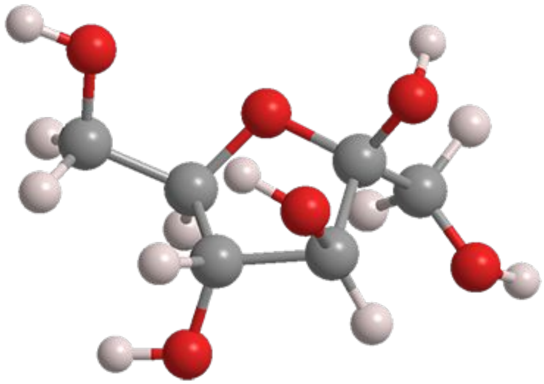
Learning On Graphs

Deepa Korani, Shagun Gupta, Cazamere Comrie, Madhav Aggarwal

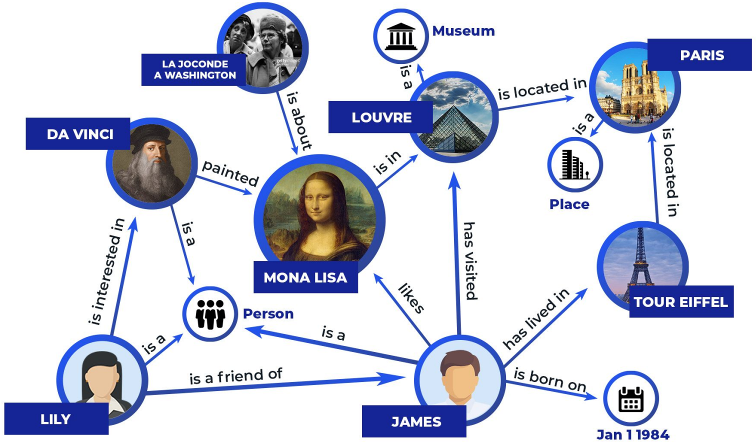
* in presenting order

What are Graphs?

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



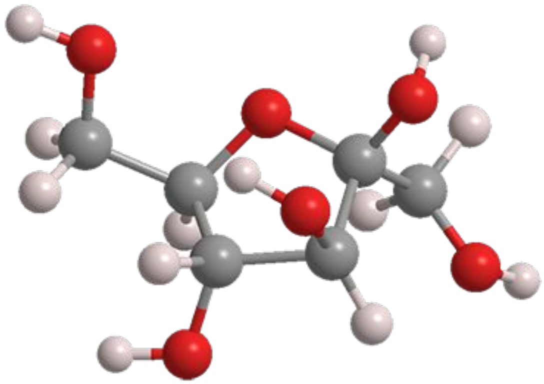
Molecule



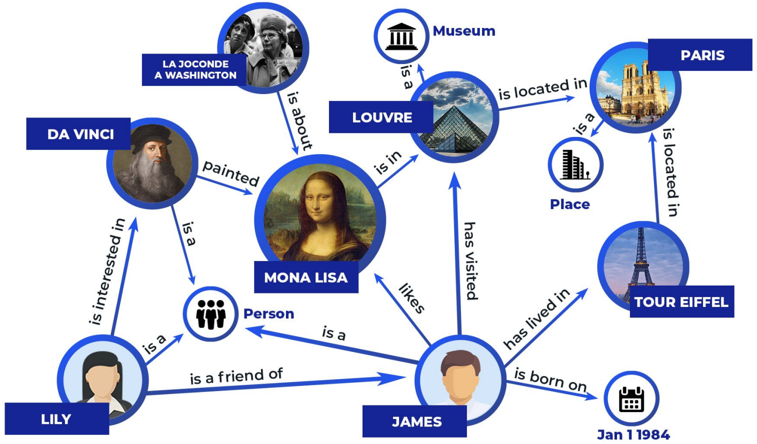
Knowledge Graph

What are Graphs?

Graphs = Nodes + Edges



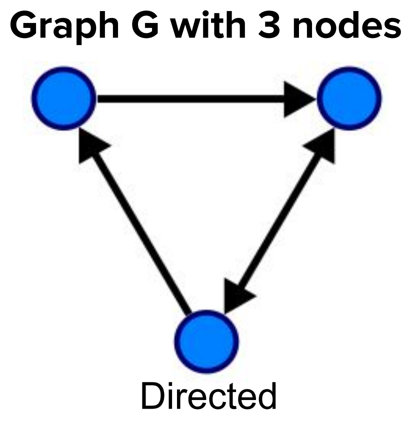
Molecule



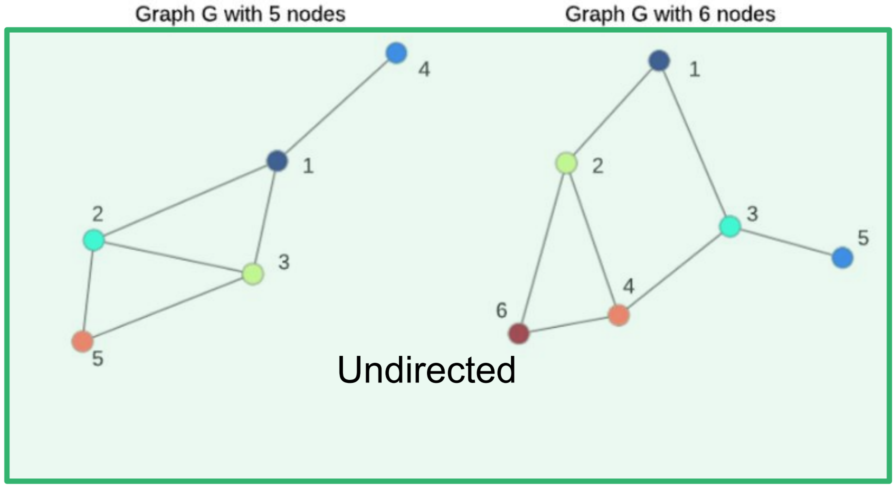
Knowledge Graph

Graph: Directed vs Undirected

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



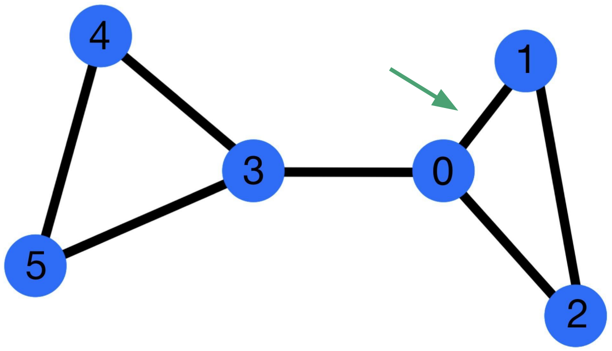
What we will focus on now



Examples:
- ❑ Phone Calls
- ❑ Following on Twitter

Examples:
- ❑ Academic collaborations
- ❑ Friendships on Facebook

Adjacency Matrix - A



A represents the edges in a given graph

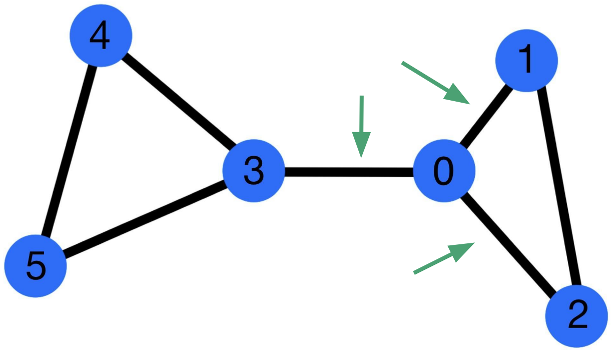
$A_{i,j} = 1$ if an edge exists between nodes i and j, else 0

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

Degree Matrix - *D*

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

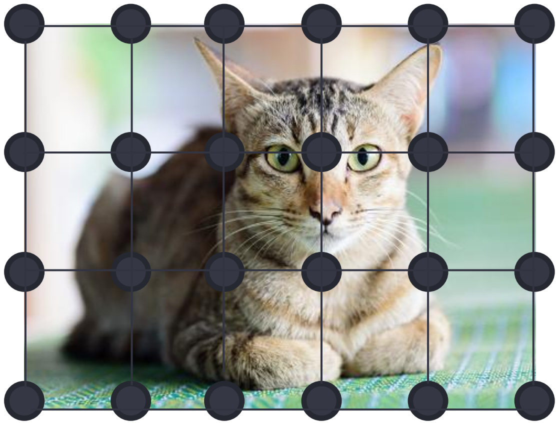
$D_{i,i}$ = degree(i)

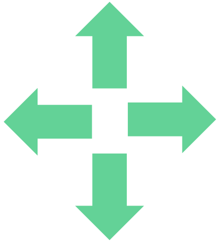


$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

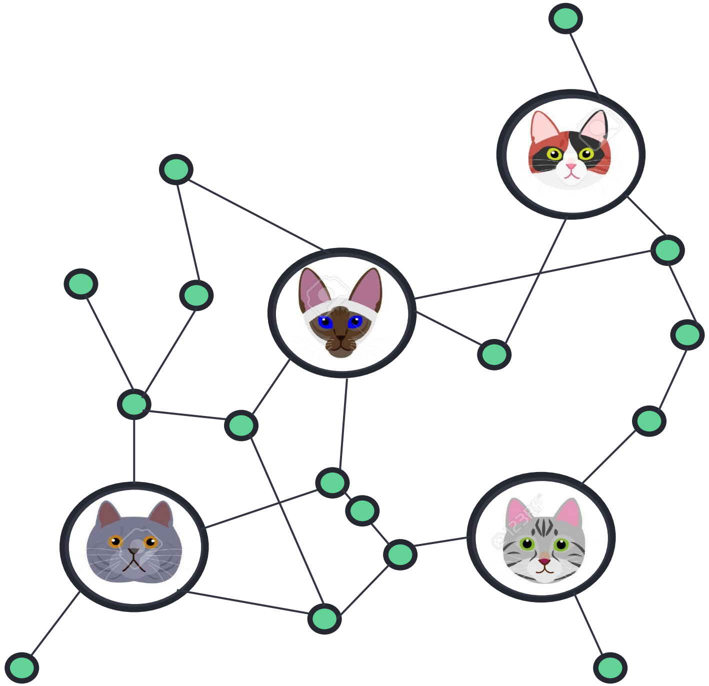
$$D = \begin{bmatrix} 3 & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \end{bmatrix}$$

GraphML vs NLP vs CV





VS.



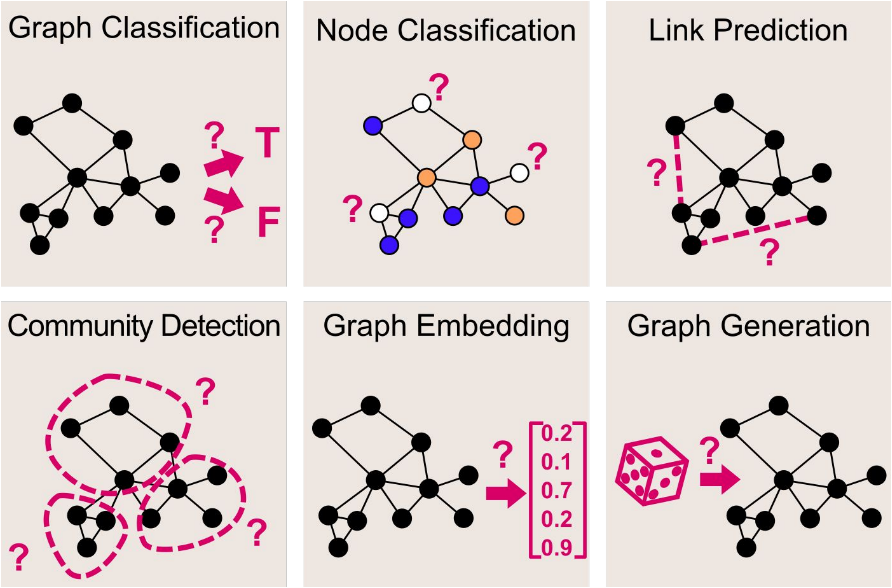
The cat sat on the mat.

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

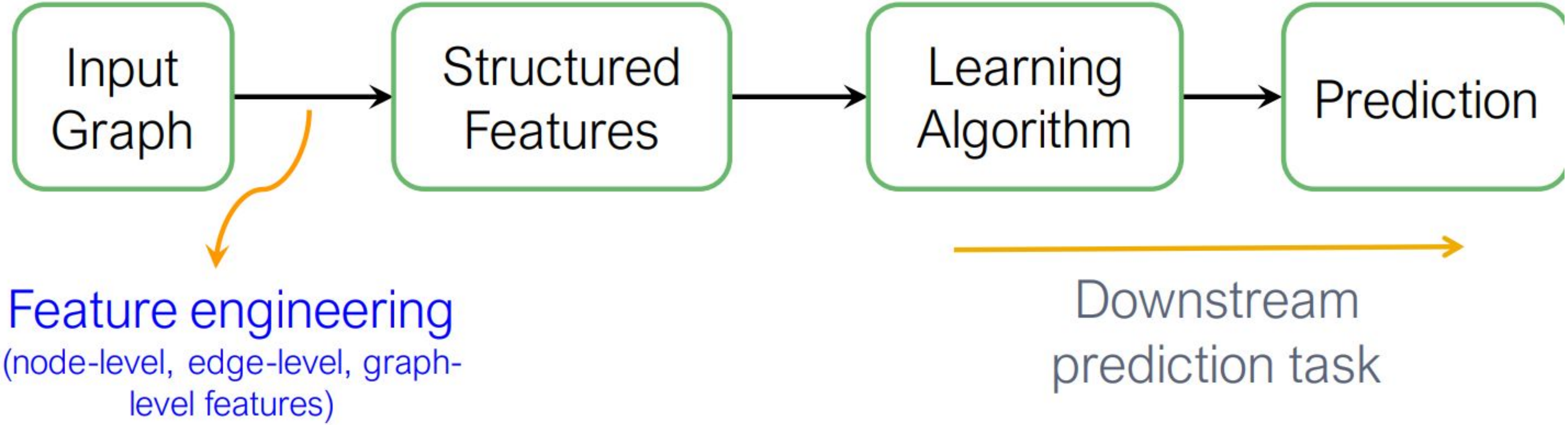
Why Do We Care About Learning on Graphs?

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

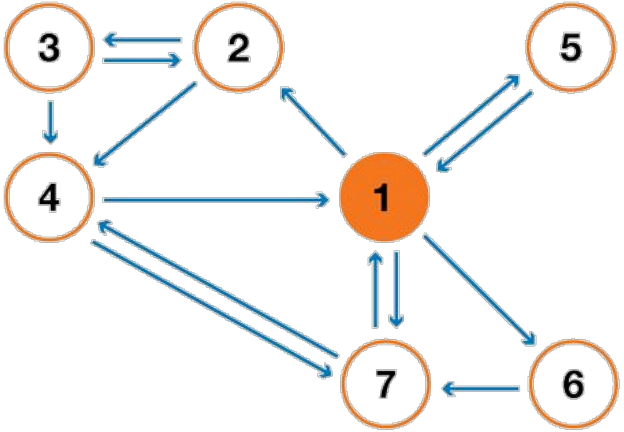
- Graph classification
- Node classification
- Link prediction
- Community detection
- Graph embedding
- Graph generation



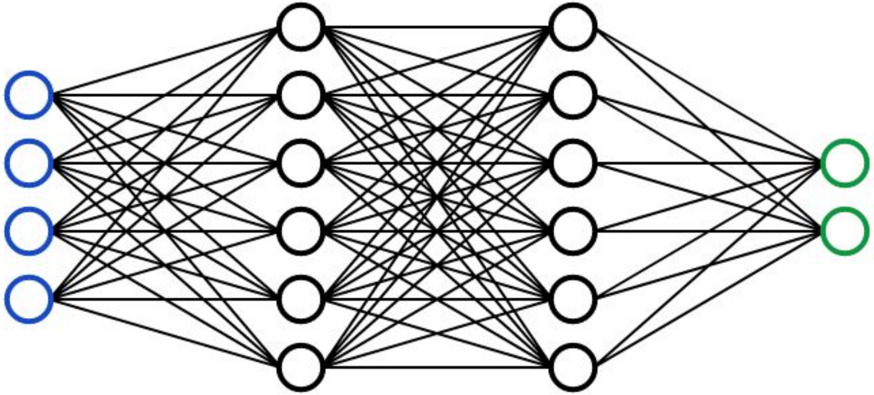
Representation Learning > Feature Engineering



Encoder-Decoder Paradigm

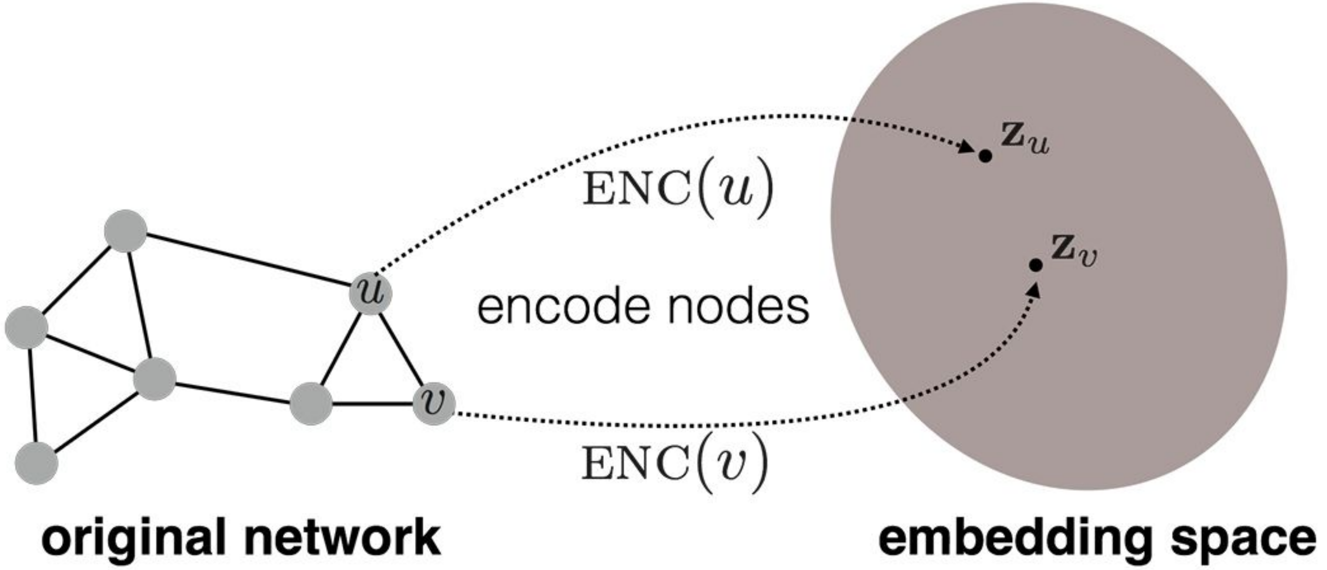


Trajectory : 1



Encoders

Maps each node to a low-dimensional vector



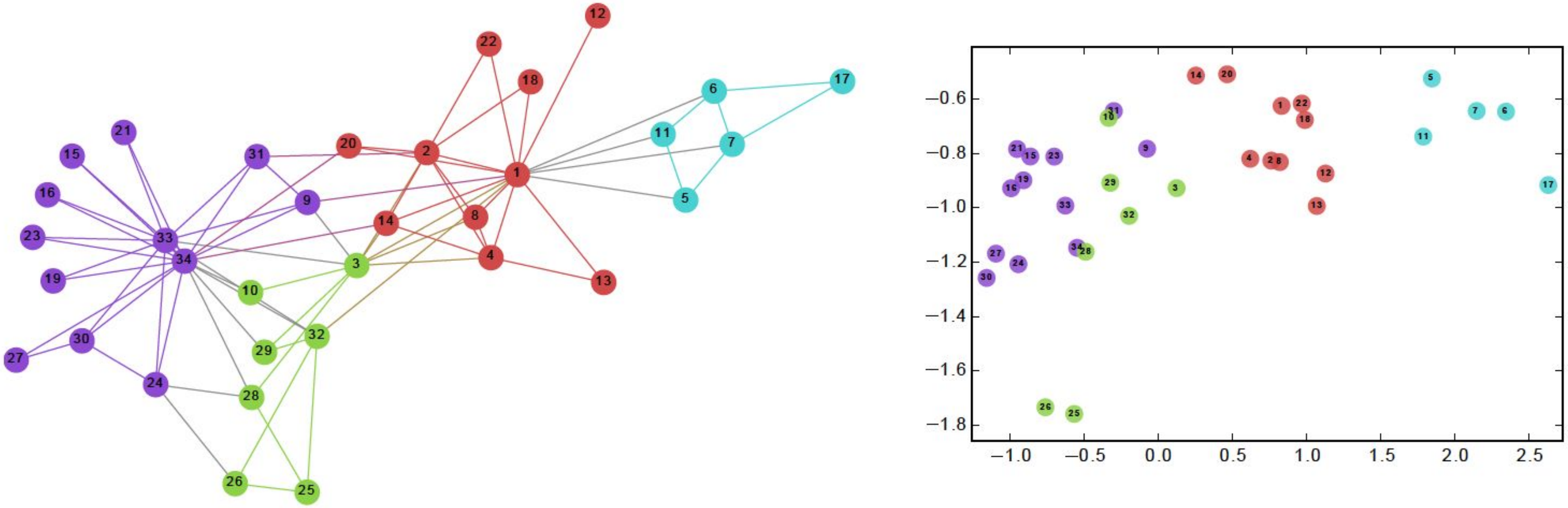
Node v

d-dimensional embedding

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

Cs224w (Jure Leskovec)

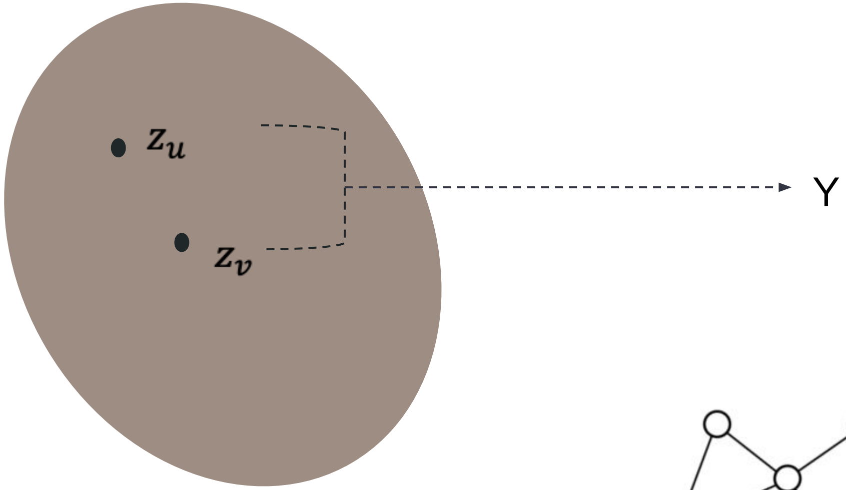
Encoder example (mapped into 2 dimensions)

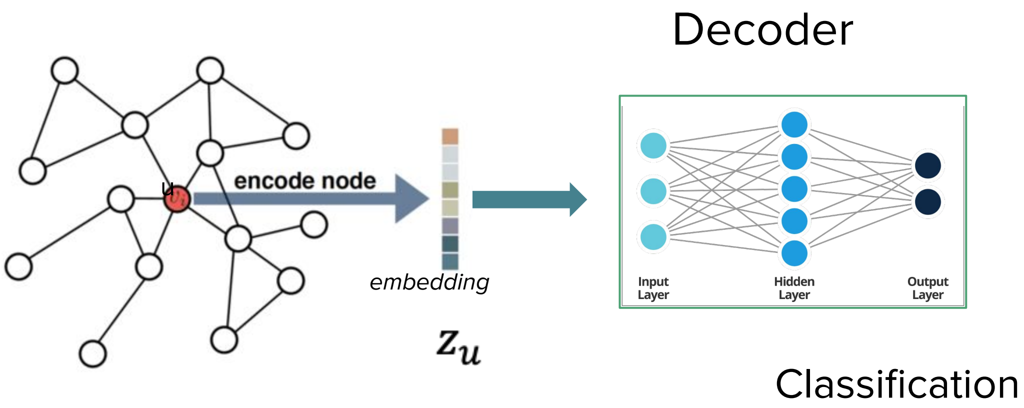


(a) Input: Karate Graph (b) Output: Representation

Decoders

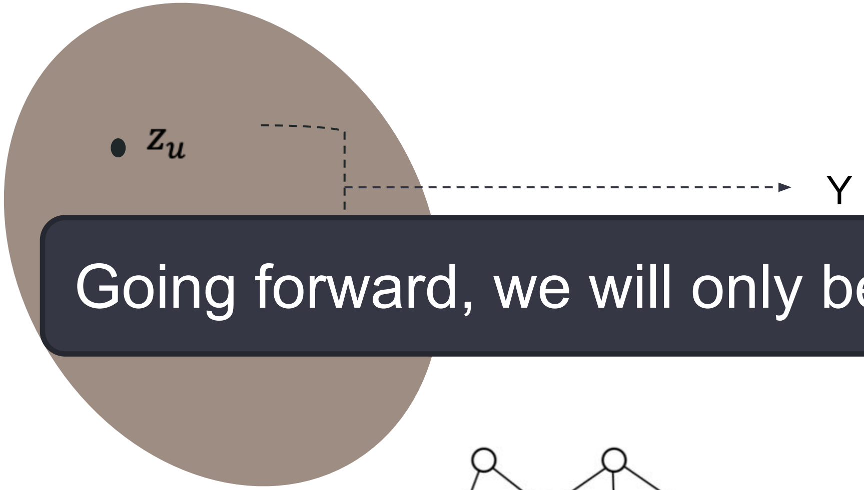
Predict Score based on embedding to match node similarity



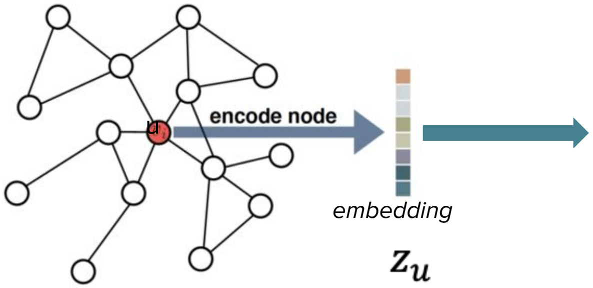


Decoders

Predict Score based on embedding to match node similarity



Going forward, we will only be discussing encoders!

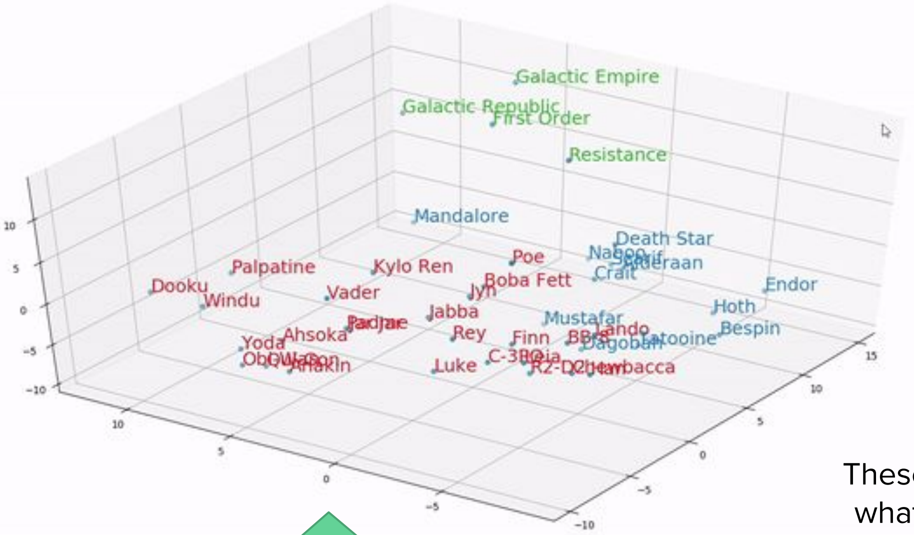


Link Prediction

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

Brief Pivot: word2vec

Encoder: maps **words** to **embedding vectors**



Can we do this on graphs?

List of sentences

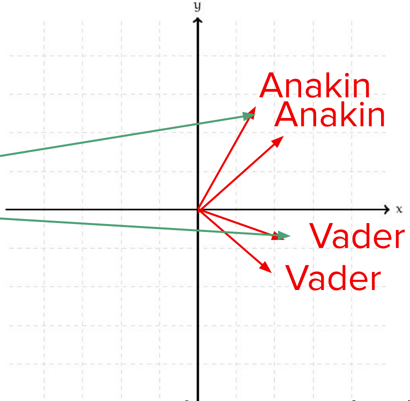
“During the Battle of Endor, the Death Star II’s energy shield was destroyed…

W W W W W

“In the third film, Anakin becomes Vader when…”

“Samuel L Jackson portrayed Mace Windu in the prequel trilogy…”

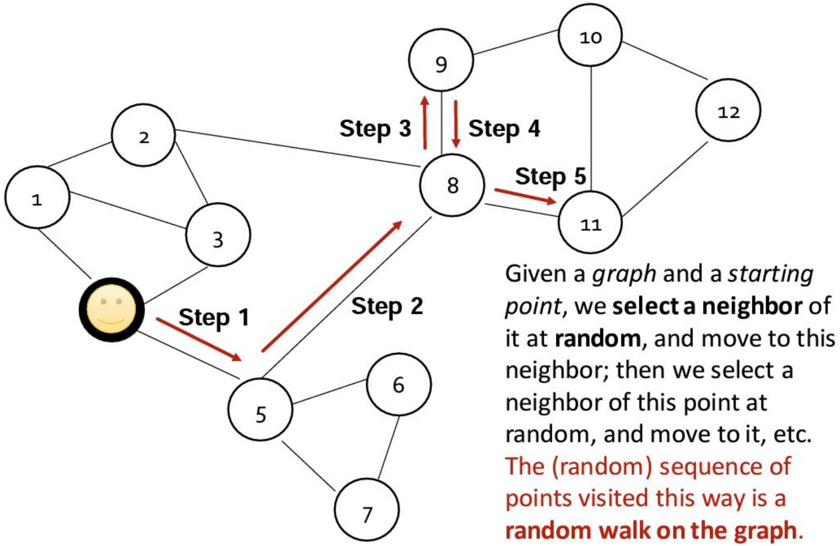
These are what we optimize (i.e SGD)!



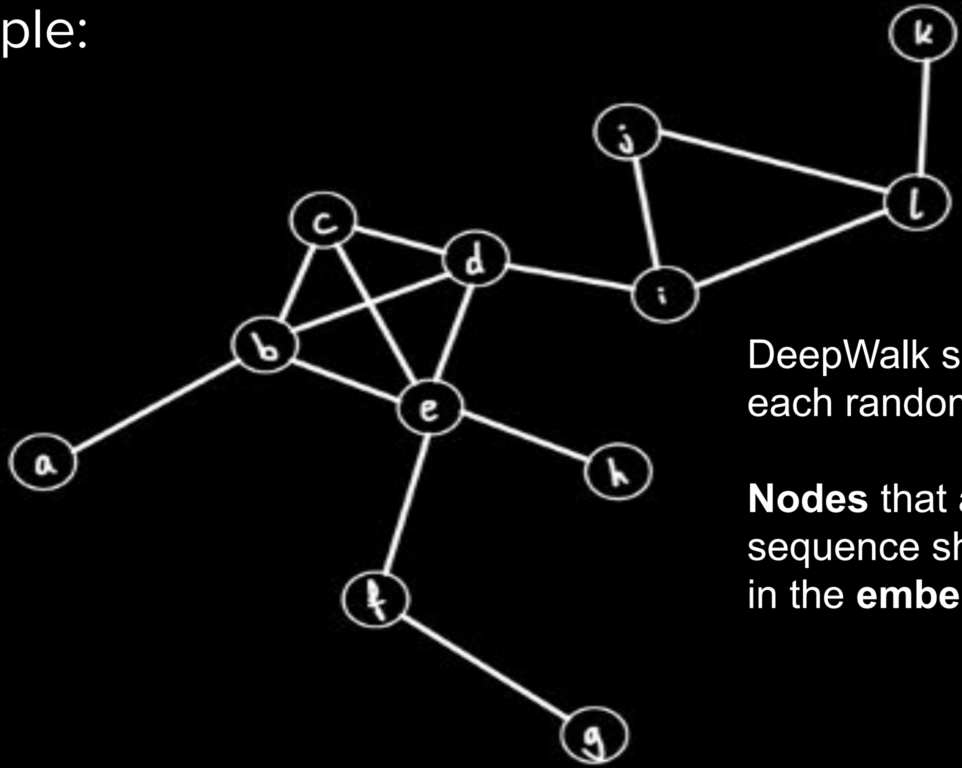
(words close in sentences → close in embedding space)

DeepWalk: word2vec For Graphs

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



Example:



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

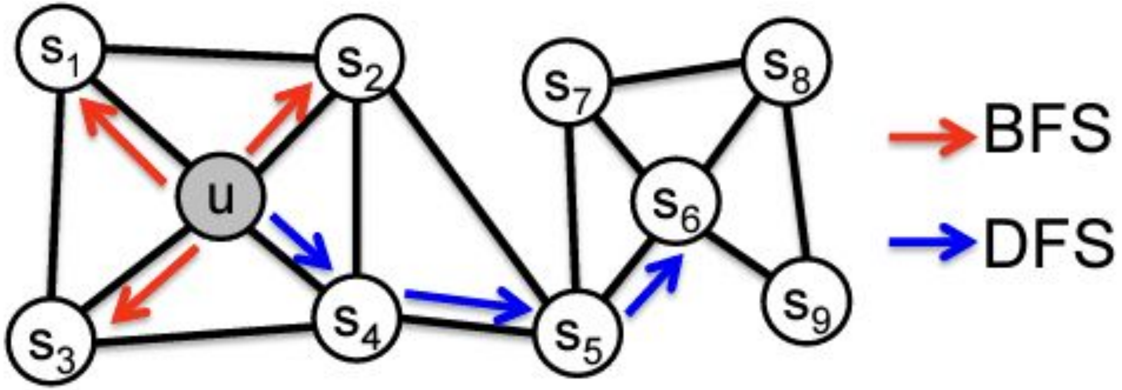
Nodes that are close together in the random walk sequence should be **embedded closer together** in the **embedding space!**

These are the **“sentences” that we generate!**

Random walk :

node2vec: The Introduction of Bias...

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

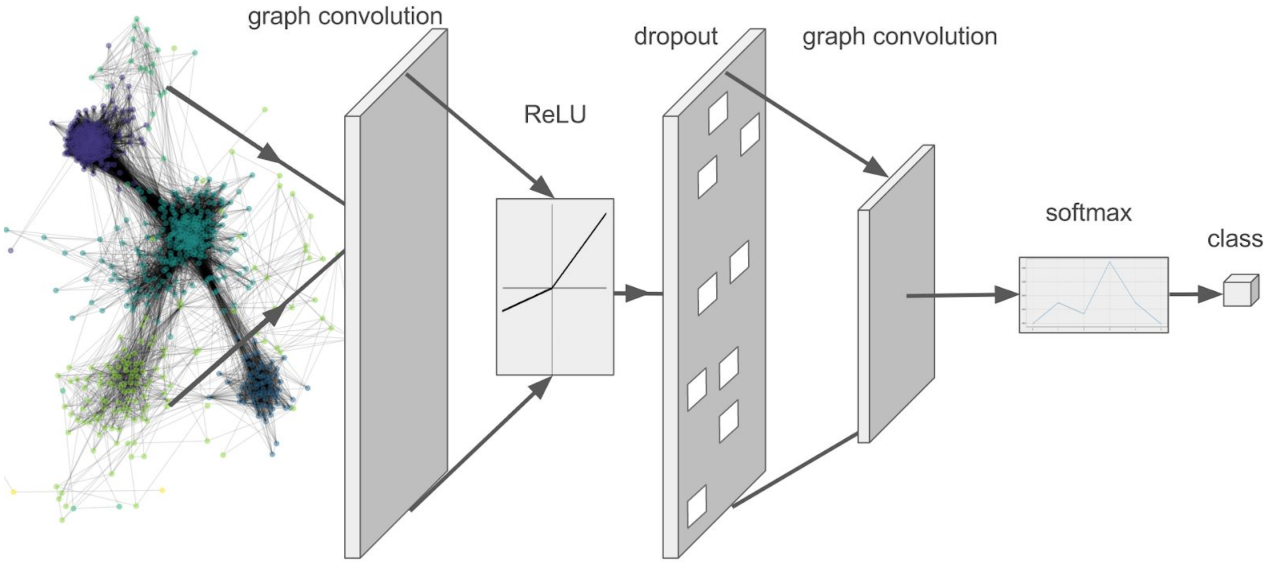


Breadth First Search (BFS) {s1, s2, s3} **Local** microscopic view — Homophily

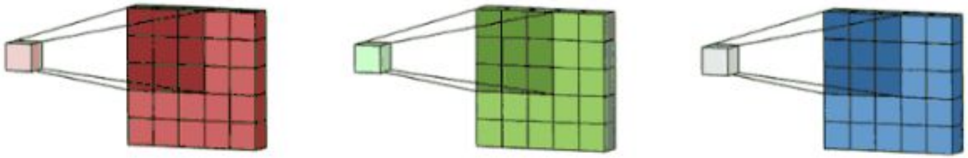
Depth First Search (DFS) {s4, s5, s6} **Global** macroscopic view — Structural equivalence

Grover and Leskovec., ACM SIGKDD, 2016

GRAPH NEURAL NETWORKS

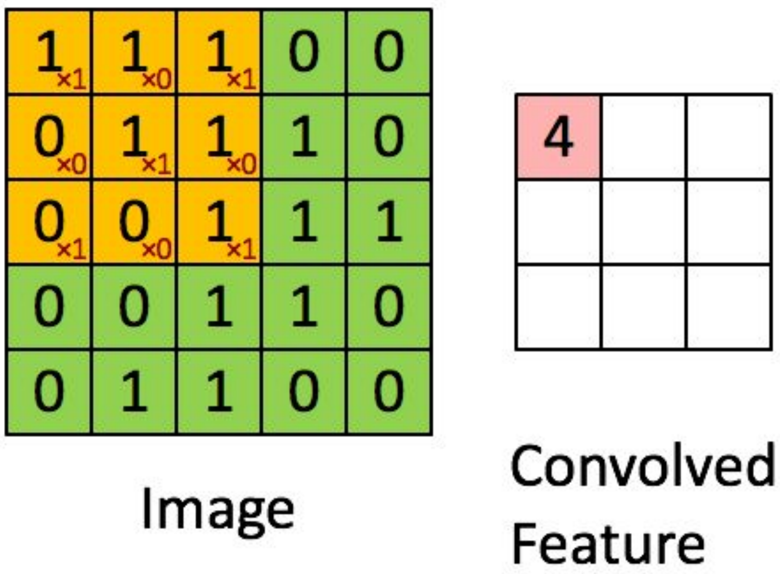


Brief Recap: Convolutional Neural Networks



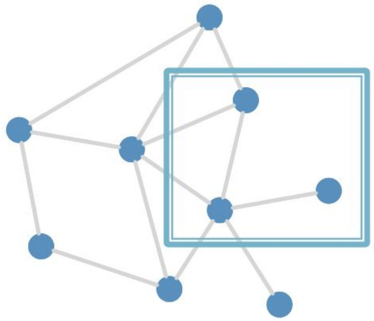
Convolutional Layer in CNN

Translation-invariant



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

Convolutions on Graphs?



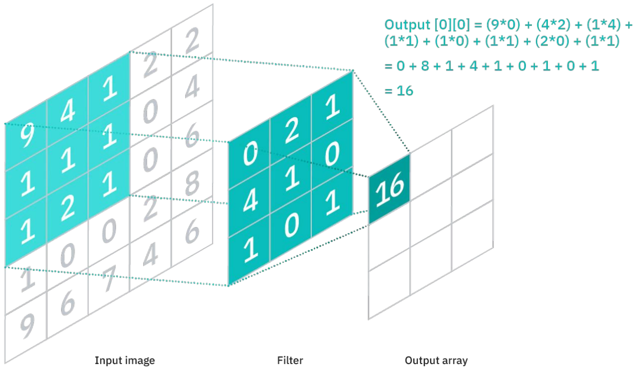
- No fixed notion of a sliding window on a graph
  - Variable number of neighbors per node
  - Every single pixel (not in a corner) in an image is surrounded by 8 neighboring pixels

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

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

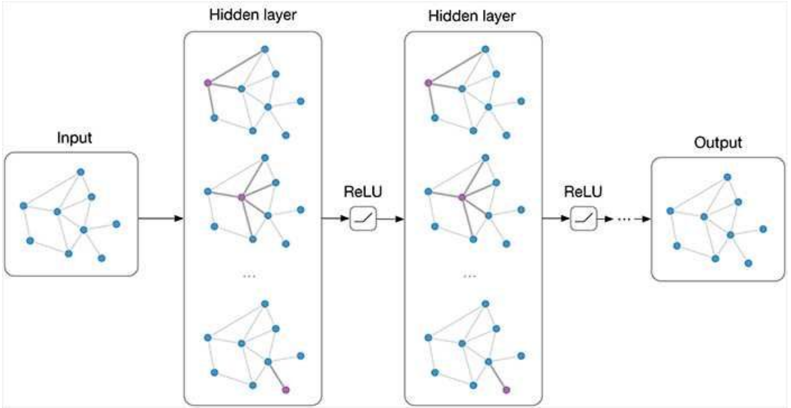
Locality vs Homophily

Image Convolutions



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



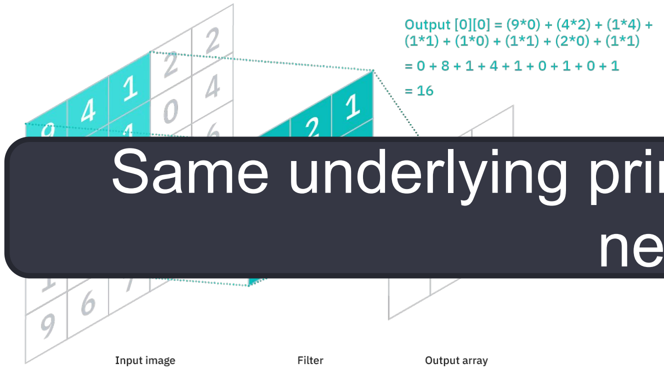
Generate next layer embedding vectors for each **node** in an input graph by **aggregating** the **transformed feature vectors** of each of the node's neighbors

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

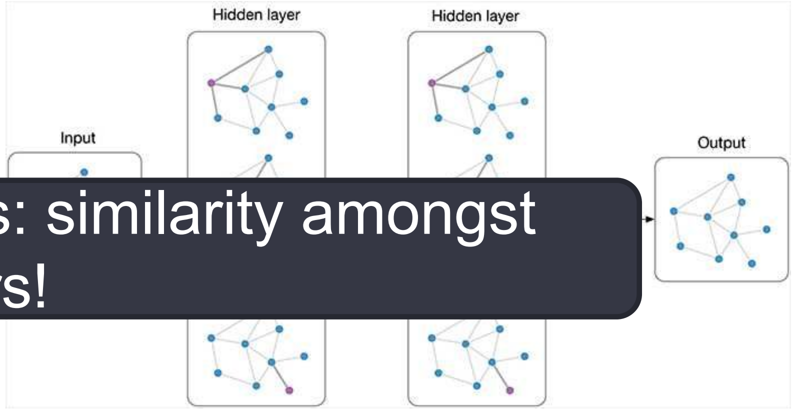
Locality vs Homophily

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

Image Convolutions



Graph Convolutions



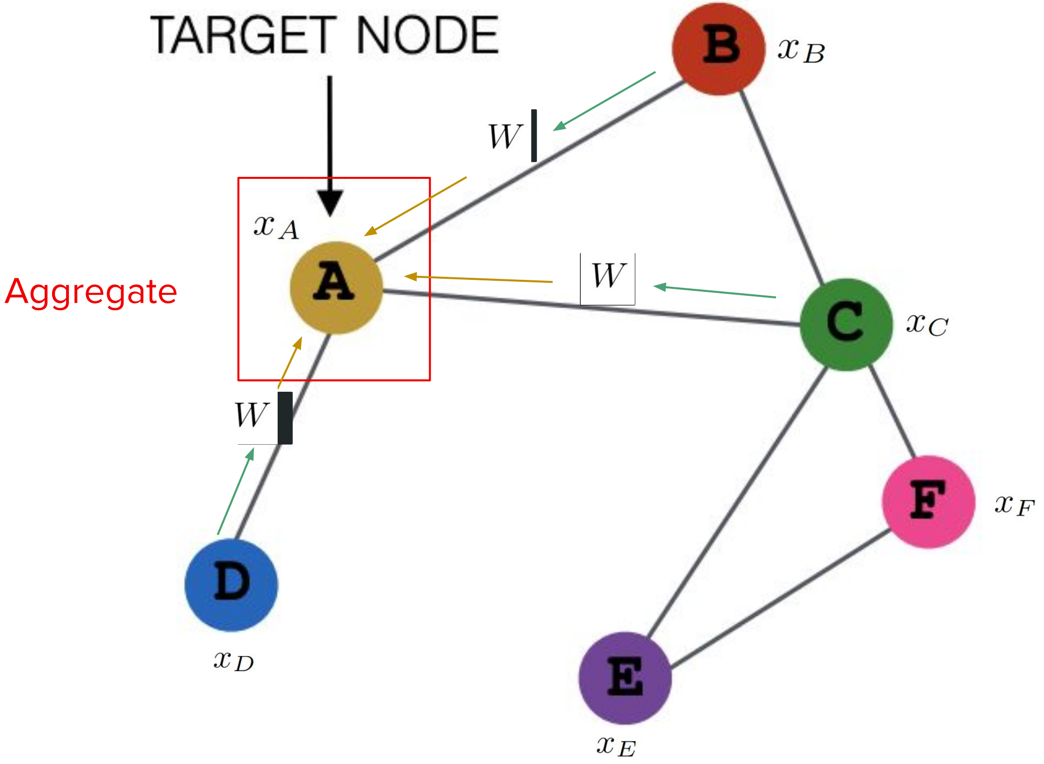
Same underlying principles: similarity amongst neighbors!

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

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



INPUT GRAPH

Transform

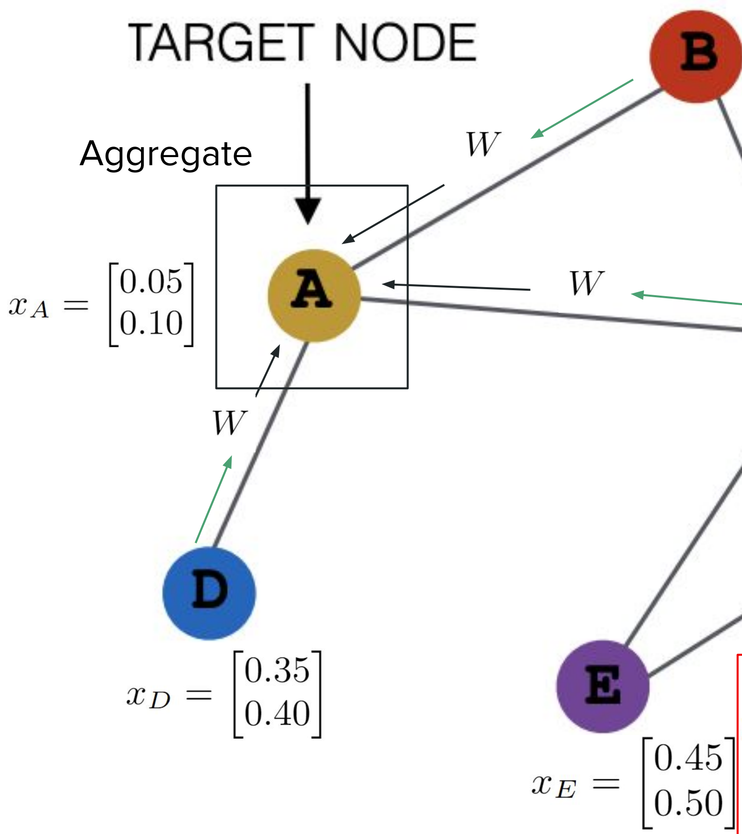
$$h_A = \sigma\left(\sum_{u \in N(A)} \frac{x_u W}{|N(A)|}\right)$$

Aggregate

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

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

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



INPUT GRAPH

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

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

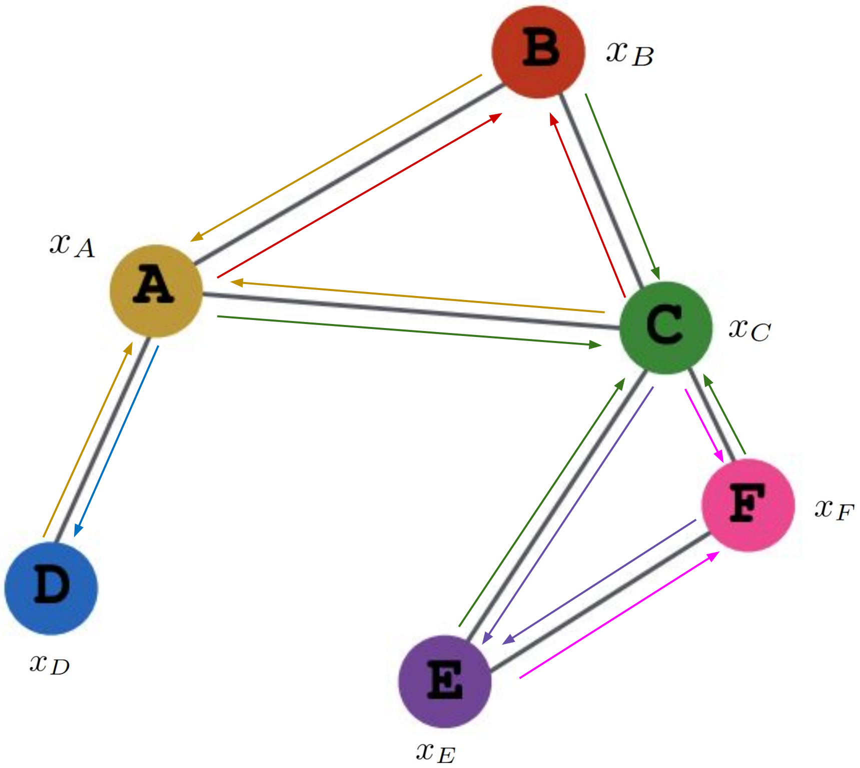
$$h_A = \sigma\left(\frac{\begin{bmatrix}0.15 & 20\end{bmatrix}\begin{bmatrix}0.1 & 0.3 & 0.5\\ 0.2 & 0.4 & 0.6\end{bmatrix}}{3} + \frac{\begin{bmatrix}0.25 & 30\end{bmatrix}\begin{bmatrix}0.1 & 0.3 & 0.5\\ 0.2 & 0.4 & 0.6\end{bmatrix}}{3} + \frac{\begin{bmatrix}0.35 & 40\end{bmatrix}\begin{bmatrix}0.1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

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

e.g ReLU

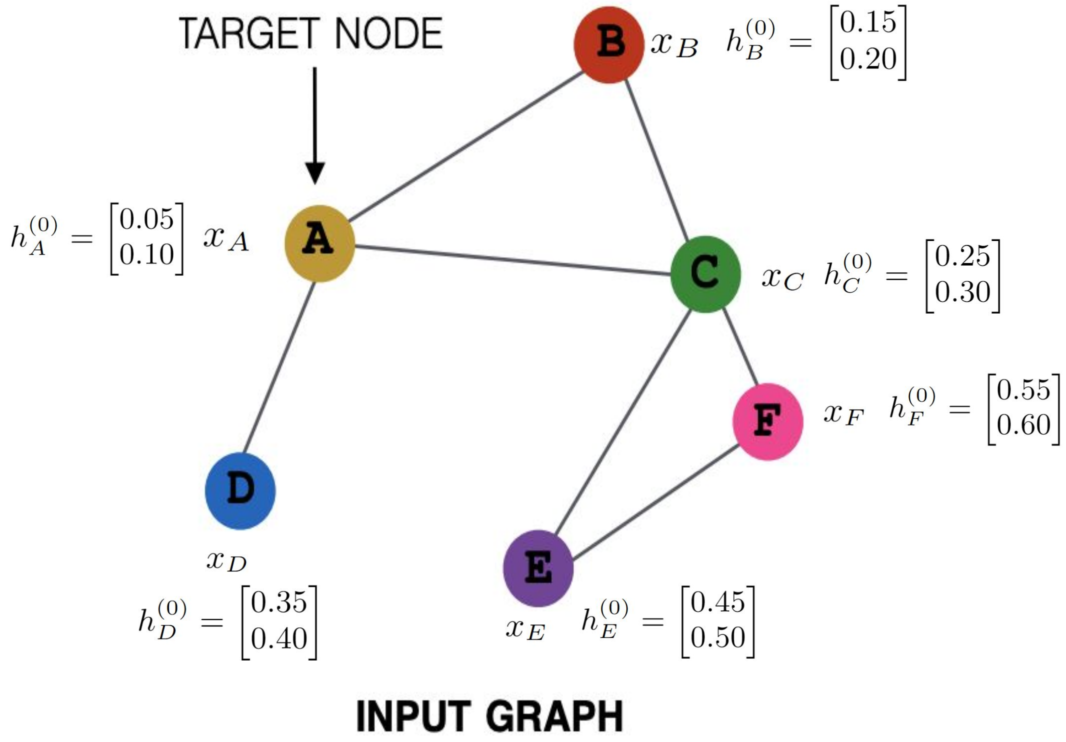
$$= \begin{bmatrix}0.145 & 0.2 & 0.255\end{bmatrix}$$

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



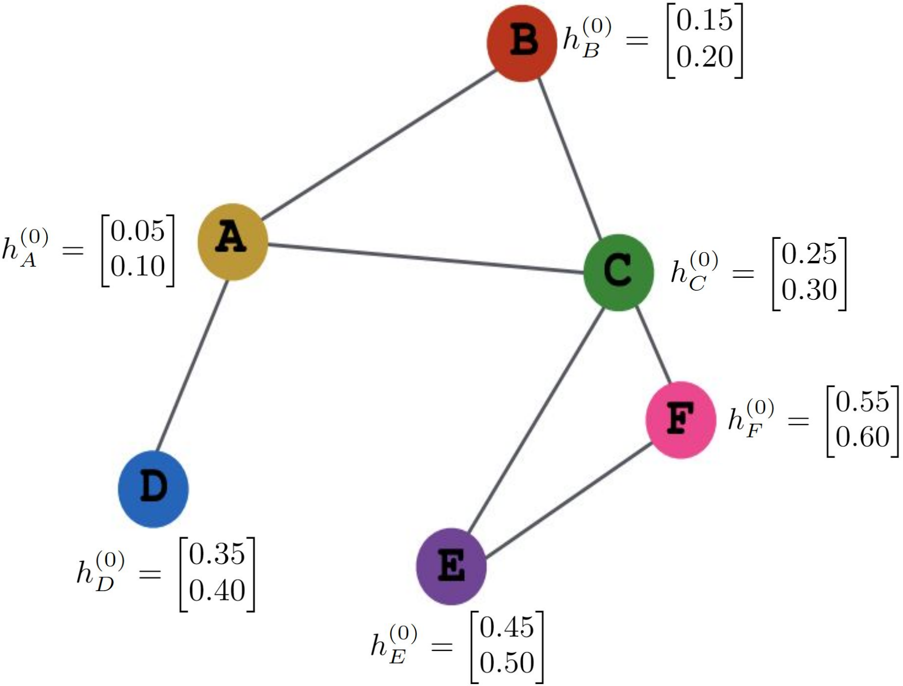
INPUT GRAPH

Example time!



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

Let's also give these some values...

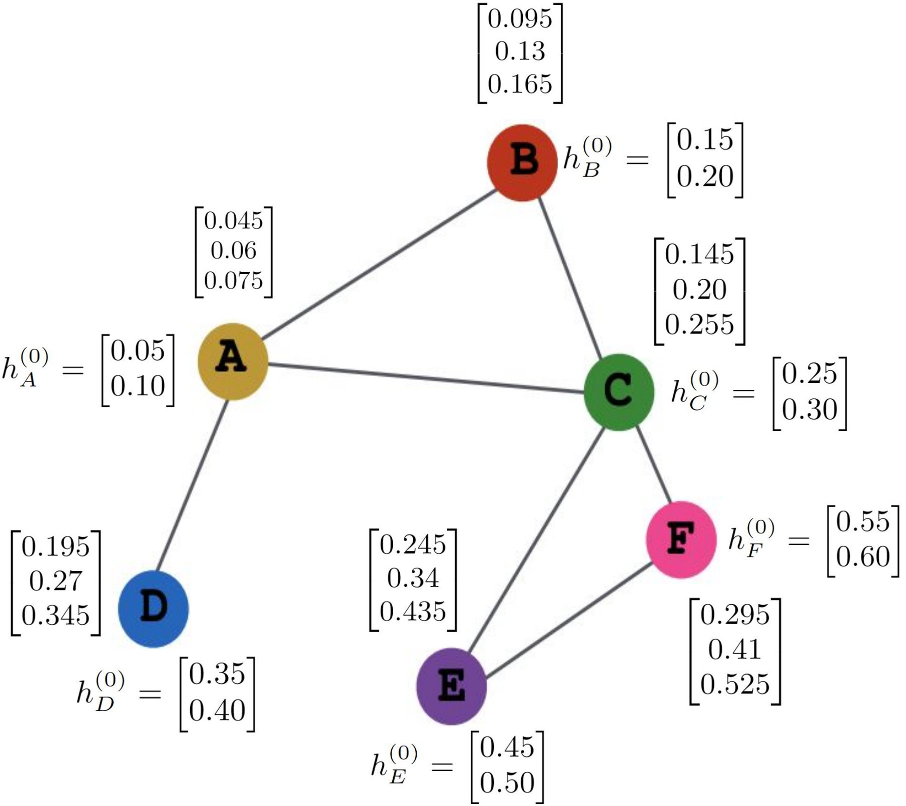


INPUT GRAPH

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

2.

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



INPUT GRAPH

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

2.

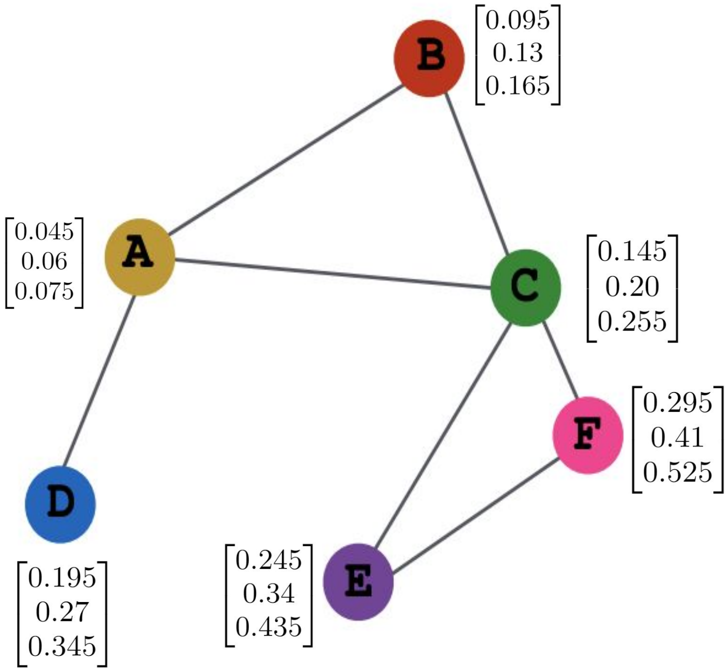
$$H^{(0)} = \begin{bmatrix} h_A^{(0)} & h_B^{(0)} & h_C^{(0)} & h_D^{(0)} & h_E^{(0)} & h_F^{(0)} \end{bmatrix}^T = \begin{bmatrix} 0.05 & 0.10 \\ 0.15 & 0.20 \\ 0.25 & 0.30 \\ 0.35 & 0.40 \\ 0.45 & 0.50 \\ 0.55 & 0.60 \end{bmatrix}$$

3. **Transform**!

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

Every node is transformed by the same weight matrix!!!

We also call this **message passing**.



INPUT GRAPH

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

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

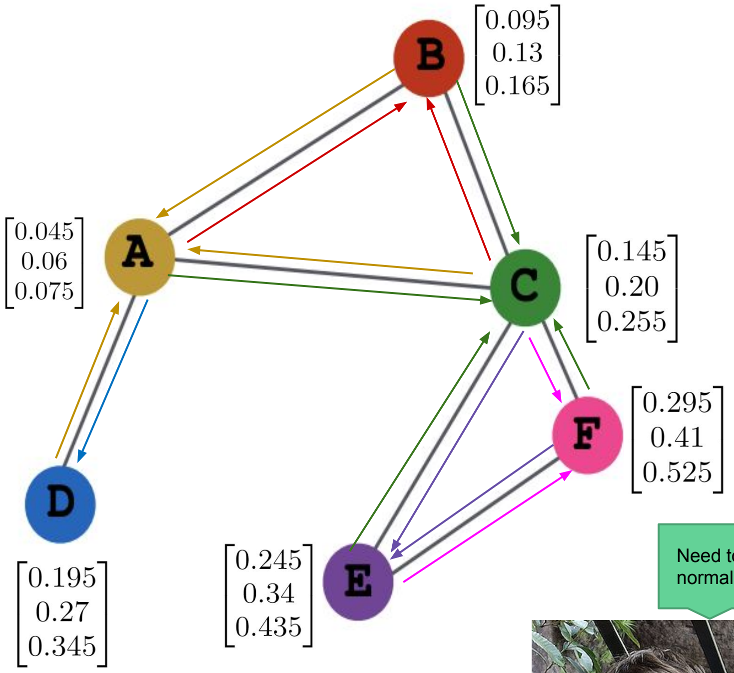
3. **Transform**!

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

4. Define adjacency matrix:

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

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



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!**

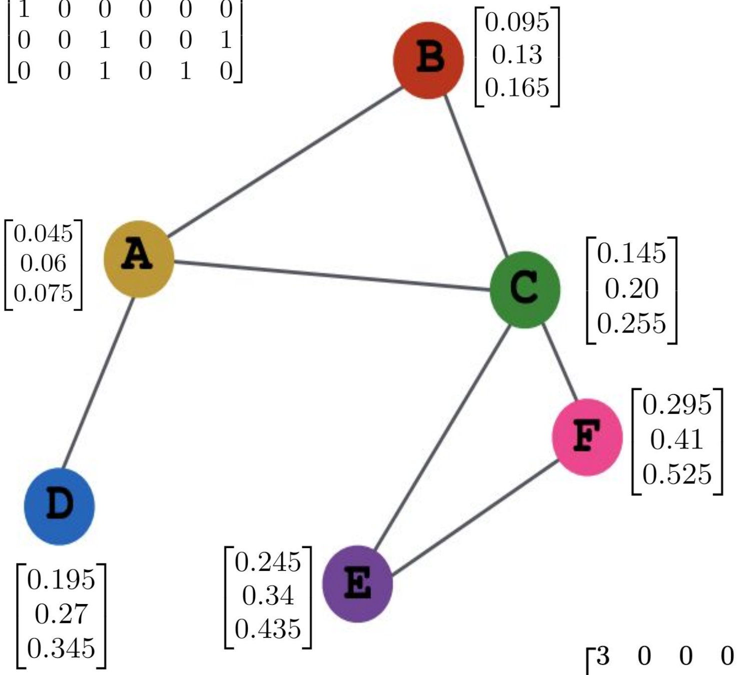
$$W_0 = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.4 & 0.6 \end{bmatrix} \qquad H^{(0)}W_0 = \begin{bmatrix} 0.045 & 0.06 & 0.075 \\ 0.095 & 0.13 & 0.165 \\ 0.145 & 0.20 & 0.255 \\ 0.195 & 0.27 & 0.345 \\ 0.245 & 0.34 & 0.435 \\ 0.295 & 0.41 & 0.525 \end{bmatrix}$$

4. Define adjacency matrix:

5. **Aggregate!**

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

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



INPUT GRAPH

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

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

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:

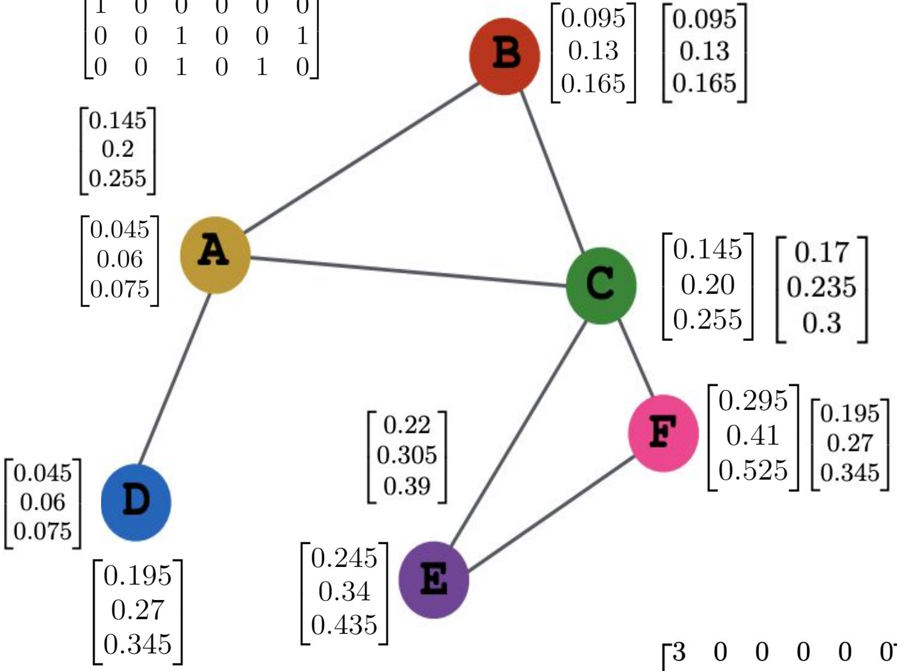
5. **Aggregate!**

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

6. Normalize

$$D^{-1} = \begin{bmatrix} \frac{1}{3} & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} \end{bmatrix} \quad D^{-1}AH^{(0)}W_0 = \begin{bmatrix} 0.145 & 0.2 & 0.255 \\ 0.095 & 0.13 & 0.165 \\ 0.17 & 0.235 & 0.3 \\ 0.045 & 0.06 & 0.075 \\ 0.22 & 0.305 & 0.39 \\ 0.195 & 0.27 & 0.345 \end{bmatrix}$$

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



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!**

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

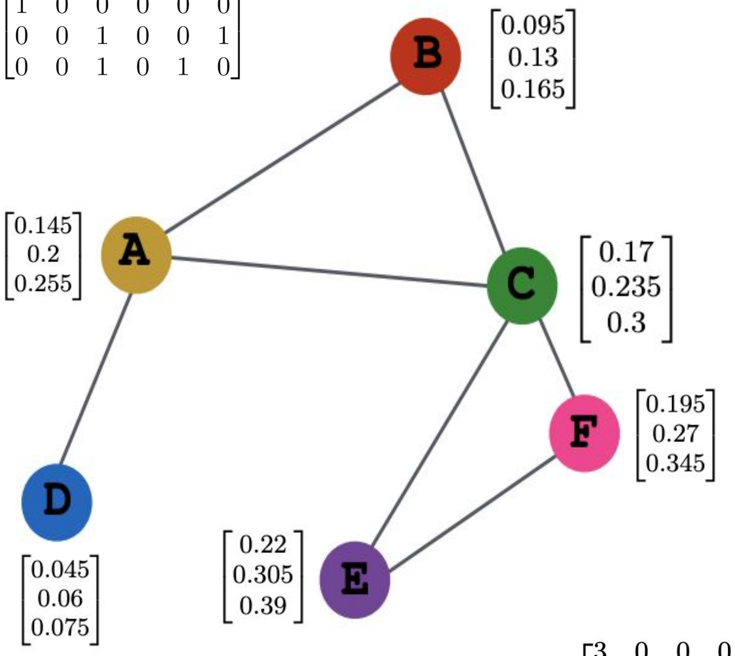
6. Normalize

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

7. Pass through **non-linearity**

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

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



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**!

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

2. Aggregate

1. Transform

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

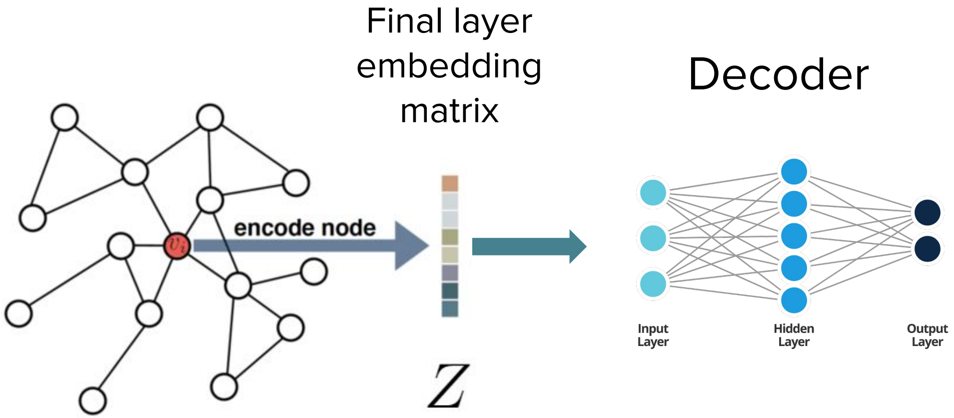
With random walks, what we're optimizing are the **final embedding vectors** themselves, **not** weights...

...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!

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

What do we do with Z?



Depends on the downstream prediction task:

- Feed *Z* into a **MLP + Softmax decoder** for **node-level classification/regression**
- Apply some decoder function on **pairs of vectors** in *Z* for **link prediction** (e.g **dot product**)
- 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 decode**
    - Just like in CNNs!

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!

Stacking GCN Layers

Final GCN update rules:

Node-level update rule:

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

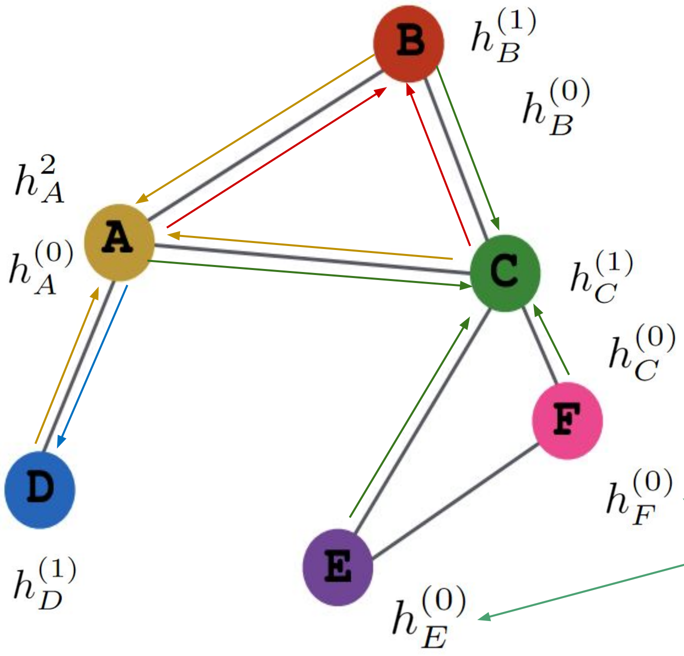
Graph-level update rule:

$$H^{(l+1)} = \sigma(D^{-1} A H^{(l)} W_l)$$

Let's just keep adding more layers, right?

BIG problem!

Stacking GCN Layers



INPUT GRAPH

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

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

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

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

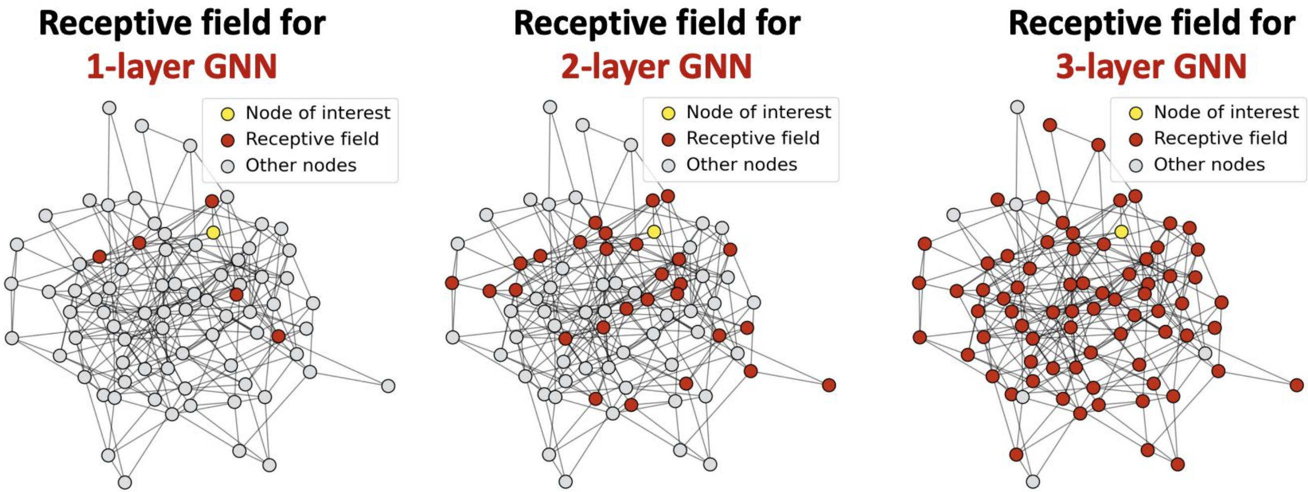
In order to calculate each node u's h_u^1 vector, we need to calculate $h_{u'}^0$ for each u' in Neighbors(u)

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: 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.

Let’s look at some methods that build on GCN

GraphSAGE

2 **BIG problems** with GCNs:

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

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

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

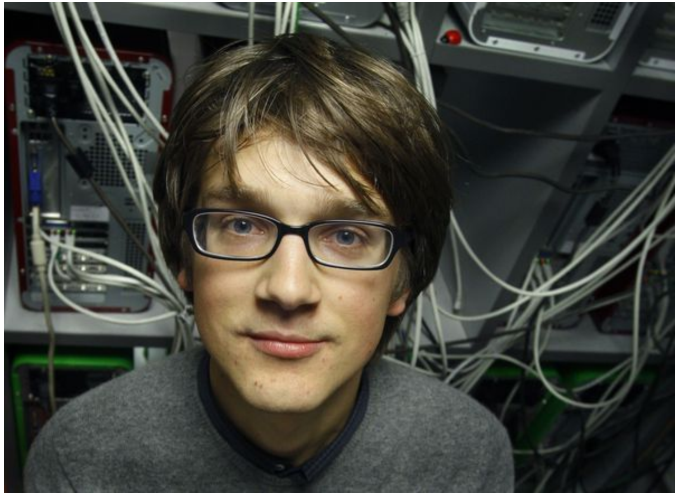
$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

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

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

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**

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?

Get rid of the non-linearities!

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

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

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

Why does this work so well?

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

Kilian Weinberger
PhD @ UPENN
Currently: Professor @ Cornell
Also currently: **Listening to students teach a CS6784 lecture**

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	$80.2 \pm 3.0^{\dagger}$	67.3 ± 0.5	$78.3 \pm 0.6^{\dagger}$
AdaLNet	$81.9 \pm 1.9^{\dagger}$	$70.6 \pm 0.8^{\dagger}$	$77.8 \pm 0.7^{\dagger}$
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

$$H^{(l+1)} = S...SSH^{(0)}W_0W_1...W_l$$
$$= S^lH^{(0)}W$$

1. Turns out GCN doesn't scale well to very large graphs due to excessive memory requirements. SGC precomputes $S^K H^{(0)}$ and only learns a single weight matrix
2. Less parameters = less overfitting = faster!

Summary

- Graphs: Combination of nodes and edges
- Learning on graphs: Classify nodes and entire graphs, predict links or detect communities and even generate graphs and their embeddings
- Feature Engineering 😡 Representation Learning 😊
- Graph Encoder: Map nodes to low-dimensional embedding vectors
- Graph Decoder: Map embedding vectors to Y
- 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
- GraphSAGE: Self-loops + treat aggregation function as a hyperparameter
- SGC: No need for non-linearities, we can still get good results much faster by collapsing weights!

Next time - Transformers on graphs!

Graphormer

The End: Just the start for GNNs