Table of Contents

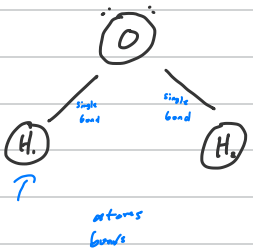
Graph Representation - Matrices

- Graphs typically represented with...
  - Something to define the structure: adjacency matrix/**edge index** (2×M - each column $\begin{bmatrix} i \\ j \end{bmatrix}$ is connection from node i to node j)
    - For weighted graph the weight is often in A if it is primary attribute
    - Edge index representation used instead of adjacency matrix because space wise it is much more efficient
      - First row is source node index of edges; second is target node index of edges; each column is an edge

↑ each node is automatically assigned unique index when you create a graph

- **Node feature matrix** X - $(N \times F)$ (N = nodes, F = features per node)
  - $\begin{matrix} \text{node1} \\ \vdots \\ \text{node N} \end{matrix} \begin{bmatrix} & \\ & \end{bmatrix}$ (F1 ... Fn)
  - For categorical features, either label encoding or one-hot encoding is used (usually one-hot); numerical values directly put in
    - Label encoding falsely implies ordinal relationship (e.g. red = 2, blue = 3); should be nominal (no natural ordering)
    - One-hot encoding will expand categorical features into multiple columns
- **Edge feature matrix** E - $(M \times F_e)$; typically used to encode features instead of weighted graph to account for multiple features (M = edges, F_e = features per edge)
  - $\begin{matrix} \text{edge1} \\ \vdots \\ \text{edge M} \end{matrix} \begin{bmatrix} & \\ & \end{bmatrix}$ (F1 ... Fn)
- **Label mapping** - {label: index, ...}
- **Graph-level features** (e.g. num) - stored in dictionary

Representing H_2O - Matrix Representation



Index left to right

↑ atoms bonds

Node Features

- Non-bonding electrons: node1 and node 3 = 0, node 2 = 4
- Label encoding only necessary if the labels provide useful information; sometimes also store labels/indices in separate dictionary to visualize more easily
  - One-hot encode node/atom type: hydrogen = [1 0], oxygen = [0 1]

$$X = \begin{matrix} \text{node1} \\ \text{node2} \\ \text{node3} \end{matrix} \begin{bmatrix} & \overbrace{\quad\quad} \\ & \end{bmatrix} = \begin{matrix} H_1 \\ O \\ H_2 \end{matrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 4 \\ 1 & 0 & 0 \end{bmatrix}$$

(atom type, num bonding)

new atom type ⇒ remake feature matrix and copy old features

Structure/Edges

$$A = \begin{bmatrix} \text{node1} \\ \text{node2} \\ \text{node3} \end{bmatrix} \begin{matrix} \text{node1} & \text{node2} & \text{node3} \\ \downarrow & \downarrow & \downarrow \end{matrix} \begin{bmatrix} A\text{-}A & A\text{-}B & A\text{-}C \\ B\text{-}A & B\text{-}B & B\text{-}C \\ C\text{-}A & C\text{-}B & C\text{-}C \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

$$\text{Edge index} = \begin{bmatrix} 0 & 1 & 1 & 2 \\ 1 & 0 & 2 & 1 \end{bmatrix}$$

Edge Features

$$E = \begin{matrix} \text{node1-node2} \\ \text{node2-node3} \end{matrix} \begin{bmatrix} \text{bond order} \\ \downarrow \\ \quad \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Mapping dictionary

node_indices = {'H1': 0, 'O': 1, 'H_2': 2} node_indices['H1'] = 2

Global Features

graph = {'name': 'water'}

Graph Representation - Dictionaries

- NetworkX represents these through dictionaries instead; for output it is list of tuples [(node 1, node 2 (if any), attribute_dict (if any)), ...] for easier interpretation
  - **_node** - dictionary that stores nodes and their attributes

    {node 1 : attribute_dict1, ... }

    e.g.

    ```
    {
      'A' : {'color' : 'red'}
      'B' : {'color' : 'blue'}
    }
    ```

    _node['A'] = {'color' : 'red'}

    _node['A']['color'] = 'red'

    (key usually union ID like index, not label)

  - **_adj** - adj was list containing node neighbors and edge attributes

    {node1 : {node1_neighbor : 1-neighbor edge_attribute_dict}

    e.g.

    ```
    {
      'A' : {
          'B' : {'weight' : 3, 'color' : 'red'}   (A-B)
          'neighbor_n' : {}
          }
      'B' : {
          }
    }
    ```

    _adj['A']['B']['weight'] = 3

  - **_graph** - stores global graph attributes

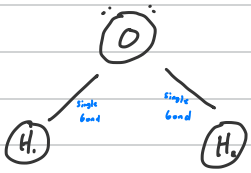
    {attribute_name 1 : attribute_value1 ,

    e.g.

    ```
    {
      'name' : 'water'
      'type' : 'undirected'
    }
    ```

    _graph['name'] = 'water'

Representing H_2O - Dictionary Representation



Index left to right

Node Features

- One-hot encode atom type: hydrogen = [1 0], oxygen = [0 1]

```
_node = {
        'H1': {
                atom_type: [1 0], non_bonding: 0
              }
        'O': { atom_type: [0 1], non_bonding: 4 }
        'H2': { atom_type: [1 0], non_bonding: 0 }
      }
```

Structure/edges

```
_adj = {
        'H1': {
                'O': {
                        'bond_order': 1
                      }
              }
        'O': {
                'H1': {'bond_order': 1}
                'H2': {'bond_order': 1}
             'H2': {'O': {'bond_order': 1}}
      }
```

Global Features

```
_graph = {'name': 'water'}
```

Message Passing

- $G = (V, E)$
  - G: graph; V: set of vertices; E: set of edges

- Each $v \in V$ has features represented in vector
- Each edge (u, v) also has features

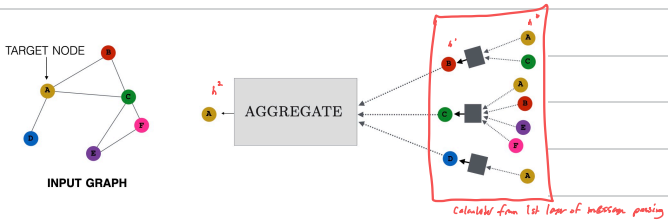
- Input to GNN can't just be flattened A because it is not permutation invariant
  - ✗ $\vec{z}_i = MLP(A[1] \oplus \dots \oplus A[|V|])$
    - $\oplus$: vector concatenation
  - $A \in \mathbb{R}^{|V| \times |V|} \Rightarrow A[:] \in \mathbb{R}^{|V|}$
    - $:$: row of A

- Since graph understanding depends on connection between all nodes, GNN must have some way to transfer information between them like a CNN
  - Adjacency matrix cannot be used for this as related nodes might not be next to each other
  - **Message passing** is solution to this

- $\vec{h}_u^k$ - kth version of **hidden embedding** for each node $u \in V$ calculated by aggregation function
  - hidden layer/not seen as output
  - Aggregation takes all of u's neighbors $v \in N(u)$ and final update uses this message and $\vec{u}$ to form new node embedding

- Message passing can be multiple layers (looks at the neighbors of the neighbors essentially); 2-layer message passing means it is done twice
  - 2: superscript; 2: 2 hops deep in graph
  - **Stacking** - having multiple sequential instances of a particular layer (e.g. message pass, fully connected/dense layers, convolution layers)

  - The computation graph of GNN for aggregation forms a tree structure



$$\vec{h}_u^{(k+1)} = Update^{(k)}\left(\vec{h}_u^{(k)}, Aggregate^{(k)}\left(\{\vec{h}_v^{(k)}, \forall v \in N(u)\}\right)\right)$$
$$= Update^{k}\left(\vec{h}_u^{k}, \vec{m}_{N(u)}^{k}\right)$$

- Update: includes non-linearity
- Aggregate: the message aggregated from neighbors
- $\vec{h}_u^0$: are the input features $\vec{x}_u$ $\forall u \in V$; aggregate takes set as input ⇒ no natural ordering of nodes' neighbors ⇒ aggregation functions must be **permutation invariant**
  - $\vec{m}_{N(u)}^{k}$: our message
- Update and aggregate are differentiable functions

- Final embedding output for that forward pass is node feature vector after **K iterations of message passing**
  - $\vec{z}_u = \vec{h}_u^{(K)}, \forall u \in V$ more hops requires more convolution layers

- **K** iterations of message passing means each node contains info within/from a K-hop neighborhood
  - K-layer message pass

More specifically...

$$\vec{h}_u^{(k)} = \sigma\left(W_{self}^{(k)} \vec{h}_u^{(k-1)} + W_{neigh}^{(k)} \sum_{v \in N(u)} \vec{h}_v^{(k-1)} + \vec{b}^{(k)} \right) \quad k \geq 1$$

W and $\vec{b}$ randomly initialized

$\vec{h}_u^{(k)}$ vs $\vec{h}_u^{(k-1)}$ emphasize different things and to make certain things more clear

- W matrices and $\vec{b}$ contain trainable parameters
- For k layer message pass (all nodes update), you get info from k hops away, and W_{self}, W_{neigh}, and b are used to update the node features
  - You can either have separate Ws and $\vec{b}$ for each layer of message passing (i.e. first layer has its W_{self_1}, W_{neighs_1}, $\vec{b}_{(1)}$) or share it across all layers
  - Separate parameters can mitigate the oversmoothing (or aggregating in a way where all nodes look the same) problem
- For homogenous (similar underlying structure) graphs and smaller datasets, may be better to have same transformation at each step of message passing (share parameters)

- Similar to MLP (linear operation followed by nonlinearity)

- Many types of message passing...

Sum Aggregation

$$\vec{m}_{N(u)} = \sum_{v \in N(u)} \vec{h}(v)$$

- Highly sensitive to node degrees and can be unstable/difficult to optimize

Mean/Average Aggregation

$$\vec{m}_{N(u)} = \frac{1}{|N(u)|} \sum_{v \in N(u)} \vec{h}_v$$

prevents issue of node degree sensitivity

- Solves *

Symmetric Normalization

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

- Motivated from spectral graph theory (normalization used for specific applications)
- Normalization can make it hard to distinguish between nodes of different degrees
- Sometimes normalization can lead to a loss of information

$$\text{Update}(\vec{h}_u, \vec{m}_{N(u)}) = \sigma\left(W_{self} \vec{h}_u + W_{neigh} \vec{m}_{N(u)} \right)$$

W parameters influence how much to weigh the current node in its updated representation

- Separate parameters for node's own features and neighbors allow model to learn different transformations for each
- $\sigma(W\vec{h} + \vec{b})$ is just 1 layer MLP (1 input 1 output)

- Above is node level representation for symmetric normalization, but can also be represented graph level with matrices
- Symmetrically normalized adjacency matrix = $\tilde{A} = D^{-1/2} A D^{-1/2}$

$d_{ii} = \sum_j a_{ij}$ = # of neighbors for node i

$$D^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{d_{11}}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\sqrt{d_{nn}}} \end{bmatrix} \qquad A = \begin{bmatrix} - A_1 - \\ \vdots \\ - A_n - \end{bmatrix}$$

$$D^{-1/2} A D^{-1/2} = \begin{bmatrix} \frac{1}{\sqrt{d_{11}}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\sqrt{d_{nn}}} \end{bmatrix} \begin{bmatrix} - A_1 - \\ \vdots \\ - A_n - \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{d_{11}}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\sqrt{d_{nn}}} \end{bmatrix}$$

$$= \begin{bmatrix} - \frac{1}{\sqrt{d_{11}}} A_1 - \\ \vdots \\ - \frac{1}{\sqrt{d_{nn}}} A_n - \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{d_{11}}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\sqrt{d_{nn}}} \end{bmatrix}$$

$$= \tilde{A}$$

Additional Set Aggregators

- **Universal approximator** - class of functions that can precisely approximate any continuous function given enough capacity/parameters
- **Set function** - takes a set as input
- Any aggregation function with following form is **universal set function approximator** ← this has been proven

$$\vec{m}_{N(u)} = MLP_\theta\left(\sum_{v \in N(u)} MLP_\phi(\vec{h}_v)\right)$$

MLP adjusts parameters to get close to ideal function

$\sum$ ← reduction function

  - This one isn't used it's more so theoretical for a "best case" scenario
  - Researchers aim to make aggregation functions with the power of this but more efficient

- **Reduction function** - takes a set/sequence of values and "reduces"/combines them into a smaller set of values or a single value
- Alternative reductions to sum, like element-wise max/min can be used

- **Transductive learning** - model learns from and makes predictions on fixed set of data points; only works for data seen during training
  - Designed for fixed dataset where you don't want to generalize (e.g., GNNExplainer)

- **Inductive learning** - model learns a function to generate embeddings based on node features and local graph structure
  - Generalizes to unseen data (e.g., PGExplainer)

- Graph Sample and Aggregate (**GraphSAGE**)
  - Samples a fixed-size set of neighbors instead of the full neighborhood/ranks message passing scalable to large graphs
  - Inductive approach
  - Can generate embeddings for nodes it hasn't seen during training

GNN Applications

- Node classification (i.e. predicting if user is bot in a social network)
  - Each node $v \in V$ has associated label y_v
  - Given $G = (V, E)$, goal is to learn representation $\vec{h}_v$ s.t. $f(\vec{h}_v) = y_v$ (f = GNN)

- Graph classification
  - Given $\{G_1, \ldots, G_n\} \subseteq G$ (set of n graphs in dataset) and labels $\{y_1, \ldots, y_n\} \subseteq Y$, learn graph embedding to predict graph label where $f(\vec{h}_G) = y_G$

- Relation/link prediction (i.e. content recommendation)

- Community detection

GNN Types

Basic GNN

- Overall GNN Pass for Classification

conv layer 1. K-layer message passing and updating node representations repeat step 1 for each conv layer

2. Get final node representations
3. Turn into graph embedding
4. Feed into classification model like MLP
5. Get classification → Calculate loss → back prop → gd → update weights → repeat until GNN is trained

- For picking subgraph at the step 2 compute probabilities of node being part of subgraph then select the nodes
  - Ex. Gumbel-Softmax (differentiable which allows for backprop)

- Can also train a GNN on a single graph for node level task
  - Nodes split into train/val/test
- All your nodes are a vector for classification
- After message passing do step 5

Graph Convolutional Network (GCN)

- Uses **self-loop message passing** – aggregation function also includes self node removes need for update function
  - Aggregate($\{\vec{h}_v^{(k-1)}, \forall v \in N(u) \cup \{u\}\}$)
  - Removes need for update function
  - Equivalent to sharing parameters of W_{self} or W_{neigh} since they are already combined when aggregating the message

- Also uses message normalization

- Thus, $\vec{h}_u^{(k)} = \sigma\left(W^{(k)} \sum_{v \in N(u) \cup \{u\}} \frac{\vec{h}_v}{\sqrt{|N(u)||N(v)|}}\right)$ (degree of a)

- One **"convolution"** produces a new feature vector for a single node in the graph
  - Focuses on local neighborhood
  - Like CNN's these can be done in parallel

- **Input dimension** - # of features for each node or edge in graph
- The aggregation and updation of node features occur in a **hidden layer**

- Dimension of hidden layer = dimension of $\vec{h}_v^{(k)}$ at the aggregation/updation
  - Message passing doesn't alter dimension of node so $\vec{h}_v$, but $W\vec{h}_v$ does (# of output neurons)
    - Shape of W = output features (neurons) × input features (size)

- dim(hidden layer) > dim(input layer) captures more complex patterns

Graph Isomorphism Network (GIN)

- Most GNN's cannot distinguish between similar graph structures
- Injective aggregation update of WL Test makes it powerful to distinguish graphs
  - As a result GIN models injective functions

- **Multiset** - a set with possibly repeating elements
- $N(u)$ is a multiset because even if the nodes are distinct, their features x_u could be the same
- GIN aggregation function over multiset must be injective and have great discriminative power
  - **Discriminative power** - how well different aggregation functions can distinguish different multisets

- Certain aggregation/readout function conditions can make GNN as powerful as WL Test

$$\vec{h}_v^{(k)} = MLP^{(k)}\left((1+\varepsilon^{(k)}) \cdot \vec{h}_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}\right)$$

($\varepsilon^{(k)}$: weighs importance of central node)

- Learnable parameters
  - MLP weights/biases
  - ε

- Hyperparameters
  - # of neighborhood aggregations and node update steps (K) - determins how far info will propagate
  - MLP architecture
  - Readout/pooling function
  - Learnable parameter initialization
  - ⋮

Neighborhood Attention and Generalized Update Methods

- Applies a trainable weight/attention to each neighbor when forming the message

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

$\alpha_{u,v}$: weight of message v

13

$\underset{1\times n}{\vec{1}^T} \underset{n\times n}{A} \underset{n\times 1}{\vec{1}}$ will sum all the entries of A

- Oversmoothing can be thought of as the influence of each $\vec{h}_u^{(0)} = \vec{x}_u$ on all of the other final node embeddings $\vec{h}_v^{(k)}$ $\forall v \in V$ in the graph
  - Can quantify the influence by the sum of the Jacobian for $\frac{\partial \vec{h}_v^{(k)}}{\partial \vec{h}_u^{(0)}}$

$$I_k(u,v) = \vec{1}^T \left(\frac{\partial}{\partial \vec{h}_u^{(0)}} \left[\vec{h}_v^{(k)} \right] \right) \vec{1}$$

I_k: Scalar

- Other measures for sensitivity of Jacobian other than sum

- Helps prove nodes more likely to be reached by k steps from node u have a stronger impact on node u representation (e.g. high degree)

Thm: For GNN models using self-update approach and specific aggregation function form, $I_k(u,v) \propto p_{G,k}(u|v)$

$p_{G,k}(u|v)$: probability of visiting node v on length-k random walk from node u

- Basically influence of a input feature on another node v in graph after GNN aggregation layers or how close/likely v is to be reached in random walk

- This theorem motivates various aggregation strategies for weighting each neighbor
- Graph Attention Networks use attention mechanisms to weigh importance of neighbor
  - GAT (attention mechanism) uses linear layer multiplied by embeddings of each node

- More generally, we may use edge and graph level features in the GNN

$$\mathbf{h}_{(u,v)}^{(k)} = \text{UPDATE}_{\text{edge}}(\mathbf{h}_{(u,v)}^{(k-1)}, \mathbf{h}_u^{(k-1)}, \mathbf{h}_v^{(k-1)}, \mathbf{h}_G^{(k-1)})$$

per edge; per graph level features; certain weights possibly applied to each

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

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

$$\mathbf{h}_G^{(k)} = \text{UPDATE}_{\text{graph}}(\mathbf{h}_G^{(k-1)}, \{\mathbf{h}_u^{(k)}, \forall u \in \mathcal{V}\}, \{\mathbf{h}_{(u,v)}^{(k)} \forall (u,v) \in \mathcal{E}\}).$$

all nodes; all edges

can switch order if it depends

Graph Pooling

- **Embedding** - transformation of one form of representation to a lower dimensional vector space
  - Dense vectors (few non-zero elements)

- f_p - Pooling function
- $\vec{z}_u$ - final node embedding for an epoch
- $\vec{z}_G$ - graph embedding used for predictions at graph level

$$f_p(\{\vec{z}_1, \ldots, \vec{z}_{|V|}\}) = \vec{z}_G$$

Mean Embedding

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

f_n: normalising function (could be identity)

- Other embeddings like taking the max of each component across graph
- Sometimes multiple embeddings are concatenated for further information for the classifier

Fine Tuning with Graph Augmentation

- Fine tuning involves...
  - Taking a pre-trained model and using a new (maybe augmented) dataset to slightly adjust model parameters to fit new task
  - Initial layers of pre-trained model are frozen (weights not updated) because they capture the general features
  - Final layers retrained on new dataset or replaced with layers more suitable to the task
    - from scratch?

- Freezing layer involves turning off requires_grad attribute for the tensors containing the parameters of the layer

- Various forms of graph augmentation that can make model more robust

- **Node feature masking** - using a boolean mask (set with threshold mask rate) and "turning off"/keeping random features for all nodes in the graph

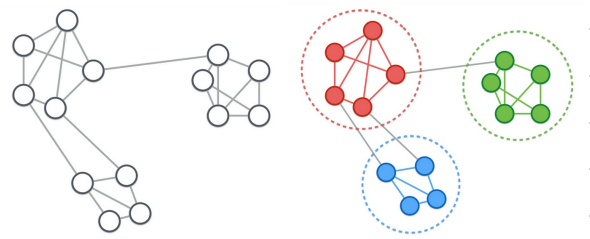
- **Edge perturbation/augmentation** - uses boolean mask with threshold to remove edges from edge index

Copying Data

- **Shallow copy** - creates a new object; for nested object (eg. 2D array) only copies references
- **Deep copy** - creates new object entirely independent of original (including nested)

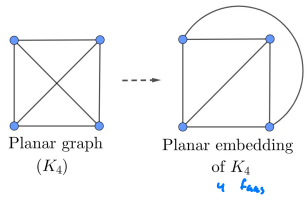
Types of Nonmolecular Graphs

- **Community graphs** - vertices are grouped into clusters with dense connections internally and sparser connections between clusters
  - **Modularity** - a measure to assess how well-defined communities are within network
    - Sometimes you conceptually **division**/partition the network/graph in order to have high internal density and low external density of communities without actually changing the structure of the graph; various division algorithms to achieve best modularity
      - This division can provide simplified analysis and better interpretability of the graph
  - **Clustering coefficient** - measures tendency of nodes to cluster together locally (e.g. tendency for node's neighbors to also be neighbors)



- **Planar graphs** - can be drawn in $\mathbb{R}^2$ without any edge intersections
  - **Connected planar graph** - planar graph with at least 1 edge between any pair of vertices
  - **Faces** - regions of planar graph surrounded by edges (including the **outer region**, the infinitely large region that surrounds the graph)

- K_n - undirected graph where every pair of distinct vertices is connected by a unique edge



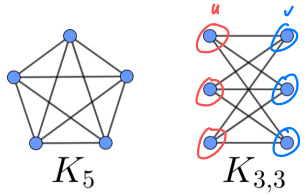
- **Bipartite Graph** - no edges between vertices in the same set when divided into two **disjoint** (no elements in common) sets U and V

- $K_{m,n}$ **(Complete Bipartite Graph)** - a graph that has two disjoint sets of vertices where each vertex in one set is connected to every vertex in the other set (all edges between the sets exist)

- **Euler's Formula:** $|V| - |E| + |F| = 2 \Leftrightarrow G = (V, E)$ is connected planar graph
  - $|V|$: cardinality of set of vertices; $|F|$: faces

- Two graphs are **homeomorphic** if they can be transformed into the other through **edge subdivision** ($A\text{-}B \rightarrow A\text{-}C\text{-}B$) or **edge contraction** ($A\text{-}B \leftarrow A\text{-}C\text{-}B$, the reverse)

- **Kuratowski's Theorem:** A graph is planar $\Leftrightarrow$ a graph does not contain a subgraph that is homeomorphic to K_5 or $K_{3,3}$



- Planar graphs used in areas like transportation to minimize overlap for efficient networks/layouts

Spectral Graph Theory - analyzes properties of graphs through eigenvalues/eigenvectors of matrices representing the graph

- A **spectrum** of a matrix is the set of its eigenvalues
- **Degree matrix (D)** - diagonal matrix where d_{ii} is degree of vertex i
- **Laplacian matrix (L)** - $L = D - A$ (A: adjacency matrix)

- L is symmetric since A and D are symmetric
- 2nd largest eigenvalue of A refers to connectivity

Datasets

- **MUTAG dataset**
  - Collection of graphs representing molecules
  - Nodes/atoms have categorical feature detailing type of atom encoded as one-hot vectors; some with edges for type of covalent bond

- **Mutagenic** - having the ability to cause permanent change in an organism's genes
  - Each of the 188 graphs has binary label indicating whether compound is mutagenic
  - Around 17 atoms per molecule on average

- **Barabási-Albert (BA) Model** - Generates graphs that exhibit **scale-free property**
  - **Node degree** - number of edges connected to a node
  - When adding node to graph, node with higher degree is likely to make another (**preferential attachment**)
  - Keep doing this until desired graph size is reached
- **Motif** - recurring, significant subgraph patterns within larger graph
  - E.g. (**cycle** - nodes connected in closed loop, **house** - looks like a house)

- Certain motifs can often explain various behavior/attributes for an entire graph (especially for molecules)

2 different motifs

- **BA2Motifs**
  - Used to evaluate performance of GNN/explainer in detecting specific patterns within larger networks
  - Contains graphs created using BA that contain motifs (can be any parts of graph where motif is)
    - Input BA into GNN → learns to identify motifs → classifies whole graph based on presence of motif
  - **Deterministic** - premade dataset

- 1000 BA graphs in dataset; 500 have 5-node cyclic motif and 500 have house motif
  - Goal is to classify graph into 1 of 2 classes based on motif it contains

Explainability

- Explainability can...
  - Increase trust in the model
  - Improve the models transparency
  - Allow people to better understand/interpret a GNN model and correct mistake patterns they make

- Subgraph should match labels of original graphs (eg. mutagenic); done to make sure it properly encapsulates whole graph

- How to identify "explainer" subgraphs in a graph because you can't feed them back into the network due to different size/distribution
  - Known as Out-of-Distribution problem
  - More specifically, distribution of data points fall outside distribution of training data (different connectivity, density, structure)

- Difference in structural properties proven via by evaluating predictive capability of subgraph by feeding that and full network into GNN (should be same label/output to be properly explained)
  - Assume GNN already trained

- Graph → Explainer GNN → Explanation subgraph (masked) → Trained GNN → classification
  (must attack subgraph to "prove" non-explainer)
  - Graph → Trained GNN → classification

- Explainer starts with random masks (= edges/node features (parameters that will be adjusted))
  - Edge matrix applied to adjacency matrix, feature mask applied to feature matrix

- Loss function
  - Prediction loss: comparison between * and *
    - Things with masks to highlight minimum edges/nodes for subgraph
    - Train until masks/parameters converge and produce good explanations

GNNExplainer

- First general graph model explainer
- Given trained GNN classifier and its predictions, the explainer outputs subgraph structure and small subset of important node features that are important in GNN classifier prediction

- Incorporates relational information as well as node features
  - Works for an individual graph or set of graphs

- **Mask** - set of learned weights
- "Select" the important edges (for the subgraph) and node features by applying a mask
  - Each edge and node feature type gets a weight between 0 and 1 weighted by importance
  - Adjusted as the explainer is trained
  - Masks applied to A and X through element-wise multiplication
    - * Edge mask M is importance of all possible edges while $A \odot M$ represents important edges that exist in the graph

- Mask approach uses **continuous relaxation** - making a discrete problem into one with continuous values

- **Mean field variational approximation** (GNNExplainer uses) - approximates complex distribution of all possible subgraphs by assuming the presence of an edge is not dependent on the other edges

- **Mutual information** - measure of how much information from one variable provides about another

- Optimization process
  - Randomly initialize probabilities of mask M (with all edges being important)
  - Apply mask to graph
  - Calculate conditional entropy (the loss) ← use $G \odot M$ (masked graph in place of G_S (don't need to extract explainer) ; out of distribution can't do
  - Calculate $\frac{\partial L}{\partial M}$ with backward propagation
  - Use gradient descent to update weights / probabilities of M
  - Repeat

- Once done set threshold on masked graph to extract subgraph (Gumbel-Softmax not needed if explainer already trained)

- Maximizing mutual information = minimizing conditional entropy (as shown below)

e.g. Node classification

- $Y = \{1, \ldots, C\}$: all possible node classes

- GNN aggregates info based off message pass branch-out (computation graph)
  - Goal is to see which features and part of computation graph best "explains" GNN prediction

G_c - node v computation graph (nodes from K iteration message pass)

- $\vec{v} \xrightarrow{G_c} \vec{z} \xrightarrow{GNN} \hat{y}$ (mask F, nodes in G_c)
- Goal is to find $G_s \subseteq G_c$ and node features $X_s \subseteq X_c$ that is most important in prediction $\hat{y}$

H - entropy term

$H(Y \mid G = G_s, X = X_s)$ (Y: GNN output)

- This conditional entropy of Y given G_s and X_s measures how much inputting G_s and X_s (assuming in distribution) would reduce uncertainty of Y

$$\max_{G_s} MI(Y, (G_s, X_s)) = H(Y) - H(Y \mid G = G_s, X = X_s)$$

looking for subgraph G_s that maximizes mutual info between subgraph and feature explain and GNN node does prediction

- Assume all node features explain for now
- Node strongly lowers probability of prediction $\hat{y}$: when removed from $G_c \Rightarrow$ important in classification $\Rightarrow$ should be part of subgraph explanation

Expected value ($\mathbb{E}$) - average outcome of experiment if repeated many times ($\mathbb{E}[Y, \text{outcome}]$)

$$H(Y \mid G_s, X_s) = -\mathbb{E}_{Y \mid G_s, X_s}\left[\log\left(P_\Phi(Y \mid G_s, X_s)\right)\right]$$

$\log_e(x)$ when x is 0-1

- $x > 1 \Rightarrow \log_e(x) > 0$ e.g. $\log_e(2) \to e^? = 2 \to ?$ has to be $< 1 \to$ multiplying a number by itself x times only increases product when > 1
- $x = 1 \Rightarrow \log_e(x) = 0$
- $0 < x < 1 \Rightarrow \log_e(x) < 0$ can't get negative number no matter how small number is

- Each possible $\hat{y} \in Y$ has its own probability so $\mathbb{E}$ does weighted average of all these; allows explain to consider correct and incorrect decisions

Good explainer $\Rightarrow$ given G_S and X_S prob of predicting Y is high $\Rightarrow$ P_Φ (between 0-1) is high / close to 1 $\Rightarrow \log(P_\Phi)$ is small negative (close to 0)

$\Rightarrow -\log(P_\Phi)$ is small positive $\Rightarrow$ max mutual info is higher

(large positive if explainer is bad)

- Ultimate takeaway is we want to minimize conditional entropy across all subgraphs sampled from G ($G_s \sim G$)
- $\min_G H(Y \mid \mathbb{E}_G[G_S], X_S)$
  - This finds entropy of average subgraph rather than every one
  - Added regularization will encourage smaller subgraph explainer

- Although good, it does not have a global understanding of the classifier and does not factor into account explanations for other graphs in the dataset
  - For each new instance, optimization process is required (each graph learns/loads its own mask for GNNExplainer)
  - Each edge gets a weight parameter: H parameters $\propto$ num_edges

PG Explainer

- Parameterized explainer that uses MLP to parameterize generation process of explanations (only includes G_S)
- Has global understanding of explanations; don't need to retrain the explainer for each graph instance
  - MLP, parameterized by ψ, is trained with all graphs in the dataset; GNNExplainer does the optimization independently and has to retrain for each new graph
  - Faster than GNNExplainer for new instances/large datasets

- Different ways to interpret deep learning
  - **White box** / transparent method - use the model's internal structure/parameters to provide insight into the model's decisions
    - e.g. Gradient to see which input features have greatest effect on decision
    - More model specific; focuses on forward/backward propagation
  - **Black box** method - use only the input and output of the model to understand its behavior
    - i.e. creates a simpler, approximate model of a more complex model

- $G_0 \xrightarrow{\text{GNN Classifier}} Y_0$

- PGExplainer also aims to maximize mutual information between $G_S \leq G_0$ and Y_0 $\Rightarrow$ minimize the conditional entropy

  - $\max_{G_S} MI(Y_0, G_S) = H(Y_0) - H(Y_0 \mid G = G_S)$
  - $2^{\text{num. edges}}$ possibilities for G_S, which is inefficient to optimize this function
    - Needs relaxation for picking G_S efficiently; assume graph edges are independent (Gilbert random graph)

- $P(G) = \prod_{(i,j) \in E} P(e_{ij})$
  - Edge probabilities sort of in bernoulli distribution

- $P(e_{ij}) : \theta_{ij}$
- $q(\Theta)$ - distribution over different configurations of G_S; basically the mask of edge probabilities?
  - The optimization will try to learn $q(\Theta)$

- $G_S \sim q(\Theta)$ means randomly generating explainer subgraphs from this distribution; we approximate this

- Ω - Latent variables (all w_{ij})
  - $\Omega = MLP_\psi(Z)$
  - MLP outputs the probabilities of certain edges being in explainer for G_0; input edges which contain edge/node features

- $G_S \approx G_S = f(G_0, \Omega, T, \epsilon)$
  - f can be Gumbel-Softmax; Ω : all w_{ij}

- Temperature (T) - adjusts sharpness/determinism of edge selection process; applied after MLP output
  - $T \to 0 \Rightarrow P(\epsilon_{ij}) \to 0$ or 1
  - $T \to \infty \Rightarrow P(\epsilon_{ij}) \to 0.5$
    - High temp $\Rightarrow$ more exploration; usually used during training

- For node classification, explainer subgraph differs per node v, so this should be part of explainer function (MLP)
  $w_{ij} = MLP_\psi([\vec{z}_i; \vec{z}_j; \vec{z}_v])$; trying to see which edges of computation graph for $\vec{z}_v$ are most important

- For graph classification graph prediction is not conditioned per node so $\vec{z}_v$ not needed
  - $MLP_\psi(\vec{z}_i \oplus \vec{z}_j) = w_{ij}$
    - Do this for all the edges simultaneously

- Final edge weight is some form of $\sigma\left(\frac{w_{ij}}{T}\right)$ (sigmoid)

- Loss function = Cross-entropy(GNN(G_o), GNN(G_s))
  - Can be between the nodes in the graph

- Various regularization techniques to get more compact explainer and include neighboring edges (connect to same node) in the explainer

- Training PGExplainer
  - Per epoch (For G_o in dataset)
    - GNN(G_o) = Y_o (node embeddings from GNN classifier)
    - $MLP_\Omega(Z) = W$ (latent variables w_{ij}) $\xrightarrow{\sigma(\frac{w}{T})}$ final edge probabilities
    - Extract subgraph
    - Obtain prediction for subgraph (assume in distribution)
    - Compute loss (minimize cross-entropy/maximize similarity between predictions)
    - Update Ψ with backprop and gd

Algorithm 2: Training algorithm for explaining graph classification

Input: A set of input graphs with i-th graph represented by $G_o^{(i)}$, node features $\mathbf{X}^{(i)}$, and a label $Y^{(i)}$, a trained GNN model: $\text{GNNE}_{\Phi_0}(\cdot)$ and $\text{GNNC}_{\Phi_1}(\cdot)$.
for each graph $G_o^{(i)}$ **do**
  $\mathbf{Z}^{(i)} \leftarrow \text{GNNE}_{\Phi_0}(G_o^{(i)}, \mathbf{X}^{(i)})$.
  $Y_o^{(i)} \leftarrow \text{GNNC}_{\Phi_1}(\mathbf{Z}^{(i)})$.
end for
for each epoch **do**
  for each graph $G_o^{(i)}$ **do**
    $\Omega \leftarrow$ latent variables calculated with Eq. (11)
    for $k \leftarrow 1$ **to** K **do**
      $\hat{G}_s^{(i,k)} \leftarrow$ sampled from Eq. (4).
      $\hat{Y}_s^{(i,k)} \leftarrow \text{GNNC}_{\Phi_1}(\text{GNNE}_{\Phi_0}(\hat{G}_s^{(i,k)}, \mathbf{X}^{(i)}))$
    end for
  end for
  Compute loss with Eq. (9).
  Update parameters Ψ with backpropagation.
end for

- Uses the reparameterization trick in * to allow for backpropagation
  - See below for VAE

- For inference after MLP extracts edge probabilities threshold can be used to extract subgraph

Variational Autoencoder (VAE)

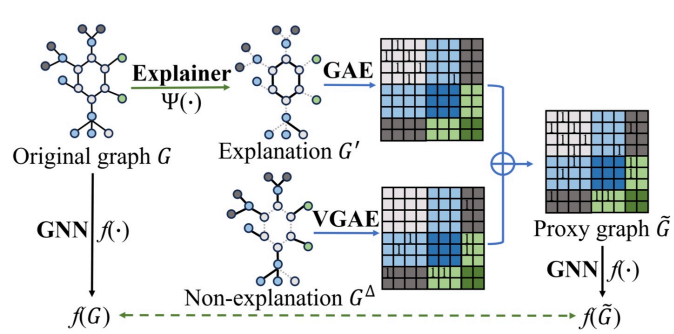
- Autoencoder - compresses data into smaller representation
  - Encoder - layers whose output have less dimensions than input (encoder maps to the latent space)
  - Decoder - takes the output of encoder to try & reconstruct the input
- Variational autoencoder - rather than $encoder(\vec{x}) = \vec{latent}$, $encoder(\vec{x}) = \vec{mean}, \vec{sd}$ → Sample from distribution → $decode(\vec{z}) = input$
  - Loss = Reconstruction loss + KL Divergence
- Sampling prevents backprop so use reparameterization trick
  - $\vec{z} = \mu + \sigma \odot \epsilon$ (μ: learning, σ: standard deviation)

Sampling not done within $\vec{z}$

  - Can do backprop on the deterministic nodes and ignore the sampling
- Disentangled variant autoencoders introduces a β that weighs how much KL Divergence is present
- Variational Graph Autoencoder - uses encoded node/graph embeddings for graph reconstruction

Proxy Explainer

- In practice difficult to train explainer with classifier (G_s) since G_s is out of distribution
- Paper suggests to use graph VAEs to generate explainer and non-explainer and combine to form in distribution $\tilde{G}$ (proxy graph) which can then be put in the classifier



- **Kullback-Leibler (KL) divergence** is a measure of the difference between two probability distributions
  - Always ≥ 0
  - Measures the amount of information lost when Q is used to approximate P
  - $KL(P\|Q) = H(P,Q) - H(P)$
    - Equivalent to difference in conditional entropy

$$D_{KL}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

(D_{KL}: distance)

- Can define distributions P and $Q \Rightarrow$ they are similar $\Rightarrow$ assign similar probabilities to the same sequences

- Algorithm
  - Generate observations of P
  - Calculate probability of Q generating the observations

Gumbel

- Function for picking edges for subgraph needs to be continuous so it is differentiable and explainer can be trained
- Additionally, threshold such as 0.51 probability of being in explainer subgraph is treated same as 0.99

- Uses **Gumbel distribution**
- **Gumbel noise** - allows for exploration for less plausible outcomes, but introduces randomness that mostly preserves edge importance

- **Argmax** - returns index of max value
- **Simplex** - space of probability distribution

Gumbel-Max

- Input: probabilities of each edge importance
- Convert to logits: $\theta_i = \log\left(\frac{p_i}{1-p_i}\right)$ for numerical stability look on
- Calculate and add Gumbel noise to logit
- Choose edge with highest value with argmax (discrete / not differentiable)

$$\vec{z} = \text{one\_hot}\left(\arg\max_i \left[g_i + \log \pi_i\right]\right)$$

($\vec{z}$: 0 or 1 for edge; g_i: noise; π_i: prob)

Gumbel-softmax

- Uses softmax as differentiable approximation instead of argmax
- τ is used for edge sharpness
  - Density of Gumbel-softmax distribution is more ∿ with low temp and —— with high temp
- Forward pass will discretize edges with argmax but backward pass will use approximation for derivative?

Discrete Denoise Diffusion Modeling (DDDM)

- Forward Diffusion Process
  - Through Markov chain, add some noise to the sample each step t; noise defined by transition matrices Q^t (amount of times you apply the noise; Q depends on current state)

- Reverse Denoising Process
  - Neural network trained to predict original data from noisy samples at each step
  - Model learns to reverse the noise

- How GNN, embed timestep t of diffusion into it to see how corrupt it is

- The model adds discrete graph edits in steps (edge addition/deletion, feature edit)
  - Train graph transformer network
  - Perturbed - modified graph
  - Sparsity - # of edges < max possible edges
  - Gaussian is continuous while discrete is more real; Gaussian blurs distinction between connected or not connected
    - Connection/lack of important for molecules so use discrete
  - Distribution in graphs refer to the statistical patterns of degrees, node/edge features (e.g. path length distribution, edge feature distribution, clustering coefficient)

Ex. node degree distribution is: # edges vs # nodes — power law

- "Noise" for graph means adding/removing edges for discrete; much more realistic and has meaning (e.g. bond either does or doesn't exist)

- Loss function is between * and *
  - Multiple components to define loss between graphs (e.g. node/edge features, degrees, weighted sum of these losses)
  - Once transformer is trained, add more noise to your dataset and repeat the process
- Training the DDDM for all graphs in dataset makes DDDM(random graph) = in distribution graph

Denoising Diffusion Probabilistic Models (DDPM)

- T - maximum number of steps in the diffusion process

- Goal of DDPM is to reverse the forward diffusion process
  - Loss function is between the probability distributions

- Per training example $\vec{x}_0 \in$ Dataset
  - Randomly choose/sample timestep $t \in [1, T]$ and noise ε
  - Add appropriate amount of that noise for that timestep to create $\vec{x}_t$
  - Model predicts the noise added
  - Calculate MSE between predicted noise and actual noise
  - Backprop and gd

- Once model is accurate, we can increase T and continue to train

- Inference
  - Perform denoising for each timestep
    - $\vec{x}_{t-1} = \text{scaling}\left(\vec{x}_t - \text{scaling}\,\varepsilon_\theta(\vec{x}_t, t)\right) + \sigma_t \vec{z}$ (random noise)
      - $\varepsilon_\theta(\vec{x}_t, \vec{t})$ - noise predicted by model
      - $\sigma_t \vec{z}$ - $z = 0$ for last t
        - small scalar decreasing with t
        - random noise

DiGress (Discrete Graph Denoising Diffusion)

- Transformers are good at handling sequential data (like iteratively adding noise to a graph which is why they are the model used
- A DDPM model that uses a transformer and explicitly models both nodes and edges
  - Graph transformers has attention mechanism that can simultaneously consider all nodes
- Discrete DDM is better since it maintains structural properties of graph

- Rather than trying to predict noise added from previous timestep it directly tries to predict $\vec{x}_0$
  - This results in model having more consistent target which results in better performance

- For diffusion model to be efficient...
  - $\vec{x}_\epsilon$ should be calculated directly from $\vec{x}_0$
  - It should directly compare $\vec{x}_0$
  - The state of the data as $t \to \infty$ should not depend on $\vec{x}_0$ (starting point for inverse generation is always the same)

$$\vec{x} \longrightarrow z^t = \bar{Q}^t \vec{x} = Q^1 \dots Q^t x$$

- $G^t \to G^{t+1}$ ⇒ Noise is applied independently to each node and edge feature

- Cross-entropy(transformer(G^t), G^0)
  - Parameters θ for transformer ϕ are adjusted for denoising (e.g. attention weights)
  - G^t is not a graph but rather probabilities for each node and edge
    - Cross entropy loss is between these probabilities and the original graph
  - λ - hyperparameter for importance of edges over nodes

↙ Digress code uses MSE

$$\ell(p^t, G) = \sum \text{cross-entropy}(x_i, \hat{p}_i^x) + \lambda \sum \text{cross-entropy}(e_{ij}, \hat{p}_{ij}^e)$$

- Digress is permutation invariant (does not since $n!$ matrices can represent the same graph)

- Training
  - $G_0 \to G^t \sim X\bar{Q}_X^t$ and $E\bar{Q}_E^t \to z = f(G^t, t) \longrightarrow$ pass: transformer$(G^t, z) \to$ calculate loss and optimize
    - Separate noise transitions for nodes and edges
    - get structural and spectral features
  - More features describing the graph can be calculated and added which improves Digress performance (like eigenvalues / valence of each atom)
    - Singular value - eigenvalue of matrix $A^T A$ (square)
      - Takes long time to compute for large graphs (3 days on A100)

- Inference
  - Iteratively denoises from $t = T$ to 1
  - $z = f(G^t, t) \to$ pass: $\phi_\theta(G^t, z) \longrightarrow$ find probability distributions of graph at t-1 based on predictions for $\vec{x}_0 \to$ sample new graph of t-1

RePaint Diffusion

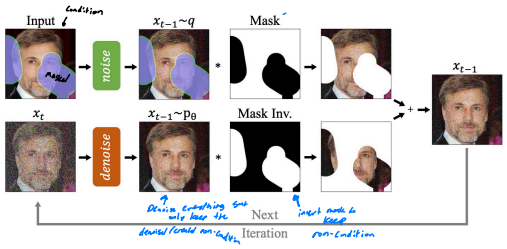
- **Condition** - does not change when passed through the diffusion model

- The binary mask
  - **Masked region (0)** - defines which parts of the input need to be generated/inpainted
  - **Unmasked region (1)** - the condition

- Input is completely noisy x
  - At each time step replace unmasked region with the condition

- **Resampling** allows RePaint to make multiple attempts at denoising per timestep

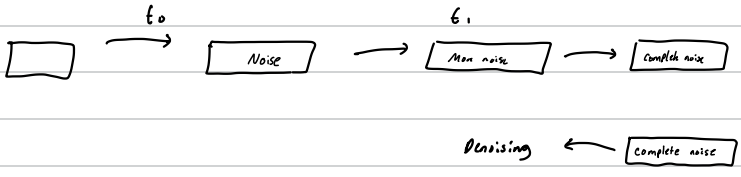
- Inference
  - Initialize $\vec{x}^T$ with random noise
  - For unmasked region, use condition with sampled noise
  - For masked regions, use models prediction of next most step
  - Apply the mask
  - Resample



- Condition is still noised but not with the DDPM

- Repaint is an algorithm that uses a pre-trained DDPM

- At



  - Using trained diffusion model, at each step of denoising insert the original condition in the corresponding space