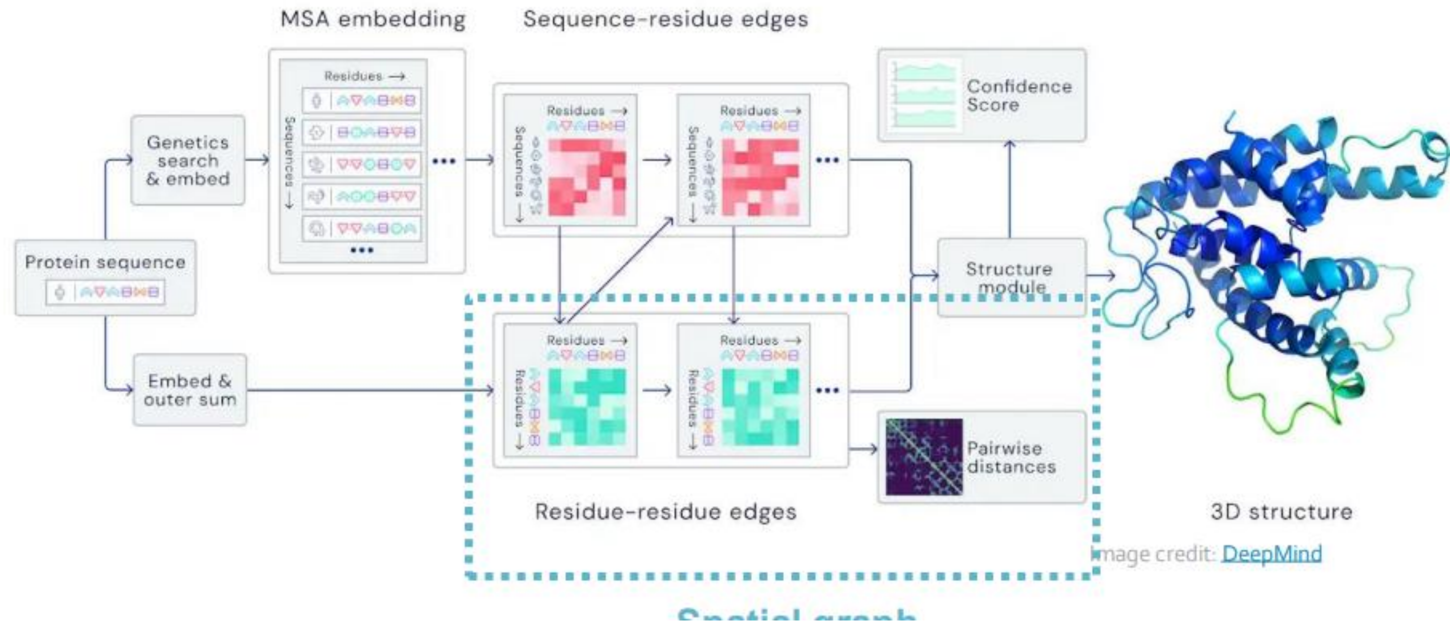
Graph Neural Networks



Slides from Jure Leskovec

ECE324, Winter 2022

Michael Guerzhoy

Graph tasks

- Node classification: predict a property of a node
  - Categorize online users/items
- Link prediction: predict whether there is a missing edge between two nodes
  - Recommend Facebook friends
- Graph classification: categorize graphs
  - Predict properties of molecules represented as graphs
- Clustering: detect if nodes form a community
  - Find social circles in a social network

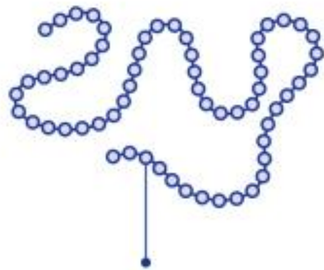
Node-level task: protein folding

Every protein is made up of a sequence of amino acids bonded together

These amino acids interact locally to form shapes like helices and sheets

These shapes fold up on larger scales to form the full three-dimensional protein structure

Proteins can interact with other proteins, performing functions such as signalling and transcribing DNA



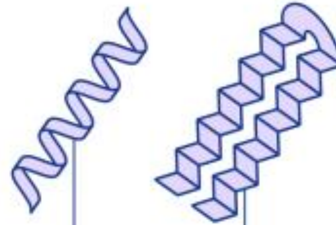


Image credit: DeepMind

Node-level task: protein folding

Predict a protein's 3D structure based on its amino acid sequence

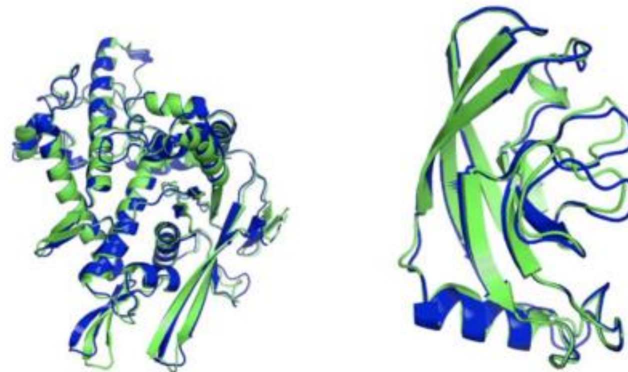
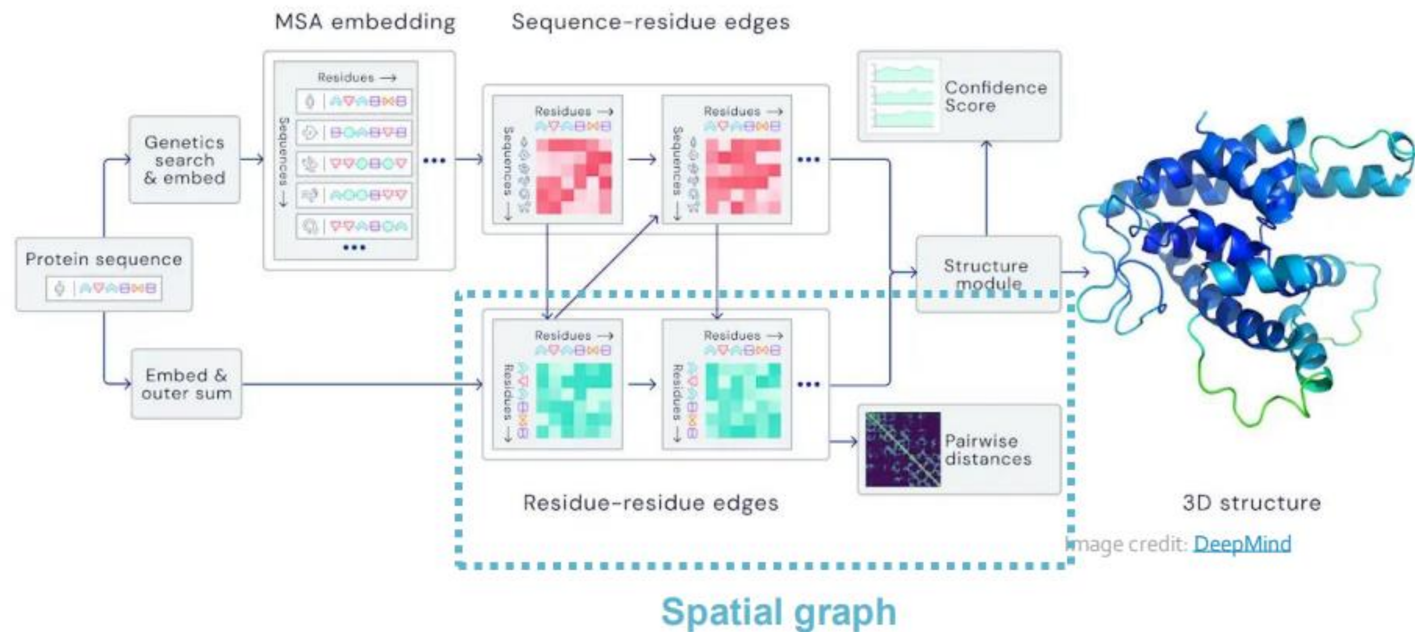


Image credit: DeepMind

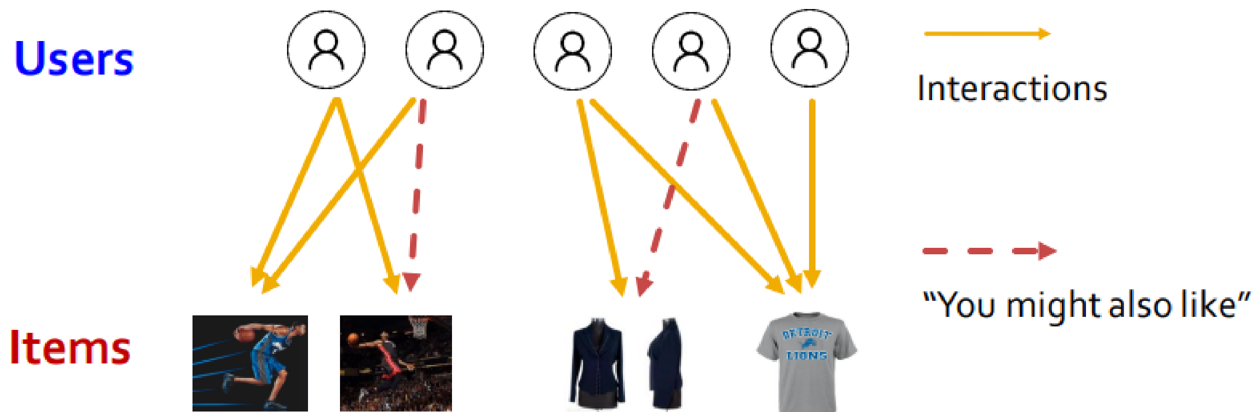
AlphaFold: idea

- **Key idea:** "Spatial graph"
  - **Nodes:** Amino acids in a protein sequence
  - **Edges:** Proximity between amino acids (residues)



Edge-level task: recommender systems

- Users interact with items
  - Watch movies, buy merchandise, listen to music
  - Nodes: users and items
  - Edges: user-item interactions
    - E.g., watching a movie, buying an item
- Goal: recommend items to users



Subgraph-level task: traffic prediction

- Nodes: road segments
- Edges: connectivity between road segments
- Prediction: time of arrival

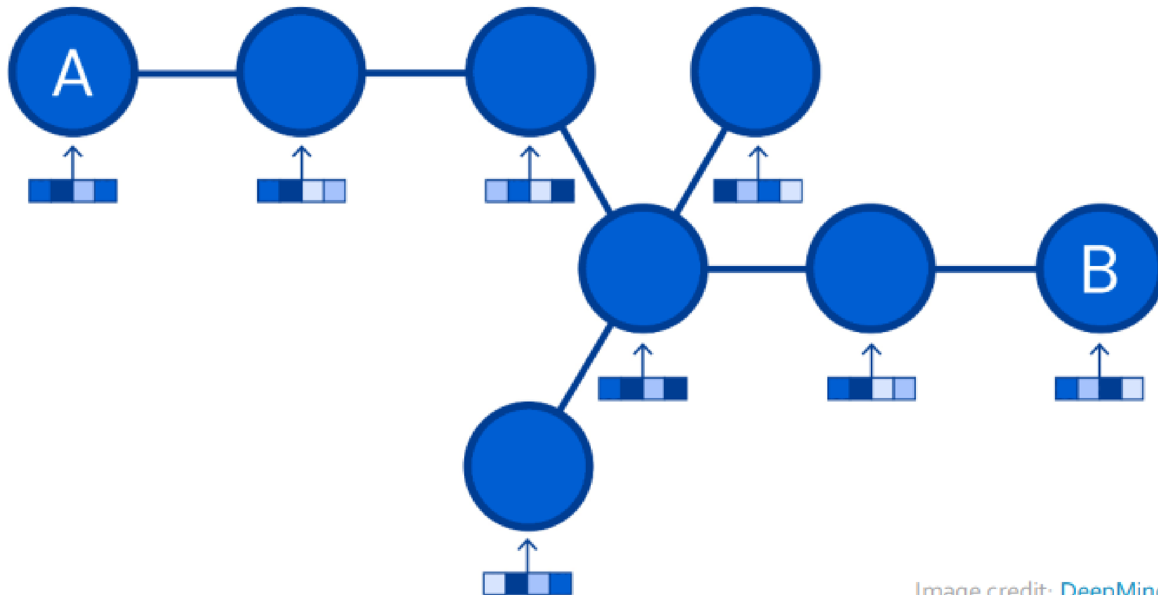
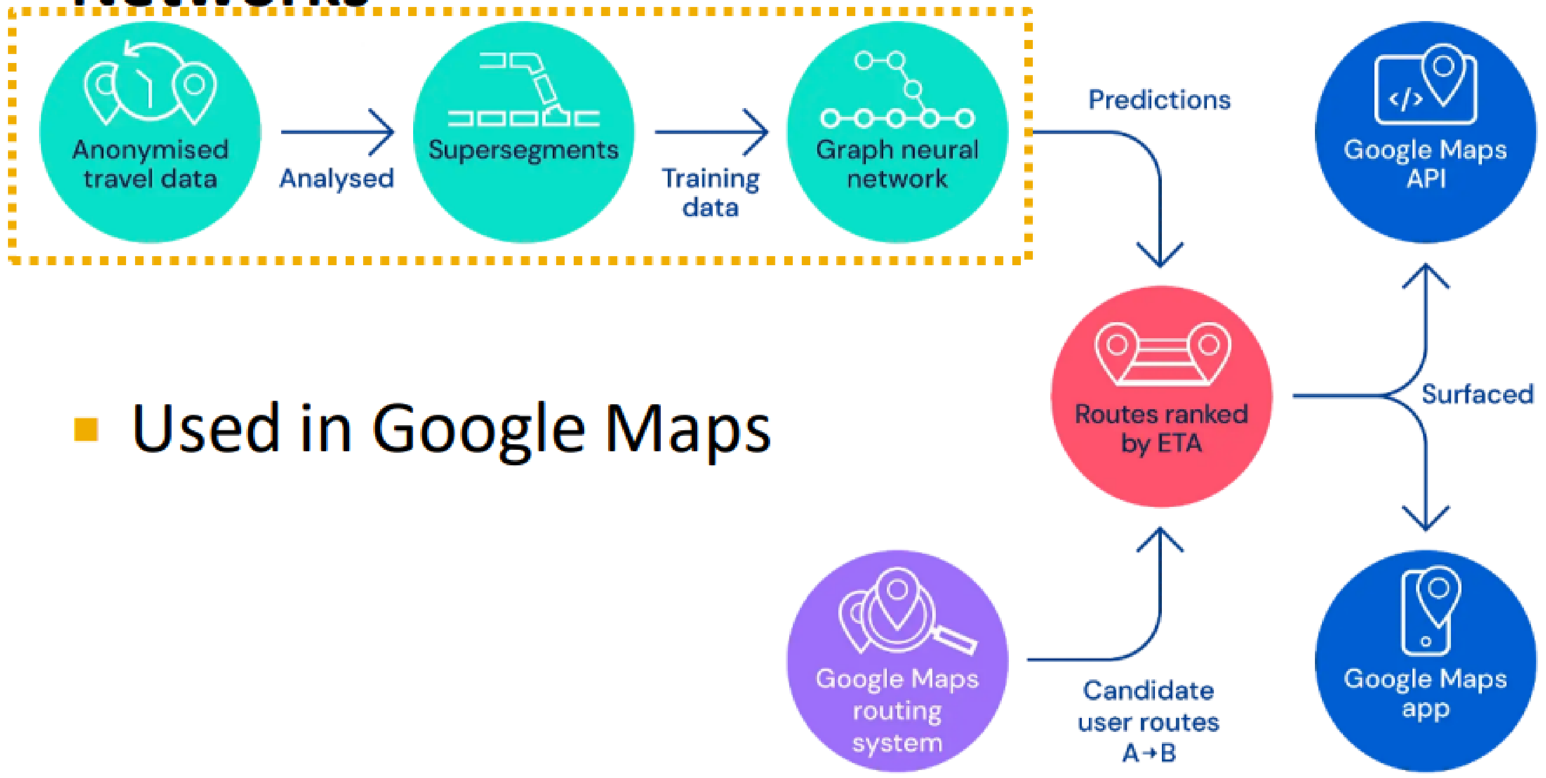


Image credit: DeepMind

Predicting Time of Arrival with Graph Neural Networks



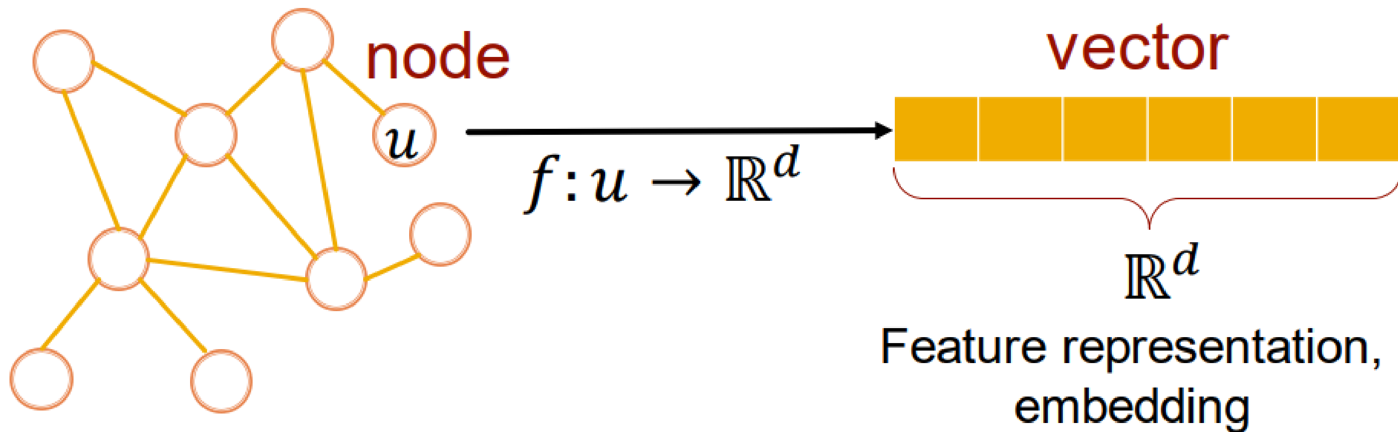
- Used in Google Maps

THE MODEL ARCHITECTURE FOR DETERMINING OPTIMAL ROUTES AND THEIR TRAVEL TIME.

Image credit: DeepMind

Node embedding

- Goal: learn a d-dimensional embeddings for graph nodes
  - Can then compute node similarity
  - Can learn to predict links from pairs of d-dimensional embeddings



Node similarity

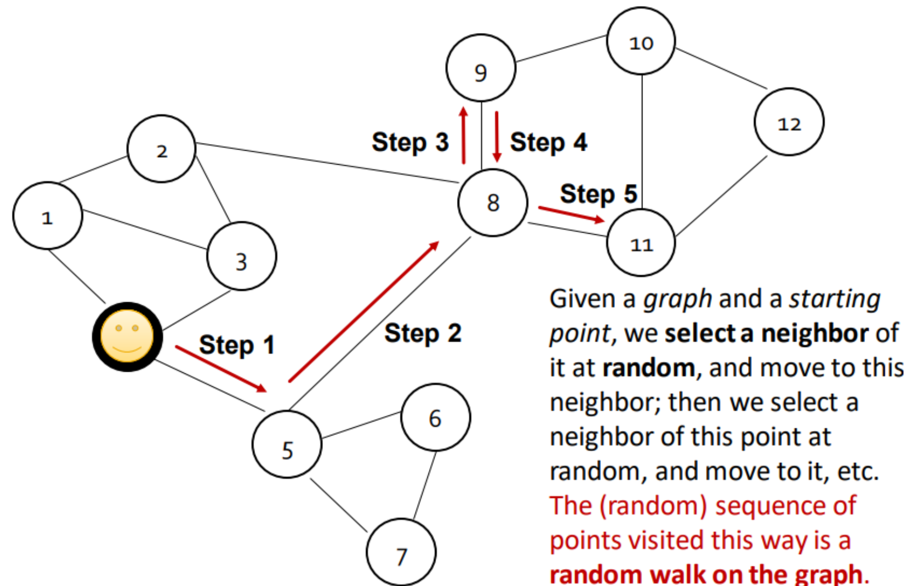
- Two nodes are similar if they
  - Are linked or
  - Share neighbours or
  - Are connected to the same kind of nodes (have similar “structural roles”)
- Similar to the idea of words being similar if they appear in the same context as the same kind of words
  - With an infinite corpus, we could just look at the co-occurrence matrix, but with limited data it’s better to learn GLoVe/word2vec embeddings

Designing node embeddings

- Decide what we mean by “similar nodes”
  - Are linked
  - Share neighbours
  - Appear in the same neighbourhoods
- Decide what we mean by “similar embeddings”
  - $z_v^T z_u$ high

Random Walk embeddings

- Two nodes are similar if they appear together in random walks on the graph



- Want $z_u^T z_v$ to be high if (u, v) co-occur on a random walk with high probability

Why Random Walks?

- Expressivity: the definition captures the idea of nodes being similar if they are linked to similar kinds of nodes
- Efficiency: don't need to account for pairs of nodes that don't co-occur
  - Frequently a large majority

Learning Embeddings

- Given G = (V, E)
- Goal is to learn a mapping $u \to z_u$
- Log-likelihood objective:
  $\max_{z} \sum_{u \in V} \log P(N_R(u) | z_u)$
- Given a node u, we want to learn feature representations that are predictive of the nodes in its random walk neighbourhood $N_R(u)$

Random Walk Optimization

- Run short fixed-length random walks starting from each node u in the graph using some random walk strategy R
- For each node u collect $N_R(u)$, the multiset of nodes visited on random walks starting from u.
- Optimize embeddings according to: given node u, predict its neighbours $N_R(u)$

$$\max_{z} \sum_{u \in U} \log P(N_R(u) | z_u)$$

- L = $-\sum_{u \in U} \log P(N_R(u)|z_u) = \sum_{u \in V} \sum_{v \in N_R(u)} -\log(\frac{\exp(z_u^T z_v)}{\sum_{n \in V} \exp(z_u^T z_n)})$

- Optimizing random walk embeddings

Finding embeddings z_u that minimize L

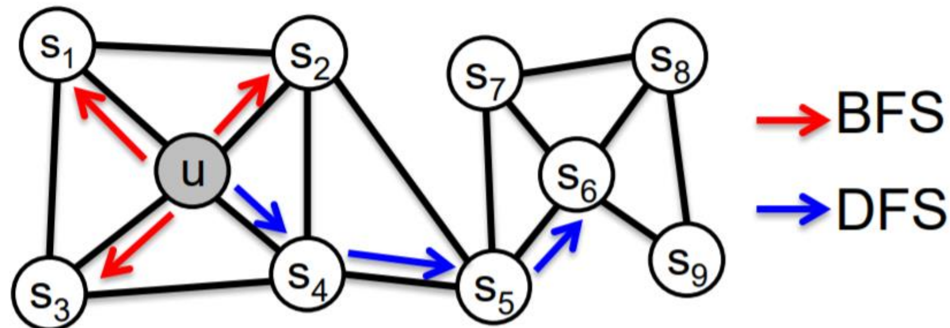
Negative sampling

- Instead of optimizing $\log\left(\frac{\exp(z_u^T z_v)}{\sum_{n\in V}\exp(z_u^T z_n)}\right)$, optimize

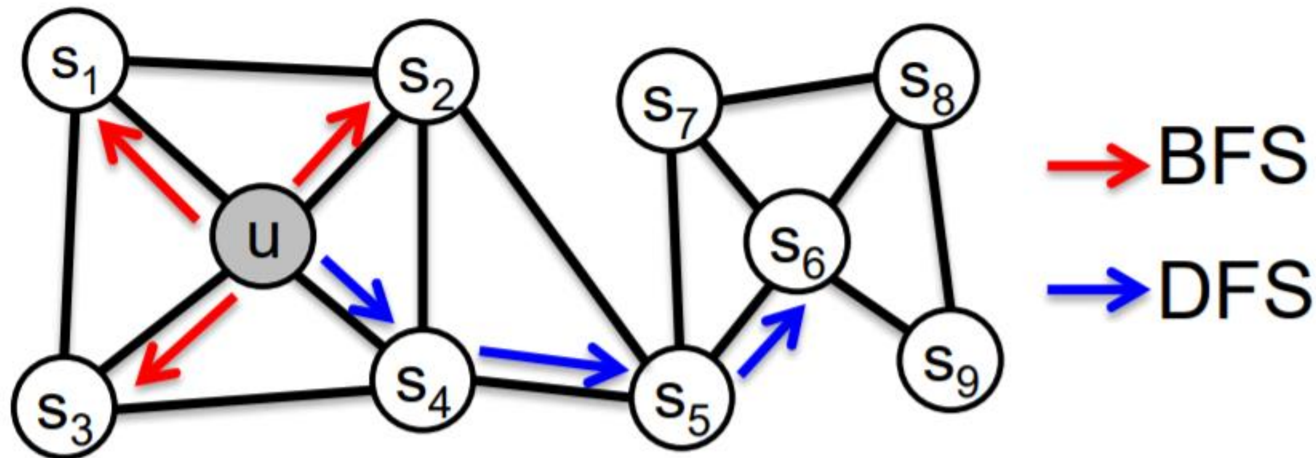
$$\log(\sigma(z_u^T z_v)) - \sum_{i=1}^{k} \log\left(\sigma\left(z_u^T z_{n_i}\right)\right), n_i \sim P_V$$

- Similar to what was done with word2vec
- Sample *k* negative nodes each with prob. proportional to its degree
- Two considerations for *k* (# negative samples):
  - Higher k gives more robust estimates
  - Higher k corresponds to higher bias toward negative events

- Learn L with stochastic gradient descent
- Random walks strategies
  - Fixed-length unbiased random walks starting from each node
  - Biased 2nd order random walk R to generate network neighborhood $N_R(u)$
    - Flexible biased random walks that can trade off between local and global views of the network



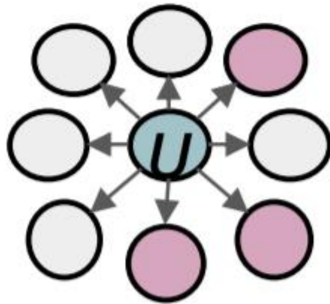
Two classic strategies to define a neighborhood $N_R(u)$ of a given node u:



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

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

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



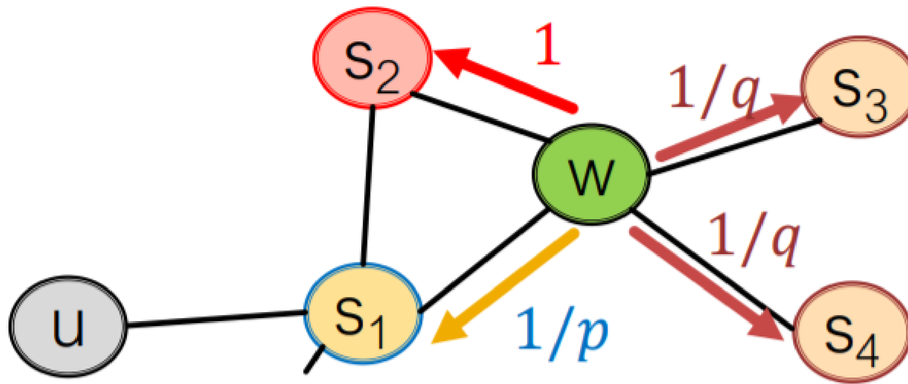
BFS:
Micro-view of neighbourhood

DFS:
Macro-view of neighbourhood

Interpolating BFS and DFS

- Biased fixed-length random walk *R* that given a node *u* generates neighbourhood $N_R(u)$
- Two parameters:
  - *Return* parameter p
    - Return back to the previous node
  - *In-Out* parameter q
    - Moving outwards (DFS) vs. inwards (BFS)
    - Intuitively, *q/p* is the ratio of BFS vs DFS

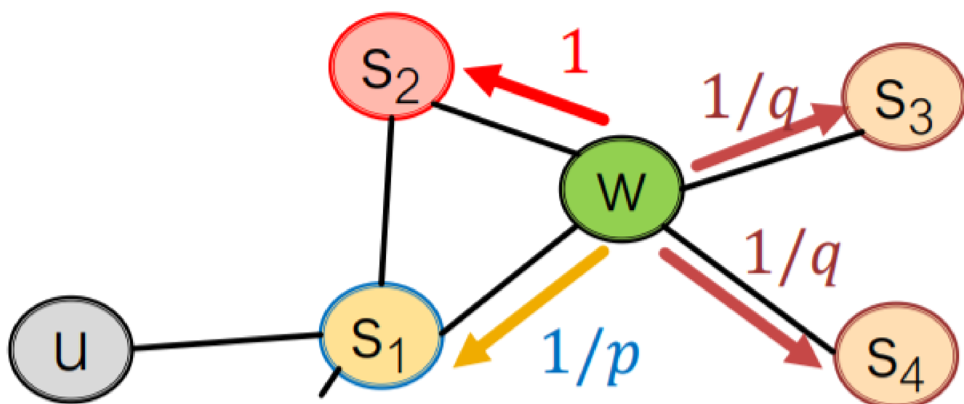
- **Walker came over edge (s_1, w) and is at w. Where to go next?**



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

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

- **Walker came over edge (s_1, w) and is at w. Where to go next?**



$w \rightarrow$

Target t	Prob.	Dist. (s_1, t)
s_1	$1/p$	0
s_2	1	1
s_3	$1/q$	2
s_4	$1/q$	2

Unnormalized transition prob. segmented based on distance from s_1

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

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

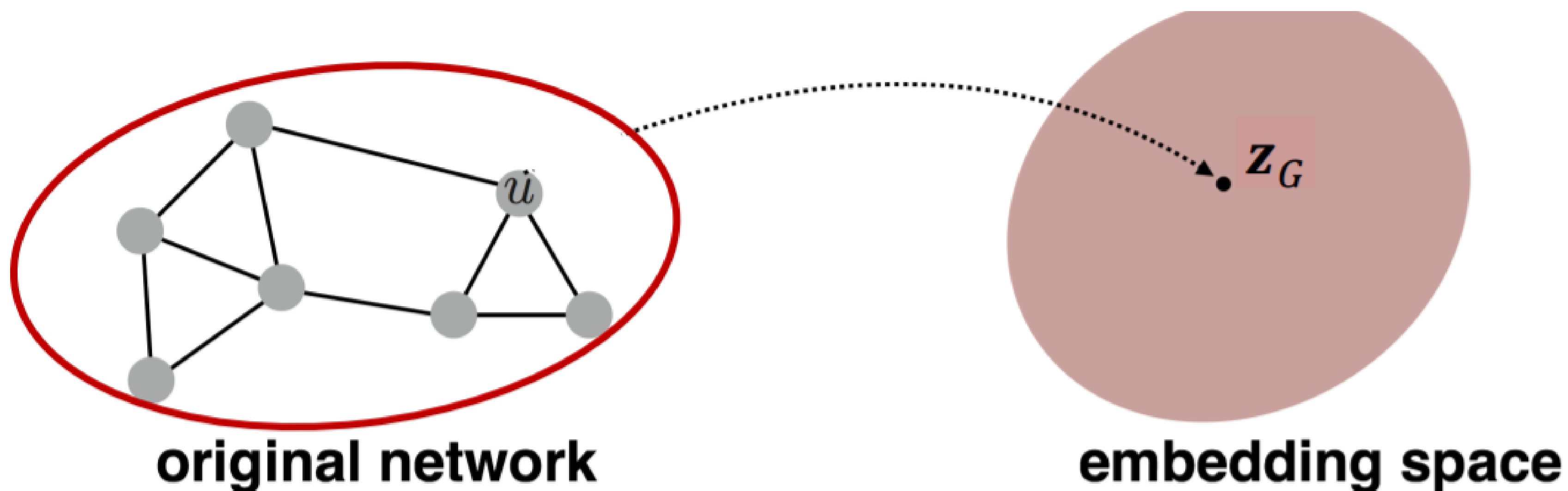
node2vec algorithm

- Compute random walk probabilities
- Simulate *r* random walks of length *l* starting from each note *u*
- Optimize the node2vec objective using SGD

- Linear-time complexity
- All 3 steps are individually parallelizable

Embedding entire graphs

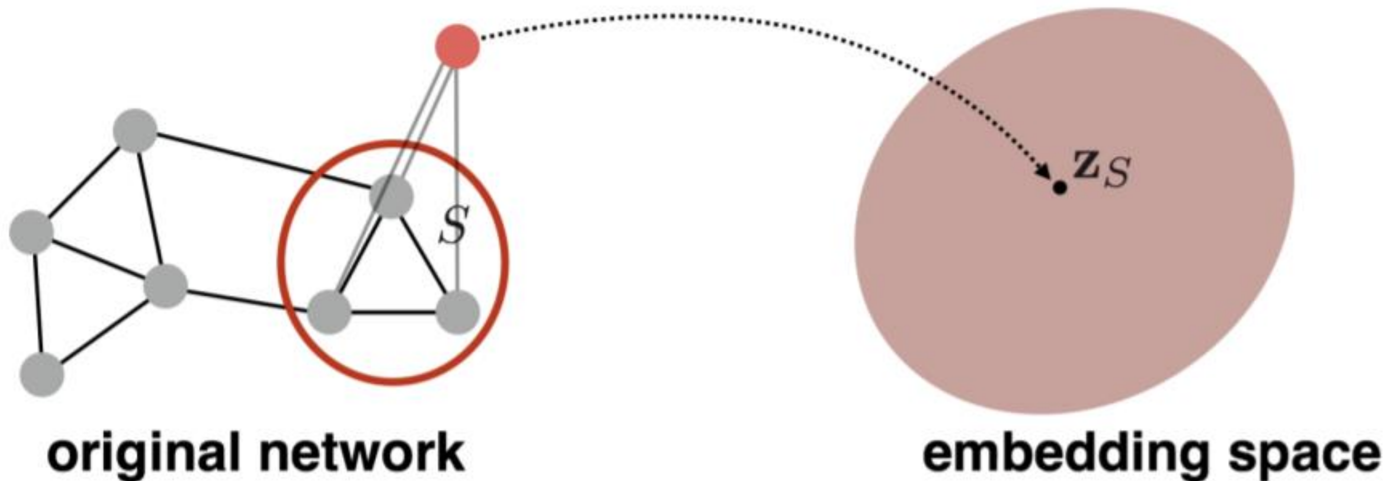
- Goal: want to embed a subgraph or an entire graph *G*. Graph embedding: z_G



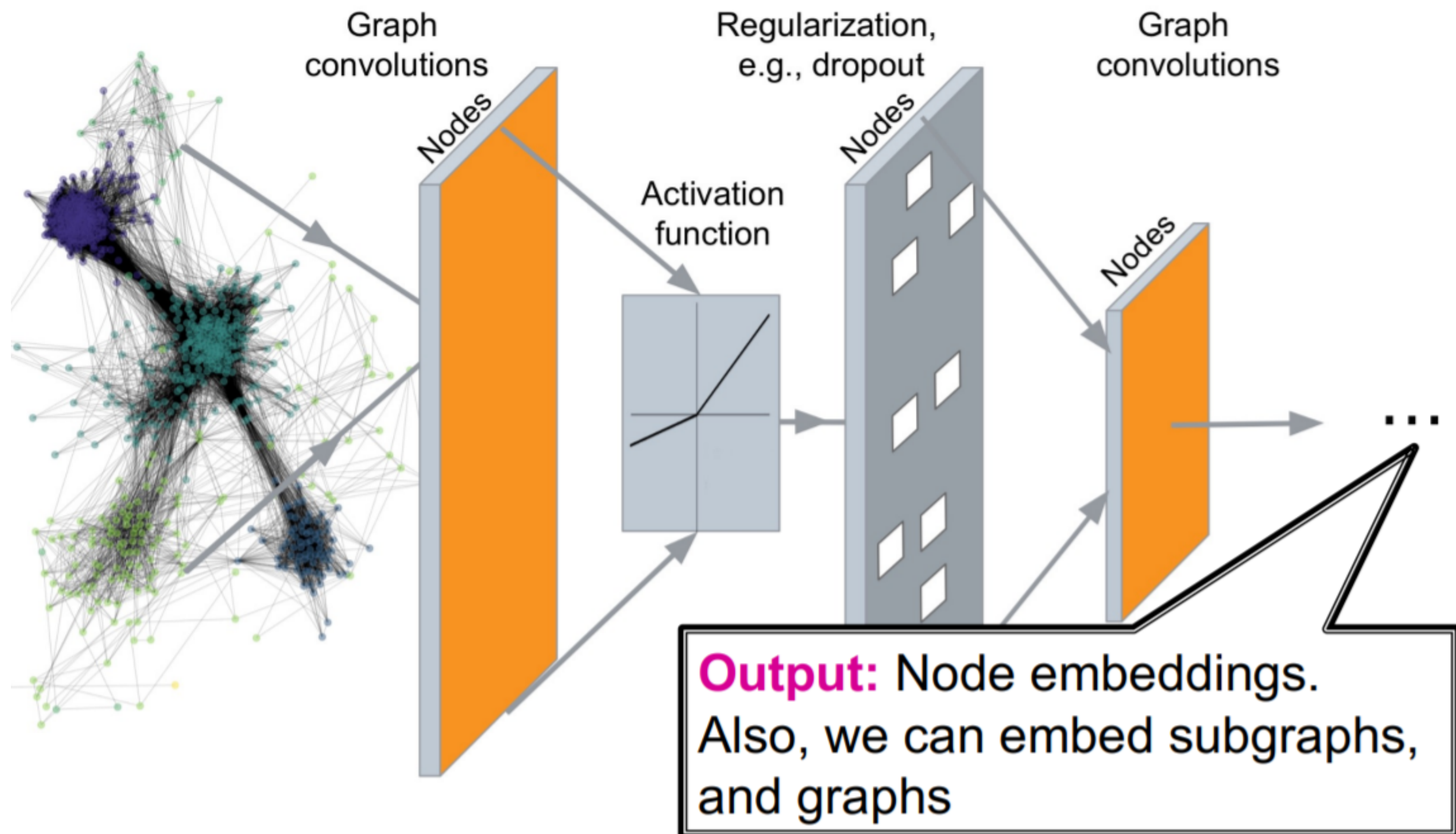
- Tasks:
  - Classifying toxic v non-toxic molecules
  - Identifying anomalous graphs

Graph embeddings

- Sum embeddings of individual nodes
  - $z_G = \sum_{v \in G} z_v$
- Introduce a "virtual node" to represent the (sub)graph and run a standard graph embedding technique on that node

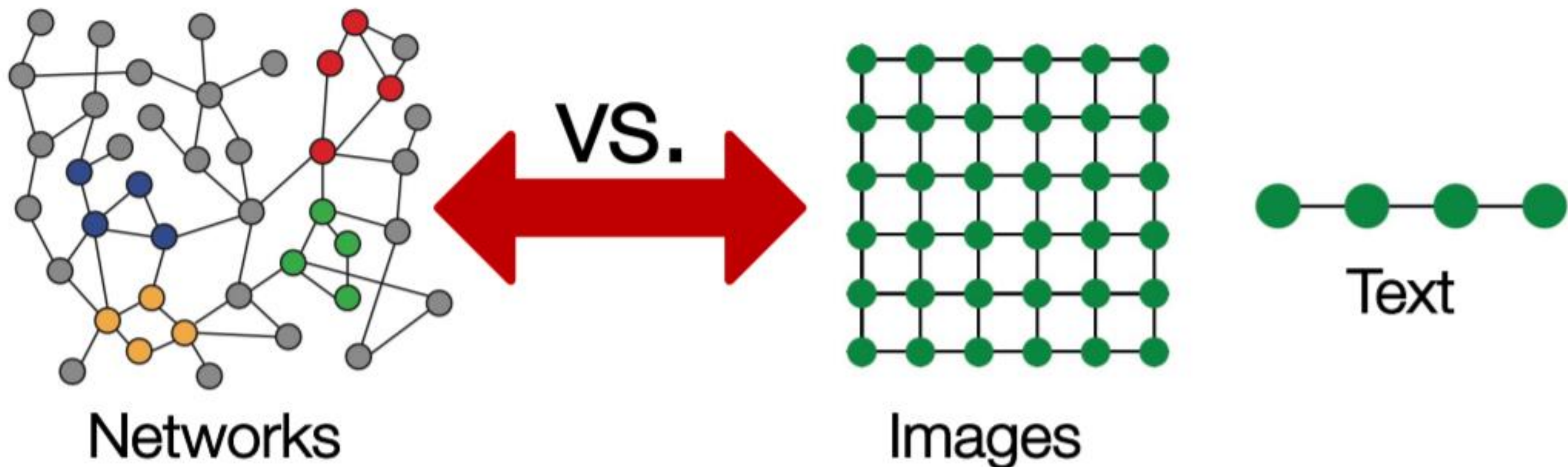


Deep Graph Encoders



Why deep learning on graphs is difficult

- Arbitrary size and complex topological structure (i.e., no spatial locality like grids)

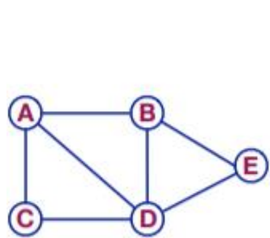


- No fixed node ordering or reference point
- Often dynamic and have multimodal features

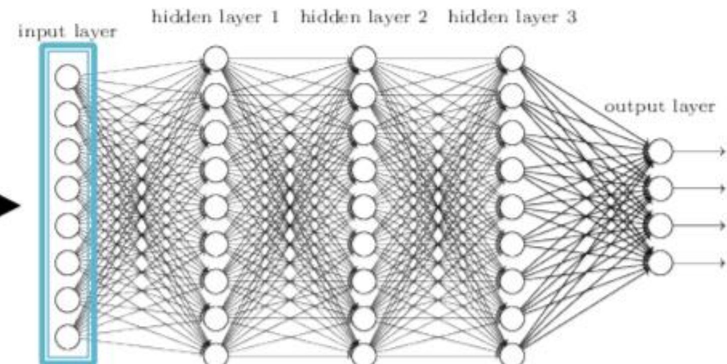
- Assume we have a graph G
- A is the adjacency matrix (assume binary)
- $X \in R^{m \times |V|}$ is a matrix of node features
- v: a node in V; $N(v)$: the set of neighbours of v.
- Node features:
  - Social networks: user profile, user image
  - Biological networks: gene expression profiles,…
  - When there is no node feature in the graph dataset:
    - Indicator vectors (one-hot encoding of a node)
    - Vector of constant 1: [1, 1, …, 1]

A Naïve Approach: A fully-connected network

- Join adjacency matrix and features
- Feed them into a deep neural net



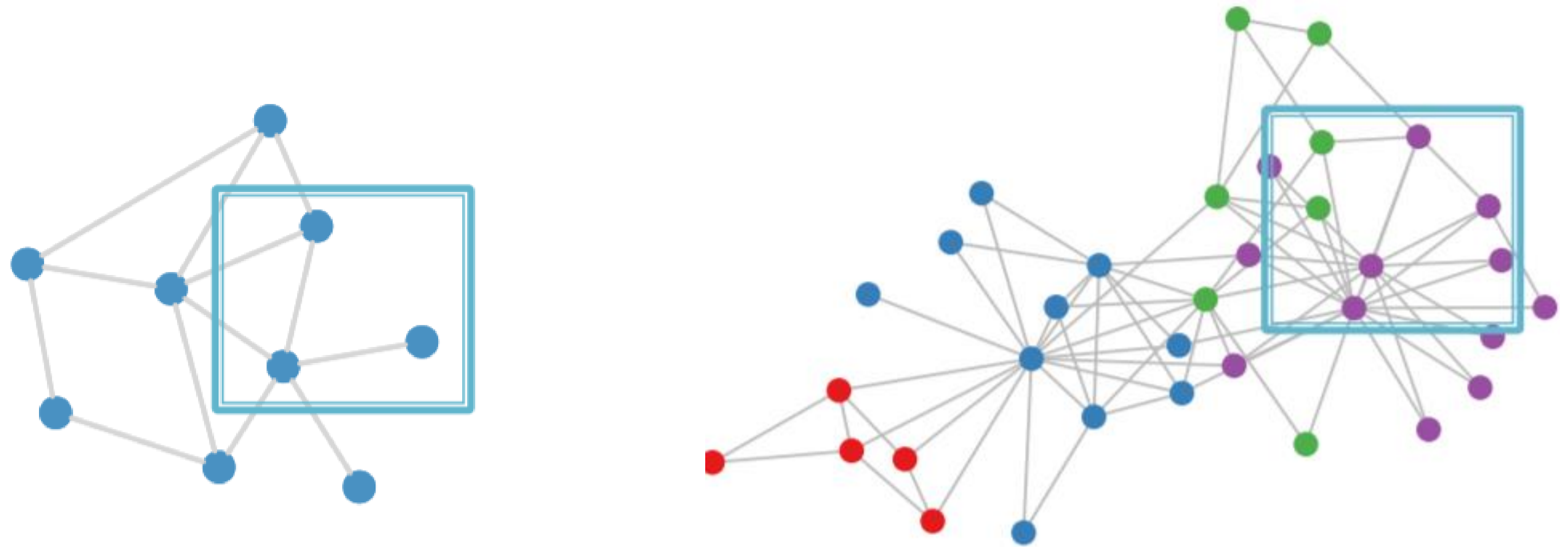
	A	B	C	D	E	Feat	
A	0	1	1	1	0	1	0
B	1	0	0	1	1	0	0
C	1	0	0	1	0	0	1
D	1	1	1	0	1	1	1
E	0	1	0	1	0	1	0



?

- Issues:
  - O(|V|) parameters
  - Not applicable to graphs of different sizes
  - Sensitive to node ordering

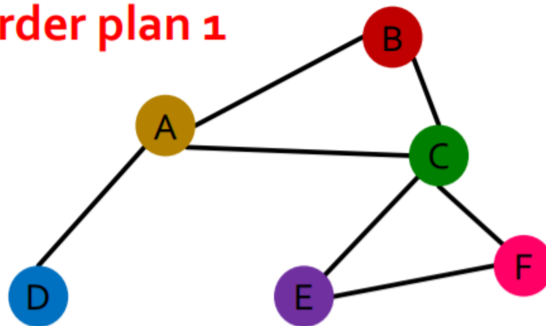
Idea: Convolutional network



Windows now need to be over neighbourhoods
Graphs are permutation invariant – the convolutional layer should account for that

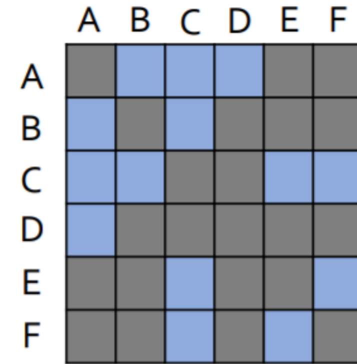
- **Graph does not have a canonical order of the nodes!**

Order plan 1

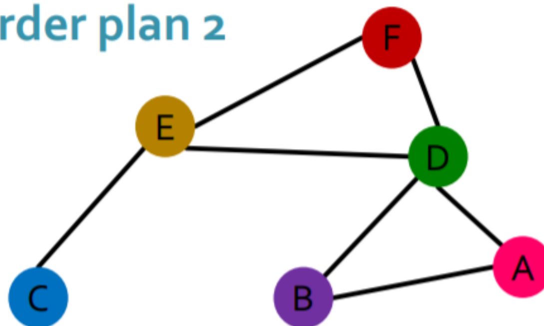


Node features X_1

Adjacency matrix A_1

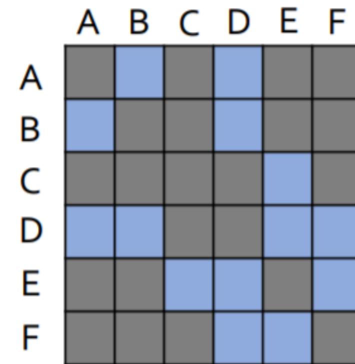


Order plan 2



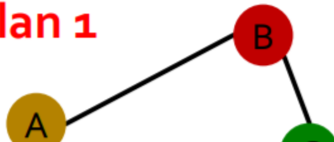
Node features X_2

Adjacency matrix A_2



- Graph does not have a canonical order of the nodes!

Order plan 1



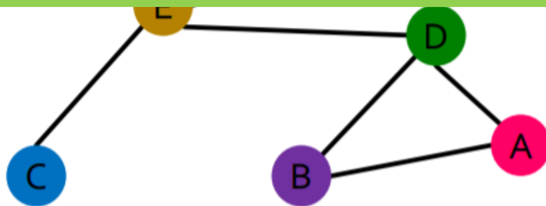
Node features X_1

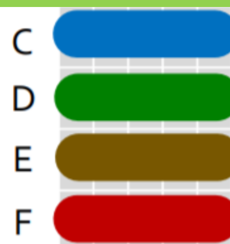
Adjacency matrix A_1

A B C D E F

A
B

Graph and node representations should be the same for Order plan 1 and Order plan 2





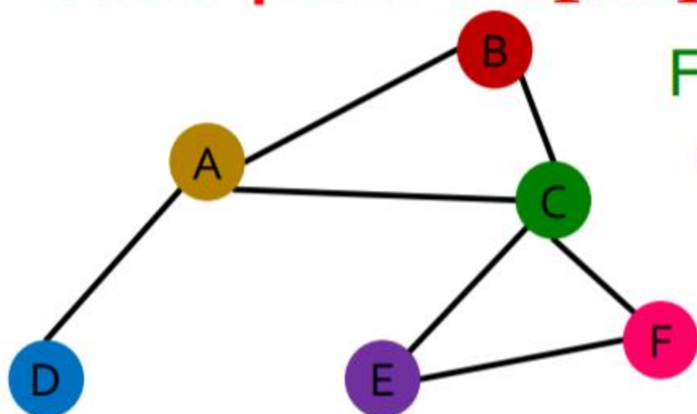
C
D
E
F

Permutation invariance

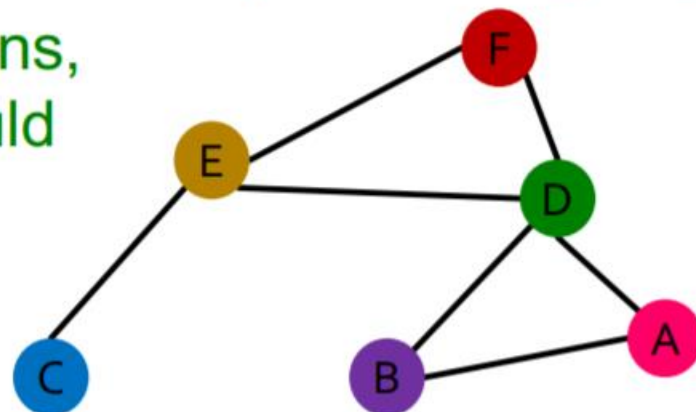
- Graph representation is the same for two order plans
- If we learn a function f that maps a graph G=(A, X) to vector R^d then $f(A_1, X_1) = f(A_2, X_2)$

Order plan 1: A_1, X_1

Order plan 2: A_2, X_2



For two order plans, output of f should be the same!



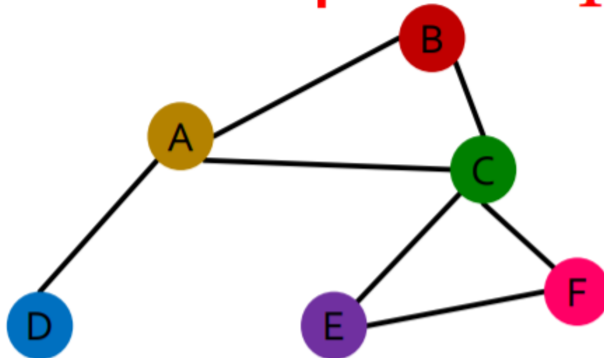
Permutation invariance

- A function f that maps a graph $G = (A, X)$ to a vector R^d
- Then if $f(A_i, X_i) = f(A_j, X_j)$ for any order plan *i* and *j*, we say *f* is a permutation invariant function

Permutation equivariance

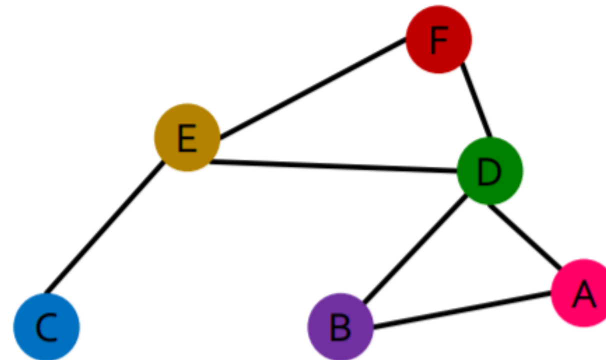
- Node representation should be the same regardless of order plans

Order plan 1: A_1, X_1



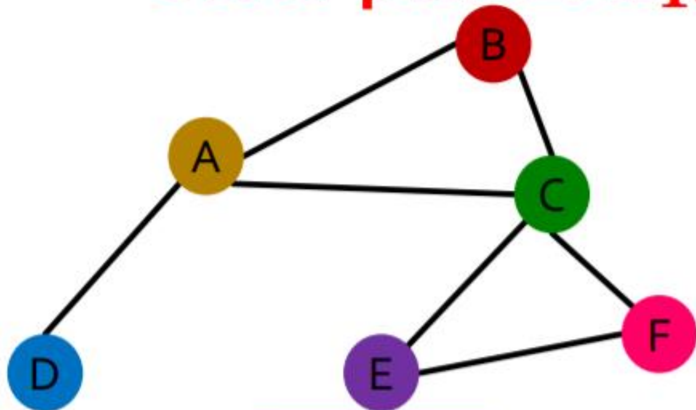
$f(A_1, X_1) =$ A B C D E F

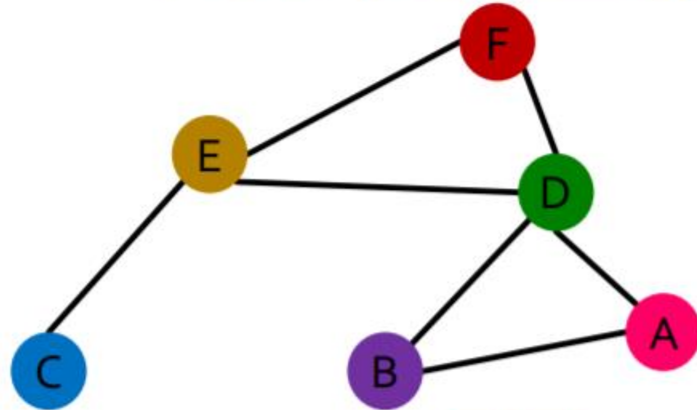
Order plan 2: A_2, X_2



$f(A_2, X_2) =$ A B C D E F

Order plan 1: A_1, X_1 Order plan 2: A_2, X_2





Representation vector of the brown node A

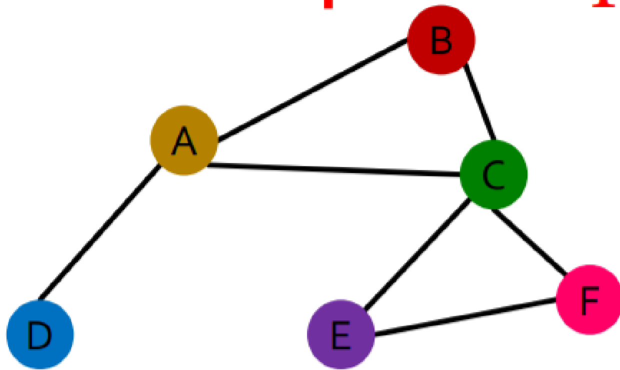
$$f(\boldsymbol{A}_1, \boldsymbol{X}_1) = \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix}$$

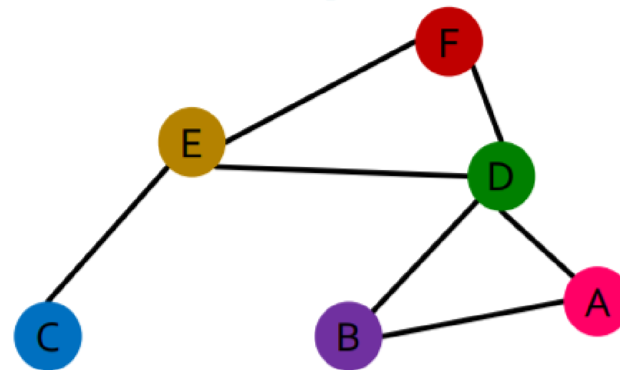
$$f(\boldsymbol{A}_2, \boldsymbol{X}_2) = \begin{matrix} A \\ B \\ C \\ D \\ E \\ F \end{matrix}$$

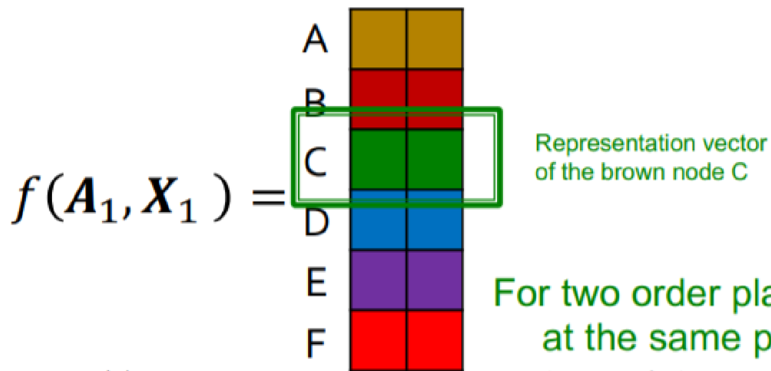
Representation vector of the brown node E

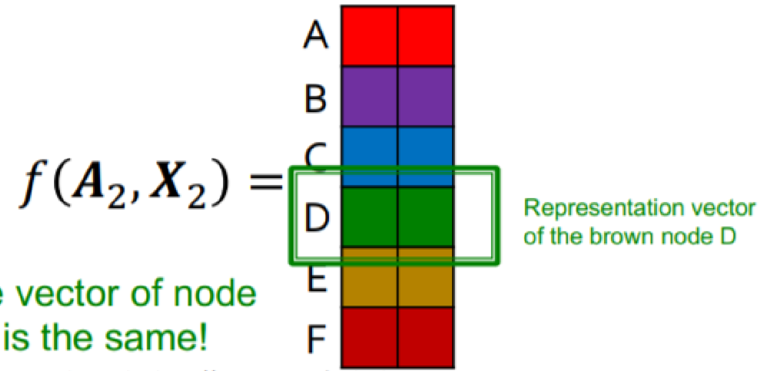
For two order plans, the vector of node at the same position is the same!

Order plan 1: A_1, X_1 Order plan 2: A_2, X_2









For two order plans, the vector of node at the same position is the same!

Permutation Equivariance

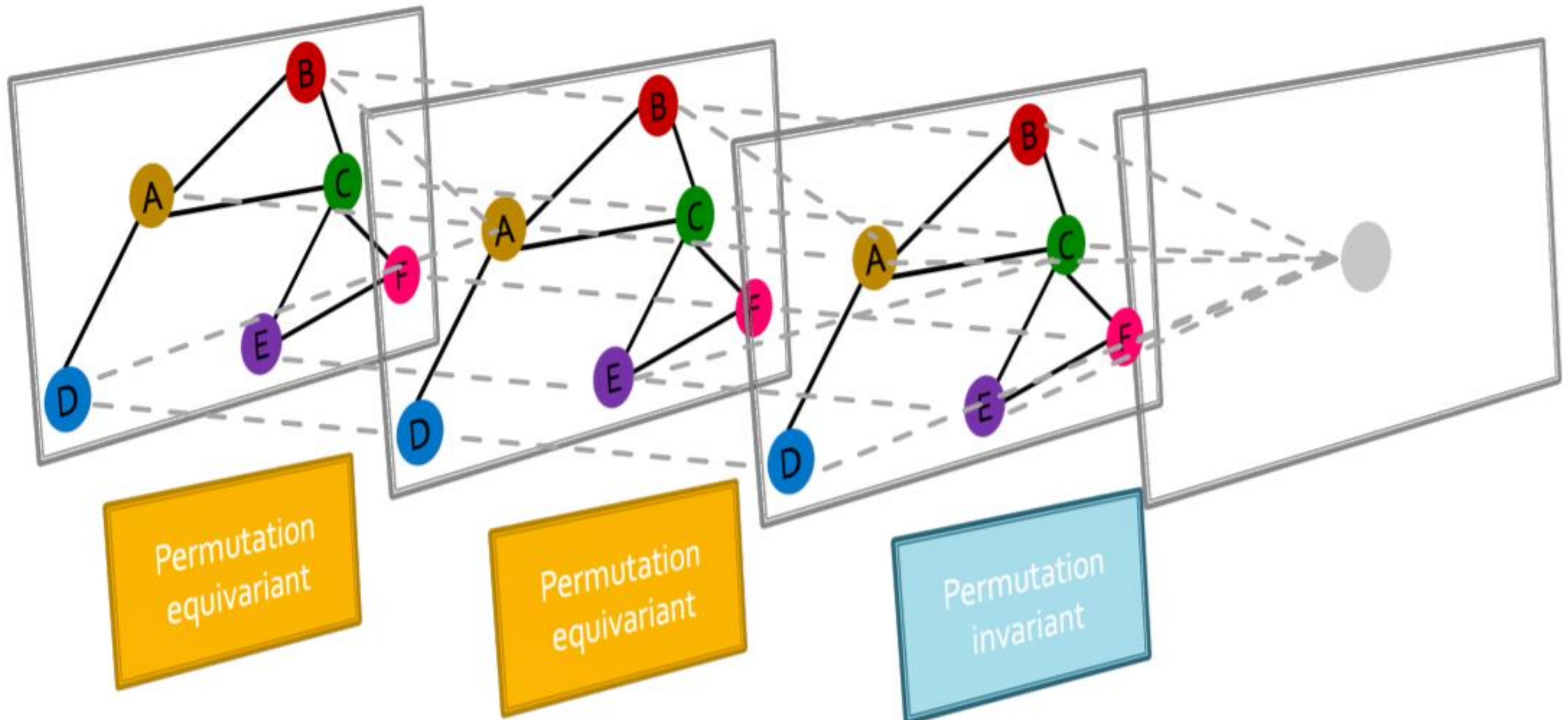
- For node representation
- Consider learning a function f that maps a graph $G = (A, X)$ to a matrix $R^{m \times d}$
  - Graph has *m* nodes, each row is the embedding of a node
- If *G2* is a permutation of *G1* such that nodes *i* and *j* are permuted, want

  $$f(G1)_i, f(G2)_i = f(G2)_j, F(G1)_j$$

    - R_i is the i-th row of *R*
- If this property holds for any pair of order plan *i* and *j*, we say f is a permutation equivariant function

Graph Neural Networks

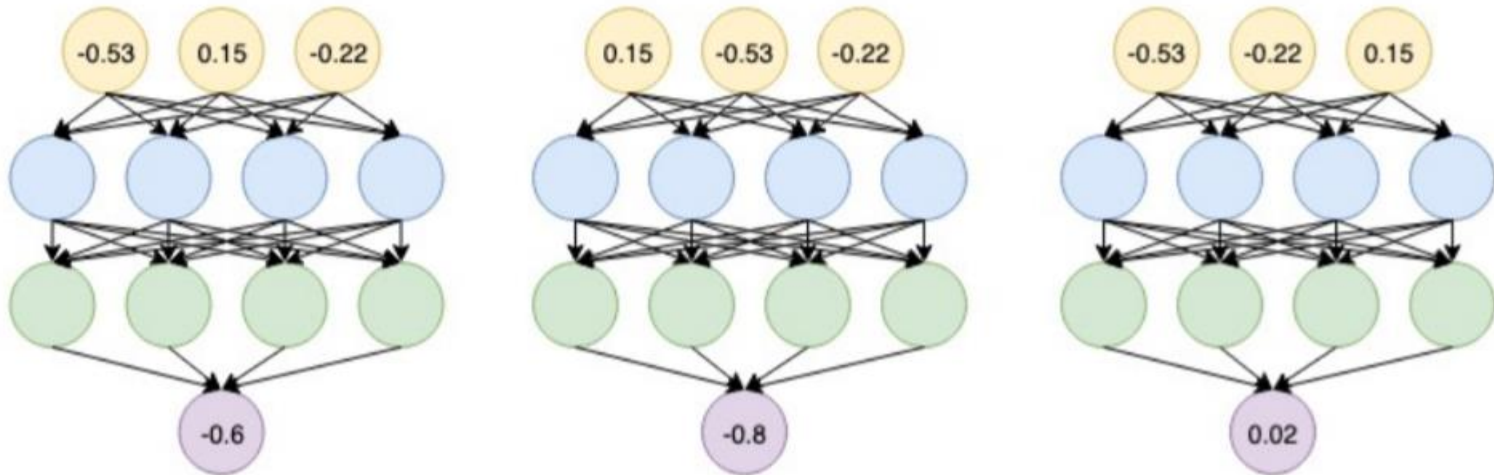
- Graph neural networks consist of multiple permutation equivariant/invariant

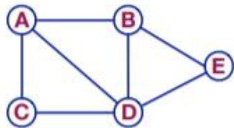


Graph Neural Networks

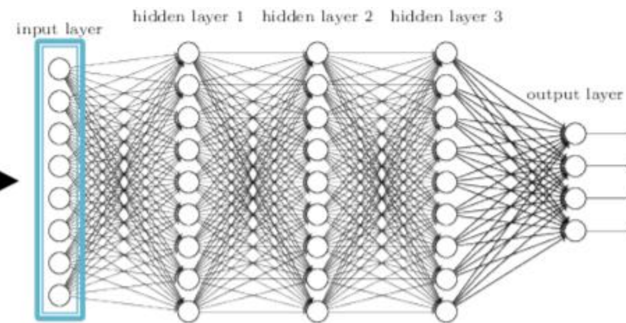
- Are MLPs permutation invariant/equivariant? No.

Switching the order of the input leads to different outputs!





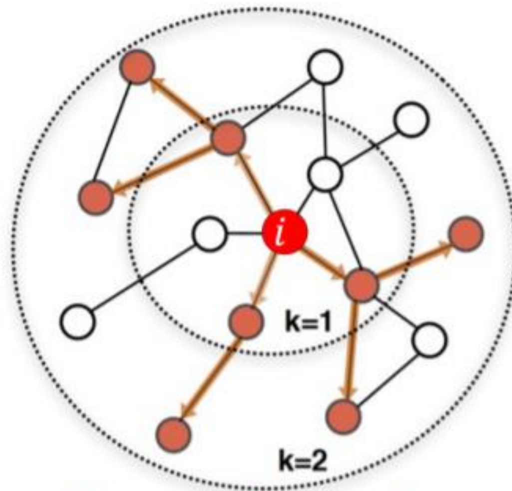
	A	B	C	D	E	Feat	
A	0	1	1	1	0	1	0
B	1	0	0	1	1	0	0
C	1	0	0	1	0	0	1
D	1	1	1	0	1	1	1
E	0	1	0	1	0	1	0



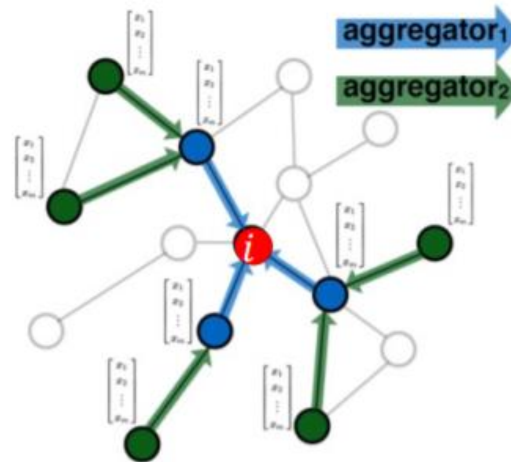
?

Graph Convolutional Networks

- Idea: node's neighbourhood defines a computation graph



Determine node computation graph

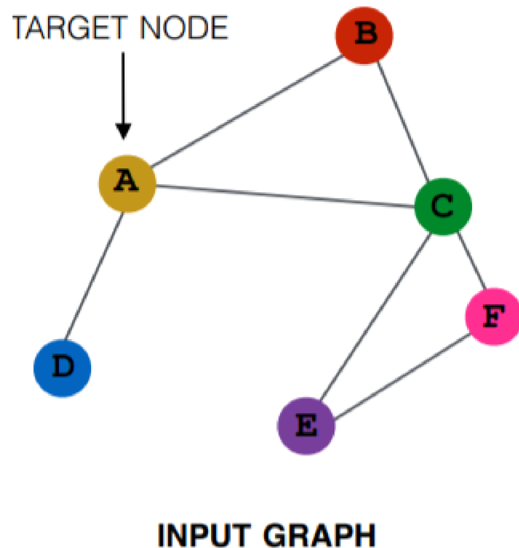


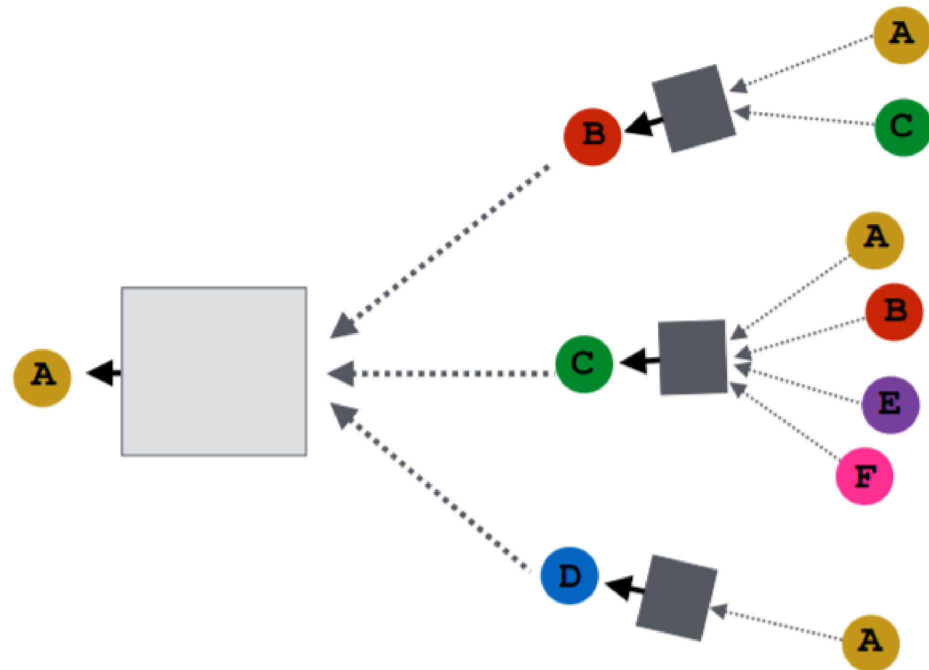
Propagate and transform information

Learn how to propagate information across the graph to compute node features

Aggregate Neighbours

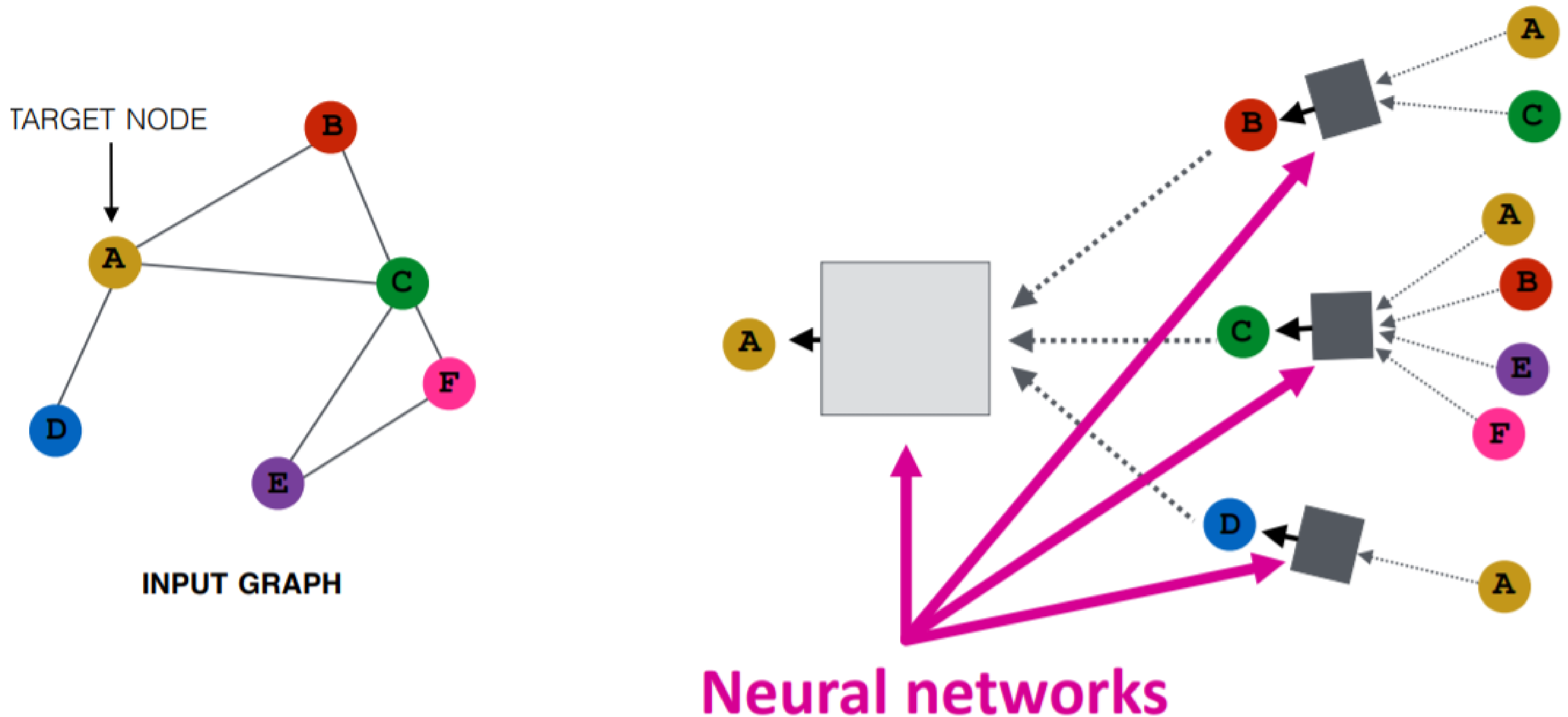
- Generate node embeddings based on local network neighbourhoods





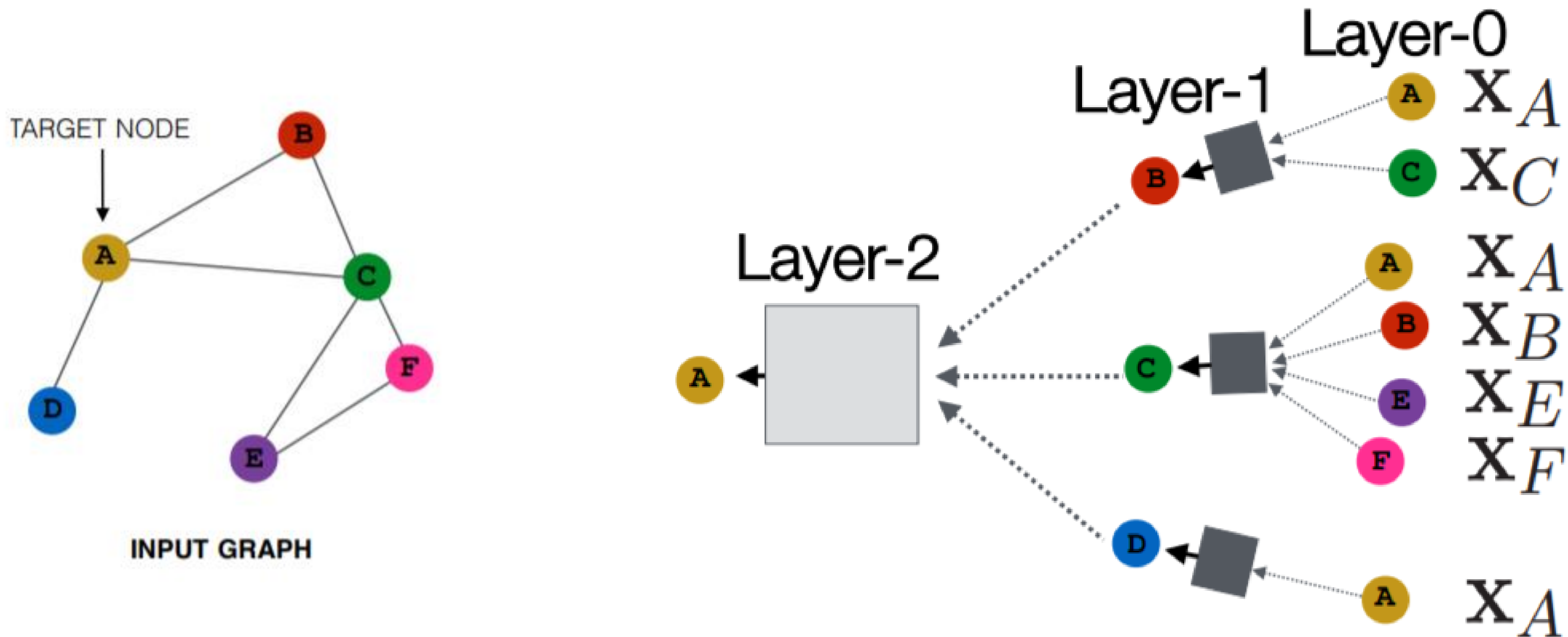
Aggregate Neighbours

- Intuition: nodes aggregate information from their neighbours using neural networks



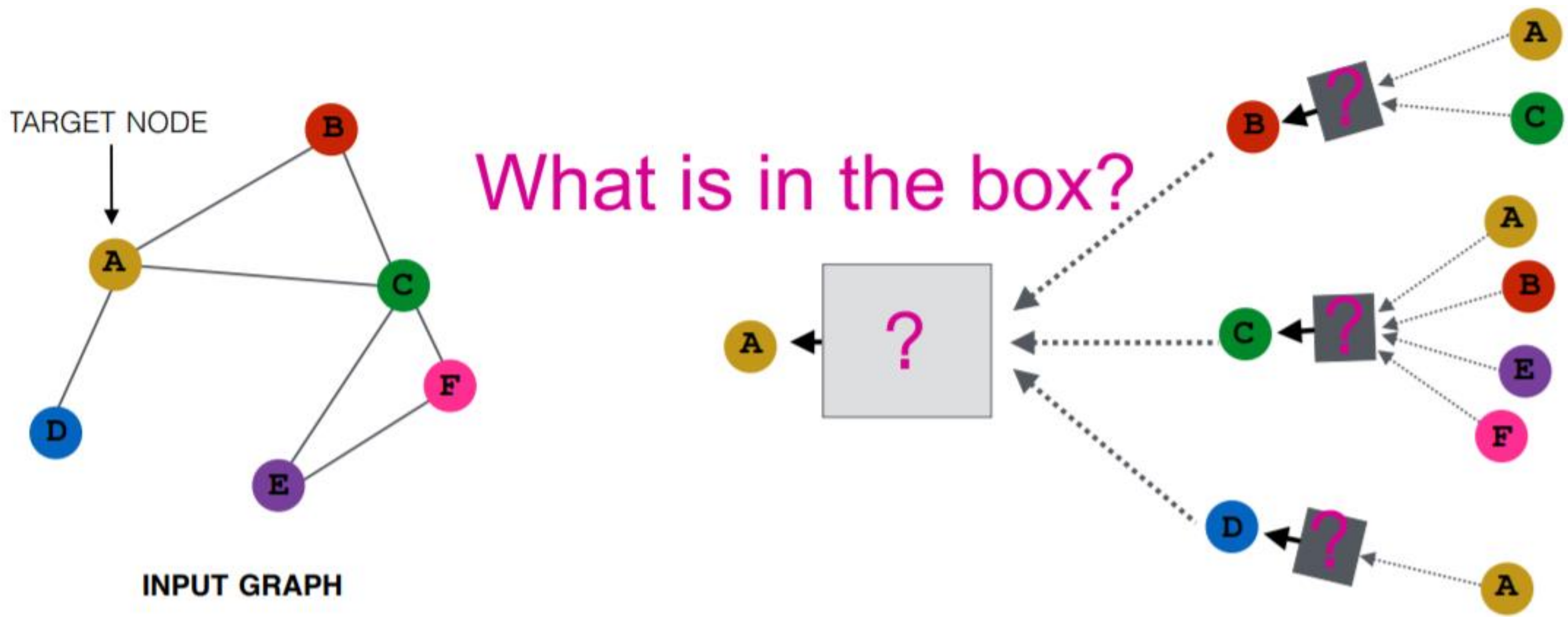
Deep Model: Many Layers

- Model can be of arbitrary depth:
  - Nodes have embeddings at each layer
  - Layer-0 embedding of node v is its input feature x_v
  - Layer-k embeddings gets information from nodes that are k hops away



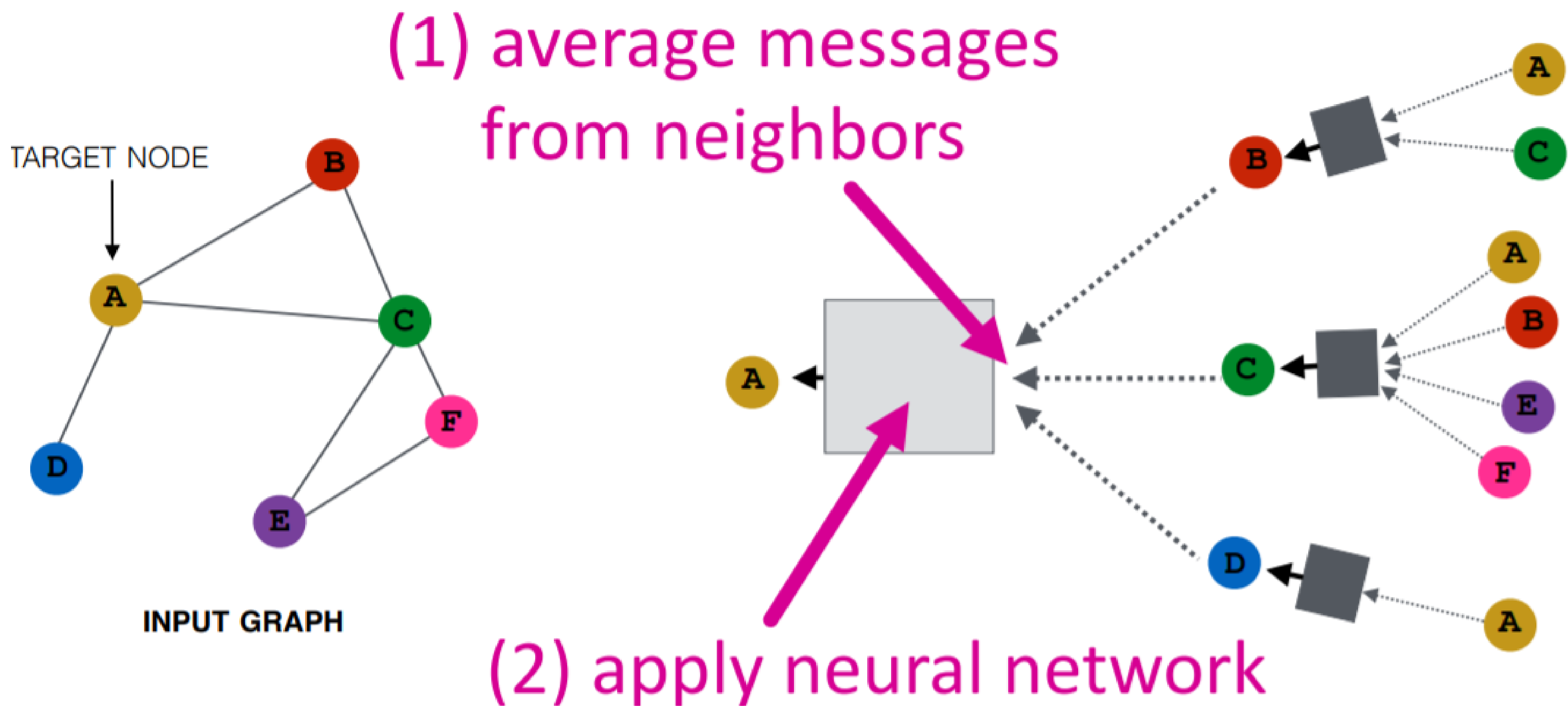
Neighbourhood aggregation

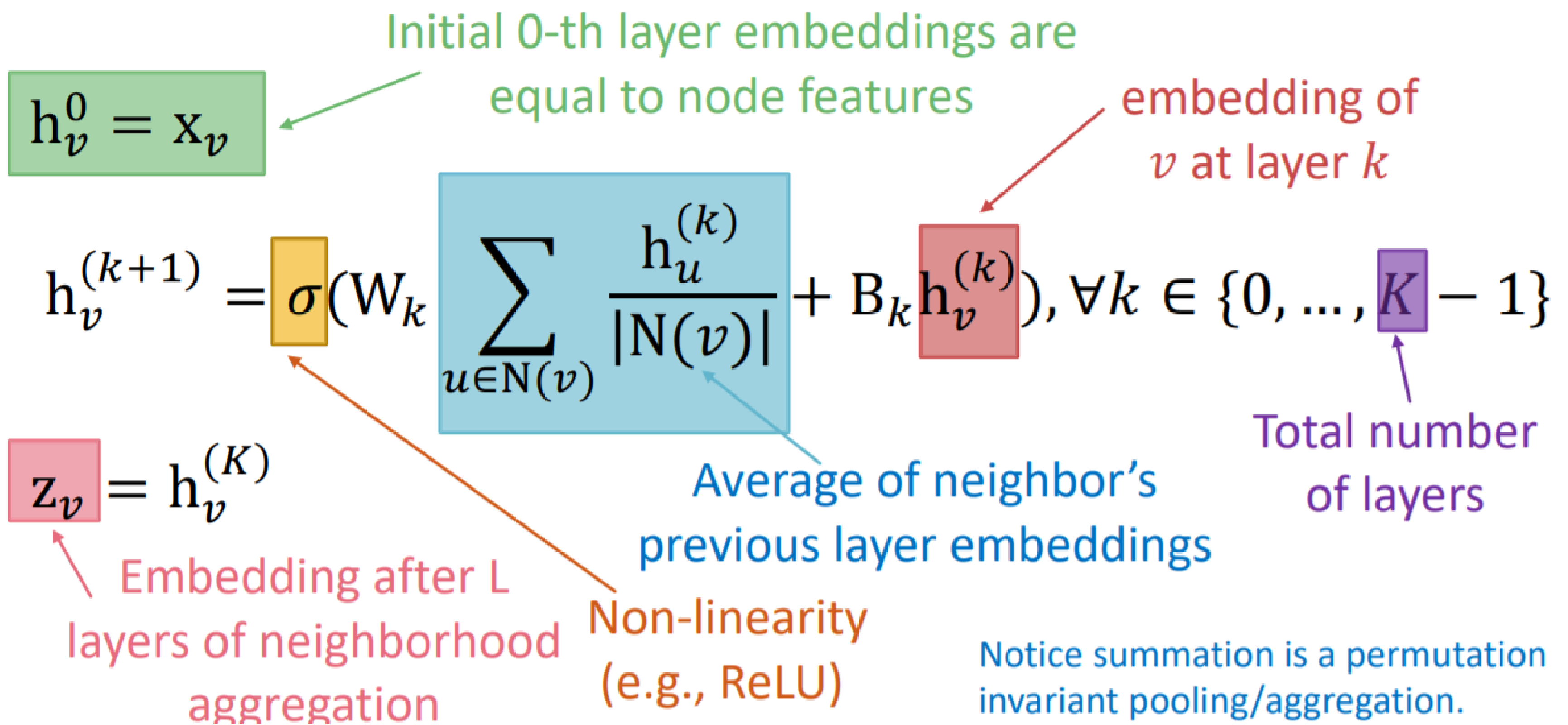
- Different approaches to aggregate information across the layers



Neighbourhood aggregation

- Basic approach: average information from neighbours and apply a neural network



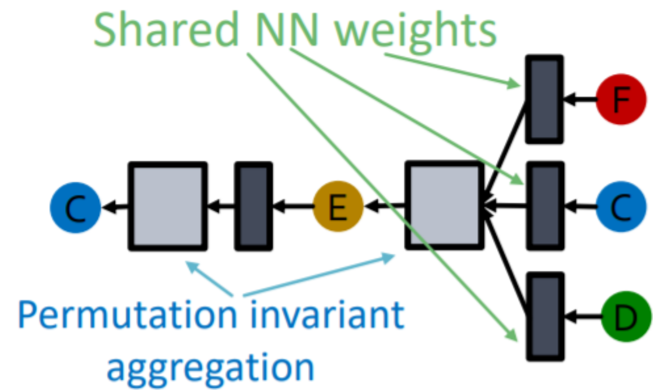


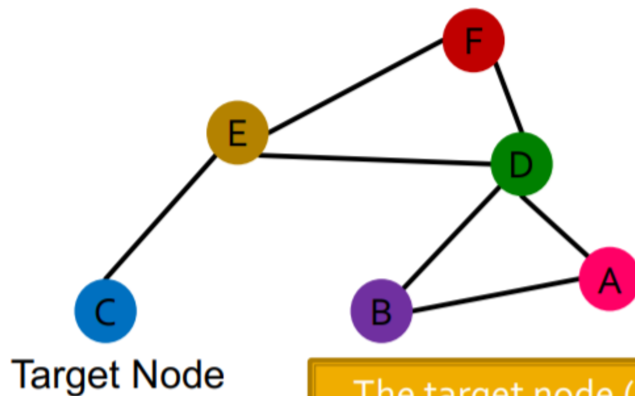
$h_v^0 = x_v$
Initial 0-th layer embeddings are equal to node features
embedding of v at layer k
$h_v^{(k+1)} = \sigma(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}), \forall k \in \{0, \dots, K-1\}$
$z_v = h_v^{(K)}$
Total number of layers
Average of neighbor's previous layer embeddings
Embedding after L layers of neighborhood aggregation
Non-linearity (e.g., ReLU)
Notice summation is a permutation invariant pooling/aggregation.

Permutation Equivariance

- Message passing and neighbour aggregation in graph convolution networks is permutation invariant
  - Aggregation/message passing to a node only depends on the neighbours

Message passing and neighbor aggregation in graph convolution networks is permutation equivariant.





The target node (blue) has the same computation graph for different order plans

Node feature X_2

A B C D E F

Adjacency matrix A_2

A B C D E F

Model parameters

Trainable weight matrices
(i.e., what we learn)

$$h_v^{(0)} = x_v$$

$$h_v^{(k+1)} = \sigma\left(W_k \sum_{u \in N(v)} \frac{h_u^{(k)}}{|N(v)|} + B_k h_v^{(k)}\right), \forall k \in \{0..K-1\}$$

$$z_v = h_v^{(K)}$$

Final node embedding

- $h_v^{(k)}$: the hidden representation of node *v* at layer *k*
  - W_k:the weight matrix for neighbourhood aggregation
  - B_k: weight matrix for transforming hidden vector of self

Model training

- Supervised loss:

$$\min_{\theta} \sum_{v} L(y(v), f(z_v))$$

  - y(v) label of v
  - L: L2 loss or cross-entropy
- Unsupervised setting
  - No node label available
  - Use the graph structure as the supervision

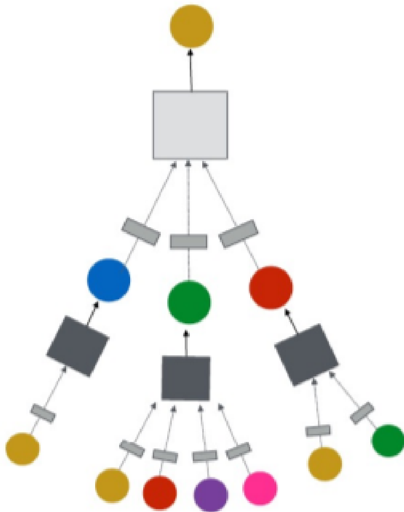
Unsupervised training

- Similar nodes should have similar embedding
- $L = \sum_{u,v} CE(y_{u,v}, similarity(z_u, z_v))$
- $y_{u,v} = 1$ is u, v are similar, 0 otherwise
- CE is the cross entropy
- Similarity can be the dot product
- Node similarity can be based on
  - Random walks
  - Node proximity in the graph
  - Adjacency matrix factorization

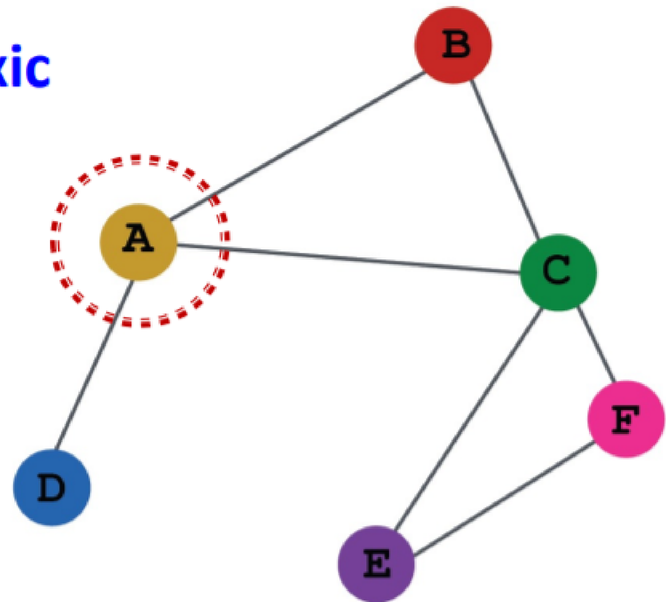
Supervised training

- Directly train the model for a supervised task (e.g., node classification

Safe or toxic drug?



Safe or toxic drug?



E.g., a drug-drug interaction network

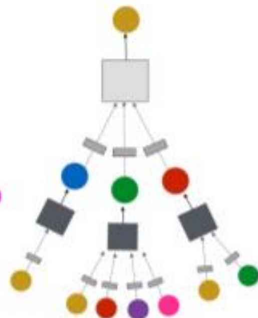
- Label: y_v
- Prediction: $z_v^T\theta$
- Cross-entropy loss:

$$\mathcal{L} = \sum_{v \in V} y_v \log(\sigma(\mathrm{z}_v^{\mathrm{T}}\theta)) + (1 - y_v)\log(1 - \sigma(\mathrm{z}_v^{\mathrm{T}}\theta))$$

Encoder output:
node embedding

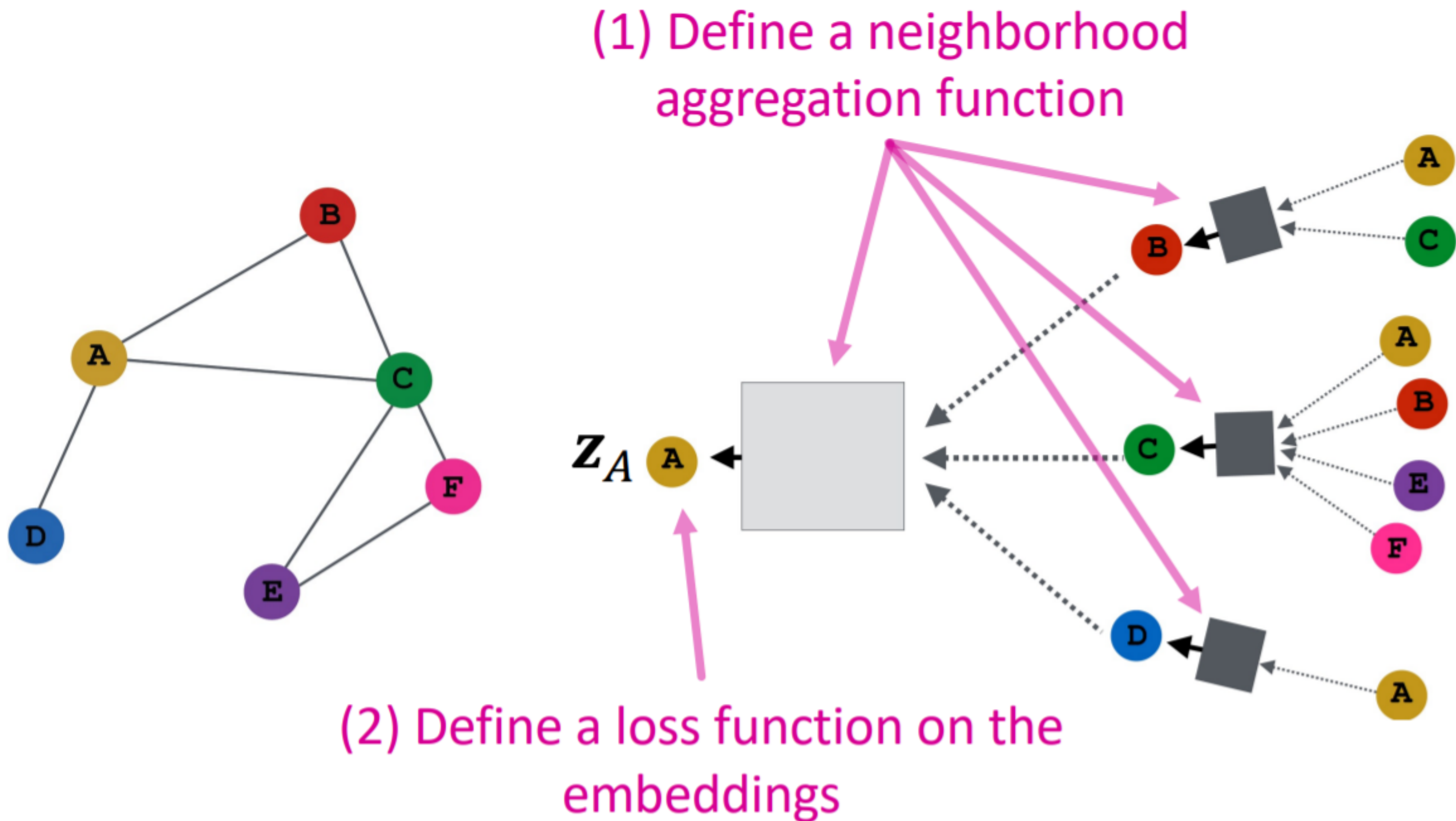
Classification
weights

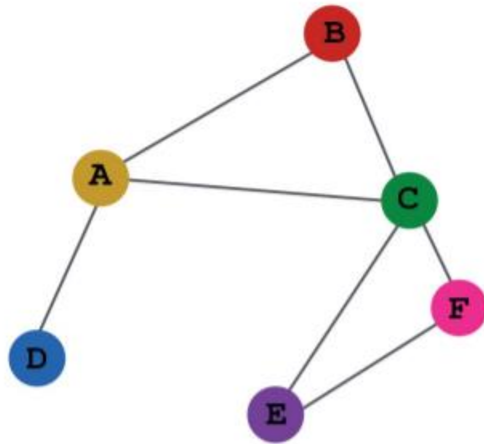
Node class
label



Safe or toxic drug?

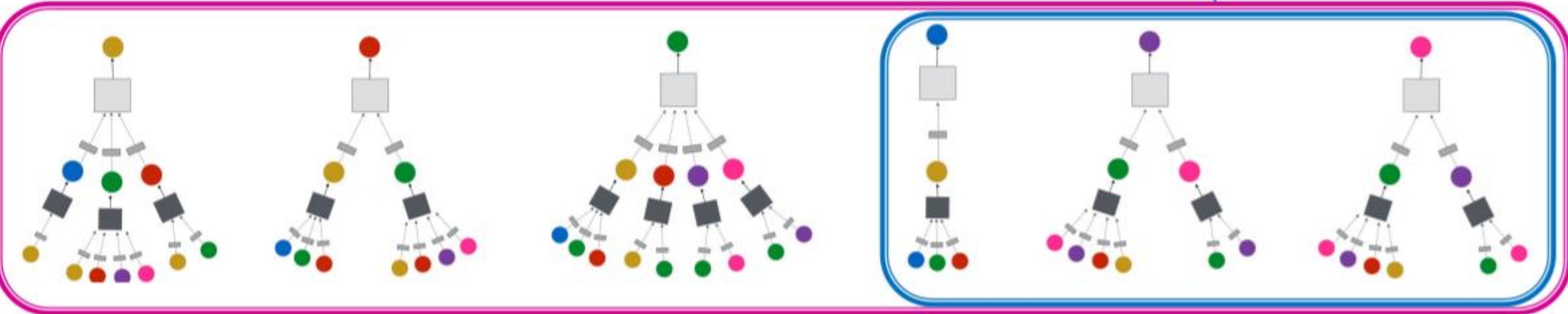
Model design: overview





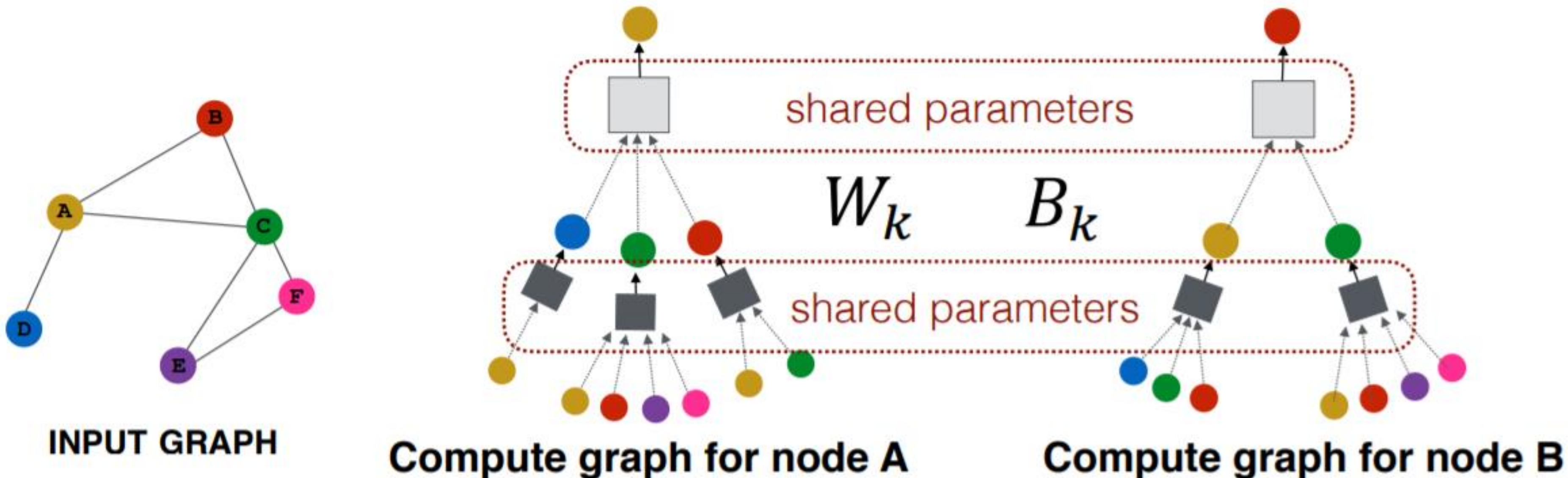
(4) Generate embeddings for nodes as needed

Even for nodes we never trained on!

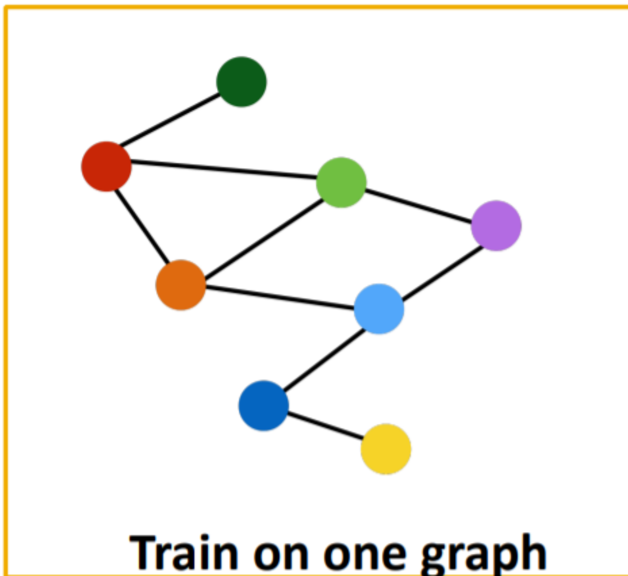


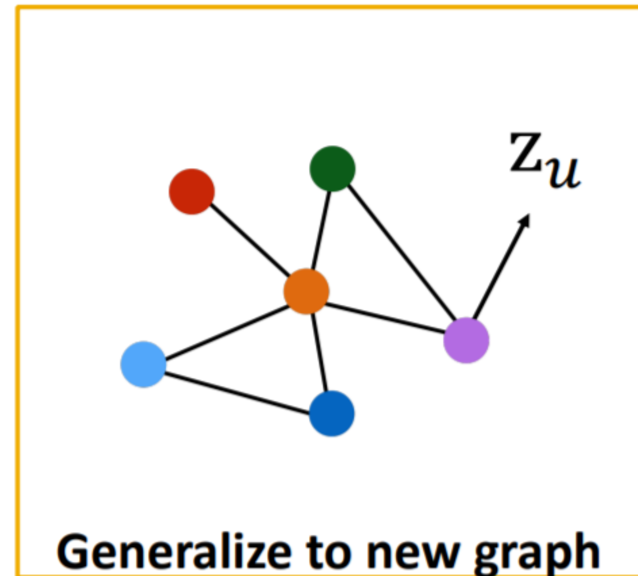
Inductive Capability

- 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: New Graphs

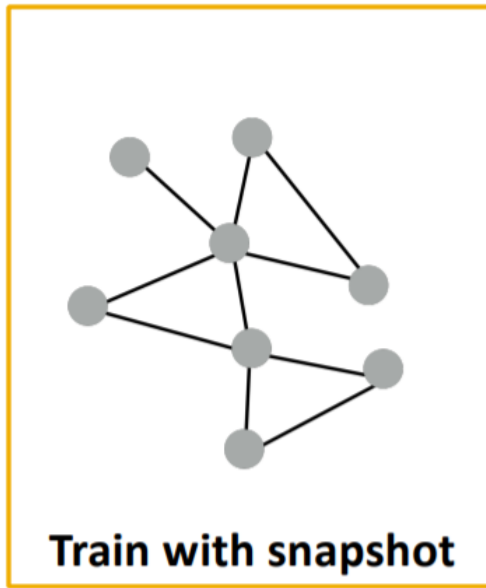


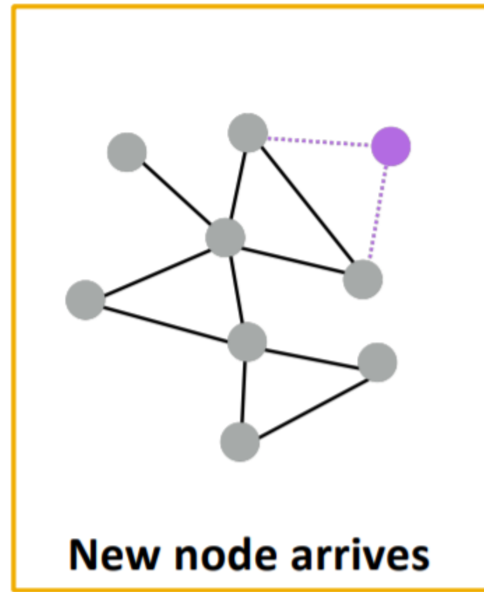


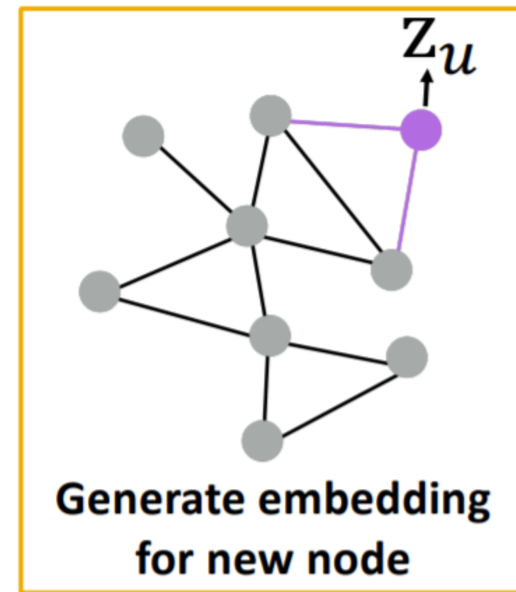
Inductive node embedding → Generalize to entirely unseen graphs

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

Inductive Capability: New Nodes



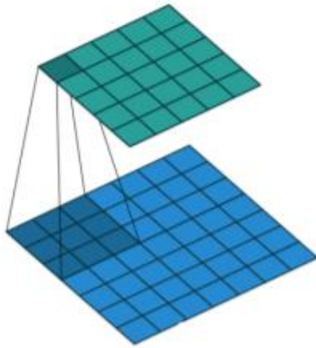




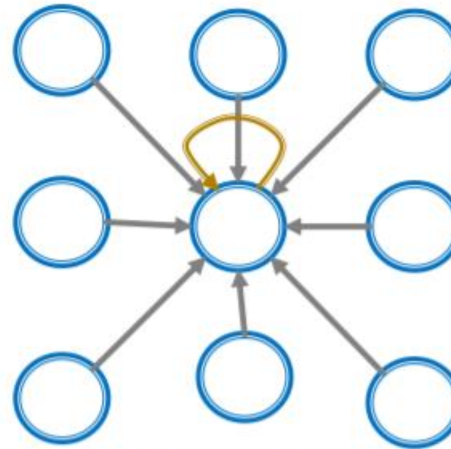
- Can generate new embeddings on the fly

GNN and CNN

Convolutional neural network (CNN) layer with 3x3 filter:



Image



Graph

GNN formulation: $h_v^{(l+1)} = \sigma(\mathbf{W}_l \sum_{u \in N(v)} \frac{h_u^{(l)}}{|N(v)|} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$

CNN formulation: $h_v^{(l+1)} = \sigma(\sum_{u \in N(v)} \mathbf{W}_l^u h_u^{(l)} + B_l h_v^{(l)}), \forall l \in \{0, \dots, L-1\}$

Key difference: We can learn different W_l^u for different "neighbor" u for pixel v on the image. The reason is we can pick an order for the 9 neighbors using **relative position** to the center pixel: {(-1,-1). (-1,0), (-1, 1), …, (1, 1)}