

Graph representation learning and graph classification

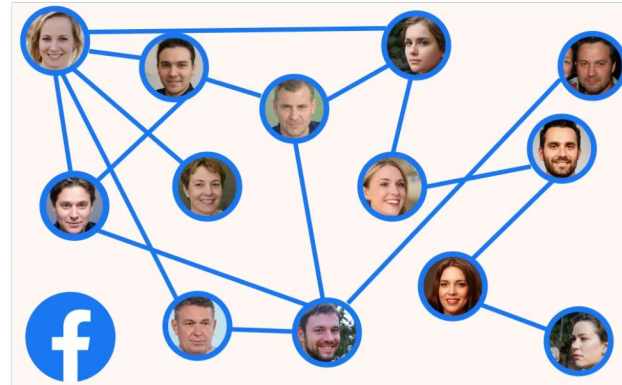
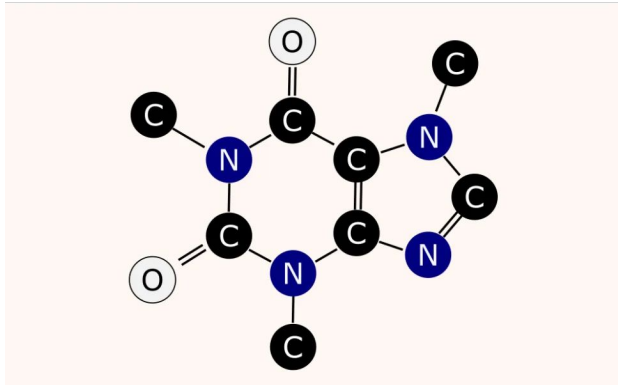
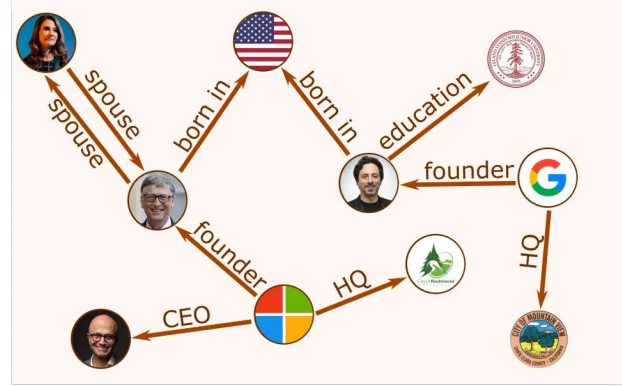
Jakub Adamczyk, Faculty of Computer Science, AGH

General plan

1. Intro
2. Graph representation learning
3. Graph feature extraction
4. Graph kernels
5. Graph Neural Networks (GNNs) intro
6. Graph convolutions
7. Fair evaluation for graphs
8. Oversmoothing and oversquashing
9. Graph pooling
10. Pretraining GNNs

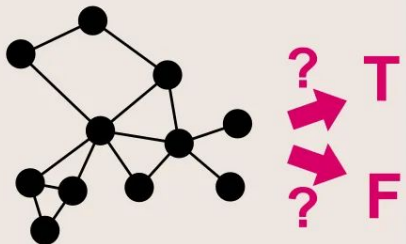
Why graphs?

Graph data

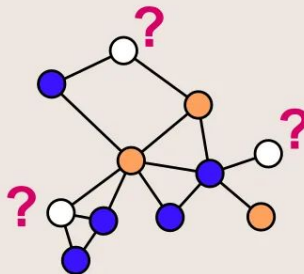


Graph learning applications

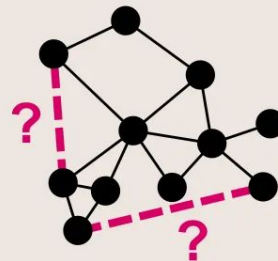
Graph Classification



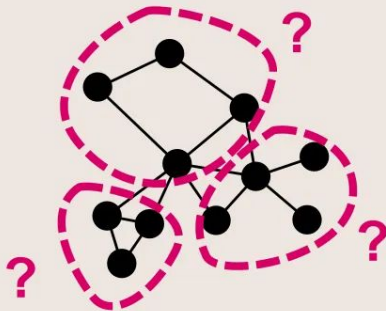
Node Classification



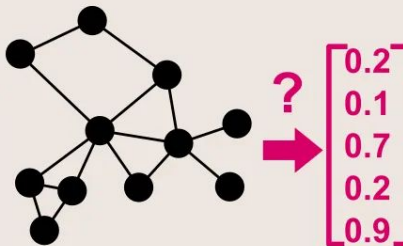
Link Prediction



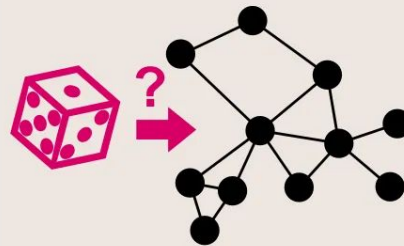
Community Detection



Graph Embedding



Graph Generation



Real world applications

- **Bayer, BenevolentAI:**

- molecular property prediction ([source](#))
- de novo drug design ([source](#))



Benevolent^{AI}

- **Google:**

- travel time prediction ([source](#))
- TPU chip design ([source](#))



- **Uber Eats, Pinterest, Alibaba:**

- recommendation systems ([source](#), [source](#), [source](#))



- **Amazon:**

- demand forecasting ([source](#))



Learning on whole graphs

- **tasks:** graph classification / regression, embedding, clustering
- lots of applications
- **bio- & chemoinformatics:**
 - molecular property prediction, e.g. "Is molecule an HIV inhibitor?"
 - protein function prediction, e.g. "Does this protein bind to target?"
 - molecular similarity search, e.g. "What molecules are similar functionally?"
- **social network analysis**, e.g. community type classification
- program analysis, vision understanding, algorithmic reasoning, ...

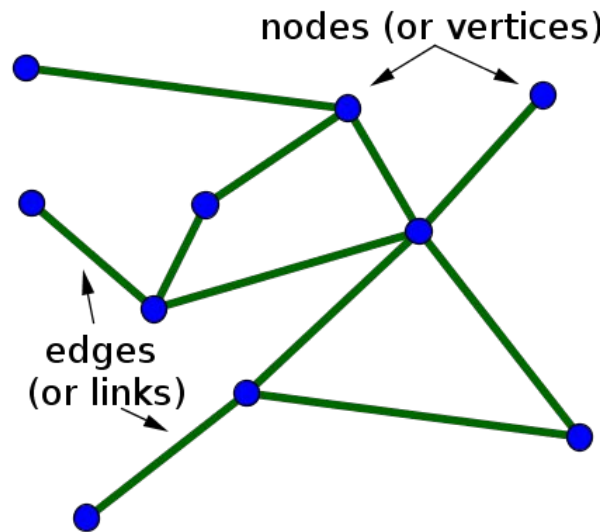
Introductory definitions

Graph

- **graph** is a **combinatorial, discrete** data structure
- this is a **non-Euclidean** structure - no natural algebraic space, axii, distance, ordering etc.
- it consists of **vertices (nodes)** and **edges (links)**, which are **sets** of size $|V|$ and $|E|$

$$G = (V, E)$$

- we have no ordering, so everything we do must be **permutation invariant** (or equivariant)

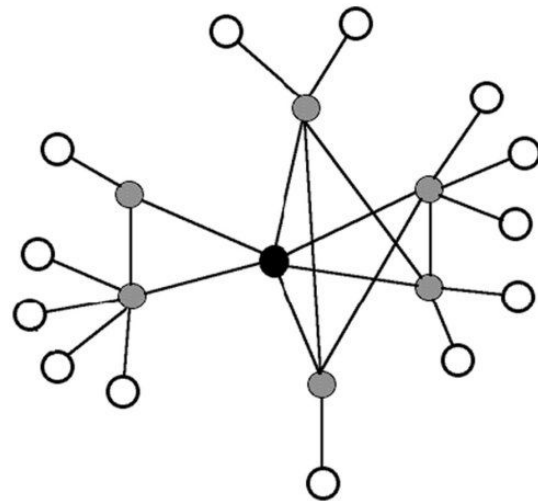
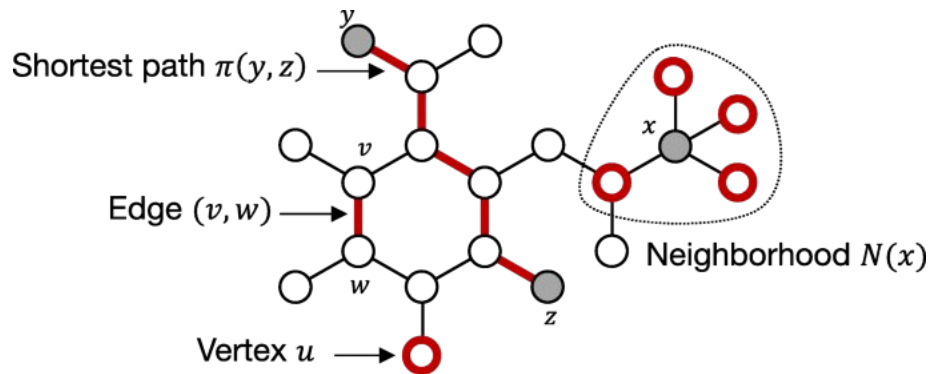


Neighbors

- nodes **connected** by an edge are **neighbors**
- we **hop** through edges to **walk** through graph
- **1-hop** or **direct neighborhood** is the set of directly connected nodes

$$N(v) = \{u | e(v, u) \in E\}$$

- we can define **k-hop neighborhoods**, i.e. all nodes where the **shortest paths** are at most k
- the **only** notion of distance that we have!



Kriege, N. M., Johansson, F. D., Morris, C. "A survey on graph kernels"

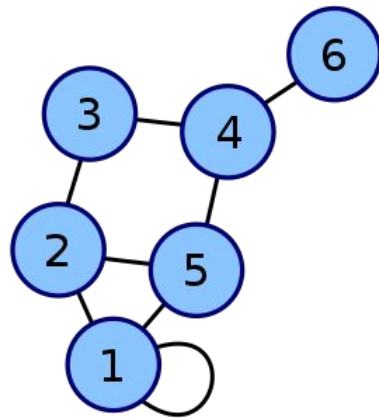
https://www.researchgate.net/figure/An-illustration-of-a-one-hop-and-two-hop-neighborhood-The-target-node-is-shown-in-black_fig2_51884663

Adjacency matrix

- graph can be represented with **adjacency matrix** A , which defines its **topology (structure)**

$$A \in \mathbb{R}^{|V| \times |V|}$$

- we consider only undirected, unweighted graphs:
 - A is symmetric
 - $A_{ij} = 1$ if edge e_{ij} exists between nodes i and j
- A is typically **sparse**, with majority of 0 (zero) values
- nodes may have **self-loops**, i.e. edges to itself



$$\begin{pmatrix} 2 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Describing nodes

- each node has a **degree**, equal to the **number of its neighbors**

$$\deg(v_i) = |N(v_i)| = |\{u | u \in N(v_i)\}|$$

- **degree matrix D** is a diagonal matrix with node degrees

$$D \in \mathbb{R}^{|V| \times |V|}$$

- **node homophily** - property where close nodes have more similar properties, represent similar objects
- we have n graphs in our dataset

Attributed graphs

- graph can be **attributed**, where nodes and/or edges have attributes; most often we have **nodes feature matrix X** , with d features

$$G = (A, X) \text{ or } G = (A, X_n, X_e) \quad X \in \mathbb{R}^{|V| \times d}$$

- sometimes there is a distinction between:
 - discrete **node / edge labels**, e.g. node degree, bond type
 - continuous **node / edge features**, e.g. atom formal charge
- feature matrix typically means the latter

Graph representation learning

What is representation learning?

*"**Representation learning** is a process in ML where algorithms extract meaningful patterns from raw data to create representations that are easier to understand and process."*

- **vectorize** graph, i.e. transform it into **embedding** (feature vector)
- can be:
 - **manual** - feature extraction
 - **semi-automated** - define algorithm, which learns from data
 - **automated** - neural networks, learn very flexibly from data
- another distinction:
 - **unsupervised** - extract features useful "in general"
 - **supervised** - extract task-relevant features

Graph representation learning

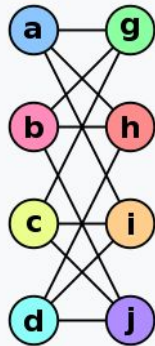
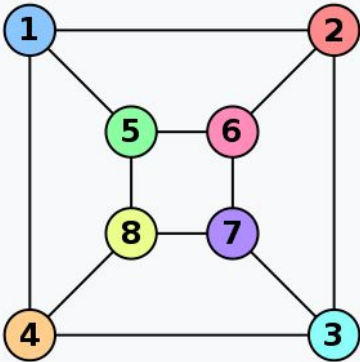
| Approach | Methods |
|--|---------------------------------|
| Manual, unsupervised | Graph feature extraction |
| Semi-automated, typically unsupervised | Graph kernels |
| Automated, typically supervised | Graph Neural Networks (GNNs) |

What is a useful representation?

- we can represent objects in many ways
- **useful properties:**
 - easy to implement
 - fast to compute
 - low memory requirements
 - allows **algebraic operations:**
 - comparison - are two objects equal?
 - distance - how similar are two objects?
 - **semantically meaningful**, e.g. captures intrinsic properties of objects

Graph isomorphism problem

- representation learning allows us to ask questions, e.g. about object similarity
- **basic question:** "Are two graphs identical"?
- known as:
 - **graph isomorphism problem**
 - exact graph matching
- "identical" graphs are isomorphic
- we can map "the same" nodes in two graphs to each other with a bijection

| Graph G | Graph H | An isomorphism between G and H |
|--|---|--|
|  |  | $f(a) = 1$ $f(b) = 6$ $f(c) = 8$ $f(d) = 3$ $f(g) = 5$ $f(h) = 2$ $f(i) = 4$ $f(j) = 7$ |

Graph isomorphism problem

- turns out to be **surprisingly hard** problem!
 - no polynomial time algorithm known, may be NP-complete
 - right now is a separate complexity group, "GI" (e.g. GI-hard, GI-complete)
- **question:** "For graphs A and B, does B contain subgraph that is isomorphic with A"?
 - **subgraph isomorphism problem**
 - even harder, NP-complete
- **question:** "How similar are two graphs?"
 - many different answers - what is "similarity"?
 - use graph topology, node features, edge features, ...?

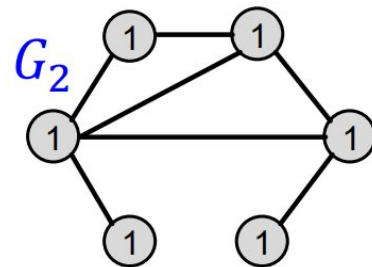
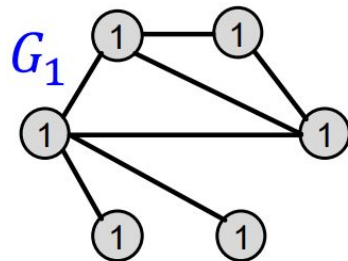
Not all is lost

- graph isomorphism is hard as a general problem
- there are lots of practical **approximations**
- we can **solve it for most graphs** using **Weisfeiler-Lehman isomorphism test (WL-test)**
 - if it says graphs are not isomorphic, it is 100% correct
 - if it says two graphs are isomorphic, they **may** be non-isomorphic
- can't differentiate some graphs, but generally powerful
- **later:** it can also tell **how** different are two graphs, i.e. similarity!

Weisfeiler-Lehman isomorphism test

Weisfeiler-Lehman test (WL-test)

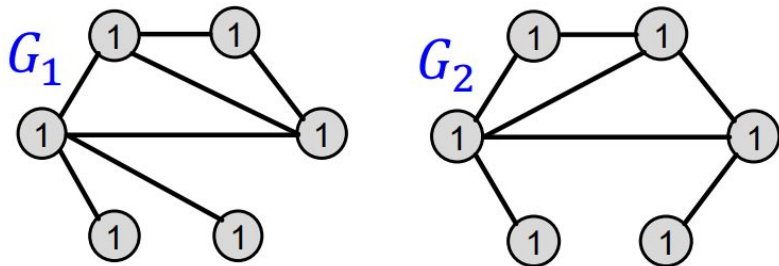
- **idea:**
 - understanding whole graph is hard
 - understanding **neighborhood** is **easy**
 - **exchange information** between neighboring nodes
- nodes gain information about graph topology
- we do this for 2 graphs at once, **in a loop**, for $|V|$ iterations
 - **any difference** = graphs are definitely **not** isomorphic, break
 - else = graphs are **probably** isomorphic



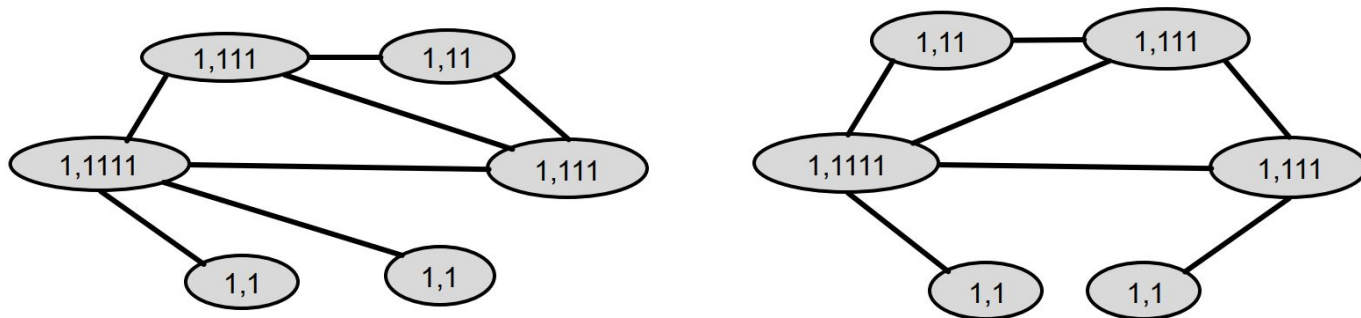
Source (theory & images): [Stanford CS224W: Machine Learning with Graphs](#) ([Youtube lecture](#))

WL-test - 1st iteration

0. Label each node uniformly

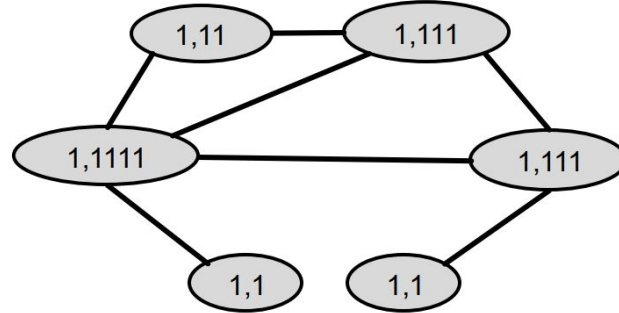
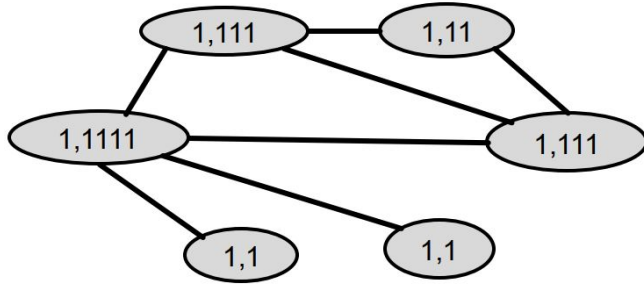


1. Send labels to neighbors & sort them

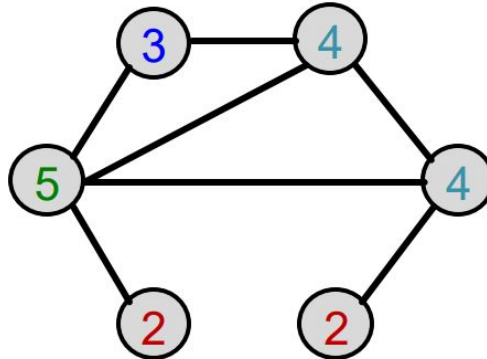
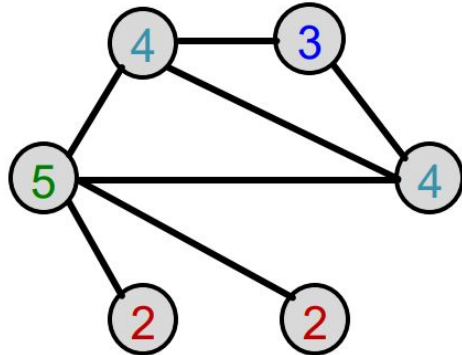


WL-test - 1st iteration

1. Send labels to neighbors & sort them



2. Compress (aggregate) labels

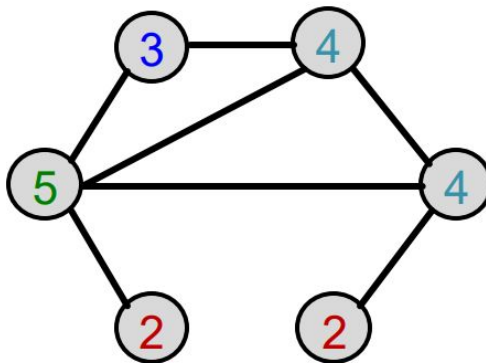
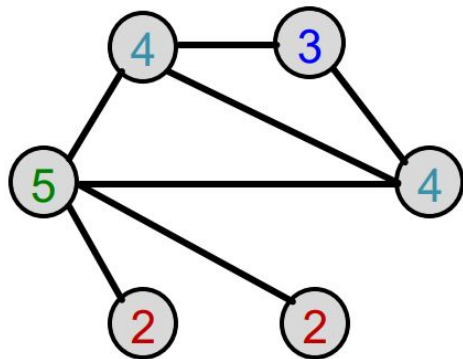


Hash table

| | | |
|--------|-----|---|
| 1,1 | --> | 2 |
| 1,11 | --> | 3 |
| 1,111 | --> | 4 |
| 1,1111 | --> | 5 |

WL-test - 1st iteration

2. Compress (aggregate) labels



Hash table

| | | |
|--------|-----|---|
| 1,1 | --> | 2 |
| 1,11 | --> | 3 |
| 1,111 | --> | 4 |
| 1,1111 | --> | 5 |

3. Loop iteration finished, compare label counts:

| Label count | 1 | 2 | 3 | 4 | 5 |
|-------------|---|---|---|---|---|
| G | 6 | 2 | 1 | 2 | 1 |
| G' | 6 | 2 | 1 | 2 | 1 |

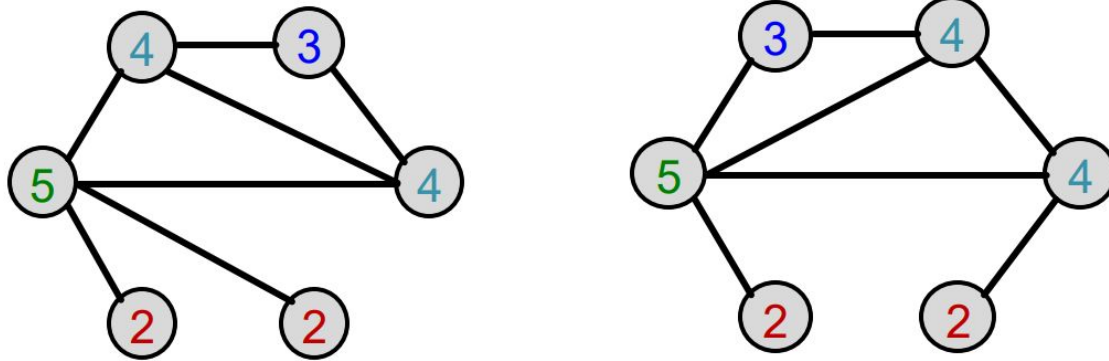
We learned:

- number of nodes
- histogram of node degrees

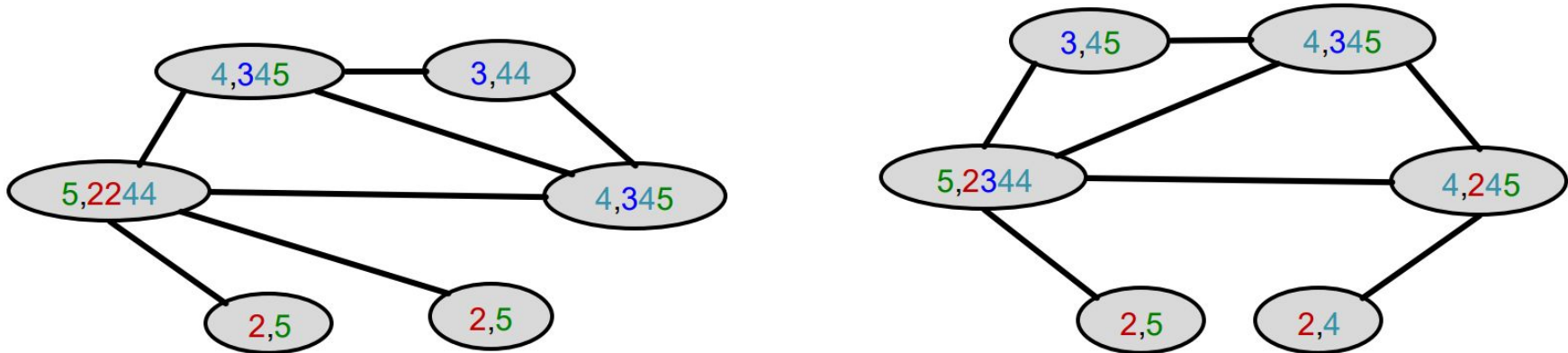
Identical -> continue with next iteration

WL-test - 2nd iteration

1. Labeled graph after 1st iteration

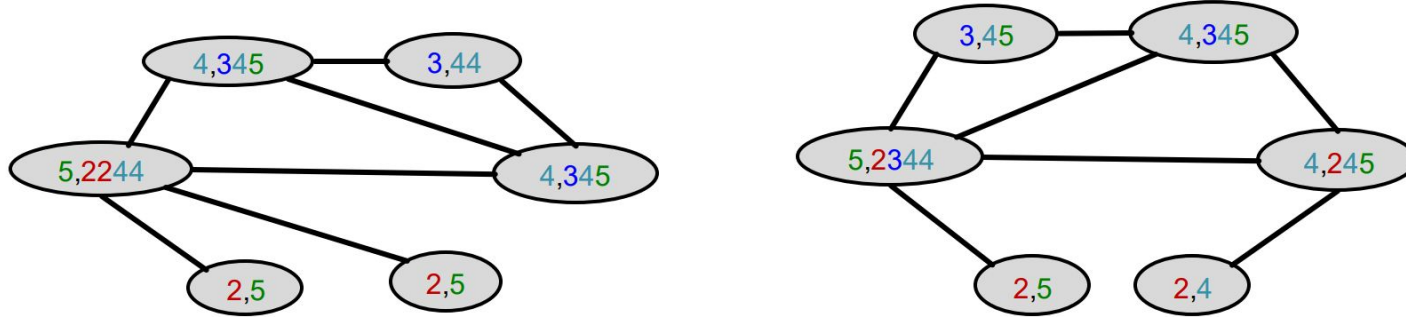


2. Send labels to neighbors & sort them

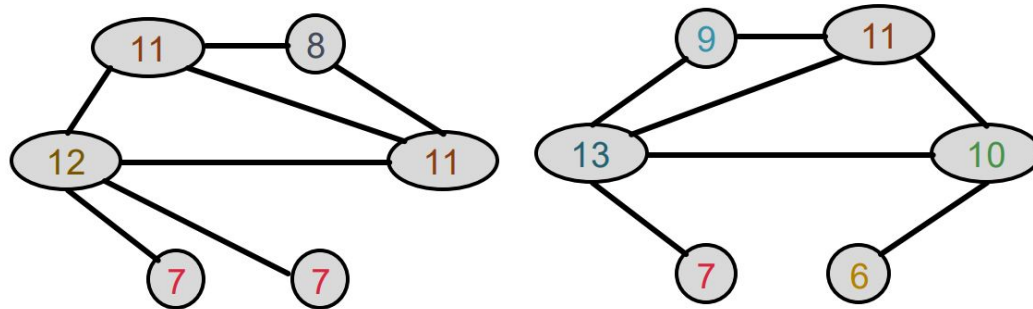


WL-test - 2nd iteration

2. Send labels to neighbors & sort them



3. Compress (aggregate) labels

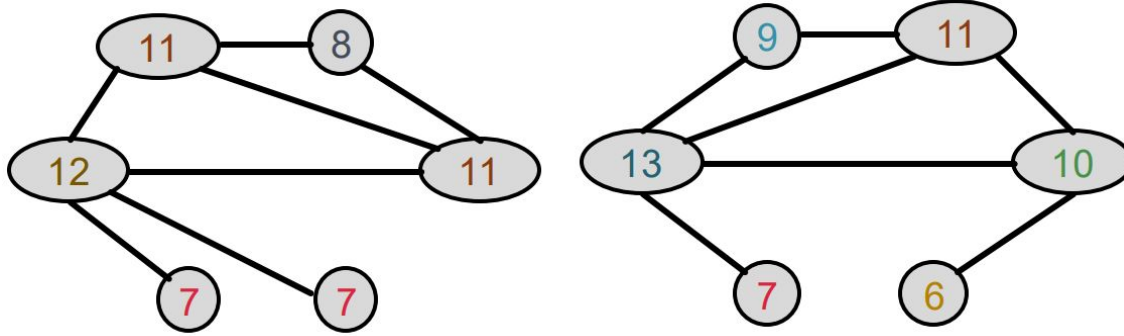


Hash table

| | | |
|--------|-----|----|
| 2,4 | --> | 6 |
| 2,5 | --> | 7 |
| 3,44 | --> | 8 |
| 3,45 | --> | 9 |
| 4,245 | --> | 10 |
| 4,345 | --> | 11 |
| 5,2244 | --> | 12 |
| 5,2344 | --> | 13 |

WL-test - 2nd iteration

3. Compress (aggregate) labels



4. Loop iteration finished, compare label counts:

| Label count | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|-------------|---|---|---|---|---|---|---|---|---|----|----|----|----|
| G | 6 | 2 | 1 | 2 | 1 | 0 | 2 | 1 | 0 | 0 | 2 | 1 | 0 |
| G' | 6 | 2 | 1 | 2 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

Different -> definitely not isomorphic, break

WL-test - complexity

- h iterations, at most $|V|$
- in each iteration we exchange $|E|$ labels
- **computational complexity:** $O(|V| * |E|)$

Further reading

- Huang, N.T., Soledad, V. *"A short tutorial on the Weisfeiler-Lehman test and its variants"*
- Shervashidze, N., et al. *"Weisfeiler-Lehman graph kernels"*

Graph feature extraction

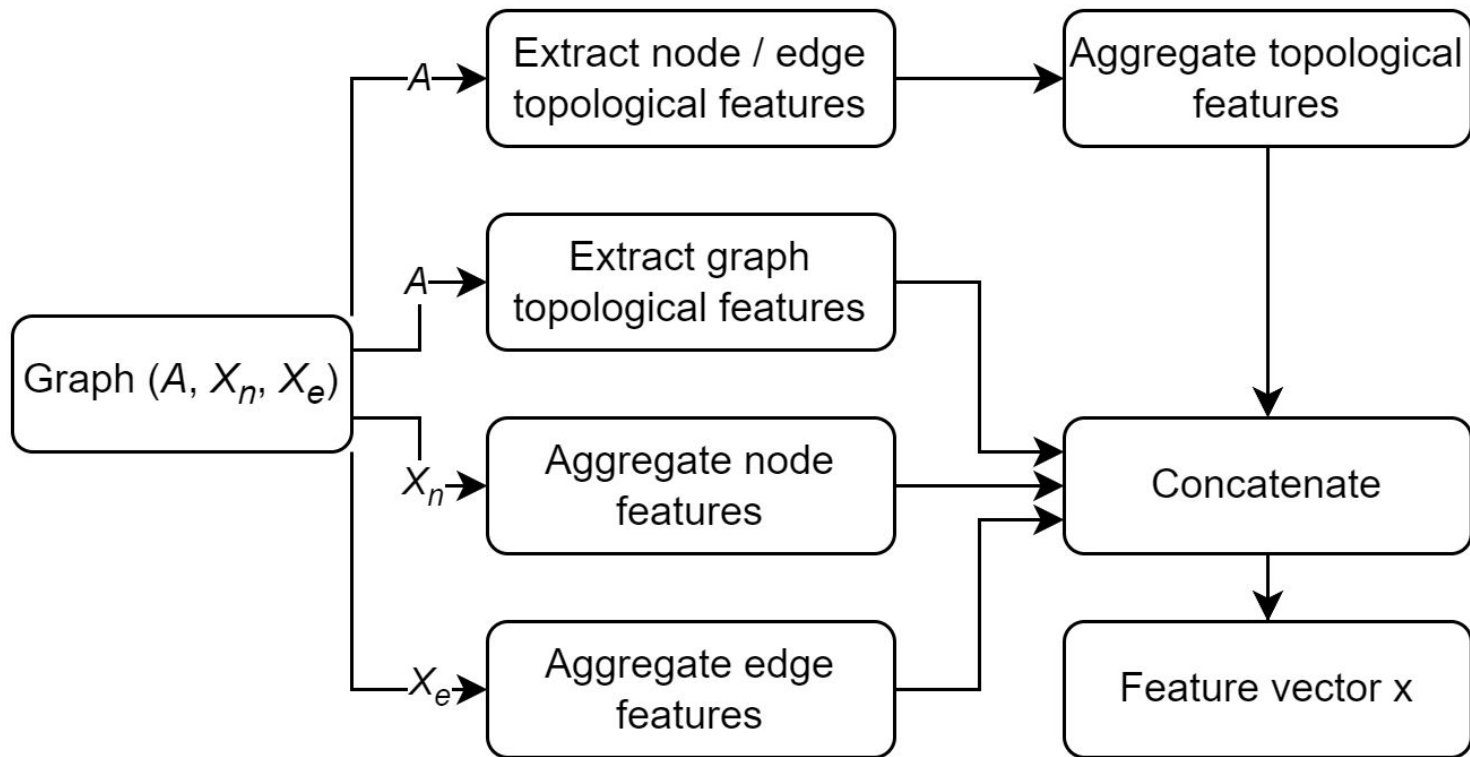
Graph feature extraction

- **idea:** manual feature engineering, either **topological** or **domain-specific**
- **topological features:**
 - based solely on graph structure
 - defined for nodes, edges, or whole graph
 - can be computed for arbitrary graph, without specific preparation
- **domain-specific features:**
 - node (and/or edge) feature matrix X
 - part of dataset - have to be gathered and/or computed beforehand!
 - domain-specific, e.g. atom/bond types in chemistry

Graph feature extraction

- **problem:**
 - node features are defined for each node separately (same for edges)
 - we need feature for the whole graph
- **aggregation:**
 - common operation in graph-based ML
 - we aggregate a feature for all nodes (edges)
 - sum, mean, max, min, stddev, histogram, ...
- **examples:**
 - histogram of node degrees, average shortest path length
 - total number of atoms/bonds of each type

Graph feature extraction pipeline



Local Degree Profile (LDP)

- Cai, C., Wang, Y. *"A simple yet effective baseline for non-attributed graph classification"*
 - **purely topological** method, based only on **node degrees**
 - surprisingly good results, often on par with much more sophisticated solutions!
 - **idea:**
 - node degree = information about its neighborhood
 - neighbors degrees = information about 2-hop neighborhood
 - **algorithm:**
 - extract 5 features: node degree + min, mean, max, stddev of **degrees of neighbors (DN)**
- $$x_i = [deg(v_i), \min(DN(v_i)), \text{mean}(DN(v_i)), \max(DN(v_i)), \text{std}(DN(v_i))]$$
- aggregate each with histogram

Local Degree Profile (LDP)

- **computational complexity:** $O(|E|)$
- original version has 4 hyperparameters and uses SVM for classification
- **deeper analysis:**

Adamczyk, J., Czech, W. *"Strengthening structural baselines for graph classification using Local Topological Profile"*

- no hyperparameter tuning necessary
- histogram with 50 equal width bins
- no normalization or preprocessing necessary
- best used with Random Forest

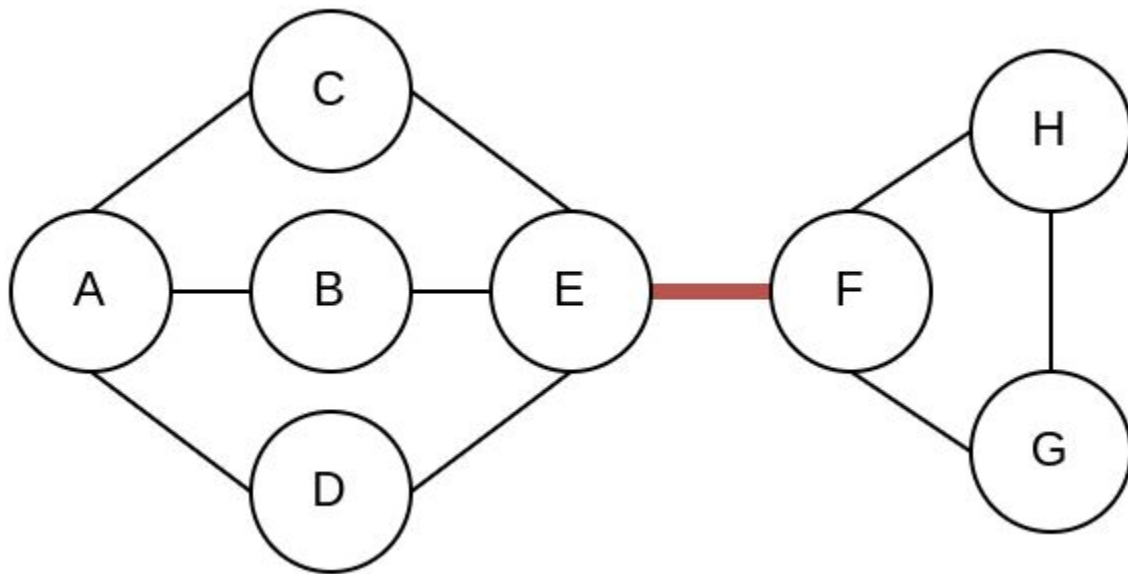
Local Topological Profile (LTP)

- Adamczyk, J., Czech, W. *"Strengthening structural baselines for graph classification using Local Topological Profile"*
- **extension** of LDP with 3 additional topological features
- **idea:** incorporate more **global** information, while still using **local** descriptors
 - consistently improve accuracy
 - simple & fast to compute
- **additional descriptors:**
 - edge betweenness centrality
 - Jaccard index
 - Local Degree Score

LTP - edge betweenness centrality

- **edge** feature
- how many **shortest paths** go through the edge
- high value means a **bridge**, edge with high **information flow** through it

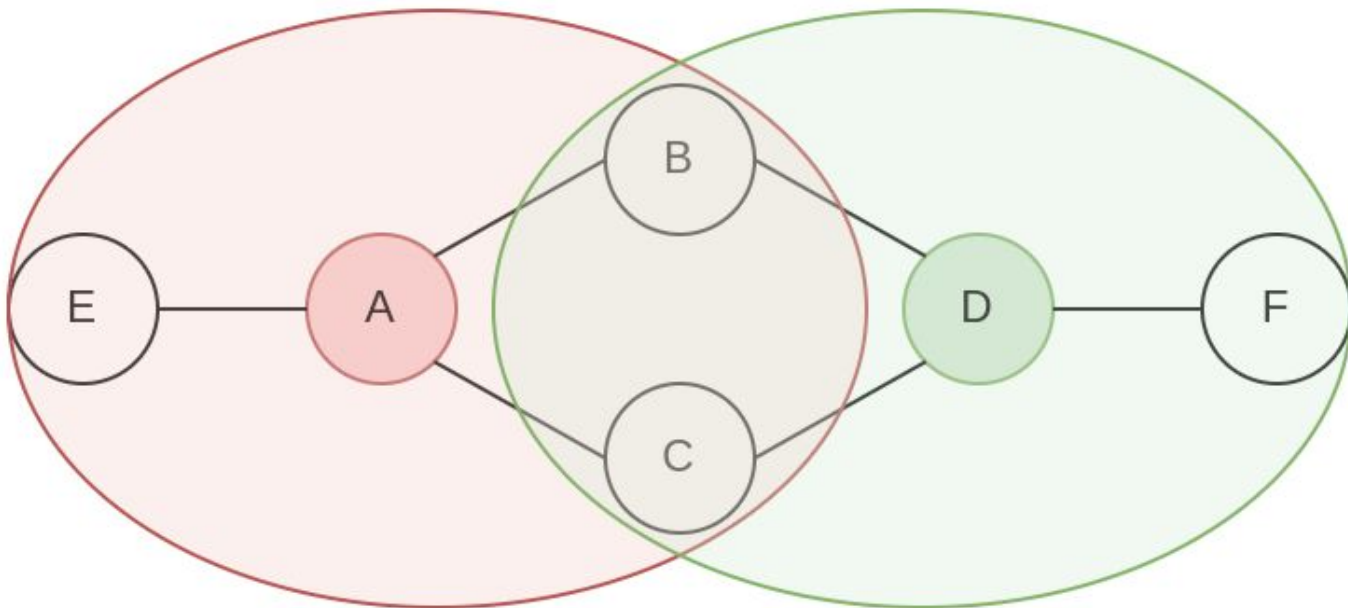
$$EBC(e) = \sum_{s,t; (s,t) \neq (u,v)} \frac{\sigma_{st}(e)}{\sigma_{st}}$$



LTP - Jaccard index

- **subgraph** feature, centered on nodes
- how much node neighborhoods **overlap**
- very cheap **3-hop neighborhood** information!

$$JI(u, v) = \frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$$

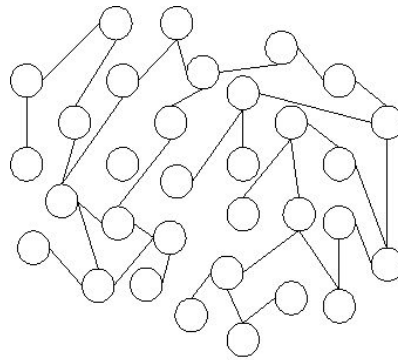


LTP - Local Degree Score

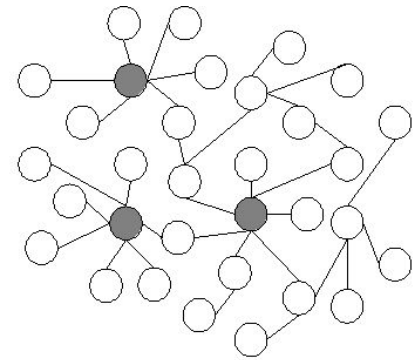
- **graph** feature, centered on edges
- how important is an edge to keep the **components** connected (e.g. clusters, communities)
- high LDS means that there are **hubs** in the graph - high degree, "influential" nodes

$$LDS(e) = \max \left(1 - \frac{\ln \text{rank}(v, u)}{\ln \text{degree}(v)}, 1 - \frac{\ln \text{rank}(u, v)}{\ln \text{degree}(u)} \right)$$

$\text{rank}(u, v)$ - for node v , how many neighbors have degree lower than u



(a) Random network



(b) Scale-free network

Coding time #1

Graph feature extraction - pros and cons

Pros:

- simple
- fast & scalable & parallelizable
- customizable
- can easily combine topology & features

Cons:

- often too simple
- aggregating loses information
- hard to incorporate complex feature interactions
- topology & features are considered almost separately

Further reading

- Cai, C., Wang, Y. *"A simple yet effective baseline for non-attributed graph classification"*
- Adamczyk, J., Czech, W. *"Strengthening structural baselines for graph classification using Local Topological Profile"*
- Galland, A., & Lelarge, M. *"Invariant embedding for graph classification"*
- de Lara, N., & Pineau, E. *"A simple baseline algorithm for graph classification"*

Graph kernels

Kernels for graph classification

- **idea:** define function that captures **structural similarity** between graphs
- **kernel functions:**
 - describe similarity between objects
 - with dot product
 - in a high-dimensional space
 - in a very flexible way

$$K(x, y) = \langle \phi(x), \phi(y) \rangle = \phi(x) \cdot \phi(y)$$

- **answer the question** "How similar are two graphs?"

Kernels for tabular data

- typical kernels for **tabular (vector) data**:

- polynomial

$$K(x, y) = (1 + x \cdot y)^k$$

- RBF

$$K(x, y) = e^{-\frac{(x-y)^2}{2\sigma}}$$

- both of those kernels are **implicit** - we never compute actual high-dimensional feature vectors
- we gather those into **kernel matrix**:

$$[K]_{i,j} = K(x_i, x_j)$$

- **kernel methods**, e.g. kernel SVM, are generic - only use object similarities, without looking at actual objects!

Graph kernels

- many graph kernels are **explicit** - when no closed analytical formula exists
- we **explicitly** calculate the kernel:

$$K(x, y) = \langle \phi(x), \phi(y) \rangle = \phi(x) \cdot \phi(y)$$

- calculate feature vectors $\phi(x)$ and $\phi(y)$
 - then calculate dot product
- feature vectors are typically **subgraph counts**
- checking all subgraphs = **subgraph isomorphism problem** = not possible in practice
- graph kernels differ in what subgraphs they count

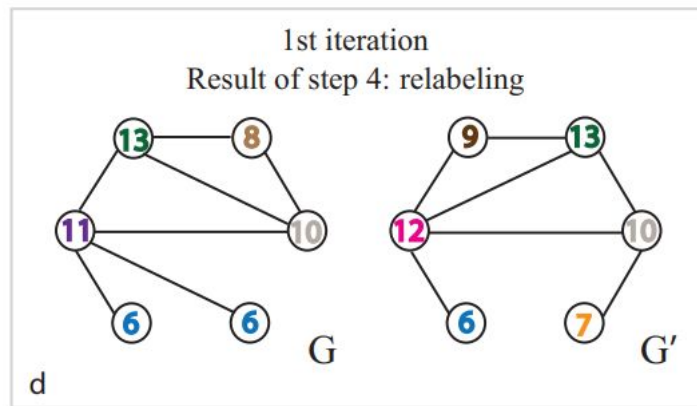
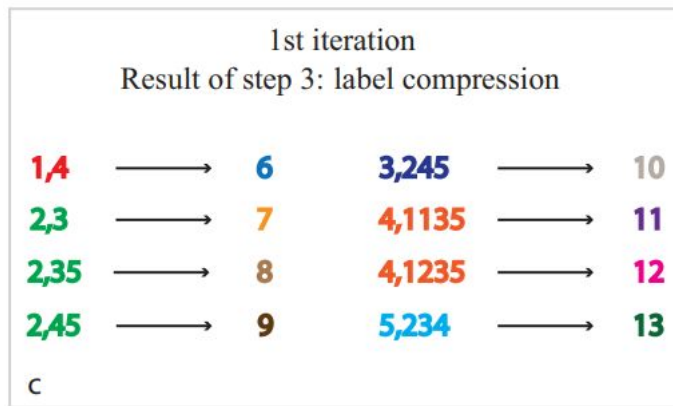
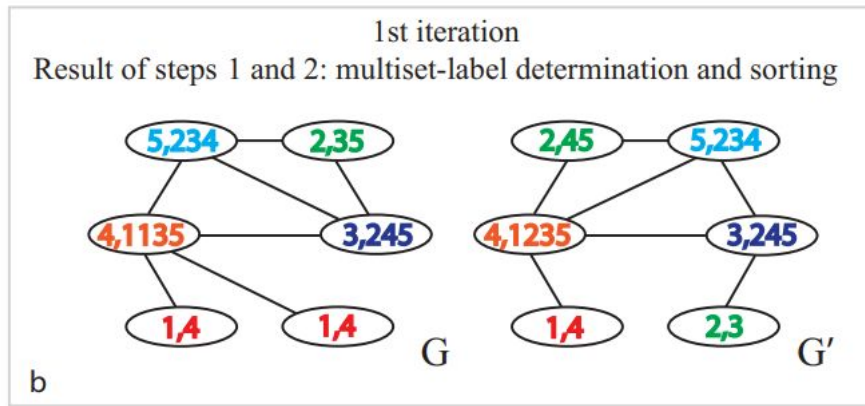
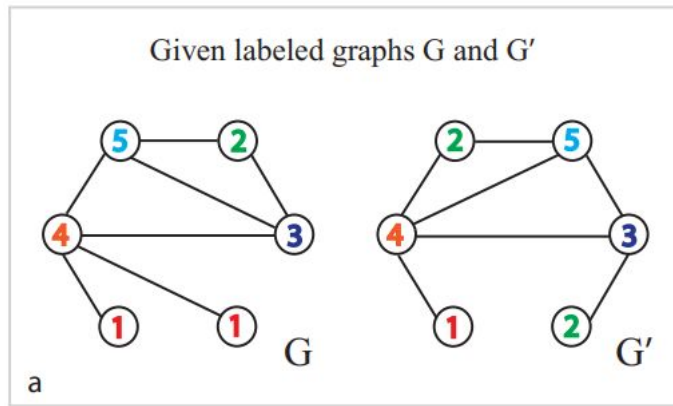
Graph kernels examples

| Graph kernel | Counted feature | Computational complexity | Classification quality |
|-------------------|----------------------|--|------------------------|
| Vertex histogram | Vertex degrees | $O(E)$ | Bad |
| Shortest paths | Shortest paths | $O(V ^4)$ | Medium |
| Random walks | Short random walks | $O(V ^6)$ (can be optimized to $O(V ^3)$) | Medium |
| Weisfeiler-Lehman | WL-test label counts | $O(h E)$ (with h iterations) | Good |

WL-kernel

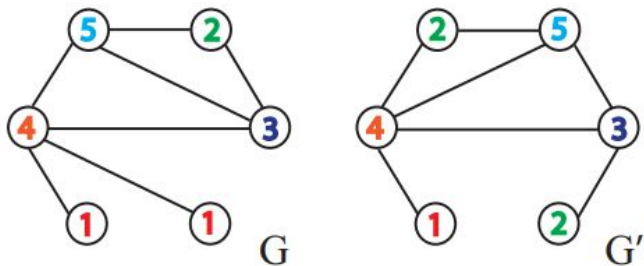
- Shervashidze, N., et al. *"Weisfeiler-Lehman graph kernels"*
- **idea:** perform h iterations of Weisfeiler-Lehman test and use label counts as feature vectors
- formally called **Weisfeiler-Lehman subtree kernel**
- often uses vertex degrees as original labels

WL-kernel example



WL-kernel example

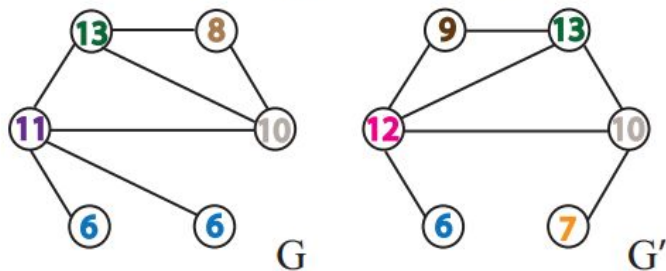
Given labeled graphs G and G'



a

1st iteration

Result of step 4: relabeling



d

End of the 1st iteration
Feature vector representations of G and G'

$$\phi_{WLsubtree}^{(1)}(G) = (\mathbf{2}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{2}, \mathbf{0}, \mathbf{1}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1})$$

$$\phi_{WLsubtree}^{(1)}(G') = (\mathbf{1}, \mathbf{2}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1}, \mathbf{1}, \mathbf{0}, \mathbf{1}, \mathbf{1})$$

Counts of
original
node labels

Counts of
compressed
node labels

$$k_{WLsubtree}^{(1)}(G, G') = \langle \phi_{WLsubtree}^{(1)}(G), \phi_{WLsubtree}^{(1)}(G') \rangle = 11.$$

e

Kernel methods complexity

- kernel matrix has **quadratic size**: $n \times n$
- we have to calculate kernel for each pair of graphs
- kernel methods complexity is **multiplied** by n^2
- **example:**
 - WL-kernel matrix: $O(h|E| * n^2)$
 - shortest paths kernel matrix: $O(|V|^4 * n^2)$
- we have to add e.g. kernel SVM on top of this!

Coding time #2

Graph kernels - pros and cons

Pros:

- complex subgraph information
- complex feature interactions
- very flexible & powerful
- often very good results
- can easily incorporate discrete labels

Cons:

- computational & memory complexity
- not scalable
- often can't incorporate continuous features

Further reading

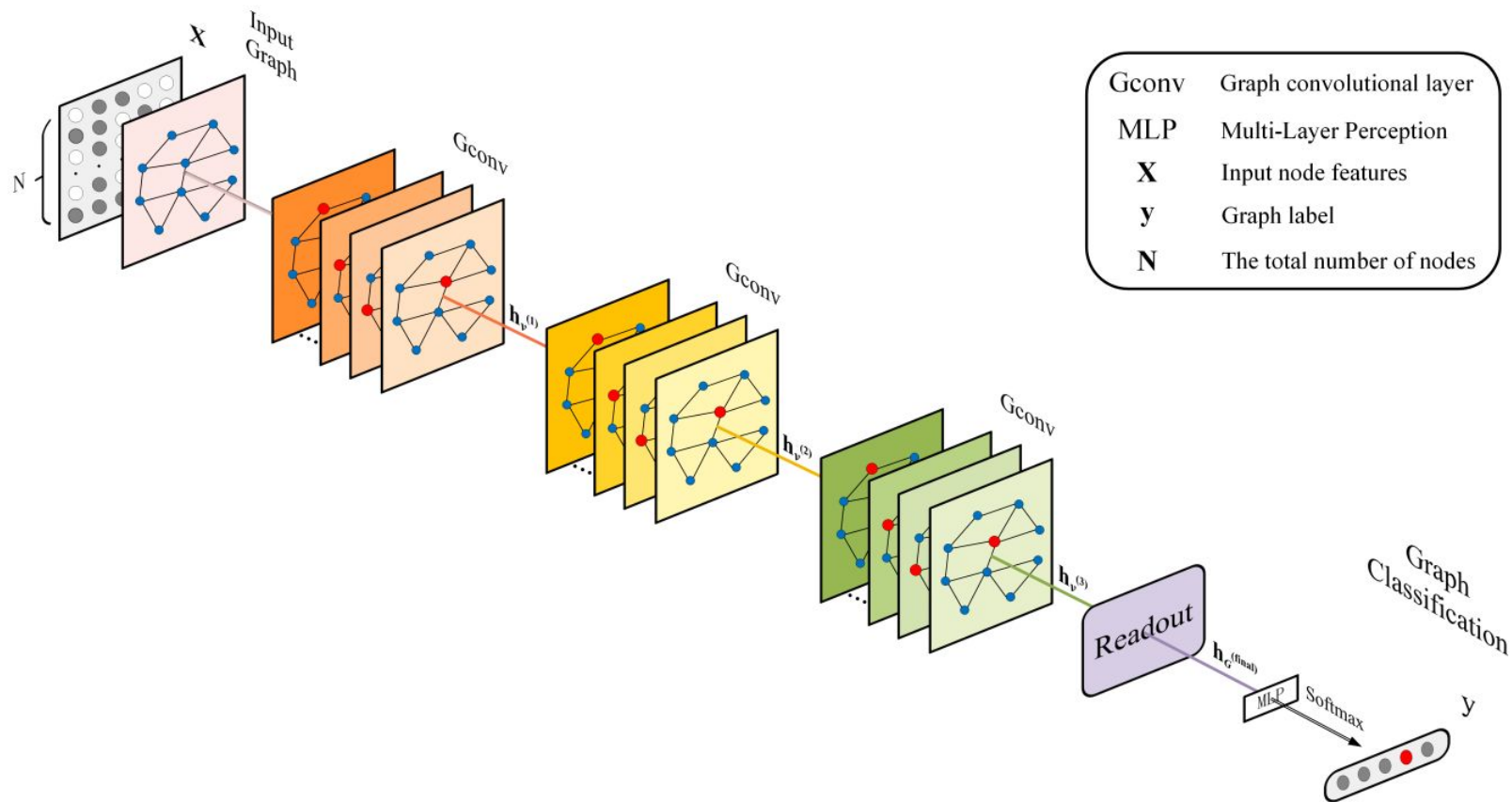
- Borgwardt, K., Oliver, S. *"An Introduction to Graph Kernels"*
- Borgwardt, K., et al. *"Graph kernels: State-of-the-art and future challenges"*
- Kriege, N. M., Johansson, F. D., Morris, C. *"A survey on graph kernels"*
- Shervashidze, N., et al. *"Weisfeiler-Lehman graph kernels"*

Graph neural networks (GNNs)

GNNs for graph classification

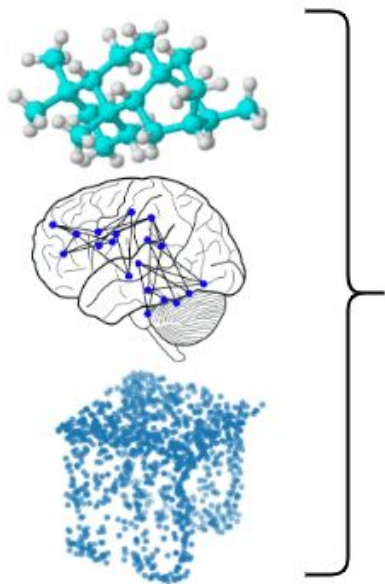
- modern ones are based on **message-passing paradigm** - very similar to WL-test!
- use a lot of **familiar** mechanisms:
 - **convolutions** - exchange information between neighbors & aggregate
 - **global pooling** - aggregate information for all nodes
 - **feedforward, backpropagation, gradient descent** - for training
- but introduce a lot of **new problems**:
 - oversmoothing, oversquashing -> hard to go deep
 - very unstable, easily overfit -> hard to train & evaluate
 - lack of data, lack of standardization -> hard to do anything

GNNs for graph classification



GNN input

- **adjacency matrix** - message passing structure, like in Weisfeiler-Lehman
- **node feature matrix** - input features, like image pixels or token embeddings
- GNNs still **require** feature engineering - for nodes!



Structure =
Adjacency matrix

$$A \in R^{N \times N}$$

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 |

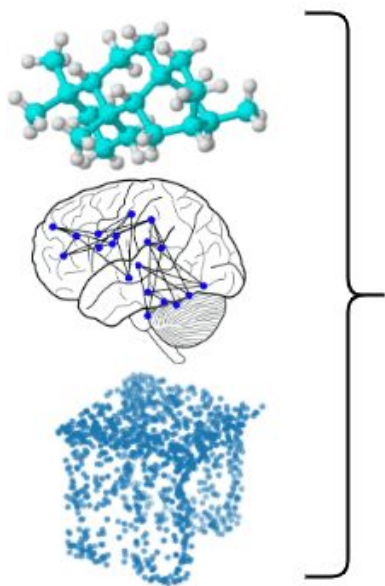
Graph Signal
(feature matrix)

$$X \in R^{N \times F}$$

node's feature vector

GNN input features

- typically **domain-specific node features**, e.g. atom type, formal charge
- for data with no features, e.g. social networks, can use e.g. constant 1s, node degree



Structure =
Adjacency matrix

$$A \in R^{N \times N}$$

| | | | |
|---|---|---|---|
| 1 | 0 | 1 | 0 |
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 |

Graph Signal
(feature matrix)

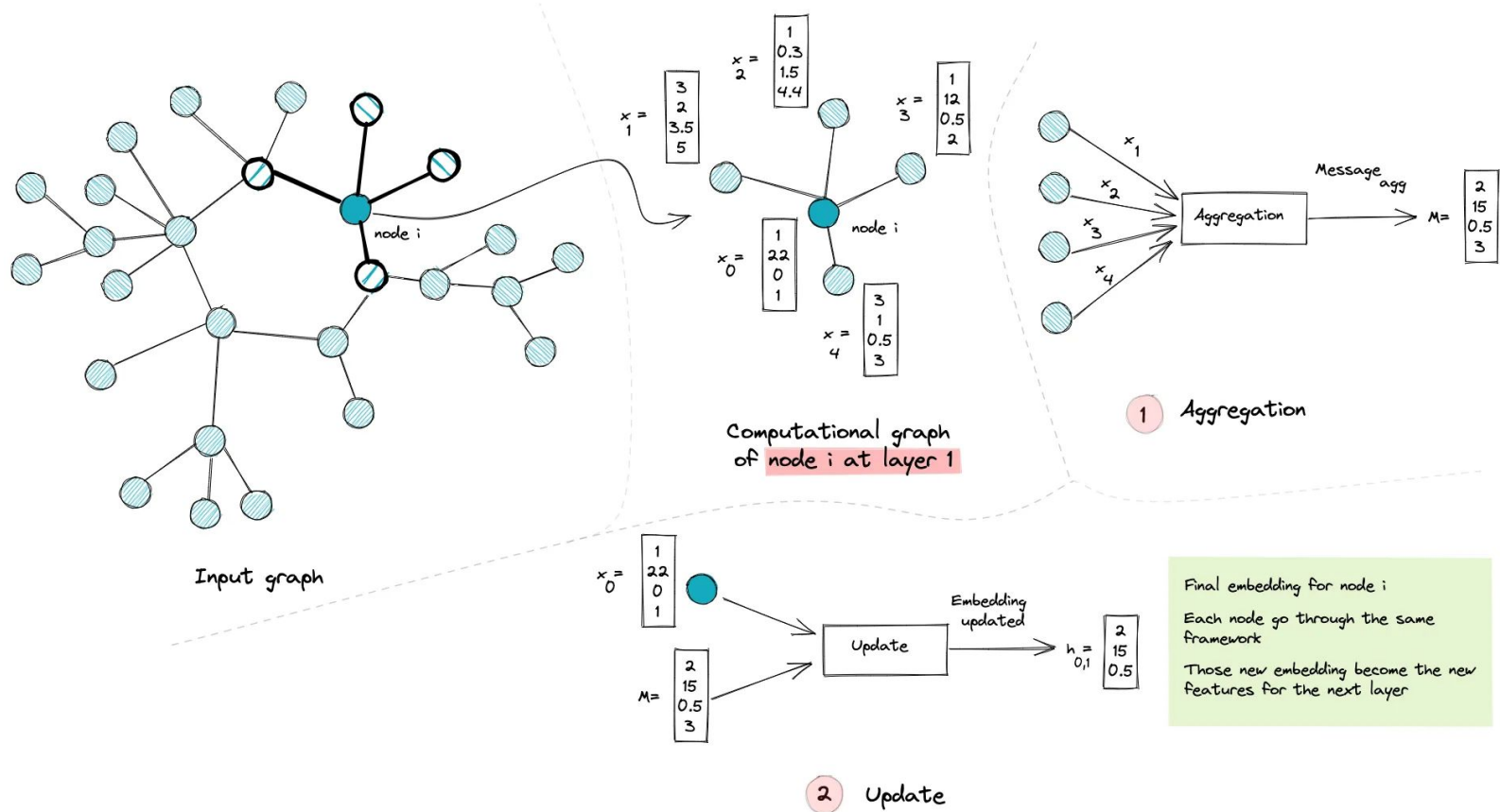
$$X \in R^{N \times F}$$

| | | | | | | | | | |
|-----------------------|--|--|--|--|--|--|--|--|--|
| node's feature vector | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |
| | | | | | | | | | |

How GNNs work

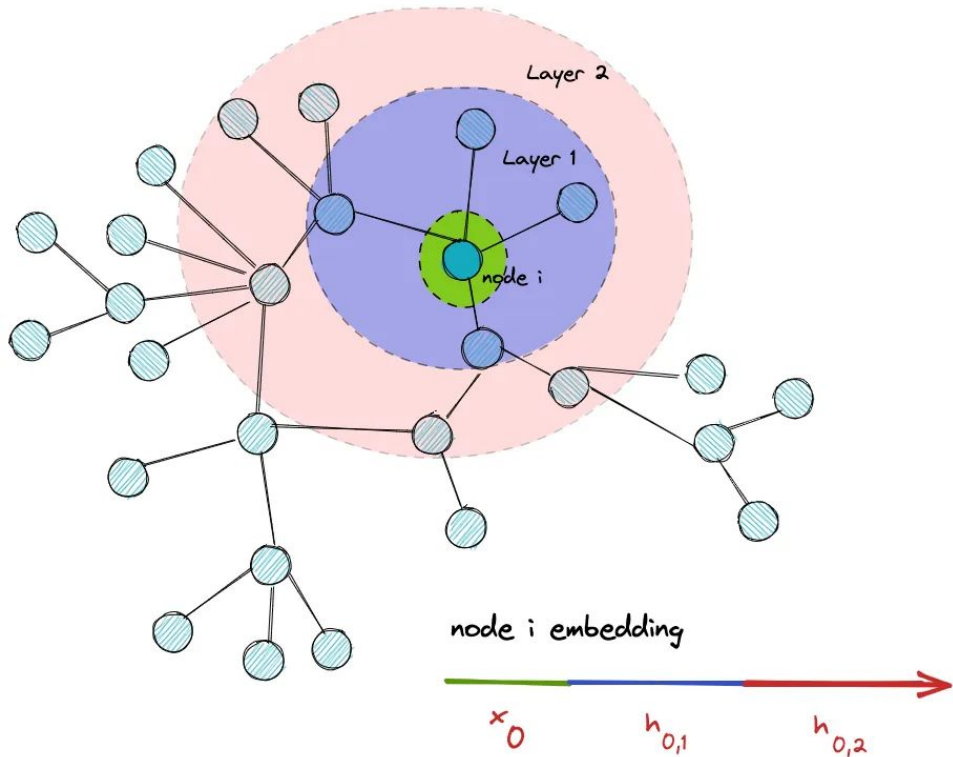
- each node starts with its **feature vector**, they become **embeddings**
- **very similar** to Weisfeiler-Lehman features
 - WL iteration ~ GNN layer
 - node gets **messages** from neighbors (their embeddings)
 - **aggregate** with own vector, to become new embedding
 - apply activation function
- **readout** layer:
 - to aggregate nodes into the graph embedding
 - **pool** each feature from all nodes, e.g. column-wise mean
- **classify** with MLP head, backpropagate

Message-passing in GNNs



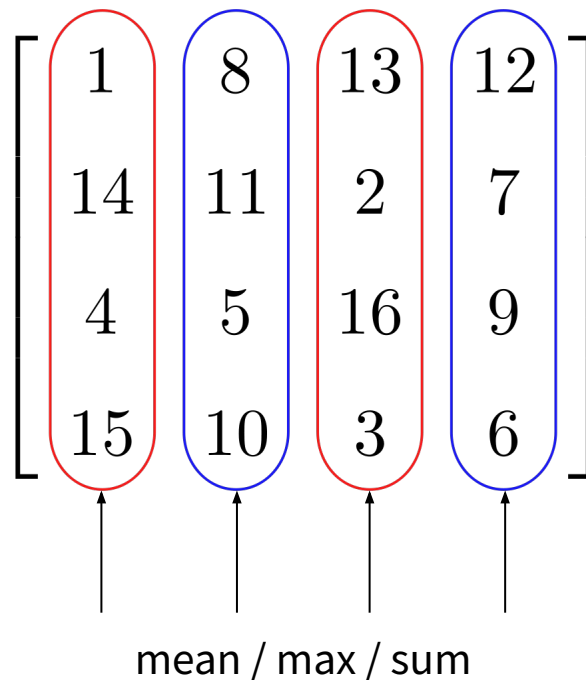
Message-passing in GNNs

- message exchange = gain information
- k layers = **k-hop neighborhood** information
- layers **smoothe** the representations over the neighborhood
- different GNNs differ in:
 - how to aggregate
 - how to update node with neighbors' information
- those are called AGGREGATE and COMBINE functions



Readout

- need to **aggregate** all nodes, after last convolution and before MLP
- **simple readout:**
 - mean, max, sum
 - applied **feature-wise**, i.e. for each column in node embeddings
- can use **multiple**, e.g. by concatenation
- there are **more sophisticated** ones, e.g. set learning, attention-based



Message-Passing Neural Networks (MPNN)

- Gilmer, J., et al. *"Neural message passing for quantum chemistry."*
- **general framework** of message passing in GNNs
- explicitly define AGGREGATE and COMBINE steps
- intuitive, similar to WL approach
- **beware:** there's also architecture, MPNN, proposed in the same work

Graph kernels - pros and cons

Pros:

- complex feature interactions
- processes small subgraphs very well
- scalability
- very flexible & powerful
- can achieve very good results

Cons:

- doesn't explore graph topology well
- long training, requires GPUs
- overfits very easily
- very unstable training
- oversmoothing, oversquashing, underreaching, ...

Spectral graph convolution

Graph convolutions are hard!

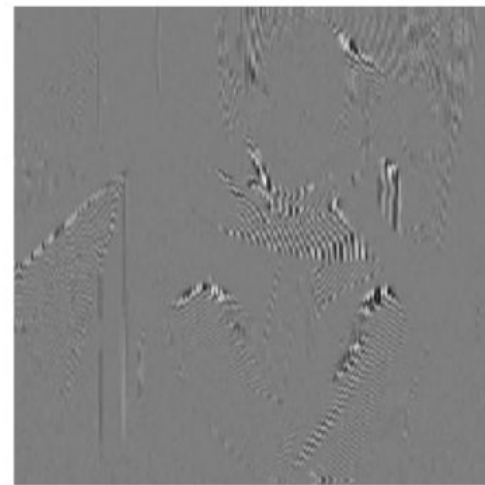
- **no one, unique definition of convolution on graphs**
- different definitions and approaches, with pros & cons
- **problems:**
 - keep permutation invariance
 - different neighborhood size
 - what even should convolutional filter look like?
 - nodes are not equally important, e.g. degree, amount of information
 - stable learning, so we can stack layers
- **Source:** N. Keriven "Graph Neural Networks: Introduction, some theoretical properties" ([link](#))

2D convolutions

- **local** pattern matching operator
- **filter** moves **across** input in spatial dimension
- filter has **constant size**, but consists of **learnable weights**

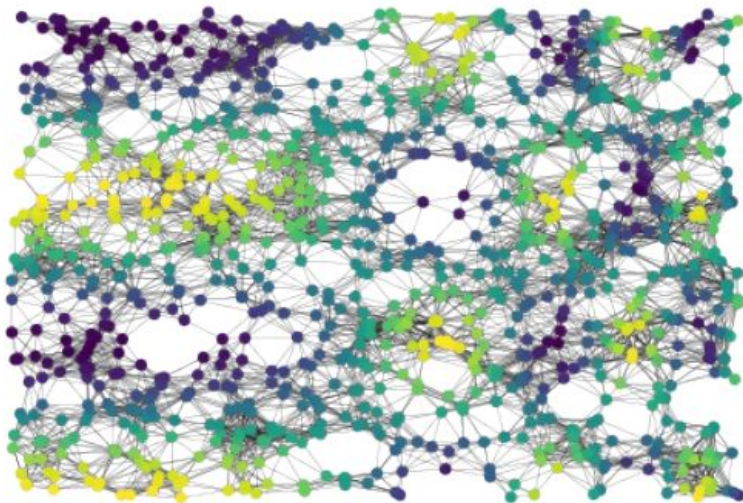


$$(x \star h)[n] = \sum_r h[r]x[n - r]$$



Graph convolutions

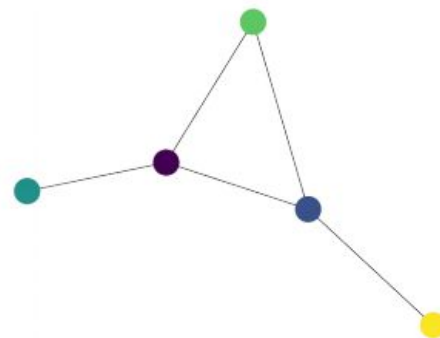
- **local** - 1-hop / k-hop neighborhood
- moving **across** input - send messages across edges
- filter has to work with **variable size input**



$$z \in \mathbb{R}^n \text{ on } G$$

?

★



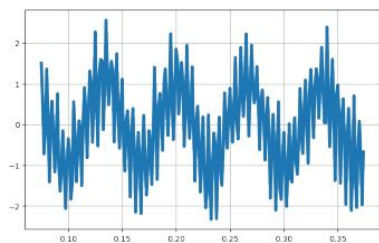
$$h \in \mathbb{R}^p \text{ on } H$$

Convolution theorem to the rescue!

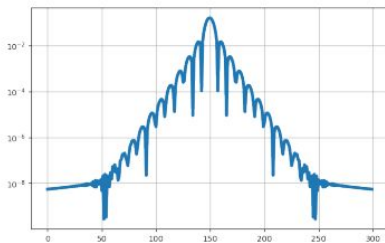
Fourier transform of the **convolution** in the spatial domain is equal to **multiplication** in the **Fourier (frequency)** domain. \longrightarrow

1. Define Fourier transform (and its inverse) for graphs
2. Transform, multiply, inverse

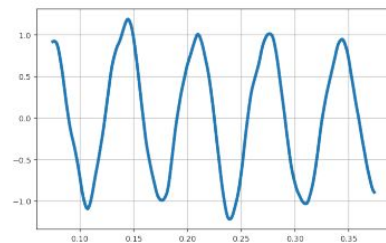
$$\mathcal{F}(x \star h) = \mathcal{F}(x) \cdot \mathcal{F}(h)$$



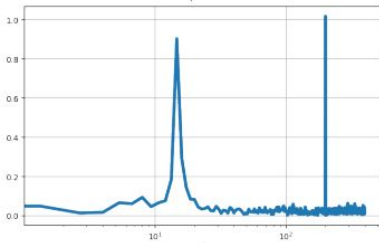
\star



$=$

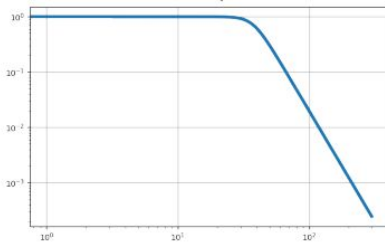


$\mathcal{F} \downarrow$

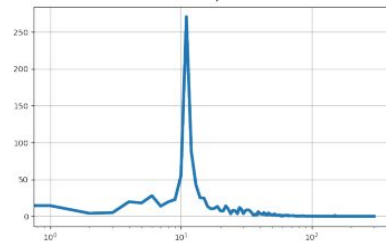


\times

$\mathcal{F} \downarrow$



$=$



$\uparrow \mathcal{F}^{-1}$

Fourier transform on graphs

- Fourier transform:

$$\mathcal{F}f(\omega) = \int f(t)e^{-2i\pi\omega t}dt = \langle f, e^{-2i\pi\omega\cdot} \rangle_{L^2}$$

- how to do this on graphs - **spectral graph theory**
- **eigendecomposition** of the **graph Laplacian** can be used instead

$$L = D - A$$

D - degree matrix, A - adjacency matrix

- represents the graph in the **frequency (spectral)** domain
- hubs (large degree nodes) dominate it & can cause numerical instability
- typically we use **normalized Laplacian** instead

Normalized graph Laplacian

- most common is **symmetrically normalized Laplacian**:

$$L_{sym} = I - D^{-1/2} A D^{-1/2} = D^{-1/2} L D^{-1/2}$$

where:

$$D^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{\deg(v_1)}} & 0 & \dots & 0 \\ 0 & \frac{1}{\sqrt{\deg(v_2)}} & \dots & 0 \\ 0 & \vdots & \ddots & 0 \\ 0 & 0 & \dots & \frac{1}{\sqrt{\deg(v_n)}} \end{bmatrix}$$

- so common, it's often written simply as L

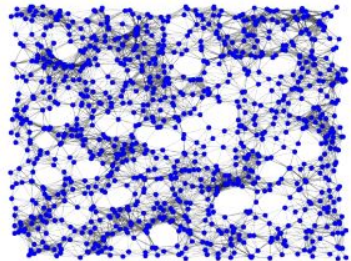
Graph Laplacian eigendecomposition

- eigendecomposition (L - normalized Laplacian):

$$L = U \Lambda U^T$$

- all real numbers (for undirected graphs)
 - all eigenvalues in range $[0, 2]$ (thanks to normalization)
- **a lot of information**, e.g.:
 - connected components
 - node homophily
 - general topology

Graph Laplacian eigendecomposition

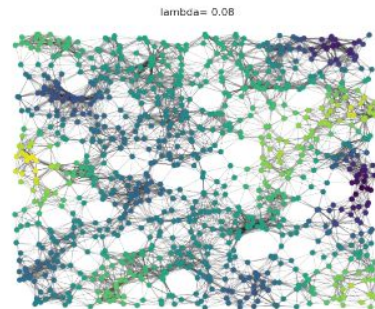
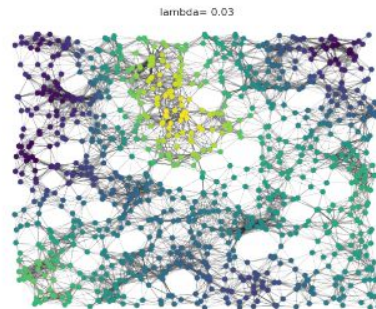
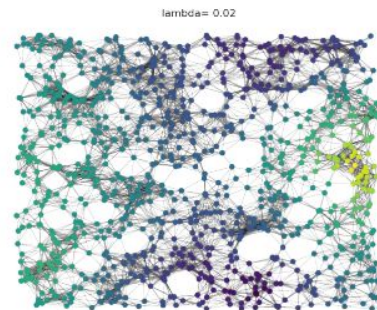
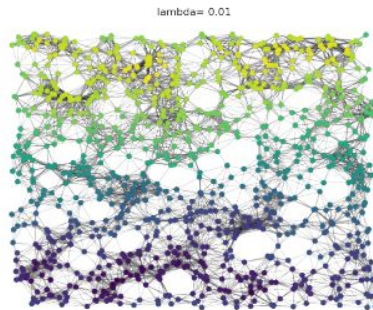
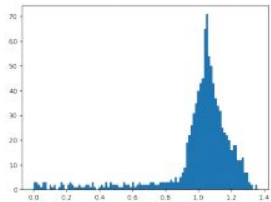


Diagonalize the Laplacian:

$$L = U \Lambda U^T$$

$$\begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}$$

$[u_1, \dots, u_n]$ Eigenvectors: "Fourier modes"



Eigenvalues: "frequencies"

$$0 = \lambda_1 \leq \dots \leq \lambda_n \leq 2$$

- **Fourier transform** $\mathcal{F}z = U^T z$
- **Inverse Fourier transform** $\mathcal{F}^{-1} \tilde{z} = U \tilde{z}$

Spectral graph convolution

- defined as:

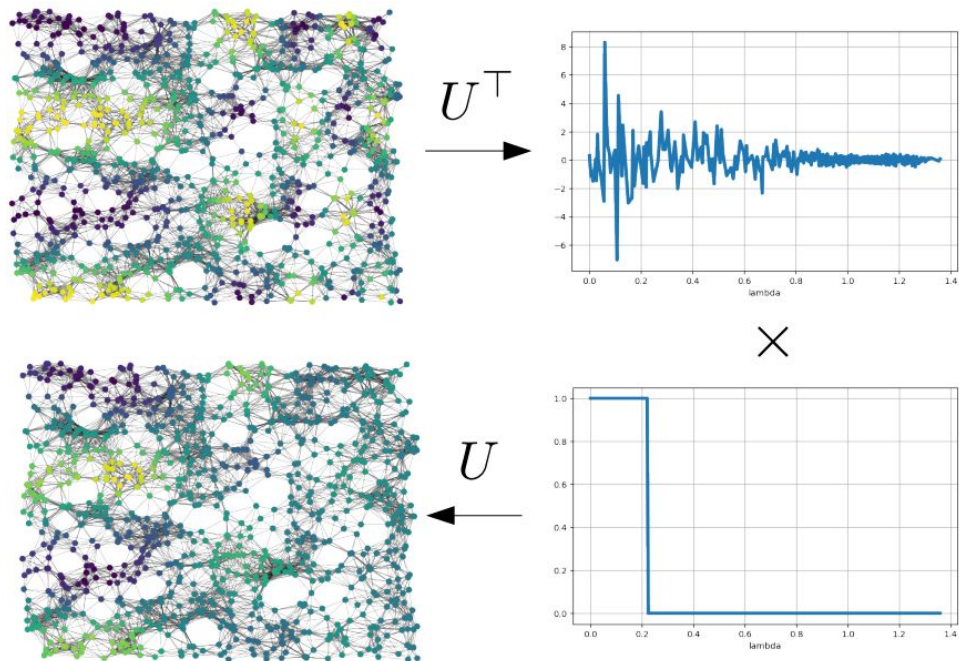
$$g_{\theta} * x = U g_{\theta} U^T x$$

where:

- $x \in \mathbb{R}^N$ - input signal, scalar per node
- $g_{\theta}(\Lambda) = \text{diag}(\theta)$, $\theta \in \mathbb{R}^N$
Convolutional filter, based on eigenvalues
- $U \in \mathbb{R}^{|V| \times |V|}$ - eigenvectors

Fourier transform $\mathcal{F}z = U^T z$

Inverse Fourier transform $\mathcal{F}^{-1}\tilde{z} = U\tilde{z}$



Spectral graph convolutional network

- Henaff, M., Bruna, J., LeCun, Y. *"Deep convolutional networks on graph-structured data"*
- defined as:

$$x^{(l+1)} = \sigma \left(b^{(l)} 1_n + \sum_i g_i^{(l)} * x_i^{(l)} \right) \quad g_\theta * x = U g_\theta U^T x$$

i - number of input filters (features)

- graph topology **does not change** between layers, only features
- we calculate eigendecomposition only once
- **parameters count:** $|V|$ per filter

Problems with spectral graph convolution

- **computational & memory complexity:**
 - eigendecomposition is $O(|V|^3)$
- **doesn't generalize well:**
 - Fourier transform is an exact basis, causes overfitting
- **not localized:**
 - processes whole graph as signal at once
 - unlike well-known convolution on image

Graph Convolutional Network (GCN)

Approximate!

- **approximation:** simplifies computation, reduces number of parameters, regularizes
 - Hammond, D.K., Vandergheynst, P., Gribonval, R. *"Wavelets on graphs via spectral graph theory"*
 - Kipf, T.N., Welling, M. *"Semi-supervised classification with graph convolutional networks"*
- **ideas:**
 - approximate eigendecomposition with Chebyshev polynomials of degree K
 - limit $K=1$
 - reduce parameters even more
 - add self-loops, i.e. edges from each node to itself, to avoid exploding gradient
- **result:** highly regularized, localized, spectral convolution

Approximating graph convolution

- approximate convolution with **Chebyshev polynomials** of order K :

$$g_{\theta'} * x \approx \sum_{k=0}^K \theta'_k T_k(\tilde{L})x$$
$$T_0(x) = 1, \quad T_1(x) = x$$
$$\tilde{L} = \frac{2}{\lambda_{max}}L - I$$

- set $K = 1$, $\lambda_{max} = 2$, $\theta'_0 = -\theta'_1$ and simplify even further:

$$g_{\theta'} * x \approx \theta(I + I - L)x$$

- observation: we can replace the Laplacian $L = I - D^{-1/2}AD^{-1/2}$:

$$g_{\theta'} * x \approx \theta(I + D^{-1/2}AD^{-1/2})x$$

Approximating graph convolution

$$g_{\theta'} * x \approx \theta(I + D^{-1/2}AD^{-1/2})x$$

- **renormalization trick** - add **self-loops** to stabilize training (lower eigenvalues):

$$\tilde{A} = A + I \quad I + D^{-1/2}LD^{-1/2} \rightarrow \tilde{D}^{1/2} \tilde{A} \tilde{D}^{1/2}$$

$$g_{\theta'} * x \approx \theta(\tilde{D}^{1/2} \tilde{A} \tilde{D}^{1/2})x$$

- **parameters count:** 1 per filter
- expand this for C input channels (features) and F filters in parallel:

$$H^{(l+1)} = \sigma(\tilde{D}^{1/2} \tilde{A} \tilde{D}^{1/2} H^{(l)} \Theta^{(l)})$$

- **spectral convolution**, but very strongly simplified, regularized and 1-localized
- stack to get k-hop neighborhood

GCN as spatial convolution

- **spatial convolution**, since it's 1-localized around each node
- **message passing** - passes feature vectors between nodes:

$$h_v^{(l+1)} = \sigma \left(W^{(l)} * \left(h_v^{(l)} + \sum_{u \in N(v)} \frac{h_u^{(l)}}{\sqrt{d(u)d(v)}} \right) \right)$$

- AGGREGATE: sum node features with neighbors (weighted with degrees)
 - COMBINE: multiply with weight matrix
- this is equal to locally **smoothing** features between neighbors, encouraging similar predictions for close nodes
- can be loosely interpreted as **continuous, differentiable** Weisfeiler-Lehman features with learnable weights

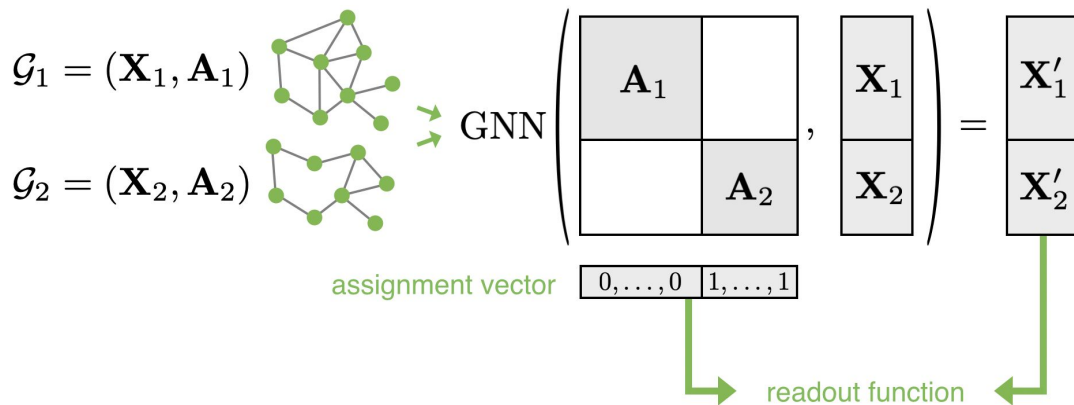
GCN for graph classification

- GCN was made for:
 - **node classification** - one large graph, node homophily
 - **transductive learning** - train & test nodes are in the graph, only test labels hidden
- but works well for graph classification!
 - Dwivedi, V. P., et al. *"Benchmarking graph neural networks"*
 - Adamczyk, J. *"Application of Graph Neural Networks and graph descriptors for graph classification"*
 - often on par with much more sophisticated architectures
- needs **readout function**

Minibatching

- we can also do **minibatch** training for multiple graphs, by concatenating matrices:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_1 & & \\ & \ddots & \\ & & \mathbf{A}_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} \mathbf{X}_1 \\ \vdots \\ \mathbf{X}_n \end{bmatrix}, \quad \mathbf{Y} = \begin{bmatrix} \mathbf{Y}_1 \\ \vdots \\ \mathbf{Y}_n \end{bmatrix}$$



- treats graph as collection of disjoint components
- if we have message-passing, we can more easily **parallelize** and **scale** on low level

Coding time #3

Other graph convolutions

Graph convolutional networks

- **GCN** is a particular architecture with a given convolution
- there is **a lot of** graph convolutions - currently **over 60** in PyTorch Geometric!
 - more expressive AGGREGATE and COMBINE
 - attention, gating
 - more sophisticated spectral filtering
 - specialized, e.g. node classification, link prediction, point clouds, knowledge graphs
- we will talk about **3 very well-known ones**:
 - GraphSAGE - spatial convolutional
 - GIN - strong theory & ties with Weisfeiler-Lehman test
 - GAT - attentional

GraphSAGE

- Hamilton, W., Ying, Z., Leskovec, J. *"Inductive representation learning on large graphs"*
- **idea:**
 - explicitly model AGGREGATE and COMBINE as **separate** steps
 - add **skip connection** to allow using node itself separately from neighbors

- **AGGREGATE** neighbors:

$$h_{N(v)}^{(l+1)} = AGG^{(l+1)}(\{h_u^{(l)}, \forall u \in N(v)\})$$

- **COMBINE** - concatenate node with aggregated neighbors and combine:

$$h_v^{(l+1)} = \sigma \left(W^{(l)} \cdot \left[(h_v^{(l)}, h_{N(v)}^{(l+1)}) \right] \right)$$

- concatenation is inspired by ResNet
- it allows to somewhat ignore neighbors if needed

GraphSAGE aggregations

- **mean:**

- average neighbors' features
- similar to GCN, but note that we concatenate with node itself later

$$h_{N(v)}^{(l+1)} = \text{MEAN}(\{h_u^{(l)}, \forall u \in N(v)\})$$

- **max:**

- 1-layer MLP with max pooling for each feature
- additional weights = learn more complex interactions

$$h_{N(v)}^{(l+1)} = \max(\{\sigma(W_{pool}^{(l)} h_u^{(l)} + b), \forall u \in N(v)\})$$

- **LSTM** - randomly permute neighbors and aggregate with LSTM; not strictly invariant

Graph Isomorphism Network (GIN)

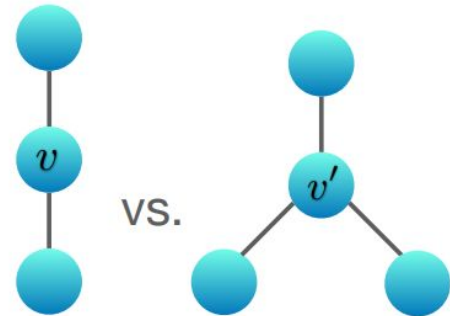
- Xu, K., et al. *"How powerful are graph neural networks?"*
- **idea:**
 - analyze theoretically power of GNNs in **distinguishing graphs**
 - are 2 node neighborhoods the same, i.e. isomorphic (including features)?
- **results:**
 - **all** message-passing GNNs are **at most as powerful** as WL-test in distinguishing graphs
 - most GNNs are **less** powerful than WL-test, i.e. can't even tell if 2 graphs are identical!
- **does it matter?**
 - GNNs are still more powerful than WL-test in actual **classification performance**
 - continuous features, task-relevant feature extraction, more sophisticated processing

Graph Isomorphism Network (GIN)

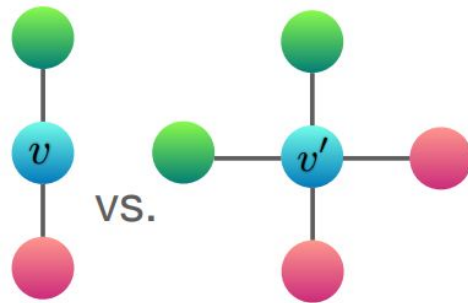
- **idea:** design graph convolution as powerful as WL-test
- authors **prove** that we need:
 - AGGREGATE = **sum** of neighbors' features
 - COMBINE = **2-layer** MLP
 - READOUT = **sum** pooling

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

- **why is that?**
 - prove that sum distinguishes graphs, where mean and max fail
 - 2-layer MLP is a powerful discriminator



(a) Mean and Max both fail



(c) Mean and Max both fail

Graph Attention Network (GAT)

- Veličković, P., et al. *"Graph attention networks"*
- **idea:**
 - neighbors have different importance based on structure, features, network layer etc.
 - use **attention** to weight neighbors, with learnable weights
 - **self-attention** with self loops, to weight node itself separately
- convolution is a weighted sum of neighbors (with self-loop), using attention weights α

$$h_u^{(l+1)} = \sigma \left(\sum_{v \in N(u) \cup \{u\}} \alpha_{uv} W^{(l)} h_v^{(l)} \right)$$

- attention can be anything - paper uses Bahdanau attention

GAT - attention weights

- **attention score:**

$$e_{uv} = \text{attention}(h_u, h_v) = \text{LeakyReLU} \left(a^T [W h_u, W h_v] \right)$$

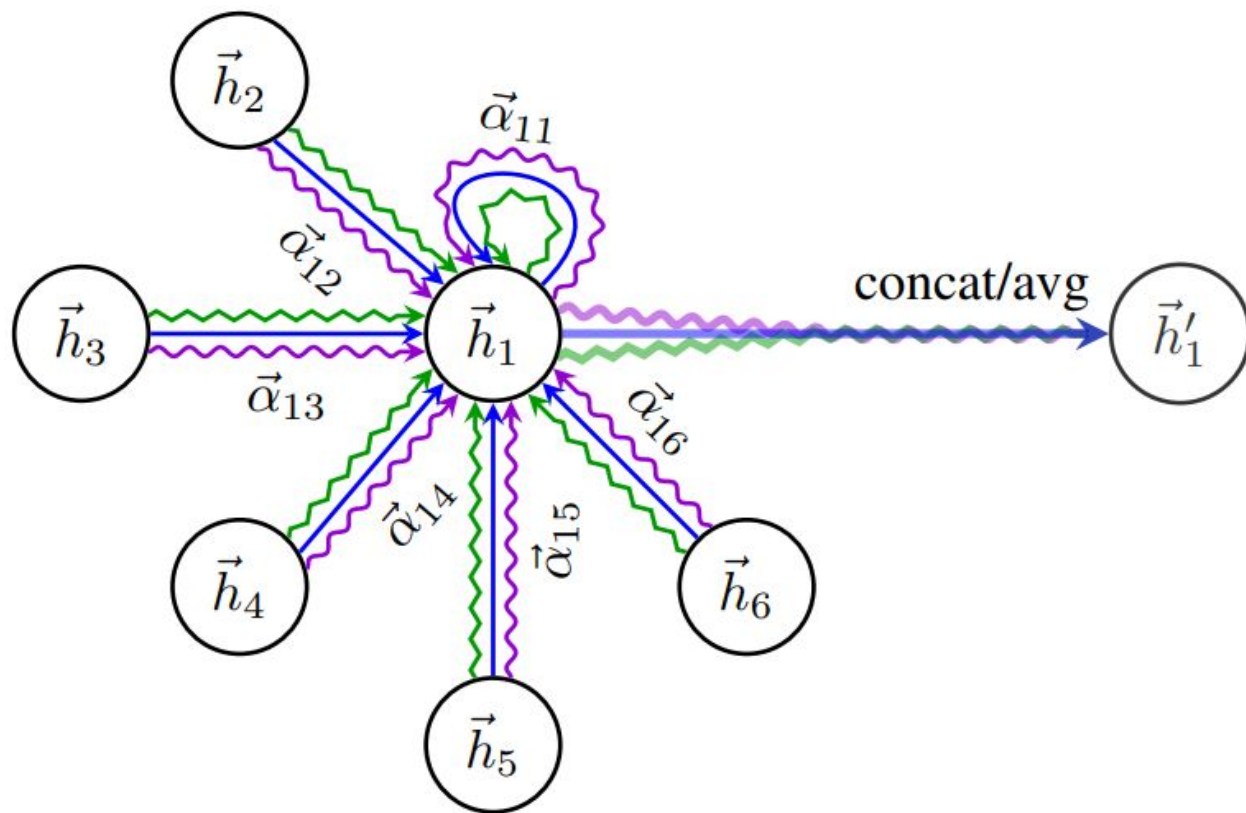
- **attention learnable parameters:** matrix W , shared for all nodes in a layer
- concatenate node and neighbor
- **attention non-linearity:** 1-layer MLP (LeakyReLU + parameters vector a)

- **attention weights** are softmax-normalized scores (for all neighbors):

$$\alpha_{uv} = \text{softmax}_v(e_{uv}) = \frac{\exp(e_{uv})}{\sum_{v \in N(u) \cup \{u\}} \exp(e_{uv})}$$

- can use **multihead attention** - do the same in parallel K times and concatenate / average neighbors

GAT - multihead attention



Coding time #4

Fair evaluation

Beware of reported results!

- **small datasets** are an old problem for graphs, and are often **too easy** or unrealistic
- **proper evaluation** requires **repeated** training and validating models, e.g.:
 - 10-fold CV for testing
 - retraining with random splits & seeds
- **unfortunately:**
 - many people got lazy!
 - using only 10-fold CV for validation and reporting validation results
 - using weird techniques & modifications
 - only tuning their own model, or using different evaluation for different models
- **do not** trust results outright, always verify

Approaches to fair evaluation

- **how to test:**
 - 10-fold cross-validation, or a lot of random splits
 - use realistic and challenging split, e.g. scaffold split for molecules
- **how to choose hyperparameters:**
 - validation set separate from test set!
 - nested CV or holdout, use the same approach, budget etc. for all algorithms
- **proper datasets** - modern, large, challenging
- **strong baselines** - include graph feature engineering (e.g. LTP) and graph kernels as baselines

Fair evaluation & benchmarks

- Errica, F., Podda, M., Bacciu, D., Micheli, A. *"A fair comparison of graph neural networks for graph classification"*
- Dwivedi, V. P., et al. *"Benchmarking graph neural networks"*
- Shchur, O., et al. *"Pitfalls of graph neural network evaluation"*
- [Open Graph Benchmark](#)
- [Therapeutics Data Commons](#)

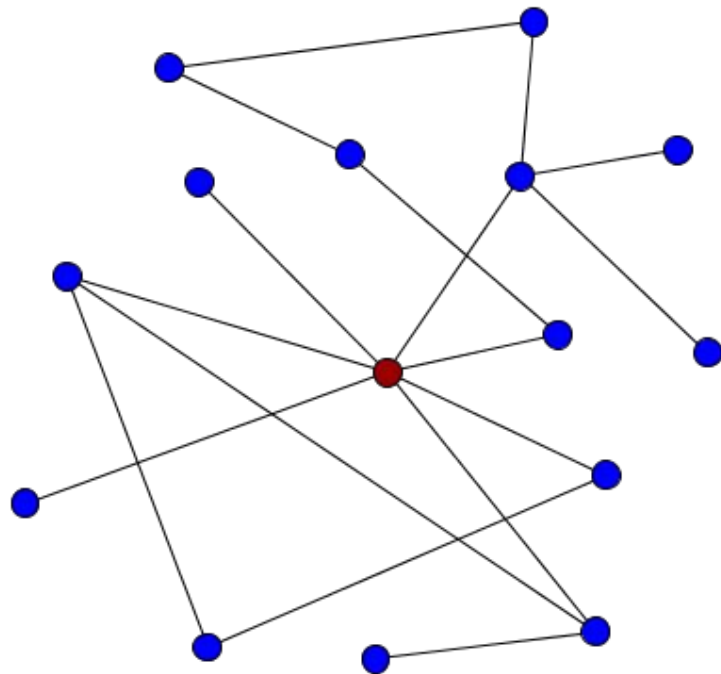
Oversmoothing and oversquashing

Oversmoothing problem

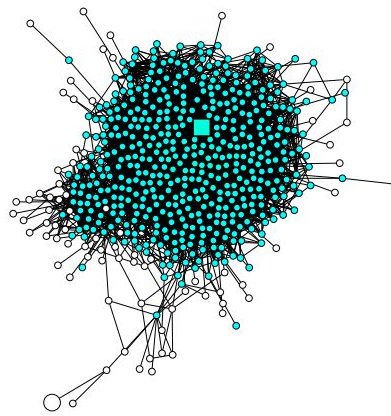
- each layer **smooths** features:
 - exchange information in a k-hop neighborhood - **good**
 - nearby nodes become more similar, good for **homophilic** graphs
- **oversmoothing:**
 - when nodes become too similar, performance degrades
 - big problem for **heterophilic** graphs, where long-range dependencies matter most
 - forces **trade-off** between extracting local & global patterns
- **we need many layers:**
 - long-range dependencies, e.g. chemical rings, large subgraphs
 - otherwise we risk **underreaching** problem

Oversquashing problem

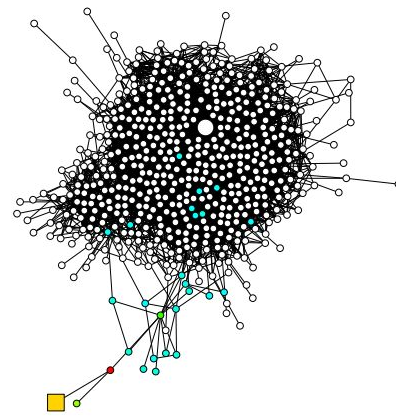
- we pass messages along **all edges**
- each node gets $\deg(v)$ messages **per layer**
- **oversquashing:**
 - more "central" nodes get too much information
 - **bottlenecks** - information can't easily travel further those nodes
 - long-range messages become **distorted**
- GraphSAGE, GAT - skip connection / attention can "quiet down" neighbors
- **we want:** only important and useful information in each layer



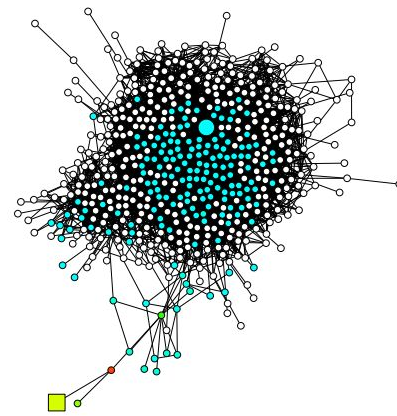
Oversmoothing & oversquashing



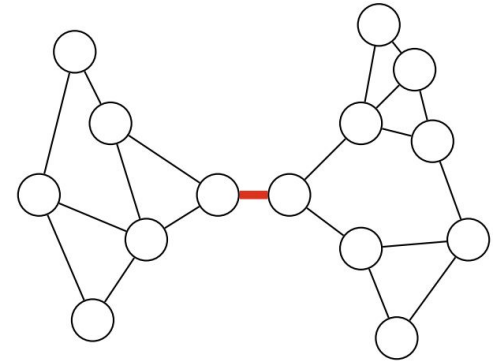
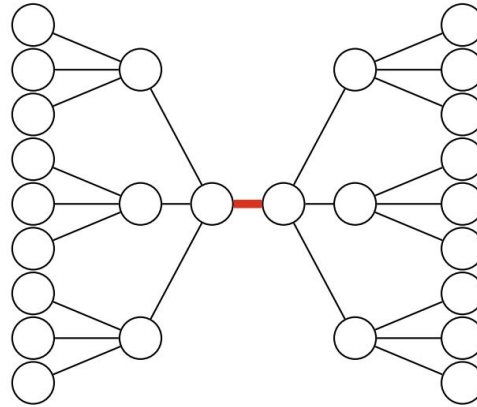
(a) 4 steps at core



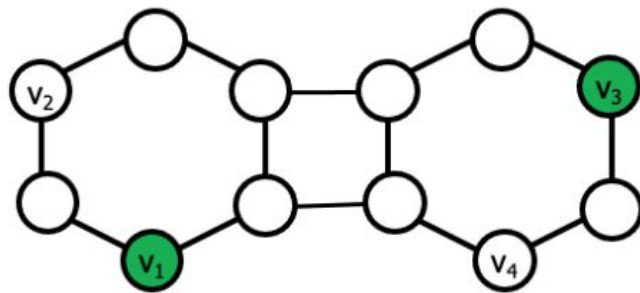
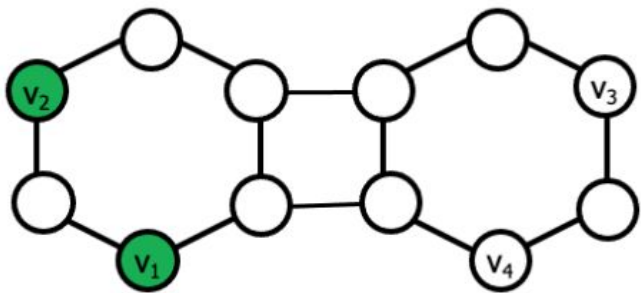
(b) 4 steps at tree



(c) 5 steps at tree

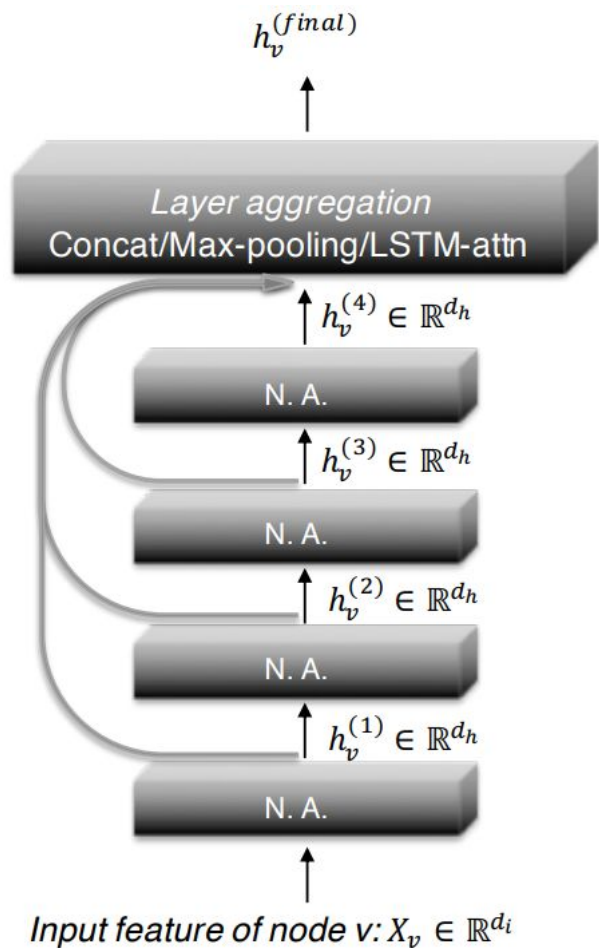


Oversmoothing & underreaching



Jumping Knowledge (JK) networks

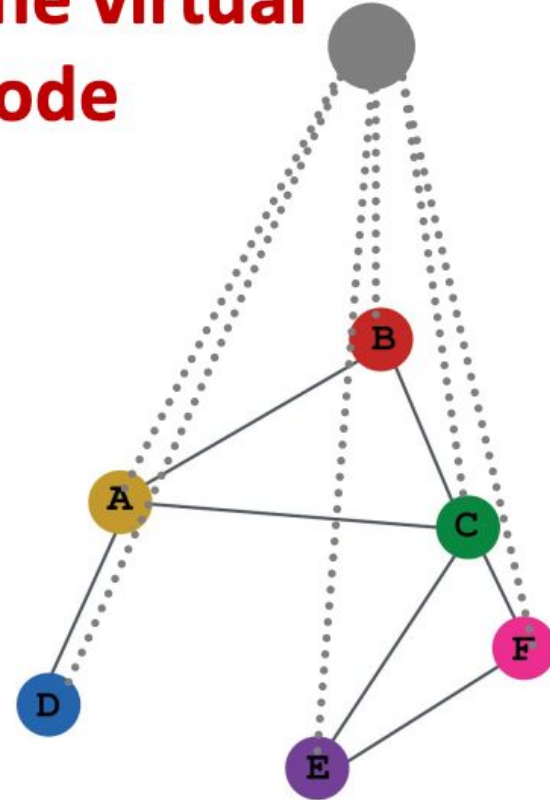
- Xu, K., et al. "Representation learning on graphs with jumping knowledge networks"
- **idea:** combine representations from all layers at the end, by using **skip connections**
- allows MLP to **adaptively choose** from node embeddings, e.g. different smoothing level, local vs global information
- **how to combine:**
 - concatenate
 - max-pooling (for features, column-wise)
 - weight with attention (via LSTM)
- **very common**, e.g. GIN paper uses JK with concatenation



Virtual node

- Pham, T., et al. *"Graph classification via deep learning with virtual nodes"*
- **idea:** shorten long-range dependencies with additional "virtual" node
 - connected to all others
 - "shortcut" for long dependencies
 - all zero features
- often **improves results**, especially for complex, heterophilic tasks, e.g. molecular property prediction
- virtual node **suffers** from oversquashing, however!

The virtual node



Message passing modifications

- **idea:**
 - modify how / where messages are passed in the graph
 - less messages = less information = reduce oversmoothing & oversquashing
- **graph rewiring:**
 - change graph edge structure, e.g. add long-range edges, remove less important ones
 - one of key ideas behind graph transformers
- **reduce message passing**, e.g.:
 - pass messages only along shortest paths in k-hop neighborhood
 - interpret edges as directed and filter out some of them
- **very active research area** - extremely hard to do this right!

Further reading

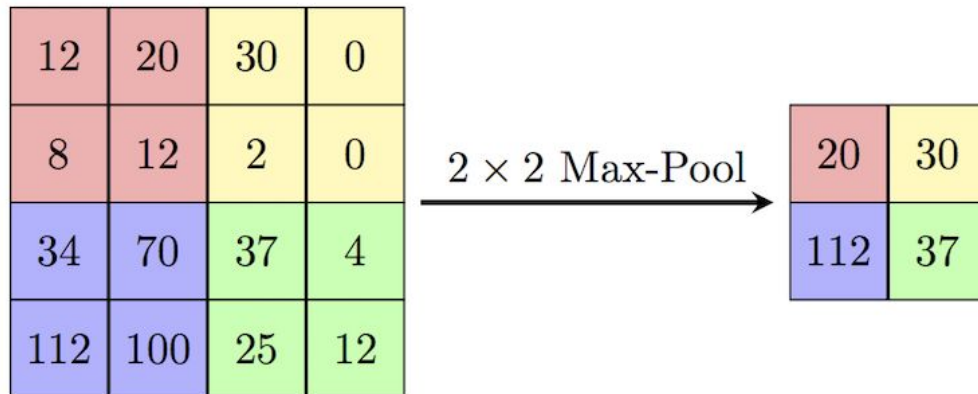
- Rusch, T. K., Bronstein, M. M., Mishra, S. *"A survey on oversmoothing in graph neural networks"*
- Alon, U., Eran Y.. *"On the bottleneck of graph neural networks and its practical implications"*
- Di Giovanni, Francesco, et al. *"How does over-squashing affect the power of GNNs?"*
- Nguyen, Khang, et al. *"Revisiting over-smoothing and over-squashing using Ollivier-Ricci curvature"*
- [Tutorial on Graph Rewiring: From Theory to Applications in Fairness](#)
- Abboud, R., Dimitrov, R., Ceylan, I. I. *"Shortest path networks for graph property prediction"*
- Yang, K., et al. *"Analyzing learned molecular representations for property prediction"*
- Deac, A., Lackenby, M., Veličković, P. *"Expander graph propagation"*
- Azabou, Mehdi, et al. *"Half-Hop: A graph upsampling approach for slowing down message passing"*

Coding time #5

Graph pooling

Pooling in CNNs

- CNNs typically **pool** nearby values every few layers, especially for image classification
- **goals:**
 - filter out unnecessary information
 - aggregate more global features
 - make smaller & more efficient

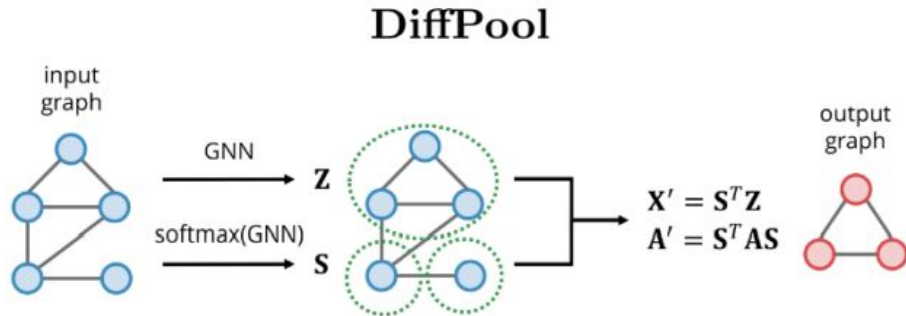


Pooling on graphs

- **how to pool?**
 - "nearby" nodes - the same problem as with convolution
 - local features can still be very important!
- we need **adaptive** and **learnable pooling**:
 - learn subgraphs structure
 - detect neighborhoods & substructures
 - task-adaptive, try not to remove important information
- **very hard**:
 - very active research area
 - no pooling is often better

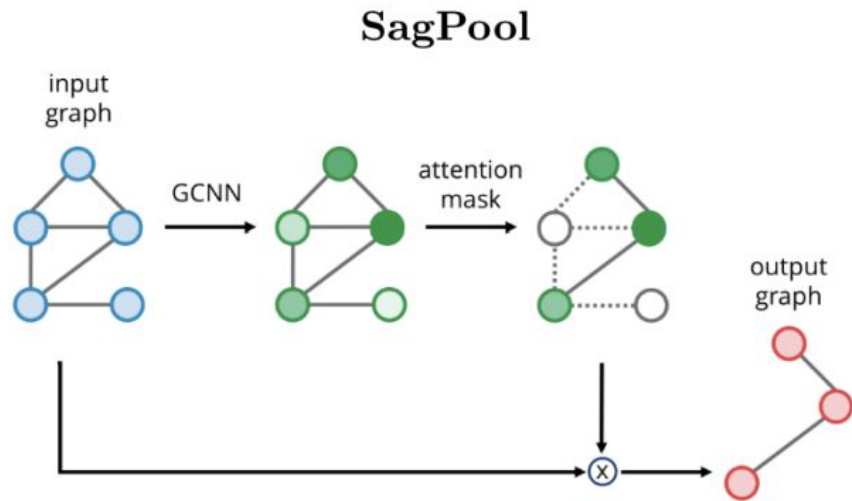
Pooling via graph coarsening

- "traditional" way, based on **node clustering**
- clustered nodes form "higher-level" graph
- typically **soft clustering**, i.e. node can be part of many clusters
- typically better results, but can be costly
- **main challenge:** how many clusters, and how to learn the clustering?
- e.g. DiffPool, MinCutPool, k-MIS



Pooling via node selection

- **remove weak nodes** from the graph
- proven to be less powerful than graph coarsening
- typically (much) worse results, but cheaper
- main challenges: do not remove important nodes, do not break graph
- e.g. Top-k, SAGPool, ASAPool



Graph pooling - current state

- **lot of papers, not much gain**
 - often pooling harms performance
 - bad evaluation procedures, lack of unified benchmarks
 - always compare with no pooling at all
- **graph coarsening seems to be better**
 - node selection became "popular" with buzzwords, e.g. self-attention
 - graph coarsening is now thought to be better direction
- **research is ongoing**
 - a lot of fascinating development here!

Graph pooling - further reading

- Liu, C., et al. *"Graph pooling for graph neural networks: Progress, challenges, and opportunities"*
- Bianchi, F. M., Lachi, V. *"The expressive power of pooling in graph neural networks"*
- Grattarola, D., et al. *"Understanding pooling in graph neural networks"*

Pretraining GNNs

Why pretrain?

- **incorporate general knowledge:**
 - e.g. large molecular datasets will expose models to general knowledge of chemistry
 - should give better results
- **less parameter to tune:**
 - reduce overfitting
 - faster
- can be interpreted as **smart weight initialization**, so at least shouldn't hurt

Pretraining is hard!

- **fully supervised pretraining:**
 - obtaining large annotated datasets is hard/costly
 - can overfit on pretraining task - requires (massively) **multi-task learning**
- **self-supervised pretraining:**
 - e.g. context prediction, contrastive learning, node/edge/subgraph prediction
 - typically **generate** subgraphs in some way
 - need to define **objective** to learn generally useful features
- **node-level vs graph-level pretraining:** we need to pretrain both to be effective and learn underlying patterns
- **can be harmful:** if done wrong / not useful for the problem, results in **negative transfer**

Pretraining strategies

- Veličković, P., et al. *"Deep graph infomax"*:
 - **contrastive learning** on subgraphs
 - objective designed to learn **both local & global patterns** at the same time
- Hu, W., et al. *"Strategies for pre-training graph neural networks"*:
 - **2-step pretraining**, to align learned representations
 - node-level **context prediction** and , graph-level fully supervised learning
- Sun, R., Dai, H., Yu, A. W. *"Does GNN Pretraining Help Molecular Representation?"*
 - **benchmark** of different approaches
 - highlight need for **good pretraining data**: a lot of it, high quality, challenging
 - **both** self-supervised and fully supervised pretraining are needed

Further reading

- Xia, J., et al. *"A survey of pretraining on graphs: Taxonomy, methods, and applications."*
- Xie, Y., et al. *"Self-supervised learning of graph neural networks: A unified review."*
- Liu, Y., et al. *"Graph self-supervised learning: A survey."*

What we didn't talk about

Other graph representation learning tasks

- **node classification:**
 - Node2Vec, DeepWalk, ...
 - various GNNs, e.g. SGC, SSGC
- **link prediction:**
 - Adamic-Adar Index, Katz Index, ...
 - various GNNs, e.g. LightGCN, SEAL
- **unsupervised graph embedding:**
 - IGE, Graph2Vec, FEATHER, ...
 - various graph autoencoders (GAEs)
- **graph generation**, e.g. combine GNNs with RL, or use genetic algorithms on graphs

Data augmentation

- **node feature augmentation:**
 - add topological descriptors to nodes, adding powerful structural information
 - e.g. Laplacian eigenvectors, subgraph counts
- **generating new graphs:**
 - graph rewiring, perturbations, adversarial attacks
 - generative models, graph diffusion, rule-based generators,
- **a lot** of different approaches:
 - Ding, K., et al. *"Data augmentation for deep graph learning: A survey"*
 - Zhao, T., et al. *"Graph data augmentation for graph machine learning: A survey"*

Randomness and GNNs

- **random weights can work well:**

- GNNs can work shockingly well **with no training** of convolutional layers!
- due to ties with WL-test and message passing, noticed even in original GCN paper
- Huang, T., et al. *"You Can Have Better Graph Neural Networks by Not Training Weights at All: Finding Untrained GNNs Tickets."*

- **random features can also work well:**

- e.g. augmentation of features with random shuffling, or outright using random node features
- Feng, W., et al. *"Graph random neural networks for semi-supervised learning on graphs."*
- Abboud, R., et al. *"The surprising power of graph neural networks with random node initialization."*

Graph transformers

- **combine:**
 - customized message passing scheme
 - topological descriptors for encoding nodes
 - attention
 - self-supervised pretraining
- **very active research**, seems to really take off with new, large datasets
- work on scalability & incorporating complex topological information
- e.g. Graph Transformer, Graphormer, GraphGPS, Molecule Attention Transformer