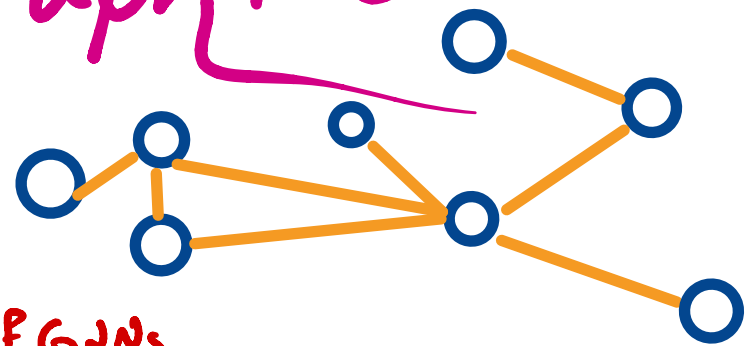
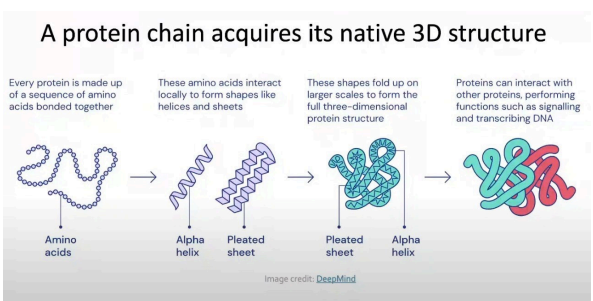
Graph Neural Nets



tasks of GNNs

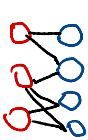
1. Node level Prediction: structure and position of a node
2. Edge level Prediction: drug side effects or graph-based recom systems
3. Graph level prediction: of an entire graph or subgraphs - drug discovery, physics simulations

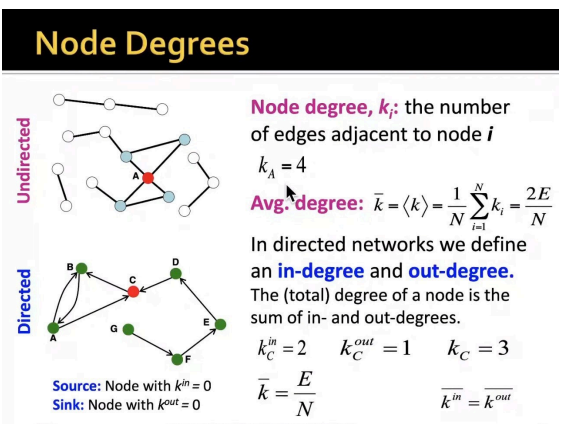


Alphafold's key idea: spatial Graph
nodes: amino acids
edges: proximity between amino acids

Bipartite Graph

a graph whose nodes can be divided into two sets U and V such that U and U are two independent sets and only interact with nodes of the other set.





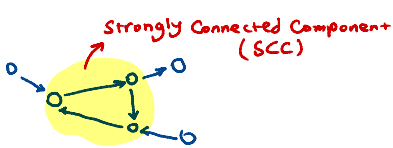
★ adjacency matrix is extremely sparse.

Connectivity

an undirected connected graph is one that has at least one path between each two nodes.

in directed graphs we have:

- Strongly connected: has a path from each node to every other and vice versa.
- Weakly Connected: is connected if we ignore the direction of edges.



Traditional ML Pipeline

train an ML model → feed new graph/node → obtain features → predict

Nodes

given $G=(V,E)$, learn function $f: V \to \mathbb{R}$

Characterizing the properties of a node

Degree

↓ makes nodes of the same degree indistinguishable.

Centralities

while degree only counts the neighboring nodes, Centrality takes their importance into account.

Eigenvector Centerality

the more important neighbors are the higher importance of the node.

$$c_v = \frac{1}{\lambda} \sum_{u \in N(v)} c_u \iff \lambda c = Ac$$

λ some constant

A: adjacency matrix
C: centrality vector

Betweenness Centrality

high value if it lies on many shortest paths between other nodes. (transition hub)

Closeness Centrality

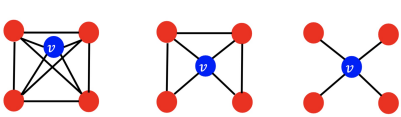
high if a node has small shortest path length to all other nodes

$$c_v := \sum_{u \neq jv} \frac{1}{|\{\text{shortest paths between } u, v\}|}$$

Clustering Coefficient

measure how connected v's neighbors are.

$$e_v := \frac{1}{\binom{k_v}{2}} |\{\text{edges among } N(v)\}| \in [0,1]$$



$e_v = 1$ $e_v = 0.5$ $e_v = 0$

A Note on Eigenvalue Centrality (why eigenvalues?)

eigenvalue centrality is the weighted sum of k neighbor centrality. This is a recursive definition. It's also similar to the definition of eigenvalues.

$$Av = \lambda v$$

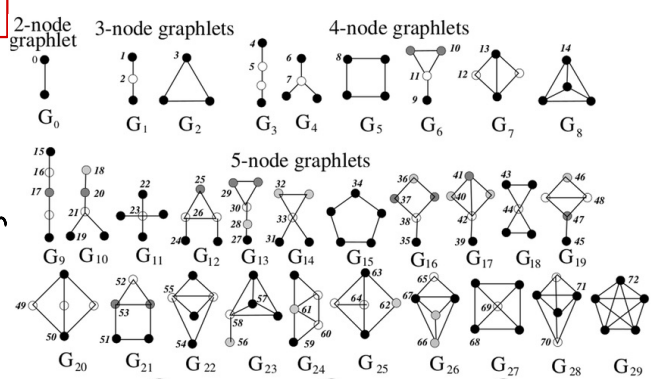
for each component of v:

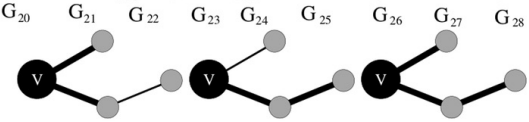
$$\sum_j A_{ij} v_j = \lambda v_i$$

this means that v_i is the weighted sum of the components of its neighbor j weighted by A_{ij}. This is in definition similar to the centrality we are looking for.

Graphlets

Small subgraphs that describe u's neighborhood. This can be encoded as a vector of degrees in the graphlet known as graphlet Degree Vector (GDV)





Orbit	0	1	2	3	4	5	6	...	72
GDV(v)	2	1	1	0	0	1	0	...	0

Edge (link-level)

based on existing links, rank all node pairs and select top k pairs. (social network)

1. links missing at random
2. Links over time

the ranking depends on our scoring system:

Local Neighborhood Overlap

number of common neighbors of the two nodes.

Jaccard's Coefficient: $\frac{|N(u) \cap N(v)|}{|N(u) \cup N(v)|}$

if two nodes aren't connected this would always be 0.

↓

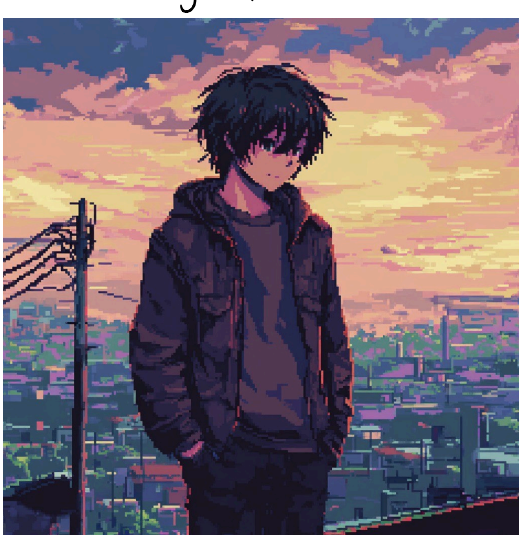
Global Neighborhood overlap

> Theorem
> the number of paths of length N between two nodes is $P^{(N)}$ or powers of the adjacency matrix.

we use Katz index for GNO

$$C := \sum_{i=1}^{\infty} \beta^i A^i = (I - \beta A)^{-1} - I$$

C sums over paths of len 1 up to ∞,
β is the discount factor to give lower importance to longer paths.



Graph-level

Kernel $K(G, G') \in \mathbb{R}$ measures similarity b/w data

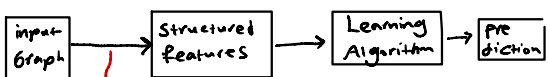
Kernel must be positive semidefinite

a type of kernel is Graphlet kernel and there are other types as well

Some Book Notes

At first glance, node classification appears to be a straightforward variation of standard supervised classification, but there are in fact important differences. The most important difference is that the nodes in a graph are not *independent and identically distributed (i.i.d.)*. Usually, when we build supervised machine learning models we assume that each datapoint is statistically *independent* from all the other datapoints; otherwise, we might need to model the dependencies between all our input points. We also assume that the datapoints are *identically distributed*; otherwise, we have no way of guaranteeing that our model will generalize to new datapoints. Node classification completely breaks this i.i.d. assumption. Rather than modeling a set of i.i.d. datapoints, we are instead modeling an interconnected set of nodes.

So in summary, traditional ML for graphs looks like this:



Feature engineering (node, graph, or edge level)

↓

Can we do this automatically?

↓

Node Embeddings

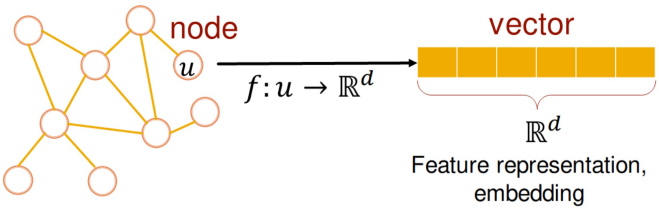


Figure 2.1: Node Level Embeddings of graphs map each node to a vector

Similarity in the embedding space must approximate similarity in the graph. this defines our goal:

this is a pair-wise decoder

$$\text{similarity}(u, v) \simeq \text{Dec}(\text{Enc}(u), \text{Enc}(v))$$

$$\text{similarity}(u, v) \simeq z_u \cdot z_v$$

the simplest approach is to use Shallow encoder where Z is a look-up table of embeddings for each node.

How do we define node similarity? this is mostly what seperates these algorithms.

Random Walk

given a graph and a starting point, we visit L number of nodes. then

$$Z_u^T Z_v \approx \text{Probability that } u \text{ and } v \text{ co-occure on a random walk}$$

the idea: if random walk from u visits v with a high probability, u and v are similar.

goal: learn $f: u \to \mathbb{R}^d$: $f(u) = z_u$

objective:

$$\max_f \sum_{u \in V} \log P(N_R(u) \mid z_u)$$

N_R is the neighborhood of u by strategy R

we want to learn representations that predict which nodes to find in RW.

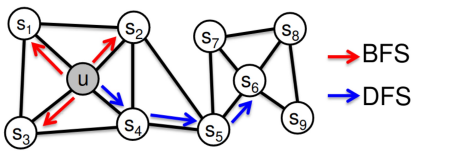
$$P(v|\boldsymbol{z}_u) := \frac{\exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_v)}{\sum_{n \in V} \exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_n)}$$

However, this function is expensive to evaluate. The two $\sum_{n \in V}$ loops already give $O(|V|^2)$ time complexity. The solution to this problem is **negative sampling**[1], which provides the estimate

$$\log \frac{\exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_v)}{\sum_{n \in V} \exp(\boldsymbol{z}_u \cdot \boldsymbol{z}_n)} \simeq \log \sigma(\boldsymbol{z}_u \cdot \boldsymbol{z}_v) - \sum_{i=1}^{k} \log \sigma(\boldsymbol{z}_u \cdot \boldsymbol{z}_n) \qquad (n_i \sim P_v)$$

node2vec

Similar to Randomwalk but has a flexible notion of network neighborhood.



two Classic strategies of neighborhood:

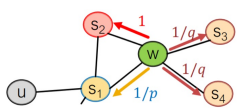
BFS: gives a Local microscopic view

DFS: gives a Global microscopic view

node 2vec can extrapolate betweenthe two

It has two hyperparams:

1. Return (p): how likely to return to previous
2. In-out (q): ratio of BFS vs. DFS



$w \to$

	Prob.
s_1	$1/p$
s_2	1
s_3	$1/q$
s_4	$1/q$

→ normalize & flip a coin!

node2vec performs good on node classification. Randomwalks are pretty efficient as well.

Predicted probability of u, v co-occurring

Summation over all vertices

$$L(\text{Enc}) := -\sum_{u \in V} \log P(N_R(u)|z_u) = -\sum_{u \in V} \sum_{v \in N_R(u)} \log P(v|z_u)$$

Summation over nodes seen on random walk from u

Graph Embedding

a simple idea is to just average over the embeddings of nodes in a graph

another idea is to sample anonymous walks embed them, and concatenate their embeddings to get the embedding of the graph.

there are more advanced methods (later)

PageRank is solved using power iteration

1. Initialize r^0 with some value
2. Iterate $r^{(t+1)} = M.r^t$
3. Stop when $|r^{(t+1)} - r^{(t)}| < \varepsilon$

limitations of node embeddings via random walks (or PageRank) is that we can't obtain embeddings for nodes not seen in training set.

another limitation is these embeddings cannot capture structural similarity. So if nodes with the same neighbors are in different parts of the graph we get similar embeddings.

Graph as a Matrix

PageRank

the idea behind Google

web in the old days was a collection of static webpages connected with link.

"Can we rank webpages"?

webpage ⟶ Node
links ⟶ edge

an idea is to treat links as votes, the more incoming links, the more important that page becomes. This becomes a recursive problem.

This is solved by Stochastic Adjacency Matrix

let page j have d_j out-links,
if $j \to i$, then $M_{ij} = \frac{1}{d_j}$

Columns of M sum up to 1

we define Rank Vector R, as r_i the importance score of page i $\sum_i r_i = 1$ then we have

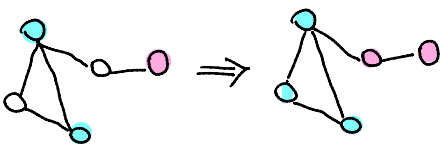
$$r = M.r$$

Solution?

Deep Representation Learning and Graph Neural Networks

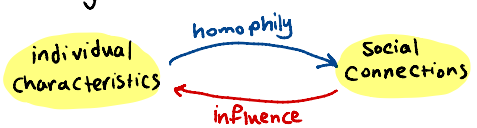
ML with Graphs

an example is classifying nodes in a graph using labels of some of the nodes. (Semi-supervised)



message Passing

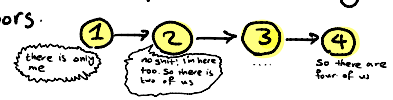
Similar to PageRank, labels of a node are influenced by their neighbors.



One way is Collective Classification where the labels are influenced by 1st degree neighbors. (Markov Assumption)

Probabilistic Relational Classifier measure the weighted average of labels in the neighborhood.

each node in a path listens, updates and passes a message to its neighbors.



Each node collects info from its neighbors, and considers the prior belief of what label it must have.

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

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

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

So the basic message passing can be thought of as a MLP

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

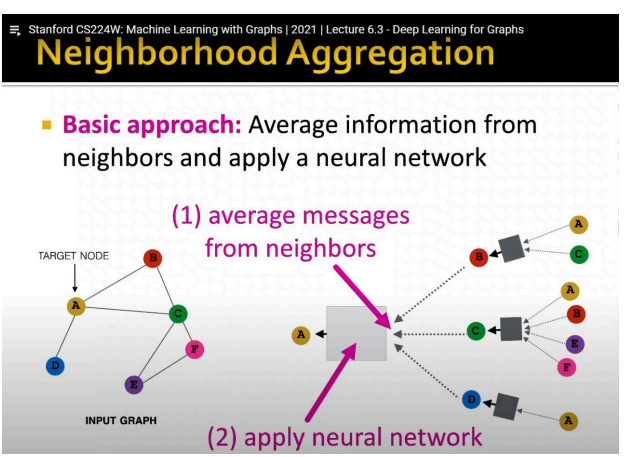
Deep Learning in Graphs

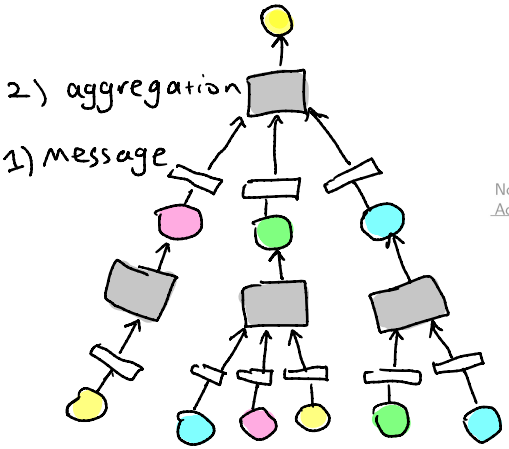
a naive approach is to feed the adjacency matrix A to a feed forward network.

the problem:

- $O(|V|)$ parameters
- Graph size is inherently baked into the size of the neuural network
- Sensitive to node ordering

the solution is to use a permutation invariant aggregation method.





different algorithms differ in how they aggregate messages and what messages they pass between nodes.

the idea of a single GNN layer is compress a set of message vectors into a single vector.

GNN Layer = Message + Aggregation

1) Message Computation

$$m_u^{(l)} = MSG^{(l)}(h_u^{(l-1)})$$

An example could be a linear transformation on the embedding of the node.

$$m_u^{(l)} = W^{(l)} h_u^{(l-1)}$$

2) Aggregation

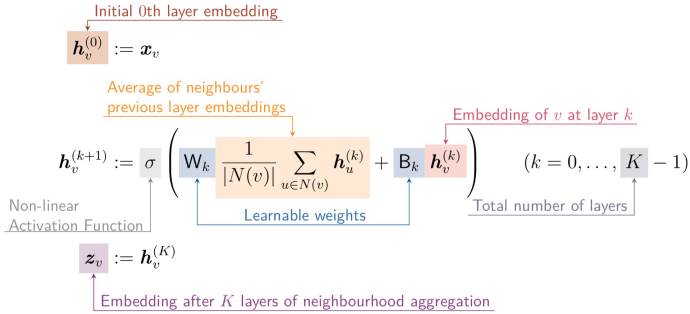
must be order-invariant as order of input nodes should not matter.

examples: Sum(·), Mean(·), Max(·)

but this way, when computing $h_v^{(l)}$ the information we already have of node v gets lost! Solution?

Compute message from node v as well, using a different computation and for aggregation, ConCat the two together.

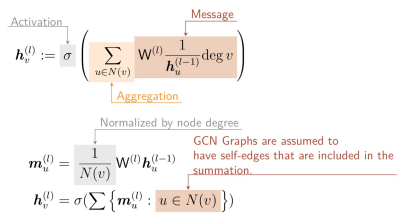
$$h_v^{(l)} = \text{CONCAT}(AGG(\{m_u^{(l)}, u \in N(v)\}), m_v^{(l)})$$



Graph Convolutional Network (GCN)

ctions are

where

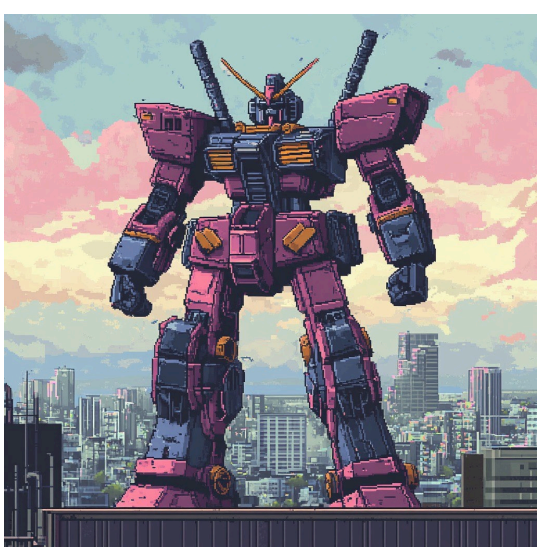


message transformation is linear and aggreg is a sum.

Graph Attention

we keep weights of relationship between node v and its neighbors u.

$$h_v^{(l)} := \sigma\left(\sum_{u\in N(v)} \alpha_{v,u} W^{(l)} h_u^{(l-1)}\right)$$



An **attention mechanism** computes $\alpha_{v,u}$. Define the **attention coëfficients**

$$e_{v,u} := a(\mathbf{W}^{(l)}\boldsymbol{h}_u^{(l-1)}, \mathbf{W}^{(l)}\boldsymbol{h}_v^{(l-1)})$$

Then we normalise $e_{v,u}$ into the attention weight using softmax:

$$\alpha_{v,u} := \frac{\exp e_{v,u}}{\sum_{k\in N(v)} \exp e_{v,k}}$$

In **Multi-head Attention**, multiple attention scores are used and the result of each attention "head" is aggregated:

$$\boldsymbol{h}_v^{(l)}[j] := \sigma\Big(\sum_{u\in N(v)} \alpha_{v,u}[j]\mathbf{W}^{(l)}\boldsymbol{h}_u^{(l-1)}\Big)$$

$$\boldsymbol{h}_v^{(l)} := \mathrm{Agg}(\boldsymbol{h}_v^{l}[j] : j)$$

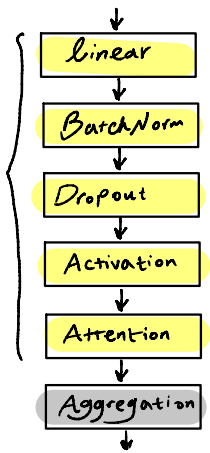
So in contrast with other domains, adding more layers to a GNN does not yield better results. we should first analyse how big we need to have the receptive field.

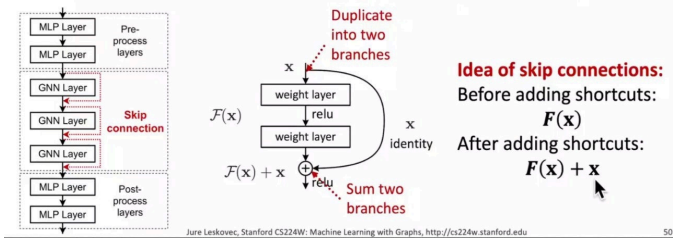
an option would be to use layers in a GNN that don't pass messages (like MLP).

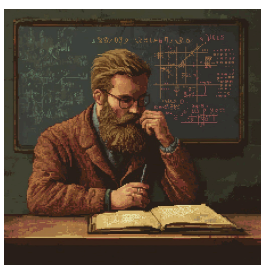
the other option is to use skip connections

a GNN layer could look like this:

Transformation







a standard way of building GNN from a layer is to simply stack them on each other.

if many layers are stacked the problem of over-smoothing occurs where the embeddings of the nodes converge.

Graph Augmentation

in many cases (sparse / dense) graphs, the simple input graph may not be suitable for computation.

Receptive Field

the set of nodes that determine the embedding of a node. as the number of GNN layers increases, the receptive field of a node covers more of the graph.

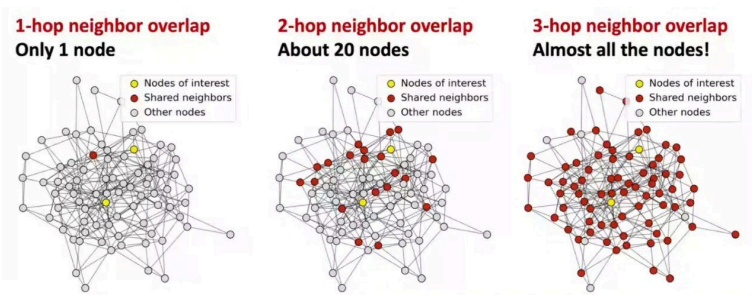
Feature Augmentation

Sometimes the input graph is just adj matrix and the nodes lack any features.

Solution: add unique one-hot ids to nodes or assign constant feature to all nodes.

Structural Aug

* adding virtual (like 2-hop neighbor) nodes to a sparse graph.
* if Graph is dense, sample node neighborhood.



Training GNN

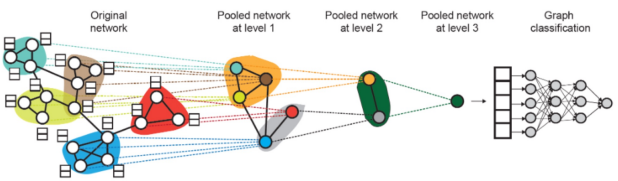


Figure 5.1: DiffPool: A Hierarchical Pooling

This is a good way of embedding entire graphs.

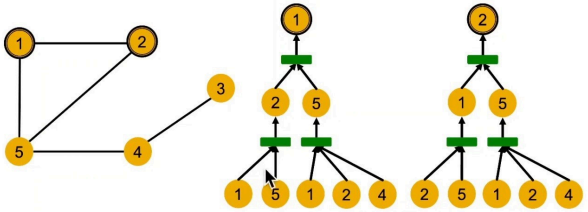
other things, such as loss functions, evaluation metrics, etc are similar to machine learning in general.

Train/Test Split in Graphs: Something special about Graphs is that there is information leakage meaning that since nodes are connected the splitted data in train and test are not totally independent.

How Expressive are GNNs?

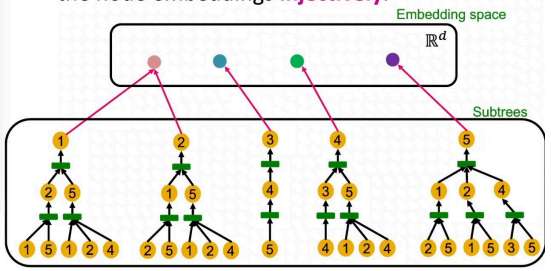
if a graph neural net has no node emb then some of the nodes would be structurally indistinguishable.

- **Ex:** Nodes 1 and 2's computational graphs.



without features, the computation graph of node 1 and 2 are the same.

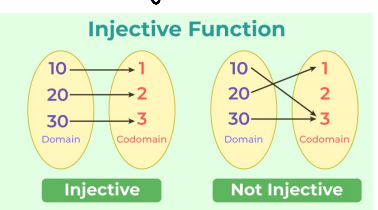
- Most expressive GNN should map subtrees to the node embeddings **injectively**.



a key factor in the expresiveness of GNNs is the aggregation function being able to distinguish different variation of neighbors.

Neighbor aggregation is a function over multi-set.

the most expressive GNN is the one which uses an injective function:



Theorem 6.1 (Xu et al. ICLR 2019). *Any injective multi-set function can be expressed as*

$$\phi\left(\sum_{x\in S} f(x)\right)$$

where ϕ, f are non-linear functions.

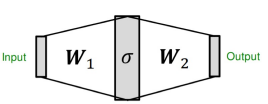


Figure 6.2: MLP with one hidden layer

Theorem 6.2 (Universal Approximation Theorem, Hornik et al., 1989). *1-hidden-layer MLP with sufficiently large hidden dimensionality and non-polynomial activation function σ can approximate any function to arbitrary accuracy.*

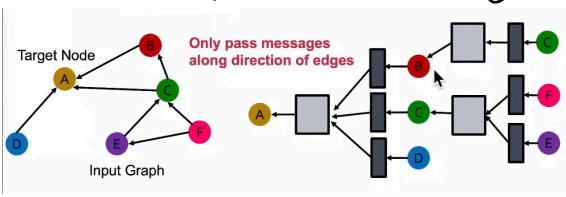
Therefore we can use the following structure to model any injective multiset function. Usually a hidden dimension of 100 to 500 is sufficient. This brings us the most expressive GNN: **Graph Isomorphism Network** (**GIN**).

$$\text{MLP}_\phi\left(\sum_{x\in S} \text{MLP}_f(x)\right)$$

GIN uses the two-MLP structure above and it is the most expressive message passing GNN.

Relational GCN

a graph where every node and relation is labeled by a type (biomedical or event graphs)



the aggregation is similar to max pooling

$$h_v^{(l)} := \sigma\left(\sum_{u \to v} W^{(l)} \frac{h_u^{(l-1)}}{\deg_{\text{in}} v}\right)$$

Activation · Message · Aggregation

now in RGCN we have different weights for each relation type.

$$h_v^{(l)} := \sigma\left(\sum_{r \in R} \sum_{u \in N^r(v)} \frac{1}{c_{v,r}} W_r^{(l)} h_u^{(l-1)} + W_0^{(l)} h_v^{(l-1)}\right)$$

r-Neighbourhood · r-degree $c_{v,r} := |N_r(v)|$

each relation has L matrices for each layer of the GNN: $W_r^{(2)}\ W_r^{(2)} \dots W_r^{(L)}$

this means rapid growth of parameters (over-fitting)

2 solutions for this:

1. Block Diagonal Mat: make the weights matrix sparse.

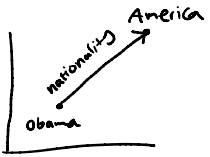
2. Weight Sharing: represent W_r as a linear combination of basic transformations which are shared across all relations. $W_r = \sum_b a_{rb} \cdot V_b$ (basic matrix; importance weights)

Knowledge Graphs

KGs are represented as (h, r, t) where head (h) has relation (r) with tail (t).

the goal is to make emb((h,r)) close to emb(t)

TransE



if fact is true, $h + r \approx t$

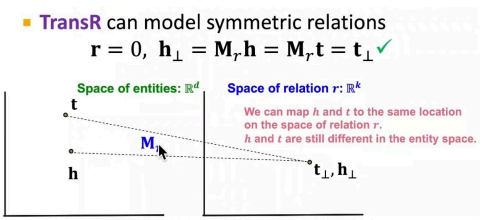
Scoring Function $-\|h + r - t\|$

it is not able to capture symmetrical relations.

TransR

uses M_r projection mat to transform from entity space to relation space and then perform TransE.

does not support composition relations. (my mother's husband is my father)



DistMul

Similar to TransR but uses different method to score embeddings. it can be thought of as Cosine Similarity between h.r and t.

$f_r(h.r, t_2) > f_r(h.r, t_1)$

Can not model inverse relations as $f_{r_2}(h, r_2, t) = f_{r_1}(h, r_2, t)$

also Can NOT model Composition Relations.

ComplEx

based on DistMul but embeds into complex vector space.

$\bar{u} = a - bi$

$u = a + bi$

$\bar{u}$ is Conjugate

Score function: $\mathrm{Re}(\sum_i h_i \cdot r_i \cdot \bar{t}_i)$ takes the real part.

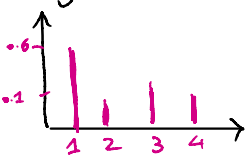
also Can NOT model Composition relations.

(* I didn't make notes for KG reasoning (next lesson))

Generative Models for Graphs

Some properties of real-world graphs:

1) Degree Distribution $P(k)$ probability that a randomly chosen node has a degree of k.



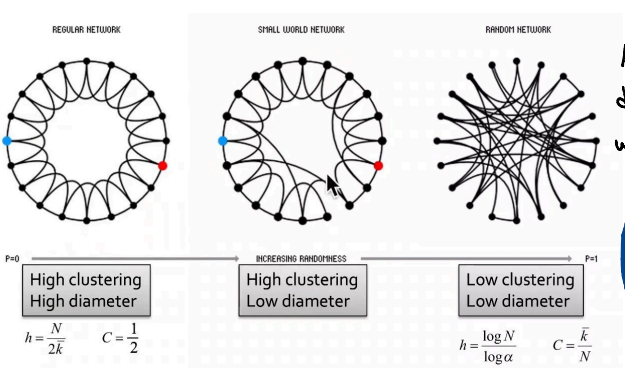
2) Clustering Coefficient: how connected are the neighbors of a node. $\frac{\text{number of edges}}{\binom{k}{2}}$

3) Path Length (diameter): the maximum or average of shortest path between any two nodes.

Small World Model

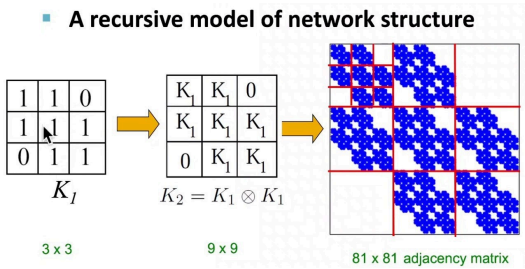
tries to have high clustering with low avg. shortest path (the two oppose each other)

1) Create a small latrice
2) rewire and add randomness



Kronecker Model

- A recursive model of network structure



the Kronecker Product takes every cell of M1 and multiplies it by the whole M2 matrix.

the model iteratively uses KP on an initiator graph

$$K_1 \otimes K_1 \otimes K_1 \otimes K_1$$

Kronecker and real graphs are similar. all you need to do is choose the right initiator matrix.

Deep Generative Models for Graphs

task1: generate graphs similar to a given set of graphs
task2: " " that optimize given objectives (such as drug discovery)

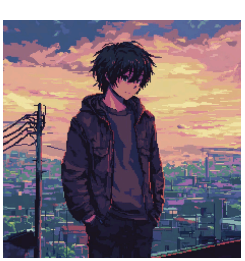
$p_{data}(G)$ —learn & sample→ $p_{model}(G)$ —sample→ G' (new graph)

make $p_{model}(x;\theta)$ close to $p_{data}(x)$

So we follow Maximum Likelihood

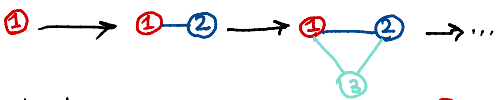
$$\theta^* = \arg\max_\theta \mathbb{E}_{x \sim p_{data}} \log p_{model}(x|\theta)$$

Idea: Chain Rule. we want to model a complex distribution over graphs, we don't know how, so we break it down to smaller distributions.



GraphRNN

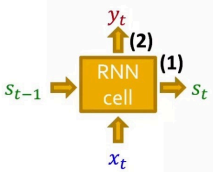
generating a graph sequentially by adding nodes and edges.



this has become a sequencing problem → RNN

- s_t: **State** of RNN after step t
- x_t: **Input** to RNN at step t
- y_t: **Output** of RNN at step t
- **RNN cell:** W, U, V: Trainable parameters

In our case s_t, x_t and y_t will be scalars (edge probabilities)



The RNN cell:

(1) Update hidden state:

$$s_t = \sigma(W \cdot x_t + U \cdot s_{t-1})$$

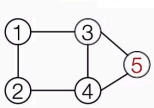
(2) Output prediction:

$$y_t = V \cdot s_t$$

as the generation goes on, a new node can connect to any/all previous nodes. the model must remember all the nodes and edges → Scale Problem!

So we use BFS ordering!

- **Breadth-First Search node ordering**



BFS ordering

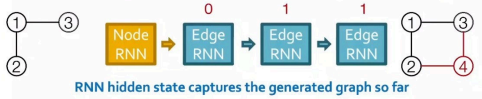
"Recipe" to generate the left graph:

- Add node 1
- Add node 2
- Connect 2 with 1
- Add node 3
- Connect 3 with 1
- Add node 4
- Connect 4 with 3 and 2

- **BFS node ordering:**
  - Since Node 4 doesn't connect to Node 1
  - We know all Node 1's neighbors have already been traversed
  - Therefore, Node 5 and the following nodes will never connect to node 1
  - We only need memory of 2 "steps" rather than n − 1 steps

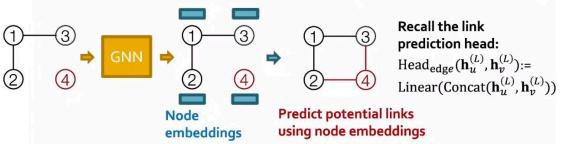
We can also use GNNs to generate graphs (GCPN)

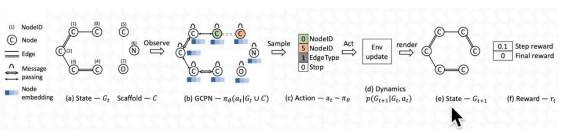
- **Sequential graph generation**
- **GraphRNN**: predict action based on **RNN hidden states**



RNN hidden state captures the generated graph so far

- **GCPN:** predict action based on **GNN node embeddings**





- **(a)** Insert nodes
- **(b,c)** Use GNN to predict which nodes to connect
- **(d)** Take an action (check chemical validity)
- **(e, f)** Compute reward

- **Two parts:**
- **(1) Supervised training:** Train policy by imitating the action given by real observed graphs. Use gradient.
  - We have covered this idea in GraphRNN
- **(2) RL training:** Train policy to optimize rewards. Use standard policy gradient algorithm.
  - Refer to any RL course, e.g., CS234

Scaling up GNNs

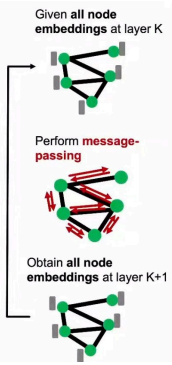
But the computation graph grows exponentially especially if we hit a hub node.

↓

neighbor sampling : at each hop, pick at most H_k neighbors.

if we use a mini-batch SGD training for a GNN and we select nodes that are independent of each other, we cannot effectively train GNN.

- **Naïve full-batch implementation**: Generate embeddings of all the nodes **at the same time.**
  - Load **the entire graph and features**
  - **At each GNN layer:**
    - **Compute embeddings of all nodes using all the node embeddings from the previous layer.**
  - Compute the loss
  - Perform gradient descent

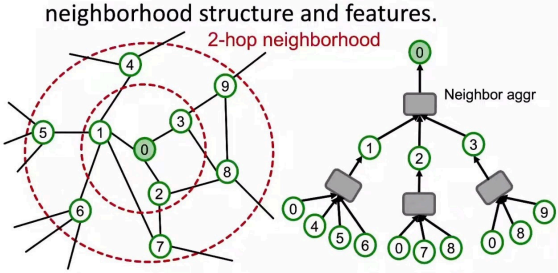


↳ it takes so long and much computation

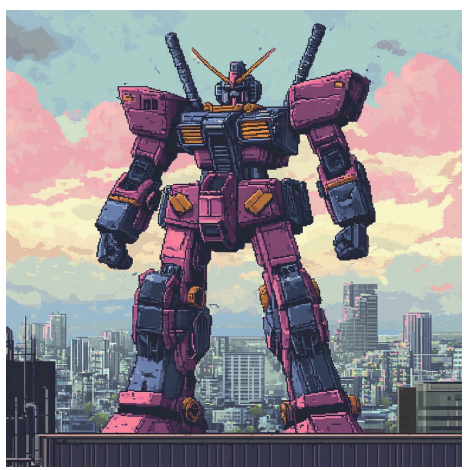
GraphSAGE neighborhood Sampling

RECALL: GNN generate node embeddings via neighborhood aggregation.

- **Observation:** A 2-layer GNN generates embedding of node "0" using 2-hop neighborhood structure and features.



if # of layers or the neighborhood is small we can ignore much of the network. now we can compute gradients in a reliable way and use SGD.



X : @Hesamation