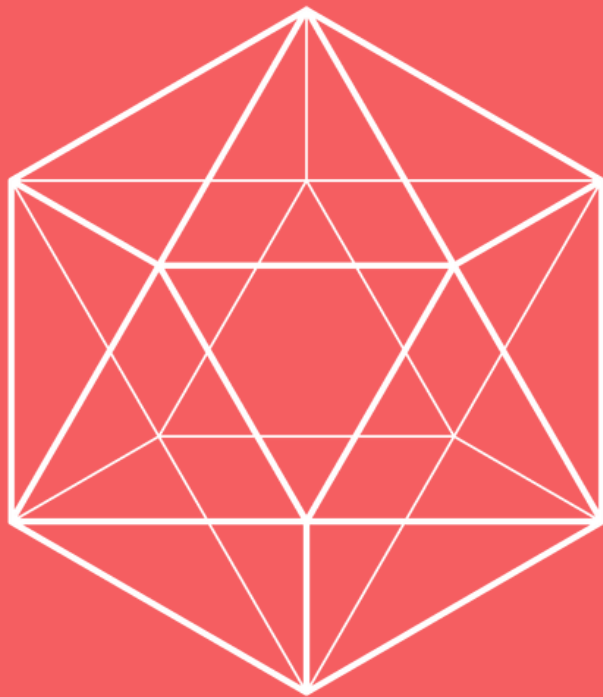


Graph Representation Learning

Part II. Graph Neural Networks

Outline

- Motivations and Challenges
- The GNN Framework
- Generalized Aggregation Methods



Motivations & Challenges

Motivation

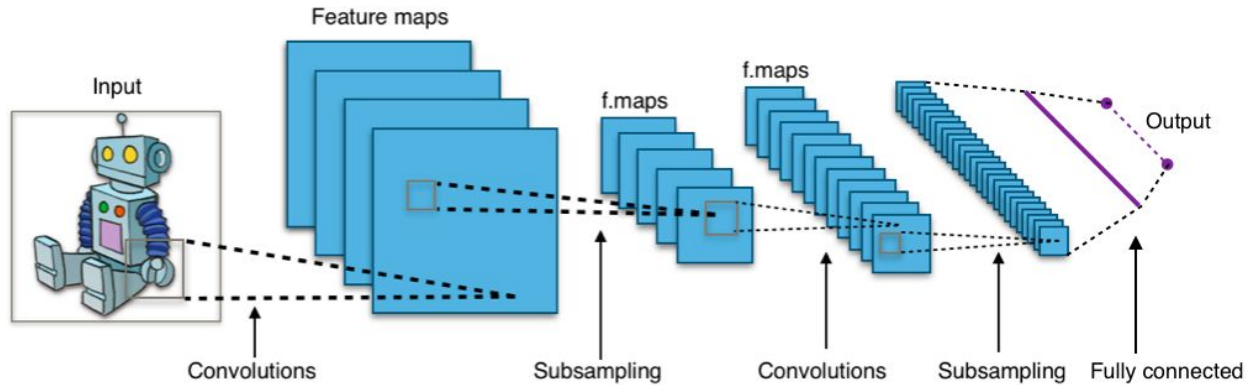
Shallow embedding approach generated the representation of nodes with **only the graph structural information**

We want to generate representations of nodes that **actually depend on the structure of the graph**, as well as **any feature information** we might have.

Therefore, we will introduce the **graph neural network (GNN) formalism**, which is a general framework for defining **deep neural networks on graph data**

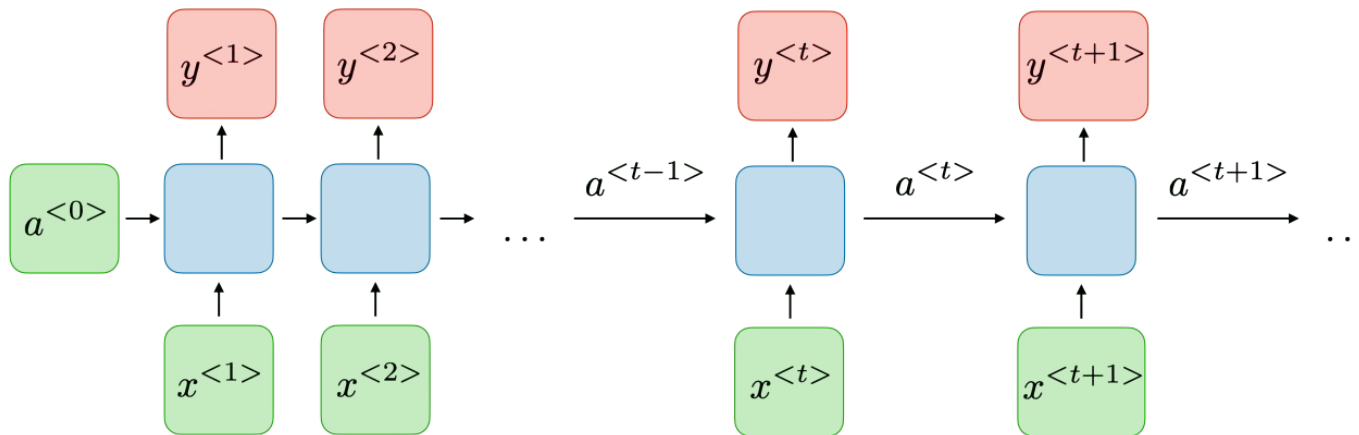
Challenges

- Grid-structured data -> Convolutional Neural Networks (CNNs)



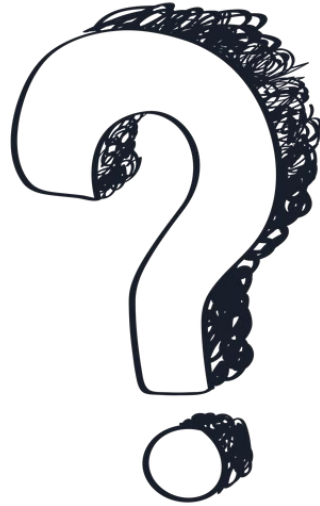
Challenges

- Sequence data -> Recurrent Neural Networks (RNNs)



Challenges

- Graphical data ->



Challenges

We need to define a new kind of deep learning architecture since the usual deep learning toolbox does not apply

The GNN Framework: Neural Message Passing

Naive Attempt

We can generate the embedding of an entire graph by simply flatten the adjacency matrix and feed the result to a multi-layer perceptron (MLP):

$$\mathbf{z}_G = \text{MLP}(\mathbf{A}[1] \oplus \mathbf{A}[2] \oplus \dots \oplus \mathbf{A}[|\mathcal{V}|])$$

where \oplus denotes the vector concatenation and $\mathbf{A}[i]$ is the i th row of the adjacency matrix

Naive Attempt

However, this approach is not permutation invariant (i.e. it depends on the arbitrary ordering of nodes that we used in the adjacency matrix), and we would like the neural networks defined on graph to be permutation invariant/equivariant

$$f(\mathbf{PAP}^\top) = f(\mathbf{A}) \quad (\text{Permutation Invariance})$$

$$f(\mathbf{PAP}^\top) = \mathbf{P}f(\mathbf{A}) \quad (\text{Permutation Equivariance})$$

- Permutation invariance means that the function does not depend on the arbitrary ordering of the rows/columns in the adjacency matrix
- Permutation equivalence means that the output of f is permuted in an consistent way when we permute the adjacency matrix

Naive Attempt

Let $f_{\theta}(\cdot) : \mathbb{X}^N \mapsto \mathbb{Y}^M$ a function parametrized by θ , and \mathcal{G} be the permutation group ($\mathcal{G} = \mathcal{S}_N$).

- If $M = 1$, f_{θ} is permutation-**invariant** if

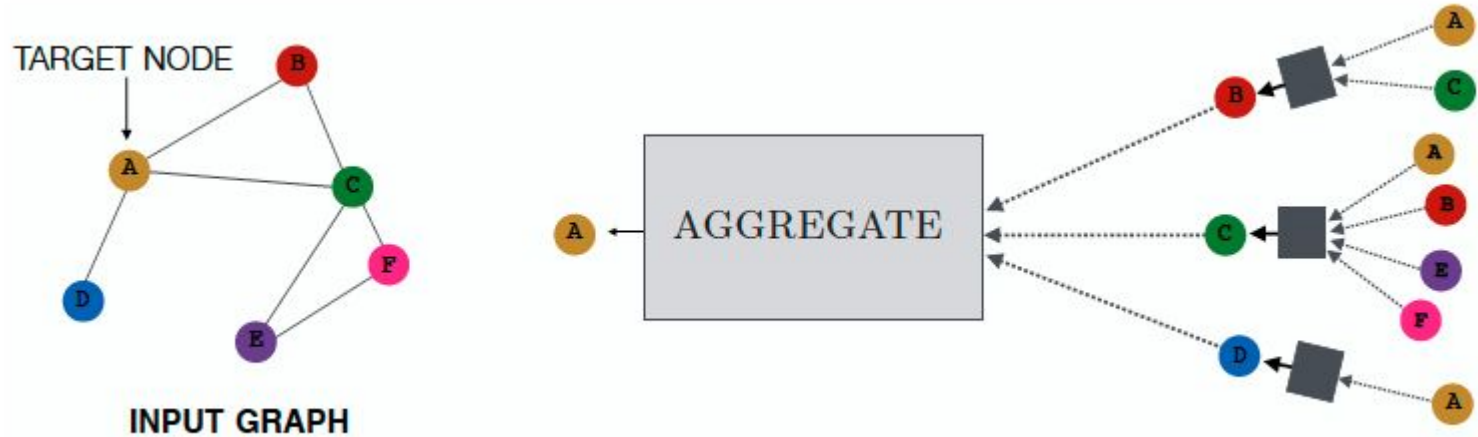
$$f_{\theta}(g \cdot \mathbf{x}) = f_{\theta}(\mathbf{x}), \quad \forall g \in \mathcal{G}, \mathbf{x} \in \mathbb{X}^N$$

- If $M = N$, f_{θ} is permutation-**equivariant** if

$$f_{\theta}(g \cdot \mathbf{x}) = g \cdot f_{\theta}(\mathbf{x}), \quad \forall g \in \mathcal{G}, \mathbf{x} \in \mathbb{X}^N$$

Neural Message Passing

Vector messages are exchanged between nodes and updated using neural networks



Overview

During each message-passing iteration in a GNN, a hidden embedding $h_u^{(k)}$ corresponding to each node $u \in V$ is updated according to information aggregated from u 's graph neighborhood $N(u)$

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

Overview

After running K iterations of the GNN message passing, we can use the output of the final layer to define the embeddings for each node

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}$$

Since the AGGREGATE function takes a set as input, GNNs defined in this way are permutation equivariant by design

Intuitions

At each iteration, every node aggregates information from its local neighborhood, and as these iterations progress each node embedding contains more and more information from further reaches of the graph

1. After the first iteration ($k = 1$), every node embedding contains information from its 1-hop neighborhood, i.e., every node embedding contains information about the features of its immediate graph neighbors, which can be reached by a path of length 1 in the graph;
2. After the second iteration ($k = 2$) every node embedding contains information from its 2-hop neighborhood
3. In general, after k iterations every node embedding contains information about its k -hop neighborhood.

Information Embedded

Structural Information

- After k iterations of GNN message passing, the embeddings for each node also encode structural information about its k -hop neighborhood, e.g. degree

Feature Information

- After k iterations of GNN message passing, the embeddings for each node also encode information about all the features in their k -hop neighborhood

The Basic GNN

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

activation function

trainable parameter matrices

bias

The Basic GNN

Equivalently, we can define the basic GNN through the UPDATE and AGGREGATE functions

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma(\mathbf{W}_{\text{self}}\mathbf{h}_u + \mathbf{W}_{\text{neigh}}\mathbf{m}_{\mathcal{N}(u)})$$

where

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



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

Self-Loops

It is common to add self-loops to the input graph and omit the explicit update step

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

The basic GNN can be written as

$$\mathbf{H}^{(t)} = \sigma \left((\mathbf{A} + \mathbf{I})\mathbf{H}^{(t-1)}\mathbf{W}^{(t)} \right)$$

which will be referred as the self-loop GNN

Self-Loops

- **Pros**
 - By adding self-loops, we no longer need to define an explicit update function
- **Cons**
 - Severely limits the expressivity of the GNN, since the information coming from the node's neighbours cannot be differentiated from the information from the node itself

Generalized Aggregation Methods

Objective

How can we generalize and improve the AGGREGATE operator we've seen before

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



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

1. Neighborhood Normalization

- Issues with summarization aggregator
 - Unstale
 - Highly sensitive to node degrees

Ex:

Suppose node u have 100x as many neighbors as node u' , then for any reasonable vector norm

$$\|\sum_{v \in \mathcal{N}(u)} \mathbf{h}_v\| \gg \|\sum_{v' \in \mathcal{N}(u')} \mathbf{h}_{v'}\|$$

1. Neighborhood Normalization

- Solution

Average aggregator

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

Symmetric Normalization

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

1. Neighborhood Normalization

- Application
 - Graph Convolutional Network (GCN)

$$\mathbf{h}_u^{(k)} = \sigma \left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)| |\mathcal{N}(v)|}} \right)$$

1. Neighborhood Normalization

- Side Notes

- Proper normalization can be essential to achieve stable and strong performance when using a GNN
- However, that normalization can also lead to a loss of information (ex: it can be hard to use the learned embeddings to distinguish between nodes of different degree)
- The use of normalization is thus an **application-specific** question
- Normalization is most helpful in tasks where node feature information is far more useful than structural information, or where there is a very wide range of node degrees that can lead to instabilities during optimization

2. Set Aggregator

- Neighborhood normalization can be a useful tool to improve GNN performance, but still not make any improvements to the AGGREGATE operator
- We would like to have a more sophisticated to aggregate neighbor embeddings than just summing over them
- The AGGREGATE operator is fundamentally a **set function**, i.e. it takes set of neighbor embeddings as input and map it to a single vector

$$\{\mathbf{h}_v, \forall v \in \mathcal{N}(u)\} \longrightarrow \mathbf{m}_{\mathcal{N}(u)}$$

2. Set Aggregator

- Set Pooling

- Universal Set Function Approximator ([Deep Sets](#)): any permutation-invariant function that maps a set of embeddings to a single embedding can be approximated to an arbitrary accuracy by a model in the form of:

$$\mathbf{m}_{\mathcal{N}(u)} = \text{MLP}_{\theta} \left(\sum_{v \in \mathcal{N}(u)} \text{MLP}_{\phi}(\mathbf{h}_v) \right)$$

↙
To prevent overfitting, it is common in practice to use a single layer MLP

↓
Can be replaced by any alternative reduction function, i.e. max/min

2. Set Aggregator

- Set Pooling
 - Often lead to small increases in performance
 - Has the potential of introduce an increased risk of overfitting (depending on the depth of the MLP)
 - In practice, it is common to use MLPs with single hidden layer

2. Set Aggregator

- Janossy Pooling
 - Use permutation-sensitive function then average the results instead of the permutation invariant function

$$\mathbf{m}_{\mathcal{N}(u)} = \text{MLP}_{\theta} \left(\frac{1}{|\Pi|} \sum_{\pi \in \Pi} \rho_{\phi} (\mathbf{h}_{v_1}, \mathbf{h}_{v_2}, \dots, \mathbf{h}_{v_{|\mathcal{N}(u)|}})_{\pi_i} \right) \quad (\star)$$

A set of permutation functions

A permutation-sensitive function (e.g. RNN). In practice, usually defined as an LSTM

2. Set Aggregator

- Janossy Pooling
 - If we use all set of possible permutations, then (★) is equivalent as a universal function approximator. However, this is not feasible in practice.
 - Two approaches employed by Janossy pooling
 - Sample a random subset of possible permutations during each application of the aggregator, and only sum over that random subset.
 - Employ a canonical ordering of the nodes in the neighborhood set; e.g., order the nodes in descending order according to their degree, with ties broken randomly

3. Neighborhood Attention

- Basic Idea
 - Assign an attention weight or importance to each neighbor, which is used to weigh this neighbor's influence during the aggregation steps

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

3. Neighborhood Attention

- Attention Weights: GAT

$$\alpha_{u,v} = \frac{\exp(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_v])}{\sum_{v' \in \mathcal{N}(u)} \exp(\mathbf{a}^\top [\mathbf{W}\mathbf{h}_u \oplus \mathbf{W}\mathbf{h}_{v'}])}$$

Trainable attention vector

Trainable matrix

Concatenation operation

3. Neighborhood Attention

- Attention Weights: some variants

$$\alpha_{u,v} = \frac{\exp(\mathbf{h}_u^\top \mathbf{W} \mathbf{h}_v)}{\sum_{v' \in \mathcal{N}(u)} \exp(\mathbf{h}_u^\top \mathbf{W} \mathbf{h}_{v'})}$$

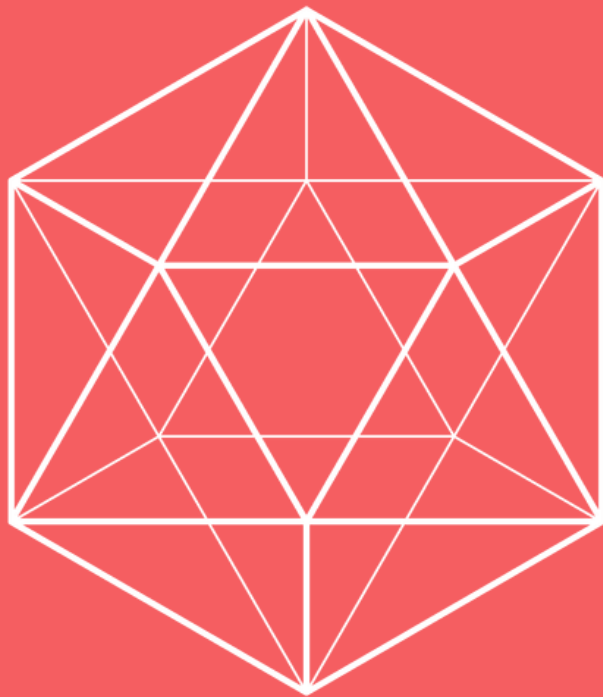
$$\alpha_{u,v} = \frac{\exp(\text{MLP}(\mathbf{h}_u, \mathbf{h}_v))}{\sum_{v' \in \mathcal{N}(u)} \exp(\text{MLP}(\mathbf{h}_u, \mathbf{h}_{v'}))}$$

3. Neighborhood Attention

- Side Notes
 - Can increasing the the representational capacity of a GNN model, especially in cases where you have prior knowledge to indicate that some neighbors might be more informative than others.

Outline

- Generalized Update Methods
- GNNs & Message Passing For Heterogeneous Graphs
- Learning Representations For Graph: Graph Pooling



Generalized Update Methods

Quick Remainder

- Linear combination

$$\text{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \sigma \left(\mathbf{W}_{\text{self}} \mathbf{h}_u + \mathbf{W}_{\text{neigh}} \mathbf{m}_{\mathcal{N}(u)} \right)$$

- Self-loop

$$\mathbf{H}^{(t)} = \sigma \left((\mathbf{A} + \mathbf{I}) \mathbf{H}^{(t-1)} \mathbf{W}^{(t)} \right)$$

Oversmoothing

- A common issue with GNN
- After several iterations of GNN message passing, the representations for all the nodes in the graph can become very similar to one another
- Intuitively, we can expect oversmoothing in cases where the information being aggregated from the node neighbors during message passing begins to dominate the updated node representations

Objective

UPDATE operator plays an equally important role as the AGGREGATE operator, we want explore other possible ways to define the UPDATE operator while try to alleviate the effect of oversmoothing.

1. Concatenation & Skip-Connection

- Intuition
 - In case of oversmoothing, the updated node representation depends strongly on the incoming messages from the neighbors, and “overwrites” the node representation from the previous layers
 - We can alleviate this by using vector concatenation, which directly preserve the information from the previous rounds of message passing during the update

1. Concatenation & Skip-Connection

- Simple Concatenation

$$\text{UPDATE}_{\text{concat}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = [\text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) \oplus \mathbf{h}_u]$$

- Linear Interpolation

$$\text{UPDATE}_{\text{interpolate}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) = \boldsymbol{\alpha}_1 \circ \text{UPDATE}_{\text{base}}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}(u)}) + \boldsymbol{\alpha}_2 \odot \mathbf{h}_u$$

learnable gating operator

$$\boldsymbol{\alpha}_1, \boldsymbol{\alpha}_2 \in [0, 1]^d \quad \& \quad \boldsymbol{\alpha}_2 = \mathbf{1} - \boldsymbol{\alpha}_1$$

1. Concatenation & Skip-Connection

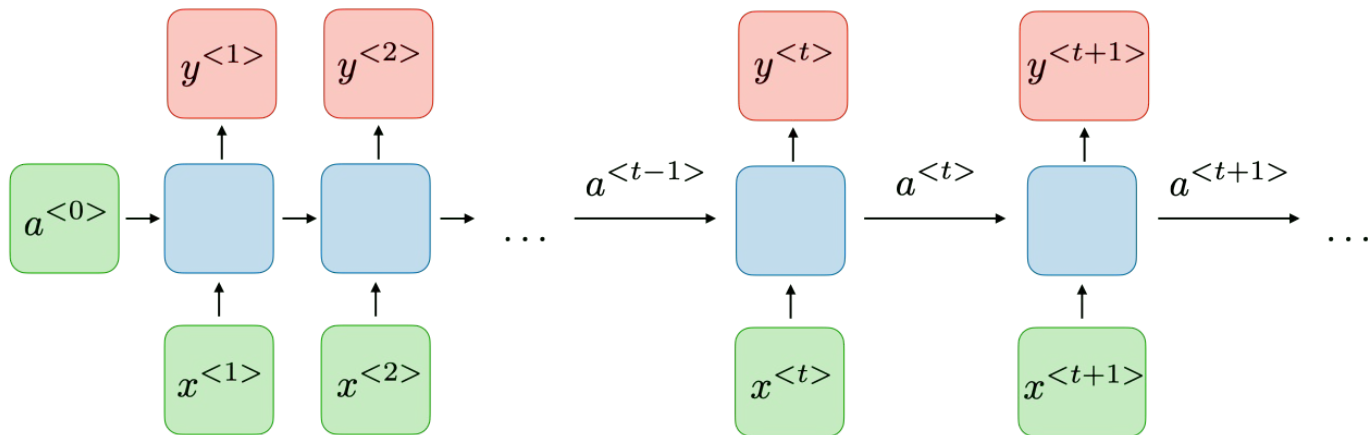
- Usage
 - In practice, most useful for node classification tasks with moderately deep GNN (2-5 layers)
 - Perform very well on homophily graphs (the prediction for each node is strongly related to the features of its local neighborhood)

2. Gated Updates

- Intuition
 - Inspired by RNN
 - We can view the GNN message passing algorithm as aggregation function receiving an observation from the neighbors, which is then used to update the representation of each node

2. Gated Updates

- RNN (Recurrent Neural Network)



$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$


$$y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

2. Gated Updates

- GRU (Gated Recurrent Unit)
 - Solved the vanilla RNN's problem of losing information in early time steps (short memory) for long sequences by adding "gates" (internal mechanisms that regulate the information flow between time steps, they can learn which data in a sequence is important to keep or throw away)

2. Gated Updates

- GRU

$$\mathbf{h}_u^{(k)} = \text{GRU}(\mathbf{h}_u^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)})$$


Gated recurrent unit; can be replaced with any updated function defined for RNN

2. Gated Updates

- Usage
 - In practice, gated updates are very effective at facilitating deep GNN architectures (>10 layers)
 - Perform very well on prediction task requires complex reasoning over the global structure of the graph, such as applications for program analysis or combinatorial optimization

3. Jumping Knowledge Transfer

- Intuition
 - We've assumed to only use node representation from the last layer of GNN

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V}$$

- Why not leverage the node representations at each layer?

3. Jumping Knowledge Transfer

$$\mathbf{z}_u = f_{\text{JK}}(\mathbf{h}_u^{(0)} \oplus \mathbf{h}_u^{(1)} \oplus \dots \oplus \mathbf{h}_u^{(K)})$$



An arbitrary differentiable function

In many applications the function can simply be defined as the identity function.


3. Jumping Knowledge Transfer

- Usage
 - Helpful for all kinds of tasks! A generally useful strategy to employ

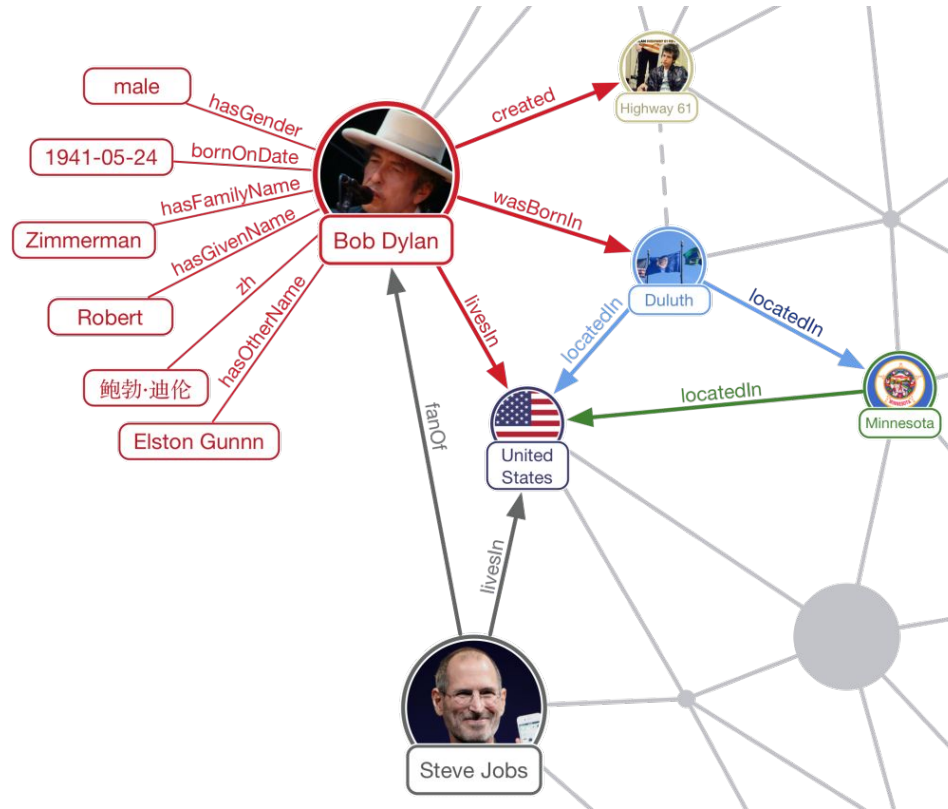
GNNs & Message Passing For Heterogeneous Graphs

Motivations

- So far our discussion of GNNs and neural message passing has implicitly assumed that we have simple graphs
- However, there are many applications where the graphs in question are multi-relational or otherwise heterogenous (e.g., knowledge graphs)

$$\mathcal{G} = (\mathcal{V}, \mathcal{E})$$

$$e = (u, \tau, v)$$

Motivations




1. Relational Graph Neural Networks

- Augment the aggregation function to accommodate multiple relation types by specifying a separate transformation matrix per relation type



too many parameters

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_{\tau}(u)} \frac{\mathbf{W}_{\tau} \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))}$$


A normalization function that can depend on both the neighborhood of the node u as well as the neighbor v being aggregated over

1. Relational Graph Neural Networks

- Solution - Parameter Sharing
- All the relation matrices are defined as linear combinations of b basis matrices

$$\mathbf{W}_{\tau} = \sum_{i=1}^b \alpha_{i,\tau} \mathbf{B}_i.$$

↓
Learnable parameters
for each relation

1. Relational Graph Neural Networks

$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_{\tau}(u)} \frac{\mathbf{W}_{\tau} \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))}$$



$$\mathbf{m}_{\mathcal{N}(u)} = \sum_{\tau \in \mathcal{R}} \sum_{v \in \mathcal{N}_{\tau}(u)} \frac{\boldsymbol{\alpha}_{\tau} \times_1 \mathcal{B} \times_2 \mathbf{h}_v}{f_n(\mathcal{N}(u), \mathcal{N}(v))}$$

tensor product along mode i

1. Relational Graph Neural Networks

The *n*-mode (matrix) product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \dots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{U}$ and is of size $I_1 \times \dots \times I_{n-1} \times J \times I_{n+1} \times \dots \times I_N$. Elementwise, we have

$$(\mathcal{X} \times_n \mathbf{U})_{i_1 \dots i_{n-1} j i_{n+1} \dots i_N} = \sum_{i_n=1}^{I_n} x_{i_1 i_2 \dots i_N} u_{j i_n}.$$

Each mode-*n* fiber [of \mathcal{X}] is multiplied by the matrix \mathbf{U} .

Learning Representations For Graph: Graph Pooling

Motivations

- So far we've seen how to learn node representations, we would also like to learn an embedding for graph
- This graph representation will be useful for tasks such as graph classification

1. Set Pooling Approaches

- Similar to the AGGREGATE operator in node representation learning, the task of graph pooling can be viewed as a problem of learning over sets
- We would like to design a pooling function that maps a set of node embeddings to a graph embedding that represents the full graph
- Two most commonly used approaches:
 - Pooling by summation
 - Pooling by LSTMs + attention

1. Set Pooling Approaches: Summation

- Simple but sufficient for small graphs

$$\mathbf{z}_G = \frac{\sum_{v \in \mathcal{V}} \mathbf{z}_v}{f_n(|\mathcal{V}|)}$$

can also use average operation

1. Set Pooling Approaches: LSTM + attention

Attention function, e.g.
dot product

$$\begin{aligned}\mathbf{q}_t &= \text{LSTM}(\mathbf{o}_{t-1}, \mathbf{q}_{t-1}), \\ e_{v,t} &= f_a(\mathbf{z}_v, \mathbf{q}_t), \forall v \in \mathcal{V}, \\ a_{v,t} &= \frac{\exp(e_{v,t})}{\sum_{u \in \mathcal{V}} \exp(e_{u,t})}, \forall v \in \mathcal{V}, \\ \mathbf{o}_t &= \sum_{v \in \mathcal{V}} a_{v,t} \mathbf{z}_v.\end{aligned}$$

Intermediate state of graph embedding at time t

1. Set Pooling Approaches: LSTM + attention

- Final Graph Embedding

$$\mathbf{z}_G = \mathbf{o}_1 \oplus \mathbf{o}_2 \oplus \dots \oplus \mathbf{o}_T$$

2. Graph Coarsening Approaches

- One limitation of the set pooling approaches is that they do not exploit the structure of the graph. There can also be benefits from exploiting the graph topology at the pooling stage
- One popular strategy to accomplish this is to perform graph **clustering** or **coarsening**

2. Graph Coarsening Approaches

- Clustering Function
 - Assume we have some clustering function that maps all the nodes in the graph to an assignment over *c clusters*
 - We can view the output of this function as an assignment matrix $S = f_c(G, Z)$, where $S[u, i] \in \mathbb{R}^+$ denotes the strength of the association between node u and cluster i

$$f_c \rightarrow \mathcal{G} \times \mathbb{R}^{|V| \times d} \rightarrow \mathbb{R}^{+|V| \times c}$$

size of node's representation vector number of clusters

2. Graph Coarsening Approaches

- Graph Coarsening
 - We will use the clustering function to coarsen the graph, i.e. produce a coarse graph of reduced size while preserving important graph properties

New Adjacency Matrix:

$$\mathbf{A}^{\text{new}} = \mathbf{S}^{\top} \mathbf{A} \mathbf{S} \in \mathbb{R}^{c \times c}$$

New Feature Matrix:

$$\mathbf{X}^{\text{new}} = \mathbf{S}^{\top} \mathbf{X} \in \mathbb{R}^{c \times d}$$

2. Graph Coarsening Approaches

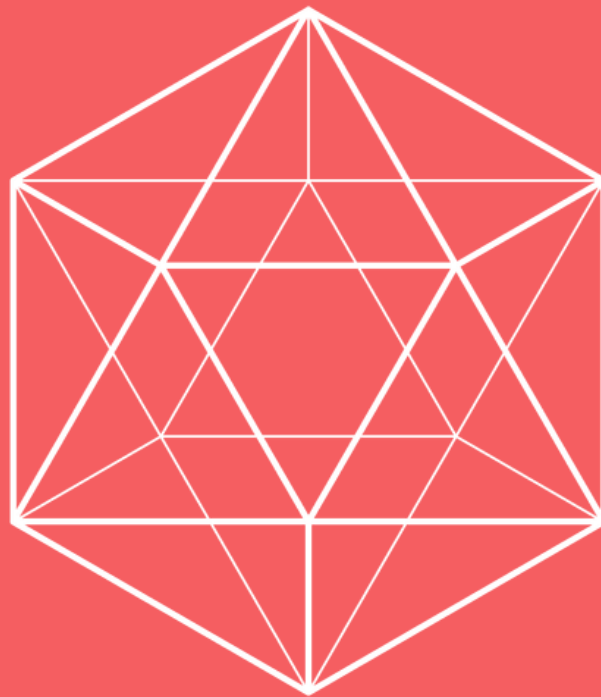
- Final Representation
 - We can then run a GNN on this coarsened graph and repeat the entire coarsening process for a number of iterations
 - The final representation of the graph is then computed by a set pooling over the embeddings of the nodes in a sufficiently coarsened graph

2. Graph Coarsening Approaches

- Can lead to strong performance
- Can be unstable and difficult to train

Outline

- GNN In Practice
 - Loss Function
 - Efficiency Concern & Node Sampling
 - Parameter Sharing & Regularization



GNN In Practice: Loss Functions

1. Node Classification

- Supervised Learning: negative log-likelihood

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

The predicted probability that node u belongs to class y_u

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

node embedding output
by the final layer of a
GNN

one-hot vector indicating
the the class of training
node u

trainable parameters

2. Graph Classification

- Same as node classification, just replace the node embedding with graph embedding z_{Gi}

3. Graph Regression

- For example, predicting molecular properties (e.g., solubility) from graph-based representations of molecules
- Square-error loss

$$\mathcal{L} = \sum_{\mathcal{G}_i \in \mathcal{T}} \|\text{MLP}(\mathbf{z}_{\mathcal{G}_i}) - y_{\mathcal{G}_i}\|_2^2$$

↙
densely connected dense layer to
map graph representation vector
to a univariate output

↓
Target value for training graph \mathcal{G}_i

GNN In Practice: Efficiency Concern & Node Sampling

Motivation

- We've discussed GNNs from the perspective of node-level message passing equations
- Directly implementing a GNN based on these equations can be computationally inefficient (e.g. if multiple nodes share neighbors, we will do redundant computation if we implement the message passing operations independently for all nodes in the graph)
- We would like to implement GNN in an efficient manner

1. Graph-Level Implementations

- No redundant computations—we compute the embedding for each *node* u exactly once when running the model
- It requires operating on the entire graph and all node features simultaneously, which may not be feasible due to memory limitations
- Limits to full-batch

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



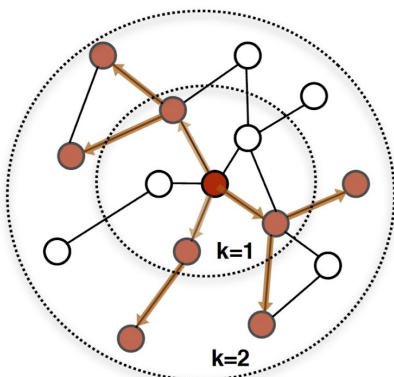
a matrix containing the layer-k embeddings of all the nodes in the graph

2. Subsampling and Mini-Batching

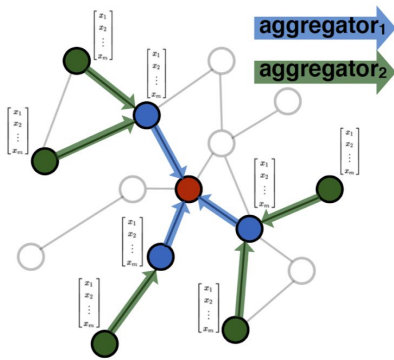
- In order to limit the memory footprint of a GNN, one can work with a subset of nodes during message passing (running the node-level GNN equations for a subset of the nodes in the graph in each batch)
- However, we suffer from information loss when run message passing on a subset of the nodes in a graph (Every time we remove a node, we also delete its edges). There is no guarantee that selecting a random subset of nodes will even lead to a connected graph, and selecting a random subset of nodes for each mini-batch can have a severely detrimental impact on model performance.

2. Subsampling and Mini-Batching

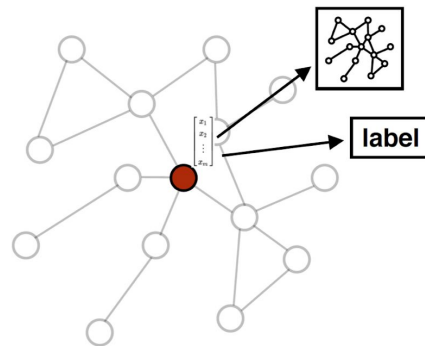
- Instead of subsample graph nodes, we can subsample node neighborhoods
 - First select a set of target nodes for a batch
 - Then recursively sample the neighbors of these nodes in order to ensure that the connectivity of the graph is maintained (normally use a fixed sample size to improve the efficiency of batched tensor operations)



1. Sample neighborhood



2. Aggregate feature information from neighbors



3. Predict graph context and label using aggregated information

GNN In Practice: Parameter Sharing and Regularization

Motivation

- Regularization is a key component of any machine learning model, it discourages the model from learning a more complex or flexible model, thus can avoid the risk of overfitting

1. Parameter Sharing Across Layers

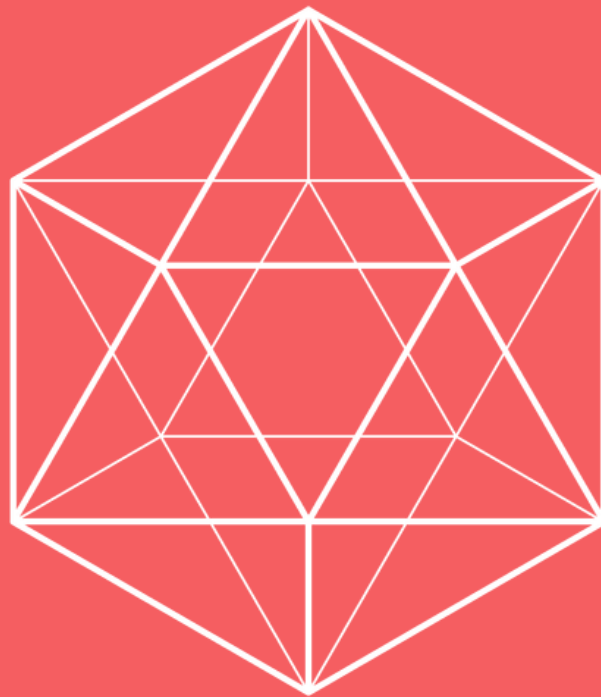
- Use the same parameters in all the AGGREGATE and UPDATE functions in the GNN
- Most effective in GNNs with more than six layers, and it is often used in conjunction with gated update functions

2. Edge Dropout

- Randomly remove (or mask) edges in the adjacency matrix during training
- This will make the GNN less prone to overfitting and more robust to noise in the adjacency matrix
- Most successful in the application of GNNs in knowledge graphs

Outline

- Theoretical Motivations
 - GNN & GSP (skip)
 - GNN & PGM
 - GNN & Graph Isomorphism



GNN & PGM

GNN & PGM

- In this probabilistic perspective, we view the embeddings z_u , $\forall u \in V$ for each node as latent variables that we are attempting to infer
- We assume that we observe the graph structure (i.e., the adjacency matrix, A) and the input node features, X , and our goal is to infer the underlying latent variables (i.e., the embeddings z_v) that can explain this observed data
- Taking a probabilistic view of graph data, we can assume that the graph structure we are given defines the dependencies between the different nodes (i.e. nodes that are connected in a graph are generally assumed to be related in some way.)

Markov Random Field

$$p(\{\mathbf{x}_v\}, \{\mathbf{z}_v\}) \propto \prod_{v \in V} \Phi(\mathbf{x}_v, \mathbf{z}_v) \prod_{(u,v) \in \mathcal{E}} \Psi(\mathbf{z}_u, \mathbf{z}_v)$$

$\{\mathbf{x}_u, \forall u \in V\}$ $\{\mathbf{z}_u, \forall u \in V\}$ potential functions, maps input to \mathbb{R}^+

Intuitively,

- $\Phi(\mathbf{x}_v, \mathbf{z}_v)$ indicates the likelihood of a node feature vector \mathbf{x}_v given its latent node embedding \mathbf{z}_v
- Ψ controls the dependency between connected nodes

We thus assume that node features are determined by their latent embeddings, and we assume that the latent embeddings for connected nodes are dependent on each other

Markov Random Field

- Our goal is to infer the distribution of latent embeddings for all the nodes $v \in V$, while also implicitly learning the potential functions Φ and Ψ
- In order to do so, a key step is computing the posterior $p(\{z_v\}|\{x_v\})$ (the likelihood of a particular set of latent embeddings given the observed features)
- In general, computing this posterior is computationally intractable—even if Φ and Ψ are known and well-defined—so we must resort to approximate methods

Mean-Field Inference

- We will approximate the posterior using some functions q_v based on the assumption

$$p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\}) \approx q(\{\mathbf{z}_v\}) = \prod_{v \in \mathcal{V}} q_v(\mathbf{z}_v)$$

- The key intuition in mean-field inference is that we assume that the posterior distribution over the latent variables factorizes into V independent distributions, one per node

Mean-Field Inference

- To obtain q_v functions that are optimal in the mean-field approximation, the standard approach is to minimize the Kullback–Leibler (KL) divergence between the approximate posterior and the true posterior

$$\text{KL}(q(\{\mathbf{z}_v\})|\{p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\}) = \int_{(\mathbb{R}^d)^{\mathcal{V}}} \prod_{v \in \mathcal{V}} q(\{\mathbf{z}_v\}) \log \left(\frac{\prod_{v \in \mathcal{V}} q(\{\mathbf{z}_v\})}{p(\{\mathbf{z}_v\}|\{\mathbf{x}_v\})} \right) \prod_{v \in \mathcal{V}} d\mathbf{z}_v.$$

- Directly minimizing the above equation is impossible, since evaluating the KL divergence requires knowledge of the true posterior. Techniques from variational inference can be used to show that $q_v(\mathbf{z}_v)$ that minimize the KL must satisfy the following fixed point equations:

$$\log(q(\mathbf{z}_v)) = c_v + \log(\Phi(\mathbf{x}_v, \mathbf{z}_v)) + \sum_{u \in \mathcal{N}(v)} \int_{\mathbb{R}^d} q_u(\mathbf{z}_u) \log(\Psi(\mathbf{z}_u, \mathbf{z}_v)) d\mathbf{z}_u$$

Mean-Field Inference

- In practice, we can approximate this fixed point solution by initializing some initial guesses $q^{(t)}$ to valid probability distributions and iteratively computing

$$\log \left(q_v^{(t)}(\mathbf{z}_v) \right) = c_v + \log(\Phi(\mathbf{x}_v, \mathbf{z}_v)) + \sum_{u \in \mathcal{N}(v)} \int_{\mathbb{R}^d} q_u^{(t-1)}(\mathbf{z}_u) \log(\Psi(\mathbf{z}_u, \mathbf{z}_v)) d\mathbf{z}_u$$

- The optimal approximation under the mean-field assumption is given by the approximate posterior $q_v(\mathbf{z}_v)$, where each latent node embedding is a function of
 - the node's feature \mathbf{x}_v and
 - the marginal distributions $q_u(\mathbf{z}_u)$, $\forall u \in \mathcal{N}(v)$ of the node's neighbors' embeddings.

Relation with GNN

- At each step, we are updating the values at each node based on the set of values in the node's neighborhood. The key distinction is that the mean-field message passing equations operate over distributions rather than embeddings, which are used in the standard GNN message passing

GNN & Graph Isomorphism

Background: Graph Isomorphism

- One of the most fundamental and well-studied tasks in graph theory
- *Def'n: Given a pair of graphs G_1 and G_2 , the goal of graph isomorphism testing is to declare whether or not these two graphs are isomorphic*
- Isomorphic graphs represent the exact same graph structure, but they might differ only in the ordering of the nodes in their corresponding adjacency matrices. We say that two graphs are isomorphic if and only if there exists a permutation matrix P such that

$$\mathbf{P}\mathbf{A}_1\mathbf{P}^\top = \mathbf{A}_2 \text{ and } \mathbf{P}\mathbf{X}_1 = \mathbf{X}_2.$$

Background: Graph Isomorphism

- Testing for graph isomorphism is a fundamentally hard problem
- A naive approach to test for isomorphism would involve the following optimization problem, and the computational complexity of this naive approach is immense at $O(|V|!)$

$$\min_{\mathbf{P} \in \mathcal{P}} \|\mathbf{P}\mathbf{A}_1\mathbf{P}^\top - \mathbf{A}_2\| + \|\mathbf{P}\mathbf{X}_1 - \mathbf{X}_2\| \stackrel{?}{=} 0$$

- Luckily, there are many practical algorithms for graph isomorphism testing that work on broad classes of graphs (for example, the WL algorithm that we introduced briefly in part 1 of the lecture)

Graph Isomorphism & Representational Capacity

- The theory of graph isomorphism testing is particularly useful for graph representation learning, it gives us a way to quantify the representational power of different learning approaches
- In previous lecture, we've talked about how a GNN can generate representations $z_G \in \mathbb{R}^d$ for graphs. Now, we can quantify the power of a learning algorithm by asking how useful these representations would be for testing graph isomorphism

Graph Isomorphism & Representational Capacity

- In particular, given learned representations $\mathbf{z}_{\mathcal{G}_1}$ and $\mathbf{z}_{\mathcal{G}_2}$ for two graphs, a “perfect” learning algorithm would have that

$$\mathbf{z}_{\mathcal{G}_1} = \mathbf{z}_{\mathcal{G}_2} \text{ if and only if } \mathcal{G}_1 \text{ is isomorphic to } \mathcal{G}_2$$

- The most natural way to connect GNNs to graph isomorphism testing is based on connections to the family of Weisfeiler-Lehman (WL) algorithms

The Weisfeiler-Lehman Algorithm

- 1-WL Algorithm

1. Given two graphs \mathcal{G}_1 and \mathcal{G}_2 we assign an initial label $l_{\mathcal{G}_i}^{(0)}(v)$ to each node in each graph. In most graphs, this label is simply the node degree, i.e., $l^{(0)}(v) = d_v \forall v \in V$, but if we have discrete features (i.e., one hot features \mathbf{x}_v) associated with the nodes, then we can use these features to define the initial labels.
2. Next, we iteratively assign a new label to each node in each graph by hashing the multi-set of the current labels within the node's neighborhood, as well as the node's current label:

$$l_{\mathcal{G}_i}^{(i)}(v) = \text{HASH}(l_{\mathcal{G}_i}^{(i-1)}(v), \{\{l_{\mathcal{G}_i}^{(i-1)}(u) \mid u \in \mathcal{N}(v)\}\}), \quad (7.60)$$

where the double-braces are used to denote a multi-set and the HASH function maps each unique multi-set to a unique new label.

The Weisfeiler-Lehman Algorithm

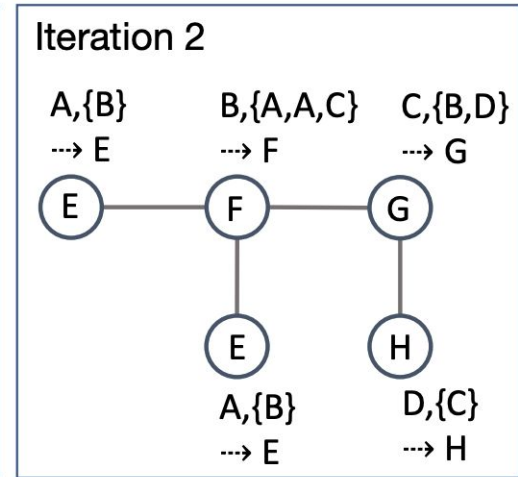
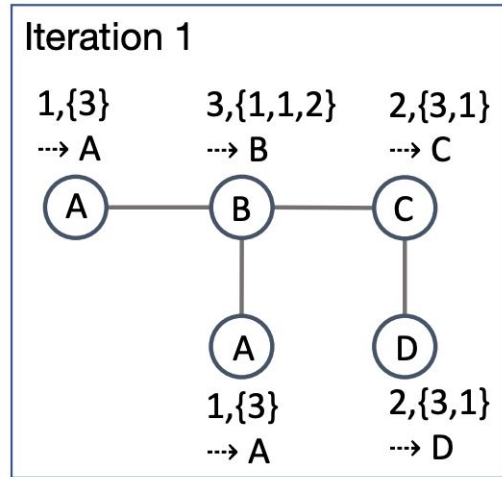
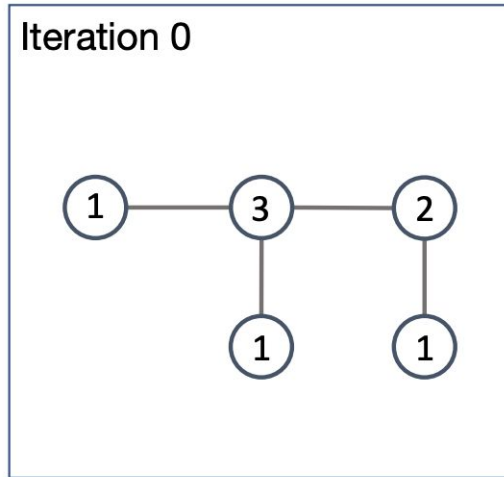
- 1-WL Algorithm (cont'd)

3. We repeat Step 2 until the labels for all nodes in both graphs converge, i.e., until we reach an iteration K where $l_{\mathcal{G}_j}^{(K)}(v) = l_{\mathcal{G}_i}^{(K-1)}(v), \forall v \in V_j, j = 1, 2$.
4. Finally, we construct multi-sets

$$L_{\mathcal{G}_j} = \{\{l_{\mathcal{G}_j}^{(i)}(v), \forall v \in \mathcal{V}_j, i = 0, \dots, K - 1\}\}$$

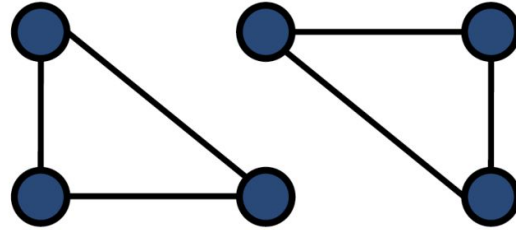
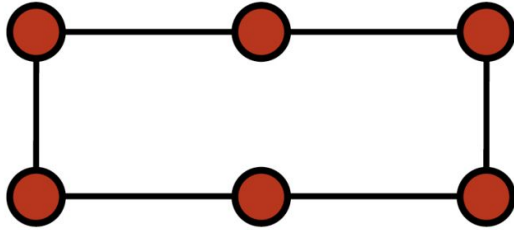
summarizing all the node labels in each graph, and we declare \mathcal{G}_1 and \mathcal{G}_2 to be isomorphic if and only if the multi-sets for both graphs are identical, i.e., if and only if $L_{\mathcal{G}_1} = L_{\mathcal{G}_2}$.

The Weisfeiler-Lehman Algorithm



The Weisfeiler-Lehman Algorithm

- The WL algorithm is known to converge in at most $|V|$ iterations and is known to successfully test isomorphism for a broad class of graphs
- Unfortunately, there are also some simple cases where the test fails



GNNs & the WL Algorithm

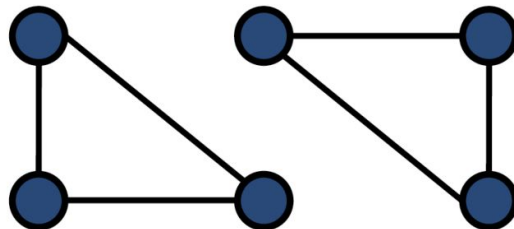
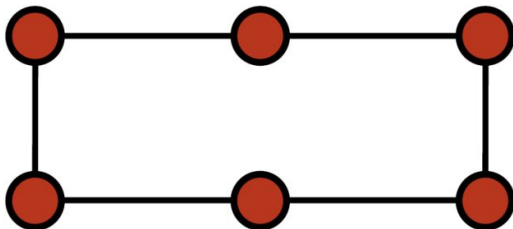
- In both approaches, we iteratively aggregate information from local node neighborhoods and use this aggregated information to update the representation of each node
- The key distinction between the two approaches is that the WL algorithm aggregates and updates discrete labels (using a hash function) while GNN models aggregate and update node embeddings using neural networks
- It has been proven that *GNNs are no more powerful than the WL algorithm when we have discrete information as node features*

Designing GNNs More Powerful Than WL

- Despite the negative result, investigating how we can make GNNs that are provably more powerful than the WL algorithm is an active area of research

I. Relational Pooling

- Considering the failure cases of the WL algorithm



- From the perspective of message passing, this limitation stems from the fact that AGGREGATE and UPDATE operations are unable to detect when two nodes share a neighbor

I. Relational Pooling

- We can consider augmenting message passing GNNs with unique node ID features
- If we use $\text{MP-GNN}(\mathbf{A}, \mathbf{X})$ to denote an arbitrary message passing GNN on input adjacency matrix \mathbf{A} and node features \mathbf{X} , then adding node IDs is equivalent to modifying the MP-GNN to the following

$$\text{MP-GNN}(\mathbf{A}, \mathbf{X} \oplus \mathbf{I})$$



column-wise matrix concatenation

- In other words, we simply add a unique, one-hot indicator feature for each node, it would allow a MP-GNN to identify when two nodes share a neighbor, which would make the two graphs distinguishable

I. Relational Pooling

- Unfortunately, however, this idea of adding node IDs does not solve the problem
- By adding unique node IDs we have actually introduced a new and equally problematic issue: the MP-GNN is no longer permutation equivariant

$$\mathbf{P}(\text{MP-GNN}(\mathbf{A}, \mathbf{X} \oplus \mathbf{I})) \neq \text{MP-GNN}(\mathbf{PAP}^\top, (\mathbf{PX}) \oplus \mathbf{I})$$

I. Relational Pooling

- To alleviate this issue, Murphy et al. [2019] propose the Relational Pooling (RP) approach, which involves marginalizing over all possible node permutations. Given any MP-GNN the RP extension of this GNN is given by

$$\text{RP-GNN}(\mathbf{A}, \mathbf{X}) = \sum_{\mathbf{P} \in \mathcal{P}} \text{MP-GNN}(\mathbf{PAP}^\top, (\mathbf{PX}) \oplus \mathbf{I})$$

- Summing over all possible permutation matrices $\mathbf{P} \in \mathcal{P}$ recovers the permutation invariance, and we retain the extra representational power of adding unique node ids
- Murphy et al. [2019] prove that the RP extension of a MP-GNN can distinguish graphs that are indistinguishable by the WL algorithm

I. Relational Pooling

- Limitations:
 - RP approach is in its computational complexity, i.e. $O(|V|!)$, which is infeasible in practice (Murphy et al. [2019] show that the RP approach can achieve strong results using various approximations to decrease the computation cost such as sampling a subset of permutations, but the the full algorithm is computationally intractable.)
 - We know that RP-GNNs are more powerful than the WL test, but we have no way to characterize how much more powerful they are

II. The k -WL test and k -GNNs

- The WL algorithm we've seen so far is the simplest in the family of k -WL algorithm, i.e. 1-WL algorithm, and it can be generalized to the k -WL algorithm for $k > 1$
- The key idea behind the k -WL algorithms is that we label subgraphs of size k rather than individual nodes

II. The k-WL test and k-GNNs

- The k-WL algorithm generates representation of a graph G by:

1. Let $s = (u_1, u_2, \dots, u_k) \in \mathcal{V}^k$ be a tuple defining a subgraph of size k , where $u_1 \neq u_2 \neq \dots \neq u_k$. Define the initial label $l_{\mathcal{G}}^{(0)}(s)$ for each subgraph by the isomorphism class of this subgraph (i.e., two subgraphs get the same label if and only if they are isomorphic).
2. Next, we iteratively assign a new label to each subgraph by hashing the multi-set of the current labels within this subgraph's neighborhood:

$$l_{\mathcal{G}}^{(i)}(s) = \text{HASH}(\{\{l_{\mathcal{G}}^{(i-1)}(s'), \forall s' \in \mathcal{N}_j(s), j = 1, \dots, k\}\}, l_{\mathcal{G}}^{(i-1)}(s)),$$

where the j th subgraph neighborhood is defined as

$$\mathcal{N}_j(s) = \{\{(u_1, \dots, u_{j-1}, v, u_{j+1}, \dots, u_k), \forall v \in \mathcal{V}\}\}. \quad (7.68)$$

II. The k-WL test and k-GNNs

3. We repeat Step 2 until the labels for all subgraphs converge, i.e., until we reach an iteration K where $l_{\mathcal{G}}^{(K)}(s) = l_{\mathcal{G}}^{(K-1)}(s)$ for every k -tuple of nodes $s \in \mathcal{V}^k$.
4. Finally, we construct a multi-set

$$L_{\mathcal{G}} = \{\{l_{\mathcal{G}}^{(i)}(v), \forall s \in \mathcal{V}^k, i = 0, \dots, K - 1\}\}$$

summarizing all the subgraph labels in the graph.

The summary $L_{\mathcal{G}}$ multi-set generated by the k-WL algorithm can be used to test graph isomorphism by comparing the multi-sets for two graphs

II. The k -WL test and k -GNNs

- For any $k \geq 2$ we can show that the $(k + 1)$ -WL test is strictly more powerful than the k -WL test
- However, note that running the k -WL requires solving graph isomorphism for graphs of size k , since **Step 1** in the k -WL algorithm requires labeling graphs according to their isomorphism type. Thus, running the k -WL for $k > 3$ is generally computationally intractable

II. The k -WL test and k -GNNs

- K-GNNs
 - People designed *k -GNNs* that learn embeddings associated with subgraphs—rather than nodes—and the message passing occurs according to subgraph neighborhoods
 - However, there are also serious computational concerns for both the k -WL test and k -GNNs, as the time complexity of the message passing explodes combinatorially as k increases

III. Invariant and equivariant k-order GNNs

- A crucial aspect of message-passing GNNs is that they are equivariant to node permutations, meaning that

$$\mathbf{P}(\text{MP-GNN}(\mathbf{A}, \mathbf{X})) = \text{MP-GNN}(\mathbf{PAP}^\top, \mathbf{PX})$$



permutation matrix

III. Invariant and equivariant k-order GNNs

- We can also define a similar notion of permutation invariance for MP-GNNs at the graph level

$$\text{POOL}(\text{MP-GNN}(\mathbf{PAP}^\top, \mathbf{PX})) = \text{POOL}(\text{MP-GNN}(\mathbf{A}, \mathbf{X}))$$

↓
permutation matrix

↓
pooling function: maps the matrix of learned node embeddings $\mathbf{Z} \in \mathbb{R}^{|V| \times d}$ to an embedding $\mathbf{z}_G \in \mathbb{R}^d$ of the entire graph.

III. Invariant and equivariant k-order GNNs

- Suppose we have an order- $(k + 1)$ tensor $\mathbf{X} \in \mathbb{R}^{|V|^{k \times d}}$, where we assume that the first k channels/modes of this tensor are indexed by the nodes of the graph
- Let $\mathbf{P} \star \mathbf{X}$ denotes the operation where we permute the first k channels of this tensor according the node permutation matrix \mathbf{P}
- We can then define an *linear equivariant layer* as a *linear operator* \mathcal{L}

$$\mathcal{L} \times (\mathbf{P} \star \mathcal{X}) = \mathbf{P} \star (\mathcal{L} \times \mathcal{X}), \forall \mathbf{P} \in \mathcal{P}$$

$$\mathcal{L} : \mathbb{R}^{|V|^{k_1} \times d_1} \rightarrow \mathbb{R}^{|V|^{k_2} \times d_2}$$

$$\mathcal{L} \in \mathbb{R}^{|V|^{2k} \times d_1 \times d_2}$$

generalized tensor product

III. Invariant and equivariant k-order GNNs

- Similarly, we can define *invariant linear operators* \mathcal{L} as

$$\mathcal{L} \times (\mathbf{P} \star \mathcal{X}) = \mathcal{L} \times \mathcal{X}, \forall \mathbf{P} \in \mathcal{P}.$$



$$\mathcal{L} : \mathbb{R}^{|\mathcal{V}|^k \times d_1} \rightarrow \mathbb{R}^{d_2}$$

$$\mathcal{L} \in \mathbb{R}^{|\mathcal{V}|^k \times d_1 \times d_2}$$

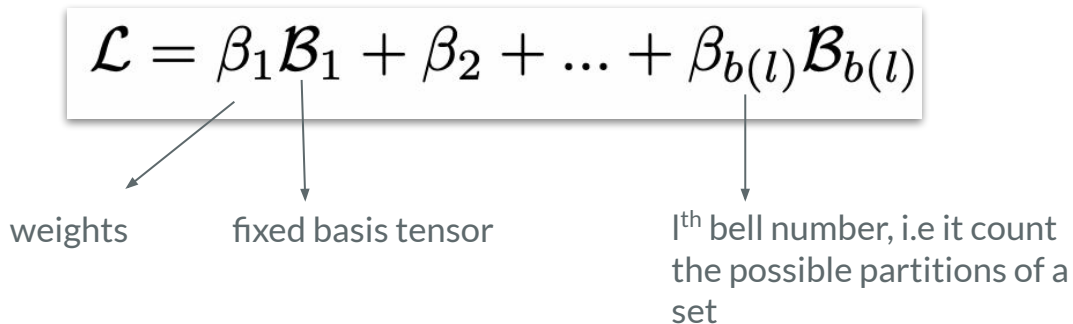
III. Invariant and equivariant k-order GNNs

- Interestingly, for a given input $\mathbf{X} \in \mathbb{R}^{|V| \times d}$, both equivariant and invariant linear operators on this input will correspond to tensors that satisfy the fixed point in equation, but the number of channels in the tensor will differ depending on whether it is an equivariant or invariant operator

$$\mathbf{P} \star \mathcal{L} = \mathcal{L}, \forall \mathbf{P} \in \mathcal{P}$$

III. Invariant and equivariant k-order GNNs

- Maron et al. [2019] show that tensors satisfying the the fixed point in the previous slides can be constructed as a linear combination of a set of fixed basis elements. In particular, any **order-l tensor** \mathcal{L} that satisfies the equation in the previous slide can be written as

$$\mathcal{L} = \beta_1 \mathcal{B}_1 + \beta_2 + \dots + \beta_{b(l)} \mathcal{B}_{b(l)}$$


weights

fixed basis tensor

l^{th} bell number, i.e it count the possible partitions of a set

- A key fact and challenge is that the number of basis tensors needed grows with l^{th} Bell number, which is an exponentially increasing series

III. Invariant and equivariant k-order GNNs

- Using these linear equivariant and invariant layers, Maron et al. [2019] define their invariant k-order GNN model based on the following composition of functions

$$\text{MLP} \circ \mathcal{L}_0 \circ \sigma \circ \mathcal{L}_1 \circ \sigma \mathcal{L}_2 \cdots \sigma \circ \mathcal{L}_m \times \mathcal{X}$$

Activation function

- The input to the k-order invariant GNN is the tensor $X \in \mathbb{R}^{|V| \times 2 \times d}$, where the first two channels correspond to the adjacency matrix and the remaining channels encode the initial node features/labels

III. Invariant and equivariant k-order GNNs

- This approach is called k-order because the equivariant linear layers involve tensors that have up to k different channels.
- Maron et al. [2019] prove that k-order models are equally powerful as the k-WL algorithm.
- However, constructing k-order invariant models for $k > 3$ is generally computationally intractable

Thank You :)

