Learning from Networks

Graph Neural Networks

Fabio Vandin

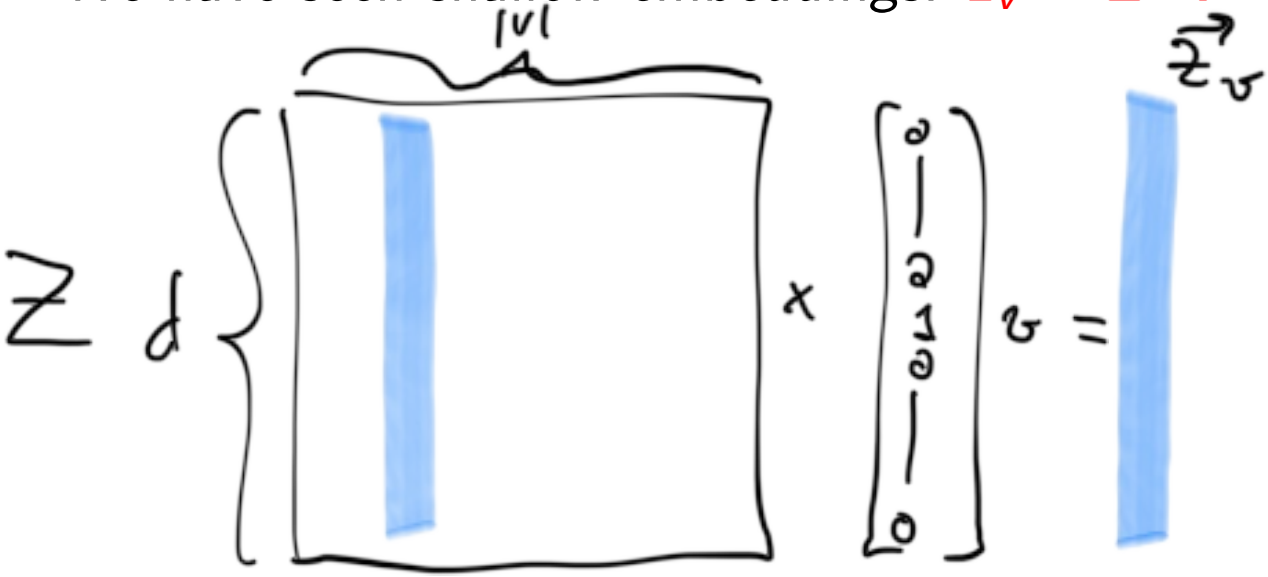
November 21st, 2024

Shallow Embeddings

Node embedding task: given graph $G = (V, E)$, represent each $v \in V$ as $ENC(v) = \mathbf{z}_v \in \mathbb{R}^d$ such that similarities $S(u, v)$ in G are approximated by (decoded) similarities $DEC(\mathbf{z}_u, \mathbf{z}_v)$ for all $u, v \in V$:

$$S(u, v) \approx DEC(\mathbf{z}_u, \mathbf{z}_v)$$

We have seen *shallow embeddings*: $\mathbf{z}_v = \mathbf{Z} \cdot \mathbf{v}$



Shallow Embeddings: Limitations

Large number of parameters: high *complexity* of the *model*

Inherently *transductive*: cannot generate embeddings for nodes that are not present during the *training* phase

We would like *inductive* embedding methods: can generate the embeddings for new nodes

Do not incorporate node features: many networks have node features that can/should be used to produce the embedding

They are *unsupervised* methods: learned embeddings are independent of the (downstream) ML task

Graph Neural Network Embeddings: Main Idea

Instead of learning $\mathbf{z}_v$ for each $v \in V$, learn a function $f : V \to \mathbb{R}^d$

f is computed by a *neural network* that depends on the structure of the graph $\Rightarrow$ *graph neural network* (GNN)

Given the graph G and features of the nodes $v \in V$, the GNN can compute the embedding of *any* node $u \in V$

Encoder-decoder framework:

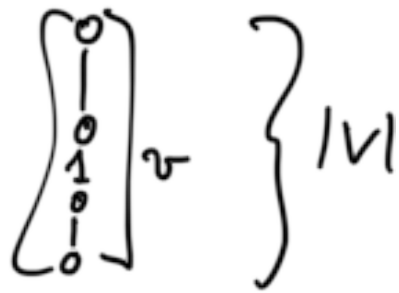
- the encoder $ENC()$ is a neural network
- other components (similarity, decoder, loss function): as for shallow embeddings

$\Rightarrow$ the parameters to be learned from the data are in the neural network

GNN: Setup

We have node features:

- social networks: user profile info, activity info, ...
- biological networks: gene expression profiles, functional information, ...
- what if no (external) features available?
  - compute node features: clustering coefficient, centrality scores, etc.
  - *one-hot encoding of a node* :



GNN: Setup

We have node features:

- social networks: user profile info, activity info, ...
- biological networks: gene expression profiles, functional information, ...
- what if no (external) features available?
  - compute node features: clustering coefficient, centrality scores, etc.
  - *one-hot encoding of a node*
  - vector of all 1's

Given

- graph $G = (V, E)$, with $|V| = n$
- $\mathbf{A}$ is the adjacency matrix of G
- each node $v \in V$ has vector $\mathbf{x}_v \in \mathbb{R}^r$ of r *features*
- $\mathbf{X} \in \mathbb{R}^{r \times n}$ is the matrix of nodes features

Goal: we want to combine the features $\mathbf{x}_v$ of a node and the structural (topological) information from G to obtain encodings for each node $v \in V$

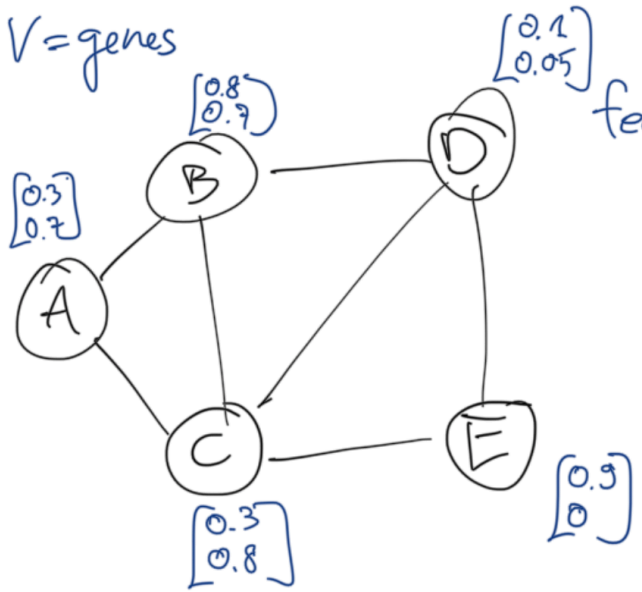
Ideas?

Naïve Approach

Idea:

- for each node $v \in V$, build an input vector that contains both the node features $\mathbf{x}_v$ and the *adjacency matrix vector* for v
- $ENC(v)$ is the output of a (deep) neural network whose input are the vectors above

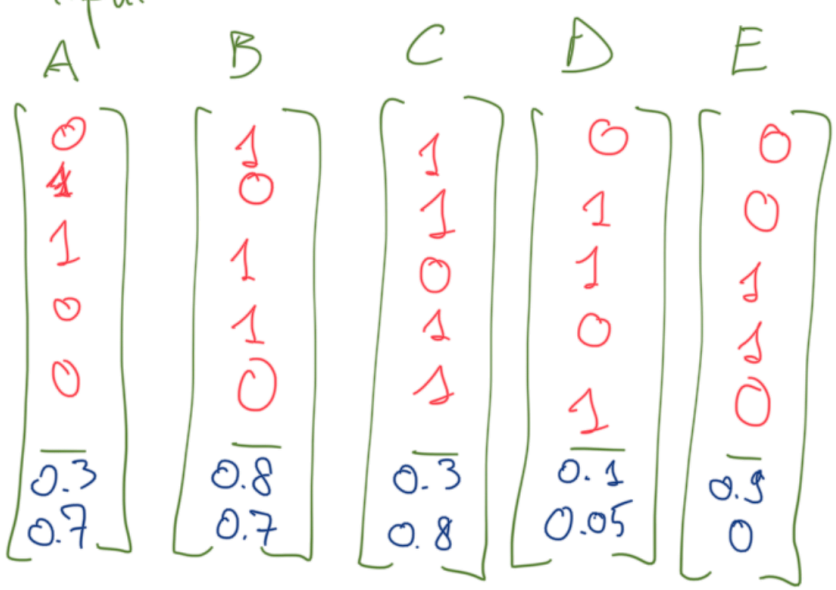
Example

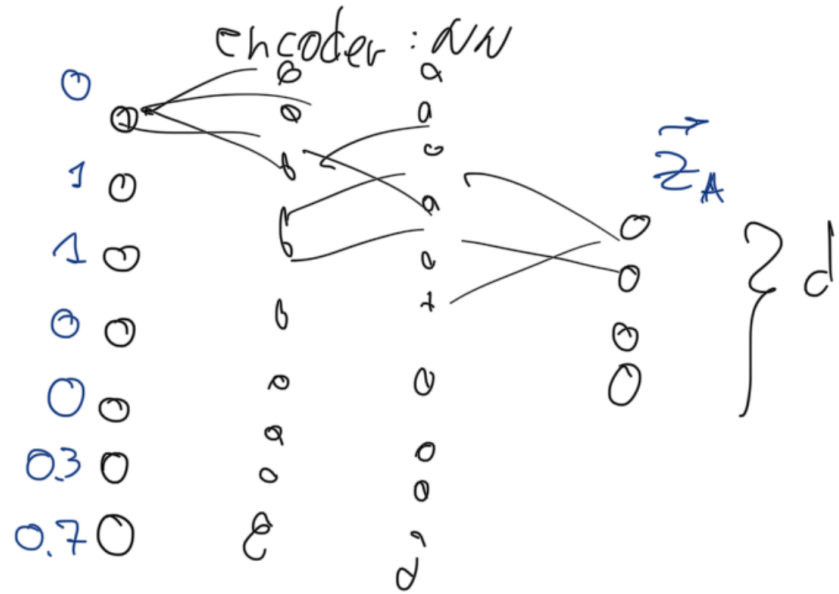


adjacency matrix:

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

input vectors for the NN:





Naïve Approach: Issues

Number of parameters? $\Omega(n)$ → because the input layer has $\Omega(n)$ nodes

Not applicable to graphs of different sizes → they have input layers of different sizes

Depends on the node ordering → if I change the order of nodes in the adjacency matrix, the embeddings of the nodes change

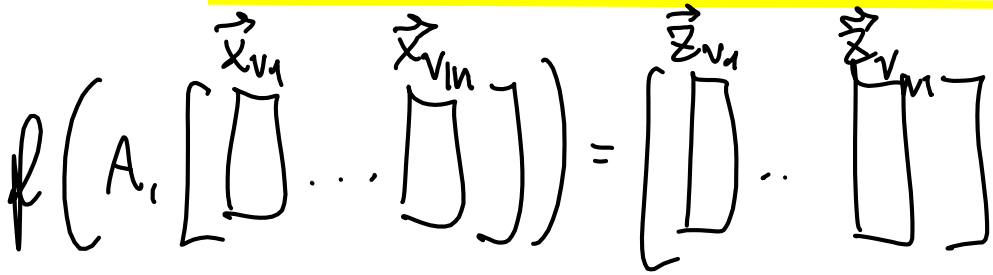
Permutation Invariance and Permutation Equivariance

We would like the embedding to be *independent* of the node ordering.

Let f be the function that given the adjacency matrix $\mathbf{A}$ of G and the feature matrix $\mathbf{X}$ of the nodes of G produces in output the embedding matrix $\mathbf{Z}$:

$$\mathbf{Z} = f(\mathbf{A}, \mathbf{X})$$

We would like f to be *permutation invariant* or *permutation equivariant*



Permutation Invariance and Permutation Equivariance (continue)

We would like it to be *permutation invariant* or *permutation equivariant*

Permutation matrix P: each row/column has exactly one 1, all other entries are 0.

Definition

f is *permutation invariant* if $f(\mathbf{PAP}^T, \mathbf{XP}^T) = f(\mathbf{A}, \mathbf{X})$ where $\mathbf{P}$ is a *permutation matrix*.

non originali, ma permutate

Definition

f is *permutation equivariant* if $f(\mathbf{PAP}^T, \mathbf{XP}^T) = f(\mathbf{A}, \mathbf{X})\mathbf{P}^T$ where $\mathbf{P}$ is a *permutation matrix*.

$\vec{P}\vec{A}$: permutare righe, $(\vec{P}\vec{A})\vec{P}^T$: permutare colonne

Example

$\ell(A, X) = z$, $G: |V| = 3$, $d = 2$

$$Z = \underbrace{\left[\begin{array}{c|c|c} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \end{array}\right]}_{|V|} d, \qquad P = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

$$Z \cdot P^T = \left[\begin{array}{c|c|c} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \end{array}\right] \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}^T = \left[\begin{array}{c|c|c} z_{11} & z_{12} & z_{13} \\ z_{21} & z_{22} & z_{23} \end{array}\right] \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix} =$$

$$= \left[\begin{array}{c|c|c} z_{13} & z_{11} & z_{12} \\ z_{23} & z_{21} & z_{22} \end{array}\right]$$

Neural Message Passing Framework

Idea

- the computation proceeds in iterations
- in iteration k, the (hidden) embedding $\mathbf{h}_v^{(k)}$ for node v is updated/computed according to the (hidden) embeddings of nodes $u \in \mathcal{N}(v)$ ⇒ solo nodi vicini, non tutti
- the output embedding for node v is the embedding $\mathbf{h}_v^{(K)}$ after K iterations
- initialization: $\mathbf{h}_v^{(0)} = \mathbf{x}_v$ for all $v \in V$

Formally:

- $\texttt{AGGREGATE}^{(k)}\left(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}\right) = \mathbf{m}_{\mathcal{N}(u)}^{(k)}$: function that given the (hidden) embeddings of neighbours of u at iteration k produces the *message* $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$.
- $\texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right)$

$\texttt{AGGREGATE}^{(k)}(\dots)$ and $\texttt{UPDATE}^{(k)}(\dots)$ are arbitrarily differentiable functions ⇒ neural networks ⇒ implementate con NN semplici

Neural Message Passing Framework

Then for each $u \in V$:

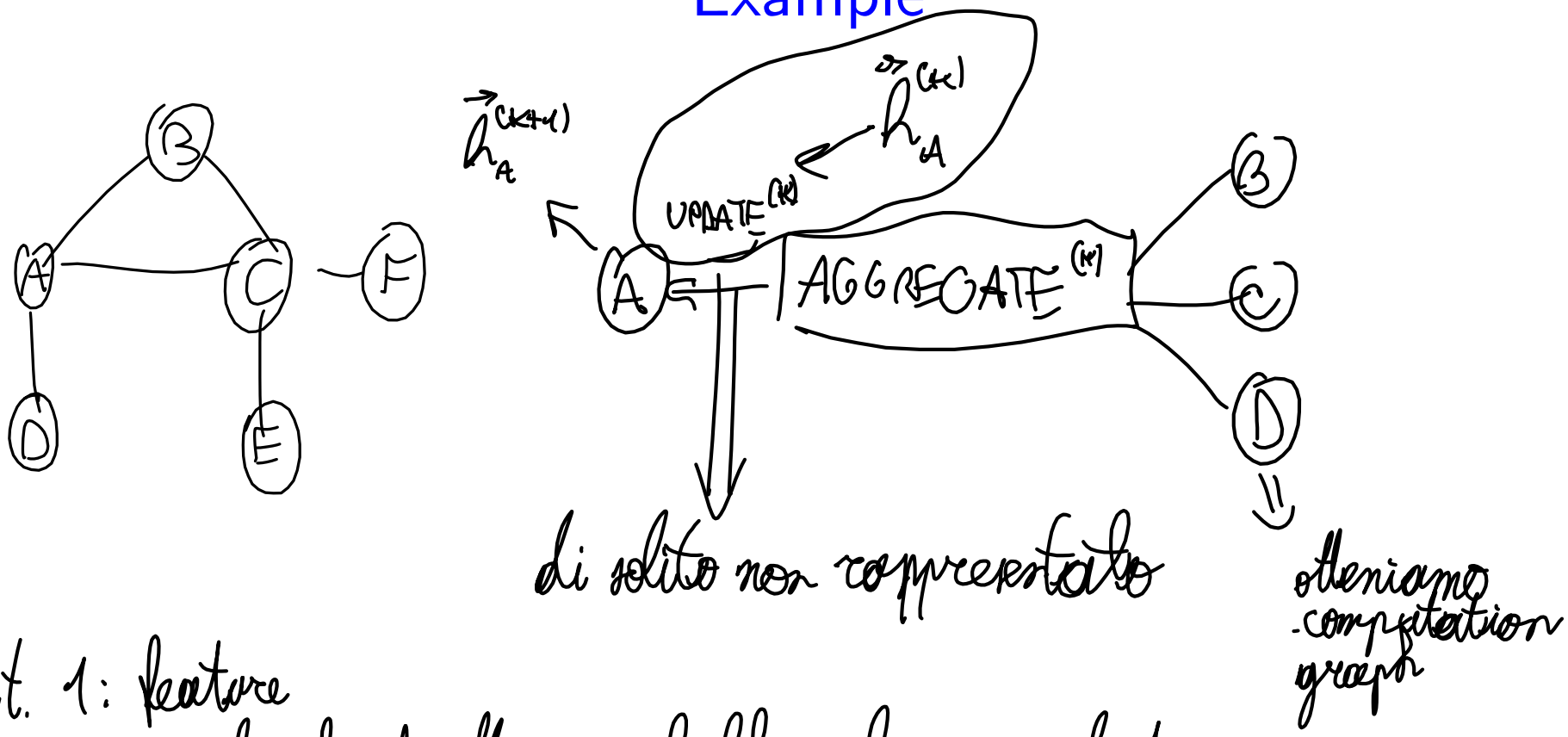
$$\mathbf{h}_u^{(k+1)} = \texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \texttt{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right)$$
$$= \texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right)$$

The embedding $\mathbf{z}_u$ of a node $u \in V$ is the embedding after K iterations:

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}$$

Note: the iterations of message passing are also called *layers* of the GNN

Example



di solito non rappresentato

otteniamo computation graph

it. 1: feature

// 2: guardando A, ottengo embedding dei vicini di A e rappr. di A da it. 1

funzioni sempre uguali all'interno di it., possono essere diverse per it. diverse

con K it., $\vec{h}_v^{(K)}$ dipende da feature di nodi a distanza $\leq K$

Example

computation graph diverso $\forall$ nodo, ma UPDATE e AGGREGATE condivise da tutti nodi

In generale, comp. graph = NNs con parametri condivisi $\Rightarrow$ riduce # parametri $\Rightarrow$ riduce complessità modello

Example

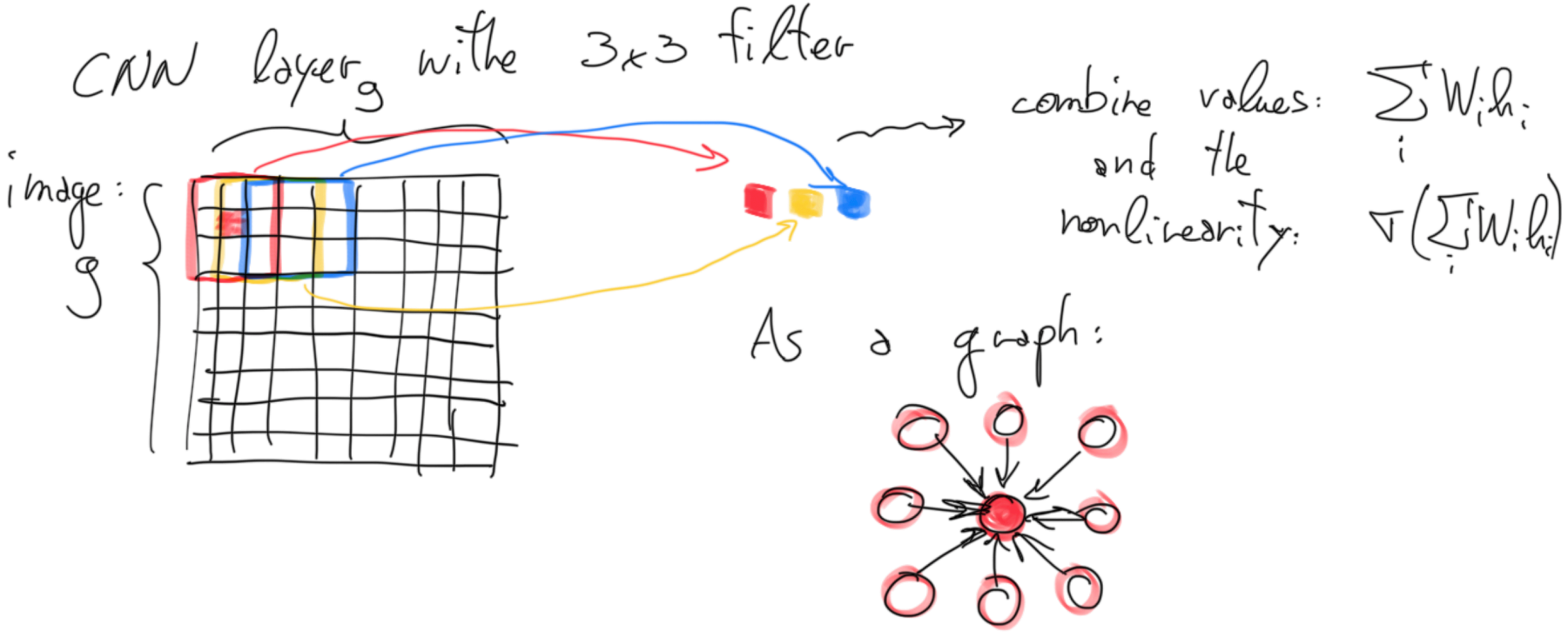
Neural Message Passing Framework

Question: are GNNs (as generally defined in the previous slide) sensible to permutations of the nodes?

The `AGGREGATE` function works on sets, so if we consider "proper" `AGGREGATE` functions (i.e., the order of the input elements does not matter) and "proper" `UPDATE` functions $\Rightarrow$ the resulting GNN is permutation invariant

Neural Message Passing: Motivation

Intuition: the local feature-aggregation behaviour of GNNs is analogous to the behavior of the convolutional filters in CNNs



Basic GNNs

The most basic version of a GNN is given by:

$$\texttt{AGGREGATE}^{(k)}\left(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\}\right) = \mathbf{m}_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)}$$

Therefore:

$$\begin{aligned}\mathbf{h}_u^{(k+1)} &= \texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right) \\ &= \sigma\left(\mathbf{W}_{\text{self}}^{(k+1)}\mathbf{h}_u^{(k)} + \mathbf{W}_{\text{neigh}}^{(k+1)}\mathbf{m}_{\mathcal{N}(u)}^{(k)} + \mathbf{b}^{(k+1)}\right)\end{aligned}$$

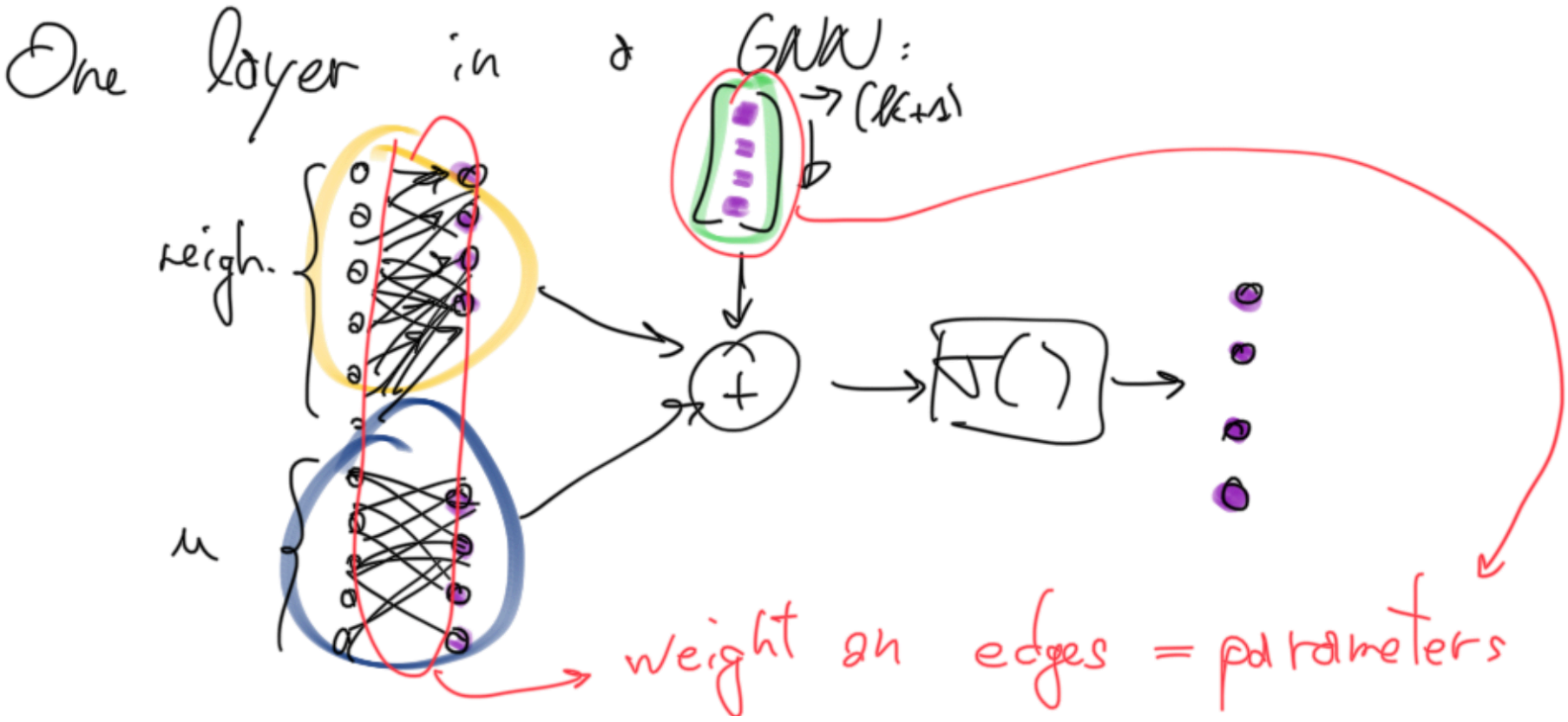
Notes

- $\mathbf{W}_{\text{self}}^{(k+1)}$ and $\mathbf{W}_{\text{neigh}}^{(k+1)}$ are trainable parameters, with $\mathbf{W}_{\text{self}}^{(k+1)}, \mathbf{W}_{\text{neigh}}^{(k+1)} \in \mathbb{R}^{d^{(k+1)} \times d^{(k)}}$
- $\mathbf{b}^{(k+1)}$ is the *bias*; often omitted in the notation
- $\sigma()$ is an *elementwise* non-linear function (e.g., ReLU)

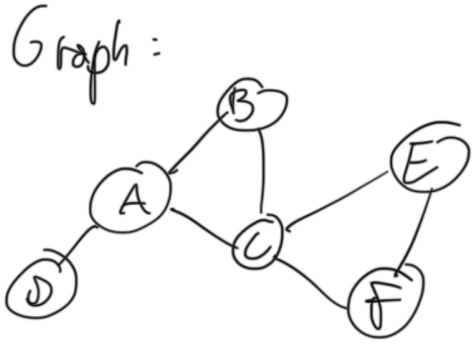
Basic GNNs (continue)

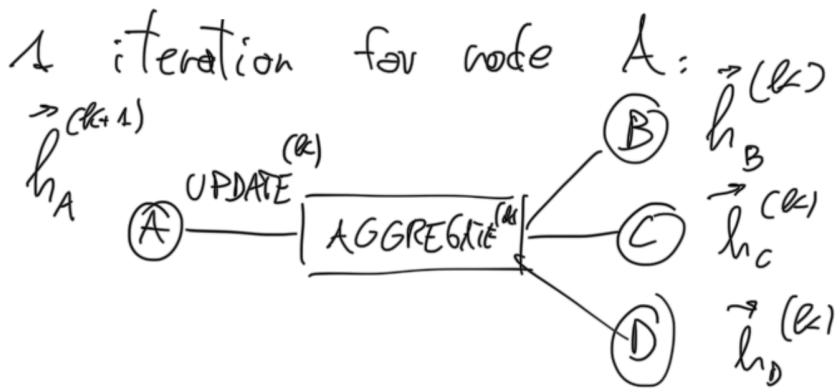
Putting all together, for each node u, in each iteration:

$$\mathbf{h}_u^{(k+1)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k+1)}\mathbf{h}_u^{(k)} + \mathbf{W}_{\text{neigh}}^{(k+1)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)} + \mathbf{b}^{(k+1)}\right)$$



Example



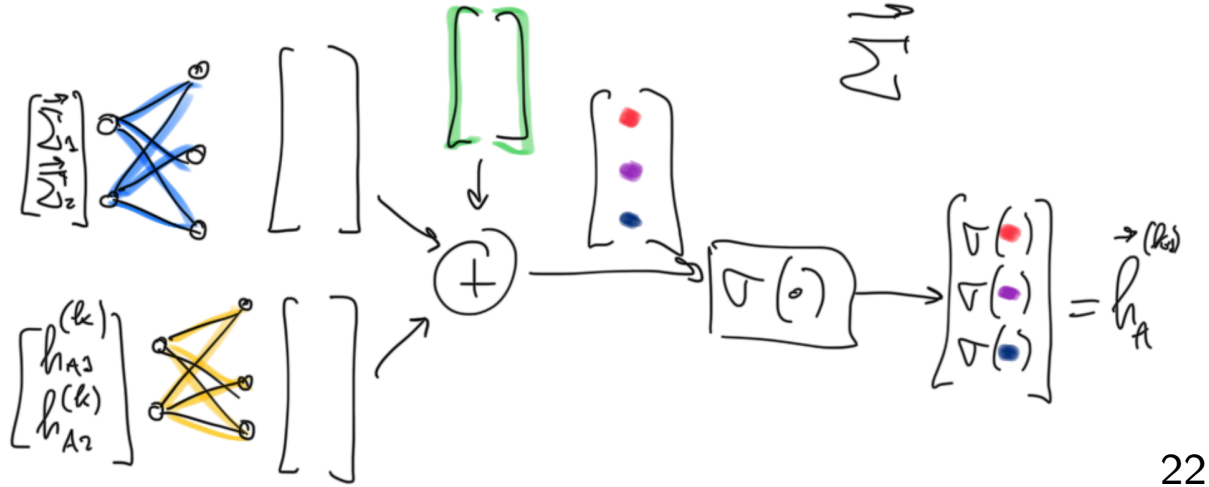


$$d^{(k)} = 2$$

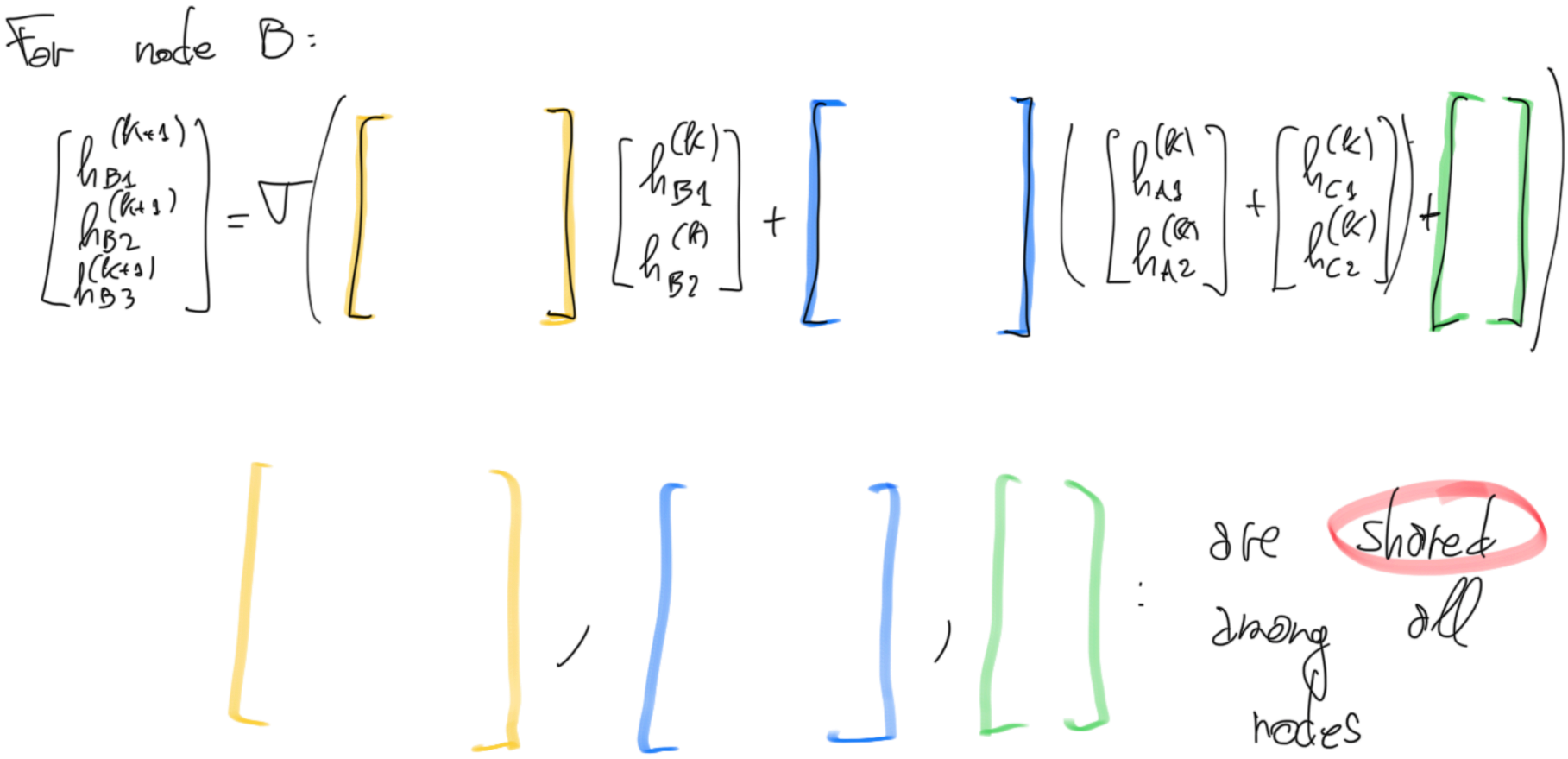
$$d^{(k+1)} = 3$$

$$\begin{bmatrix} h_{A1}^{(k+1)} \\ h_{A2}^{(k+1)} \\ h_{A3}^{(k+1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} W_{s11} & W_{s12} \\ W_{s21} & W_{s22} \\ W_{s31} & W_{s32} \end{bmatrix} \begin{bmatrix} h_{A1}^{(k)} \\ h_{A2}^{(k)} \end{bmatrix} + \begin{bmatrix} W_{n11} & W_{n12} \\ W_{n21} & W_{n22} \\ W_{n31} & W_{n32} \end{bmatrix} \left(\begin{bmatrix} h_{B1}^{(k)} \\ h_{B2}^{(k)} \end{bmatrix} + \begin{bmatrix} h_{C1}^{(k)} \\ h_{C2}^{(k)} \end{bmatrix} + \begin{bmatrix} h_{D1}^{(k)} \\ h_{D2}^{(k)} \end{bmatrix} \right) + \begin{bmatrix} b_1^{(k+1)} \\ b_2^{(k+1)} \\ b_3^{(k+1)} \end{bmatrix} \right)$$

Σ



Example



Basic GNNs: Parameter Sharing

The neural networks in different iterations can share parameters

$\mathbf{W}_{\text{self}}^{(k+1)}$, $\mathbf{W}_{\text{neigh}}^{(k+1)}$, $\mathbf{b}^{(k+1)}$ can be shared across the GNN iterations (layers).

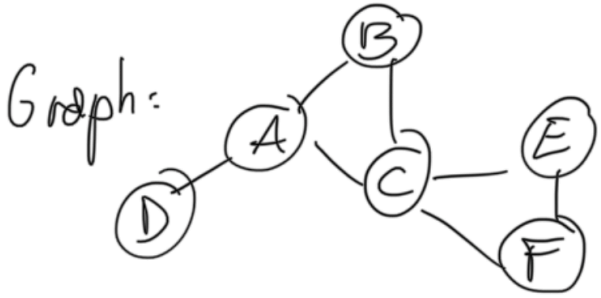
That is $\mathbf{W}_{\text{self}}^{(k+1)} = \mathbf{W}_{\text{self}}$, $\mathbf{W}_{\text{neigh}}^{(k+1)} = \mathbf{W}_{\text{neigh}}$ and $\mathbf{b}^{(k+1)} = \mathbf{b}$ for all $k \in \{0, \ldots, K-1\}$

In this case:

$$\mathbf{h}_u^{(k+1)} = \sigma\left(\mathbf{W}_{\text{self}}\mathbf{h}_u^{(k)} + \mathbf{W}_{\text{neigh}} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)} + \mathbf{b}\right)$$

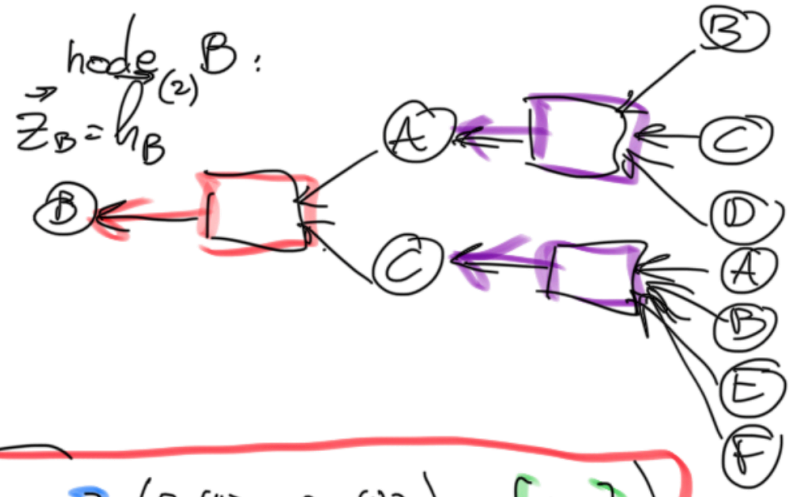
Note: this implies that $d^{(k)} = d \quad \forall k = 1, \ldots, K$

Example



$K = 2$

$d = 3$



$$\begin{bmatrix} h_{B1}^{(2)} \\ h_{B2}^{(2)} \\ h_{B3}^{(2)} \end{bmatrix} = \sigma\left(\begin{bmatrix} W_{S11} & W_{S12} & W_{S13} \\ W_{S21} & W_{S22} & W_{S23} \\ W_{S31} & W_{S32} & W_{S33} \end{bmatrix} \begin{bmatrix} h_{B1}^{(1)} \\ h_{B2}^{(1)} \\ h_{B3}^{(1)} \end{bmatrix} + \overbrace{\left[\quad \right]}^{3} \left(\begin{bmatrix} h_{A1}^{(1)} \\ h_{A2}^{(1)} \\ h_{A3}^{(1)} \end{bmatrix} + \begin{bmatrix} h_{C1}^{(1)} \\ h_{C2}^{(1)} \\ h_{C3}^{(1)} \end{bmatrix} \right) + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

$$\begin{bmatrix} h_{A1}^{(1)} \\ h_{A2}^{(1)} \\ h_{A3}^{(1)} \end{bmatrix} = \sigma\left(\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix} \begin{bmatrix} h_{A1}^{(0)} \\ h_{A2}^{(0)} \\ h_{A3}^{(0)} \end{bmatrix} + \begin{bmatrix} & & \\ & & \\ & & \end{bmatrix} \left(\vec{h}_B^{(0)} + \vec{h}_C^{(0)} + \vec{h}_D^{(0)} \right) + \begin{bmatrix} \\ \\ \end{bmatrix} \right)$$

Graph-Level Equations

Many GNNs can be succinctly defined using *graph-level* equations.

For the basic GNN, they are:

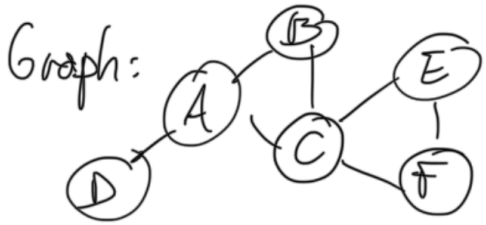
$$\mathbf{H}^{(k)} = \sigma\left(\mathbf{A}\mathbf{H}^{k-1}\mathbf{W}_{\text{neigh}} + \mathbf{H}^{k-1}\mathbf{W}_{\text{self}} + \mathbf{B}\right)$$

where:

- $\mathbf{A}$ is the adjacency matrix of G
- $\mathbf{H}^k \in \mathbb{R}^{|V| \times d^{(k)}}$ is the matrix of node representations at layer k of the GNN: each row is the representation of a node at layer k
- $\mathbf{B} \in \mathbb{R}^{|V| \times d^{(k)}}$ is the *bias* matrix, where each row is equal to the bias vector $\mathbf{b}$

Note: the GNN can be efficiently implemented with a small number of *sparse* matrix operations.

Example



$$H^{(k)} = \begin{bmatrix} \vec{h}_A^{(k)} \\ \vec{h}_B^{(k)} \\ \vec{h}_C^{(k)} \\ \vec{h}_D^{(k)} \\ \vec{h}_E^{(k)} \\ \vec{h}_F^{(k)} \end{bmatrix} = \sigma\left(\begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{bmatrix} \vec{h}_A^{(k-1)} \\ \vec{h}_B^{(k-1)} \\ \vec{h}_C^{(k-1)} \\ \vec{h}_D^{(k-1)} \\ \vec{h}_E^{(k-1)} \\ \vec{h}_F^{(k-1)} \end{bmatrix} \begin{bmatrix} W_{neigh} \end{bmatrix} + \right.$$

$$\left. + \begin{bmatrix} \vec{h}_A^{(k-1)} \\ \vec{h}_B^{(k-1)} \\ \vec{h}_C^{(k-1)} \\ \vec{h}_D^{(k-1)} \\ \vec{h}_E^{(k-1)} \\ \vec{h}_F^{(k-1)} \end{bmatrix} \begin{bmatrix} W_{self} \end{bmatrix} + \begin{bmatrix} \vec{b} \\ \vec{b} \\ \vec{b} \\ \vec{b} \\ \vec{b} \\ \vec{b} \end{bmatrix} \right)$$

first row

$$\begin{bmatrix} \vec{h}_A^{(k)} \end{bmatrix} = \sigma\left(\begin{bmatrix} \vec{h}_B^{(k-1)} + \vec{h}_C^{(k-1)} + \vec{h}_D^{(k-1)} \end{bmatrix} \begin{bmatrix} W_{neigh} \end{bmatrix} + \begin{bmatrix} \vec{h}_A^{(k-1)} \end{bmatrix} \begin{bmatrix} W_{self} \end{bmatrix} + \begin{bmatrix} \vec{b} \end{bmatrix} \right)$$

Message Passing with Self-loops

Simplification of neural message passing: omit the *update* operator by adding self loops to the graph.

With this modification, each iteration of a general GNN is:

$$\mathbf{h}_u^{(k+1)} = \texttt{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u) \cup \{u\}\})$$

Message Passing with Self-loops

Simplification of neural message passing: omit the *update* operator by adding self loops to the graph.

With this modification, each iteration of a general GNN is:

$$\mathbf{h}_u^{(k+1)} = \texttt{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u) \cup \{u\}\})$$

This is a simplification of a general GNN

- pro: reduces overfitting
- con: limits the *expressivity* of the GNN

Message Passing with Self-loops (continue)

For the basic GNN: adding self-loops means sharing parameters between $\mathbf{W}_{\text{self}}$, $\mathbf{W}_{\text{neigh}}$, that is:

$$\mathbf{W}_{\text{self}} = \mathbf{W}_{\text{neigh}} = \mathbf{W}$$

Example

Generalized Neighborhood Aggregation

The basic GNN can be improved and generalized in several ways.

We now see how to generalize the AGGREGATE operator.

For each $u \in V$:

$$\mathbf{h}_u^{(k+1)} = \texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \texttt{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right)$$
$$= \texttt{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right)$$

Neighborhood Normalization

Issue of Basic GNN: it is sensitive to node degrees.

Neighborhood Normalization

Issue of Basic GNN: it is sensitive to node degrees.

Solution: normalize the `AGGREGATE` operator taking into account the degree.

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \frac{\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k)}}{|\mathcal{N}(u)|}$$

Better solution in practice:

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v^{(k)}}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}$$

Graph convolutional networks (GCNs)

Introduced by Kipf and Welling, *Semi-supervised classification with graph convolutional networks*, ICLR 2016.

Uses the neighborhood normalization with self-loops.

For each $u \in V$:

GraphSAGE

Introduced by W.L. Hamilton, R. Ying, and J. Leskovec. *Inductive representation learning on large graphs*, NeurIPS (2017).

Introduced the generalized aggregation AGGREGATE operator.

Main differences with GCNs:

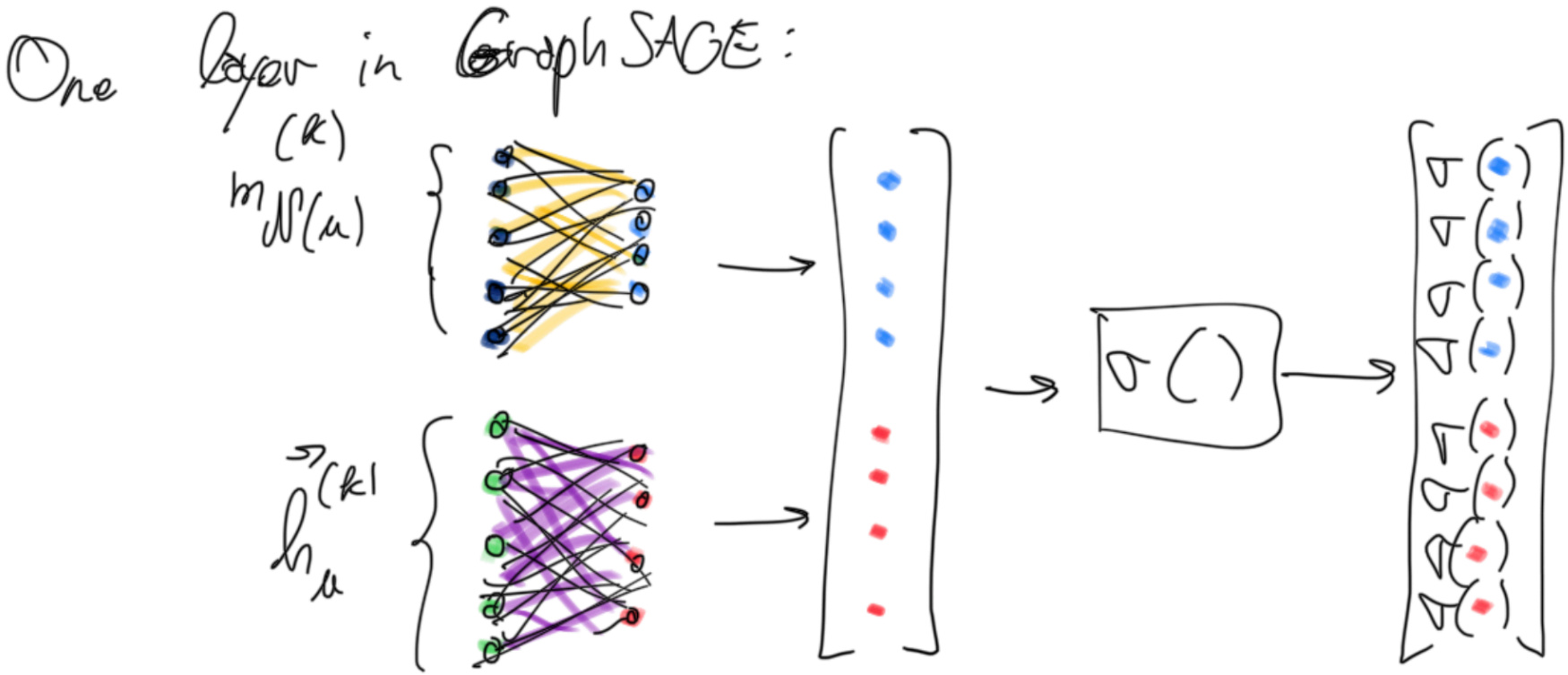
- use of generalized AGGREGATE operator
- for each node u, a transformation of $\mathbf{h}_u^{(k)}$ is *concatenated* with the transformation of $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ (before the elementwise nonlinear operator is applied)

GraphSAGE (continue)

For each node $u \in V$:

$$\mathbf{h}_u^{(k+1)} = \sigma\left(\left[\mathbf{W}^{(k+1)}\mathbf{m}_{\mathcal{N}(u)}^{(k)}, \mathbf{B}^{(k+1)}\mathbf{h}_u^{(k)}\right]\right)$$

where $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \texttt{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u) \cup \{u\}\})$



GraphSAGE (continue)

For each node $u \in V$:

$$\mathbf{h}_u^{(k+1)} = \sigma\left(\left[\mathbf{W}^{(k+1)}\mathbf{m}_{\mathcal{N}(u)}^{(k)}, \mathbf{B}^{(k+1)}\mathbf{h}_u^{(k)}\right]\right)$$

where $\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \texttt{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u) \cup \{u\}\})$

Common choices for `AGGREGATE`:

- mean of $\mathbf{h}_v^{(k)}$:

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v^{(k)}}{|\mathcal{N}(u)|}$$

- first apply a simple NN to each $\mathbf{h}_v^{(k)}$ (with learnable parameters θ), and then apply an elementwise operator γ (e.g., mean/max):

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \gamma\left(\left\{NN_\theta(\mathbf{h}_v^{(k)}), \forall u \in V\right\}\right)$$

- more complex functions (e.g., LSTM)

Graph Attention Network (GAT)

Introduced by P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. *Graph attention networks*, ICLR (2018).

Neighborhood attention: assign an *attention* weight to each neighbor, which is used to weigh this neighbor's influence during aggregation

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v^{(k)}$$

$\alpha_{u,v}$: attention on node $v \in \mathcal{N}(u)$ when aggregating information at node u

The attention weights $\alpha_{u,v}$ are learned from data.

GAT: Attention Weights

$$\mathbf{m}_{\mathcal{N}(u)}^{(k)} = \sum_{v \in \mathcal{N}(u)} \alpha_{u,v} \mathbf{h}_v^{(k)}$$

Various definitions of $\alpha_{u,v}$ have been used. The original one is:

$$\alpha_{u,v} = \frac{e^{\mathbf{a}^T [\mathbf{W}\mathbf{h}_u^{(k)}, \mathbf{W}\mathbf{h}_v^{(k)}]}}{\sum_{v \in \mathcal{N}(u)} e^{\mathbf{a}^T [\mathbf{W}\mathbf{h}_u^{(k)}, \mathbf{W}\mathbf{h}_v^{(k)}]}}$$

with

- $[\mathbf{W}\mathbf{h}_u^{(k)}, \mathbf{W}\mathbf{h}_v^{(k)}]$: concatenation of $\mathbf{W}\mathbf{h}_u^{(k)}$ and $\mathbf{W}\mathbf{h}_v^{(k)}$
- $\mathbf{W}$ is a matrix of parameters
- $\mathbf{a}^T$ is an *attention* vector

Generalized Update Operators

The UPDATE operator can be generalized as well (see the book if interested).
Why is this useful?

K. Xu, et al. *Representation learning on graphs with jumping knowledge networks*, ICML (2018) proved a result about *over-smoothing*, which we describe with the informal version below,

Generalized Update Operators

The UPDATE operator can be generalized as well (see the book if interested).
Why is this useful?

K. Xu, et al. *Representation learning on graphs with jumping knowledge networks*, ICML (2018) proved a result about *over-smoothing*, which we describe with the informal version below,

Xu et al. consider *GCN-style networks*:

- AGGREGATE is essentially the mean
- UPDATE is done with self-loops

Informal version: when using a GCN-style network with K layers, the *influence* of node u on node v is proportional the probability $\Pr_k[v|u]$ of reaching node v on a random walk with K steps starting from node u.

GNNs: Graph Level

What about using GNNs to learn the embedding $\mathbf{z}_G$ of a graph G?

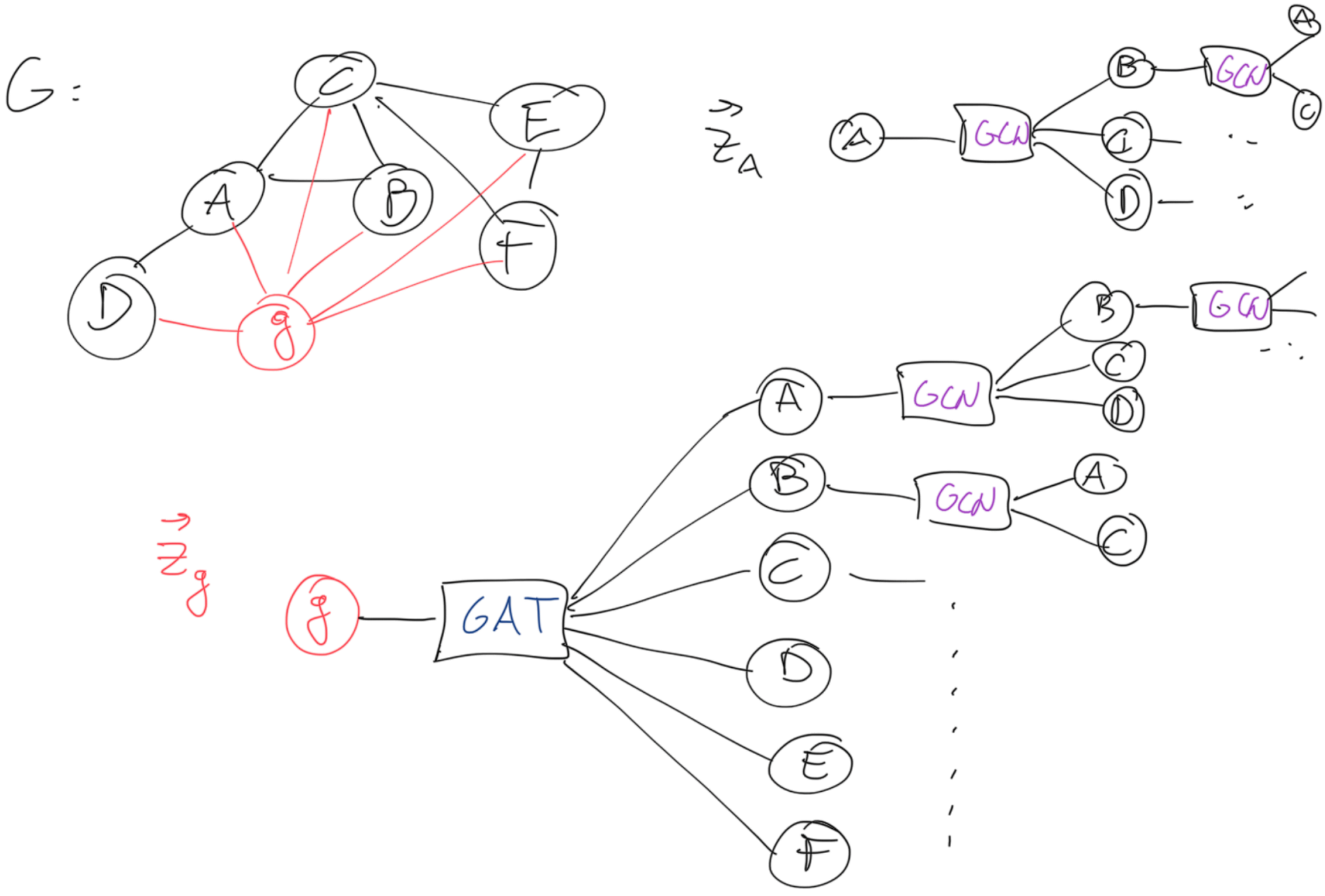
Approaches seen for graph embeddings usually work well, at least for small graphs:

- $\mathbf{z}_G = \sum_{u \in V} \mathbf{z}_u$
- add a node g connected to all $u \in V$, then $\mathbf{z}_G = \mathbf{z}_g$

More complex approaches recently explored (e.g., use LSTM).

Note: the second approach above can be made more flexible by using a specific neural network for g

GNNs: Graph Level (continue)

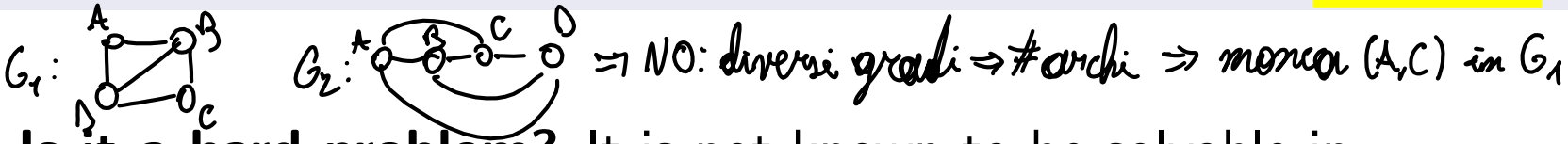


GNNs: a Theoretical Motivation

We will now see a theoretical motivation for GNNs, by comparing them with an algorithm for the *graph isomorphism problem*.

Definition

Given two graphs G_1 and G_2, the *graph isomorphism problem* requires to determine whether G_1 is isomorphic to G_2: $G_1 \simeq G_2$



Is it a hard problem? It is not known to be solvable in polynomial time nor to be NP-complete.

Note: ideally we would like a graph embedding technique to *solve* the graph isomorphism problem, that is: $\mathbf{z}_{G_1} = \mathbf{z}_{G_2}$ if and only if $G_1 \simeq G_2$

Weisfieler-Leman (WL) Algorithm

Algorithm for the graph isomorphism problem.

Note: does not always produce the correct answer!

- if WL outputs that $G_1 \not\simeq G_2$ then $G_1 \not\simeq G_2$
- if WL outputs that $G_1 \simeq G_2$, then it may be that $G_1 \not\simeq G_2$

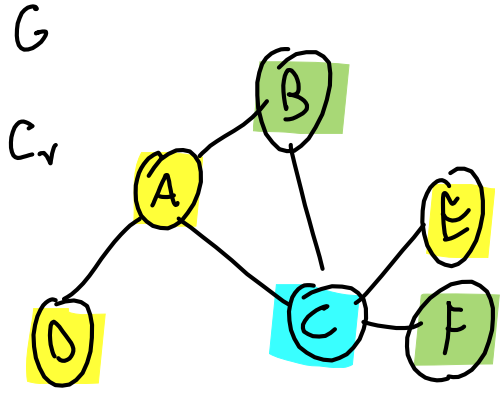
It is known that WL produces the correct answer for a large class of graphs.

WL Algorithm

Builds on a *color* refinement algorithm for the vertices of a graph G.

no random color

Given a coloring C_V of the vertices V of $G = (V, E)$, let $P(C_V)$ the partition of the vertices V defined by the coloring C_V: two vertices u, v are in the same set of the partition if and only if u and v have the same color.



$P(C_V) = \{A, D, E\}, \{C\}, \{B, F\}$

WL Algorithm (continue)

Algorithm `ColorRefinement`(G)
Input: graph $G = (V, E)$
Output: coloring of V

"color": hash value basato su hash values di vicini ⇒ calcolato efficientemente

$C_{curr} \leftarrow$ assign the same color u to all nodes $u \in V$;
repeat
  $C_{prev} \leftarrow C_{curr}$;
  $C_{curr} \leftarrow$ for each pair u and v where u and v have the same color in C_{prev}: assign different colors to u and v if and only if there is some color c such that u and v have different number of neighbours of color c in C_{prev};
until $P(C_{prev}) = P(C_{curr})$;
return C_{curr};

Analysis:

- `ColorRefinement`(G) stops after at most $|V|$ iterations;
- each iteration requires $O(|V|^2)$ operations
- the complexity of `ColorRefinement`(G) is $O(|V|^3)$

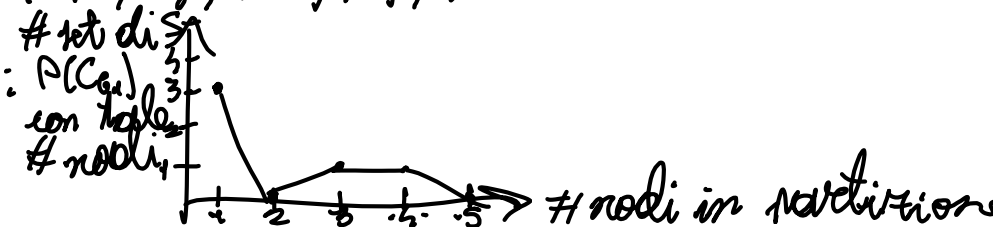
WL Algorithm (continue)

Algorithm $\texttt{WL}(G_1, G_2)$
Input: graphs $G_1 = (V_1, E_1)$, $G_2 = (V_2, E_2)$
Output: yes/no

$C_{G_1} \leftarrow \texttt{ColorRefinement}(G_1)$;
$C_{G_2} \leftarrow \texttt{ColorRefinement}(G_2)$;
return $\mathit{histogram}(P(C_{G_1})) = \mathit{histogram}(P(C_{G_2}))$;

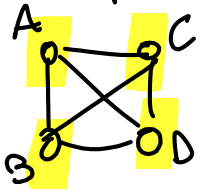
Analysis: the complexity of $\texttt{WL}(G_1, G_2)$ is $O\left(|V_1|^3 + |V_2|^3\right)$

G_1: $V = \{1, \ldots, 10\}$ ⇒ dopo colorRefinement(G_1), ho coloring C_{G_1} con
$P(C_{G_1}) = \{1, 3, 5, 10\}, \{7, 8, 9\}, \{2\}, \{4\}, \{6\}$ ⇒
⇒ histogram($P(C_{G_1})$):



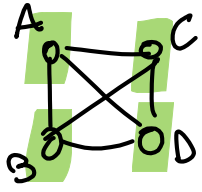
Example

Color refinement (G_1):

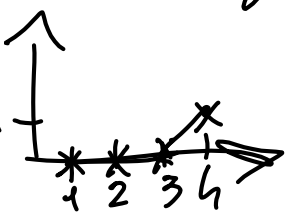


$P = \{A, B, C, D\}$

B: 3 vicini gialli, A: 3, C: 3, D: 3 $\Rightarrow$ coloring non cambia

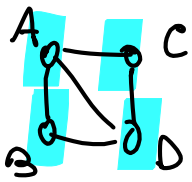


$P = \{A, B, C, D\} \Rightarrow C_{G_1}$:



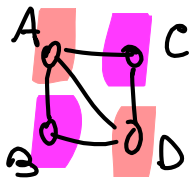
Example

ColorRefinement(G_2):



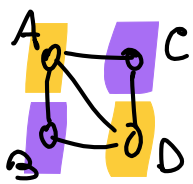
$P = \{A, B, C, D\} \Rightarrow$ A: 3 vicini blu, B: 2, C: 2, D: 3

it 2:

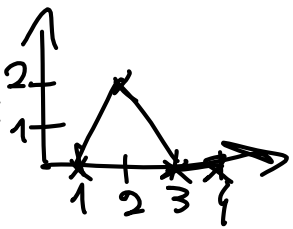


$P = \{A, D\}, \{B, C\} \Rightarrow$ A: 2,1 B: 0,2 C: 0,2 D: 2,1

it. 3:



$P = \{A, D\}, \{B, C\} \Rightarrow C_{G_2}$:



output: $G_1 \neq G_2 \Rightarrow$ sorretto

Example

WL "no" con $G_1 \simeq G_2$: impossibile

WL "yes" con $G_1 \not\simeq G_2$: possibile

WL algorithm and GNNs

Theorem

Consider a GNN with K message passing layers of the following form:

$$\mathbf{h}_u^{(k+1)} = \mathit{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathit{AGGREGATE}^{(k)}(\{\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\})\right)$$

where $\mathit{AGGREGATE}$ is differentiable and permutation invariant and UPDATE is differentiable. Assume that the input is made of discrete features: $\mathbf{h}_u^{(0)} = \mathbf{x}_u \in \mathbb{Z}^d, \forall u \in V$.
Then $\mathbf{h}_u^{(K)} \neq \mathbf{h}_v^{(K)}$ only if u and v have different labels after K iterations of the WL algorithm.

Informally: GNNs are *no more powerful than* the WL algorithm when we have discrete information as node features.

The result generalizes to graphs: if the WL algorithm does not distinguish (non-isomorphic) graphs G_1 and G_2, then any GNN (with the form as above) is incapable of distinguishing G_1 and G_2.

WL algorithm and GNNs (continue)

Theorem

There exists a GNN with the form defined in the previous theorem such that $\mathbf{h}_u^{(K)} = \mathbf{h}_v^{(K)}$ *only if* u *and* v *have the same labels after* K *iterations of the WL algorithm.*

Informally: there are GNNs that are *as powerful as* the WL algorithm.

Which GNNs are as powerful as the WL algorithm?

- the basic GNN? YES!
- basic GNN with neighborhood normalization? NO!
- GCN? NO!
- GraphSAGE? depends on the `AGGREGATION` operator

Graph Isomorphism Networks (GINs)

Xu, Keyulu, et al. "How powerful are graph neural networks?" *ICLR 2019*.
Update of a node:

$$\mathbf{h}_u^{(k+1)} = \texttt{MLP}^{(k)}\left(\left(1+\varepsilon^{(k)}\right)\mathbf{h}_u^{(k)} + \sum_{v\in\mathcal{N}(u)} \mathbf{h}_v^{(k)}\right)$$

where $\varepsilon^{(k)}$ is learnable parameter.

GIN has the maximal representational power for a (message-passing) GNN: if two graphs can be distinguished by GIN, they can be also distinguished by the WL algorithm, and vice versa

GNNs for Node Embeddings

In general: GNNs can be used to obtain node embeddings.

In terms of the encoder-decoder framework:

- *encoder*: GNN
- *similarity function*: see the ones for shallow embeddings
- *decoder*: see the ones for shallow embeddings
- *loss*: see the ones for shallow embeddings

task-dependent GNN

However: GNNs can also be used for *supervised* tasks.

GNNs for Supervised-Tasks

Common supervised tasks for graphs:

- node classification
- graph classification

GNNs: Node Classification

Input: some nodes have a label and can be used to train the GNN.

Each node u in the training set has a label, encode by a $0-1$ vector $\mathbf{y}_u$ of dimension c.

Loss function: negative log-likelihood loss of softmax classification function

Given the embedding $\mathbf{z}_u$ of node u and the corresponding vector $\mathbf{y}_u$, the softmax classification function is

$$\text{softmax}(\mathbf{z}_u, \mathbf{y}_u) = \sum_{i=1}^{c} \mathbf{y}_u[i] \frac{e^{\mathbf{z}_u^T \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_u^T \mathbf{w}_j}}$$

linear model per label i

input: embedding z_u

unico termine, di label corretta

with $\mathbf{w}_i$ for $i = 1, \ldots, c$ trainable parameters.

GNNs: Node Classification (continue)

Then the loss function is:

$$\mathcal{L} = \sum_{u \in V_{\text{train}}} -\log\left(\text{softmax}(\mathbf{z}_u, \mathbf{y}_u)\right)$$

where V_{train} is the set of *training nodes*.

Note: there are different types of nodes for node classification

- the set V_{train} of *training nodes*: both the nodes and the labels are used during training (i.e., to learn the GNN parameters)
- the set V_{trans} of transductive test nodes: the nodes, but not their labels, are used during training. These nodes do not (directly) contribute to $\mathcal{L}$ ⇒ non considerarli potrebbe cambiare altri embedding
- the set V_{ind} of inductive test nodes: the nodes are completely *unobserved* during training

GNNs: Graph Classification

Input: set of graphs, where each graph G has a label represented by a $0-1$ vector $\mathbf{y}_G$ of dimension c

Loss: same as for node classification, where the embedding $\mathbf{z}_G$ is computed for a graph G.

$$\text{softmax}(\mathbf{z}_G, \mathbf{y}_G) = \sum_{i=1}^{c} \mathbf{y}_G[i] \frac{e^{\mathbf{z}_G^T \mathbf{w}_i}}{\sum_{j=1}^{c} e^{\mathbf{z}_G^T \mathbf{w}_j}}$$

$$\mathcal{L} = \sum_{G \in \mathcal{G}_{\text{train}}} -\log\left(\text{softmax}(\mathbf{z}_G, \mathbf{y}_G)\right)$$

where $\mathcal{G}_{\text{train}}$ is the set of graphs used for training.

GNNs: other tasks

GNNs can be used for several other tasks

- edge prediction: predict *missing edges*
- regression task for nodes
- regression task for graphs

GNNs: Additional Notes

What algorithm is used to learn GNN parameters? **SGD**

Training all nodes of a GNN simultaneously can require a lot of resources (time, memory) $\Rightarrow$ **Mini-batching:** obtain the final representation z_u for a small set of nodes at the time

Problem: how do we make sure that the computation graph is connected but not all nodes are used? $\Rightarrow$ Start from the target nodes and use **subsampling**: sample a fixed number of neighbours for each node.

The loss function can be combined with *regularization* (e.g., ℓ_2 regularization)

GNNs: Frameworks

If you are starting to use GNNs, here are some pointers to GNN Python libraries:

PyTorch Geometric
https://pytorch-geometric.readthedocs.io/en/latest/

Deep Graph Library https://www.dgl.ai/

Graph Nets https://github.com/deepmind/graph_nets

Spektral https://graphneural.network/