Machine Learning on Graphs and Sets

G. Nikolentzos and M. Vazirgiannis

LIX, École Polytechnique

ALTEGRAD 2020-2021

Outline

Outline

Message Passing Neural Networks for Learning Node Representations

Consist of a series of message passing layers usually followed by one or more fully-connected layers

The message passing phase runs for T time steps and updates the representation of each vertex $\mathbf{h}_v^t$ based on its previous representation and the representations of its neighbors:

$$\mathbf{m}_v^{(t+1)} = \mathrm{AGGREGATE}\left(\left\{\mathbf{h}_u^{(t)} \middle| u \in \mathcal{N}(v)\right\}\right)$$

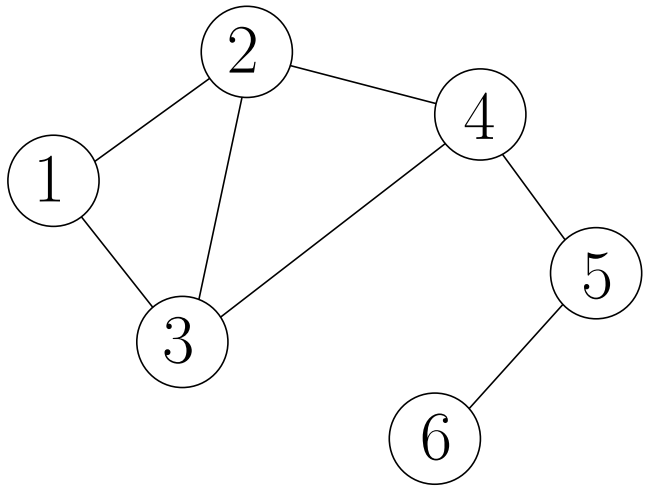
$$\mathbf{h}_v^{(t+1)} = \mathrm{COMBINE}\left(\mathbf{h}_v^{(t)}, \mathbf{m}_v^{(t+1)}\right)$$

where $\mathcal{N}(v)$ is the set of neighbors of v, and $\mathrm{AGGREGATE}$ and $\mathrm{COMBINE}$ are message functions and vertex update functions respectively

* a node's neighbors have no natural ordering
$\hookrightarrow$ the $\mathrm{AGGREGATE}$ function operates over an unordered set of vectors $\rightarrow$ must be invariant to permutations of the neighbors

Example of Message Passing Layer

$$\mathbf{h}_1^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_1^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_3^{(t)})$$
$$\mathbf{h}_2^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_1^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_3^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_4^{(t)})$$
$$\mathbf{h}_3^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_3^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_1^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_4^{(t)})$$
$$\mathbf{h}_4^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_4^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_3^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_5^{(t)})$$
$$\mathbf{h}_5^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_5^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_4^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_6^{(t)})$$
$$\mathbf{h}_6^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_6^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_5^{(t)})$$



Remark: Biases are omitted for clarity

Graph Convolutional Network (GCN)

Each message passing layer of the GCN model is defined as:

$$\mathbf{h}_v^{(t+1)} = \mathrm{ReLU}\left(\mathbf{W}^{(t)} \frac{1}{1+d(v)} \mathbf{h}_v^{(t)} + \sum_{u \in \mathcal{N}(v)} \mathbf{W}^{(t)} \frac{1}{\sqrt{(1+d(v))(1+d(u))}} \mathbf{h}_u^{(t)}\right)$$

where $d(v)$ is the degree of node v

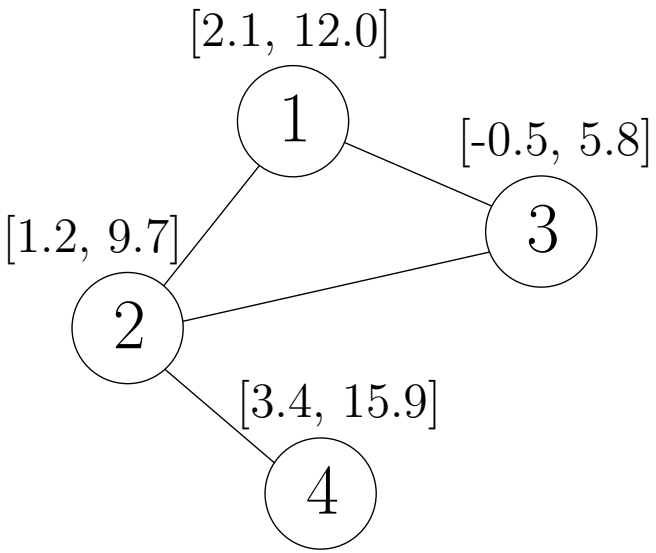
In matrix form, the above is equivalent to:

$$\mathbf{H}^{(t+1)} = \mathrm{ReLU}\left(\hat{\mathbf{A}}\, \mathbf{H}^{(t)}\, \mathbf{W}^{(t)}\right)$$

where $\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\, \tilde{\mathbf{A}}\, \tilde{\mathbf{D}}^{-\frac{1}{2}}$, $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ and $\tilde{\mathbf{D}}$ is a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_{j=1}^{n} \tilde{\mathbf{A}}_{ij}$

[Kipf and Welling, ICLR'17]

Example of Message Passing Layer of GCN (1/2)



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

$$\mathbf{X} = \begin{bmatrix} 2.1 & 12.0 \\ 1.2 & 9.7 \\ -0.5 & 5.8 \\ 3.4 & 15.9 \end{bmatrix}$$

We compute matrices $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{D}}$:

$$\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I} = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

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

Example of Message Passing Layer of GCN (2/2)

And then matrix $\hat{\mathbf{A}}$:

$$\hat{\mathbf{A}} = \tilde{\mathbf{D}}^{-\frac{1}{2}}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-\frac{1}{2}} = \begin{bmatrix} 0.333 & 0.288 & 0.333 & 0 \\ 0.288 & 0.25 & 0.288 & 0.353 \\ 0.333 & 0.288 & 0.333 & 0 \\ 0 & 0.353 & 0 & 0.5 \end{bmatrix}$$

The parameters of the message passing layer are as follows:

$$\mathbf{W} = \begin{bmatrix} 1.064 & 0.211 & -0.557 \\ -1.282 & 0.614 & 0.996 \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} -1.177 & -0.540 & 1.331 \end{bmatrix}$$

The representations of the first message passing layer are computed as follows:

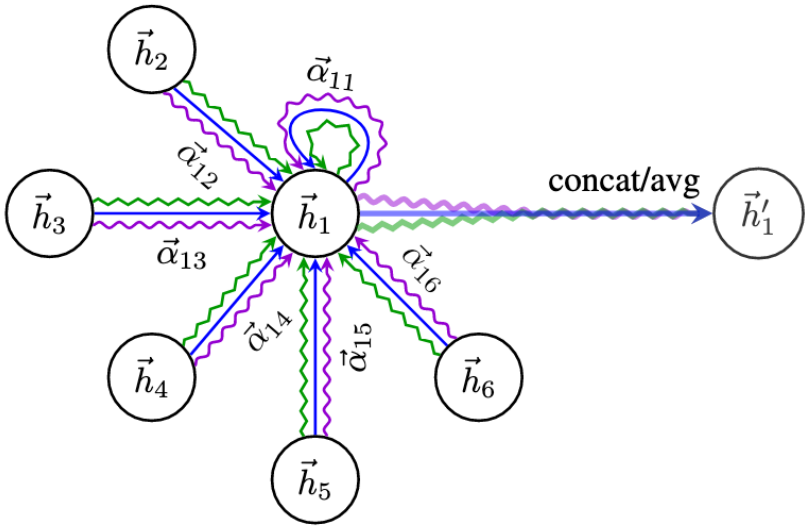
$$\mathbf{H} = \mathrm{ReLU}\Big(\hat{\mathbf{A}}(\mathbf{X}\mathbf{W} + \mathbf{b})\Big) = \begin{bmatrix} 0 & 5.024 & 9.466 \\ 0 & 7.859 & 13.588 \\ 0 & 5.024 & 9.466 \\ 0 & 6.971 & 11.281 \end{bmatrix}$$

Graph Attention Network (GAT)

- **Idea**: Messages from some neighbors may be more important that messages from others
- GAT applies self-attention on the nodes
- For nodes $v_j \in \mathcal{N}(v_i)$, computes attention coefficients that indicate the importance of node v_j's features to node v_i:

$$\alpha_{ij}^{(t)} = \frac{\exp\Big(\text{LeakyReLU}\big(\mathbf{a}^\top[\mathbf{W}^{(t)}\mathbf{h}_i^{(t)}||\mathbf{W}\mathbf{h}_j^{(t)}]\big)\Big)}{\sum_{k\in\mathbf{N}_i}\exp\Big(\text{LeakyReLU}\big(\mathbf{a}^\top[\mathbf{W}^{(t)}\mathbf{h}_i^{(t)}||\mathbf{W}^{(t)}\mathbf{h}_k^{(t)}]\big)\Big)}$$

where $[\cdot||\cdot]$ denotes concatenation of two vectors and $\mathbf{a}$ is a trainable vector



Graph Attention Network (GAT)

Then the representations of the nodes are updated as follows:

$$\mathbf{h}_i^{(t+1)} = \sigma\Big(\sum_{j \in \mathcal{N}_i} \alpha_{ij}^{(t)} \mathbf{W}^{(t)} \mathbf{h}_j^{(t)}\Big)$$

In matrix form, the above is equivalent to:

$$\mathbf{H}^{(t+1)} = \sigma\Big(\big(\mathbf{A} \odot \mathbf{T}^{(t)}\big)\mathbf{H}^{(t)}\mathbf{W}^{(t)}\Big)$$

where $\odot$ denotes elementwise product and is matrix such that $\mathbf{T}_{ij}^{(t)} = \alpha_{ij}^{(t)}$

More than one attention mechanisms can be employed by concatenating/averaging their respective node representations, e.g., for averaging:

$$\mathbf{h}_i^{(t+1)} = \sigma\Big(\frac{1}{K}\sum_{k=1}^{K}\sum_{j \in \mathcal{N}_i} [\alpha_k^{(t)}]_{ij} \mathbf{W}_k^{(t)} \mathbf{h}_j^{(t)}\Big)$$

where $[\alpha_k^{(t)}]_{ij}$ are the attention coefficients computed by the k^{th} attention mechanism, and $\mathbf{W}_k^{(t)}$ is the corresponding weight matrix

[Veličković et al., ICLR'18]

GraphSAGE

The GraphSAGE model can deal with very large graphs
$\hookrightarrow$ the model does not take into account all neighbors of a node, but uniformly samples a fixed-size set of neighbors

Let $\mathcal{N}^k(v)$ be a uniformly drawn subset (of size k) from the set $\mathcal{N}(v)$
The message passing scheme of GraphSAGE is defined as follows:

$$\mathbf{m}_v^{(t)} = \mathrm{AGGREGATE}^{(t)}\left(\left\{\mathbf{h}_u^{(t)} \middle| u \in \mathcal{N}^k(v)\right\}\right)$$

$$\mathbf{h}_v^{(t+1)} = \sigma\left(\mathbf{W}^{(t)}\left[\mathbf{h}_v^{(t)} || \mathbf{m}_v^{(t)}\right]\right)$$

$$\mathbf{h}_v^{(t+1)} = \frac{\mathbf{h}_v^{(t+1)}}{\left|\left|\mathbf{h}_v^{(t+1)}\right|\right|_2}$$

The model draws different uniform samples at each iteration

[Hamilton et al., NIPS'17]

GraphSAGE

The model uses one of the following trainable aggregation functions:

1. **Mean aggregator**: the mean operator computes the elementwise mean of the representations of the neighbors and the node itself (the concatenation step, i.e., second Equation of previous slide is skipped):

$$\mathbf{h}_v^{(t+1)} = \sigma\left(\mathbf{W}^{(t)} \frac{\mathbf{h}_v^{(t)} + \sum_{u \in \mathcal{N}^k(v)} \mathbf{h}_u^{(t)}}{d(v)+1}\right)$$

   where $d(v)$ is the degree of node v

2. **LSTM aggregator**: the representations of the neighbors are passed on to an LSTM architecture
   ⚠ LSTMs are not permutation invariant

3. **Pooling aggregator**: an elementwise max-pooling operation is applied to aggregate information across the neighbor set:

$$\text{AGGREGATE}_{\text{pool}}^{(t)} = \max\left(\left\{\sigma\left(\mathbf{W}_{\text{pool}}^{(t)} \mathbf{h}_u^{(t)}\right) \middle| u \in \mathcal{N}^k(v)\right\}\right)$$

   where max denotes the elementwise max operator

Skip Connections

Idea: Instead of using only the final node representations $\mathbf{h}_v^{(T)}$ (i.e., obtained after T message passing steps), can also use the representations of the earlier message passing layers $\mathbf{h}_v^{(1)}, \ldots, \mathbf{h}_v^{(T-1)}$

Multi-hop information

- As one iterates, vertex representations capture more and more global information
- However, retaining more local, intermediary information might be useful too.
- Thus, we concatenate the representations produced at the different steps, finally obtaining $\mathbf{h}_v = [\mathbf{h}_v^{(1)} || \mathbf{h}_v^{(2)} || \ldots || \mathbf{h}_v^{(T)}]$

[Xu et al., ICML'18]

Outline

Graph Autoencoders (GAE)

One of the main problems in representation learning for graphs is the following: How can we learn node embedding representations in an unsupervised fashion?

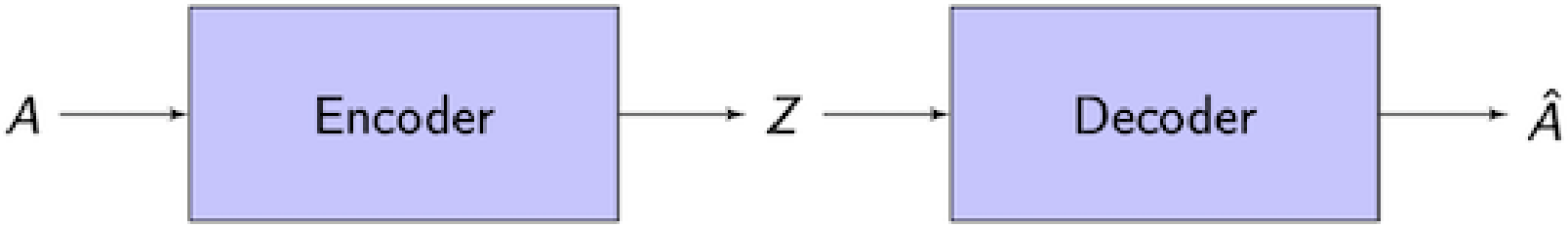
- DeepWalk
- node2vec

⋮

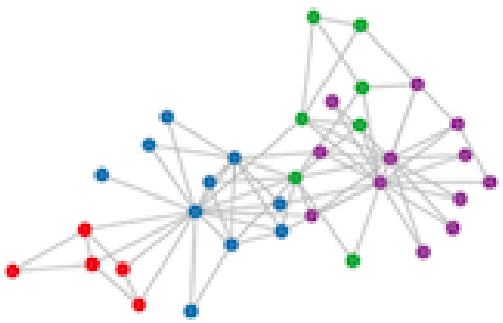
In the last few years: several attempts to generalize autoencoders to graphs:

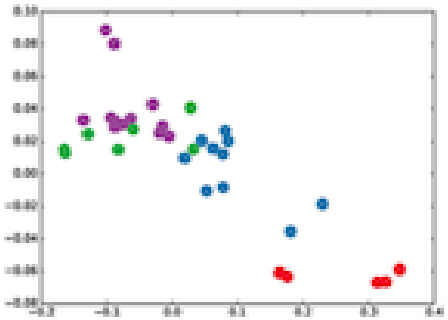
- input: $n \times n$ adjacency matrix $\mathbf{A}$ and (potentially) an $n \times d$ node features matrix $\mathbf{X}$, stacking-up d-dimensional vectors associated to each node
- objective: derive an $n \times d$ latent representation matrix $\mathbf{Z}$ (*encoding step*) from which we can reconstruct (*decoding step*) $\mathbf{A}$

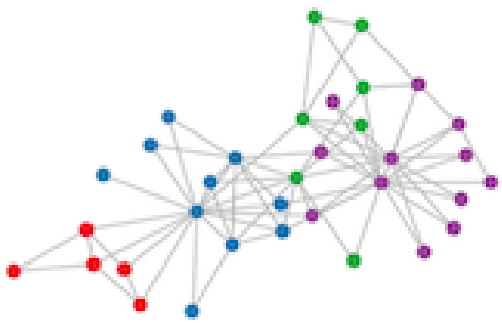
Graph Autoencoders (GAE)



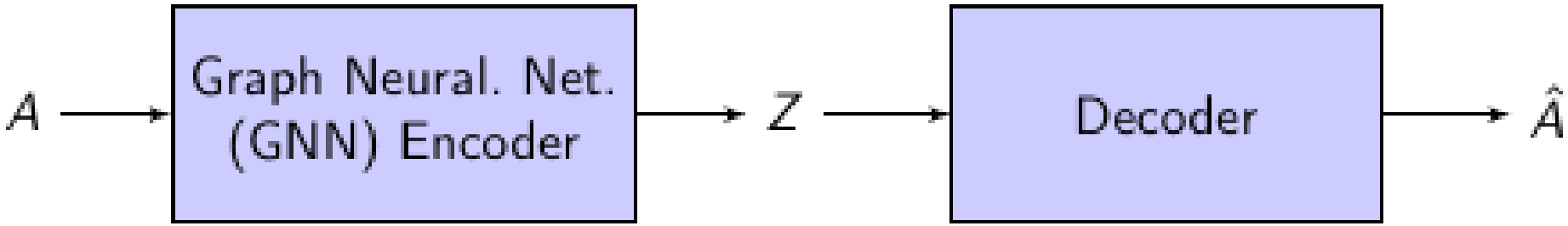
$$A = \begin{pmatrix} 0 & 1 & 0 & \dots & 1 \\ 1 & 0 & 1 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 1 & 0 & 0 & \dots & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 0.523 & -1.012 \\ 2.127 & 0.316 \\ 0.912 & 0.127 \\ \dots & \dots \\ -1.210 & 0.026 \end{pmatrix} \quad \hat{A} = \begin{pmatrix} 0 & 1 & 0 & \dots & 1 \\ 1 & 0 & 1 & \dots & 1 \\ 0 & 1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 1 & 1 & 0 & \dots & 0 \end{pmatrix}$$







Graph Autoencoders (GAE)



Encoder: usually a **Graph Neural Network**, e.g.:

- Graph Convolutional Network (GCN)
- Graph Attention Network (GAT)
- GraphSAGE

⋮

Most graph autoencoders rely on **multi-layer GCN** encoders

Graph Autoencoders (GAE)

Graph AE:

1. **encoder:** $\boxed{\mathbf{Z} = \text{GNN}(\mathbf{A}, \mathbf{X})}$
2. **decoder:** $\boxed{\hat{\mathbf{A}} = \sigma(\mathbf{Z}\mathbf{Z}^\top)}$
   i.e., for all node pairs (i, j), we have $\hat{\mathbf{A}}_{ij} = \sigma(\mathbf{z}_i^\top \mathbf{z}_j)$

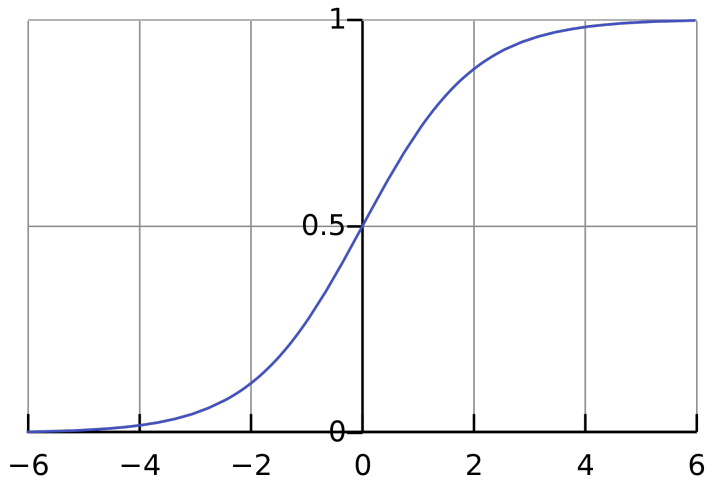


Figure: Sigmoid activation: $\sigma(x) = \frac{1}{1+e^{-x}}$

Reconstruction Loss[1]**:** capturing the similarity between $\mathbf{A}$ and $\hat{\mathbf{A}}$

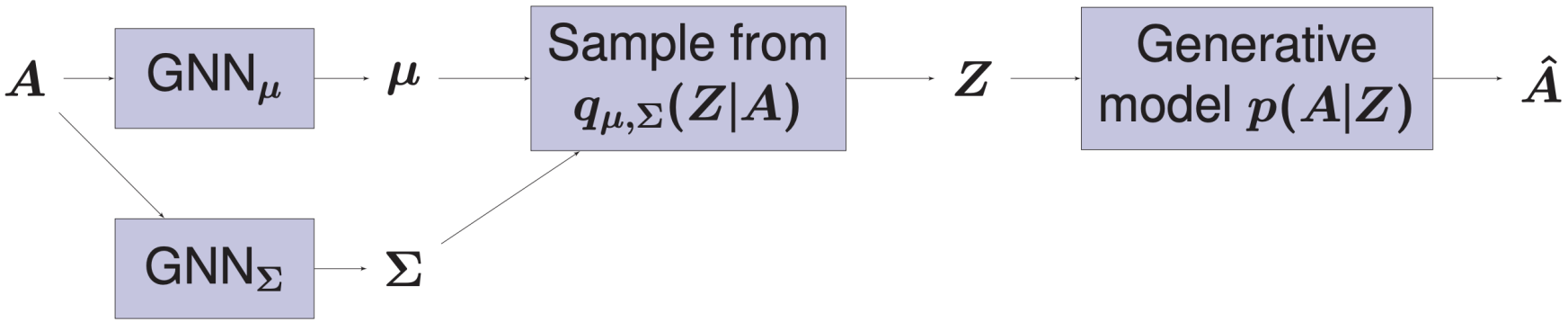
- e.g., **cross-entropy loss**: $-\sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{A}_{ij}\log(\hat{\mathbf{A}}_{ij}) + (1-\mathbf{A}_{ij})\log(1-\hat{\mathbf{A}}_{ij}))$
- or **MSE** loss: $\sum_{i=1}^{n}\sum_{j=1}^{n}(\mathbf{A}_{ij} - \hat{\mathbf{A}}_{ij})^2$

[1] in losses, we usually reweight positive terms or use negative sampling, if G is sparse

Graph Variational Autoencoders (GVAE)

Also, **Graph VAE**

- extend **Variational Autoencoders (VAE)** to graph structures



Maximizing a lower bound of the model's likelihood (**ELBO**):

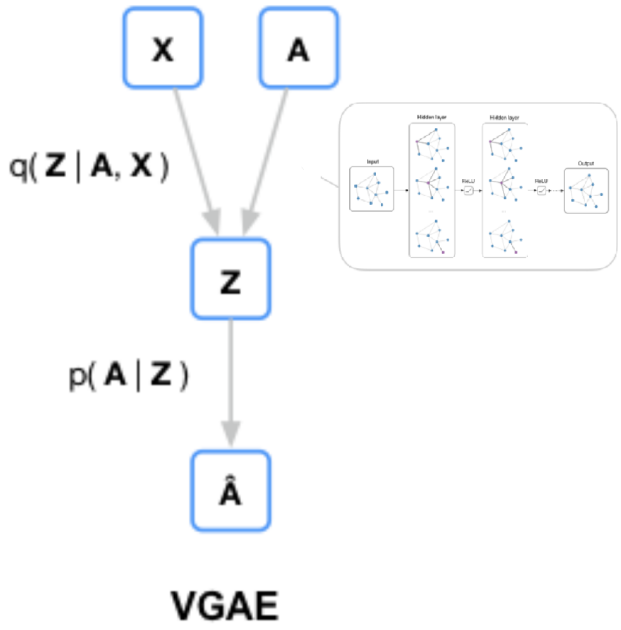
$$\mathcal{L} = \mathbb{E}_{q(\mathbf{Z}|\mathbf{A})}\Big[\log p(\mathbf{A}|\mathbf{Z})\Big] - \mathcal{D}_{KL}(q(\mathbf{Z}|\mathbf{A})||p(\mathbf{Z}))$$

[Kipf and Welling, Bayesian Deep Learning Workshop'16]

Graph Variational Autoencoders (GVAE)

Encoder: $q(\mathbf{Z}|\mathbf{X},\mathbf{A}) = \prod_{i=1}^{n} q(\mathbf{z}_i|\mathbf{X},\mathbf{A})$ where $q(\mathbf{z}_i|\mathbf{X},\mathbf{A}) = \mathcal{N}(\mathbf{z}_i|\boldsymbol{\mu}_i, \text{diag}(\boldsymbol{\sigma}_i^2))$

Gaussian parameters learned by 2 GCN:
$\boldsymbol{\mu} = \text{GNN}_{\mu}(\mathbf{X},\mathbf{A})$ and $\log \boldsymbol{\sigma} = \text{GNN}_{\sigma}(\mathbf{X},\mathbf{A})$



Decoder: $p(\mathbf{A}|\mathbf{Z}) = \prod_{i=1}^{n} \prod_{j=1}^{n} p(\mathbf{A}_{ij}|\mathbf{z}_i,\mathbf{z}_j)$ where $p(\mathbf{A}_{ij} = 1|\mathbf{z}_i,\mathbf{z}_j) = \sigma(\mathbf{z}_i^{\top}\mathbf{z}_j)$

Maximizing ELBO: $\boxed{\mathcal{L} = \mathbb{E}_{q(\mathbf{Z}|\mathbf{X},\mathbf{A})}\Big[\log p(\mathbf{A}|\mathbf{Z})\Big] - \mathcal{D}_{KL}(q(\mathbf{Z}|\mathbf{X},\mathbf{A})||p(\mathbf{Z}))}$

Performing full-batch gradient descent, using the *re-parameterization trick*, and choosing a Gaussian prior $p(\mathbf{Z}) = \prod_i p(\mathbf{z}_i) = \prod_i \mathcal{N}(\mathbf{z}_i|0,\mathbf{I})$.

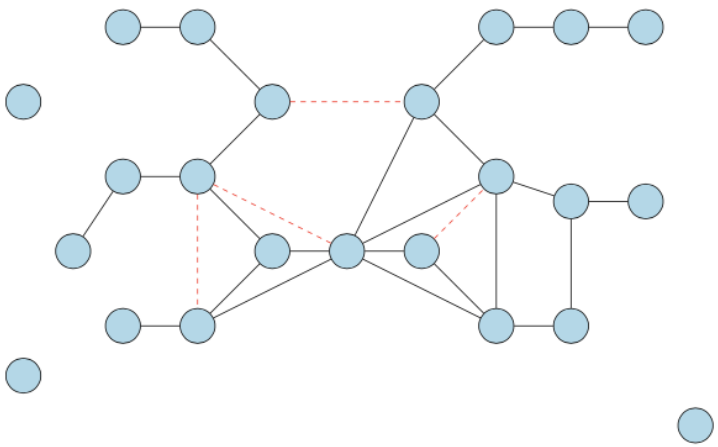
Applications of Graph Autoencoders

The embedding spaces learned via Graph AE and VAE led to many promising applications during the past few years:

- link prediction
- node clustering
- recommendation

$\vdots$

Recall: Link Prediction

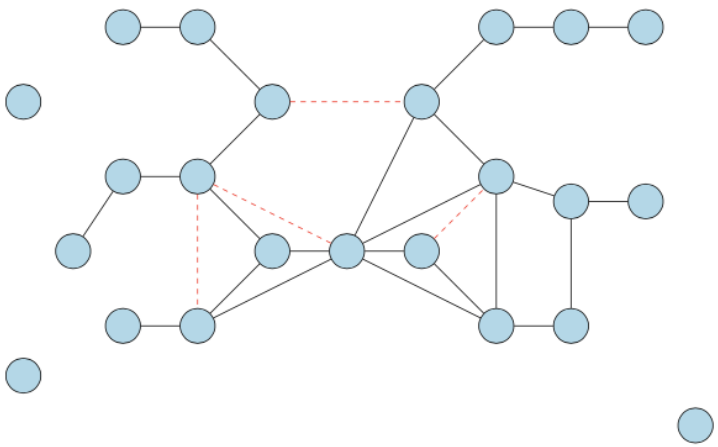


Test set:

- missing edges
- unconnected pairs of nodes

Pair of nodes	**Are nodes connected in ground-truth G?**
(v_1, v_2)	1
(v_3, v_4)	1
(v_5, v_6)	1
...	...
(v_7, v_8)	0
(v_9, v_{10})	0
(v_{11}, v_{12})	0

Recall: Link Prediction



Test set:

- missing edges
- unconnected pairs of nodes

Pair of nodes	**Are nodes connected in ground-truth G?**
(v_1, v_2)	1
(v_3, v_4)	1
(v_5, v_6)	1
...	...
(v_7, v_8)	0
(v_9, v_{10})	0
(v_{11}, v_{12})	0

$\longrightarrow$ **binary classification task**, identify missing edges from incomplete train graph

Link Prediction with Graph Autoencoders

Method	Cora		Citeseer		Pubmed	
	AUC	AP	AUC	AP	AUC	AP
SC [5]	84.6 ± 0.01	88.5 ± 0.00	80.5 ± 0.01	85.0 ± 0.01	84.2 ± 0.02	87.8 ± 0.01
DW [6]	83.1 ± 0.01	85.0 ± 0.00	80.5 ± 0.02	83.6 ± 0.01	84.4 ± 0.00	84.1 ± 0.00
GAE*	84.3 ± 0.02	88.1 ± 0.01	78.7 ± 0.02	84.1 ± 0.02	82.2 ± 0.01	87.4 ± 0.00
VGAE*	84.0 ± 0.02	87.7 ± 0.01	78.9 ± 0.03	84.1 ± 0.02	82.7 ± 0.01	87.5 ± 0.01
GAE	91.0 ± 0.02	92.0 ± 0.03	89.5 ± 0.04	89.9 ± 0.05	$\mathbf{96.4} \pm 0.00$	$\mathbf{96.5} \pm 0.00$
VGAE	$\mathbf{91.4} \pm 0.01$	$\mathbf{92.6} \pm 0.01$	$\mathbf{90.8} \pm 0.02$	$\mathbf{92.0} \pm 0.02$	94.4 ± 0.02	94.7 ± 0.02

[Kipf and Welling, Bayesian Deep Learning Workshop'16]

Node Embedding - Cora Graph

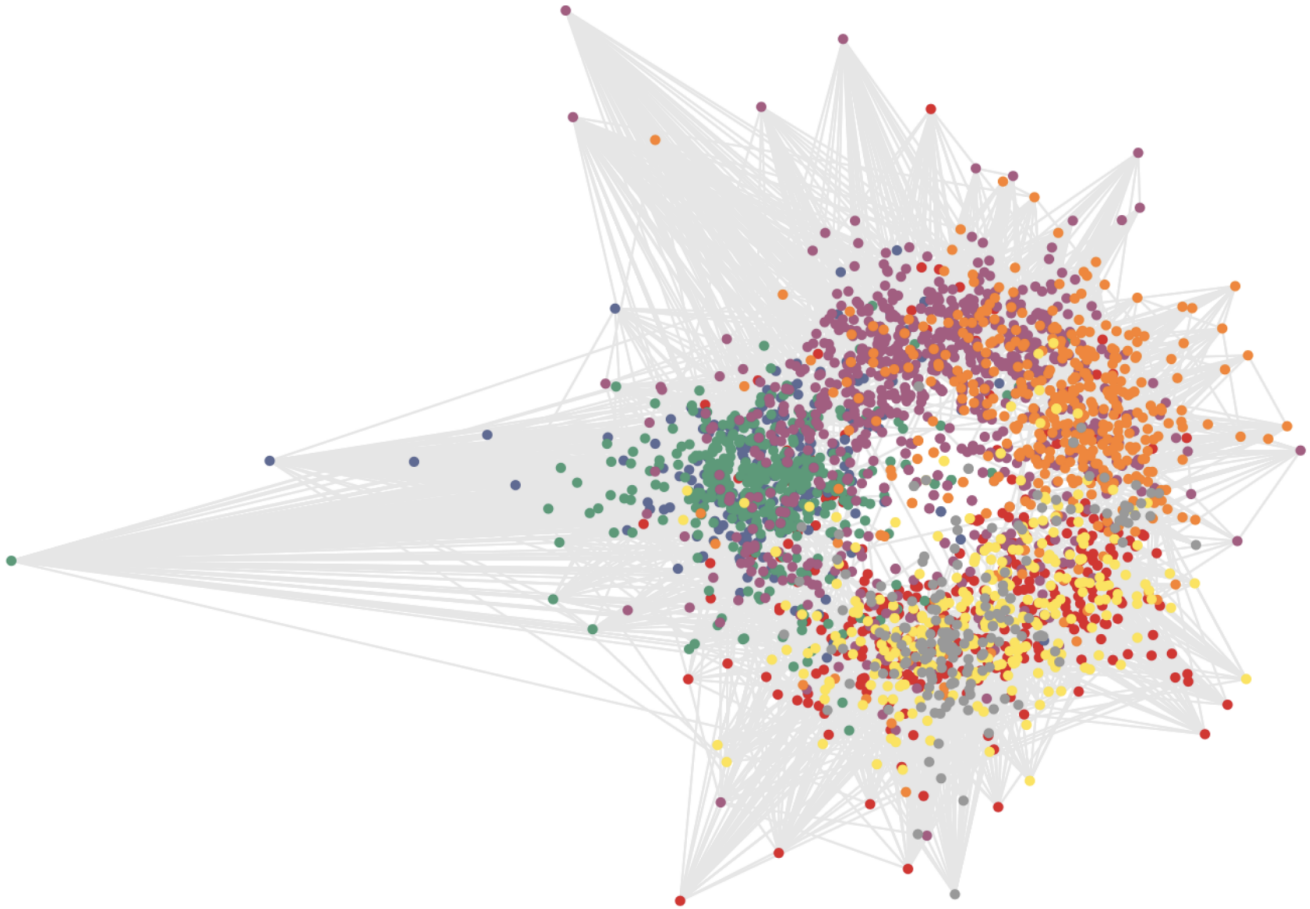


Figure: Projection of latent space representations, from Graph VAE model trained on Cora citation network. Colors denote document classes i.e. node labels (not provided during training)

Graph Autoencoders for Recommendation

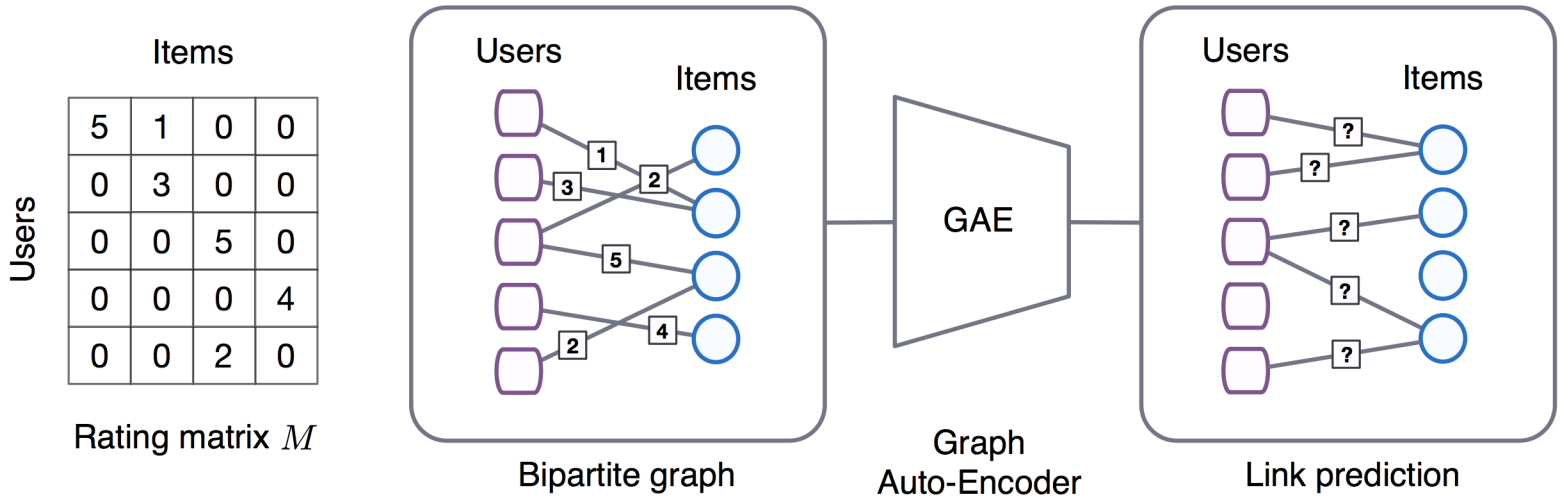
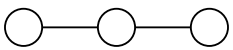


Figure: Using GAE for Matrix Completion and Recommendation

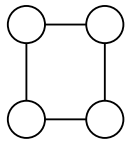
[van den Berg et al., KDD'18 Deep Learning Day]

Outline

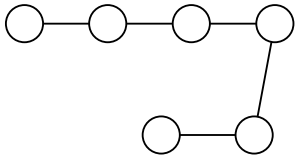
Graph Classification



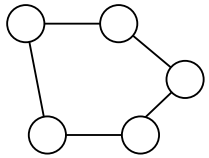
class -1



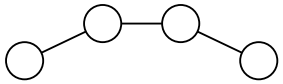
class 1



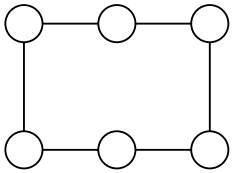
class -1



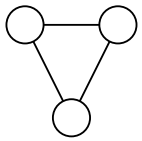
???



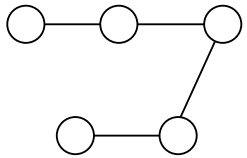
class -1



class 1



class 1



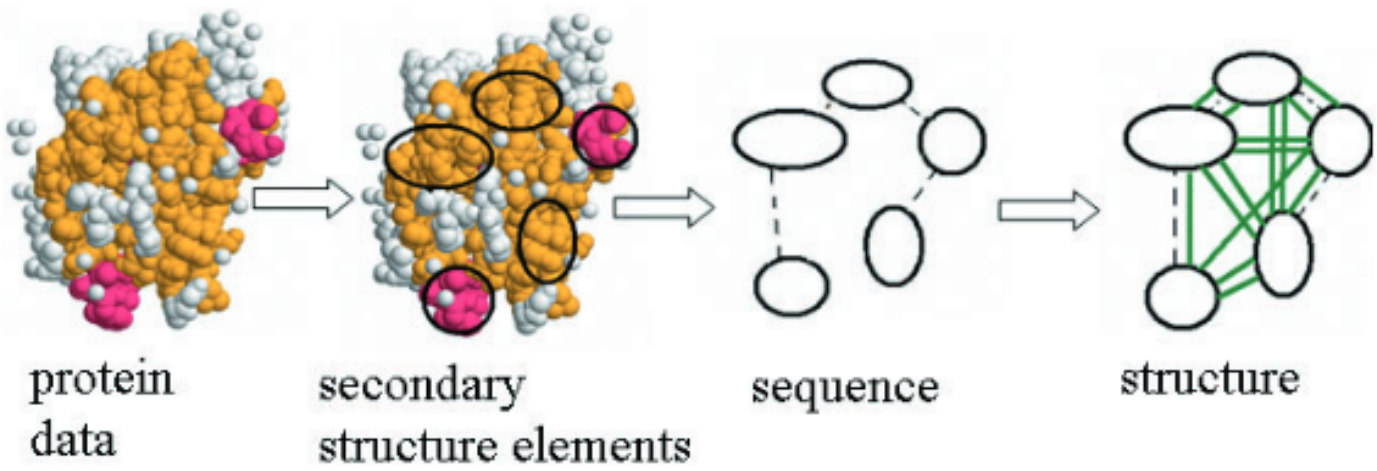
???

- Input data $G \in \mathcal{G}$
- Output $y \in \{-1, 1\}$
- Training set $\mathcal{S} = \{(G_1, y_1), \ldots, (G_n, y_n)\}$
- Goal: estimate a function $f : \mathcal{G} \rightarrow \in \{-1, 1\}$ to predict y from $f(G)$

Motivation - Protein Function Prediction

For each protein, create a graph that contains information about its

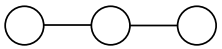
- structure
- sequence
- chemical properties



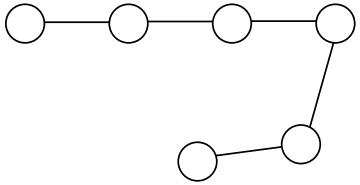
Perform **graph classification** to predict the function of proteins

[Borgwardt et al., Bioinformatics 21]

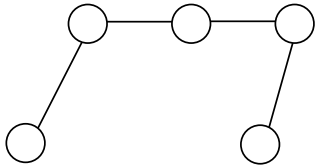
Graph Regression



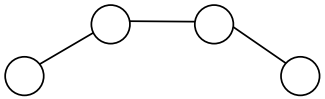
G_1
$y_1 = 3$



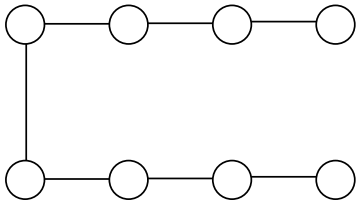
G_2
$y_2 = 6$



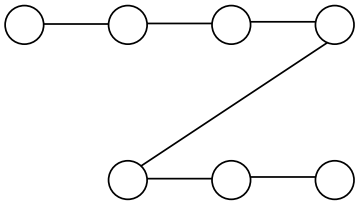
G_5
$y_5 = ???$



G_3
$y_3 = 4$



G_4
$y_4 = 8$

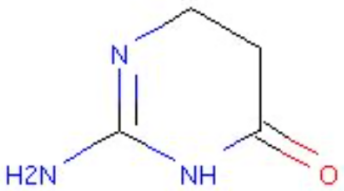


G_6
$y_6 = ???$

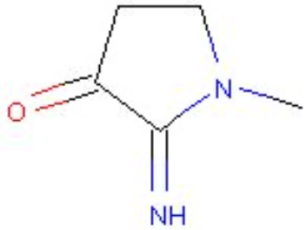
- Input data $G \in \mathcal{G}$
- Output $y \in \mathbb{R}$
- Training set $\mathcal{S} = \{(G_1, y_1), \ldots, (G_n, y_n)\}$
- Goal: estimate a function $f : \mathcal{G} \to \mathbb{R}$ to predict y from $f(G)$

Motivation - Molecular Property Prediction

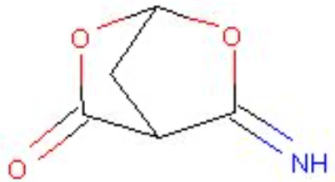
12 targets corresponding to molecular properties: ['mu', 'alpha', 'HOMO', 'LUMO', 'gap', 'R2', 'ZPVE', 'U0', 'U', 'H', 'G', 'Cv']



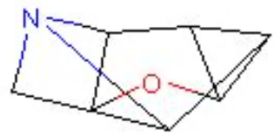
SMILES: NC1=NCCC(=O)N1
Targets: [2.54 64.1 -0.236 -2.79e-03 2.34e-01 900.7 0.12 -396.0 -396.0 -396.0 -396.0 26.9]



SMILES: CN1CCC(=O)C1=N
Targets: [4.218 68.69 -0.224 -0.056 0.168 914.65 0.131 -379.959 -379.951 -379.95 -379.992 27.934]

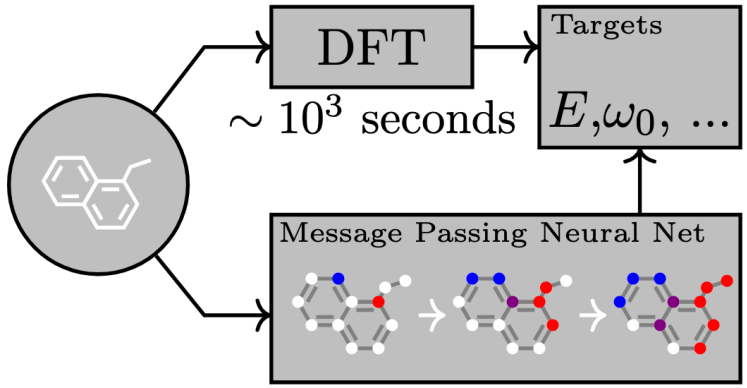


SMILES: N=C1OC2CC1C(=O)O2
Targets: [4.274 61.94 -0.282 -0.026 0.256 887.402 0.104 -473.876 -473.87 -473.869 -473.907 24.823]



SMILES: C1N2C3C4C5OC13C2C5
Targets: [? ? ? ? ? ? ? ? ? ? ? ?]

Perform **graph regression** to predict the values of the properties



[Gilmer et al., ICML'17]

Outline

Message Passing Neural Networks for Learning Graph Representations

Consist of a series of message passing layers followed by a readout function

Step 1: The message passing phase runs for T time steps and updates the representation of each vertex $\mathbf{h}_v^t$ based on its previous representation and the representations of its neighbors:

$$\mathbf{m}_v^{(t+1)} = \mathrm{AGGREGATE}\left(\left\{\mathbf{h}_u^{(t)} \middle| u \in \mathcal{N}(v)\right\}\right)$$

$$\mathbf{h}_v^{(t+1)} = \mathrm{COMBINE}\left(\mathbf{h}_v^{(t)}, \mathbf{m}_v^{(t+1)}\right)$$

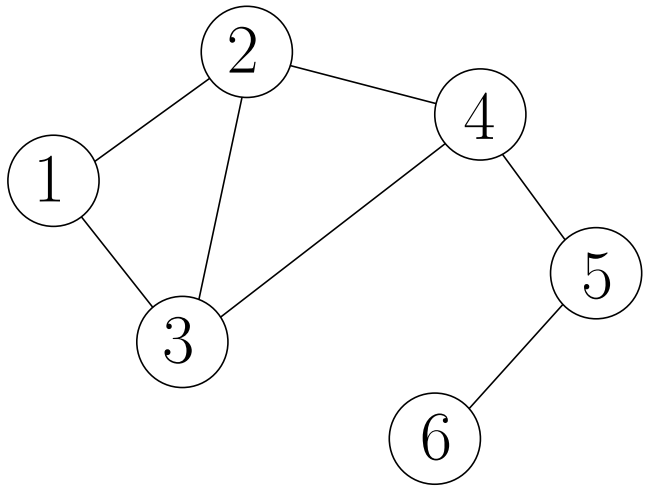
where $\mathcal{N}(v)$ is the set of neighbors of v, and $\mathrm{AGGREGATE}$ and $\mathrm{COMBINE}$ are message functions and vertex update functions respectively

Step 2: The readout step computes a feature vector for the whole graph using some readout function R:

$$\mathbf{h}_G = \mathrm{READOUT}\left(\left\{\mathbf{h}_v^{(T)} \middle| v \in G\right\}\right)$$

Example of Message Passing Layer

$$\mathbf{h}_1^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_1^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_3^{(t)})$$
$$\mathbf{h}_2^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_1^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_3^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_4^{(t)})$$
$$\mathbf{h}_3^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_3^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_1^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_4^{(t)})$$
$$\mathbf{h}_4^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_4^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_2^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_3^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_5^{(t)})$$
$$\mathbf{h}_5^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_5^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_4^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_6^{(t)})$$
$$\mathbf{h}_6^{(t+1)} = f(\mathbf{W}_0^{(t)}\mathbf{h}_6^{(t)} + \mathbf{W}_1^{(t)}\mathbf{h}_5^{(t)})$$

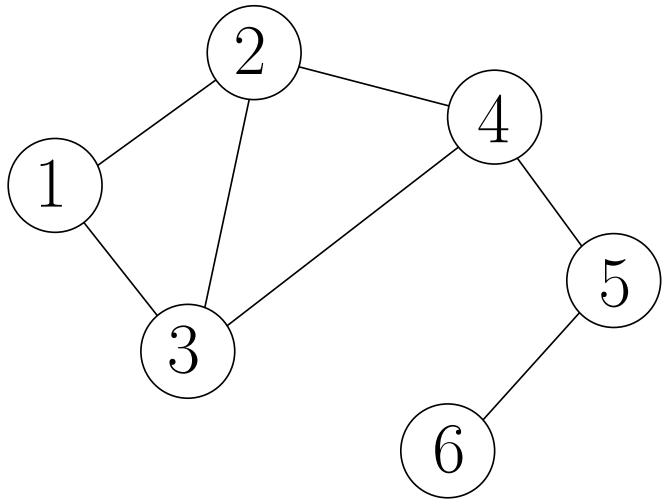


Remark: Biases are omitted for clarity

Readout Step Example

Output of message passing phase:

$$\left\{\mathbf{h}_1^{(T)}, \mathbf{h}_2^{(T)}, \mathbf{h}_3^{(T)}, \mathbf{h}_4^{(T)}, \mathbf{h}_5^{(T)}, \mathbf{h}_6^{(T)}\right\}$$



Graph representation:

$$\mathbf{h}_G = \frac{1}{6}\left(\mathbf{h}_1^{(T)} + \mathbf{h}_2^{(T)} + \mathbf{h}_3^{(T)} + \mathbf{h}_4^{(T)} + \mathbf{h}_5^{(T)} + \mathbf{h}_6^{(T)}\right)$$

How Can we Build Message Passing Neural Networks for Learning Graph Representations?

1. Take a message passing neural network that can produce node representations
2. Add a readout function to the model. Simple and popular functions.
   - sum aggerator: computes the sum of the representations of the nodes of the graph
   $$\mathbf{h}_G = \sum_{v \in V} \mathbf{h}_v^{(T)}$$
   - mean aggerator: computes the sum of the representations of the nodes of the graph
   $$\mathbf{h}_G = \frac{1}{n} \sum_{v \in V} \mathbf{h}_v^{(T)}$$
   - max aggerator: an elementwise max-pooling operation is applied to the representations of the nodes of the graph
   $$\mathbf{h}_G = \max \left(\left\{ \mathbf{h}_v^{(T)} \right) \middle| v \in V \right\} \right)$$
   where max denotes the elementwise max operator

Example of Simple Readout Functions

Suppose we have a graph consisting of 3 nodes and we have that:

$$\mathbf{h}_1^{(T)} = \begin{bmatrix} 1.2 & 1.4 & -1.0 \end{bmatrix} \qquad \mathbf{h}_2^{(T)} = \begin{bmatrix} -2.4 & -0.6 & 1.3 \end{bmatrix}$$
$$\mathbf{h}_3^{(T)} = \begin{bmatrix} 1.5 & 1.3 & -0.9 \end{bmatrix}$$

Then, we can produce graph representations as follows:

- sum aggerator:

$$\mathbf{h}_G = \mathbf{h}_1^{(T)} + \mathbf{h}_2^{(T)} + \mathbf{h}_3^{(T)} = \begin{bmatrix} 0.3 & 2.1 & -0.6 \end{bmatrix}$$

- mean aggerator:

$$\mathbf{h}_G = \frac{1}{3}\left(\mathbf{h}_1^{(T)} + \mathbf{h}_2^{(T)} + \mathbf{h}_3^{(T)}\right) = \begin{bmatrix} 0.1 & 0.7 & -0.2 \end{bmatrix}$$

- max aggerator:

$$\mathbf{h}_G = \max\left(\left\{\mathbf{h}_1^{(T)}, \mathbf{h}_2^{(T)}, \mathbf{h}_3^{(T)}\right\}\right) = \begin{bmatrix} 1.5 & 1.4 & 1.3 \end{bmatrix}$$

Step 1: The network updates the states of the nodes as follows:

$$\mathbf{m}_v^{(t+1)} = \mathbf{h}_v^{(t)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(t)}$$

$$\mathbf{h}_v^{(t+1)} = \sigma\left(\mathbf{H}_{d(v)}^{(t)} \mathbf{m}_v^{(t+1)}\right)$$

where $d(v)$ is degree of vertex v and $\mathbf{H}_{d(v)}^{(t)}$ a learned matrix for each time step t and vertex degree $d(v)$

Step 2: The network computes the graph representation as:

$$\mathbf{h}_G = \sum_{t=0}^{T} \sum_{v \in V} \text{softmax}(\mathbf{W}^{(t)} \mathbf{h}_v^{(t)})$$

The output $\mathbf{h}_G$ is then fed to a fully-connected neural network

[Duvenaud et al, NIPS'15]

Deep Graph Convolutional Neural Network (DGCNN)

Step 1: Aggregates node information in local neighborhoods to extract local substructure information:

$$\mathbf{H}^{(t+1)} = f\left(\tilde{\mathbf{D}}^{-1}\tilde{\mathbf{A}}\mathbf{H}^{(t)}\mathbf{W}^{(t)}\right)$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$, $\mathbf{D}$ is a diagonal matrix such that $\tilde{\mathbf{D}}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$, and f is a nonlinear activation function

After T iterations, the model concatenates the outputs $\mathbf{H}^{(t)}$, for $t = 1, \ldots, T$ horizontally to form a concatenated output:

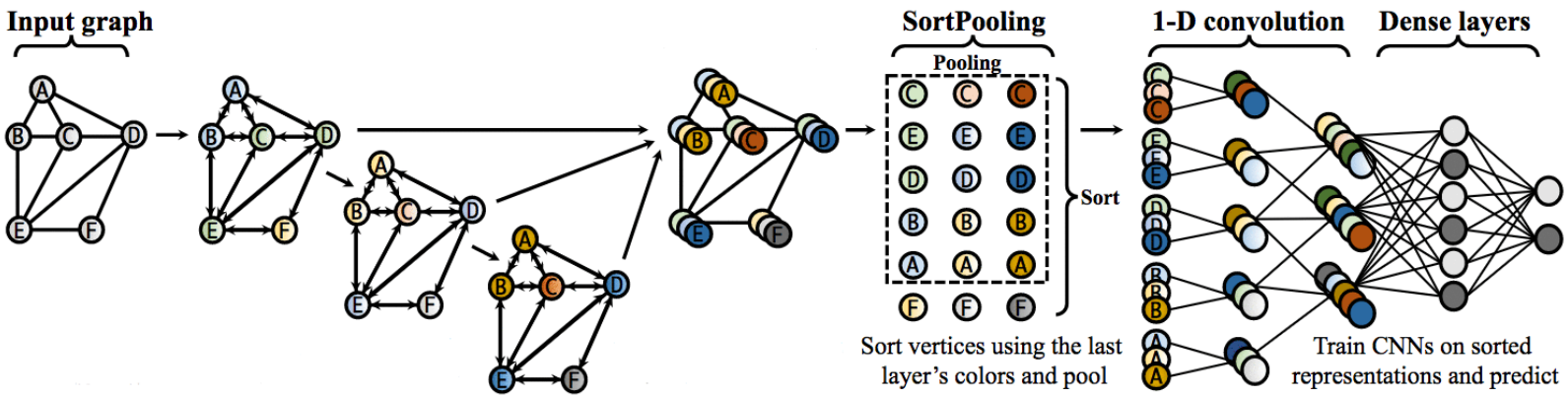
$$\mathbf{H} = \left[\mathbf{H}^{(1)}||\mathbf{H}^{(2)}||\ldots||\mathbf{H}^{(T)}\right]$$

[Zhang et al, AAAI'18]

Deep Graph Convolutional Neural Network (DGCNN)

Step 2: Employs the so-called SortPooling layer:

- Sorts the output **H** of previous step row-wise:
  - vertices are sorted in a descending order based on the last component of **H**
  - vertices that have the same value in the last component are compared based on the second to last component and so on
- Unifies the sizes of the outputs to handle graphs with different numbers of vertices:
  - Truncates/extends the output tensor in the first dimension from n to k
- Output is then passed to traditional CNN



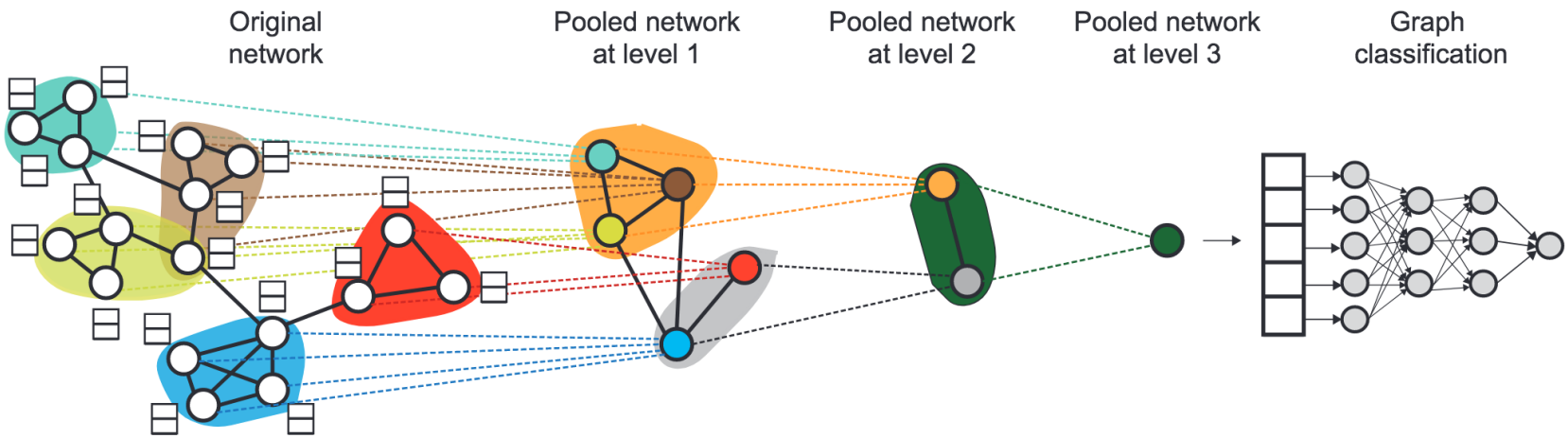
Differentiable Graph Pooling (DiffPool)

Idea: Simple readout functions too flat
$\hookrightarrow$ Aggregate information in a hierarchical way to capture the entire graph

The DiffPool model

- learns hierarchical pooling analogous to CNNs
- sets of nodes are pooled hierarchically
- soft assignment of nodes to next-level nodes

A different GNN is learned at every level of abstraction



Differentiable Graph Pooling (DiffPool)

A matrix $\mathbf{S}^{(t)} \in \mathbb{R}^{n_t \times n_{t+1}}$ is associated with each DiffPool layer

- corresponds to the learned cluster assignment matrix at layer t
- each row corresponds to one of the n_t nodes (or clusters) at layer t and each column to one of the n_{t+1} clusters at the next layer $t+1$
- it provides a soft assignment of each node at layer l to a cluster in the next coarsened layer $t+1$

Each DiffPool layer coarsens the input graph:

$$\mathbf{X}^{(t+1)} = \mathbf{S}^{(t)^\top} \mathbf{Z}^{(t)}$$

$$\mathbf{A}^{(t+1)} = \mathbf{S}^{(t)^\top} \mathbf{A}^{(t)} \mathbf{S}^{(t)}$$

where $\mathbf{A}^{(t+1)}$ is the coarsened adjacency matrix, and $\mathbf{X}^{(t+1)}$ is a matrix of embeddings for each node/cluster

Differentiable Graph Pooling (DiffPool)

- DiffPool generates the assignment and embedding matrices using two separate message passing neural networks
- Both are applied to the input cluster node features $\mathbf{X}^{(t)}$ and coarsened adjacency matrix $\mathbf{A}^{(t)}$

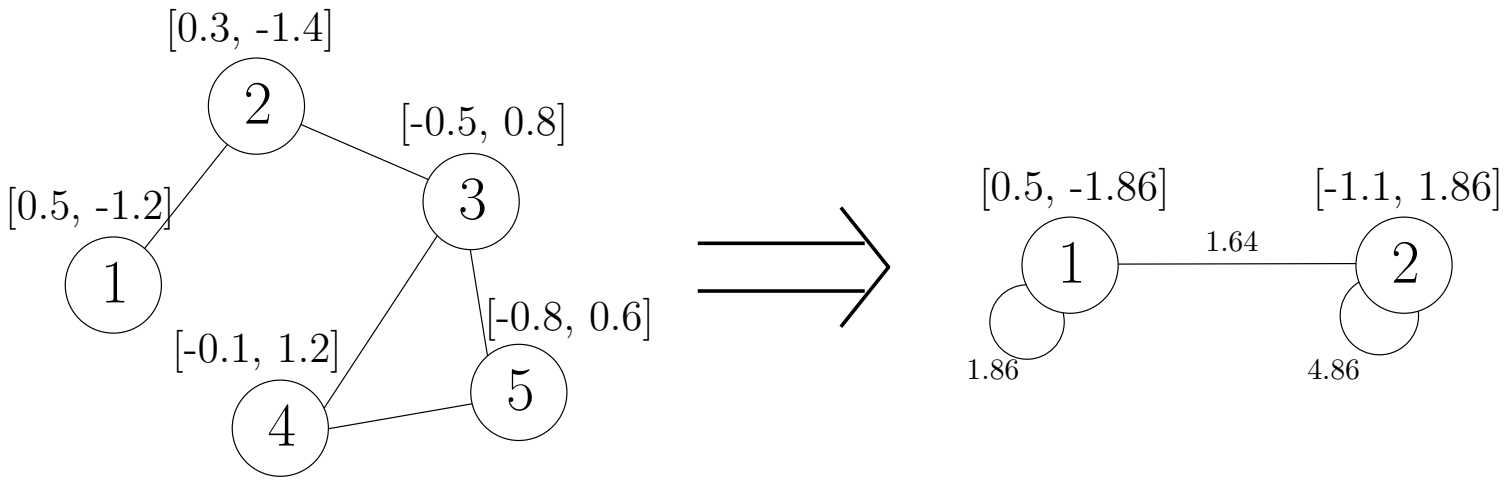
$$\mathbf{Z}^{(t)} = \text{GNN}^{(t)}_{\text{embed}}(\mathbf{A}^{(t)}\mathbf{X}^{(t)})$$
$$\mathbf{S}^{(t)} = \text{softmax}\big(\text{GNN}^{(t)}_{\text{pool}}(\mathbf{A}^{(t)}\mathbf{X}^{(t)})\big)$$

  where the softmax function is applied in a row-wise fashion
- $\text{GNN}^{(t)}_{\text{embed}}$ generates new representations for the input nodes
- $\text{GNN}^{(t)}_{\text{pool}}$ generates a probabilistic assignment of the input nodes to n_{t+1} clusters

Example of Coarsening Procedure of DiffPool

$$\mathbf{A}^{(1)} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 & 0 \end{bmatrix} \quad \mathbf{Z}^{(1)} = \begin{bmatrix} 0.5 & -1.2 \\ 0.3 & -1.4 \\ -0.5 & 0.8 \\ -0.1 & 1.2 \\ -0.8 & 0.6 \end{bmatrix} \quad \mathbf{S}^{(1)} = \begin{bmatrix} 0.9 & 0.1 \\ 0.8 & 0.2 \\ 0.2 & 0.8 \\ 0.1 & 0.9 \\ 0.1 & 0.9 \end{bmatrix}$$

$$\mathbf{X}^{(2)} = \mathbf{S}^{(1)\top}\mathbf{Z}^{(1)} = \begin{bmatrix} 0.5 & -1.86 \\ -1.1 & 1.86 \end{bmatrix} \quad \mathbf{A}^{(2)} = \mathbf{S}^{(1)\top}\mathbf{A}^{(1)}\mathbf{S}^{(1)} = \begin{bmatrix} 1.86 & 1.64 \\ 1.64 & 4.86 \end{bmatrix}$$



How Powerful Are Message Passing Graph Neural Networks?

- The expressive power of standard message passing graph neural networks has recently started being investigated
- Xu et al. and Morris et al. showed independently that they are **at most** as powerful as the Weisfeiler-Lehman (WL) test of isomorphism in terms of distinguishing between non-isomorphic graphs

Lemma (Xu et al., ICLR'19)

Let G_1 and G_2 be any two non-isomorphic graphs. If a standard graph neural network $A : \mathcal{G} \to \mathbb{R}^d$ maps G_1 and G_2 to different embeddings, the Weisfeiler-Lehman graph isomorphism test also decides G_1 and G_2 are not isomorphic.

- Several well-established models are less powerful than the WL test (e.g., GCN, GraphSAGE, GAT)
- However, compared to Weisfeiler-Lehman kernel:
  - more flexible $\to$ can adapt to the learning task
  - can handle continuous node features

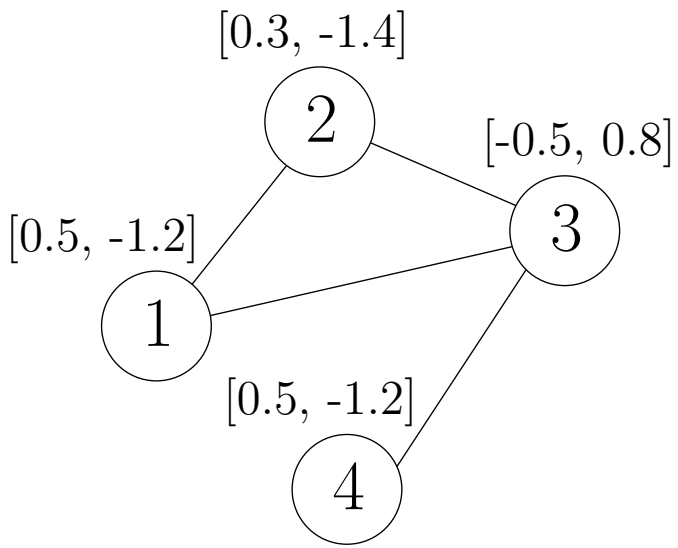
[Xu et al., ICLR'19; Morris et al., AAAI'19]

How Powerful Are Message Passing Graph Neural Networks?

Multiset: a multiset is a generalized concept of a set that allows multiple instances for its elements

When node features are from a countable universe:

- Features of all nodes can be thought of as a multiset
- Neighborhood of each node can be thought of as a multiset
- Node representations at deeper layers are also from a countable universe



The representations of the neighbors of node 3 correspond to a multiset, e.g., $\{[0.5, -1.2], [0.3, -1.4], [0.5, -1.2]\}$ is a multiset

The representations of all the nodes of the graph also correspond to a multiset, e.g., $\{[0.5, -1.2], [0.3, -1.4], [-0.5, 0.8], [0.5, -1.2]\}$ is a multiset

How Powerful Are Message Passing Graph Neural Networks?

The $\mathrm{AGGREGATE}$, $\mathrm{COMBINE}$ and $\mathrm{READOUT}$ functions of a message passing model are injective $\Rightarrow$ The model is as powerful as the WL test

The $\mathrm{AGGREGATE}$ and $\mathrm{READOUT}$ functions operate on multisets of node representations

Question: Are commonly-employed $\mathrm{AGGREGATE}$ and $\mathrm{READOUT}$ functions injective or not?

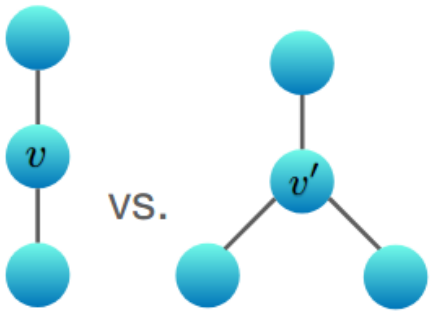
How Powerful Are Message Passing Graph Neural Networks?

The AGGREGATE, COMBINE and READOUT functions of a message passing model are injective $\Rightarrow$ The model is as powerful as the WL test

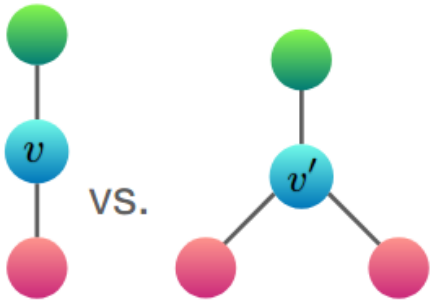
The AGGREGATE and READOUT functions operate on multisets of node representations

Question: Are commonly-employed AGGREGATE and READOUT functions injective or not?

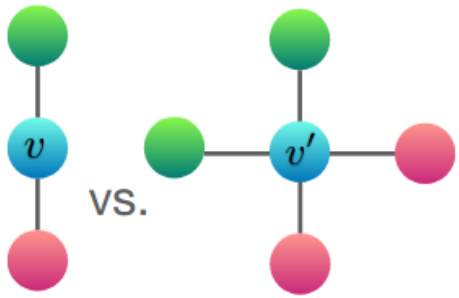
Turns out that mean and max functions are **not** injective!



(a) Mean and Max both fail



(b) Max fails



(c) Mean and Max both fail

On the other hand, sum aggregators can represent injective, in fact, universal functions over multisets

Graph Isomorphism Network (GIN)

GIN is a message passing neural network that

- models injective multiset functions for the neighborhood and node aggregation
- has the same power as the Weisfeiler-Lehman test

Step 1: GIN updates node representations as follows:

$$\mathbf{h}_v^{(t+1)} = \text{MLP}^{(t)}\left(\left(1+\epsilon^{(t)}\right)\mathbf{h}_v^{(t)} + \sum_{u \in \mathcal{N}(v)} \mathbf{h}_u^{(t)}\right)$$

where ϵ is an irrational number

Step 2: Utilizes the following graph-level readout function which uses information from all iterations of the model:

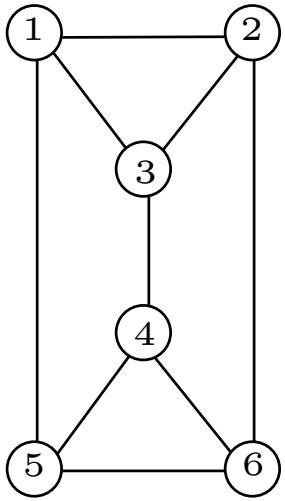
$$\mathbf{h}_G = \left[\sum_{v \in G} \mathbf{h}_v^{(0)} || \ldots || \sum_{v \in G} \mathbf{h}_v^{(T)}\right]$$

[Xu et al., ICLR'19]

Further Limitations of Standard GNNs

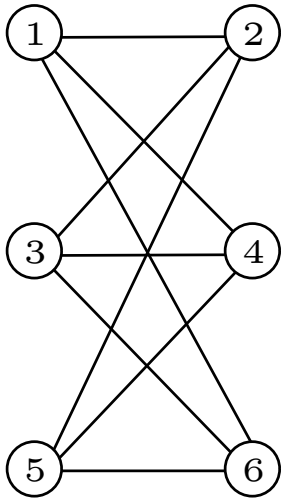
All nodes are annotated with the same feature $\mathbf{h}_i^0 = 1$ for all $i \in \{1, \ldots, 6\}$

$$\mathbf{h}_1^1 = f(\mathbf{W}_0^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_5^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_2^1 = f(\mathbf{W}_0^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_6^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_3^1 = f(\mathbf{W}_0^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_4^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_4^1 = f(\mathbf{W}_0^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_5^0 + \mathbf{W}_1^0\mathbf{h}_6^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_5^1 = f(\mathbf{W}_0^0\mathbf{h}_5^0 + \mathbf{W}_1^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_6^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_6^1 = f(\mathbf{W}_0^0\mathbf{h}_6^0 + \mathbf{W}_1^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_5^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$



G_1

$$\mathbf{h}_1^1 = f(\mathbf{W}_0^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_6^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_2^1 = f(\mathbf{W}_0^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_5^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_3^1 = f(\mathbf{W}_0^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_6^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_4^1 = f(\mathbf{W}_0^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_5^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_5^1 = f(\mathbf{W}_0^0\mathbf{h}_5^0 + \mathbf{W}_1^0\mathbf{h}_2^0 + \mathbf{W}_1^0\mathbf{h}_4^0 + \mathbf{W}_1^0\mathbf{h}_6^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$
$$\mathbf{h}_6^1 = f(\mathbf{W}_0^0\mathbf{h}_6^0 + \mathbf{W}_1^0\mathbf{h}_1^0 + \mathbf{W}_1^0\mathbf{h}_3^0 + \mathbf{W}_1^0\mathbf{h}_5^0) = f(\mathbf{W}_0^0 + 3\mathbf{W}_1^0)$$



G_2

More Powerful Graph Neural Networks

High-dimensional Weisfeiler-Lehman test of isomorphism $\rightarrow$ a generalization of the WL which colors tuples from V^k instead of nodes

- k-GNN **[Morris et al., AAAI'19]**

High-order neighborhoods

- k-hop **[Nikolentzos et al., Neural Networks 130]**

Approaches that consider the average of all possible permutations of nodes

- RelationalPooling **[Murphy et al., ICML'19]**

Coloring schemes

- CLIP **[Dasoulas et al., IJCAI'20]**

Invariant and equivariant linear layers

- k-order graph networks **[Maron et al., ICLR'19]**

Outline

Patchy-San

Patchy-San is an approach which:

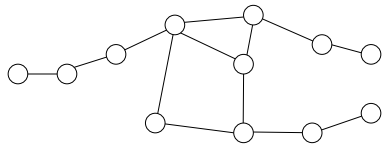
- extracts a set of subgraphs from each graph
- imposes an ordering on the nodes of each subgraph
  $\hookrightarrow$ no need to apply permutation invariant functions
- feeds the extracted subgraphs into a conventional CNN

The method can be decomposed in three main steps:

- select a node
- construct its neighborhood
- normalize the selected subgraph (i.e., order the neighboring nodes)

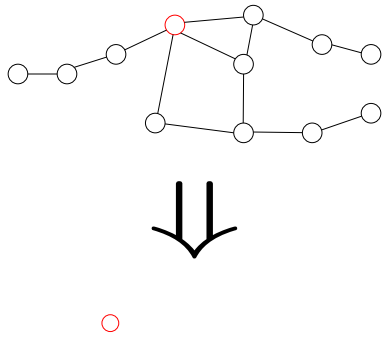
[Niepert et al, ICML'16]

Patchy-San



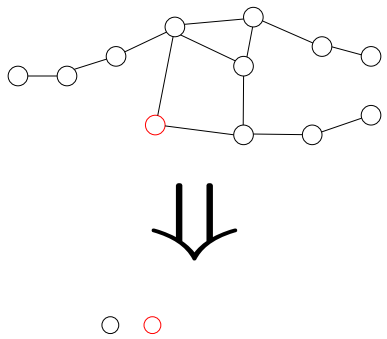
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)



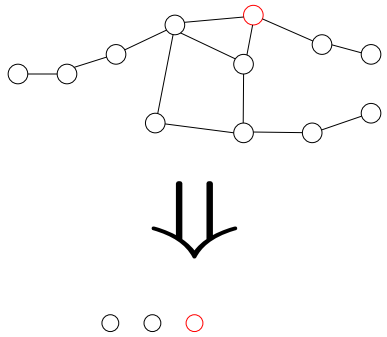
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)



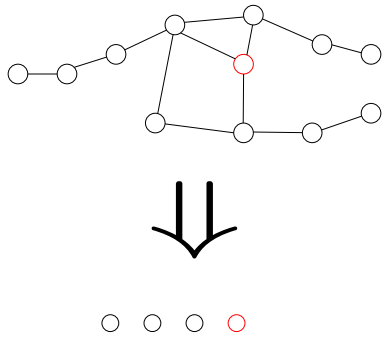
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)



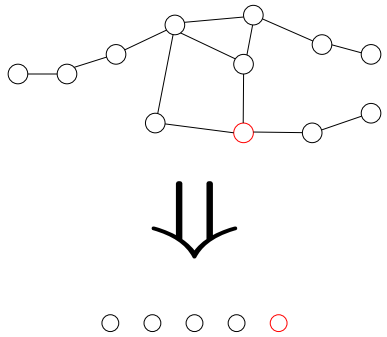
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)



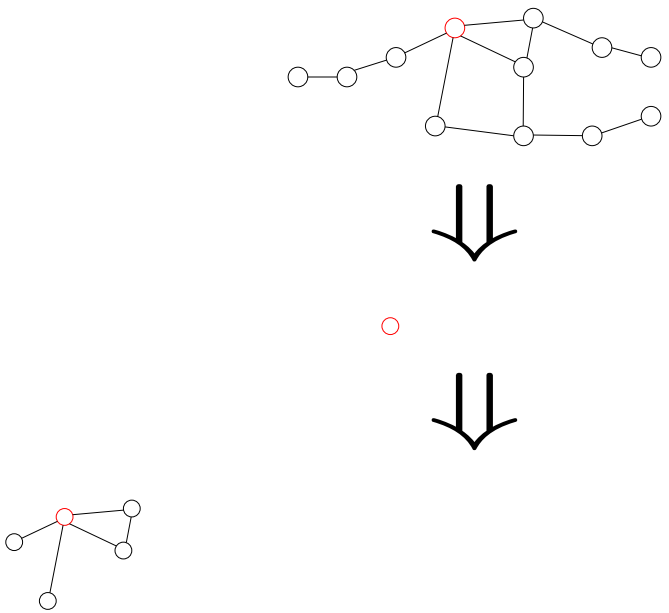
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)



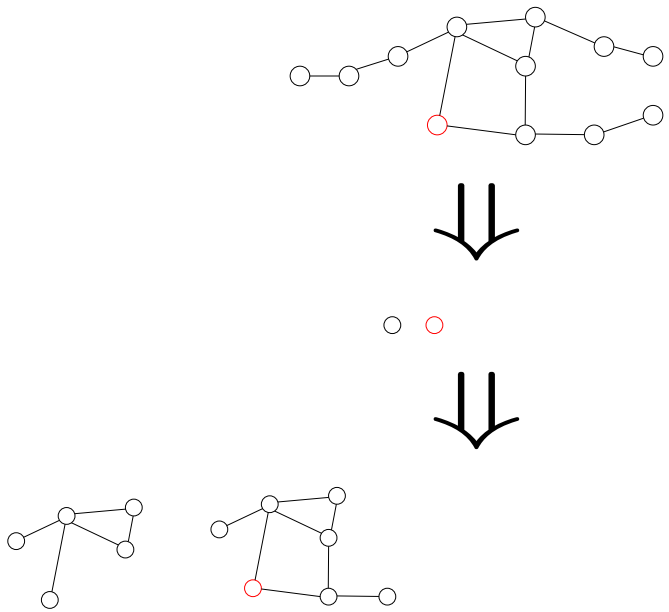
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e.g., using BFS)



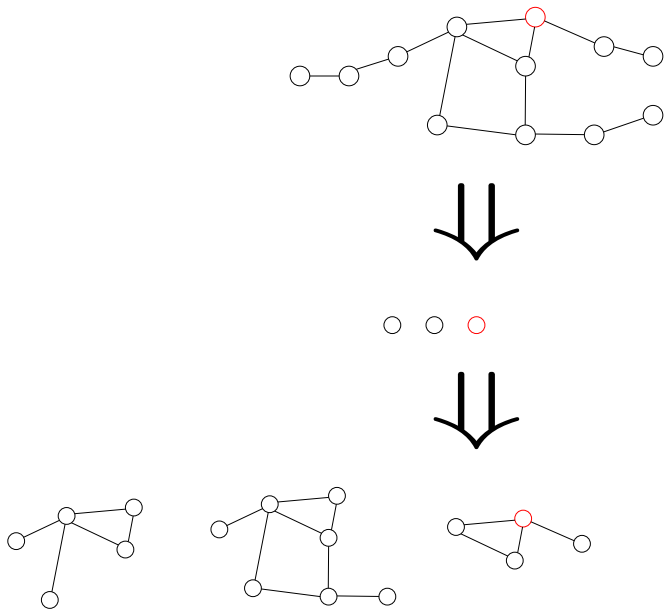
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e.g., using BFS)



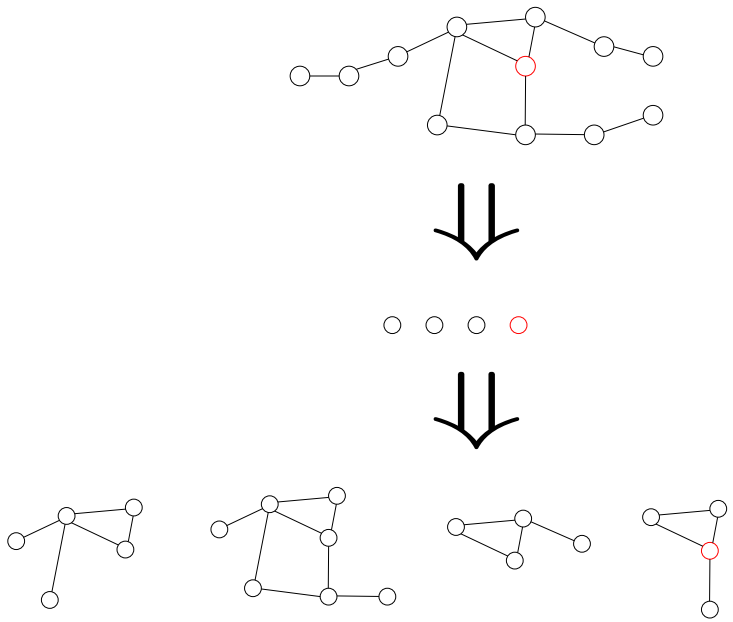
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e.g., using BFS)



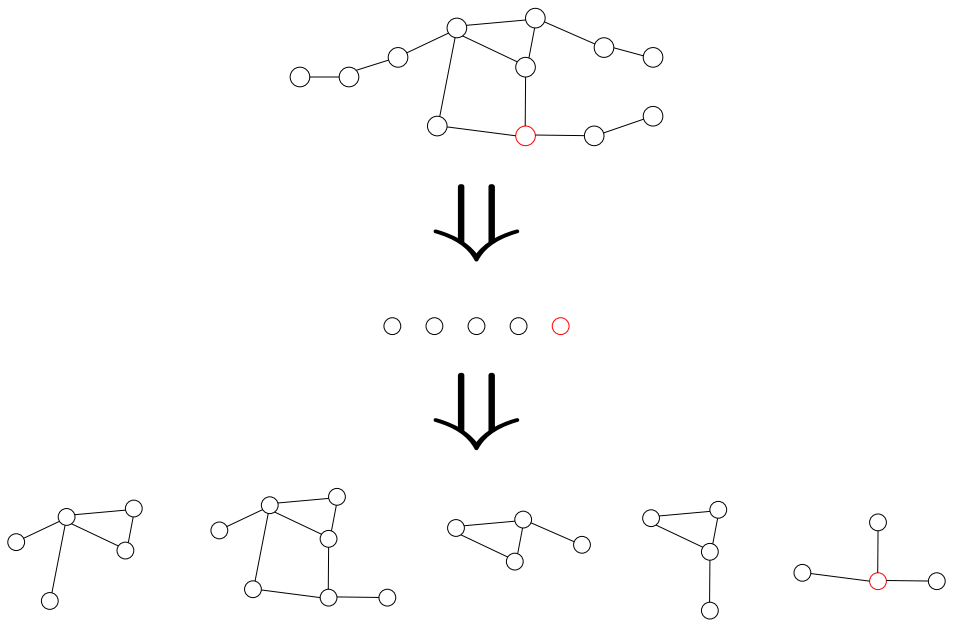
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e.g., using BFS)



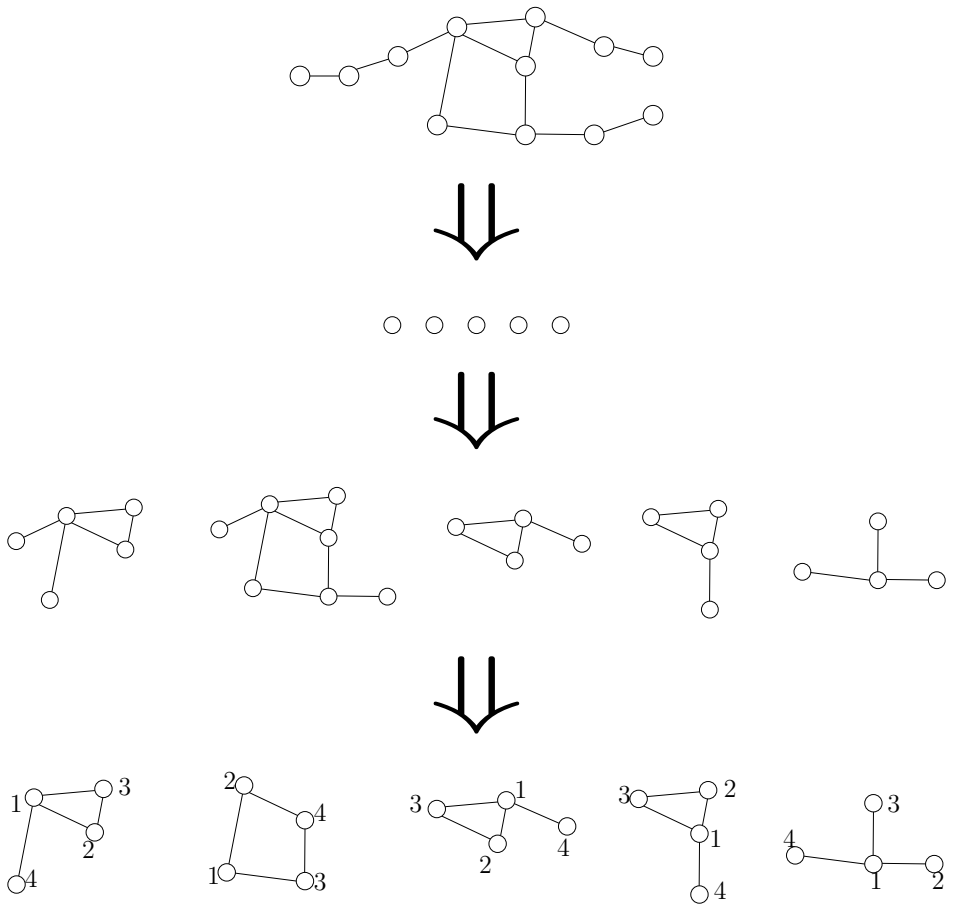
Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e.g., using BFS)

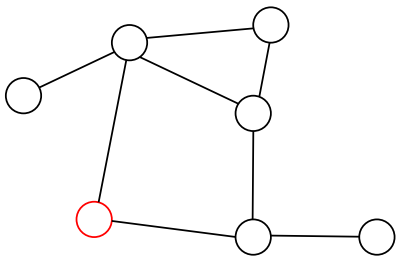


Patchy-San

- Select $w = 5$ nodes (e.g., using a centrality measure)
- For each selected node, extract a neighborhood of at least $k = 4$ nodes (e.g., using BFS)
- Keep exactly $k = 4$ nodes and arrange them in an order

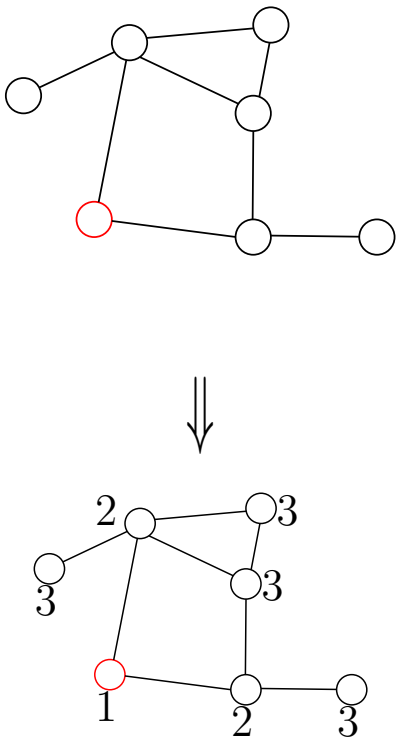


Neighborhood Normalization



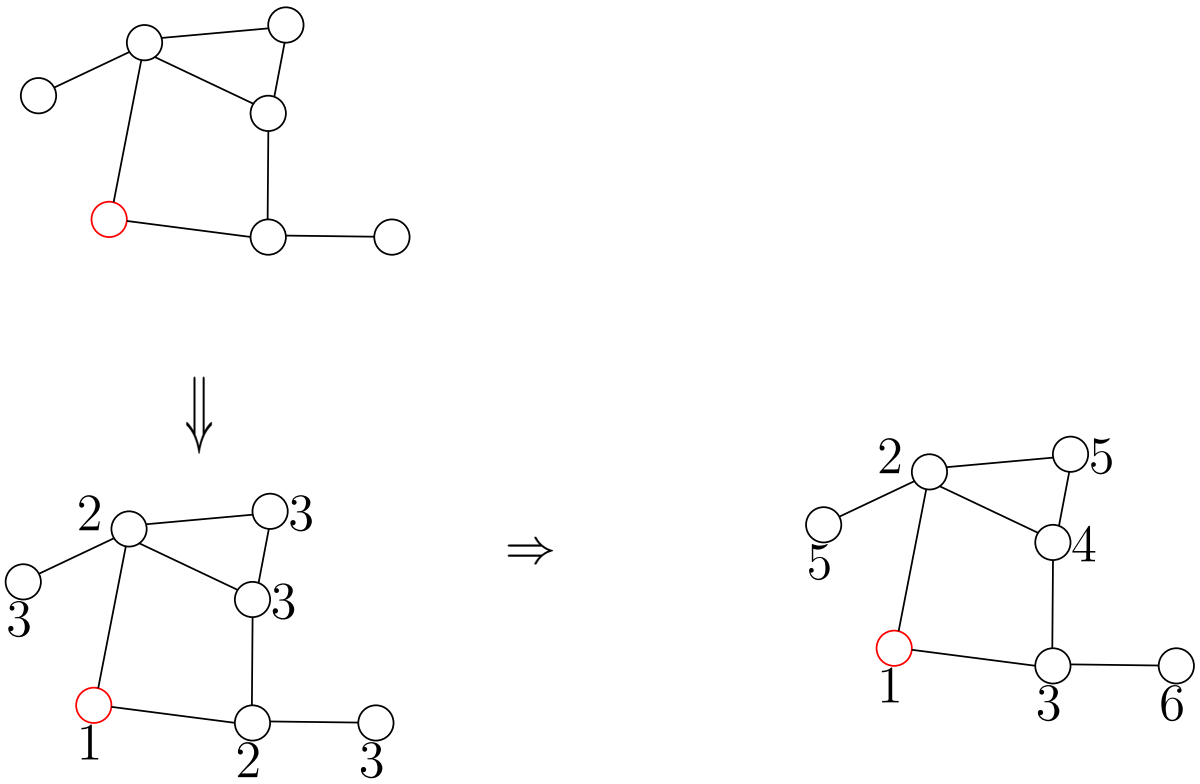
Neighborhood Normalization

- Distance to root



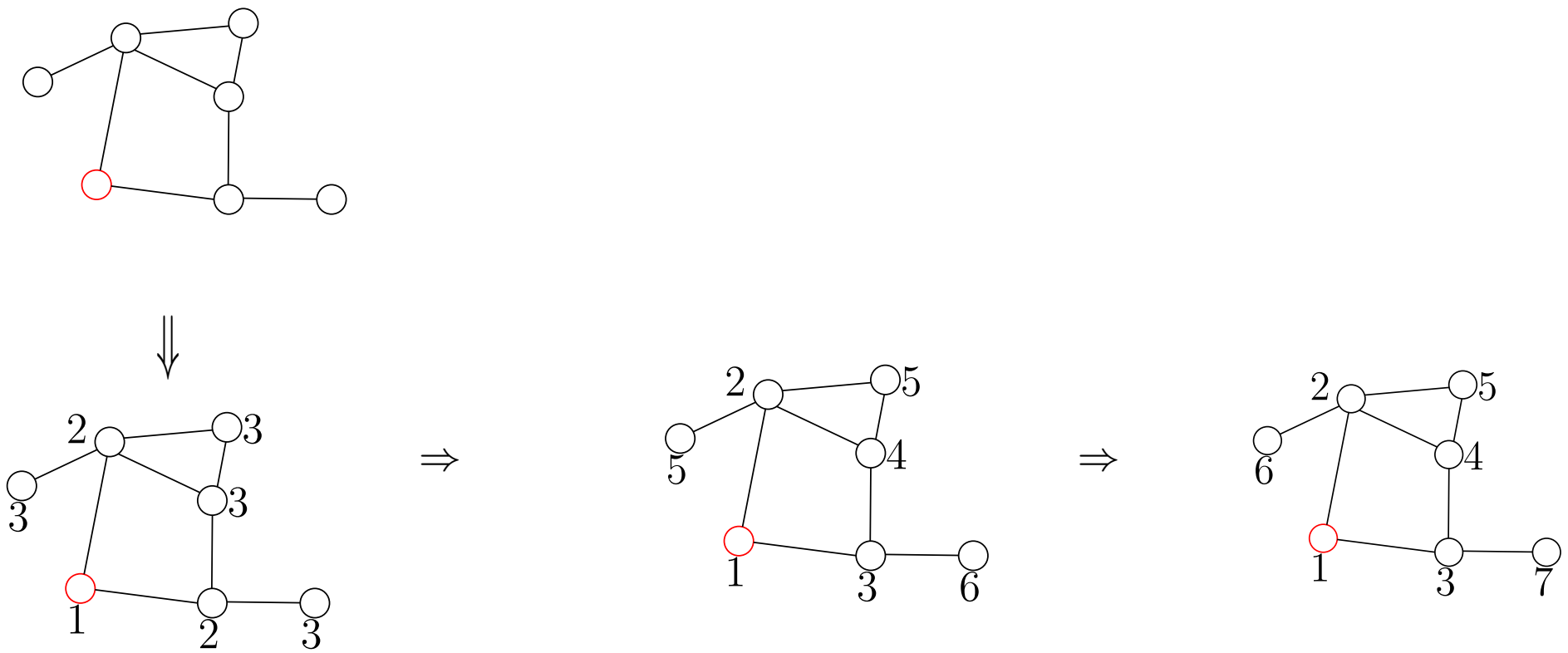
Neighborhood Normalization

- Distance to root
- Use centrality measures



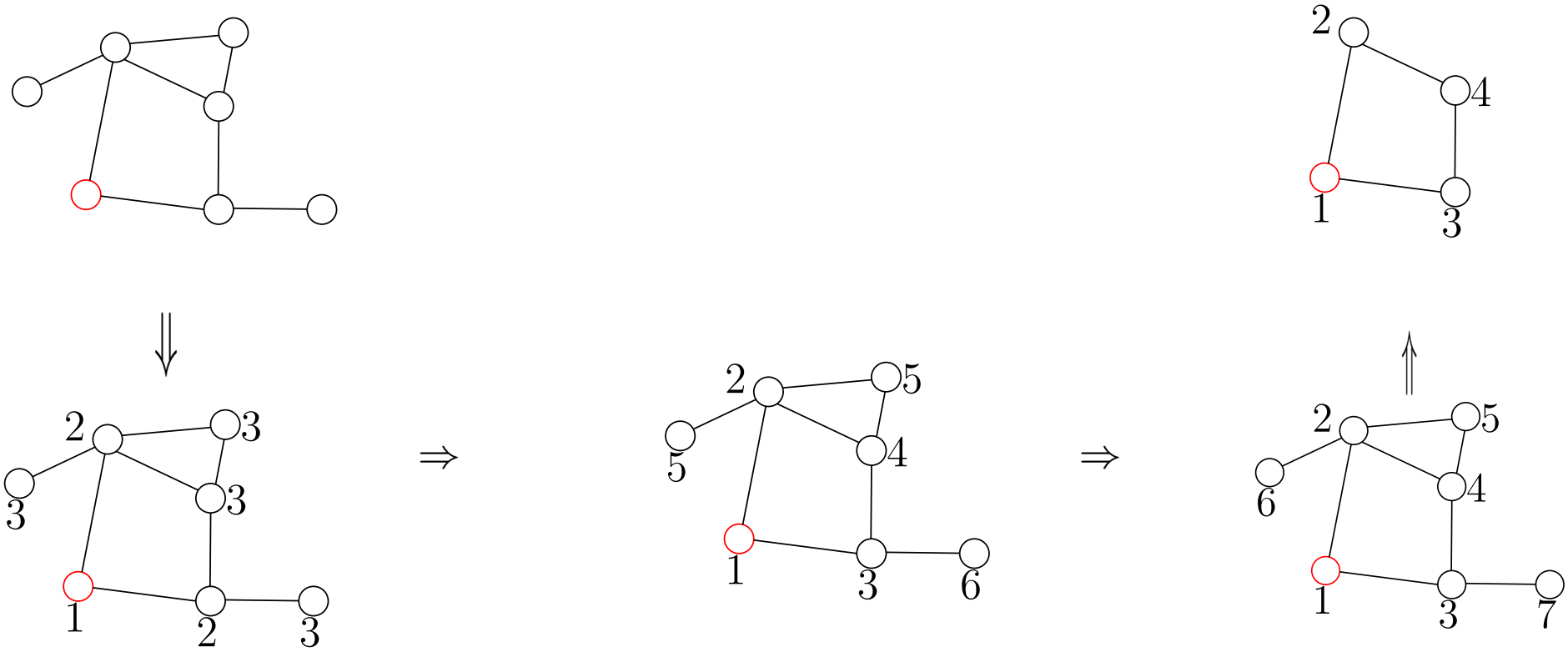
Neighborhood Normalization

- Distance to root
- Use centrality measures
- Canonicalization



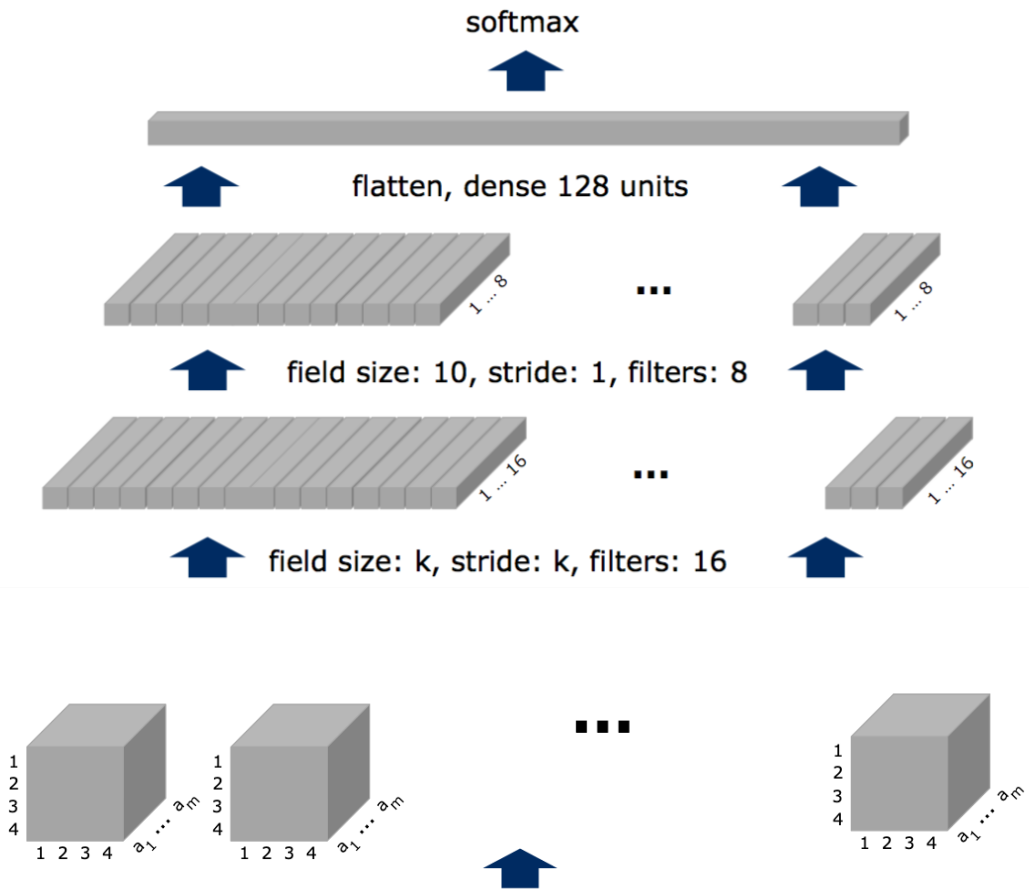
Neighborhood Normalization

- Distance to root
- Use centrality measures
- Canonicalization
- Remove extra nodes



CNN Architecture of Patchy-San

- The $k \times k$ adjacency matrices of the neighborhoods of the w vertices given as input to a CNN: filters of size $k \times k$
- In case of node labels (n discrete labels) $\rightarrow$ create a $k \times n$ matrix to serve as an indicator matrix: Each indicator matrix convolved with filters of size $k \times n$
- These two types of matrices correspond to different channels of the CNN
- Two convolutional layers in total, followed by a fully-connected feedforward network



Random Walk Graph Neural Network (RWNN)

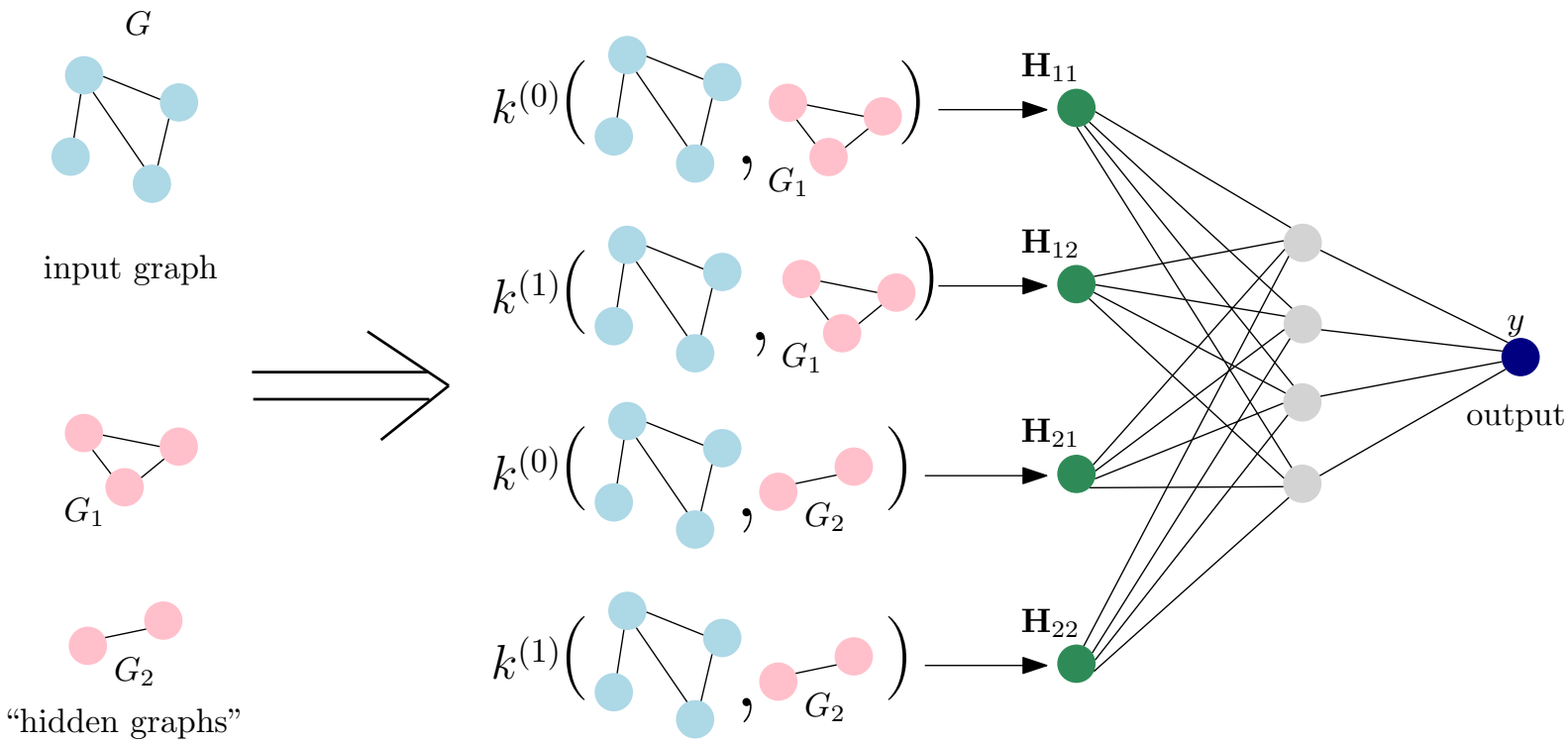
Idea: Existing graph neural networks are counter-intuitive
$\hookrightarrow$ features they learn are in the form of vectors

The Random Walk Graph Neural Network is a model

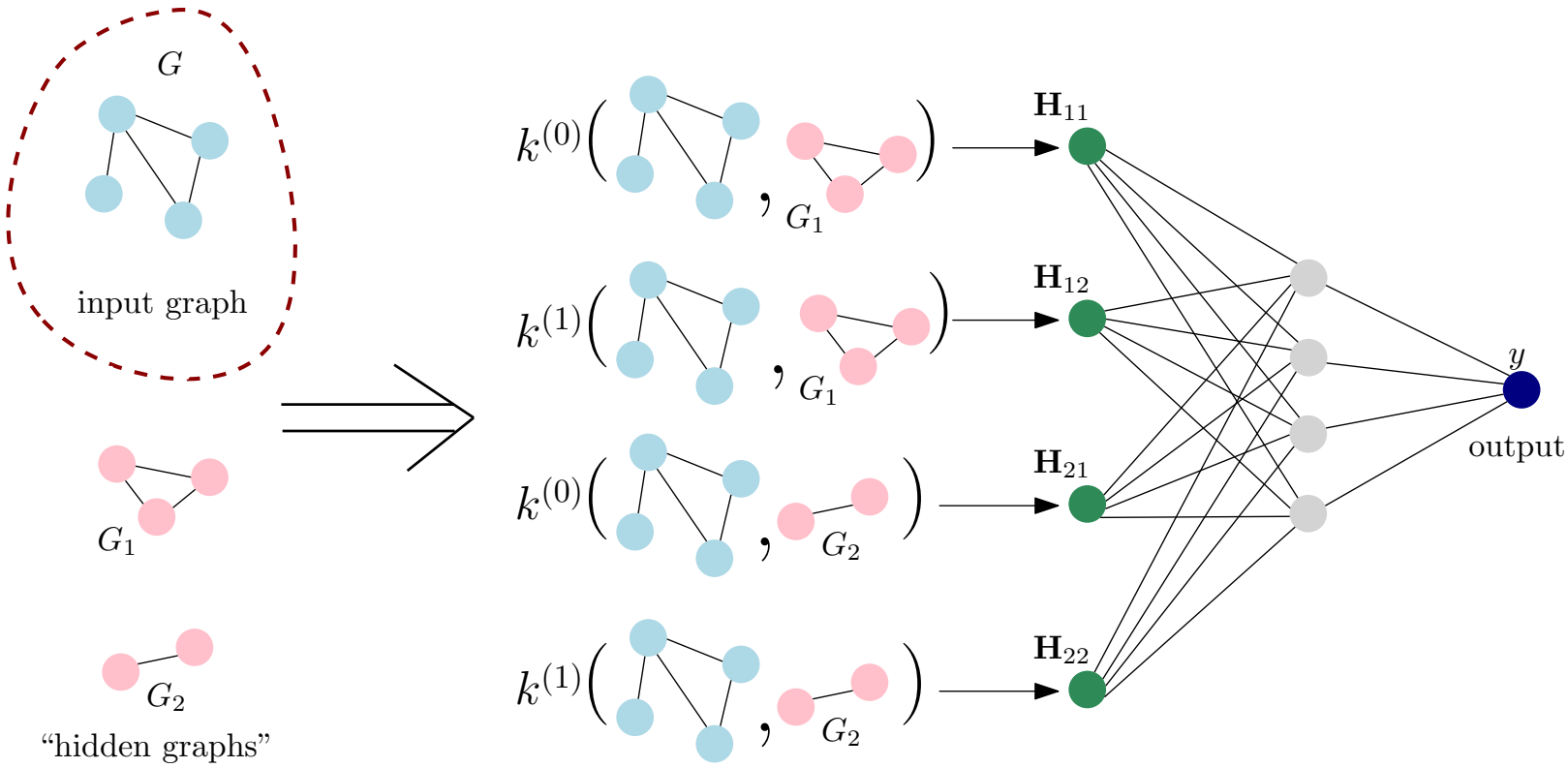
- employs a random walk kernel to learn *graph features* (in the form of fixed-size graphlets) that contribute to interpretable/intuitive graph representations
- An efficient computation scheme to reduce the time and space complexity of the proposed model

[Nikolentzos and Vazirgiannis, NeurIPS'20]

Random Walk Graph Neural Network (RWNN)

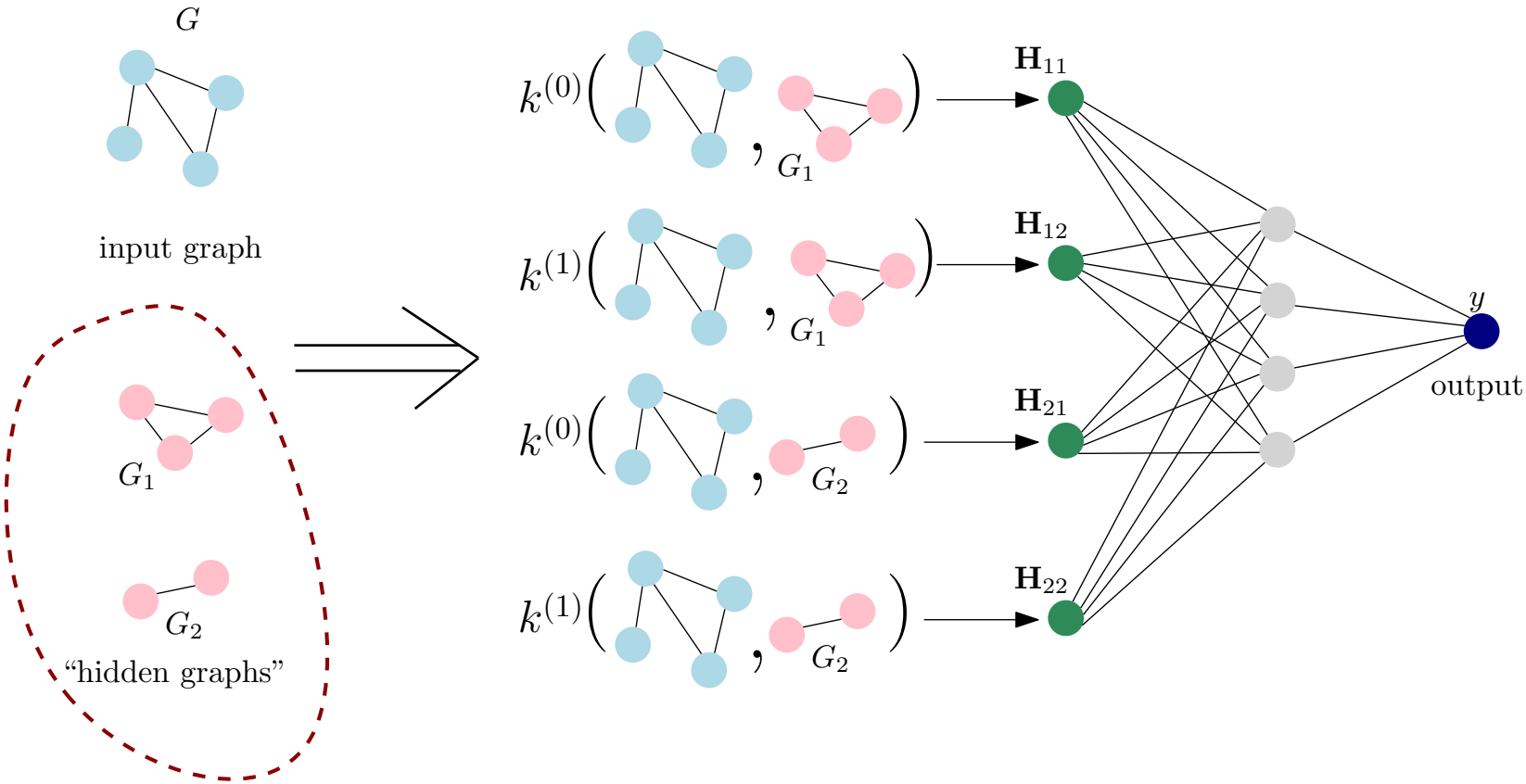


Random Walk Graph Neural Network (RWNN)



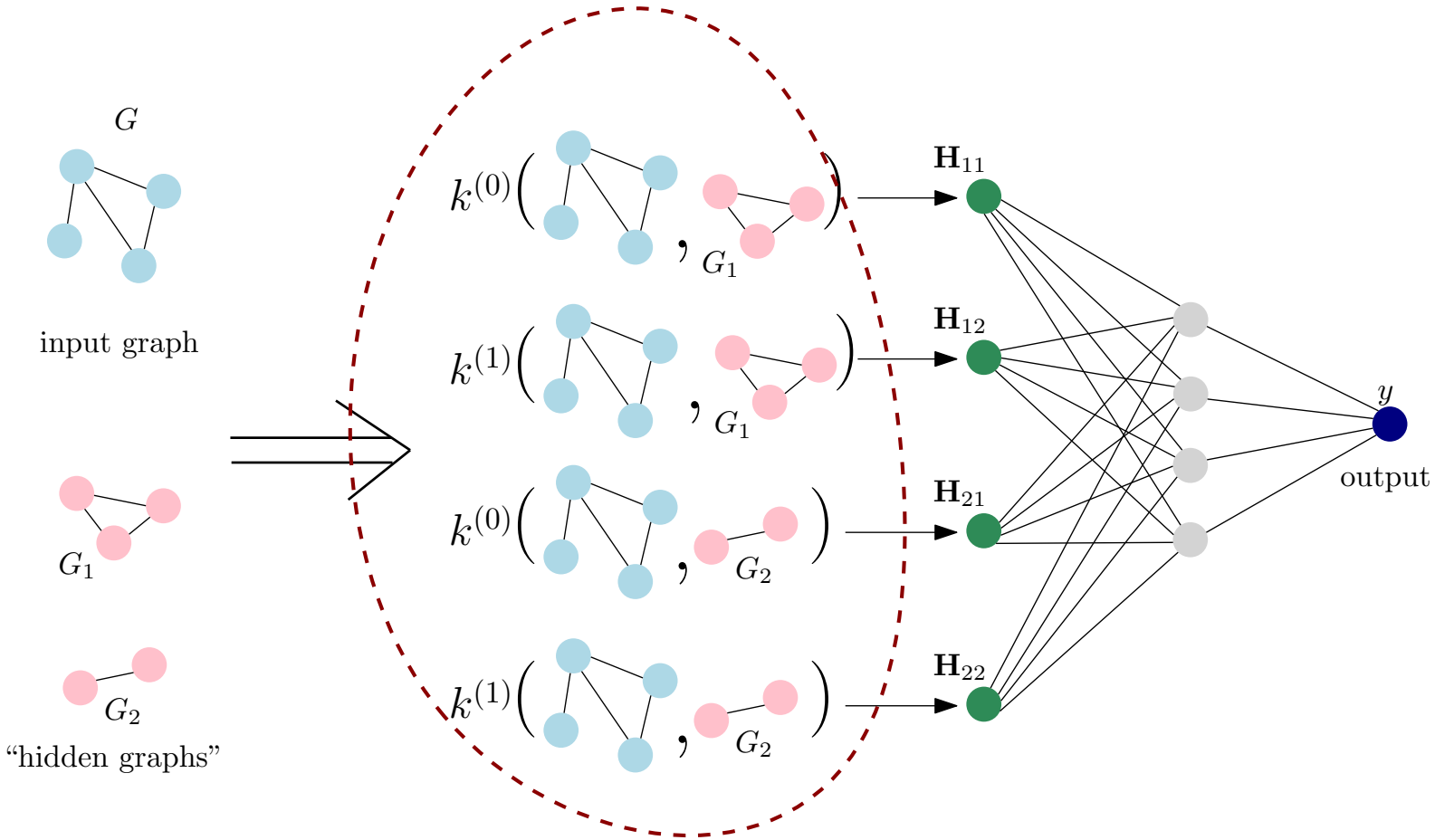
Given an input graph G

Random Walk Graph Neural Network (RWNN)



and a set of trainable "hidden graphs" $G_1, G_2, \ldots$

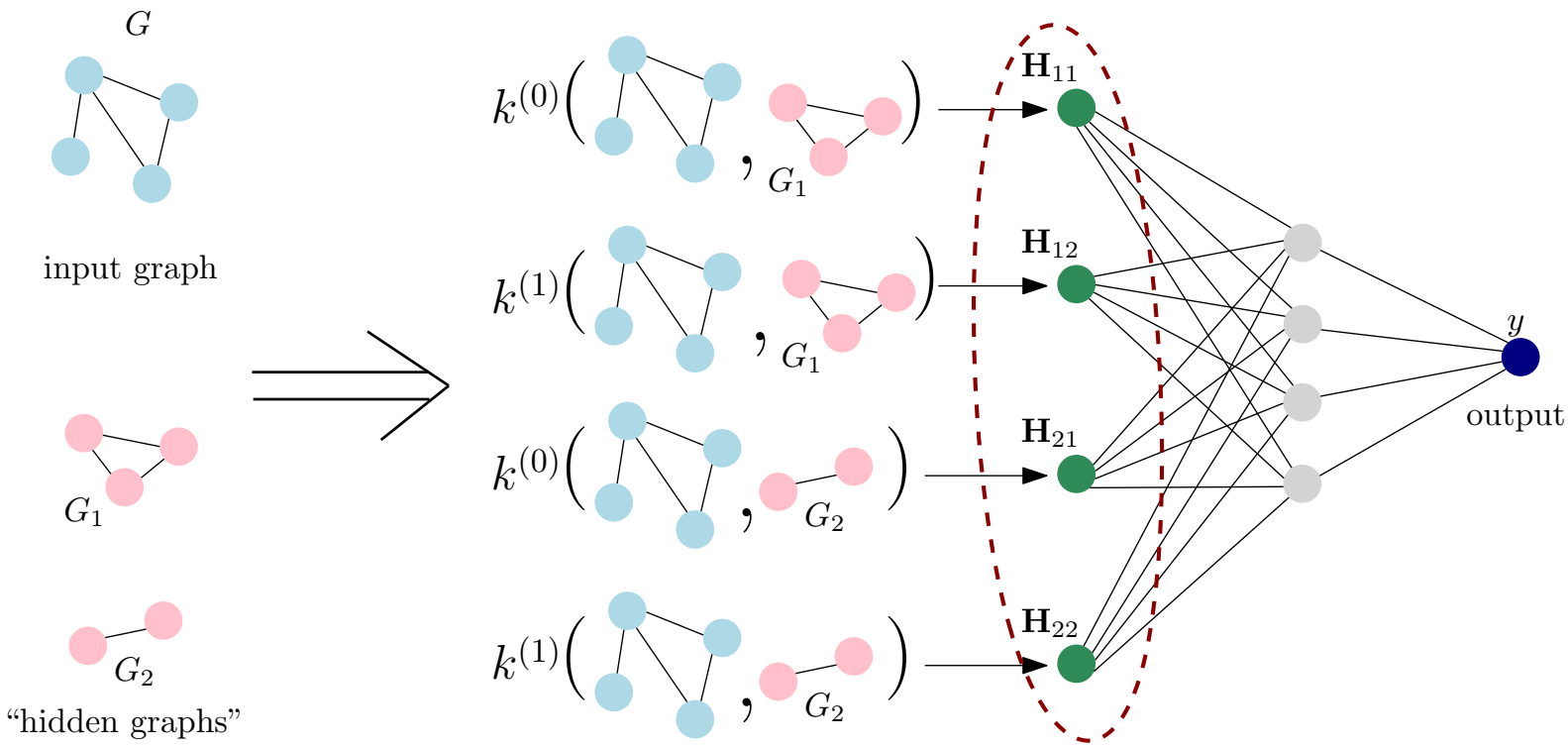
Random Walk Graph Neural Network (RWNN)



The model computes the following kernel between the input graph G and each "hidden graph" $G_{\mathtt{i}}$:

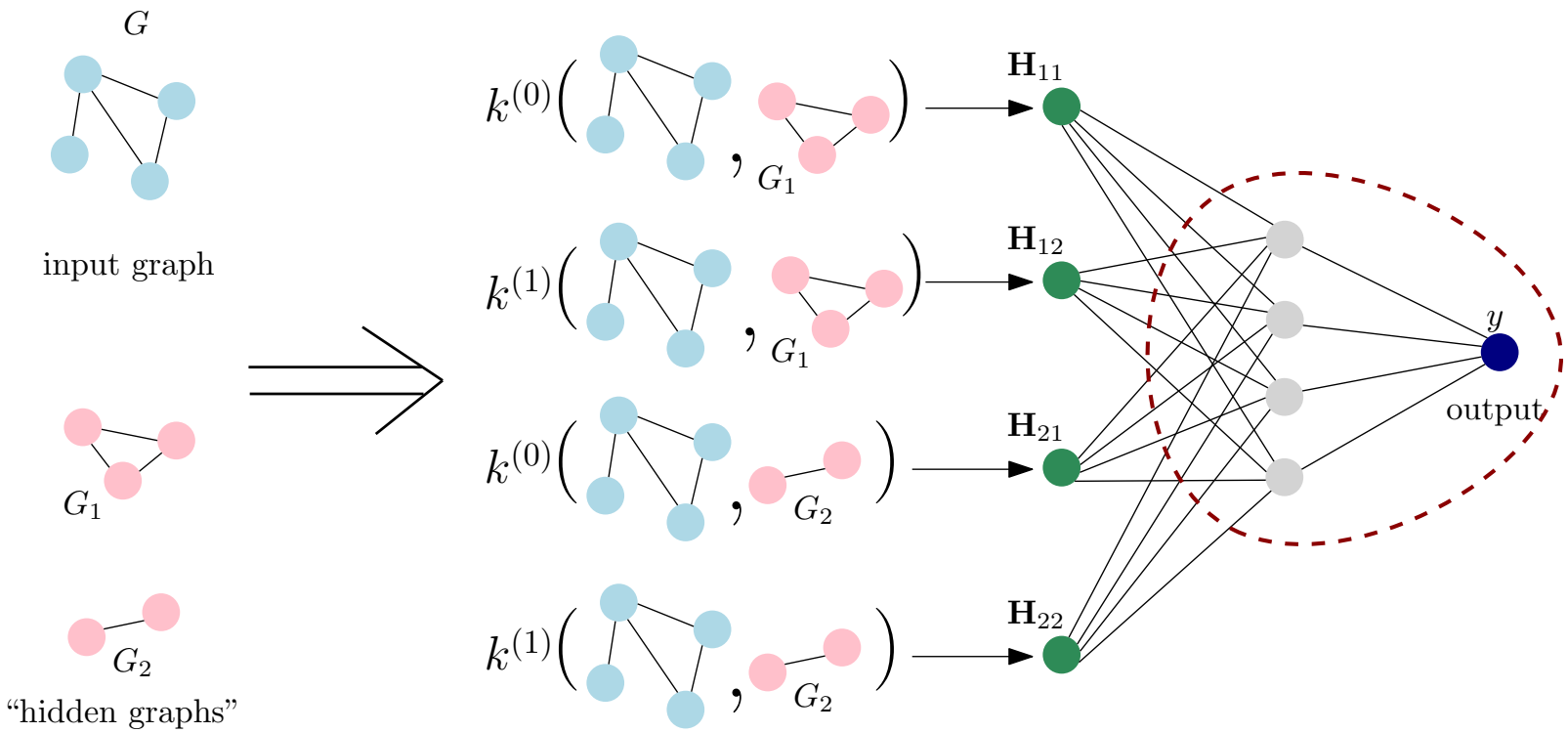
$$k^{(p)}(G, G_{\mathtt{i}}) = \sum_{i=1}^{|V_\times|} \sum_{j=1}^{|V_\times|} \mathbf{s}_i \mathbf{s}_j \left[\mathbf{A}_\times^p\right]_{ij}$$

Random Walk Graph Neural Network (RWNN)



For each input graph G, we build a matrix $\mathbf{H} \in \mathbb{R}^{N \times P+1}$ where $\mathbf{H}_{ij} = k^{(j-1)}(G, G_i)$

Random Walk Graph Neural Network (RWNN)



Matrix **H** is flattened and fed into a fully-connected neural network to produce the output

Experimental Evaluation - Graph Classification

We evaluated RWNN on the following standard graph classification datasets from bio/chemo-informatics and social networks

Dataset	ENZYMES	NCI1	PROTEINS	D&D	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
Max # vertices	126	111	620	5,748	136	89	3,782	3,648	492
Min # vertices	2	3	4	30	12	7	6	22	32
Average # vertices	32.63	29.87	39.05	284.32	19.77	13.00	429.61	508.50	74.49
Max # edges	149	119	1,049	14,267	1,249	1,467	4,071	4,783	40,119
Min # edges	1	2	5	63	26	12	4	21	60
Average # edges	62.14	32.30	72.81	715.66	96.53	65.93	497.75	594.87	2,457.34
# labels	3	37	3	82	-	-	-	-	-
# graphs	600	4,110	1,113	1,178	1,000	1,500	2,000	4,999	5,000
# classes	6	2	2	2	2	3	2	5	3

Experimental Setup

Published as a conference paper at ICLR 2020

A Fair Comparison of Graph Neural Networks for Graph Classification

Federico Errica*
Department of Computer Science
University of Pisa
federico.errica@phd.unipi.it

Marco Podda*
Department of Computer Science
University of Pisa
marco.podda@di.unipi.it

Davide Bacciu*
Department of Computer Science
University of Pisa
bacciu@di.unipi.it

Alessio Micheli*
Department of Computer Science
University of Pisa
micheli@di.unipi.it

- 10-fold CV for model assessment and an inner holdout technique with a 90%/10% training/validation split for model selection
- After each model selection $\rightarrow$ train 3 times on the whole training fold, holding out a random fraction (10%) of the data to perform early stopping
- Final test fold score obtained as the mean of these 3 runs

Baselines

- Graph Kernels
  - Shortest path kernel (SP) **[Borgwardt and Kriegel, ICDM'05]**
  - Graphlet kernel (GR) **[Shervashidze et al., AISTATS'09]**
  - Weisfeiler-Lehman subtree kernel (WL) **[Shervashidze et al., JMLR'11]**
- Graph Neural Networks
  - DGCNN **[Zhang et al., AAAI'18]**
  - DiffPool **[Ying et al., NIPS'18]**
  - ECC **[Simonovsky and Komodakis, CVPR'17]**
  - GIN **[Xu et al., ICLR'19]**
  - GraphSAGE **[Hamilton et al., NIPS'17]**

Graph Classification - Real World Datasets

	MUTAG	D&D	NCI1	PROTEINS	ENZYMES
SP	80.2 (± 6.5)	**78.1** (± 4.1)	72.7 (± 1.4)	**75.3** (± 3.8)	38.3 (± 8.0)
GR	80.8 (± 6.4)	75.4 (± 3.4)	61.8 (± 1.7)	71.6 (± 3.1)	25.1 (± 4.4)
WL	84.6 (± 8.3)	**78.1** (± 2.4)	**84.8** (± 2.5)	73.8 (± 4.4)	50.3 (± 5.7)
DGCNN	84.0 (± 6.7)	76.6 (± 4.3)	76.4 (± 1.7)	72.9 (± 3.5)	38.9 (± 5.7)
DiffPool	79.8 (± 7.1)	75.0 (± 3.5)	76.9 (± 1.9)	73.7 (± 3.5)	59.5 (± 5.6)
ECC	75.4 (± 6.2)	72.6 (± 4.1)	76.2 (± 1.4)	72.3 (± 3.4)	29.5 (± 8.2)
GIN	84.7 (± 6.7)	75.3 (± 2.9)	80.0 (± 1.4)	73.3 (± 4.0)	**59.6** (± 4.5)
GraphSAGE	83.6 (± 9.6)	72.9 (± 2.0)	76.0 (± 1.8)	73.0 (± 4.5)	58.2 (± 6.0)
1-step RWNN	**89.2** (± 4.3)	77.6 (± 4.7)	71.4 (± 1.8)	74.7 (± 3.3)	56.7 (± 5.2)
2-step RWNN	88.1 (± 4.8)	76.9 (± 4.6)	73.0 (± 2.0)	74.1 (± 2.8)	57.4 (± 4.9)
3-step RWNN	88.6 (± 4.1)	77.4 (± 4.9)	73.9 (± 1.3)	74.3 (± 3.3)	57.6 (± 6.3)

	IMDB BINARY	IMDB MULTI	REDDIT BINARY	REDDIT MULTI-5K	COLLAB
SP	57.7 (± 4.1)	39.8 (± 3.7)	89.0 (± 1.0)	51.1 (± 2.2)	**79.9** (± 2.7)
GR	63.3 (± 2.7)	39.6 (± 3.0)	76.6 (± 3.3)	38.1 (± 2.3)	71.1 (± 1.4)
WL	**72.8** (± 4.5)	**51.2** (± 6.5)	74.9 (± 1.8)	49.6 (± 2.0)	78.0 (± 2.0)
DGCNN	69.2 (± 3.0)	45.6 (± 3.4)	87.8 (± 2.5)	49.2 (± 1.2)	71.2 (± 1.9)
DiffPool	68.4 (± 3.3)	45.6 (± 3.4)	89.1 (± 1.6)	53.8 (± 1.4)	68.9 (± 2.0)
ECC	67.7 (± 2.8)	43.5 (± 3.1)	OOR	OOR	OOR
GIN	71.2 (± 3.9)	48.5 (± 3.3)	89.9 (± 1.9)	**56.1** (± 1.7)	75.6 (± 2.3)
GraphSAGE	68.8 (± 4.5)	47.6 (± 3.5)	84.3 (± 1.9)	50.0 (± 1.3)	73.9 (± 1.7)
1-step RWNN	70.8 (± 4.8)	47.8 (± 3.8)	**90.4** (± 1.9)	51.7 (± 1.5)	71.7 (± 2.1)
2-step RWNN	70.6 (± 4.4)	48.8 (± 2.9)	90.3 (± 1.8)	51.7 (± 1.4)	71.3 (± 2.1)
3-step RWNN	70.7 (± 3.9)	47.8 (± 3.5)	89.7 (± 1.2)	53.4 (± 1.6)	71.9 (± 2.5)

Outline

What is a set?

A set is a well-defined collection of distinct objects

Complex data sets decomposed into sets of simpler objects

↪ NLP: documents as sets of word embeddings

↪ Graph Mining: graphs as sets of node embeddings

↪ Computer Vision: images as sets of local features

Machine learning on sets has attracted a lot of attention recently

- Set classification
- Set regression

Set Classification

$S_1 = \{1, 4, 2\}$

$y_1 = -1$

$S_2 = \{5, 0, 8, 10\}$

$y_2 = -1$

$S_6 = \{2, 6, 3, 5\}$

$y_6 = ???$

$S_3 = \{3, 7\}$

$y_3 = 1$

$S_4 = \{3, 5, 6\}$

$y_4 = 1$

$S_7 = \{4, 2, 5\}$

$y_7 = ???$

$S_5 = \{5\}$

$y_5 = -1$

- Let $\mathcal{X}$ be a set
- Input data $S \in 2^{\mathcal{X}}$
- Output $y \in \{-1, 1\}$
- Training set $\{(S_1, y_1), \ldots, (S_n, y_n)\}$
- Goal: estimate a function $f : 2^{\mathcal{X}} \to \in \{-1, 1\}$ to predict y from $f(S)$

Limitations of Standard Machine Learning Models

Conventional machine learning models cannot handle sets:

- expect fixed dimensional data instances
  $\hookrightarrow$ sets allowed to vary in the number of elements

- not invariant to permutations of features
  - a learning algorithm for sets needs to produce identical representations for any permutation of the elements of an input set
  - for instance, a model f needs to satisfy the following for any permutation π of the set's elements:

$$f(\{x_1, \ldots, x_M\}) = f(\{x_{\pi(1)}, \ldots, x_{\pi(M)}\})$$

Outline

Neural Networks for Sets

Recent approaches:

- unordered sets $\rightarrow$ ordered sequences $\rightarrow$ RNN **[Vinyals et al., ICLR'16]**
- DeepSets **[Zaheer et al., NIPS'17]** and PointNet **[Qi et al., CVPR'17]** transform the vectors of the sets into new representations, then apply permutation-invariant functions
- PointNet++ **[Qi et al., NIPS'17]** and SO-Net **[Li et al., CVPR'18]** apply PointNet hierarchically in order to better capture local structures
- Set Transformer **[Lee et al., ICML'19]**, a neural network that uses self-attention to model interactions among the elements of the input set
- RepSet **[Skianis et al., AISTATS'20]**, a neural network that generates representations for sets by comparing them against some trainable sets

SOTA in supervised learning tasks:

- regression: population statistic estimation, sum of digits
- classification: point cloud classification, outlier detection

Theorem (Zaheer et al., NIPS'17)

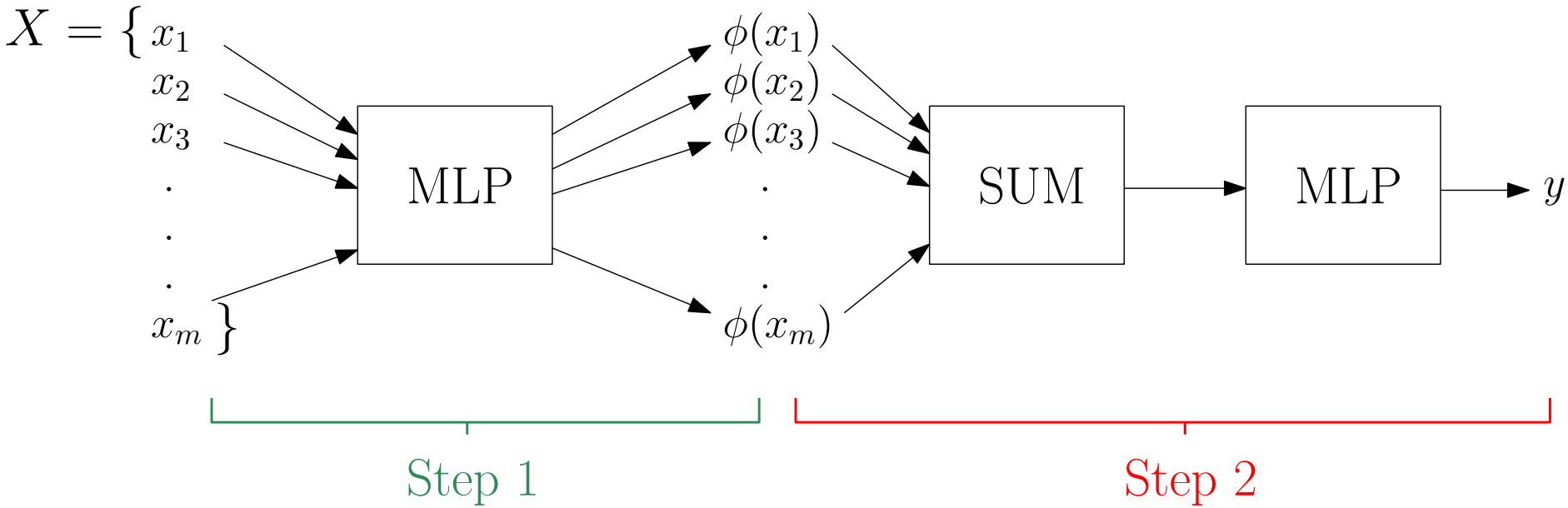
If $\mathfrak{X}$ is a countable set and $\mathcal{Y} = \mathbb{R}$, then a function $f(X)$ operating on a set X having elements from $\mathfrak{X}$ is a valid set function, i.e., invariant to the permutation of instances in X, if and only if it can be decomposed in the form $\rho(\sum_{x \in X} \phi(x))$, for suitable transformations ϕ and ρ.

DeepSets achieves permutation invariance by replacing ϕ and ρ with multi-layer perceptrons (universal approximators)

DeepSets consist of the following two steps:

1. Each element x_i of each set is transformed (possibly by several layers) into some representation $\phi(x_i)$
2. The representations $\phi(x_i)$ are added up and the output is processed using the ρ network in the same manner as in any deep network (e.g., fully connected layers, nonlinearities, etc.)

DeepSets



Step 1: The elements $x_1, \ldots, x_m$ of the input set X are transformed into representations $\phi(x_1), \ldots, \phi(x_m)$

Step 2: A representation for the entire set is produced as $z_X = \phi(x_1) + \ldots + \phi(x_m)$ and is also transformed as follows $y = \rho(z_X)$ to produce the output

Experiments - Point Cloud Classification

Objective: classify point-clouds
$\hookrightarrow$ point-clouds are sets of low-dimensional vectors (typically 3-dimensional vectors representing the x, y, z-coordinates of objects)

Dataset: ModelNet40 $\rightarrow$ consists of 3-dimensional representations of $9,843$ training and $2,468$ test instances belonging to 40 classes of objects

Setup: point-clouds directly passed on to DeepSets

Model	Instance Size	Representation	Accuracy
3DShapeNets [25]	30^3	voxels (using convolutional deep belief net)	77%
VoxNet [26]	32^3	voxels (voxels from point-cloud + 3D CNN)	83.10%
MVCNN [21]	$164\times164\times12$	multi-vew images (2D CNN + view-pooling)	90.1%
VRN Ensemble [27]	32^3	voxels (3D CNN, variational autoencoder)	95.54%
3D GAN [28]	64^3	voxels (3D CNN, generative adversarial training)	83.3%
DeepSets	$\mathbf{5000\times3}$	point-cloud	$90\pm.3\%$
DeepSets	$\mathbf{100\times3}$	point-cloud	$82\pm2\%$

Experiments - Text Concept Retrieval

Objective: retrieve words belonging to a "concept" given few words from the concept

Example: given the set of words $\{tiger, lion, cheetah\}$, retrieve other related words like jaguar and puma, which all belong to the concept of big cats

Setup: query word added to set and new set fed to DeepSet which produces a score

Method	LDA-1k (Vocab = 17k)					LDA-3k (Vocab = 38k)					LDA-5k (Vocab = 61k)				
	Recall (%)			**MRR**	**Med.**	**Recall (%)**			**MRR**	**Med.**	**Recall (%)**			**MRR**	**Med.**
	@10	**@100**	**@1k**			**@10**	**@100**	**@1k**			**@10**	**@100**	**@1k**		
Random	0.06	0.6	5.9	0.001	8520	0.02	0.2	2.6	0.000	28635	0.01	0.2	1.6	0.000	30600
Bayes Set	1.69	11.9	37.2	0.007	2848	2.01	14.5	36.5	0.008	3234	1.75	12.5	34.5	0.007	3590
w2v Near	**6.00**	**28.1**	**54.7**	0.021	**641**	4.80	21.2	43.2	0.016	2054	4.03	16.7	35.2	0.013	6900
NN-max	4.78	22.5	53.1	0.023	779	5.30	24.9	54.8	0.025	672	4.72	21.4	47.0	0.022	1320
NN-sum-con	4.58	19.8	48.5	0.021	1110	5.81	27.2	60.0	**0.027**	453	4.87	23.5	53.9	0.022	731
NN-max-con	3.36	16.9	46.6	0.018	1250	5.61	25.7	57.5	0.026	570	4.72	22.0	51.8	0.022	877
DeepSets	5.53	24.2	54.3	**0.025**	696	**6.04**	**28.5**	**60.7**	**0.027**	**426**	**5.54**	**26.1**	**55.5**	**0.026**	**616**

Experiments - Image Tagging

Objective: retrieve all relevant tags corresponding to an image

Setup: features of the image are concatenated to the embeddings of the tags, and then the whole set is passed on to DeepSets to assign a single score to the set

Method	ESP game				IAPRTC-12.5			
	P	R	F1	N+	P	R	F1	N+
Least Sq.	35	19	25	215	40	19	26	198
MBRM	18	19	18	209	24	23	23	223
JEC	24	19	21	222	29	19	23	211
FastTag	46	22	30	**247**	**47**	26	34	**280**
Least Sq.(D)	**44**	32	**37**	232	46	30	36	218
FastTag(D)	**44**	32	**37**	229	46	**33**	**38**	254
DeepSets	39	**34**	36	246	42	31	36	247

Experiments - Set Anomaly Detection

Objective: find the anomalous face in each set

Architecture: consists of 9 2d-convolution and max-pooling layers followed by the DeepSets model, and a softmax layer that assigns a probability value to each set member

THANK YOU !

Acknowledgements
G. Salha

http://www.lix.polytechnique.fr/dascim/