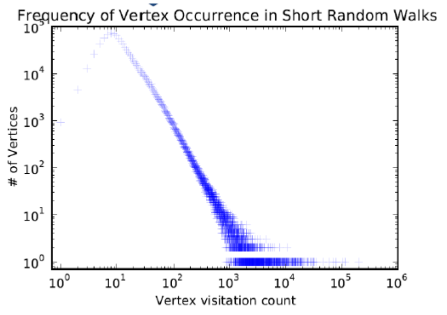


Wikipedia Article Text

$v_{71} \rightarrow$	$v_{24} \rightarrow$	$v_{5} \rightarrow$	$v_{1} \rightarrow$	$v_{17} \rightarrow$	$v_{80} \rightarrow$
$v_{92} \rightarrow$	$v_{2} \rightarrow$	$v_{3} \rightarrow$	$v_{1} \rightarrow$	$v_{12} \rightarrow$	$v_{73} \rightarrow$
$v_{37} \rightarrow$	$v_{34} \rightarrow$	$v_{9} \rightarrow$	$v_{1} \rightarrow$	$v_{10} \rightarrow$	$v_{94} \rightarrow$
$v_{73} \rightarrow$	$v_{64} \rightarrow$	$v_{5} \rightarrow$	$v_{1} \rightarrow$	$v_{12} \rightarrow$	$v_{1} \rightarrow$
$v_{75} \rightarrow$	$v_{14} \rightarrow$	$v_{6} \rightarrow$	$v_{1} \rightarrow$	$v_{13} \rightarrow$	$v_{61} \rightarrow$

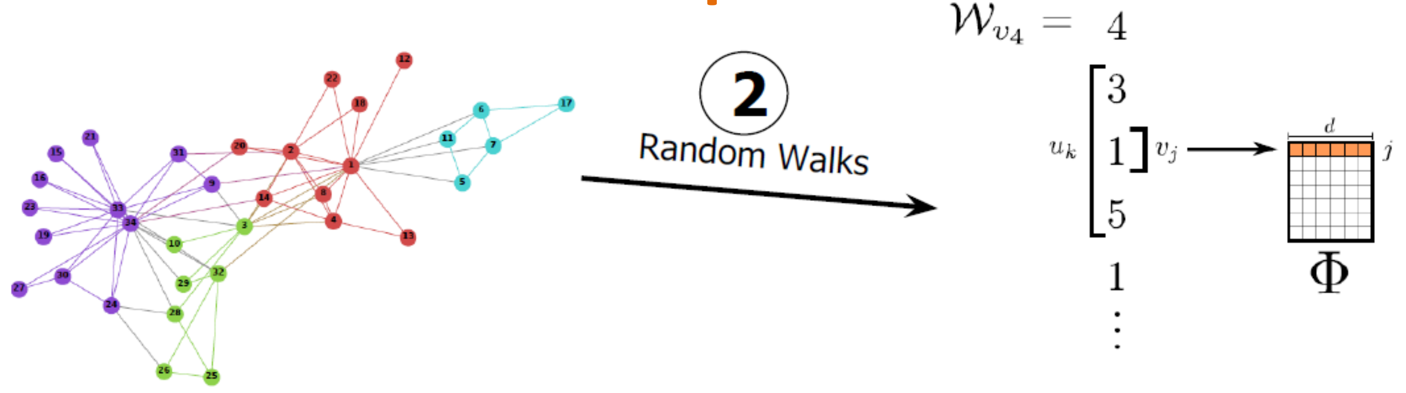

Scale Free Graph

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



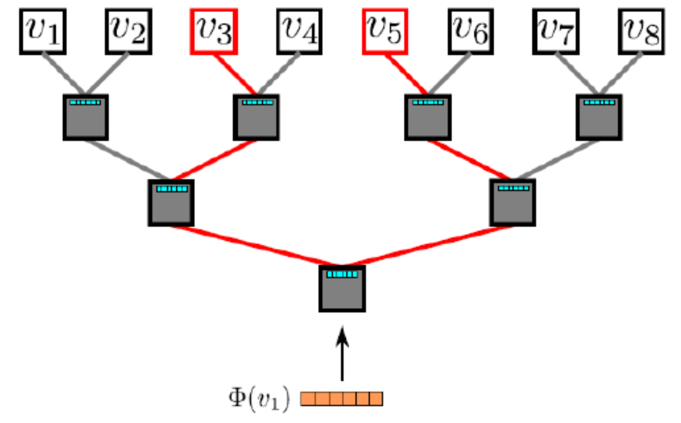
YouTube Social Graph

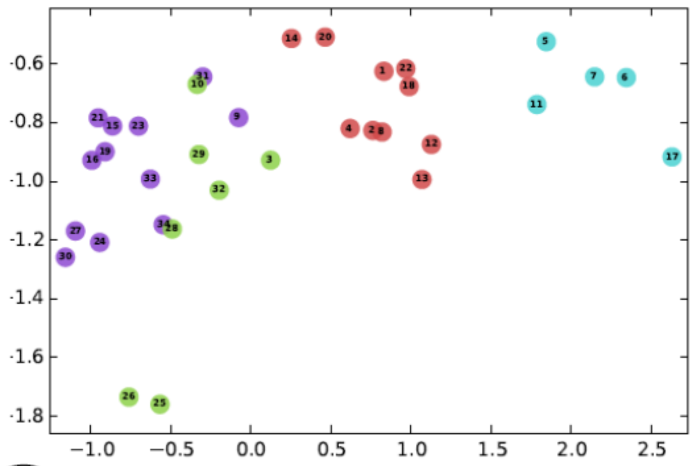
DeepWalk



1 Input: Graph

3 Representation Mapping



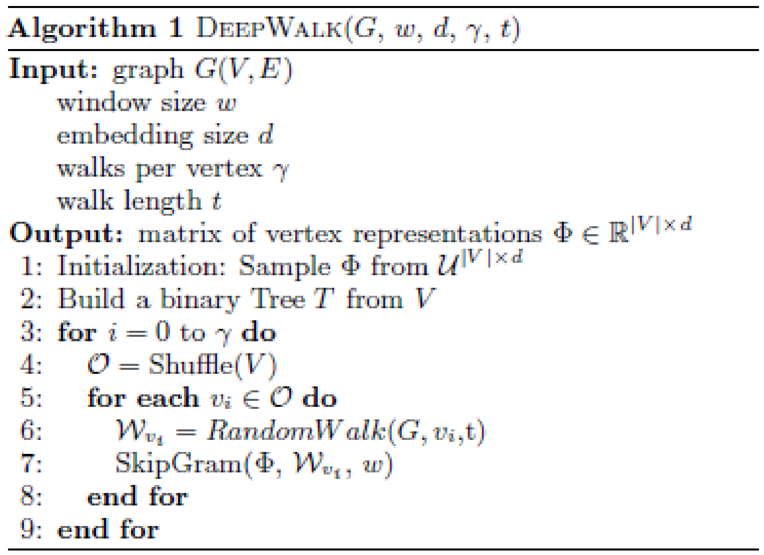


4 Hierarchical Softmax

5 Output: Representation

DeepWalk

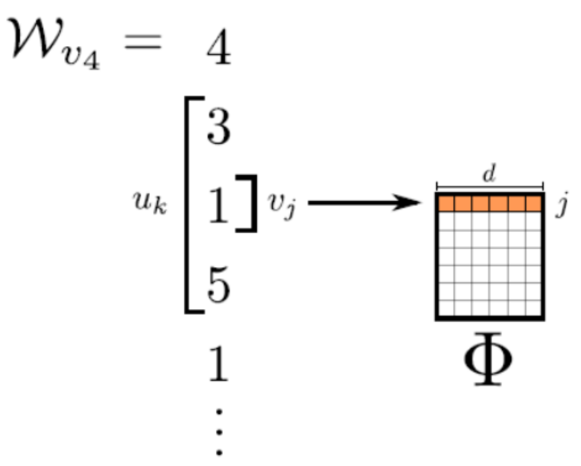
- Window *w*
- Generate *γ* random walks for *each vertex* in the graph
- Each short random walk has length *t*
  (intuitively, sentence length)
- Pick the next step *uniformly* from the node neighbors



```
Algorithm 1 DeepWalk($G$, $w$, $d$, $\gamma$, $t$)
Input: graph $G(V, E)$
    window size $w$
    embedding size $d$
    walks per vertex $\gamma$
    walk length $t$
Output: matrix of vertex representations $\Phi \in \mathbb{R}^{|V| \times d}$
 1: Initialization: Sample $\Phi$ from $\mathcal{U}^{|V| \times d}$
 2: Build a binary Tree $T$ from $V$
 3: for $i = 0$ to $\gamma$ do
 4:     $\mathcal{O} = \text{Shuffle}(V)$
 5:     for each $v_i \in \mathcal{O}$ do
 6:         $\mathcal{W}_{v_i} = RandomWalk(G, v_i, t)$
 7:         SkipGram($\Phi$, $\mathcal{W}_{v_i}$, $w$)
 8:     end for
 9: end for
```

Representation mapping

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



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

Maximize: $\Pr(v_3|\Phi({\color{red}v_1}))$

$\Pr(v_5|\Phi({\color{red}v_1}))$

Algorithm 2 SkipGram(Φ, $\mathcal{W}_{v_i}$, w)

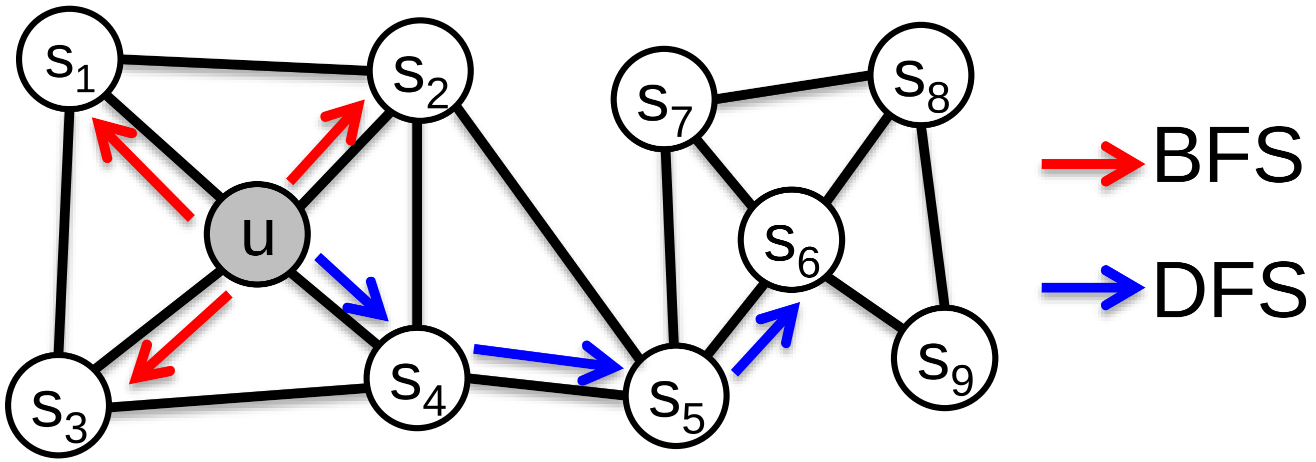
```
1: for each $v_j \in \mathcal{W}_{v_i}$ do
2:   for each $u_k \in \mathcal{W}_{v_i}[j - w : j + w]$ do
3:     $J(\Phi) = -\log \Pr(u_k \mid \Phi(v_j))$
4:     $\Phi = \Phi - \alpha * \frac{\partial J}{\partial \Phi}$
5:   end for
6: end for
```

Random Walk strategies

- DeepWalk just runs fixed-length, unbiased random walks starting from each node
- Node2vec: biased random walks that can trade off between **local** and **global** views of the network

node2vec: Biased Walks

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

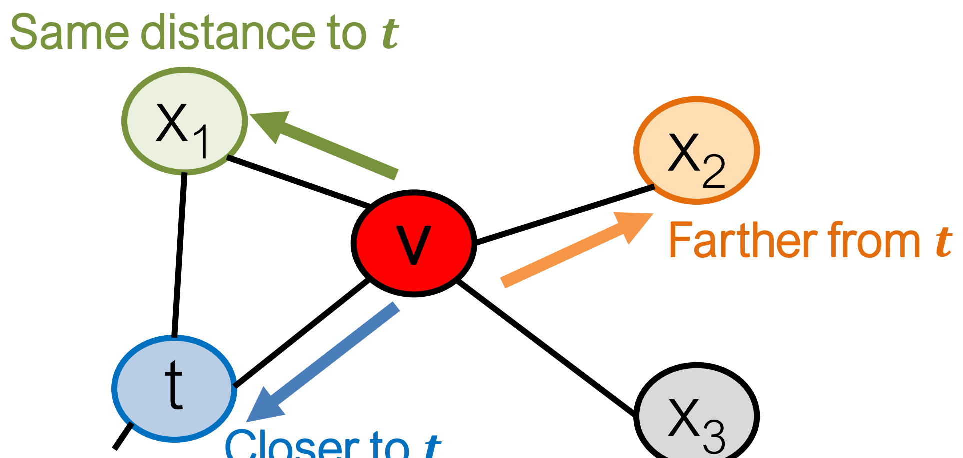


$N_{BFS}(u) = \{ s_1, s_2, s_3 \}$ Local microscopic view (BFS)

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

Biased 2nd Order Random Walks

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

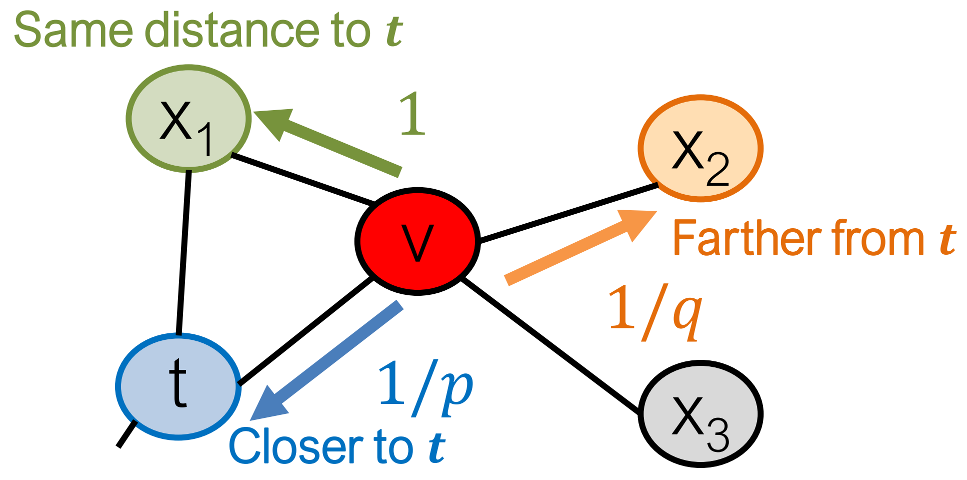


How much far away from *t*? Only three possible choices:

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

Biased Random Walks

At V from t, where to go next?



How much far away from t?

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

p, q model transition probabilities

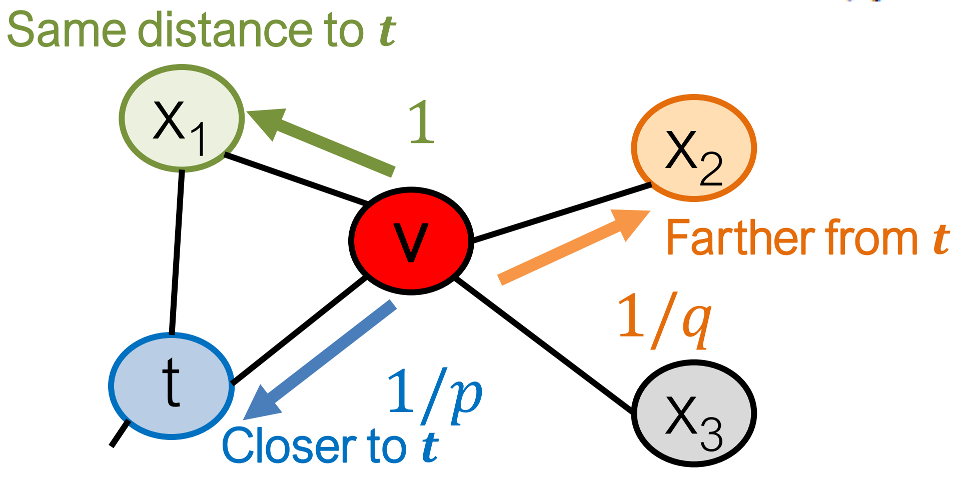
p return parameter

q ”walk away” parameter

Biased Random Walks

Walker at V from t, where to go next

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



BFS-like walk: Low value of p

DFS-like walk: Low value of q

p, q model transition probabilities
- p return parameter
- q ”walk away” parameter

$N_S(u)$ are the nodes visited by the walker

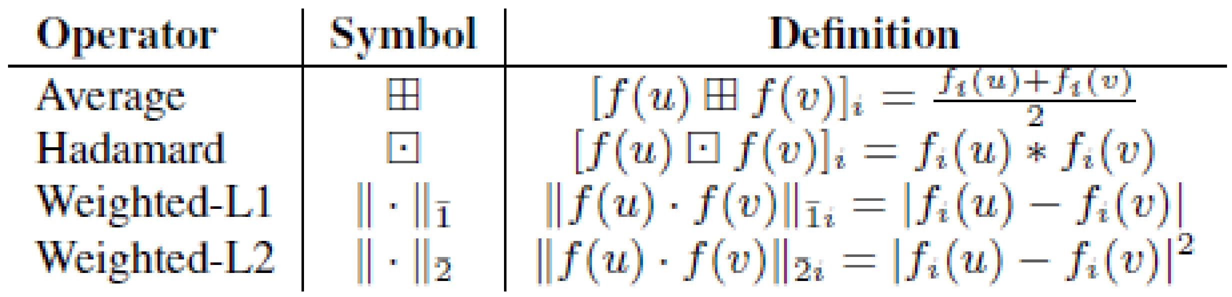
Interpolating BFS and DFS

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

- Two parameters:
  - Return parameter p:
    - Return back to the previous node
  - In-out parameter q:
    - Moving outwards (DFS) vs. inwards (BFS)

node2vec

Also learns edge vectors based on the vectors of their endpoints



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

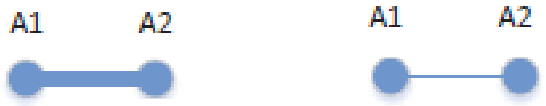
Node embeddings

Three approaches based on:

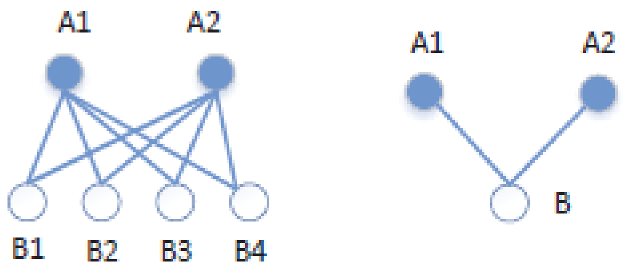
- Adjacency matrix
- Multi-hop neighborhoods
  - HOPE
  - GraRep
- Random-walks
  - DeepWalk
  - Node2Vec

GraRep

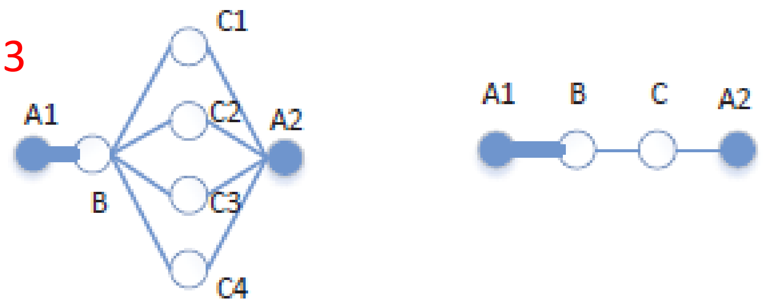
Path of length *k* =1



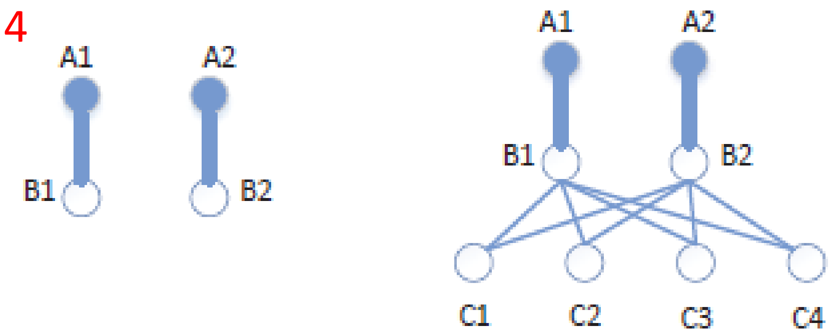
Path of length k = 2



Path of length *k* = 3



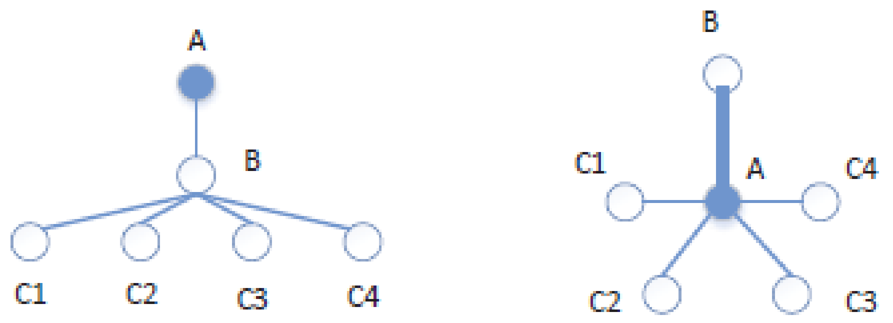
Path of length *k* = 4



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

GraRep

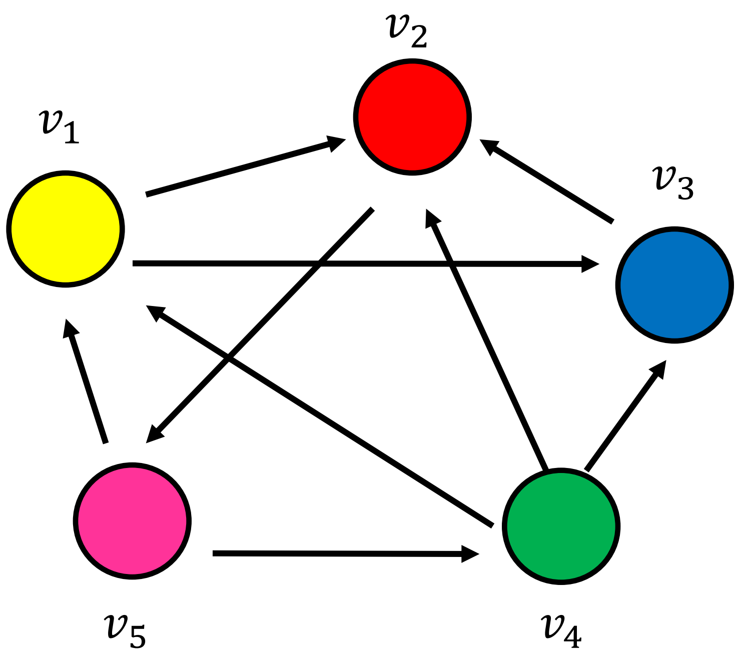
But not all k-neighbors equally important



Nearest neighborhoods more important

Maintain different k-step information differently in the graph representation

GraRep

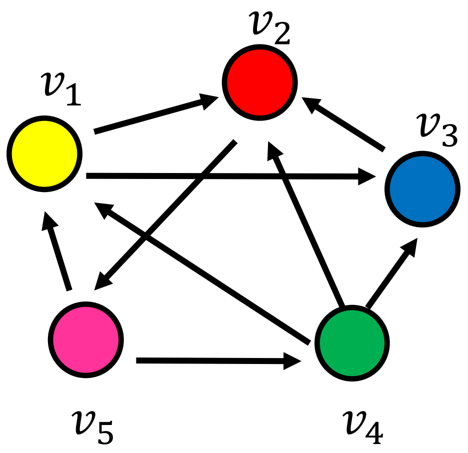


$$A = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{bmatrix} \quad D = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

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

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

GraRep



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

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

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

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

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

GraRep

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

1. Minimize the loss for *a specific k*

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

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

Basic idea:

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

GraRep

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

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

Skip-gram model

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

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

Learn two representations:

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

GraRep

Use negative sampling (*) and maximum likelihood
Assume for a given *k*, the collection of all paths from G that start from *i* and end at *j*.
Maximize

(1) Probability that these pairs came from the graph, and
(2) Probability that all other pairs did not come form the graph

$$L_k(i) = \overbrace{\sum_{j \in V} (p_k(j|i) \log \sigma(z_i \cdot z_j))}^{\text{probability that pair (i, j) came from the graph}} + \lambda \overbrace{E_{j' \sim p_k(V)}[\log \sigma(-z_i \cdot z_{j'})]}^{\text{probability that pair (i, j) did not come from the graph}}$$

hyper parameter indicating the number of negative samples

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

σ: sigmoid function

GraRep

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

Local objective for a specific pair of nodes

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

As before, compute the gradient and use stochastic gradient descent

Or solve by setting = 0 and get

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

Summary

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

GRAPH NEURAL NETWORKS

Outline

- Basic Variant
- Convolution GNNs
- GraphSAGE
- Gated GNNs

Embeddings: Key Components

Encoder maps each node to a low-dimensional vector.

d-dimensional embedding

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

node in the input graph

Similarity function specifies how relationships in vector space map to relationships in the original network.

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

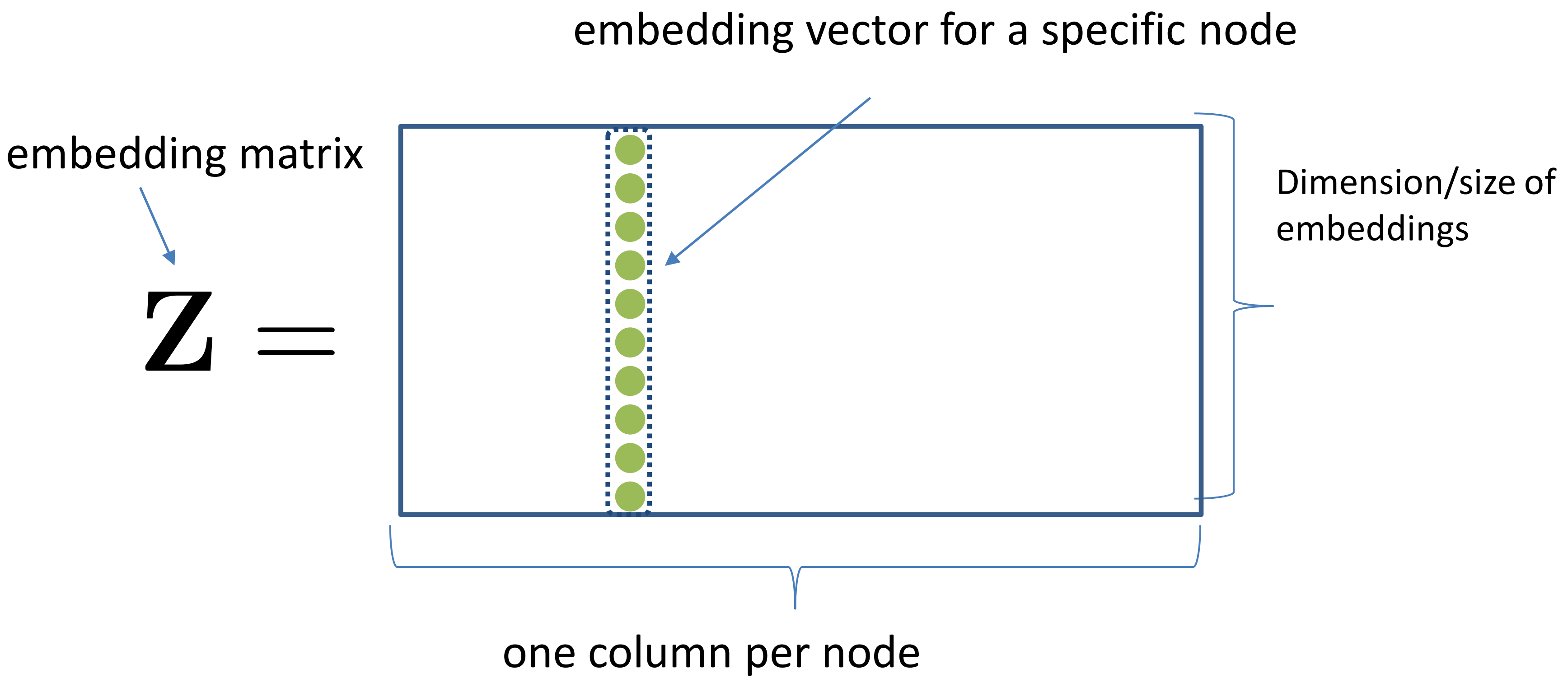
Similarity of *u* and *v* in the original network

dot product between node embeddings (*)

(*) other distance measures than dot products could be used (e.g., Euclidean distance), but the dot product is the standard measure of similarity used.

From “Shallow” to “Deep”

So far, *“shallow” encoders*, i.e. embedding lookups:


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

From “Shallow” to “Deep”

“deeper” methods based on *graph neural networks.*

$$\mathrm{ENC}(v) =$$ complex function that depends on graph structure.

- In general, all these more complex encoders can be *combined with the similarity functions* we discussed

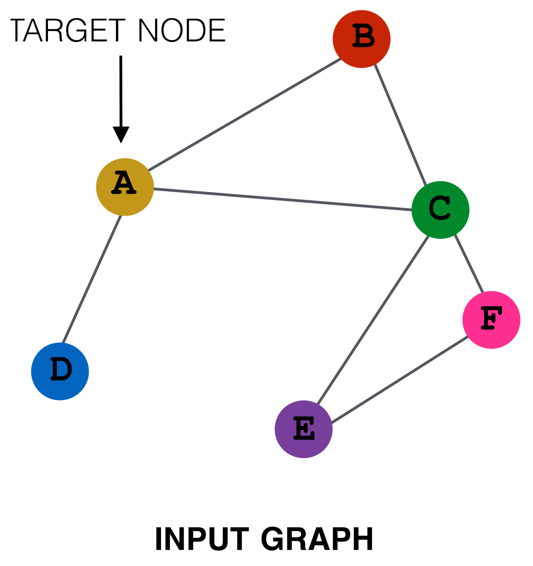
Setup

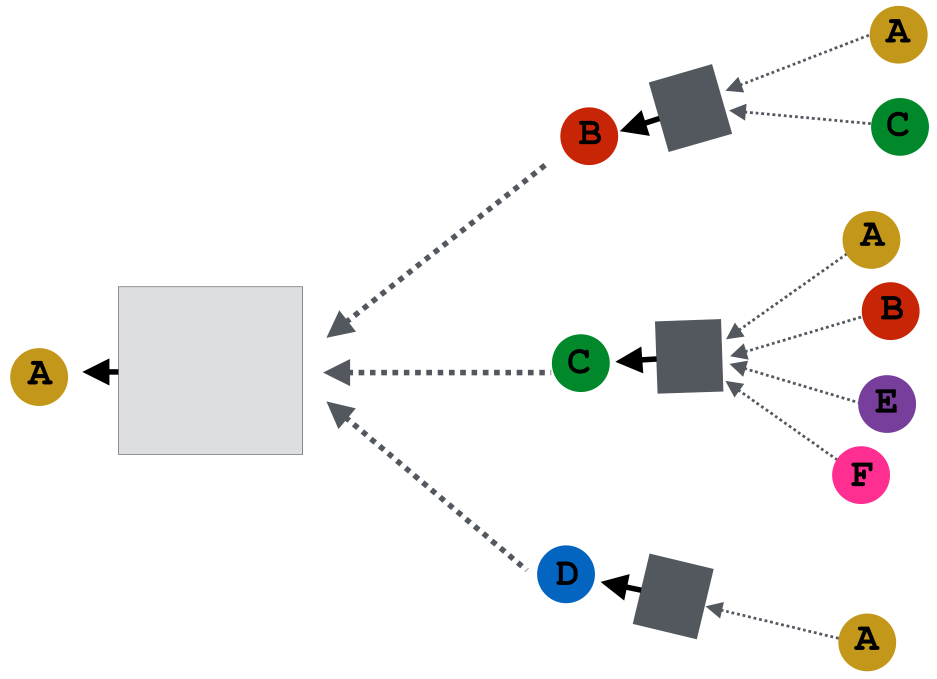
Assume we have a graph G:

- V is the vertex set.
- A is the adjacency matrix (assume binary).
- $X \in R^{m \times |V|}$ is a matrix of *m* node features (input representation of a node)
  - Categorical attributes, text, image data
    - E.g., profile information in a social network.
  - Node degrees, clustering coefficients, etc.
  - Indicator vectors (i.e., one-hot encoding of each node)

Neighborhood Aggregation

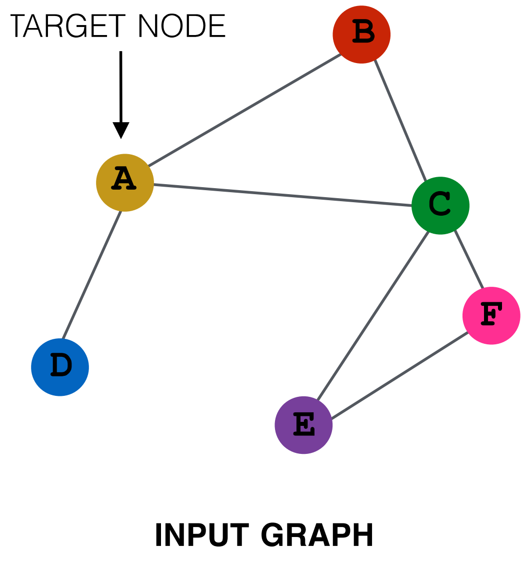
<span style="color:red">Key idea:</span> Generate node embeddings based on *local neighborhoods.*

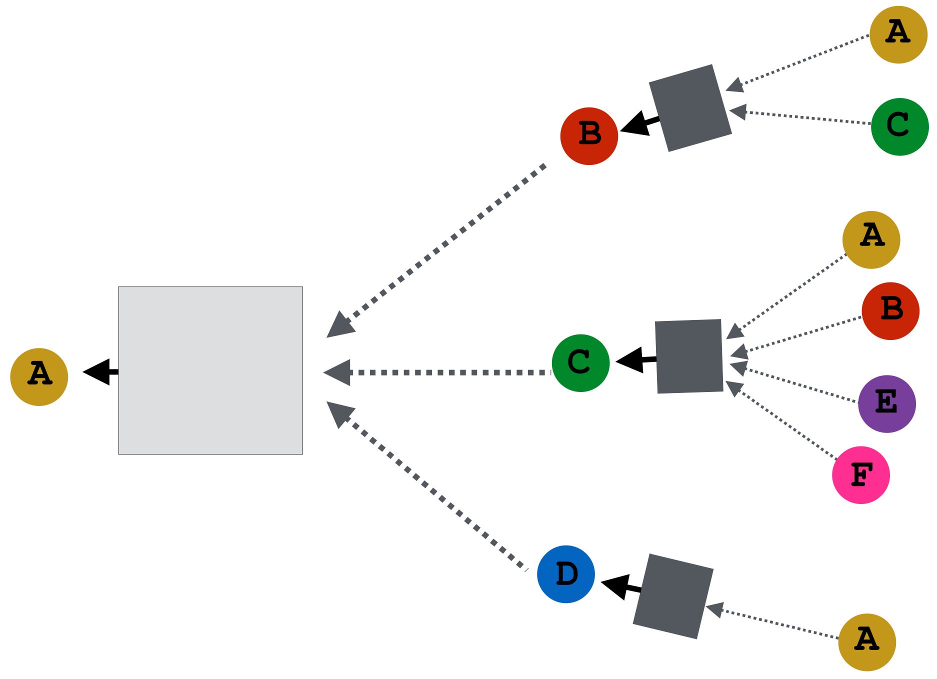




Neighborhood Aggregation

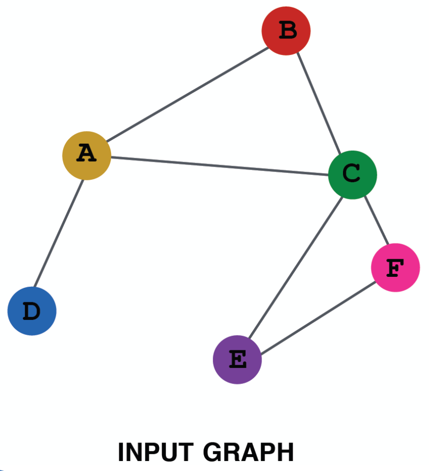
Intuition: Nodes aggregate information from their neighbors using *neural networks*



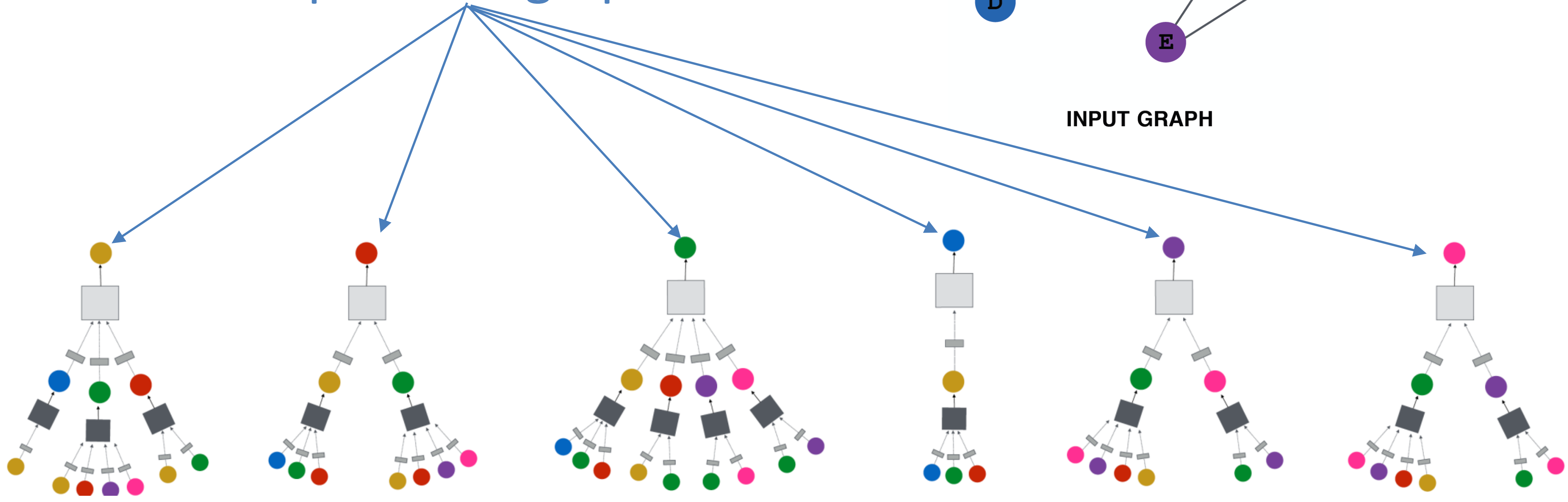


Neighborhood Aggregation

Intuition: Network neighborhood defines a computation graph

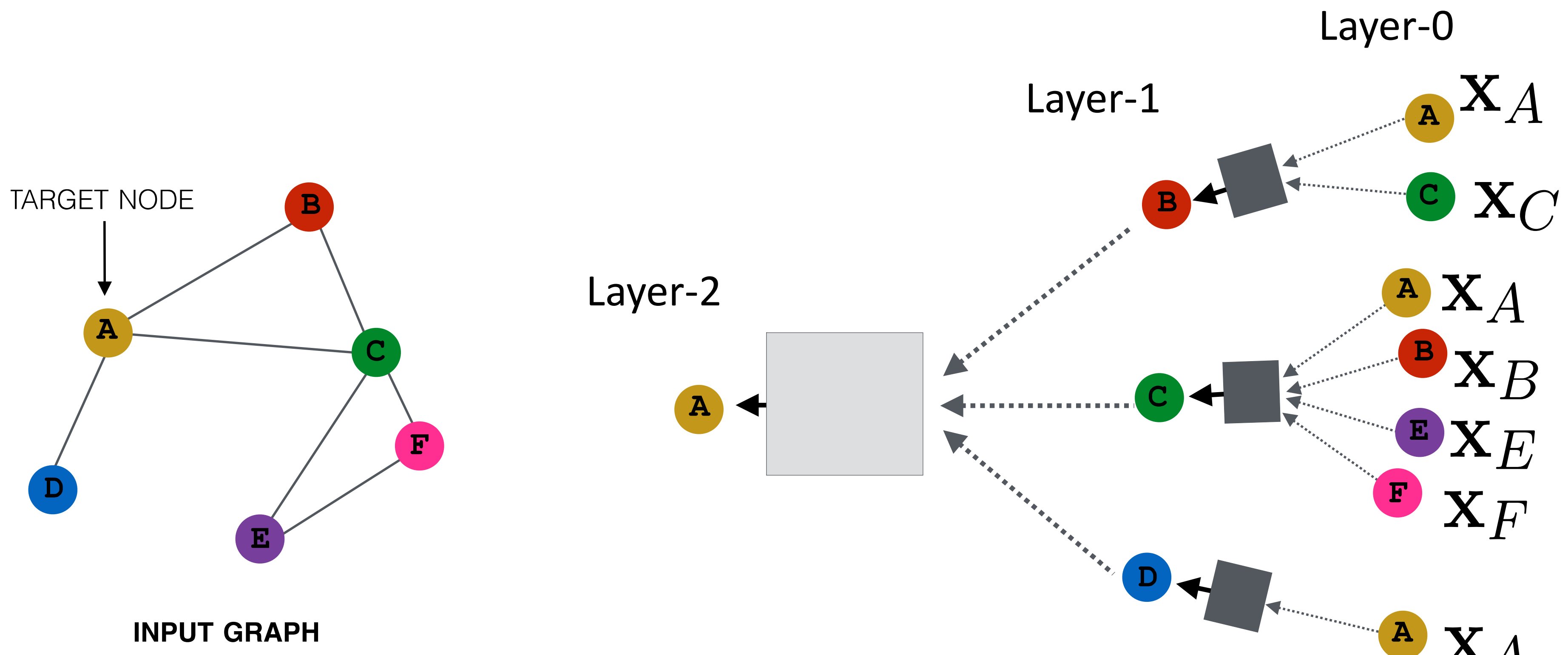


Every node defines a unique computation graph!



Neighborhood Aggregation

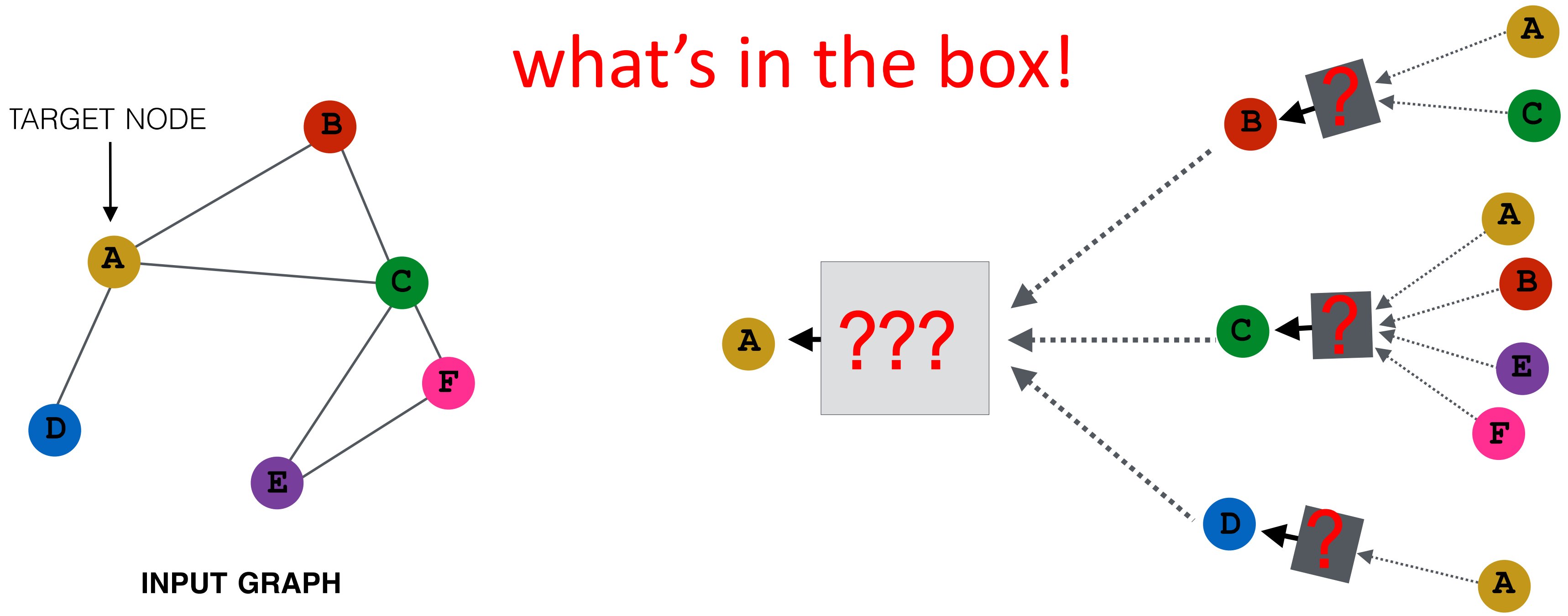
- Nodes have embeddings at each layer.
- Model can be of *arbitrary depth*.
- *layer-0* embedding of node u is its *input feature*, i.e. x_u.



Neighborhood Aggregation

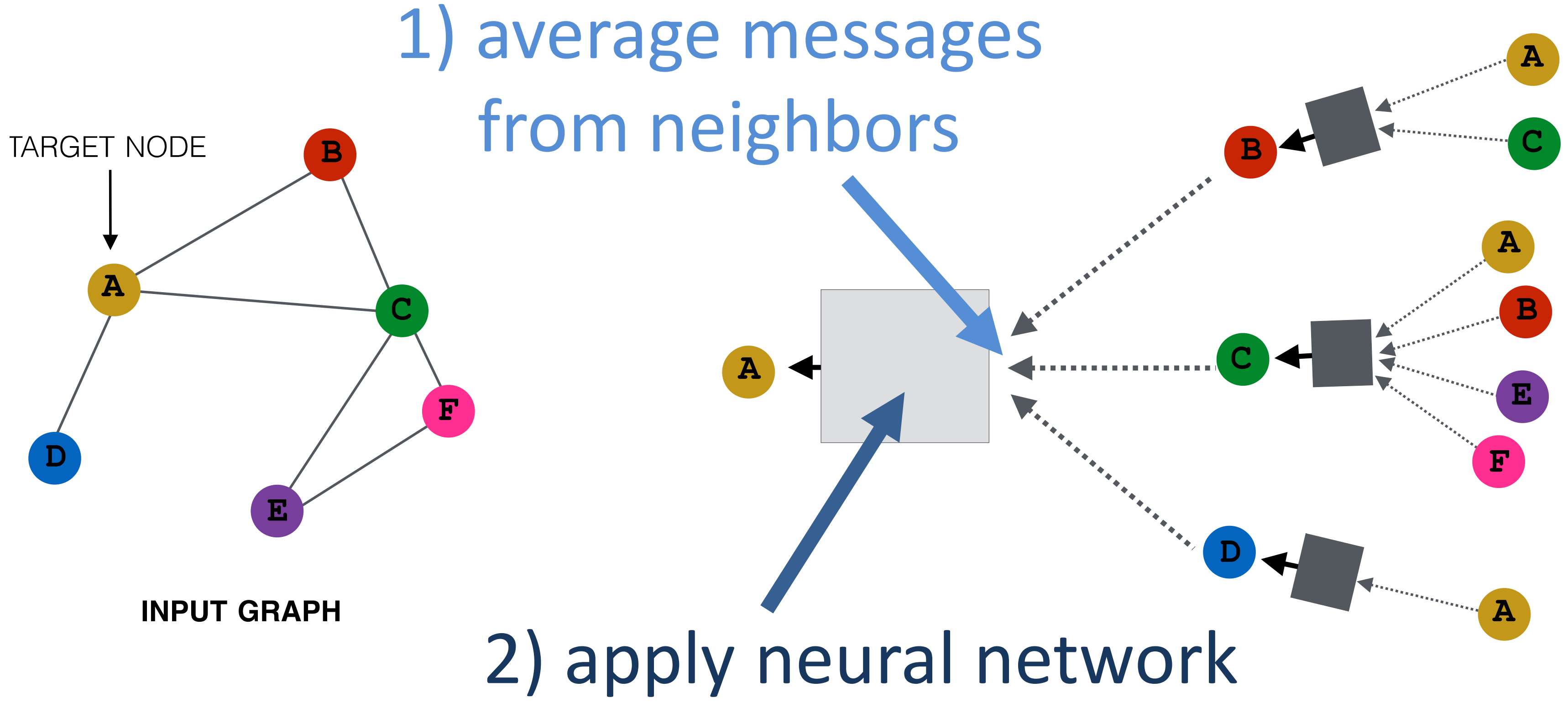
Key distinctions in how different approaches aggregate information across the layers.

what's in the box!



Neighborhood Aggregation

Basic approach: Average neighbor information and apply a neural network.



The Math

Basic approach: Average neighbor information and apply a neural network.

$$\boxed{\mathbf{h}_v^0 = \mathbf{x}_v}$$

Initial "layer 0" embeddings are equal to node features

previous layer embedding of v

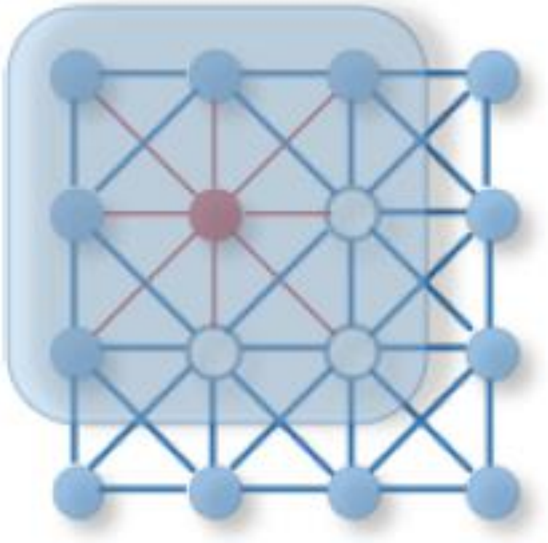
$$\boxed{\mathbf{h}_v^k} = \boxed{\sigma}\left(\mathbf{W}_k \boxed{\sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}} + \mathbf{B}_k \boxed{\mathbf{h}_v^{k-1}}\right), \quad \forall k > 0$$

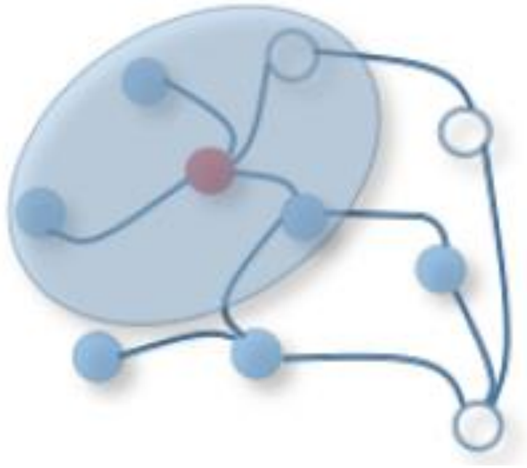
kth layer embedding of v

non-linearity (e.g., ReLU or tanh)

average of neighbor's previous layer embeddings

Graph Convolution



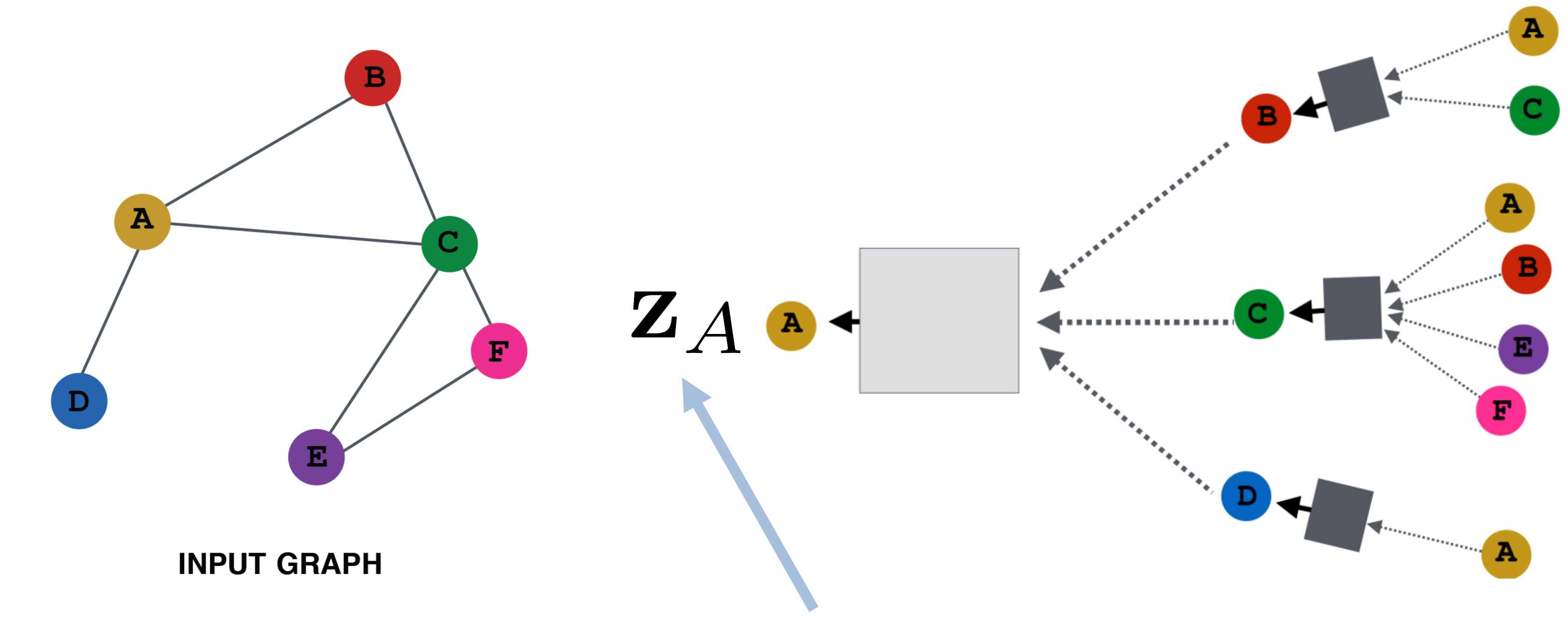


- Each *pixel* in an image as a node with neighbors determined by the *filter size*.
- 2D convolution takes a weighted *average of pixel values of the red node along with its neighbors*.
- Neighbors of a node are *ordered* and have a *fixed size*.

- To get a hidden representation of the red *graph node*, takes the average value of the node features of the red node along with its neighbors.
- Neighbors of a node are *unordered* and *variable in size*.

Training the Model

How do we train the model to generate high-quality embeddings?



Need to define a loss function on the embeddings, $L(z_u)$!

Training the Model

trainable matrices
(i.e., what we learn)

$$\mathbf{h}_v^0 = \mathbf{x}_v$$

$$\mathbf{h}_v^k = \sigma\left(\boxed{\mathbf{W}_k} \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \boxed{\mathbf{B}_k} \mathbf{h}_v^{k-1}\right), \quad \forall k \in \{1, ..., K\}$$

$$\boxed{\mathbf{z}_v = \mathbf{h}_v^K}$$

- After K-layers of neighborhood aggregation, we get output embeddings for each node.
- We can feed these embeddings into *any loss function* and run *stochastic gradient descent* to train the aggregation parameters.

Training the Model

- Train in an unsupervised manner using only the graph structure.
- Unsupervised loss function can be anything from the last section, e.g., based on
  - Random walks (node2vec, DeepWalk)
  - Graph factorization
  - i.e., train the model so that “similar” nodes have similar embeddings.

Training the Model

λ number of negative samples

$$L(z_u) = -\log\left(\sigma(z_u{}^{\mathrm{T}} z_v)\right) - \lambda\ E_{v_n \sim P_n(v)}[\log \sigma(-z_u^T z_{v_n}))$$

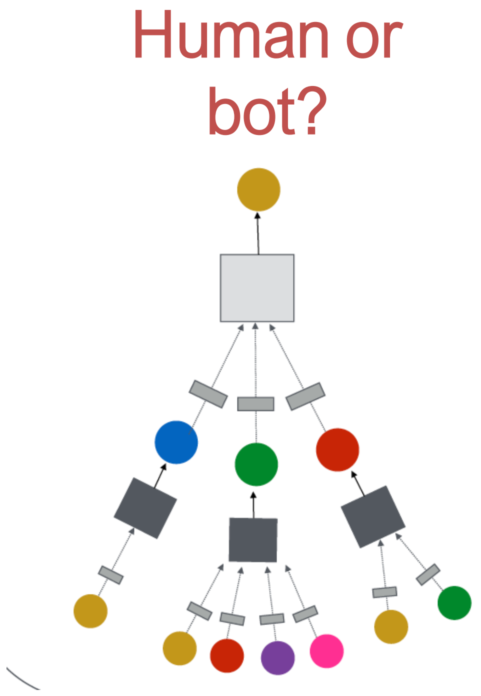
v a node that co-occurs near u on fixed-length random walk

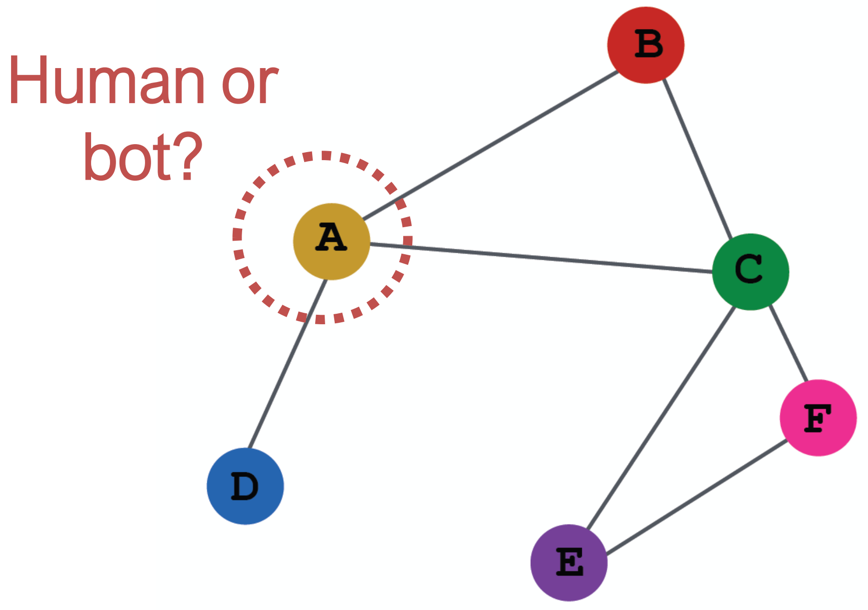
P_n negative sampling distribution

- representations z_u feed into thε loss function are generated from the features contained within a node's local neighborhood, rather than training a unique embedding for each node

Training the Model

Alternative: Directly train the model for a supervised task (e.g., node classification):





e.g., an online social network

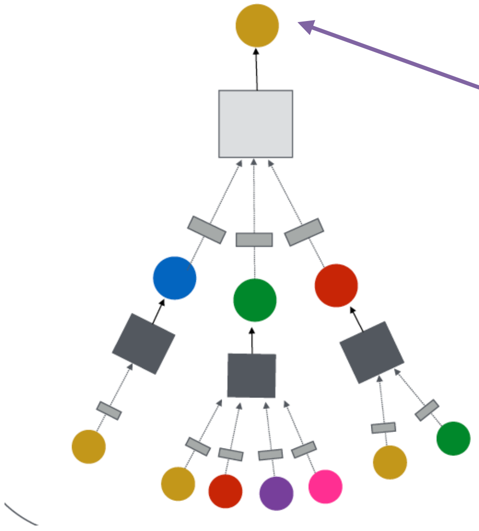
Training the Model

Alternative: Directly train the model for a supervised task (e.g., node classification):

Human or bot?

classification weights

$$\mathcal{L} = \sum_{v \in V} \boxed{y_v} \log(\sigma(\boxed{\mathbf{z}_v}^\top \boxed{\boldsymbol{\theta}})) + (1 - \boxed{y_v}) \log(1 - \sigma(\boxed{\mathbf{z}_v}^\top \boxed{\boldsymbol{\theta}}))$$

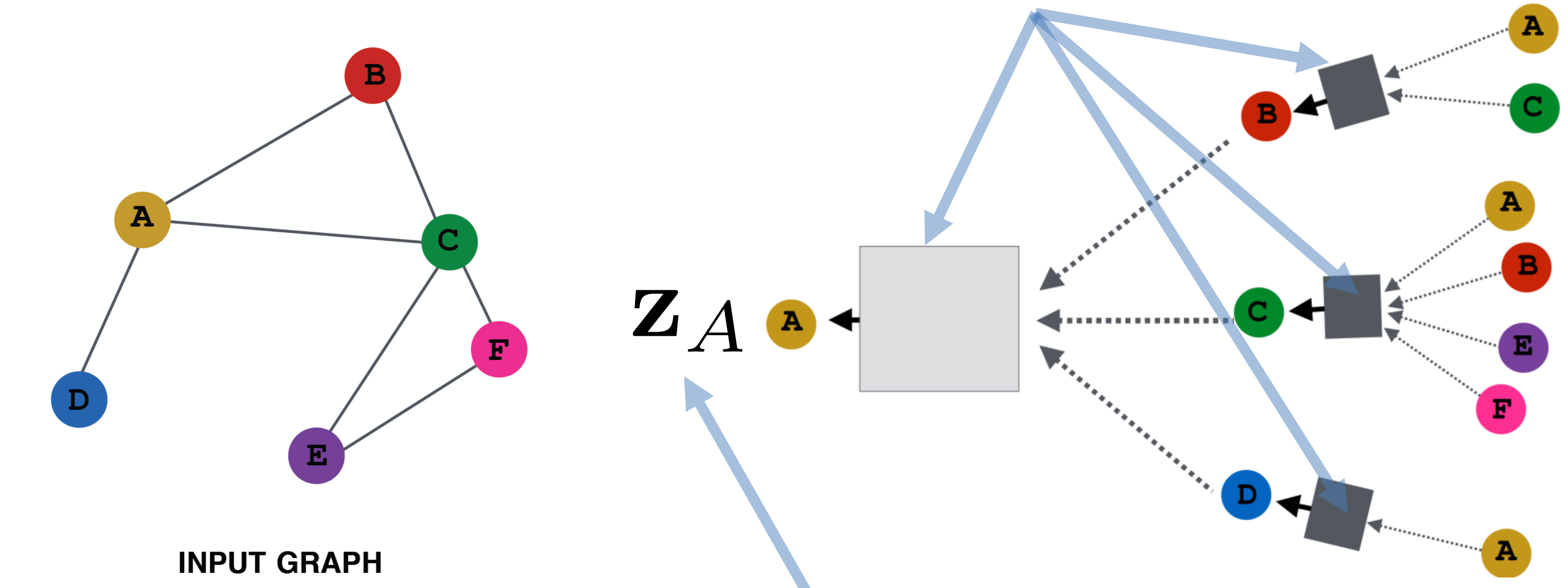


output node embedding

node class label

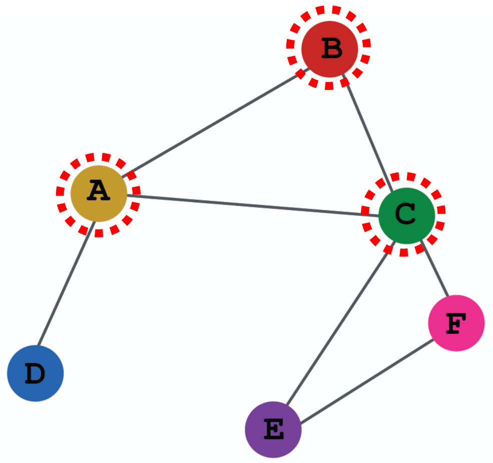
Overview of Model Design

1) Define a neighborhood aggregation function.



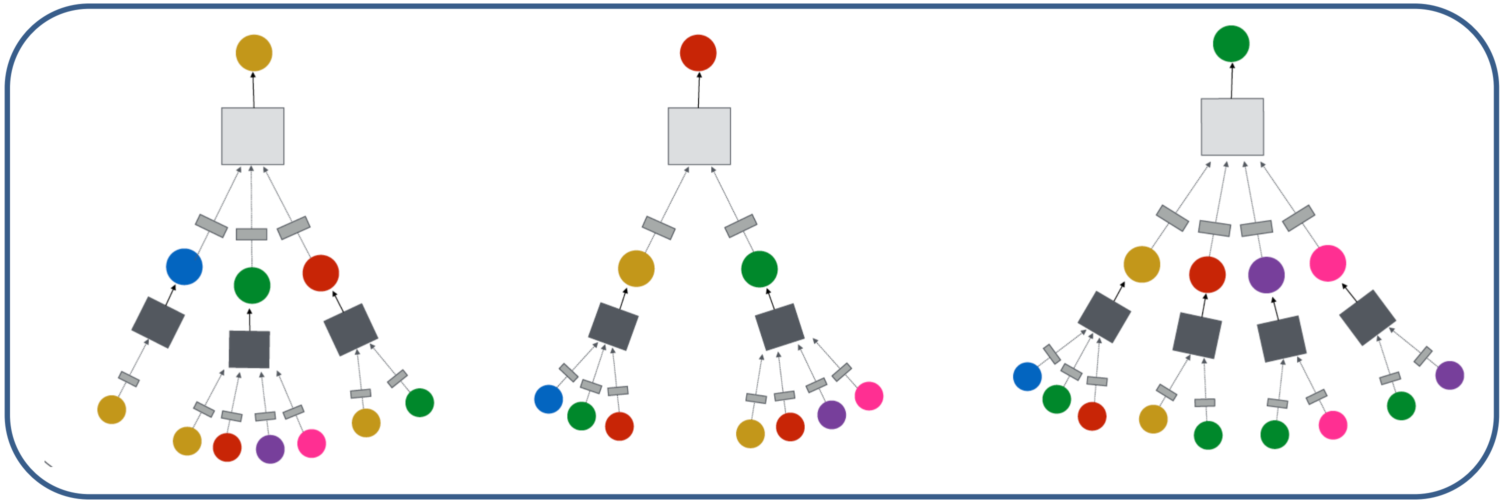
2) Define a loss function on the embeddings, $L(z_u)$

Overview of Model Design

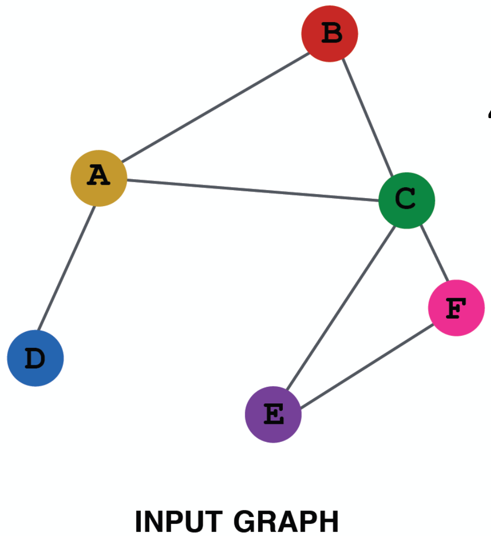


INPUT GRAPH

3) Train on a set of nodes, i.e., a batch of compute graphs

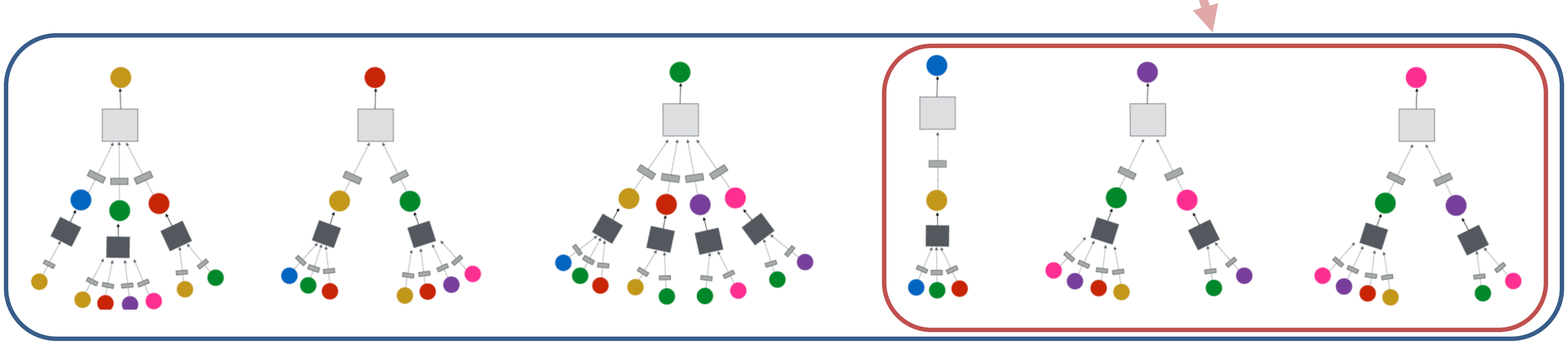


Overview of Model



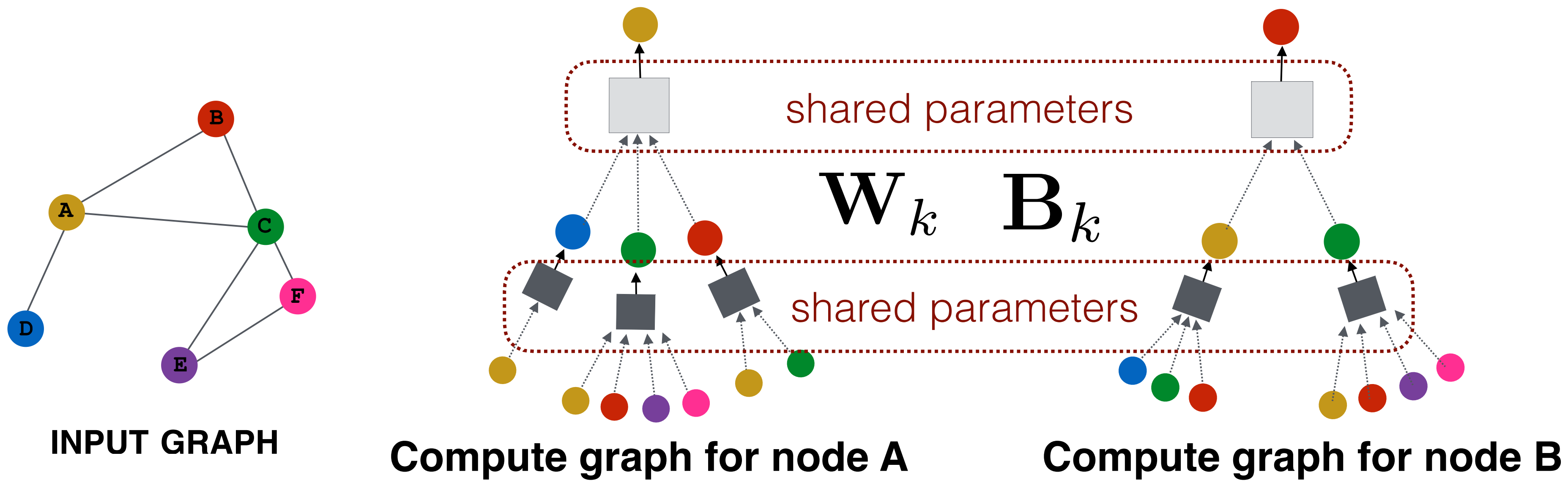
4) Generate embeddings for nodes as needed

Even for nodes we never trained on!!!!

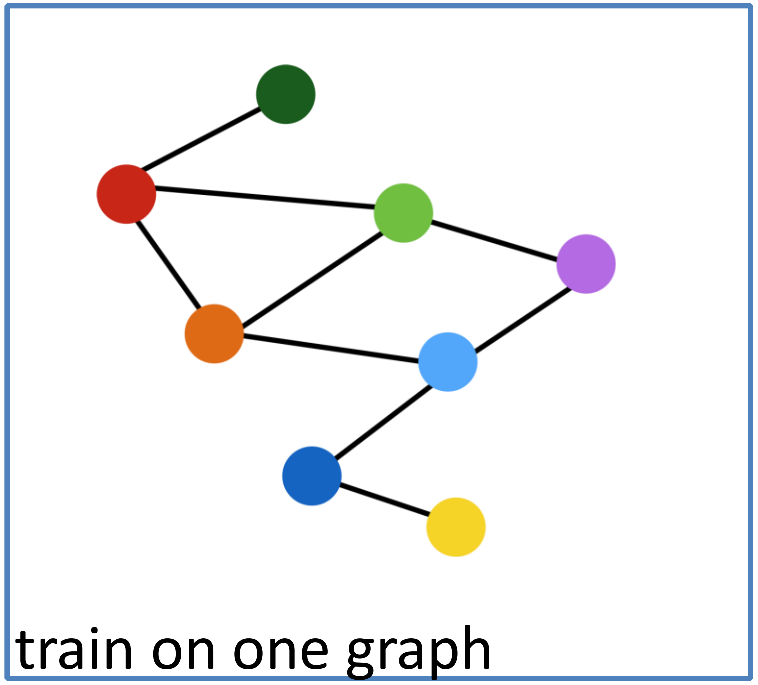


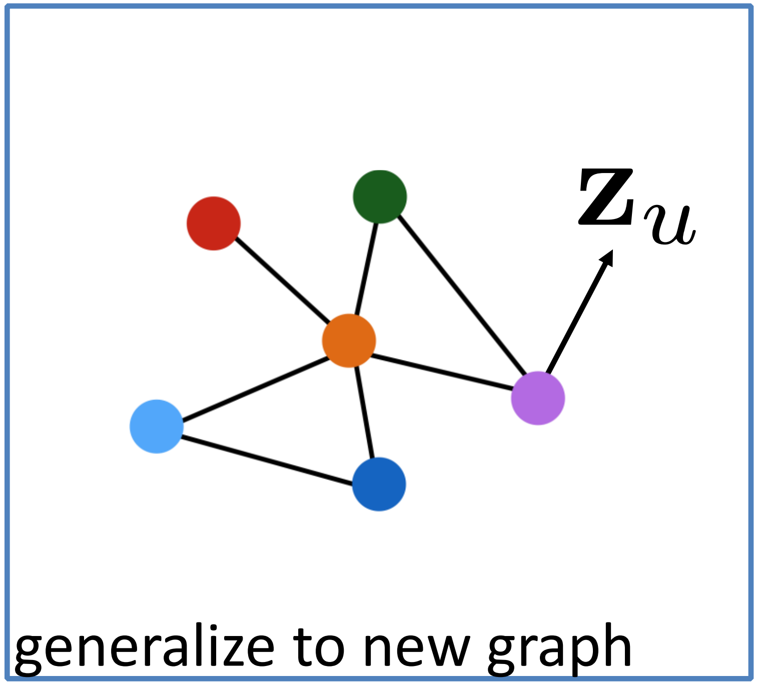
Inductive Capability

- The same aggregation parameters are shared for all nodes.
- The number of model parameters is sublinear in |V| and we can generalize to unseen nodes!



Inductive Capability

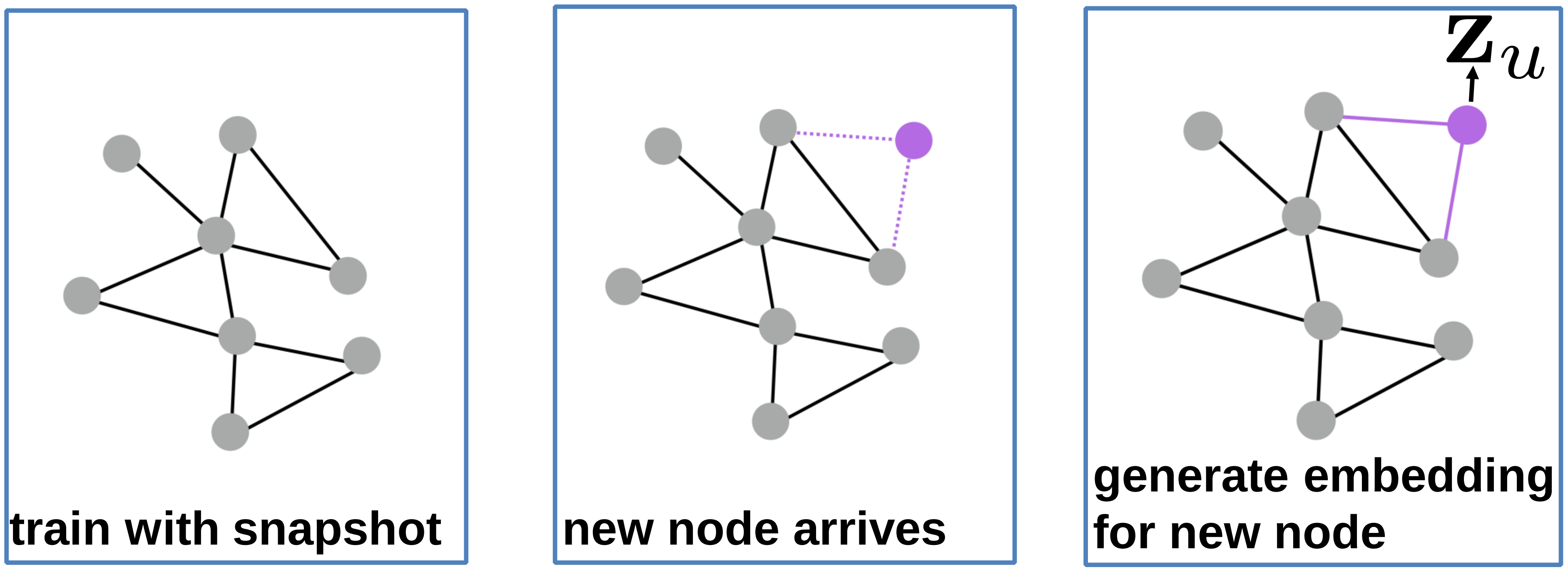




Inductive node embedding → generalize to entirely unseen graphs

e.g., train on protein interaction graph from model organism A and generate embeddings on newly collected data about organism B

Inductive Capability



Many application settings constantly encounter previously unseen nodes.

e.g., Reddit, YouTube, GoogleScholar, ….

Need to generate new embeddings "on the fly"

Quick Recap

Basic variant: Generate node embeddings by aggregating neighborhood information.

– Allows for parameter sharing in the encoder.

– Allows for inductive learning.

Graph Convolutional Networks (GCNs)

Variation on the neighborhood aggregation idea:

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v) \cup v} \frac{\mathbf{h}_u^{k-1}}{\sqrt{|N(u)||N(v)|}}\right)$$

Kipf et al., 2017. *Semisupervised Classification with Graph Convolutional Networks*. ICLR.

Graph Convolutional Networks

Basic Neighborhood Aggregation

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \mathbf{B}_k \mathbf{h}_v^{k-1}\right)$$

VS.

GCN Neighborhood Aggregation

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v) \cup v} \frac{\mathbf{h}_u^{k-1}}{\sqrt{|N(u)||N(v)|}}\right)$$

same matrix for self and neighbor embeddings

per-neighbor normalization

Graph Convolutional Networks

Empirically, they found this configuration to give the best results.

- More parameter sharing.
- Down-weights high degree neighbors.

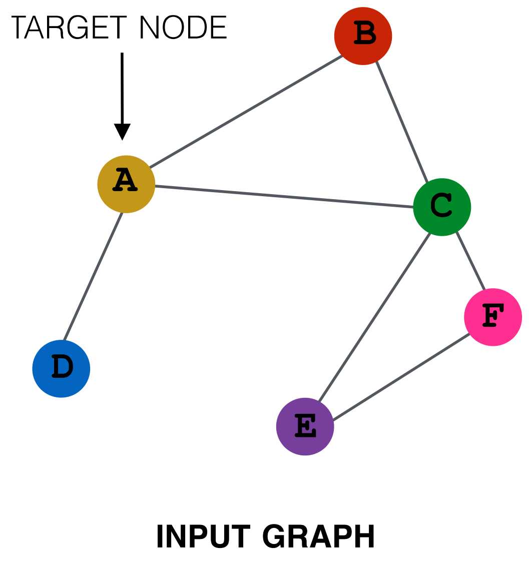
$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v) \cup v} \frac{\mathbf{h}_u^{k-1}}{\sqrt{|N(u)||N(v)|}}\right)$$

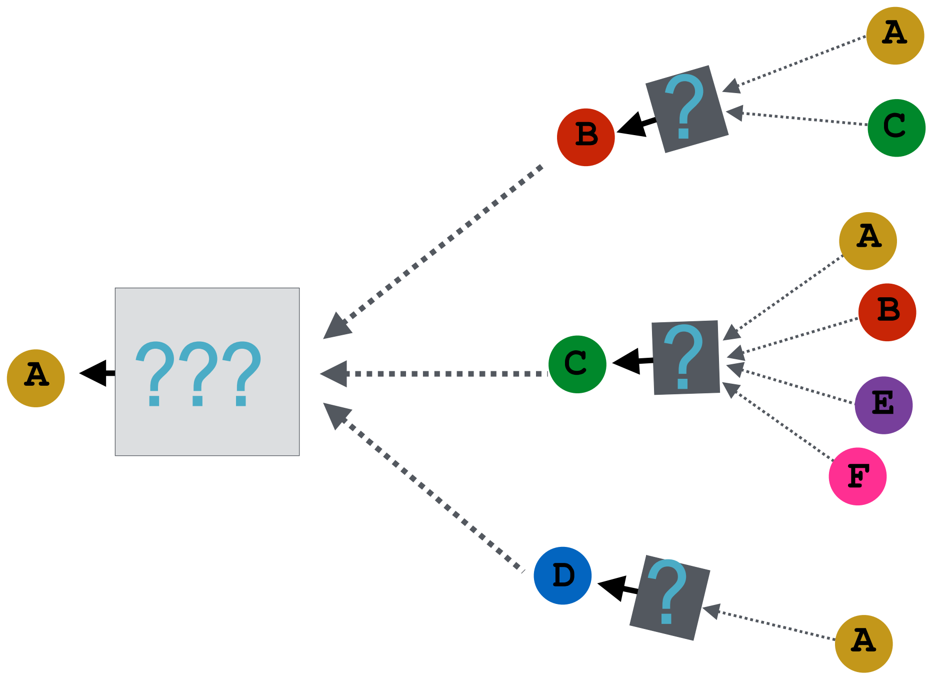
use the same transformation matrix for self and neighbor embeddings

instead of simple average, normalization varies across neighbors

GraphSAGE Idea

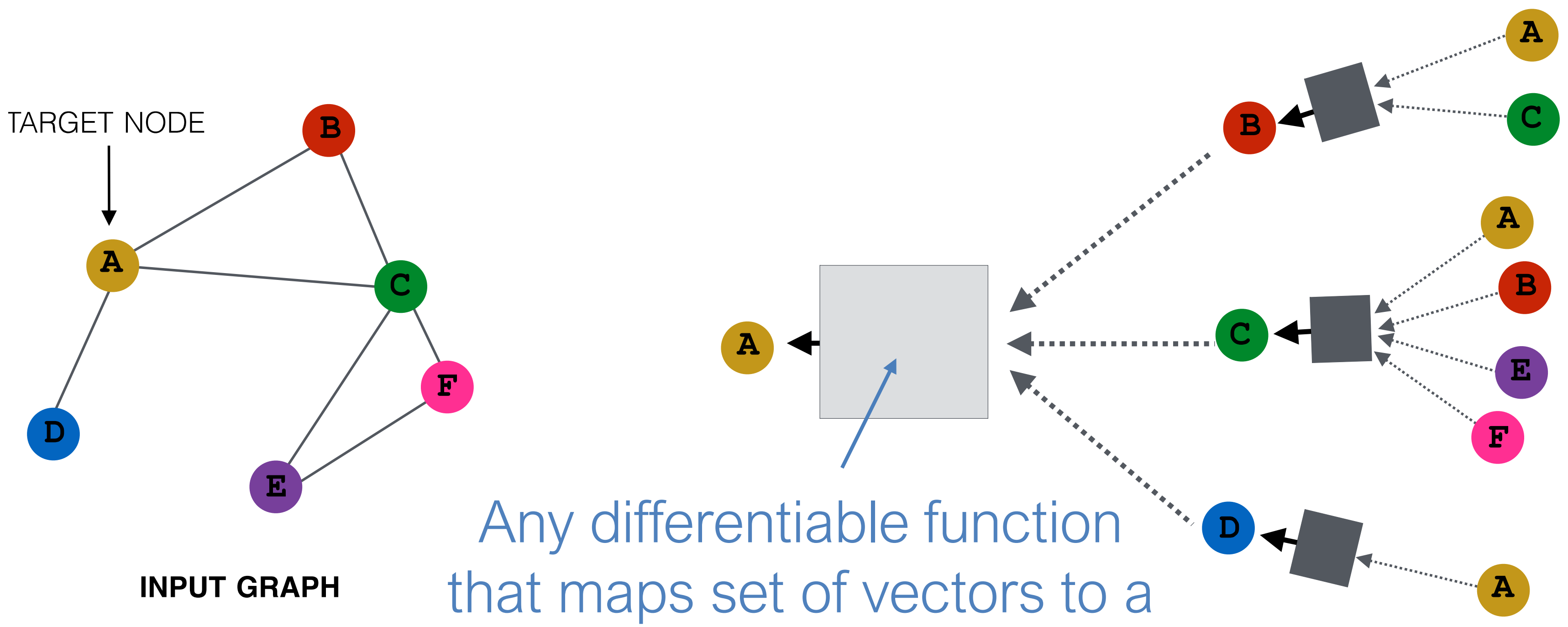
So far we have aggregated the neighbor messages by taking their (weighted) average, can we do better?





Hamilton et al., 2017. *Inductive Representation Learning on Large Graphs*. NIPS

GraphSAGE Idea



Any differentiable function that maps set of vectors to a single vector.

$$\mathbf{h}_v^k = \sigma\left(\left[\mathbf{A}_k \cdot \boxed{\text{AGG}(\{\mathbf{h}_u^{k-1}, \forall u \in N(v)\})}, \mathbf{B}_k \mathbf{h}_v^{k-1}\right]\right)$$

GraphSAGE Differences

- Simple neighborhood aggregation:

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \mathbf{B}_k \mathbf{h}_v^{k-1}\right)$$

- GraphSAGE:

concatenate self embedding and neighbor embedding

$$\mathbf{h}_v^k = \sigma\left(\left[\mathbf{W}_k \cdot \underbrace{\boxed{\mathrm{AGG}\left(\{\mathbf{h}_u^{k-1}, \forall u \in N(v)\}\right)}}_{\text{generalized aggregation}}, \mathbf{B}_k \mathbf{h}_v^{k-1}\right]\right)$$

GraphSAGE Variants

- Mean:

$$\text{AGG} = \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|}$$

- Pool
  - Transform neighbor vectors and apply symmetric vector function

element-wise mean/max

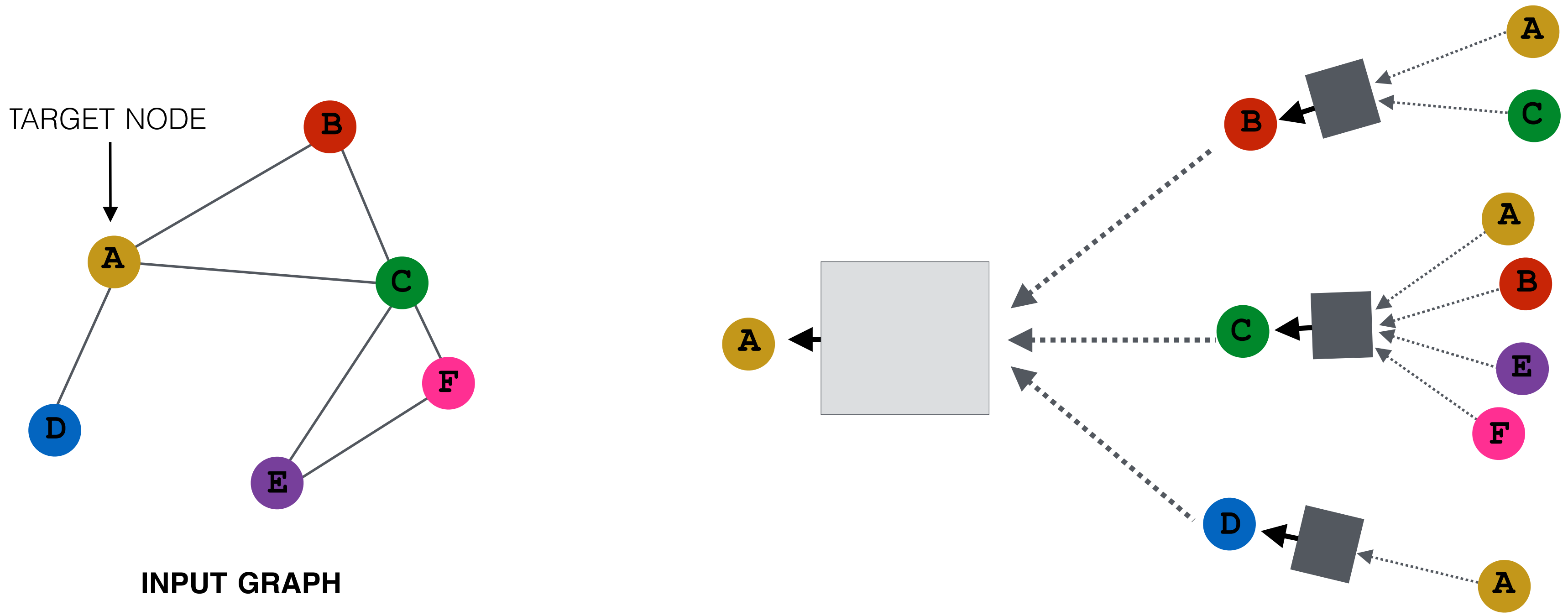
$$\text{AGG} = \boxed{\gamma}\left(\{\mathbf{Q}\mathbf{h}_u^{k-1}, \forall u \in N(v)\}\right)$$

- LSTM:
  - Apply LSTM (Long Short-Term Memory) to random permutation of neighbors.

$$\text{AGG} = \text{LSTM}\left([\mathbf{h}_u^{k-1}, \forall u \in \pi(N(v))]\right)$$

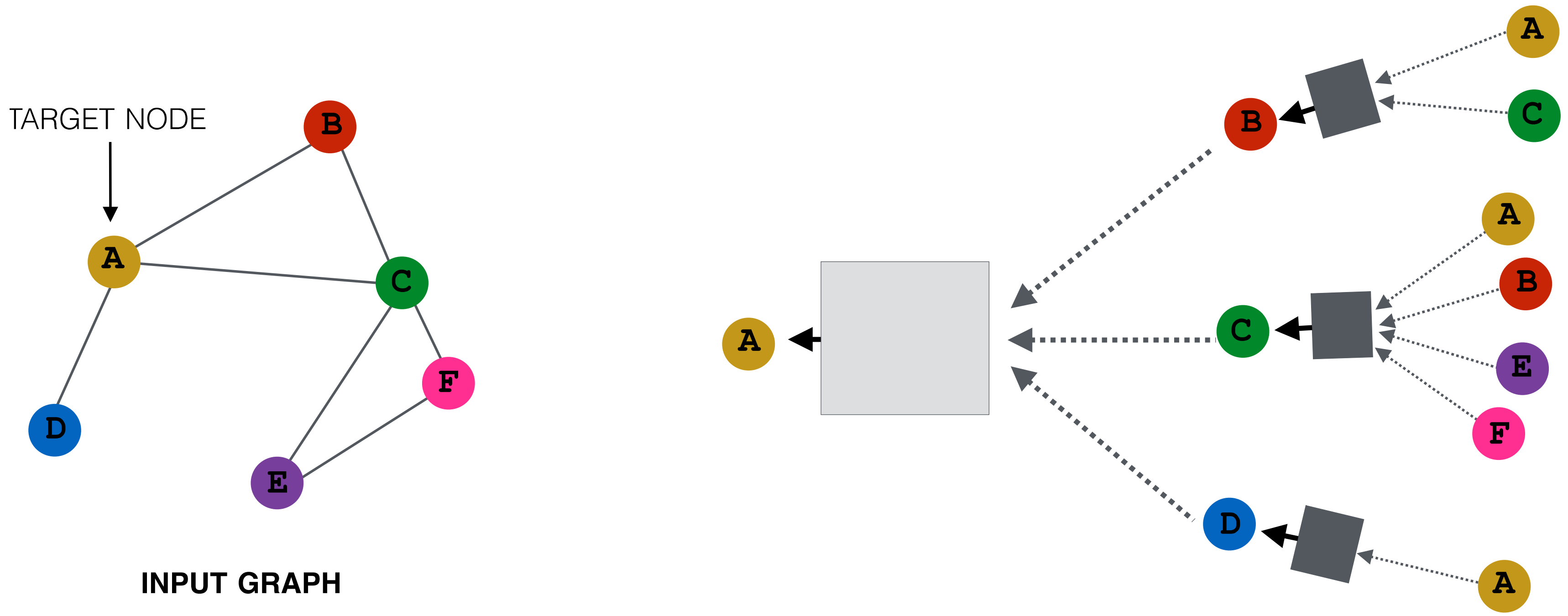
Neighborhood Aggregation

- **Basic idea:** Nodes aggregate "messages" from their neighbors using neural networks



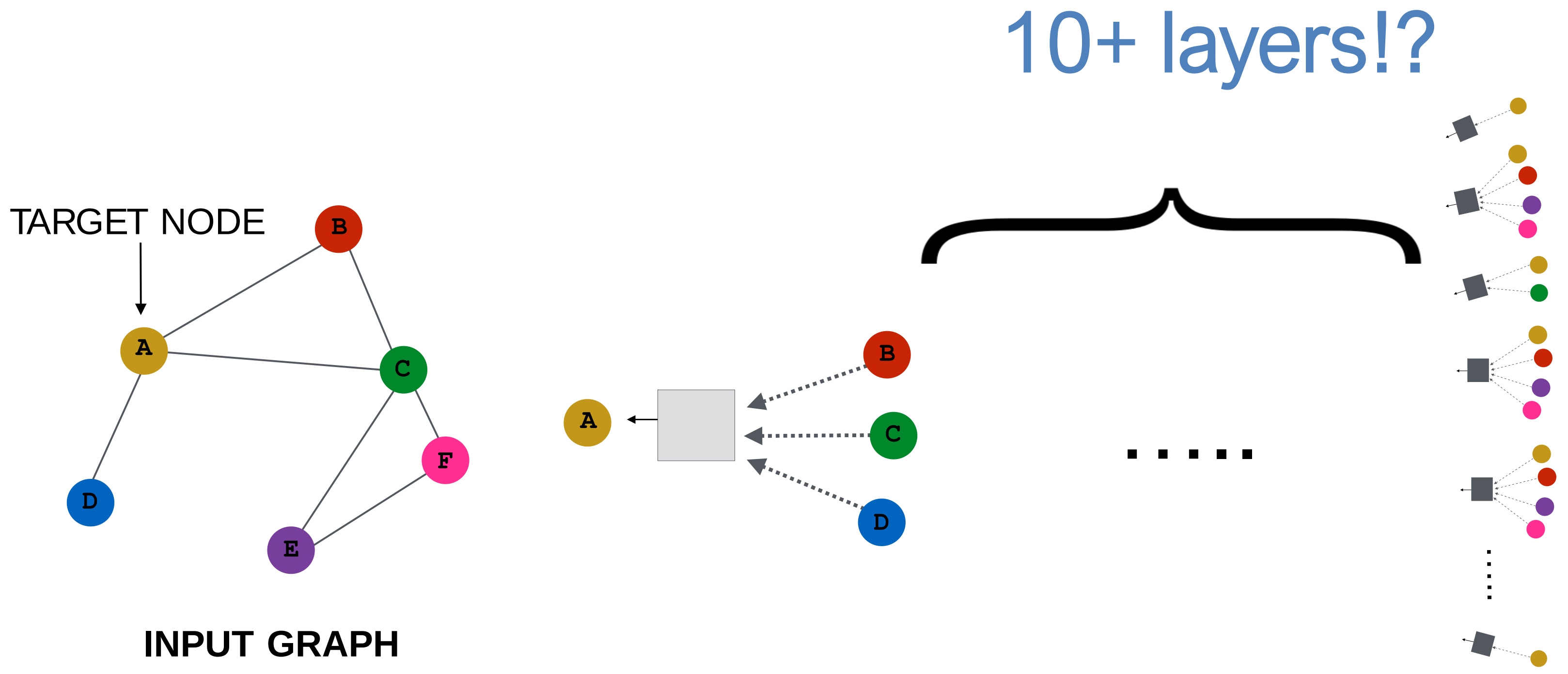
Neighborhood Aggregation

- GCNs and GraphSAGE **generally only 2-3 layers deep.**



Neighborhood Aggregation

- **But what if we want to go deeper?**



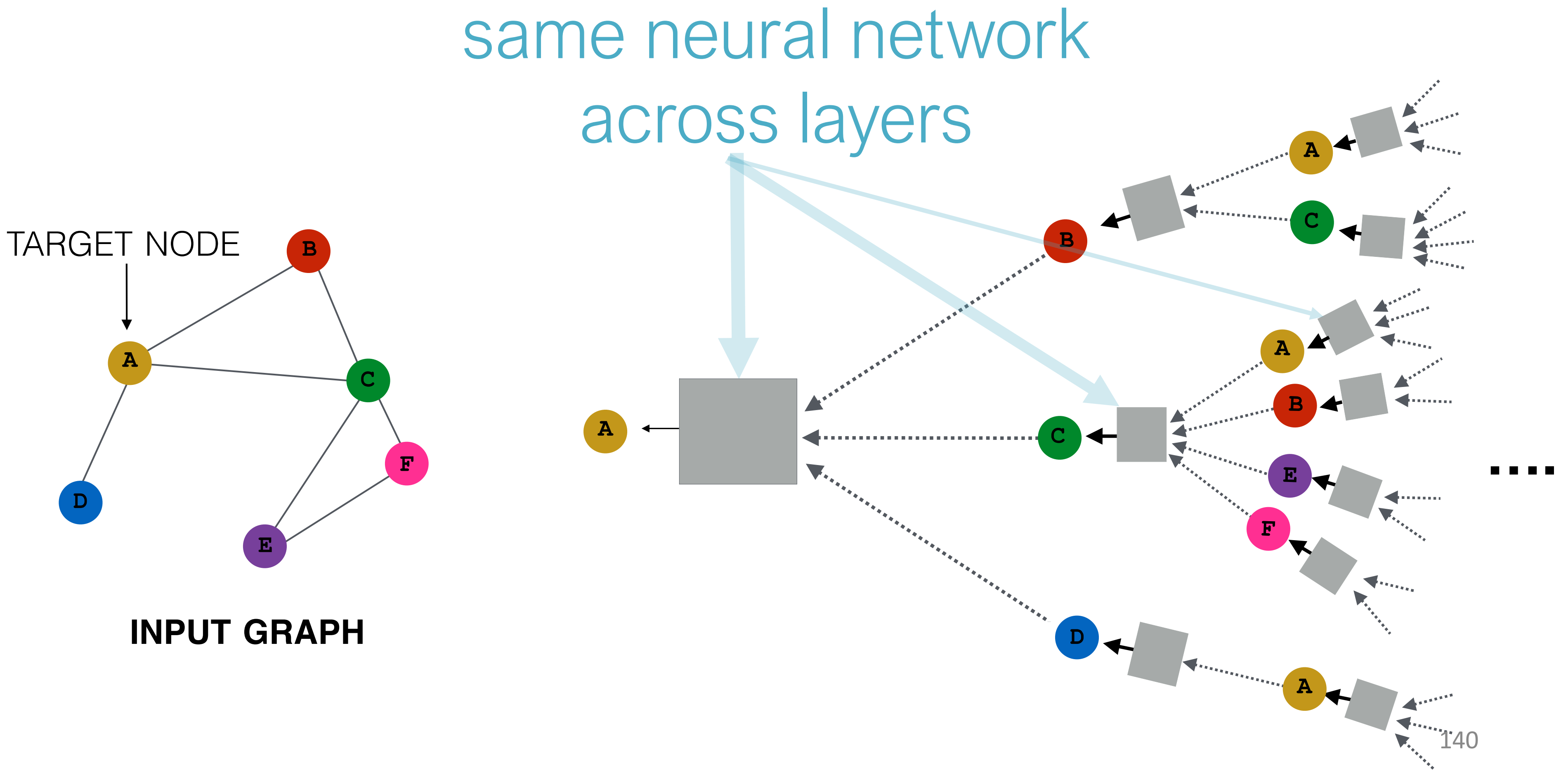
Gated Graph Neural Networks

- How can we build models with many layers of neighborhood aggregation?
- **Challenges:**
  - Overfitting from too many parameters.
  - Vanishing/exploding gradients during backpropagation.
- **Idea:** Use techniques from modern recurrent neural networks

Li et al., 2016. Gated Graph Sequence Neural Networks. ICLR.

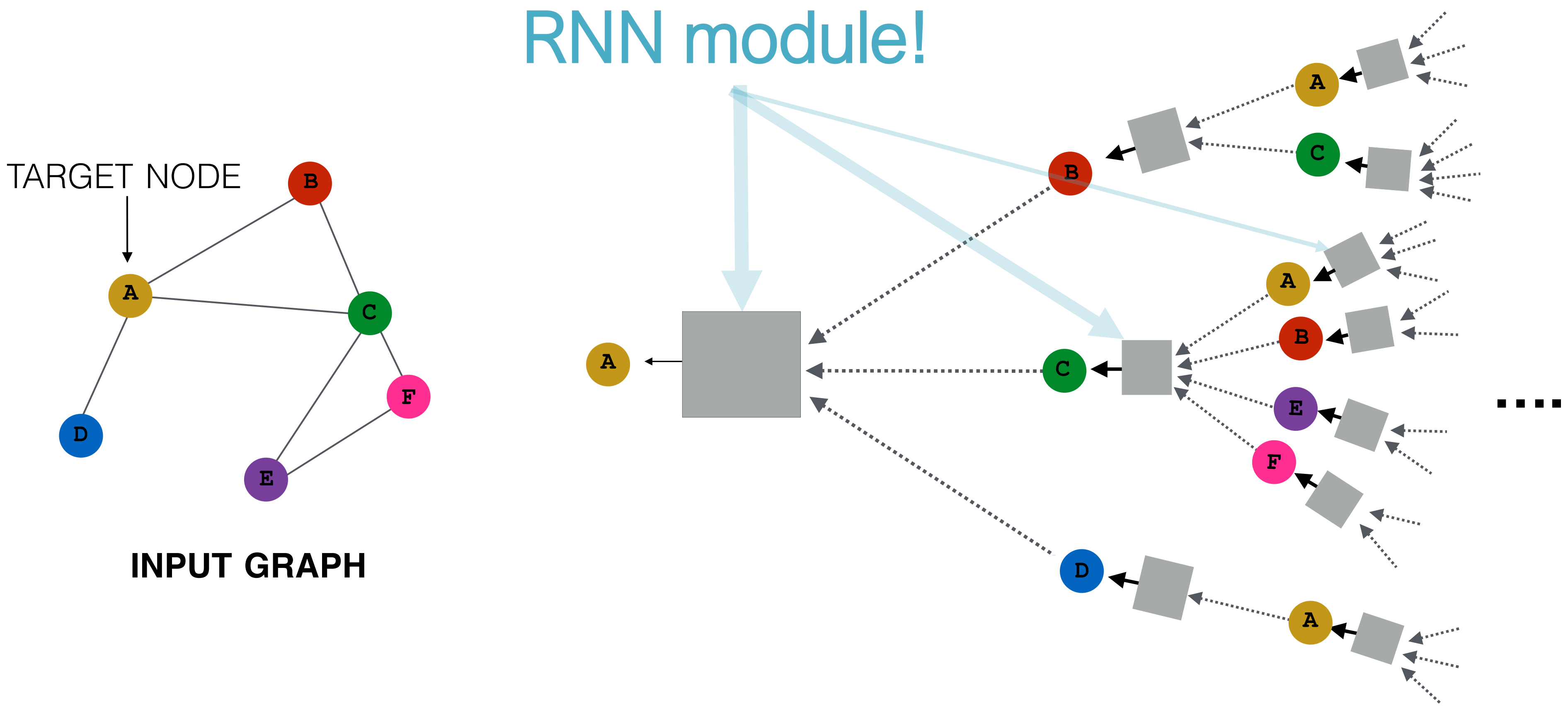
Gated Graph Neural Networks

- **Idea 1:** Parameter sharing across layers.



Gated Graph Neural Networks

- **Idea 2:** Recurrent state update.



Summary

- **Graph convolutional networks**
  - Average neighborhood information and stack neural networks.
- **GraphSAGE**
  - Generalized neighborhood aggregation.
- **Gated Graph Neural Networks**
  - Neighborhood aggregation + RNNs

Acknowledgement

Most slides adopted from the following tutorial

William Hamilton, Rex Ying, Jure Leskovec and Rok Sosic, Representation Learning on Networks. Held at WWW 2018 (April 24, Lyon, France).