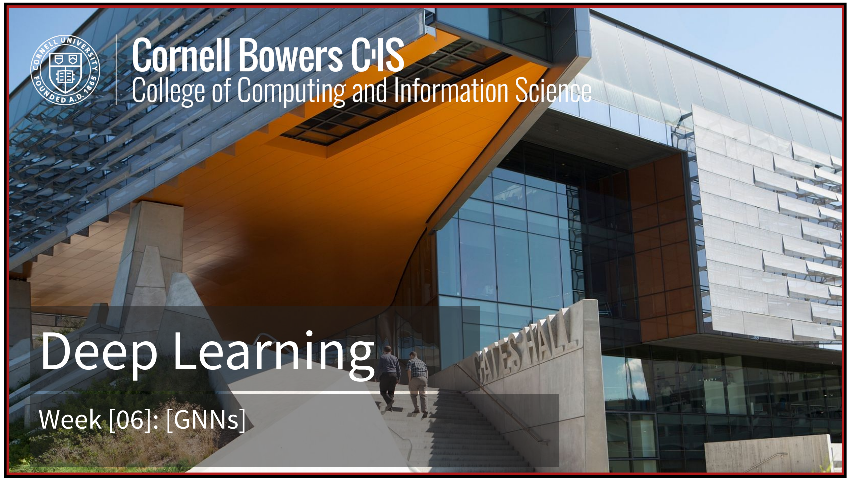


Cornell Bowers C·IS
College of Computing and Information Science

Deep Learning

Week [06]: [GNNs]

Cornell Bowers C·IS

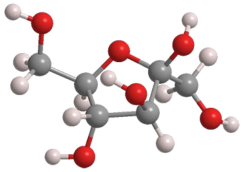
Logistics

- HW2 is due today
  - One extra slip day because of huggingface maintenance
  - Run the cells in the coding assignment
  - Turn off GPUs after using
- Project Proposal is due March 7
- HW 3 will be released at the end of the week
- We won't release HW solutions
- Change in office hours
  - Zach's office hours will be 4-5 on Wednesday's (instead of 6-7pm)
  - Varsha's office hours will be 1-2pm Tuesday's (instead of 9-10am)

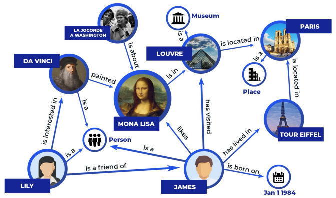
Cornell Bowers C·IS

What are Graphs?

Graphs are a general language for describing and analyzing entities with relations/interactions



Molecule

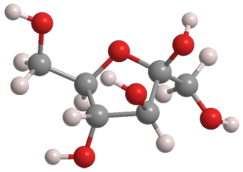


Knowledge Graph

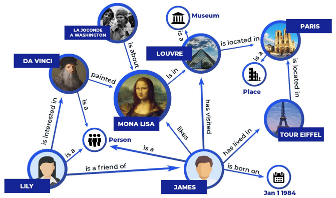
Cornell Bowers C·IS

What are Graphs?

Graphs = Nodes + Edges



Molecule

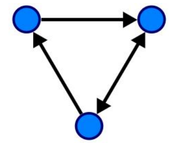


Knowledge Graph

Graph: Directed vs Undirected

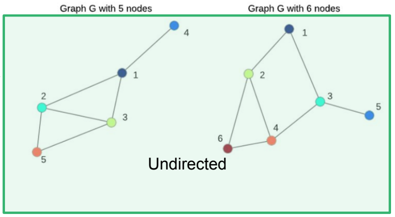
How the edges link the nodes allows us to distinguish between undirected graphs vs directed graphs

Graph G with 3 nodes



Directed

What we will focus on now

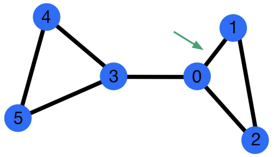


Undirected

Examples:
- ❑ Phone Calls
- ❑ Following on Twitter

Examples:
- ❑ Academic collaborations
- ❑ Friendships on Facebook

Adjacency Matrix - *A*

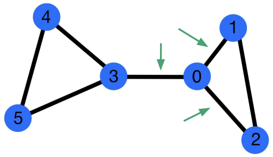


A represents the edges in a given graph

$A_{i,j}$ = 1 if an edge exists between nodes i and j, else 0

$$A = \begin{bmatrix} & 1 & & & & \\ 1 & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

Degree Matrix - *D*

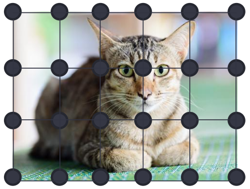


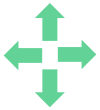
D is a diagonal matrix, where each diagonal entry represents the degree of each node in a given graph

$D_{i,i}$ = degree(i)

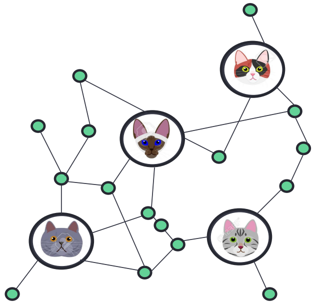
$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix} \qquad D = \begin{bmatrix} 3 & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

GraphML vs NLP vs CV



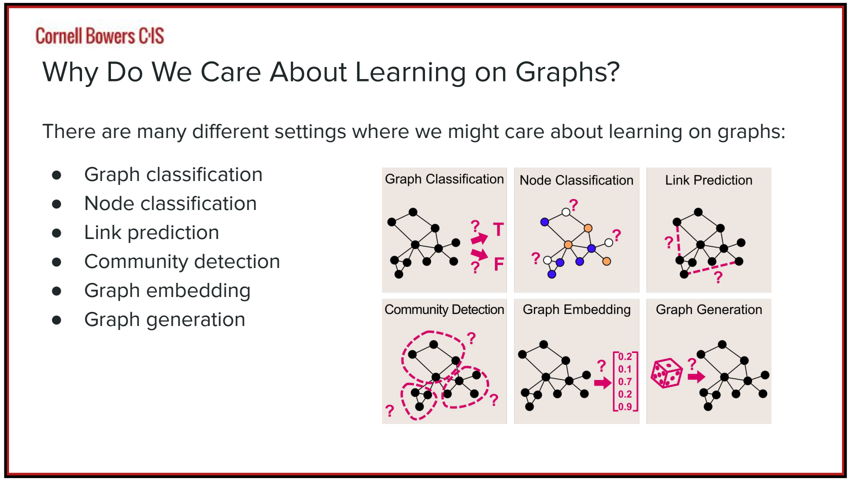


VS.



The cat sat on the mat.

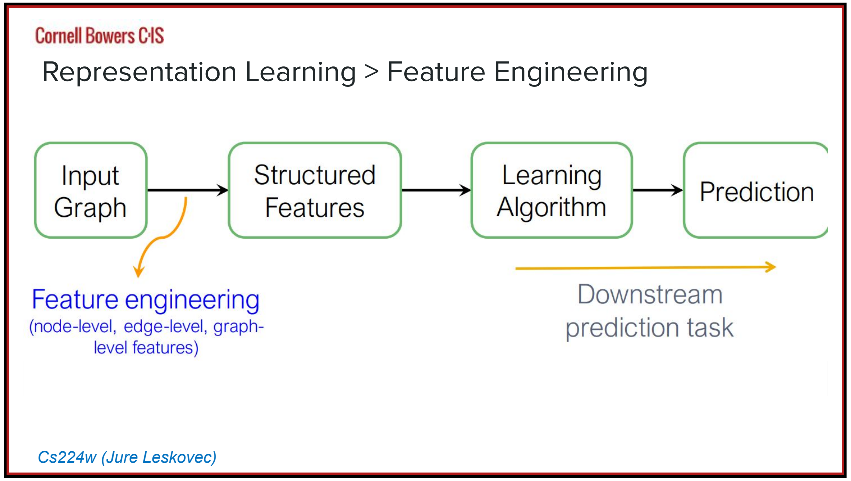
- ➡ No spatial locality (unlike grids)
- ➡ No rank ordering or fixed reference point



Why Do We Care About Learning on Graphs?

There are many different settings where we might care about learning on graphs:

- Graph classification
- Node classification
- Link prediction
- Community detection
- Graph embedding
- Graph generation



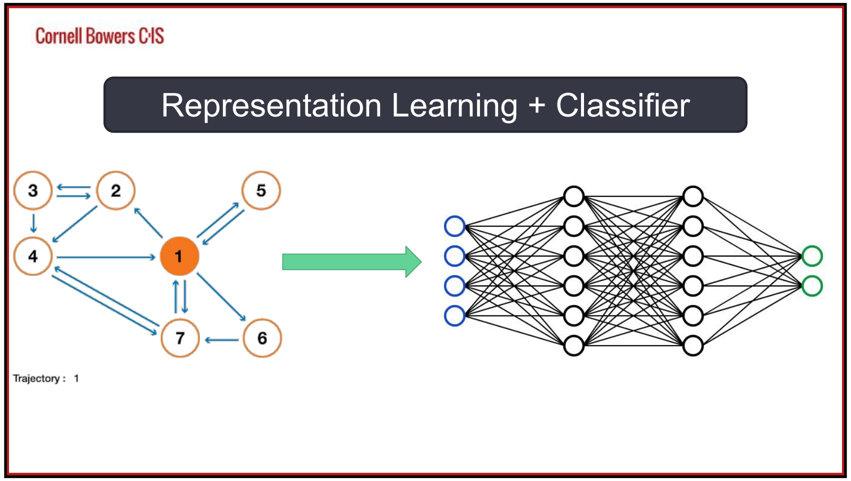
Representation Learning > Feature Engineering

Input Graph → Structured Features → Learning Algorithm → Prediction

Feature engineering
(node-level, edge-level, graph-level features)

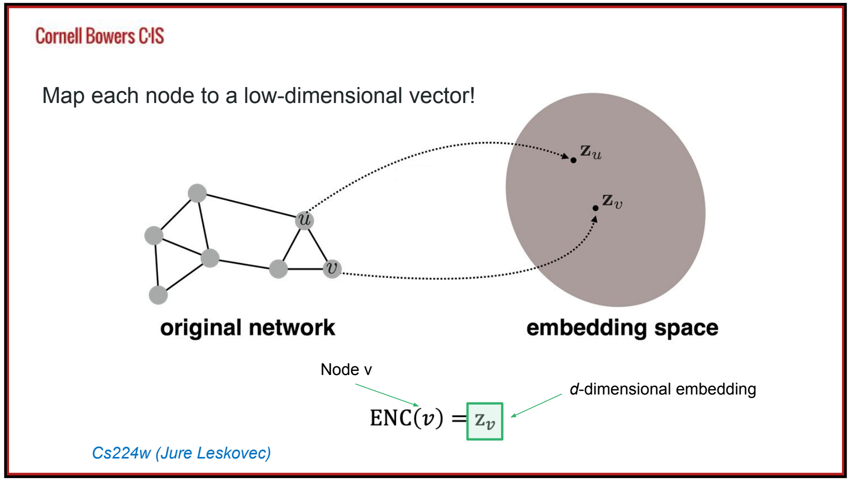
Downstream prediction task

Cs224w (Jure Leskovec)



Representation Learning + Classifier

Trajectory : 1



Map each node to a low-dimensional vector!

original network

embedding space

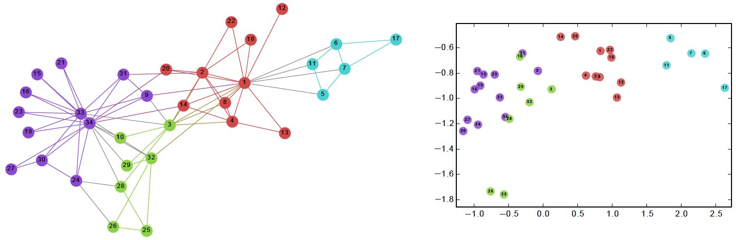
Node v

d-dimensional embedding

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

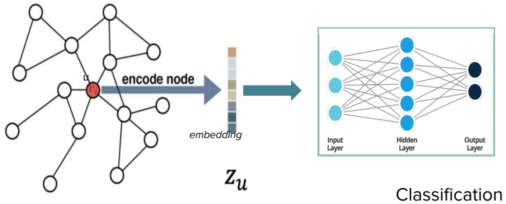
Cs224w (Jure Leskovec)

Example Graph mapped into 2 dimensions



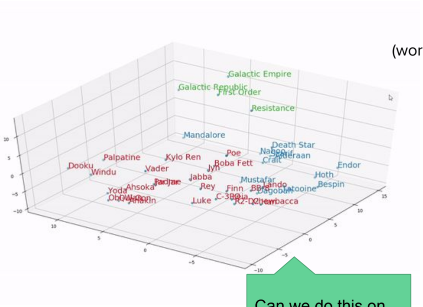
(a) Input: Karate Graph (b) Output: Representation

Classify Embeddings!



Classification

Brief Review: word2vec



(words close in sentences → close in embedding space)

List of sentences

"During the Battle of Endor, the Death Star II's energy shield was destroyed..."

W W W W W

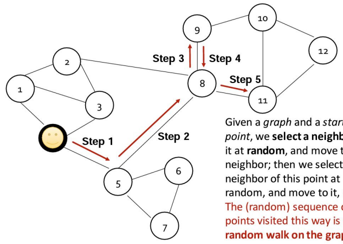
"In the third film, Anakin becomes Vader when..."

"Samuel L Jackson portrayed Mace Windu in the prequel trilogy..."

Can we do this on graphs?

DeepWalk: word2vec For Graphs

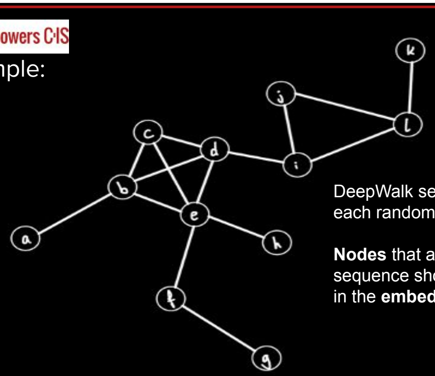
This is **exactly** the same optimization as word2vec, but we instead optimize over **sequences of random walks on a graph**.



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

Cornell Bowers C·IS

Example:



DeepWalk selects the next node to traverse to in each random walk **purely at random (unbiased)**

Nodes that are close together in the random walk sequence should be **embedded closer together** in the **embedding space!**

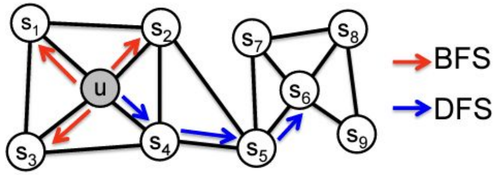
These are the **"sentences" that we generate!**

Random walk :

Cornell Bowers C·IS

node2vec: The Introduction of Bias...

node2vec = DeepWalk + control over local vs global exploration (via two additional hyperparameters that we won't discuss in detail)



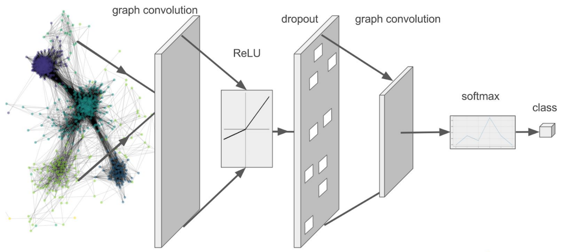
Breadth First Search (BFS) {s1, s2, s3} **Local** microscopic view — Homophily

Depth First Search (DFS) {s4, s5, s6} **Global** macroscopic view — Structural equivalence

Grover and Leskovec., ACM SIGKDD, 2016

Cornell Bowers C·IS

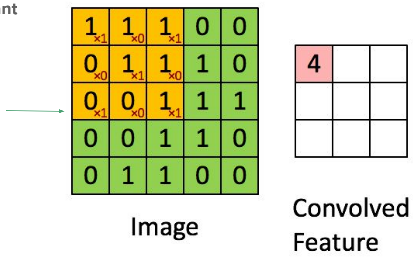
GRAPH NEURAL NETWORKS



Cornell Bowers C·IS

Convolutional Layer in CNN

Translation-invariant



$1_{\times 1}$	$1_{\times 0}$	$1_{\times 1}$	0	0
$0_{\times 0}$	$1_{\times 1}$	$1_{\times 0}$	1	0
$0_{\times 1}$	$0_{\times 0}$	$1_{\times 1}$	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

How about for non-Euclidean data? Can we do something similar with graphs?

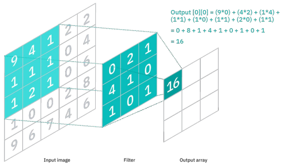
Can you perform convolutions on graphs? What would that look like? Will you run into any problems?

Locality: you can tell a lot about a particular **pixel** based on the properties of their neighbors

Locality vs Homophily

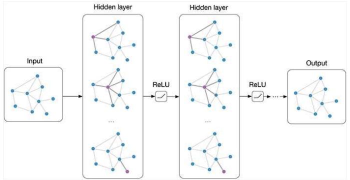
Homophily: you can tell a lot about a particular **node** based on the properties of their neighbors

Image Convolutions


Output [0][0] = (9*0) + (4*2) + (1*4) + (1*1) + (1*0) + (1*1) + (2*0) + (1*1)
= 0 + 8 + 1 + 4 + 1 + 0 + 1 + 0 + 1
= 16
Input image
Filter
Output array

Generate next layer embedding vectors for each **pixel** in an input image by **aggregating** the **transformed feature vectors** of each of the pixel's neighbors

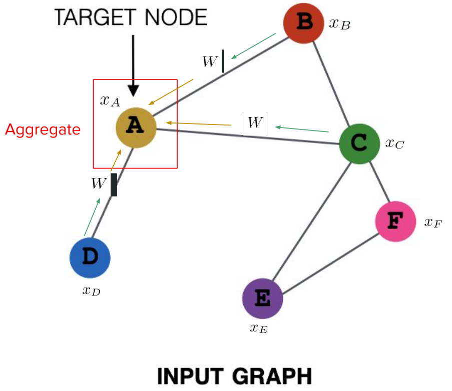
Graph Convolutions



Generate next layer embedding vectors for each **node** in an input graph by **aggregating** the **transformed feature vectors** of each of the node's neighbors

Let's look at a single layer of a graph convolution

Thomas Kipf
PhD @ University of Amsterdam
Currently: Research scientist @ Google Brain



$$h_A = \sigma\left(\underbrace{\sum_{u \in N(A)} \frac{\overbrace{x_u W}^{\text{Transform}}}{|N(A)|}}_{\text{Aggregate}}\right)$$

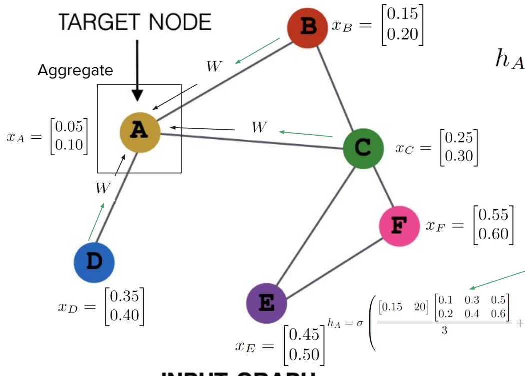
Note: Aggregation function **MUST be permutation-invariant**!

- Mean()
- Sum()
- Max()

Let's choose Mean() for now...

TARGET NODE

Aggregate


B
$x_B = \begin{bmatrix}0.15\\0.20\end{bmatrix}$
W
W
$x_A = \begin{bmatrix}0.05\\0.10\end{bmatrix}$
A
C
$x_C = \begin{bmatrix}0.25\\0.30\end{bmatrix}$
W
F
$x_F = \begin{bmatrix}0.55\\0.60\end{bmatrix}$
D
$x_D = \begin{bmatrix}0.35\\0.40\end{bmatrix}$
E
$x_E = \begin{bmatrix}0.45\\0.50\end{bmatrix}$

INPUT GRAPH

$$h_A = \sigma\left(\sum_{u \in N(A)} \frac{x_u W}{|N(A)|}\right)$$

$$W = \begin{bmatrix}0.1 & 0.3 & 0.5\\0.2 & 0.4 & 0.6\end{bmatrix}$$

$$h_A = \sigma\left(\frac{\begin{bmatrix}0.15 & 20\end{bmatrix}\begin{bmatrix}0.1 & 0.3 & 0.5\\0.2 & 0.4 & 0.6\end{bmatrix}}{3} + \frac{\begin{bmatrix}0.25 & 30\end{bmatrix}\begin{bmatrix}0.1 & 0.3 & 0.5\\0.2 & 0.4 & 0.6\end{bmatrix}}{3} + \frac{\begin{bmatrix}0.35 & 40\end{bmatrix}\begin{bmatrix}0.1 & 0.3 & 0.5\\0.2 & 0.4 & 0.6\end{bmatrix}}{3}\right)$$

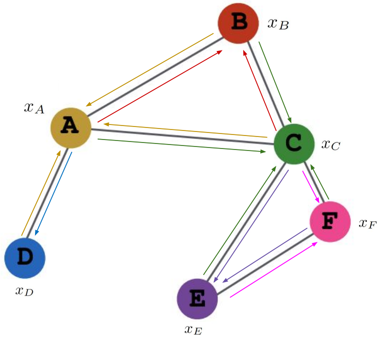
$$= \sigma\left(\begin{bmatrix}0.145 & 0.2 & 0.255\end{bmatrix}\right)$$

e.g ReLU

$$= \begin{bmatrix}0.145 & 0.2 & 0.255\end{bmatrix}$$

Cornell Bowers C·IS

We repeat this process of **transforming** and **aggregating** neighboring embedding vectors for **every** node in the graph

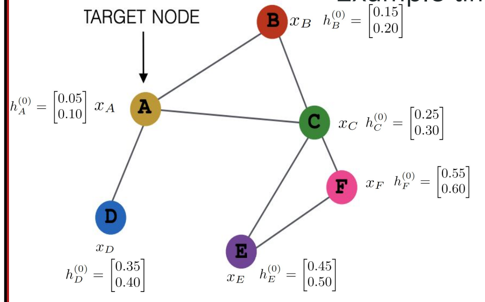


INPUT GRAPH

Cornell Bowers C·IS

Example time!

TARGET NODE

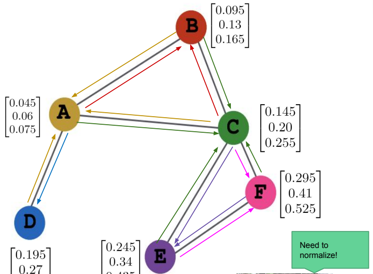

B
x_B $h_B^{(0)} = \begin{bmatrix}0.15\\0.20\end{bmatrix}$
$h_A^{(0)} = \begin{bmatrix}0.05\\0.10\end{bmatrix}$ x_A
A
C
x_C $h_C^{(0)} = \begin{bmatrix}0.25\\0.30\end{bmatrix}$
F
x_F $h_F^{(0)} = \begin{bmatrix}0.55\\0.60\end{bmatrix}$
D
x_D
$h_D^{(0)} = \begin{bmatrix}0.35\\0.40\end{bmatrix}$
E
x_E $h_E^{(0)} = \begin{bmatrix}0.45\\0.50\end{bmatrix}$

INPUT GRAPH

1. $x_v \rightarrow h_v^{(0)}$

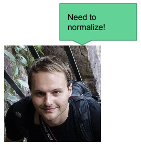
Let's also give these some values...

Cornell Bowers C·IS


B
$\begin{bmatrix}0.095\\0.13\\0.165\end{bmatrix}$
$\begin{bmatrix}0.045\\0.06\\0.075\end{bmatrix}$
A
C
$\begin{bmatrix}0.145\\0.20\\0.255\end{bmatrix}$
F
$\begin{bmatrix}0.295\\0.41\\0.525\end{bmatrix}$
D
$\begin{bmatrix}0.195\\0.27\\0.345\end{bmatrix}$
$\begin{bmatrix}0.245\\0.34\\0.435\end{bmatrix}$
E

INPUT GRAPH

Need to normalize!



1. $x_v \rightarrow h_v^{(0)}$

2. $$H^{(0)} = \begin{bmatrix}h_A^{(0)} & h_B^{(0)} & h_C^{(0)} & h_D^{(0)} & h_E^{(0)} & h_F^{(0)}\end{bmatrix}^T = \begin{bmatrix}0.05 & 0.10\\0.15 & 0.20\\0.25 & 0.30\\0.35 & 0.40\\0.45 & 0.50\\0.55 & 0.60\end{bmatrix}$$

3. **Transform**!

$$W_0 = \begin{bmatrix}0.1 & 0.3 & 0.5\\0.2 & 0.4 & 0.6\end{bmatrix} \quad H^{(0)}W_0 = \begin{bmatrix}0.045 & 0.06 & 0.075\\0.095 & 0.13 & 0.165\\0.145 & 0.20 & 0.255\\0.195 & 0.27 & 0.345\\0.245 & 0.34 & 0.435\\0.295 & 0.41 & 0.525\end{bmatrix}$$

4. Define adjacency matrix:

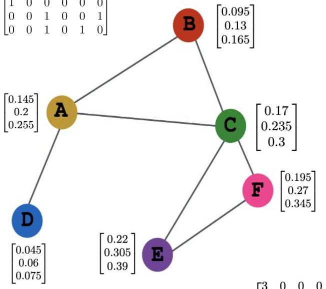
5. **Aggregate!**

$$A = \begin{bmatrix}0 & 1 & 1 & 1 & 0 & 0\\1 & 0 & 1 & 0 & 0 & 0\\1 & 1 & 0 & 0 & 1 & 1\\1 & 0 & 0 & 0 & 0 & 0\\0 & 0 & 1 & 0 & 0 & 1\\0 & 0 & 1 & 0 & 1 & 0\end{bmatrix} \quad AH^{(0)}W_0 = \begin{bmatrix}0.435 & 0.6 & 0.765\\0.19 & 0.26 & 0.33\\0.68 & 0.94 & 1.2\\0.045 & 0.06 & 0.075\\0.44 & 0.61 & 0.78\\0.39 & 0.54 & 0.69\end{bmatrix}$$

You left multiply by D^{-1}. Compute D^{-1} and then think about why we multiply by D^{-1}.

(As a reminder, D is the diagonal matrix where each entry is a the degree of each vertex)

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



INPUT GRAPH

$$D = \begin{bmatrix} 3 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 \end{bmatrix}$$

1. $x_v \rightarrow h_v^{(0)}$

2. $$H^{(0)} = \begin{bmatrix} h_A^{(0)} & h_B^{(0)} & h_C^{(0)} & h_D^{(0)} & h_E^{(0)} & h_F^{(0)} \end{bmatrix}^T = \begin{bmatrix} 0.05 & 0.10 \\ 0.15 & 0.20 \\ 0.25 & 0.30 \\ 0.35 & 0.40 \\ 0.45 & 0.50 \\ 0.55 & 0.60 \end{bmatrix}$$

3. **Transform**!

$$W_0 = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix} \quad H^{(0)}W_0 = \begin{bmatrix} 0.045 & 0.06 & 0.075 \\ 0.095 & 0.13 & 0.165 \\ 0.145 & 0.20 & 0.255 \\ 0.195 & 0.27 & 0.345 \\ 0.245 & 0.34 & 0.435 \\ 0.295 & 0.41 & 0.525 \end{bmatrix}$$

4. Define adjacency matrix:

5. **Aggregate!**

$$AH^{(0)}W_0 = \begin{bmatrix} 0.435 & 0.6 & 0.765 \\ 0.19 & 0.26 & 0.33 \\ 0.68 & 0.94 & 1.2 \\ 0.045 & 0.06 & 0.075 \\ 0.44 & 0.61 & 0.78 \\ 0.39 & 0.54 & 0.69 \end{bmatrix}$$

6. Normalize

$$D^{-1}AH^{(0)}W_0 = \begin{bmatrix} 0.145 & 0.2 & 0.255 \\ 0.095 & 0.13 & 0.165 \\ 0.17 & 0.235 & 0.3 \\ 0.045 & 0.06 & 0.075 \\ 0.22 & 0.305 & 0.39 \\ 0.195 & 0.27 & 0.345 \end{bmatrix}$$

7. Pass through non-linearity

$$H^{(1)} = \sigma(D^{-1}AH^{(0)}W_0)$$
$$= ReLU(D^{-1}AH^{(0)}W_0)$$
$$= Z$$

This is the **only**
thing we **optimize**!

$$H^{(1)} = \sigma(\underbrace{\boxed{D^{-1}A}}_{\text{2. Aggregate}}\underbrace{\boxed{H^{(0)}W_0}}_{\text{1. Transform}})$$

This is the **only**
thing we **optimize**!

$$Z = \underbrace{\boxed{\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}}}_{\text{2. Aggregate}}\underbrace{\boxed{X\Theta}}_{\text{1. Transform}}$$

This is often the equation you will see for GCNs!

https://arxiv.org/pdf/1609.02907.pdf

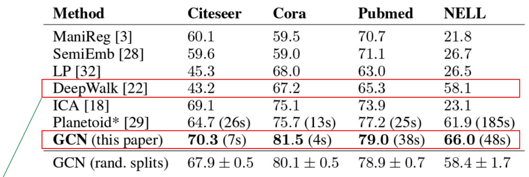
Cornell Bowers C·IS

GCN > Random Walks

$$H^{(1)} = \sigma(D^{-1}AH^{(0)}W_0)$$

Just plug test nodes here!

Table 2: Summary of results in terms of classification accuracy (in percent).



Method	Citeseer	Cora	Pubmed	NELL
ManiReg [3]	60.1	59.5	70.7	21.8
SemiEmb [28]	59.6	59.0	71.1	26.7
LP [32]	45.3	68.0	63.0	26.5
DeepWalk [22]	43.2	67.2	65.3	58.1
ICA [18]	69.1	75.1	73.9	23.1
Planetoid* [29]	64.7 (26s)	75.7 (13s)	77.2 (25s)	61.9 (185s)
GCN (this paper)	**70.3** (7s)	**81.5** (4s)	**79.0** (38s)	**66.0** (48s)
GCN (rand. splits)	67.9 ± 0.5	80.1 ± 0.5	78.9 ± 0.7	58.4 ± 1.7

With random walks, what we're optimizing are the **final embedding vectors** themselves, **not** weights…

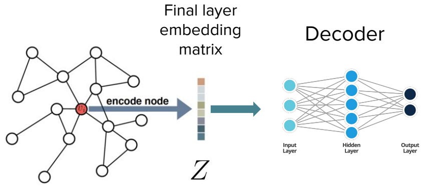
…so for every **new/unseen node** that we're given (e.g in a test set), we have to use SGD AGAIN to optimize their embeddings, which is **computationally expensive!**

THIS IS A BIG REASON WHY WE USE A WEIGHT MATRIX!

Cornell Bowers C·IS

What do we do with Z?

$$H^{(1)} = \sigma(D^{-1}AH^{(0)}W_0)$$
$$= ReLU(D^{-1}AH^{(0)}W_0)$$
$$= Z$$



Depends on the downstream prediction task:

- Feed *Z* into a **MLP + Softmax decoder** for **node-level classification/regression**
- For **graph-level predictions** (e.g classifying an entire graph), can **concat/sum/mean all vectors in *Z***, and then feed this long vector into a **MLP**
  - Just like in CNNs!

Cornell Bowers C·IS

Stacking GCN Layers

$$H^{(1)} = \sigma(D^{-1}AH^{(0)}W_0)$$

Input to the next layer

Note: new weight matrix! Weight matrices in GNNs are **layer-specific.**

$$H^{(2)} = \sigma(D^{-1}AH^{(1)}W_1)$$

But D and A never change!

Cornell Bowers C·IS

Stacking GCN Layers

Final GCN update rules:

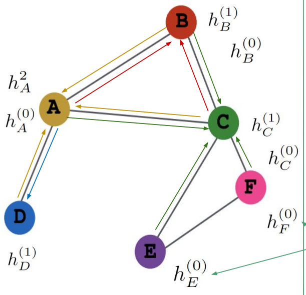
Node-level update rule: $$h_v^{l+1} = \sigma\left(\sum_{u \in N(v)} \frac{h_u^l W_l}{|N(v)|}\right)$$

Graph-level update rule: $$H^{(l+1)} = \sigma(D^{-1}AH^{(l)}W_l)$$

Let's just keep adding more layers, right?

BIG problem!

Stacking GCN Layers



INPUT GRAPH

$$H^{(2)} = \sigma(D^{-1}AH^{(1)}W_1)$$

In order to calculate A's h_A^2 vector, we need to calculate h_u^1 for each u in Neighbors(A)

H^3 looks at neighbors' neighbors' neighbors, etc…this becomes MASSIVE on large graphs

$$h_A^2 = \sigma\left(\sum_{u \in N(A)} \frac{h_u^1 W_1}{|N(A)|}\right)$$

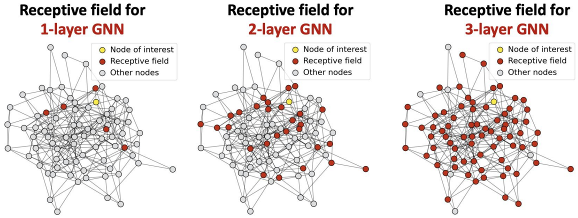
In order to calculate each node u's h_u^1 vector, we need to calculate $h_{u'}^0$ for each u' in Neighbors(u)

Just to calculate h_A^2, we need to look at A's neighbors' neighbors

$$h_u^1 = \sigma\left(\sum_{u' \in N(u)} \frac{h_{u'}^0 W_0}{|N(u)|}\right)$$

Therefore, number of layer in graph neural networks is a **very important hyperparameter!**

The over-smoothing problem


Receptive field for 1-layer GNN

Receptive field for 2-layer GNN

Receptive field for 3-layer GNN

Receptive field: the set of all nodes that are used to calculate an l-th layer embedding vector for a node v

Here, we encounter the **over-smoothing problem**, where final-layer node embeddings (in *Z*) become highly similar.

GraphSAGE

2 **BIG problems** with GCNs:

Problem 1: h_v^{L+1} doesn't aggregate h_v^L

$$\boxed{h_v^{l+1} = \sigma\left(\sum_{u \in N(v)} \frac{h_u^l W_l}{|N(v)|}\right)}$$

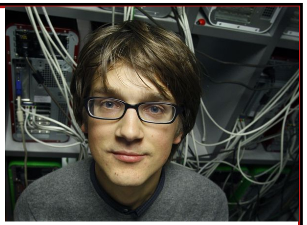
Solution 1: Add **self-loops**!

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

Now *v* will additionally sum their own embedding vector along with *v*'s neighbors!

Problem 2: Just Mean()? How about the rest?

Solution 2: Make the aggregation function a **hyperparameter**!



Jure Leskovec
Postdoc @ Cornell
Currently: Professor @ Stanford
Until very recently: Chief Scientist @ Pinterest
Created **node2vec**

https://arxiv.org/pdf/1706.02216.pdf

GraphSAGE > GCN

Table 1: Prediction results for the three datasets (micro-averaged F1 scores). Results for unsupervised and fully supervised GraphSAGE are shown. Analogous trends hold for macro-averaged scores.

	Citation		Reddit		PPI	
Name	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1	Unsup. F1	Sup. F1
Random	0.206	0.206	0.043	0.042	0.396	0.396
Raw features	0.575	0.575	0.585	0.585	0.422	0.422
DeepWalk	0.565	0.565	0.324	0.324	—	—
DeepWalk + features	0.701	0.701	0.691	0.691	—	—
GraphSAGE-GCN	0.742	0.772	**0.908**	0.930	0.465	0.500
GraphSAGE-mean	0.778	0.820	0.897	0.950	0.486	0.598
GraphSAGE-LSTM	0.788	0.832	**0.907**	**0.954**	0.482	**0.612**
GraphSAGE-pool	**0.798**	**0.839**	0.892	0.948	**0.502**	0.600

Simplifying GCNs

Remember this?

Graph-level update rule: $$H^{(l+1)} = \boxed{\times}(\boxed{D^{-1}A}H^{(l)}W_l)$$

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

Get rid of the non-linearities!

$$H^1 = SH^{(0)}W_0$$

Why does this work so well?

The strength of GNNs comes from their ability to **propagate node features**, not from non-linearities

https://arxiv.org/pdf/1902.07153.pdf

Write an expression for $H^{(l+1)}$

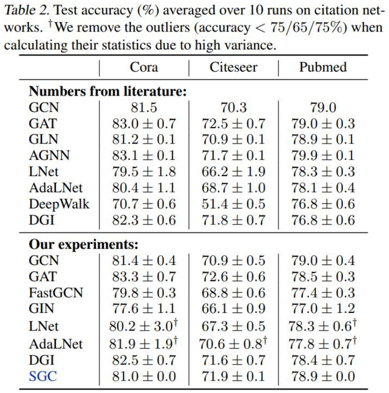


Table 2. Test accuracy (%) averaged over 10 runs on citation networks. †We remove the outliers (accuracy $<$ 75/65/75%) when calculating their statistics due to high variance.

	Cora	Citeseer	Pubmed
Numbers from literature:			
GCN	81.5	70.3	79.0
GAT	83.0 ± 0.7	72.5 ± 0.7	79.0 ± 0.3
GLN	81.2 ± 0.1	70.9 ± 0.1	78.9 ± 0.1
AGNN	83.1 ± 0.1	71.7 ± 0.1	79.9 ± 0.1
LNet	79.5 ± 1.8	66.2 ± 1.9	78.3 ± 0.3
AdaLNet	80.4 ± 1.1	68.7 ± 1.0	78.1 ± 0.4
DeepWalk	70.7 ± 0.6	51.4 ± 0.5	76.8 ± 0.6
DGI	82.3 ± 0.6	71.8 ± 0.7	76.8 ± 0.6
Our experiments:			
GCN	81.4 ± 0.4	70.9 ± 0.5	79.0 ± 0.4
GAT	83.3 ± 0.7	72.6 ± 0.6	78.5 ± 0.3
FastGCN	79.8 ± 0.3	68.8 ± 0.6	77.4 ± 0.3
GIN	77.6 ± 1.1	66.1 ± 0.9	77.0 ± 1.2
LNet	80.2 ± 3.0†	67.3 ± 0.5	78.3 ± 0.6†
AdaLNet	81.9 ± 1.9†	70.6 ± 0.8†	77.8 ± 0.7†
DGI	82.5 ± 0.7	71.6 ± 0.7	78.4 ± 0.7
SGC	81.0 ± 0.0	71.9 ± 0.1	78.9 ± 0.0

Summary

- Learning on graphs: Classify nodes and entire graphs, predict links or detect communities and even generate graphs and their embeddings
- Feature Engineering 😡 Representation Learning 😊
- Random Walks, DeepWalk + node2vec: word2vec on graphs, embed nearby nodes on the random walk closer together
- GCN: CNN on graphs, transform + aggregate neighbors. Homophily in GCNs similar to locality in CNNs.
- Over-smoothing problem: Can't stack too many layers