A Hands-on Introduction to Graph Deep Learning, with Examples in PyTorch Geometric - I

Machine Learning and Dynamical Systems Seminar

November 2, 2023

Gabriele Santin (gabrielesantin.github.io)
Antonio Longa (antoniolonga.github.io)
Steve Azzolin (steveazzolin.github.io)
Francesco Ferrini (francescoferrini.github.io)

Introduction
About us

Gabriele Santin
Assistant professor at University of Venice (Venice, Italy)
gabriele.santin@unive.it
https://gabrielesantin.github.io/

Antonio Longa
Assistant professor University of Trento (Trento, Italy)
antonio.longa@unitn.it
https://antoniolonga.github.io/

Steve Azzolin
ELLIS PhD Student at FBK and University of Trento (Trento, Italy)
steve.azzolin@unitn.it
https://steveazzolin.github.io/

Francesco Ferrini
PhD Student at University of Trento, (Trento, Italy)
francesco.ferrini@unitn.it
https://francescoferrini.github.io/

Introduction
Organization and material

Tutorial in four parts (slides + Jupyter notebooks available):

- **Part I:** November 2, Presenter: **GS**
  Goals: Motivations, Intro of basic concepts, definition of GNNs
- **Part II:** November 9, Presenter: **AL**
  Goals: Implementation of GNNs: How to implement a full GNN pipeline in PyTorch Geometric.
- **Part III:** November 16, Presenter: **SA**
  Goals: Explainability of GNNs: How to inspect a model to try to understand the learned decision pattern.
- **Part IV:** November 23, Presenter: **FF**
  Goals: Heterogeneity in GNNs: How can GNNs effectively model and incorporate a diversity of nodes and edges with different types.

Introduction
Why PyTorch Geometric

PyG *(PyTorch Geometric)* is a library built upon PyTorch to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data.

Where to get it:

GitHub repository: https://github.com/pyg-team/pytorch_geometric
Official page: https://pyg.org/

First paper: Matthias Fey and Jan E. Lenssen, *Fast Graph Representation Learning with PyTorch Geometric,* arXiv:1903.02428, 2019

Introduction
Why PyTorch Geometric

PyG *(PyTorch Geometric)* is a library built upon PyTorch to easily write and train Graph Neural Networks (GNNs) for a wide range of applications related to structured data.

Where to get it:

GitHub repository: https://github.com/pyg-team/pytorch_geometric
Official page: https://pyg.org/

First paper: Matthias Fey and Jan E. Lenssen, *Fast Graph Representation Learning with PyTorch Geometric,* arXiv:1903.02428, 2019

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

Introduction
Useful resources

Official documentation: https://pytorch-geometric.readthedocs.io/en/stable/
(including installation instructions)

Introduction
Useful resources

Official documentation: https://pytorch-geometric.readthedocs.io/en/stable/

External resources: https://pytorch-geometric.readthedocs.io/en/stable/notes/resources.html

Introduction
Useful resources

Official documentation: https://pytorch-geometric.readthedocs.io/en/stable/

External resources: https://pytorch-geometric.readthedocs.io/en/stable/notes/resources.html

Our **tutorials**:

- Pytorch Geometric tutorial
  https://antoniolonga.github.io/Pytorch_geometric_tutorials/index.html
- Advanced Pytorch Geometric tutorial
  https://antoniolonga.github.io/Advanced_PyG_tutorials/index.html

- Next session coming soon: check Steve Azzolin's page https://steveazzolin.github.io/

Introduction
Outline - Part I

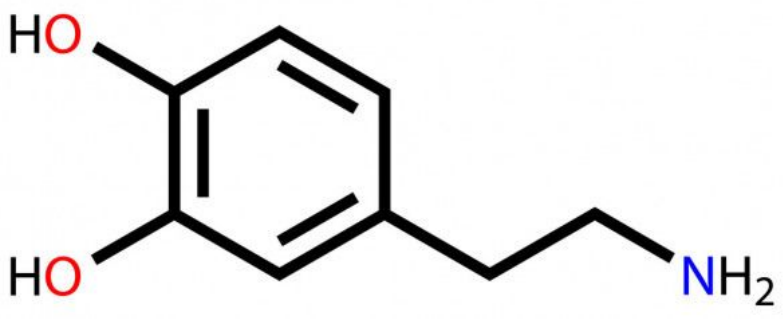
- Motivation
- Basic graph theory
- Graph Neural Networks
- Some computational aspects

Motivation
Applications of Graph Neural Networks

Modelling of relational data:

Biological / Chemical structures:
- Nodes are chemical elements
- Edges are chemical bonds

Example use case:
Classify a molecule's property based on its structure (graph classification/regression)



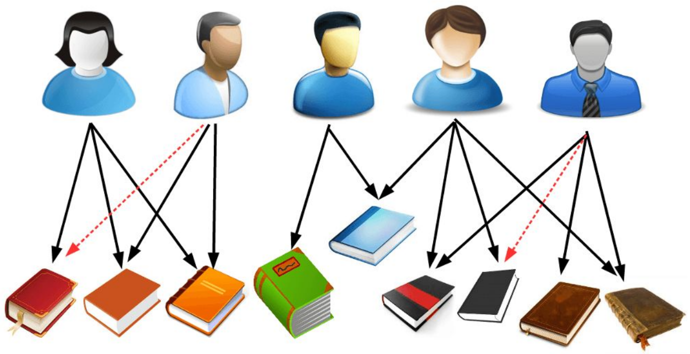
Motivation
Applications of Graph Neural Networks

Modelling of relational data:

User behavior
- Nodes are users or products
- Edges are users' preferences

Example use case:

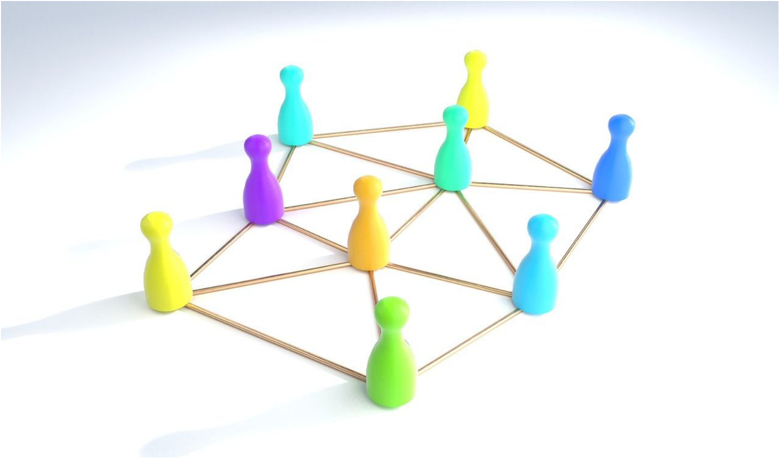
Recommender Systems: Predict a user's preference (edge prediction)



Motivation
Applications of Graph Neural Networks

Modelling of relational data:

Network science:
- Nodes are persons
- Edges are interactions or contacts



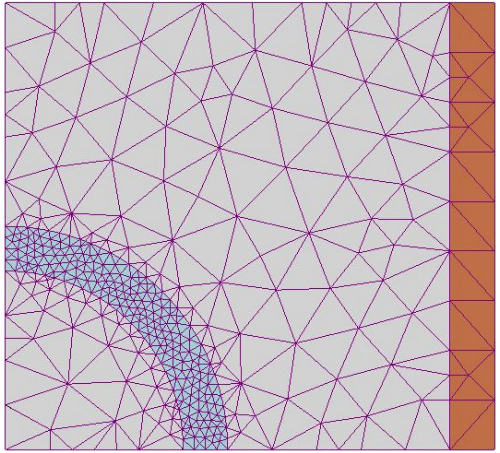
Example use case:

Epidemic modelling: Classify a person's infection status based on its contact network (node classification)

Motivation
Opportunities for GNNs in Computational Sciences

Many **problems** come naturally in **graph** form:
- Meshes (Finite Elements Method (FEM), …)
- Triangulated surfaces (e.g. for manifold discretization)
- …

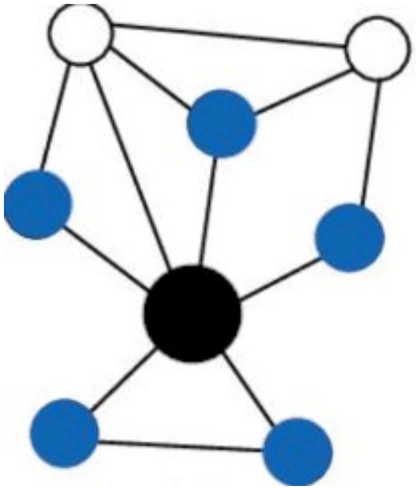
Typical **use cases** for Machine Learning:
- PDE/ODE solution methods (e.g. PINNs, DeepONets, …)
- Surrogates to approximate a forward map:
    - UQ
    - Inverse problems
    - Parameter optimization



Motivation
Opportunities for GNNs in Computational Sciences

Topology awareness
No need to vectorize, then process →Keep the **graph structure**

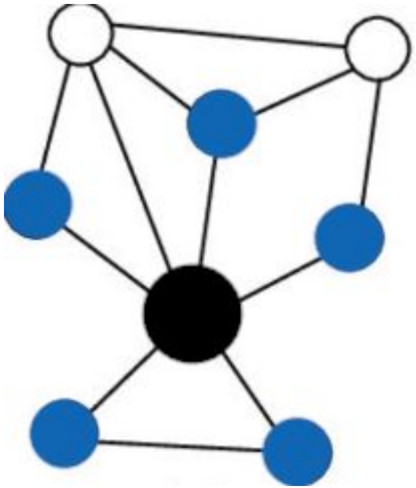
- Traditional NNs: The data points are (variable, output)
- GNNs: The data samples are (graph, output)



Motivation
Opportunities for GNNs in Computational Sciences

Topology awareness
No need to vectorize, then process →Keep the **graph structure**

- Traditional NNs: The data points are (variable, output)
- GNNs: The data samples are (graph, output)



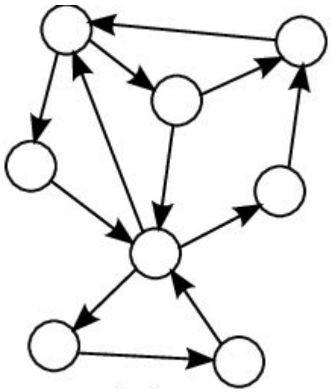
Inductive learning
GNNs work by neighbouring aggregation → **local**

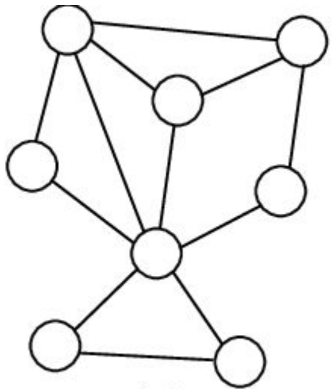
- Scalability (large graphs)
- Transferability across graphs (e.g. across scales, parameters, adaptive meshes, …)

Basic graph theory
Graphs

Graph G = (V, E)

- **V** : Set of n nodes
- **$E \subset V \times V$** : Set of m edges
    - Undirected: $(u,v) \in E$ implies $(v,u) \in E$

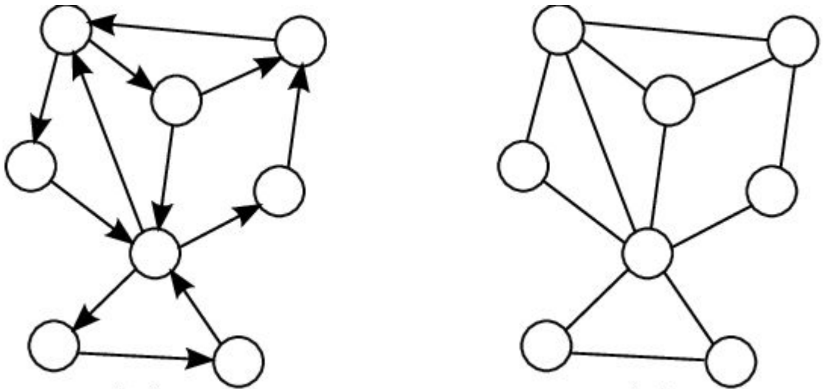




Basic graph theory
Graphs

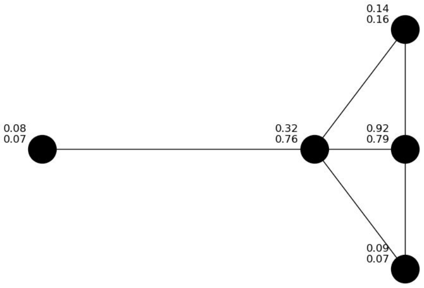
Graph G = (V, E)

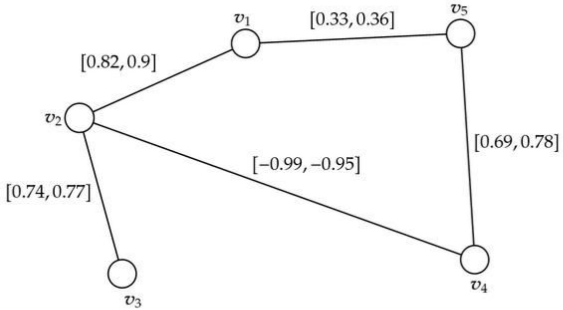
- **V** : Set of n nodes
- **E⊂V×V** : Set of m edges
    - Undirected: (u,v)∈E implies (v,u)∈E



Features:

- **Node features X^V**:
    - $n \times d_v$ dim. Matrix
    - E.g.: node pos. in a mesh
- **Edge features X^E**:
    - $m \times d_E$ dim. Matrix
    - E.g.: distance btw nodes

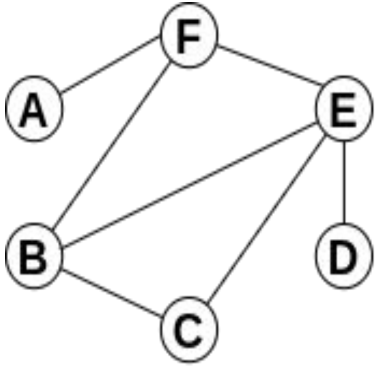




Basic graph theory
Neighbors, Adjacency, Laplacian

Useful notions given G = (V, E):

- **Node neighborhood**
  - $N^1(v) = \{u \in V: (u, v) \in E\}$
  - $N^k(v)$ similar notion, with path of length k



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

Basic graph theory
Neighbors, Adjacency, Laplacian

Fix an **enumeration** $V = \{v_1, \ldots, v_n\}$

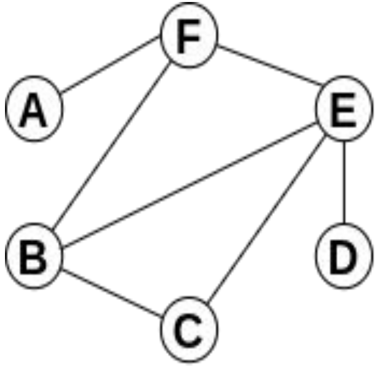
- **Adjacency matrix** $A = (a_{ij})_{i,j=1,\ldots,n}$:

  $n \times n$ matrix with $a_{ij} = 1$ iff $(v_i, v_j) \in E$

- **Degree matrix** $D = \mathrm{diag}(d_{11}, \ldots, d_{nn})$:

  $n \times n$ matrix with $d_{ii} = \deg(v_i)$

- **Laplacian matrix** $L = D - A$



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

Basic graph theory
Neighbors, Adjacency, Laplacian

Fix an **enumeration** $V = \{v_1, \ldots, v_n\}$

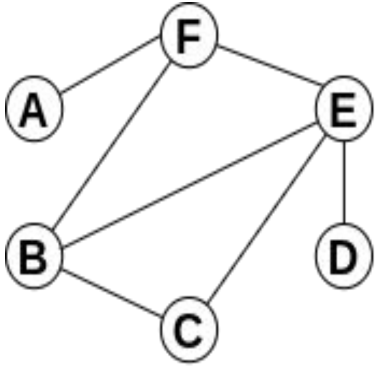
- **Adjacency matrix** $A = (a_{ij})_{i,j=1,\ldots,n}$:

  $n \times n$ matrix with $a_{ij} = 1$ iff $(v_i, v_j) \in E$

- **Degree matrix** $D = \mathrm{diag}(d_{11}, \ldots, d_{nn})$:

  $n \times n$ matrix with $d_{ii} = \deg(v_i)$

- **Laplacian matrix** $L = D - A$

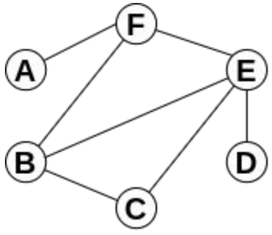


	A	B	C	D	E	F
A	0	0	0	0	0	1
B		0	1	0	1	1
C			0	0	1	0
D				0	1	0
E					0	1
F						0

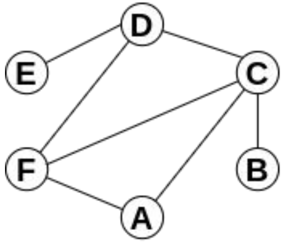
Vectorized representations of a graph $G = (V, E)$

Basic graph theory
Invariance and equivariance

Multiple **equivalent representations** of the same graph:
The representation depends on the **enumeration** of the nodes



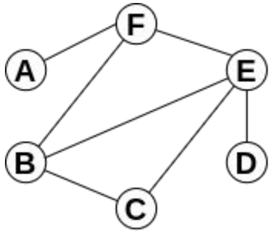
$$\rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 0 \end{bmatrix}$$



$$\rightarrow \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

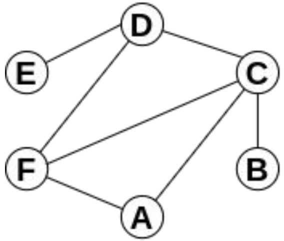
Basic graph theory
Invariance and equivariance

Multiple **equivalent representations** of the same graph:
The representation depends on the **enumeration** of the nodes



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

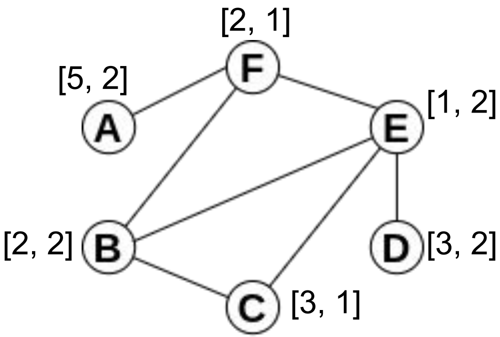
Barrier for the application of plain NNs



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

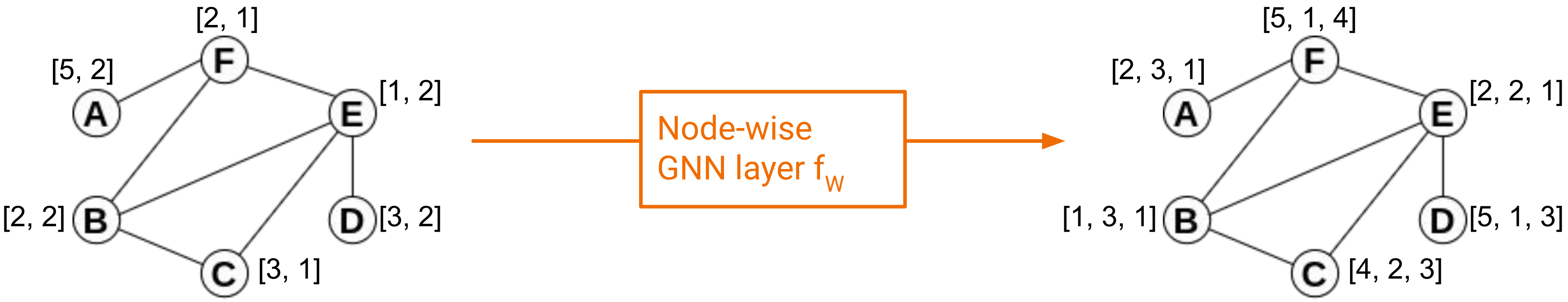
We need representations/layers which are
- **Permutation-invariant** for graph
- **Permutation-equivariant** for nodes

Graph Neural Networks
Learning node representations



Node features $X^{(0)} := X^V$ (or any assignment)
Dimension $d^{(0)}=d_V$

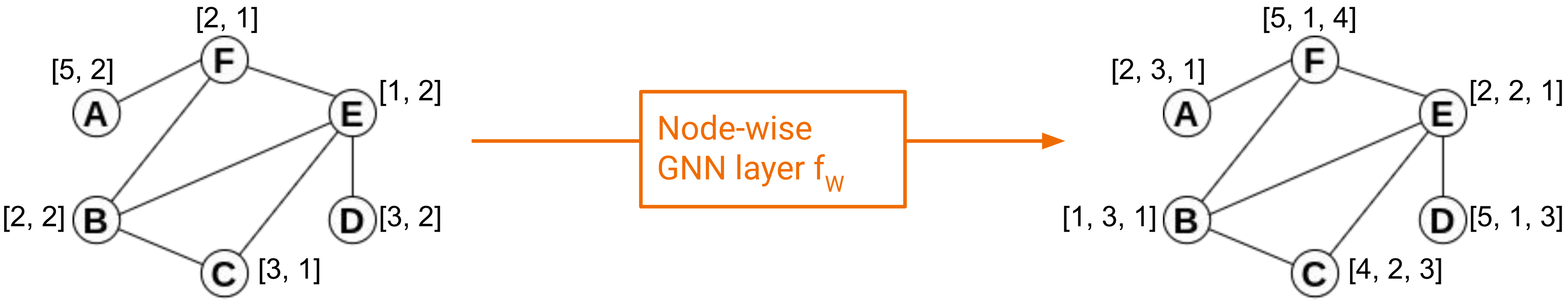
Graph Neural Networks
Learning node representations



Node features $X^{(0)}$:= X^V (or any assignment)
Dimension $d^{(0)}$=d_V

Updated node features $X^{(1)}$
Dimension $d^{(1)}$

Graph Neural Networks
Learning node representations



Node features $X^{(0)} := X^V$ (or any assignment)
Dimension $d^{(0)}=d_V$

Updated node features $X^{(1)}$
Dimension $d^{(1)}$

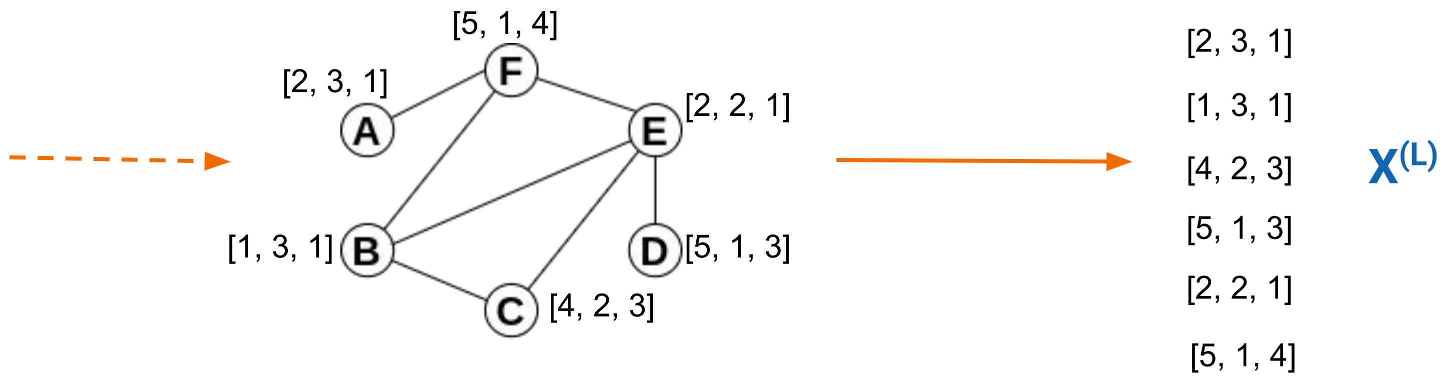
Key properties of $f_w(v)$:

- **Locality:** It depends only on v and $N^k(v)$ (mostly for k=1)
- **Invariance:** Invariant w.r.t. permutations of $N^k(v)$ and
  Independent of $|N^k(v)|$

Graph Neural Networks
Stacking

How to use a GNN layer?

- **Stacking/composition** of multiple layers
- **Final layer:** forget the topology, keep the node features
- These are **node embeddings**/mapped features
- Feed them to any **ML/DL method** (usually a final fully connected layer)

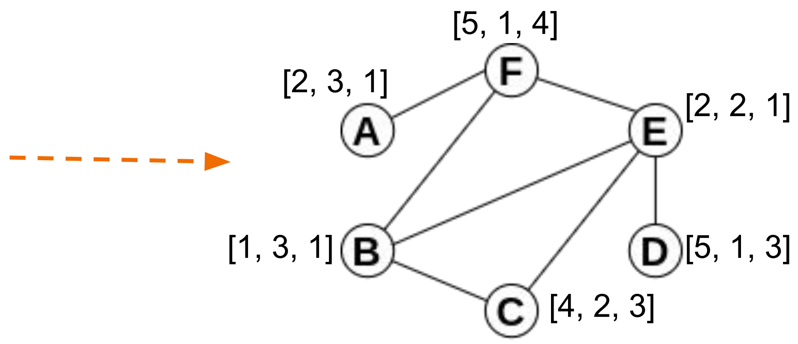


Graph Neural Networks
Stacking

Optimize the layer's parameters by minimizing a loss function based on a dataset

How to use a GNN layer?

- **Stacking/composition** of multiple layers
- **Final layer:** forget the topology, keep the node features
- These are **node embeddings**/mapped features
- Feed them to any **ML/DL method** (usually a final fully connected layer)



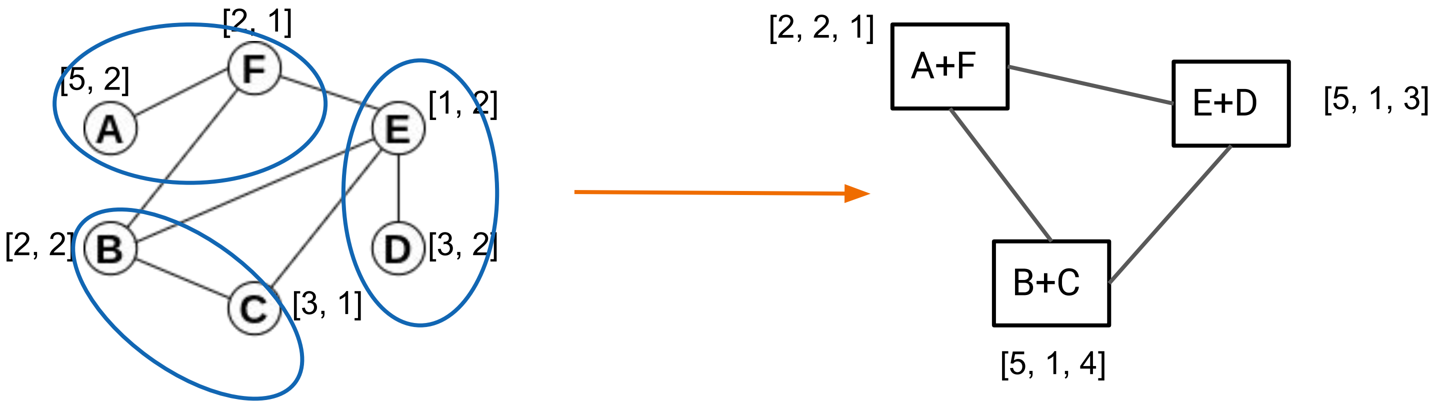
[2, 3, 1]
[1, 3, 1]
[4, 2, 3]
[5, 1, 3]
[2, 2, 1]
[5, 1, 4]

$\mathbf{X}^{(L)}$

Graph Neural Networks
Pooling

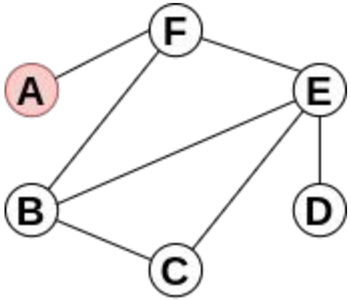
Pooling for graph representation

- Aggregate the final node representation (e.g. sum, mean, max, …)
- Pooling layers: dedicated layers that create a new (smaller) graph



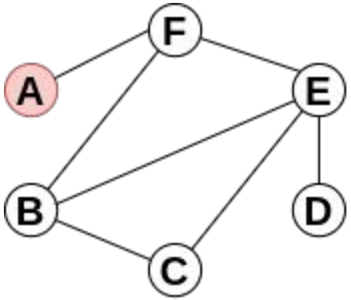
Graph Neural Networks
HowTo: Message passing

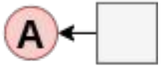
Flow of **computation** along the node structure



Graph Neural Networks
HowTo: Message passing

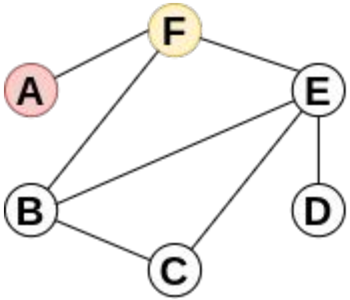
Flow of **computation** along the node structure

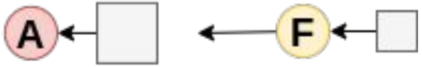




Graph Neural Networks
HowTo: Message passing

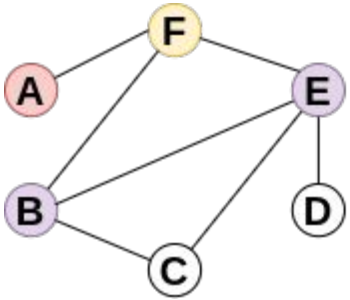
Flow of **computation** along the node structure

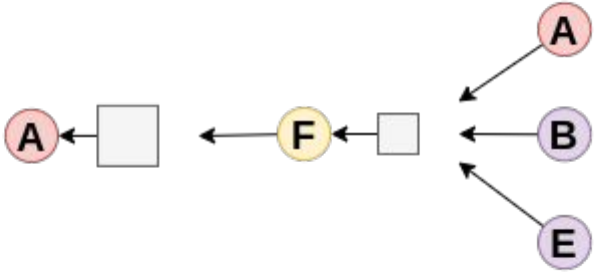




Graph Neural Networks
HowTo: Message passing

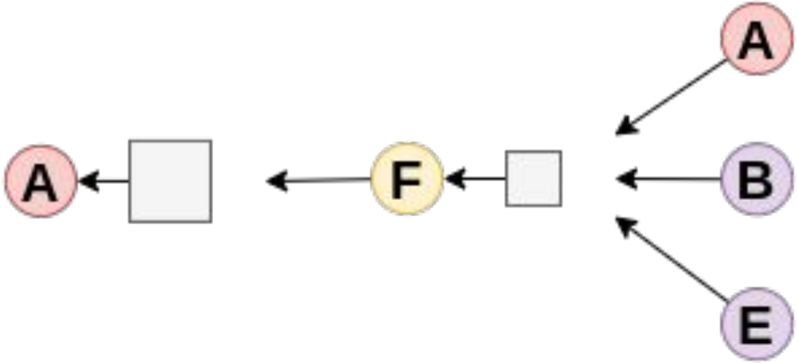
Flow of **computation** along the node structure





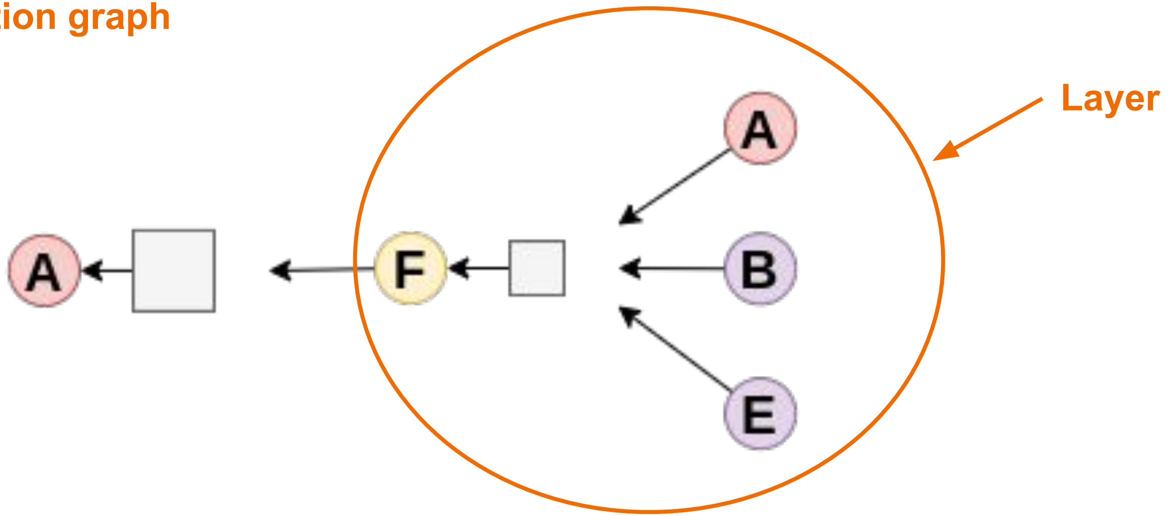
Graph Neural Networks
HowTo: Message passing

Resulting **computation graph**



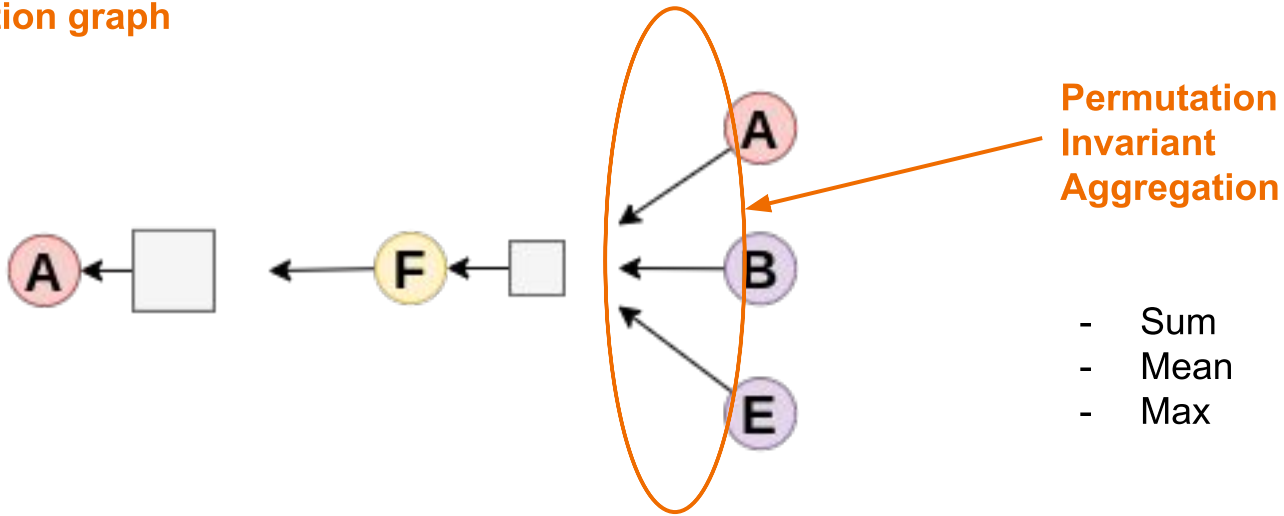
Graph Neural Networks
HowTo: Message passing

Resulting **computation graph**



Graph Neural Networks
HowTo: Message passing

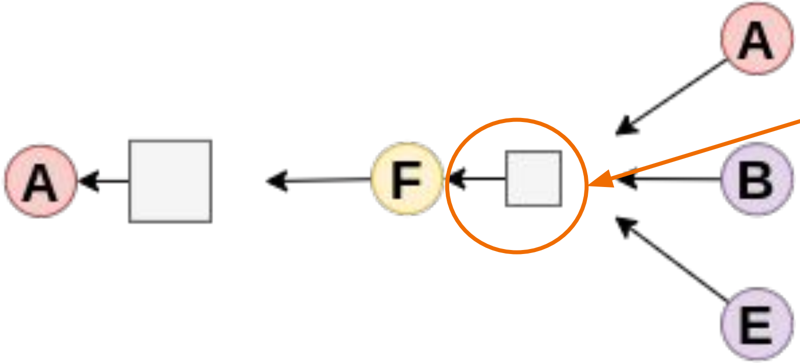
Resulting **computation graph**



- Sum
- Mean
- Max

Graph Neural Networks
HowTo: Message passing

Resulting **computation graph**



Lernable non-linear Mapping

Affine map
+
Component-wise nonlinearity

Graph Neural Networks
HowTo: Message passing

Properties:

- Used in actual **implementations**
- Fully **parallelizable** across the nodes
- Layer-wise application **removes redundancy**

Graph Neural Networks
HowTo: Message passing

Properties:
- Used in actual **implementations**
- Fully **parallelizable** across the nodes
- Layer-wise application **removes redundancy**

Insights: Risk of “oversmoothing”
- Too many layers → Convergence to **uniform node representations**
- Number of layers related to the graph diameter
- **Limit** for truly deep architectures

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

- We show a **prototype layer,** real examples will follow

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Input node features: $n \times d^{(i-1)}$

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1}\ A\ X^{(i-1)}\ W^{(i)})$$

Output node features: $n \times d^{(i)}$

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Learnable weights: $d^{(i-1)} \times d^{(i)}$

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Adjacency matrix: $n \times n$
(possibly with added self loops)

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Inverse degree matrix: $n \times n$
(possibly with added self loops)

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Component-wise nonlinear activation

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Linearly transformed features: $n \times d^{(i)}$

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Neighborhood-aggregated features: $n \times d^{(i)}$
Sum aggregation →**Permutation invariance**

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Neighborhood-aggregated features: $n \times d^{(i)}$
Degree normalization $\rightarrow$ **Indep. from $|N^k(v)|$**

Graph Neural Networks
HowTo: Linear algebra

Formalized **representation**

$$X^{(i)} = \sigma(D^{-1} A X^{(i-1)} W^{(i)})$$

Neighborhood-aggregated features: $n \times d^{(i)}$
Added **nonlinearity**

Graph Neural Networks
HowTo: Linear algebra

Properties:

- Not used in actual **implementations**
- Useful for layer analysis
- Equivalent to message passing via **sparse matrix** operations

Graph Neural Networks
HowTo: Linear algebra

Properties:

- Not used in actual **implementations**
- Useful for layer analysis
- Equivalent to message passing via **sparse matrix** operations

G:

$$X^{(i)} = \sigma(\boxed{D^{-1} A X^{(i-1)}} W^{(i)})$$

Insights:

- Clear model transferability / inductive learning

Graph Neural Networks
HowTo: Linear algebra

Properties:

- Not used in actual **implementations**
- Useful for layer analysis
- Equivalent to message passing via **sparse matrix** operations

Insights:

- Clear model transferability / inductive learning

G:

$$X^{(i)} = \sigma(\boxed{D^{-1} A X^{(i-1)}} W^{(i)})$$

G': $\boxed{D^{-1} A X^{(i-1)}}$

Graph Neural Networks
The main types of layers

Complete list In PyG: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

- Fully **modular** and **composable**

Graph Neural Networks
The main types of layers

Complete list In PyG: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

- Fully **modular** and **composable**
- Described in the following (with PyG notation):
    - GCN
    - GraphSAGE
    - GAT
    - Cheb
    - GIN

Graph Neural Networks
The main types of layers

Complete list In PyG: https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html

- Fully **modular** and **composable**
- Described in the following (with PyG notation):
    - GCN
    - GraphSAGE
    - GAT
    - Cheb
    - GIN

Shown without the nonlinear transform (custom choice)

Graph Neural Networks
The main types of layers: GCN

Graph Convolutional Network

In PyG: **GCNConv**

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta}$$

T. N. Kipf, M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, 2016

Graph Neural Networks
The main types of layers: GCN

Graph Convolutional Network

In PyG: **GCNConv**

$$\mathbf{X}' = \hat{\mathbf{D}}^{-1/2}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-1/2}\mathbf{X}\mathbf{\Theta}$$

Adjacency and degree matrices with added self loops
Symmetric multiplication

T. N. Kipf, M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, 2016

Graph Neural Networks
The main types of layers: GraphSAGE

Graph Sample and Aggregate

In PyG: **SAGEConv**

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \mathrm{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

W. L. Hamilton, R. Ying, J. Leskovec, *Inductive Representation Learning on Large Graphs*, 2017

Graph Neural Networks
The main types of layers: GraphSAGE

Graph Sample and Aggregate

In PyG: **SAGEConv**

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \mathrm{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

Hidden dependency on A, D

W. L. Hamilton, R. Ying, J. Leskovec, *Inductive Representation Learning on Large Graphs*, 2017

Graph Neural Networks
The main types of layers: GraphSAGE

Graph Sample and Aggregate

In PyG: **SAGEConv**

$$\mathbf{x}'_i = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \mathrm{mean}_{j \in \mathcal{N}(i)} \mathbf{x}_j$$

Different weighting of the node/neigh. features

W. L. Hamilton, R. Ying, J. Leskovec, *Inductive Representation Learning on Large Graphs*, 2017

Graph Neural Networks
The main types of layers: GAT

Graph Attention Network

In PyG: **GATConv**

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j \in \mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j$$

Different weighting of the node/neigh. features

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, *Graph Attention Networks*, 2018

Graph Neural Networks
The main types of layers: GAT

Graph Attention Network

In PyG: **GATConv**

$$\mathbf{x}'_i = \alpha_{i,i}\Theta\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\Theta\mathbf{x}_j$$

“Attention” parameters: Learnable edge weights

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, *Graph Attention Networks*, 2018

Graph Neural Networks
The main types of layers: GAT

Graph Attention Network

In PyG: **GATConv**

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j$$

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}$$

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, *Graph Attention Networks*, 2018

Graph Neural Networks
The main types of layers: GAT

Graph Attention Network

In PyG: **GATConv**

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j$$

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^\top[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}$$

Learnable feature weights

Same dim. as the node features

- Transferability

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, *Graph Attention Networks*, 2018

Graph Neural Networks
The main types of layers: Cheb

Chebyshev Spectral Graph Convolution

In PyG: **ChebConv**

$$\mathbf{X}' = \sum_{k=1}^{K} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, 2016

Graph Neural Networks
The main types of layers: Cheb

Chebyshev Spectral Graph Convolution

In PyG: **ChebConv**

$$\mathbf{X}' = \sum_{k=1}^{K} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

Multi-level weights

M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, 2016

Graph Neural Networks
The main types of layers: Cheb

Chebyshev Spectral Graph Convolution

In PyG: **ChebConv**

$$\mathbf{X}' = \sum_{k=1}^{K} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

$$\mathbf{Z}^{(1)} = \mathbf{X}$$

$$\mathbf{Z}^{(2)} = \mathbf{\hat{L}} \cdot \mathbf{X}$$

$$\mathbf{Z}^{(k)} = 2 \cdot \mathbf{\hat{L}} \cdot \mathbf{Z}^{(k-1)} - \mathbf{Z}^{(k-2)}$$

Scaled and normalized laplacian

M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, 2016

Graph Neural Networks
The main types of layers: Cheb

Chebyshev Spectral Graph Convolution

In PyG: ChebConv

$$\mathbf{X}' = \sum_{k=1}^{K} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

$$\mathbf{Z}^{(1)} = \mathbf{X}$$
$$\mathbf{Z}^{(2)} = \hat{\mathbf{L}} \cdot \mathbf{X}$$
$$\mathbf{Z}^{(k)} = 2 \cdot \hat{\mathbf{L}} \cdot \mathbf{Z}^{(k-1)} - \mathbf{Z}^{(k-2)}$$

Chebyshev approximation of a spectral filter

M. Defferrard, X. Bresson, P. Vandergheynst, Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering, 2016

Graph Neural Networks
The main types of layers: GIN

Graph Isomorphism Network

In PyG: **GINConv**

$$\mathbf{X}' = h_{\mathbf{\Theta}}\left(\left(\mathbf{A} + (1+\epsilon)\cdot\mathbf{I}\right)\cdot\mathbf{X}\right)$$

K. Xu, W. Hu, J. Leskovec, S. Jegelka, *How Powerful are Graph Neural Networks?*, 2019

Graph Neural Networks
The main types of layers: GIN

Graph Isomorphism Network

In PyG: **GINConv**

$$\mathbf{X}' = h_{\Theta}\left(\left(\mathbf{A} + (1+\epsilon)\cdot \mathbf{I}\right)\cdot \mathbf{X}\right)$$

Self loops with learnable weight

K. Xu, W. Hu, J. Leskovec, S. Jegelka, *How Powerful are Graph Neural Networks?*, 2019

Graph Neural Networks
The main types of layers: GIN

Graph Isomorphism Network

In PyG: **GINConv**

$$\mathbf{X}' = h_{\Theta}\left(\left(\mathbf{A} + (1+\epsilon)\cdot\mathbf{I}\right)\cdot\mathbf{X}\right)$$

Learnable MLP (on vector data)

K. Xu, W. Hu, J. Leskovec, S. Jegelka, *How Powerful are Graph Neural Networks?*, 2019

Some computational aspects
Introduction to the example used in part II

Node regression problem from a simple test case in **Model Order Reduction**:
Thermal block via **PyMOR** https://docs.pymor.org/latest/getting_started.html

The problem:

- Partial Differential Equation (PDE) in $[0, 1]^2$
- Model of the heat distribution in a solid plate
- Plate divided in a 2x2 grid of blocks with different heat conductivity: 4-dim parameter
- Zero temperature on the boundary of the square

Some computational aspects
Introduction to the example used in part II

Node regression problem from a simple test case in **Model Order Reduction**:
Thermal block via **PyMOR** https://docs.pymor.org/latest/getting_started.html

Problem discretization: get G = (V, E)

- **V, E**: Generate a uniform mesh: equally spaced points with diameter h, triangulated

- FIx a value of the parameter

- Define $\mathbf{X}^{\mathbf{V}}$ as the concatenation of (node position, parameter value, diameter value)

- Define the target values **y**: Solve the problem with the Finite Element Method (FEM)

Some computational aspects
Introduction to the example used in part II

Node regression problem from a simple test case in **Model Order Reduction**:
Thermal block via **PyMOR** https://docs.pymor.org/latest/getting_started.html

Data generation for the node regression problem:

- Generate discretizations with
  - Diameters **h=0.01, 0.02, …, 0.1** (implying different topologies)
  - Parameters $\mathbf{p} = \mathbf{p}_1, \dots, \mathbf{p}_4$ randomly generated

Some computational aspects
Introduction to the example used in part II

Node regression problem from a simple test case in **Model Order Reduction**:
Thermal block via **PyMOR** https://docs.pymor.org/latest/getting_started.html

Data generation for the node regression problem:

- Generate discretizations with
  - Diameters **h=0.01, 0.02, …, 0.1** (implying different topologies)
  - Parameters $\mathbf{p} = \mathbf{p}_1, \ldots, \mathbf{p}_4$ randomly generated

Goal: Demonstrate model learning and transferability over multiple graphs

Some computational aspects
Introduction to the example used in part II

Node regression problem from a simple test case in **Model Order Reduction**:
Thermal block via **PyMOR** https://docs.pymor.org/latest/getting_started.html

Data generation for the node regression problem:

- Generate discretizations with
  - Diameters **h=0.01, 0.02, …, 0.1** (implying different topologies)
  - Parameters $\mathbf{p} = \mathbf{p}_1, \ldots, \mathbf{p}_4$ randomly generated

Goal: Demonstrate model learning and transferability over multiple graphs

Warning: we show no fine-tuning, no spectacular accuracies