V1.1

Video 1:→ Neural Message Passing

→ Goal: learn meaningful co-ordinates (a.k.a embeddings) for each node in the graph

How do those embeddings help in plotting the nodes in coordinate space?

Why do we need to be neighbourhood aware?

It builds a neighbourhood-aware embeddings for each node in a graph

graph → how? → co-ordinates

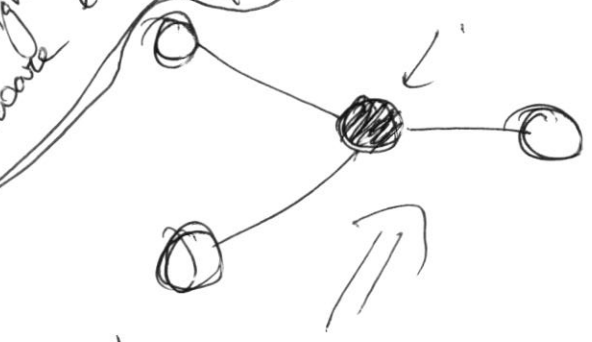
map the nodes to co-ordinate space and draw decision boundaries for classification downstream

→ Not just co-ordinates but a meaningful embeddings

Meaningful :- Embeddings that reflect/store info^n. about the neighbouring nodes

→ "Neighbourhood-aware" embeddings

e.g. if two nodes share many neighbouring nodes, these two nodes should be close to each other in the co-ordinate space



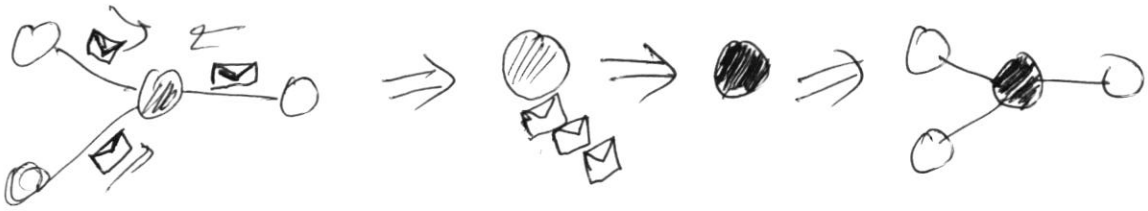
Nodes : people

Node features : age, network → vector

edges :- in phone contacts

edge features : no. of phone calls in last year

Neural Message Passing framework:



(2)

v1.2

This new feature is in addition to features 'age' and 'networkTh' of the individual

After one/more iterations of neural message passing, the updated node has not just the 'age' and 'networkTh' features but also a new feature that computed by aggregating the features (messages) of the neighbours and then updating the previous value of this additional feature.

Reach with Packet Notation

Neural Message Passing has 3 main functions:-

1. Message function: → Computes messages using node/edge features
2. Aggregation fn. → combines all the messages from neighbouring nodes in a fixed-length vector.
3. Update fn. → computes the new value of embedding using aggregated messages and the old node embedding.

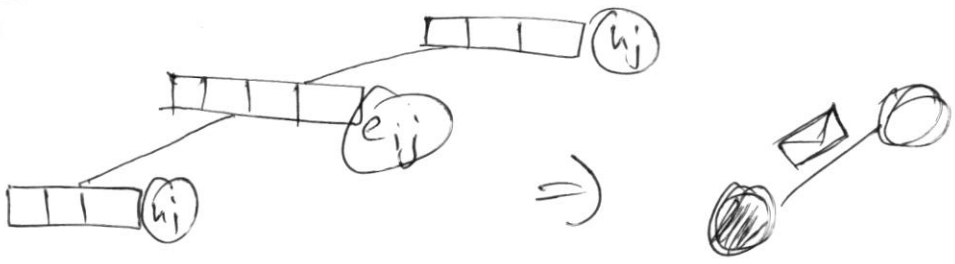
Message fn.

$$m_{ij}^{(k)} = M\left(h_i^{(k)}, h_j^{(k)}, e_{ij}\right)$$

→ message from node 'j' to 'i' in k^{th} iteration of message passing

→ Message fn.

v1.3



Examples of Message fn. from literature

→ $m_{ij}^{(k)} = h_j^{(k)}$ → Copy of the neighbour's features

→ $m_{ij}^{(k)} = \frac{1}{c_{ij}} h_j^{(k)}$ ← Normalized (scaled by constant c_{ij}) features of neighbour.

→ $m_{ij}^{(k)} = \alpha\left(h_i^{(k)}, h_j^{(k)}\right) h_j^{(k)}$ ← Scale neighbour's features h_j by a constant

α ← Attention fn.

The neighbours of node 'i' and find a [similarity] fn that looks at all the features of node 'i' and, say, node 'j'.

Based on the [similarity] in the features of two nodes 'i' and 'j', the attention fn. comes up with an attention score 'α'.

Then, the feature h_j is scaled by 'α'

Applying $\alpha(\,)$ on all the neighbours of a node, each neighbour gets a diff-weight that finally sum to 1.

VI.4

Aggregation fn.

$$\hat{m}_i^{(k)} = \bigoplus_{j \in N_i} m_{ij}^{(k)}$$

→ ⊕ aggregates all the messages m_{ij} coming from ~~one~~ the N_i neighbours of i^{th} node.

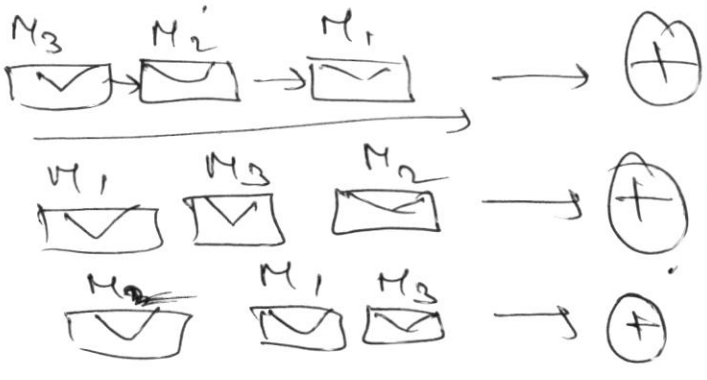
Properties of Aggregation fn.

~~① Permutation invariant; gives the same answer regard~~

This fn. should give the same answer regardless of :-

① No. of neighbouring nodes ← Fixed length representation

② Order of the i/p messages ← Permutation invariant.



Same answer

Aggregation Function Examples :-

fixed length rep. + Permutation invariant

① $\hat{m}_i^{(k)} = \sum_{j \in N_i} m_{ij}^{(k)}$ (Sum)

② $\hat{m}_i^{(k)} = \max_{j \in N_i} \left(m_{ij}^{(k)} \right)$ (Max)

③ $\hat{m}_i^{(k)} = \frac{1}{|N_i|} \sum_{j \in N_i} m_{ij}^{(k)}$ (Average)

~~Different kinds~~ Update fn.

$$h_i^{(k+1)} = \phi\left(h_i^{(k)}, \hat{m}_i^{(k)}\right)$$

$h_i^{(k)}$: current value of embedding

$\hat{m}_i^{(k)}$: aggregated message

Examples of Update fn.

① $h_i^{(k+1)} = \sigma\left(W^{(k+1)} \hat{m}_i^{(k)}\right)$

$\hat{m}_i^{(k)}$ is included in $\hat{m}_i^{(k)}$ is included

self-connecting edge

② $h_i^{(k+1)} = \sigma\left(W_{self}^{(k+1)} h_i^{(k)} + W_{neigh}^{(k+1)} \hat{m}_i^{(k)} + b^{(k+1)}\right)$

bias

③ $h_i^{(k+1)} = \sigma\left(W^{(k+1)} \text{Concat}\left(h_i^{(k)}, \hat{m}_i^{(k)}\right)\right)$

aggregated message

Unlike the 1st Update fn.; this fn. doesn't just aggregate the feature of the node (h_i) with the incoming messages. Instead, it treats the node $(h_i^{(k)})$ separately and concatenates it with the aggregated message.

it's own feature is included as one of the messages from neighbours

A message to itself

$\hat{m}_i^{(k)} = h_i^{(k)}$

$\hat{m}_i^{(k)}$

$\forall i$

Message ~~Passing~~ Framework:-

$$h_i^{(k+1)} = \sigma\left(W^{(k+1)} \sum_{j \in N_i} \frac{1}{c_{ij}} h_j^{(k)}\right)$$

$\frac{1}{c_{ij}} h_j^{(k)}$: message fn.

$\sum_{j \in N_i}$: aggregation fn.

$\sigma(W^{(k+1)} \ldots)$: Update fn.

V1.6

GraphSage :

Function to compute neighbourhood-aware node embeddings

$$h_i^{(k+1)} = \sigma\left(W^{(k+1)} \text{CONCAT}\left(h_i^{(k)}, \frac{1}{|N_i|} \sum_{j \in N_i} h \right. \right.$$

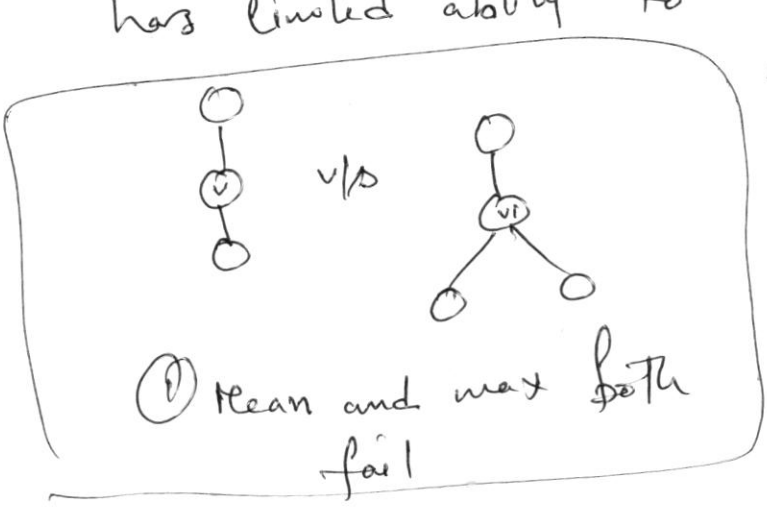
average as aggr fn.

→ Different choices for Message, Aggregate and Update functions are applicable depending on the application/data

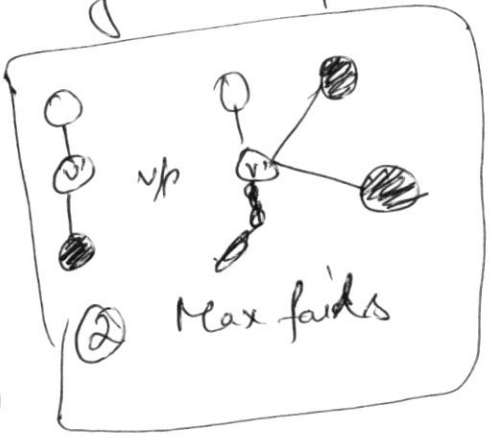
Limitations :-

(1) Repeating the message passing ~~too many~~ multiple times will cause the well-known "over-smoothing" problem, which makes the node embeddings become a self-similar blob.

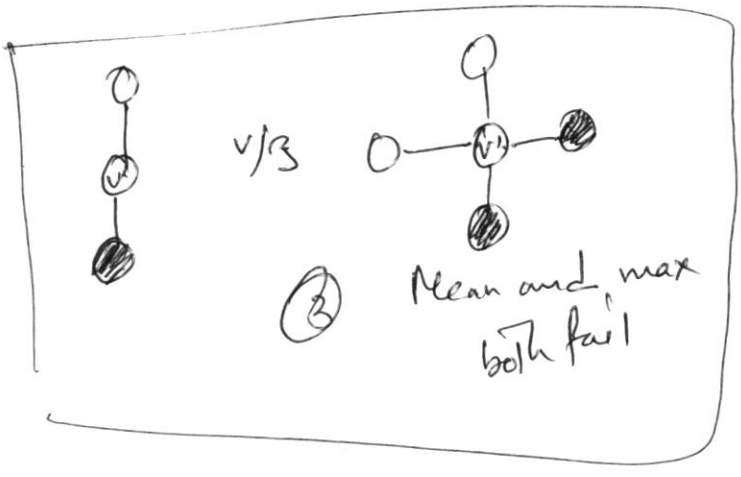
(2) Aggregate fn. applied to pair-wise messages has limited ability to distinguish certain structures



(1) Mean and max both fail



(2) Max fails



(3) Mean and max both fail

V2.1

Prof to GNN

Permutation Symmetries

GOAL: FIND A FUNCTION THAT COMPUTES NEIGHBOURHOOD-AWARE NODE EMBEDDINGS

1) Nodes in a graph have no inherent order

2) Creating an adjacency matrix out of a graph assigns an order to the nodes in the graph.

THE FUNCTION MUST NOT DEPEND ON OUR ARBITRARY ORDERING OF THE INPUTS

3) Traditional ML models depend on the order of the inputs i.e. if you shuffle the columns of your input features, the model won't work.

We want a Model that gives the same answer regardless of the ordering of the nodes, because the ordering is arbitrary.

e.g.:-

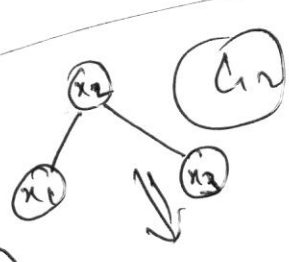
$$X = \begin{bmatrix} & x_1 & x_2 & x_3 \\ x_1 & 0 & 1 & 1 \\ x_2 & 1 & 0 & 0 \\ x_3 & 1 & 0 & 0 \end{bmatrix}$$

(Adjacency Matrix)

$MLP_\theta(X)$

Different ordering

CHECK V2.3



A '1' in adjacency matrix represents a connection b/n the 2 nodes

$$\tilde{X} := \begin{bmatrix} & x_1 & x_2 & x_3 \\ x_1 & 0 & 1 & 0 \\ x_2 & 1 & 0 & 1 \\ x_3 & 0 & 1 & 0 \end{bmatrix}$$

(Adjacency matrix)

$MLP_\theta(\tilde{X})$

$O \neq \tilde{O}$

A test sample (graph) will

The learnt model will give different predictions for different orderings of the nodes in the same graph (or input feature maps)

→ G_1 and G_2 are essentially the same graph in different ordering (arbitrary) of the nodes result in different adjacency matrices.

→ An MLP trained with different ordering of the nodes i.e. diff adjacency matrices will result in different set of learnt weights.

v2.2

Soln: to the above problem

(1) Build these permutation symmetries into your st

(2) Train the MLP on all possible orderings/permu of the nodes of all the training ~~graphs~~ graphs.

↳ Very expensive → (1) So many comb^n s. poss
(2) So many training samples.

This problem of symmetry also exists in CNNs.

↳ e.g. A face in top left corner of an image is same as a face in bottom left corner of the image.

⇒ Can't train the n/w by providing ~~possible posⁿ s.~~ of a feature in all poss'ble posⁿ s. of the image.

⇒ They overcame the problem by convolution i.e. scanning a filter across the image

↳ This scanning property puts translational symmetry into the model itself.

3) Can be useful to get a representation of a node's neighbourhood
A node's neighbourhood is a set of ~~input~~ nodes.

Permutation Invariance : ^ Shuffle The input ordering)

(Order information is ignored) but get the same result every time.

Whatever be the internal order of the elements of the set, the o/p remain the same (inputs)

$$f(Px) = f(x)$$

(Permutation of the order of inputs)

$$X = \begin{matrix} & A & B & C \\ A & 0 & 0 & 1 \\ B & 1 & 0 & 0 \\ C & 0 & 1 & 0 \end{matrix}$$

$$P_{(1,2,3)}\,X = \begin{matrix} & A & C & B \\ B & 1 & 0 & 0 \\ A & 0 & 1 & 0 \\ C & 0 & 0 & 1 \end{matrix}$$

It essentially maps a set of i/ps to an o/p. It doesn't depend on the ordering

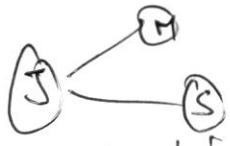
Permutation Equivariance ; - if you shuffle The i/p

(Order information is maintained) You have to shuffle the o/p in the same way to get the same result.

$$\rightarrow f(Px) = Pf(x) \leftarrow$$

RBFBR pg V2.6

Example : Three people : James (J), Sally (S) and Mary (M).



'James' is socially connected to 'Mary' and 'Sally'.

Permutation Invariance

Soln: (1) for permutation Symmetry problem

~~Aggrega~~ Using Aggregation fn.

Like SUM / AVG / MAX / MIX

These aggregation functions are examples of permutation invariant functions

e.g. $\sum (S-J-M) = \sum (J-M-S) = \sum (J-S-M)$

Similarly for AVG / MAX / ~~MIX~~ MIN functions.

v2.4

Soln: ② Permutation Equivariance Using Node application of a fn.

$$f(Px) = Pf(x)$$

Permutation Equivariance maintains the ordering information such that - it doesn't matter whether you permute before or after.

→ One way to do/achieve this is by :→

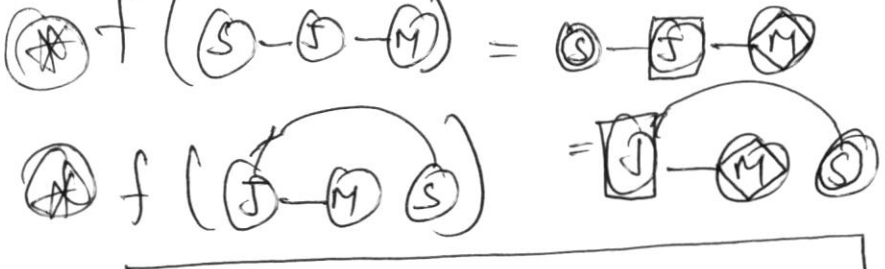
Applying the same fn. to every node individually

$$h_i = \phi(n_i)$$

$$f(Px) = Pf(x)$$

$$\Rightarrow \phi\left(P_{(3,1,2)}X\right) = \begin{bmatrix} -\phi(n_3)- \\ -\phi(n_1)- \\ -\phi(n_2)- \end{bmatrix} = P_{(3,1,2)}$$

Order of i/p node ≡ Order of o/p nodes



i/p order = o/p order

fn. is applied on each input. ~~Whatever be the integral ordering of the elements of the set, the o/p is the~~

~~Graph is a collection of nodes and~~

Graph → Set of Nodes + Edges

→ { Collection of Nodes ≡ Set
  Edges ≡ Adjacency Matrix }

→ { Permuting nodes ≡ Permuting elements of a set
  Permuting edges ≡ " " rows and columns of the adjacency matrix. }

Because a set is not just a graph is an adjacency matrix too

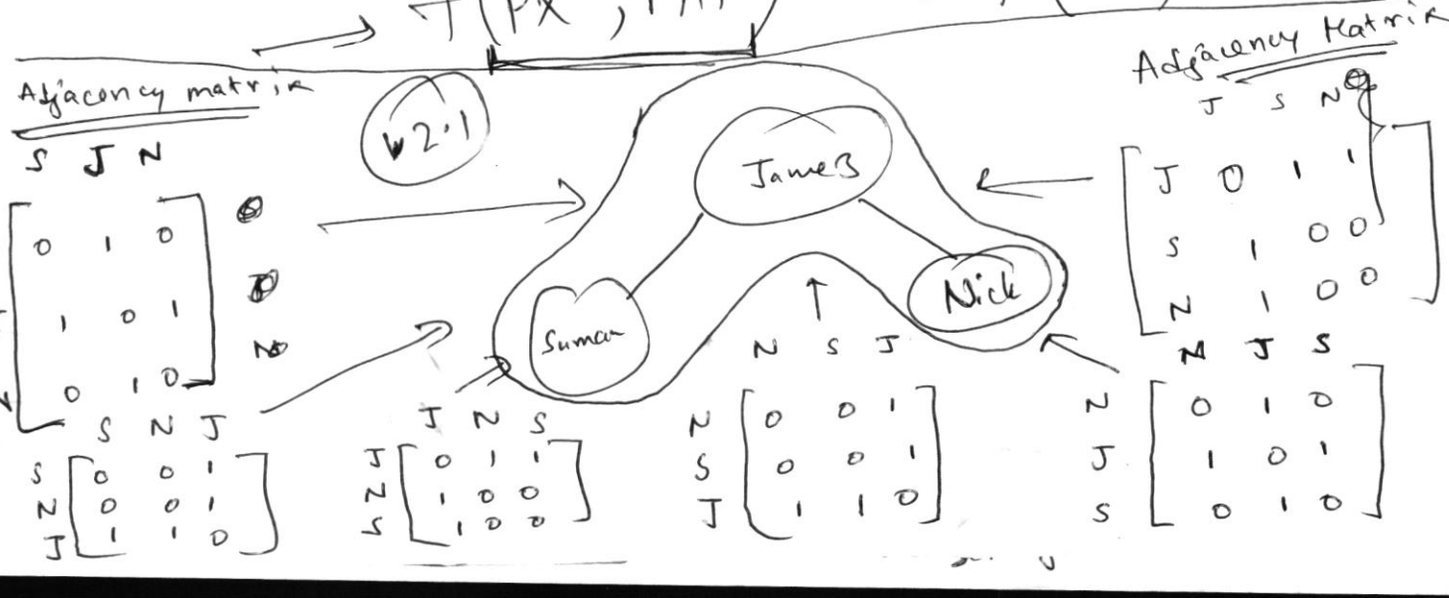
Updated Def. of Permutation Invariance and Permutation Equivariance

Permutation Invariance :

$$f(PX, PAP^T) = f(X, A)$$

Permutation ~~Equivariance~~ Equivariance :

$$f(PX, PAP^T) = Pf(X, A)$$



v2.6

v2.1 Different ~~matrices for~~ adjacency matrices represent
⇒ The same graph.

⇒ Any row and any column in an adjacency matrix can be assigned to any node of the graph.

v2.3 Permutation Invariance :—

$$f\left(\{x_j : j \in N_i\}\right) = h_{N_i}$$

($\{x_j : j \in N_i\}$: set of nodes in a neighbourhood)

← embedding that represents the entire neighbourhood

→ So, i/p to a permutation invariant fn. is a (set).

Permutation Equivariance :—

$$h_i = f(x_i)$$

(h_i: Node-level o/p; x_i: individual node)

In page (v2.4) we had :

$$\phi(X) = \begin{bmatrix} — \phi(x_1) — \\ — \phi(x_2) — \\ — \phi(x_3) — \end{bmatrix}$$

Function ~~to create~~ to compute node embeddings

→ But, a graph is not just a ~~colle~~ set of nodes, but also has edges. So, the function has to take the edges too as i/p.

EXAMPLES: GRAPHSAGE in v1.6

$$\phi(X, A) = \begin{bmatrix} — \phi(x_1, \{x_j : j \in N_1\}) — \\ — \phi(x_2, \{x_j : j \in N_2\}) — \\ — \phi(x_3, \{x_j : j \in N_3\}) — \end{bmatrix}$$

($\{x_j : j \in N_i\}$: set of neighbours; A: Adjacency matrix)

Fn. to compute "neighbourhood" aware node-embeddings

permutation equivariance function

$$\phi(X, A) = \begin{bmatrix} - \phi(x_1, \{x_j : j \in N_1\}) - \\ - \phi(x_2, \{x_j : j \in N_2\}) - \\ \vdots \\ - \phi(x_n, \{x_j : j \in N_n\}) - \end{bmatrix}$$

A ← Adjacency matrix

ϕ ← GNN that ~~compiles neighbourhood-aware node embedding~~ without varying with

→ ~~Message fn~~ Update fn. (Permutation equivariant fn.)

SUMMARY:-

GNNs are functions that produce node-level outputs (taking neighbourhoods into account) that do not depend on our arbitrary ordering (of nodes in adjacency matrix)

i.e. permutation equivariant fn. ϕ

$$\phi(X, A) = \begin{bmatrix} - \phi\left(x_1, \bigoplus_{j \in N_1} x_j\right) - \\ - \phi\left(x_2, \bigoplus_{j \in N_2} x_j\right) - \\ \vdots \\ - \phi\left(x_n, \bigoplus_{j \in N_n} x_j\right) - \end{bmatrix}$$

(ϕ (permutation equivariant))

$\bigoplus$ ← must be permutation invariant. Otherwise, the fn. ϕ won't produce equivariant o/p.

Each ϕ on an input node and its neighbourhood could produce different o/p depending on the ordering of neighbouring node-messages.

Only when the i/p to ϕ is constant/invariant, we can expect an overall equivariant o/p.

(V2-8)

So, we want a fn. (a GNN) :—

① that doesn't depend on the ordering of neighbouring nodes

② But does ~~once~~ maintain the ordering of nodes. Otherwise we don't know which ~~node~~ 'embedding' is computed for which 'node' of the graph.

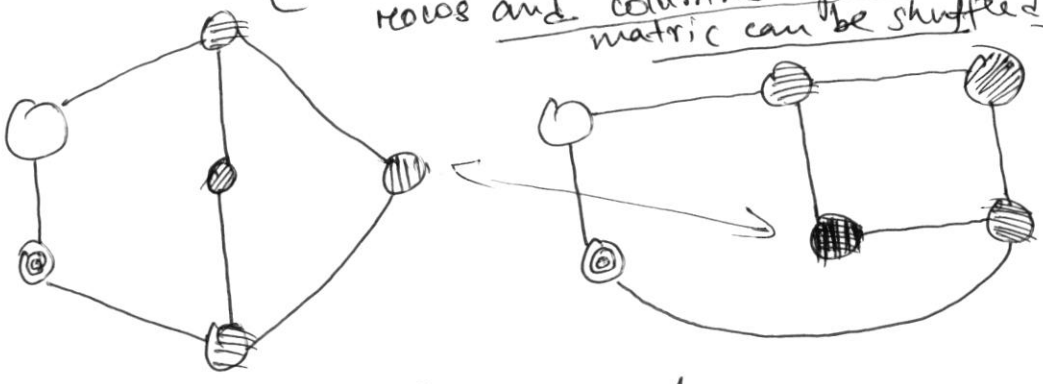
Sol^{n}. → Get node-level embeddings that respect node ordering (Permutation Equivariant) by applying a fn. to each node neighbourhood (Permutation Invariant w.r.t to nodes of neighbourhood i.e. doesn't depend on the order of the nodes in the neighbourhood set).

V3.1

Graph Isomorphism

Two graphs are isomorphic if they are essentially identical but perhaps have been assigned a different ordering (i.e. different adjacency matrices)

→ Graphs don't have an inherent order and any order that we impose is arbitrary (e.g. by putting nodes into an adjacency matrix) [There are many ways in which the rows and columns of an adjacency matrix can be shuffled]



Isomorphic graphs

Does Graph Isomorphism matter for GNNs? If so why?

→ If G1 and G2 are isomorphic graphs, does your GNN give the same o/p?

→ If G1 and G2 are not isomorphic, does your GNN give different o/ps?

G1 (X1, A1) and G2 (X2, A2) would be isomorphic if there's a P such that :-

$$P A_1 P^T = A_2 \quad \text{and} \quad P X_1 = X_2$$

↳ i.e. if you can re-order one to obtain the other

v3.2 How to find if two graphs G_1 and G_2 are isom

Sol^n:① Brute-force: Test every possible permutation of the adjacency matrix of G_1 until you obtain the adjacency matrix of G_2 or you just run out of permutations.

But the no. of permutations scales with the factorial of the no. of nodes

e.g. $\boxed{100! = 9e+157} = 9 \times 10^{157}$

So, Brute-force is not a practical approach.

Sol^n:-② Weisfeiler-Lehman(wl): A family of algorithms for testing whether two graphs are isomorphic.

1-WL steps:-

~~For each node in the graph:~~

Step ① Assign an initial label (say, 0) to each node of the graph

Step ② Assign a new label by mapping each tuple (current label, set of neighbouring labels) to a new label (e.g. 1, 2, 3...)

$$\Rightarrow \boxed{l^{(i+1)}(u) = \mathcal{L}\left((l^{(i)}(u), \{\{l^{(i)}(v) : v \in N(u)\}\}\right)}$$

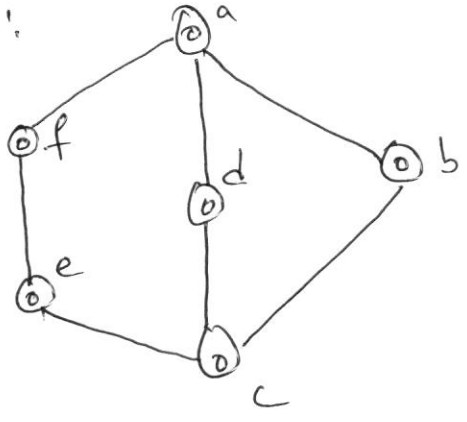
Step ③: Partition the graph into node subsets that share the same colour

Step ④: Repeat step 2-3 until the graph partition stop changing

Perform the

Example:

Step 1:

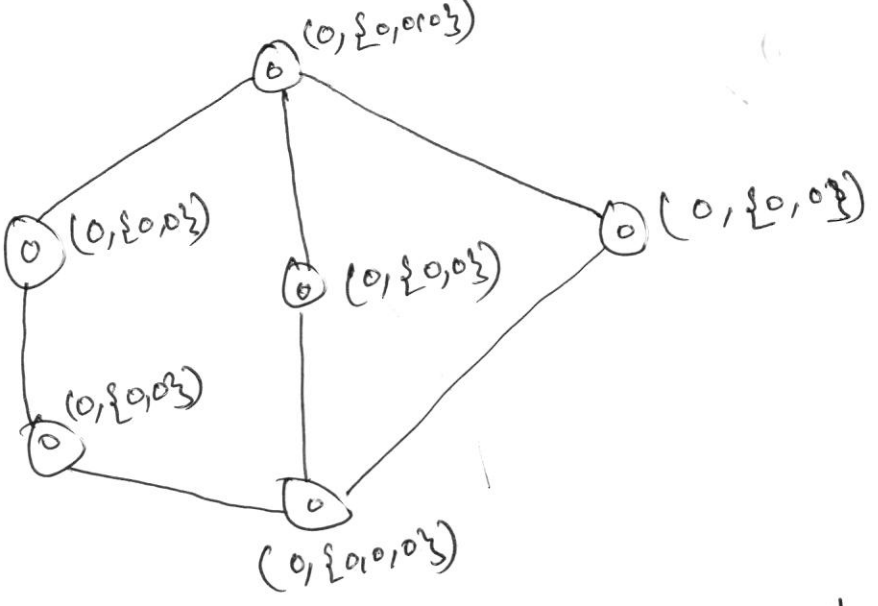


All nodes have same label '0'.

Label '0' is represented by colour blue

Step 2:- Create a tuple for each node

Tuple: (current label, set of neighbouring labels)



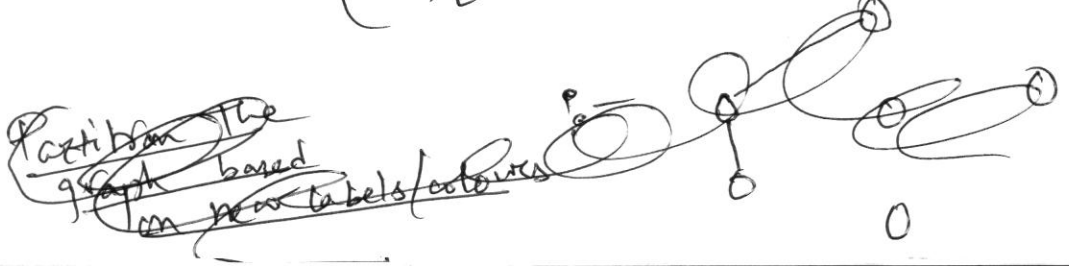
Step 3: Assign a unique colour to each unique tuple

Initial — [0] blue

(0,{0,0}) — [1] yellow

(0,{0,0,0}) — [2] blue

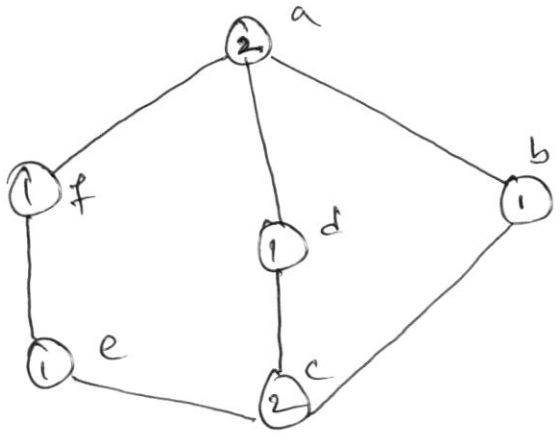
Partition the graph based on new labels/colours



v3.4 Partition

~~Recolour~~ The graph based on the new labels / colours

Step 3:



Partition

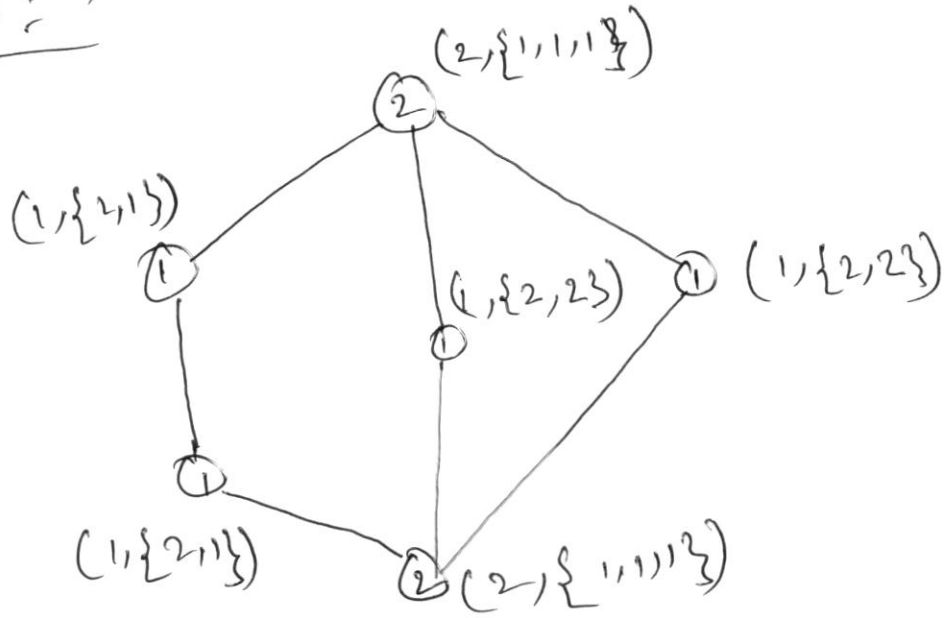
{ }

1 — {b, d, e, f}

2 — {a, c}

Repeat the above steps on the newly labeled graph

Step 4:



Unique Tuples

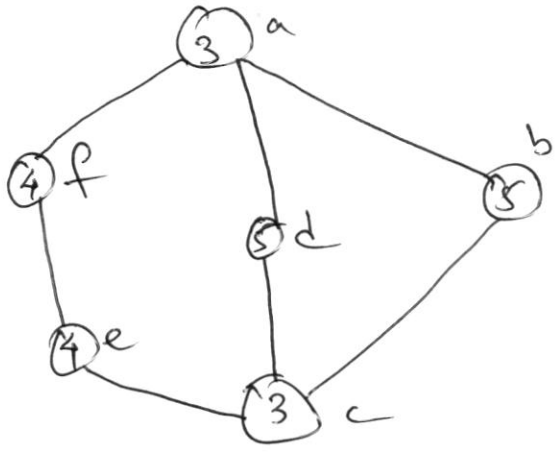
(2,{1,1,1}) — 3 — black

(1,{2,1}) — 4 — Green

(1,{2,2}) — 5 — red

Unique labels, colour

Step 5: Partition the graph based on the new labels.



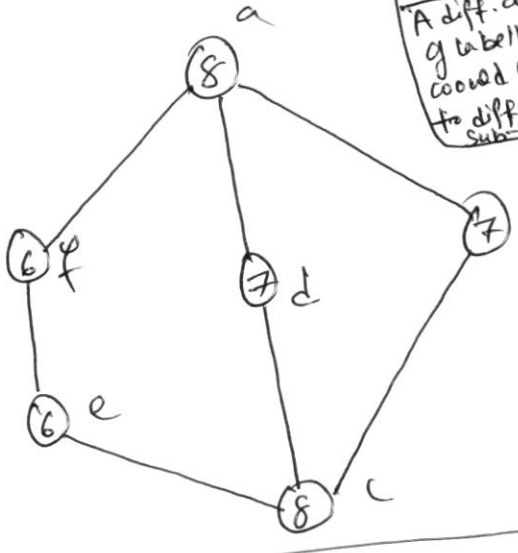
Partitions

3	4	5
{a, c}	{f, e}	{b, d}

NOTE:- The elements of these sub-sets could vary depending on how the nodes are assigned d/labels.

A diff. order of labelling could lead to diff sub-sets.

Step 6/7:



Partitions:

6	7	8
{f, e}	{b, d}	{a, c}

So, you can see the two partitions are identical. The two partitions lead to the same sub-sets of nodes. So, the algorithm has terminated/converged.

If two graphs or more graphs $G_1, G_2, G_3 \ldots$ converge to the same sub-sets of nodes i.e. partitions, it implies they are all isomorphic.

v3.6

If ~~three~~ two or more graphs $G_1, G_2, G_3 \ldots$ converge ~~with the~~ in the 1-WL test, with the same no. of sub-sets - each containing the same no. of nodes — , they are isomorphic.

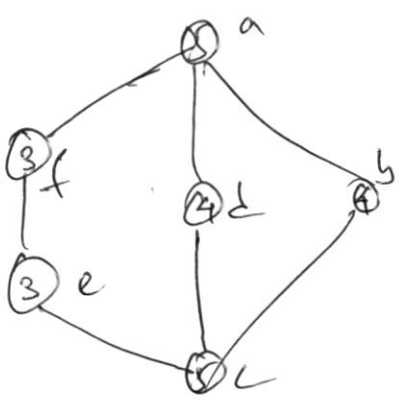
~~Two partitions~~ So, while comparing the partitions of two or ~~dif~~ more different graphs, the labels of the elements (nodes) in their subsets don't matter.

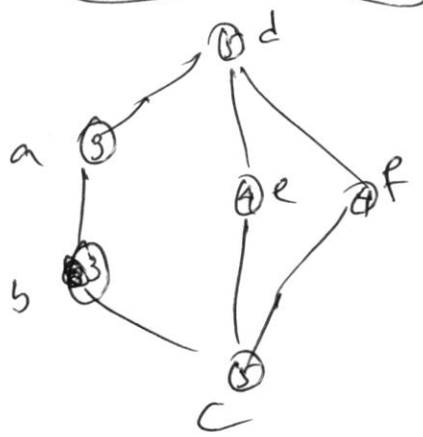
~~If~~ The labels will vary because isomorphic graphs differ in the ordering of the node labels.

~~Only the structural~~ in the partition

→ The exact labels don't matter.

→ Only the structural information - The no. of nodes per sub-set (partition) - matters.





Partitions

3	4	5
{e,f}	{b,d}	{a,c}

3	4	5
{a,b}	{e,f}	{c,d}

Some graphs which are not isomorphic will still pass the 1-WL test.

→ Failing the 1-WL test guarantees that the two graphs are not isomorphic

→ Passing the 1-WL test doesn't guarantee that the two graphs are isomorphic. They may still be non-isomorphic.

The following eqn. in Step 2 of the 1-WL test:

$$l^{(i+1)}(u) = \mathcal{L}\left(\left(l^{(i)}(u), \{\{ l^{(i)}(v) : v \in N(u) \}\}\right)\right)$$

(Putting new labels to nodes based on their current label and the labels of the neighbouring nodes)

And, the GCN fn. in page v2.7:-

$$\phi(x, A) = \begin{bmatrix} -\phi\left(x_1, \bigoplus_{j \in N_1} x_j\right)- \\ -\phi\left(x_2, \bigoplus_{j \in N_2} x_j\right)- \\ \vdots \\ -\phi\left(x_n, \bigoplus_{j \in N_n} x_j\right)- \end{bmatrix}$$

These two fns. bear similarity in the sense that both:- (1) Are applied on each node of the graph
(2) Take into a/c the neighbouring nodes
(3) Take into a/c the current node

V3 8

1-WL :- $l^{(i+1)}(u) = \phi\left(\left(l^{(i)}(u), \{\{l^{(i)}(v) : v \in N(u)\}\}\right)\right)$

GRAPHSAGE :- $h_i^{(k+1)} = \sigma\left(W^{(k+1)} \cdot \text{CONCAT}\left(h_i^{(k)}, \frac{1}{|N_i|}\sum_{j \in N_i} h_j^{(k)}\right)\right)$

So,

1-WL is effectively doing message passing iteratively

→ 1-WL first assigns a new label to each unique tuple

i.e.

1-WL test assigns a new ~~table~~ label to each kind of (node + neighbourhood) multi-set

BUT, MP-GNN (message passing) can be less discriminative than such "deliberately mapping of every unique multi-set

e.g. Average $(\{1,1,1\}) = =$ Average $(\{1,1\})$. [Average as aggregation fn.]

↳ Different neighbourhoods but same o/p.

But, 1-WL could assign a different label to $\{1,1,1\}$ and $\{1,1\}$.

So, the discriminative power of MP-GNN ≤ Discriminative power of 1-WL

GNNs can at most be as powerful as 1-WL

1-WL provides an upper-bound ~~for~~ to discriminative power of MP-GNN.

⇒ Two graphs maynot be ~~isomorphic~~ isomorphic. But, there can be a measure of similarity b/w the two graphs.

How to measure this similarity?

① Perform the 1-WL test on a ~~each~~ graph

② Create a list of each type of labels encountered along with the no. of nodes having ~~ea~~ those labels until the convergence of the iterations of 1-WL test.

e.g.

Label maps		No. of nodes
~~tin~~ Initial →	0	$\hat{x}_0$ 6
$(0,\{0,0\})$ →	1	$\hat{x}_1$ 4
$(0,\{0,0,0\})$ →	2	$\hat{x}_2$ 2
$(1,\{1,2\})$ →	3	$\hat{x}_3$ 2
$(1,\{2,2\})$ →	4	$\hat{x}_4$ 2
$(2,\{1,1,1\})$ →	5	$\hat{x}_5$ 2

③ Create a vector of ~~these features~~;

$$6\hat{x}_0 + 4\hat{x}_1 + 2\hat{x}_2 + 2\hat{x}_3 + 2\hat{x}_4 + 2\hat{x}_5$$

④ Perform ~~these~~ above steps for the other graph and create its vector.

⑤ Take a dot product of ~~those~~ these 2 vectors. The result is a measure of similarity.

How Powerful are GNNs?

- 1-WL test and GNN are analogous
  - The "message passing" step collects all the neighbours' labels into a multi-set (i.e. a set where the same item can occur more than once)

SUMMARY: In 1-WL test, each unique neighbourhood gets a label and additional iteration considers larger neighbourhoods.

Similar to the message passing iterations in GNN.

WHY

Each successive message passing iteration of GNN produces "neighbourhood-aware" embeddings that consider a larger neighbourhood.

So, the question "How Powerful are GNNs?" can be reduced to the question "Can a GNN map each unique neighbourhood to a different embedding/label?"

REMINDER:- A GNN must produce "neighbourhood-aware" embeddings for each node of the n/w without being dependent on the order of the nodes in the neighbourhood of the current node.

V4·2

G_1 and G_2 are two distinct graphs.

Let's say, (1) GNN 'A' distinguishes ~~If~~ G_1 and G_2. But 1-WL test

⇓

(2) 1-WL produces identical partitions / sub-sets of no

⇓

(3) 1-WL obtained ~~same~~ identically labeled ~~sets~~ graphs for G_1 and G_2.

⇓

(4) GNN 'A' will fundamentally be given these identic label multi-sets and its Aggregate and update fn. will map these to identical o/ps and therefore can't distinguish them too.

So, if 1-WL test fails, GNN will fail to

↳ 1-WL is the upper bound for GNN

When Do GNNs meet this Upper Bound ~~set~~ Set by 1-WL test

Ans:→ When the Aggregate and Update fns are Injective functions.

(Injective fn. → Maps distinct i/ps to distinct o/ps)

Distinct multisets of node features are mapped to distinct embeddings.

GIN (Graph Isomorphism Network)

An Injective Aggregate fn.

1) Do One-Hot-Encoding for each ~~theme~~ unique label in the graph

2) Represent each label as One-hot-encoded vector.

3) ~~Add all~~ Add (Sum) all ~~the~~ the vectors (corresponding to each label) in a multi-~~set~~ set.

12) $L = U \Lambda U^T$, $\Lambda = \text{diag}([\lambda_0, \cdots \lambda_{N-1}])$, U = matrix of Eigen-vectors as col

$\{a, b\}$ → |1|1|

$\{a, a, b, b\}$ → |2|2|

∴ The above two neighbours can be distinguished by this injective ~~fn~~ aggregator fn.

On the other hand, SUM and AVERAGE fns can't distinguish these two neighbourhoods.

Next, we need an injective Update fn ∅

node's previous representation

$$h_v^{(k)} = \emptyset\left((1+\epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in N_v} h_u^{(k-1)} \right)$$

Why do we need the constant 'ϵ'?

example :→ $\emptyset(h_1, \{h_2\})$ and $\emptyset(h_2, \{h_1\})$

$h_1 + \epsilon h_1 + h_2 \neq h_1 + h_2 + \epsilon h_2$

If there were no ϵ, or, ϵ = 0,

LHS = $h_1 + h_2$

RHS = $h_1 + h_2$

So, ~~both~~ $\emptyset(h_1, \{h_2\})$ and $\emptyset(h_2, \{h_1\})$ would become ~~it~~ indistinguishable.

GIN :- (Graph Isomorphism Network)

$$h_v^{(k)} = MLP^{(k)}\left((1+\epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)} \right)$$

one-hot-encoded

V5.1

The Graph Laplacian

→ Degree matrix of a graph is a diagonal matrix in which diagonal element represents the degree of a node of the graph.

↳ which diagonal element represents which node is arbitrarily chosen

→ Degree of a node is the no. of edges attached to a node

→ In an undirected graph, each loop increases the degree of a vertex by 2.

→ In a directed graph, the term 'degree' may refer either to 'indegree' (no. of incoming edges at each vertex) or 'outdegree' (no. of outgoing edges at each vertex).

Degree of i^{th} node $= d(i,i) = \sum_j A_{ij}$ = Sum of all moves along i^{th} column (node) of the Adjacency matrix.

So, $d_j = \sum_j A_{ij}$

→ Laplacian (L) $= D - A$, where $D \rightarrow$ Degree matrix, $A \rightarrow$ Adjacency matrix } of same graph

$$L_{ij} = \begin{cases} d_i & \text{if } i = j \\ -1 & \text{if } (i,j) \in \text{edge set} \\ 0 & \text{otherwise} \end{cases} = \delta_{ij} d_i - A_{ij}$$

[if $(i=j)$, $\delta_{ij} = 1$ and $A_{ij} = 0$

if $(i \neq j)$, $\delta_{ij} = 0$, $A_{ij} = 1$ if connected, otherwise 0]

→ The diagonal elements of an adjacency matrix are zeros. Rest of the elements can be 1s or 0s depending on whether there is an edge b/w the nodes represented by the corresponding (row, column) pair.

→ Graph Laplacian is a pre-requisite for understanding Graph Convolutions

$$(Lx)_i = \sum_j (\delta_{ij} d_j - A_{ij}) x_j$$

(Laplacian operating on a graph node)

(i^{th} element of this product) $= d_i x_i - \sum_j A_{ij} x_j$ (for $i = j$)

1. Laplacian (L) $L = D - A$
   - $-1 \rightarrow$ where there are connections
   - $d_i \rightarrow$ for diagonal elements
   - $0 \rightarrow$ else

2. Adjacency matrix (A)
   '1' for connections
   '0' for others

3. $D \rightarrow$ degree for diagonal elements

12) $L = U \Lambda U^T$, $\Lambda = \text{diag}([\lambda_0, \ldots \lambda_{N-1}])$, U = matrix of eigen-vectors as column

V5.2

Degree of i^{th} node of a graph = Sum of all the elements in the i^{th} column of the adjacency matrix (A) of the graph.

$$\Rightarrow \boxed{d_j = \sum_j A_{ij}}$$

i^{th} node/element

$L_x \rightarrow$ Laplacian operator applied on all nodes of a graph

$$(Lx)_i = \sum_j (\delta_{ij} d_i - A_{ij}) x_j$$

why $\sum_j$? where did $\sum$ come from L_2.

$\sum_j'$ means across all 'j's for comp

$$= d_i x_i - \sum_j A_{ij} x_j$$

$\sum A$.

$$= \sum_j A_{ij} (x_i - x_j)$$

Difference b/n the embeddings of the node you care about (i) and those of all its neighbours

$$= \sum_{(i,j) \in \mathcal{E}} (x_i - x_j)$$

$\mathcal{E} \leftarrow$ Set of all edges in the graph

$A_{ij} \rightarrow$ zeros out all the elements except the connections/edges to nodes (j)

$\sum_j'$ over all nodes can be replaced by $\sum$ over just the connections

Difference b/n a node's value and its neighbour's values

$$(Lx)_i = \sum_i (\delta_{ij} d_i - A_{ij}) x_i$$

$$= d_i x_i$$

'L' is discrete equivalent of '∇'.

In physics, Laplacean operator ('∇') is "average diff b/n a point and a small sphere around that po

In discrete setting, operat on graph nodes, $\int_0^\infty$ is replaced by

V5.3

$$(Lx)_i = \sum_j A_{ij} (x_i - x_j)$$ ← A_{ij} zeros everything other than the connected ones

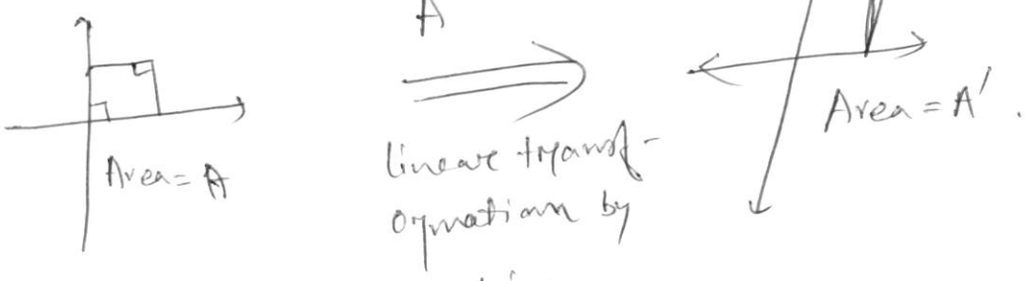
So, $\sum_j A_{ij} (x_i - x_j) = \sum_{(i,j) \in \mathcal{E}} (x_i - x_j)$

$$= \sum_{(i,j) \in \mathcal{E}} (x_i - x_j)$$ ← Sum of the Differences b/w the embeddings of the node we care about - (ith node) and all the embeddings of all its neighbouring nodes ~~(jth node)~~ (denoted by x_j).

~~Basic Line~~

Refresher on Basic Linear Algebra

Determinant →



Determinant = Scaling factor of a chunk of area by applying a linear transformation matrix 'A'.

→ Determinant is a measure of the transformation brought about by applying a $X \times m^n$ matrix 'A'.

12) $L = U \Lambda U^T$, $\Lambda = \text{diag}([\lambda_0, \ldots \lambda_{N-1}])$, U = matrix of Eigen-vectors as columns

V5.4

A determinant of 0 means the area/space was collapsed to a point/line (a reduction in dimensionality) by applying the transformation matrix.

$\Rightarrow$ In such a case, there is ~~not~~ no inverse transformation.

A negative determinant ~~means~~ means the orientation of space is flipped (RHS $\rightarrow$ RHS, Right hand ~~thumb rule~~ co-ordinate system $\rightarrow$ Left hand ~~thumb rule~~ co-ordinate system)

~~$Ax = \lambda x$~~

Eigen Vectors tell you the axes ~~of~~ along which there is just stretching and no rotation $\Rightarrow$ Vectors that are ~~or~~ mapped exactly back on themselves, but perhaps shorter or longer.

$Ax = \lambda x$

$\lambda \rightarrow$ Eigen Value

$x \rightarrow$ Eigen Vector

More the eigen value, greater the stretching along eigen vector

More the squishing/squeezing along eigen-vector, ~~less~~ smaller the eigen-value

$\rightarrow$ The Eigen-value of an eigen-vector tells how much the eigen-vector is ~~be~~ scaled

λ and $||$ (determinant) are analogous

$\lambda \rightarrow$ amount of scaling of eigen Vector

$|| \rightarrow$ " " " " a space/area

Product of all eigen values of a matrix 'A' $= ||$ = Overall amt. of scaling

v5.5

$$Ax = \lambda x$$

A Transformation matrix 'A' is known.

$x \rightarrow$ Eigen vector

$\lambda \rightarrow$ Eigen Value

find x and λ.

If a vector 'x' is such that $Ax = \lambda x$, then 'x' is an Eigen vector if λ = constant

$$Ax = \lambda x$$

$$\Rightarrow (A - \lambda I)x = 0 \quad -(1)$$

$(A - \lambda I)$ — Transformation matrix

$$\Rightarrow x = 0 \quad \text{or} \quad A - \lambda I = 0 \checkmark$$

$$\Rightarrow |A - \lambda I| = 0$$

$\downarrow$

Find λ.

Plug-in λ in (1) to get 'x'.

Every row of a Laplacian sums to 'zero'.

Proof:- $(L1)_i = \sum_j L_{ij}$ ← '$\sum_j$' for constant 'i' occurs upon multiplying 'L' by a vector of $1 = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$

across all 'j's for constant 'i'

$$= \sum_j (\delta_{ij} d_i - A_{ij})$$

$$= \left(\delta_{ij} d_i - A_{ij}\right)_{i=j} + \sum_{j \neq i} (\delta_{ij} d_i - A_{ij})$$

$$= \left(1 \cdot d_i - 0\right)_{i=j} + \sum_{j \neq i} (0 \cdot d_i - A_{ij})$$

$$= d_i - \sum_{j \neq i} A_{ij}$$

$$= d_i - d_i = 0$$

$\Rightarrow$ '1' is an eigenvector of L with eigen value = 0

'L' is a transformation matrix here

'L' is a vector of 1's

No. of 1's depends on no. of nodes in graph

$|L| = 0$

L has no inverse

$\therefore (L1)_i = 0$

$\Rightarrow \lambda I = 0 \Rightarrow \lambda = 0$ ← Eigen Value

So, 1 is an eigen-vector of L with eigen-value = 0.

12) $L = U \Lambda U^{-1}$, $\Lambda = \text{diag}([\lambda_0, \cdots \lambda_{N-1}])$, U = matrix of Eigen-vectors as column

v5.6

Real Symmetric Matrix :-

1) Real valued elements
2) ith row = ith column

Symmetric matrix

'L' is a real-symmetric matrix

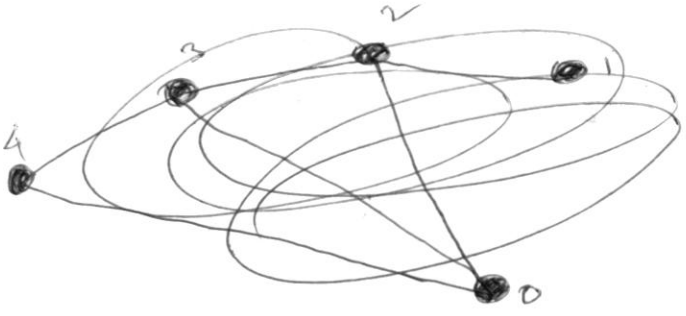
∴ Matrix is equal to its transpose

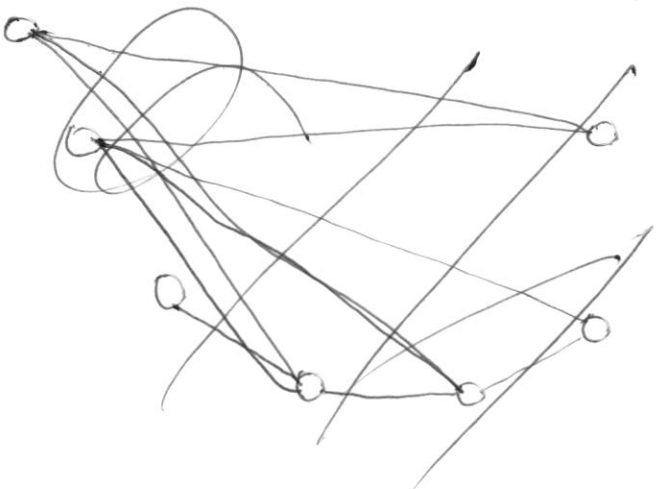
A Real Symmetric Matrix has real eigen-values and orthogonal eigen vectors.

All eigen-values of 'L' are non-negative

→ Then, how do we have negative determinants
⇒ |L| can't be < 0.

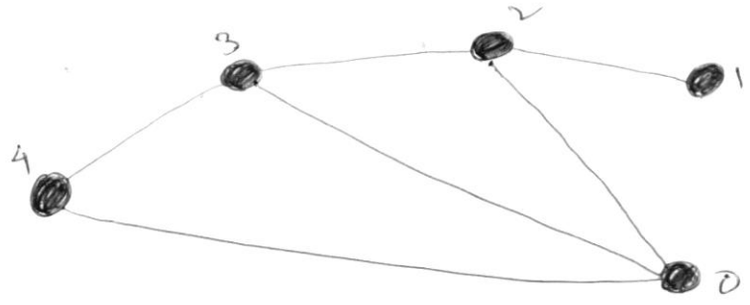
'L' (Laplacian) is a Real Symmetric Matrix



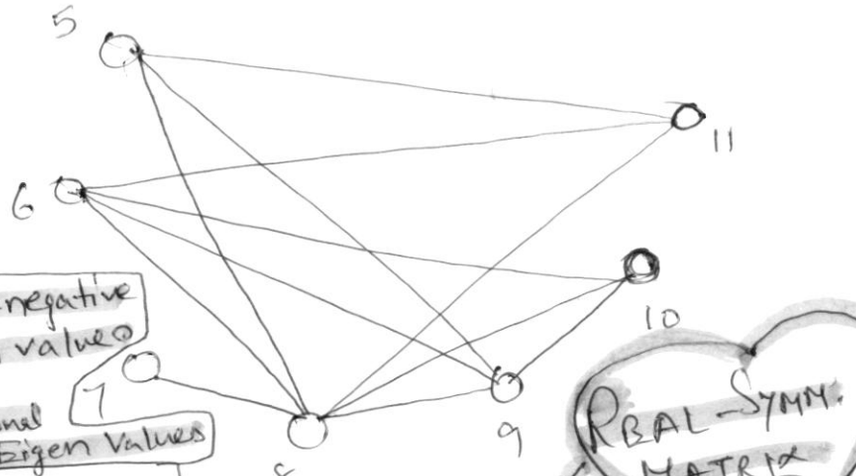


v5.7

Graph Partitioning



A graph with 2 components.



$d_0 - 3$	$d_9 - 4$
$d_1 - 1$	$d_{10} - 3$
$d_2 - 3$	$d_{11} - 3$
$d_3 - 3$	Check in the following laplacian matrix
$d_4 - 2$	$\sum row = 0$
$d_5 - 3$	
$d_6 - 4$	
$d_7 - 1$	
$d_8 - 6$	

Non-negative Eigen values

Orthogonal Eigen Values

REAL – Values

SYMM. – i^{th} row = i^{th} column

REAL-SYMM. MATRIX V5.6

Laplacian of the full graph

	0	1	2	3	4	5	6	7	8	9	10	11
0	d_0	0	-1	-1	-1	0	0	0	0	0	0	0
1	0	d_1	-1	0	0	0	0	0	0	0	0	0
2	-1	-1	d_2	-1	0	0	0	0	0	0	0	0
3	-1	0	-1	d_3	-1	0	0	0	0	0	0	0
4	-1	0	0	-1	d_4	0	0	0	0	0	0	0
5	0	0	0	0	0	d_5						
6	0	0	0	0	0		d_6					
7	0	0	0	0	0			d_7				
8	0	0	0	0	0				d_8			
9	0	0	0	0	0					d_9		
10	0	0	0	0	0						d_{10}	
11	0	0	0	0	0							d_{11}

L1 The upper left half of the 'L' is the 'L' of the upper component of graph

The lower left half of the 'L' is fully black due to lack of any connection b/n the two components of the graph

The upper right half of the 'L' is fully black (or zero) due to lack of any connection b/n the two components of the graph

L2 The lower right half of the 'L' matrix is the 'L' of the lower component of the graph.

Laplacian:-

1) -1 for edge connections
2) degree(d_i) for diagonal elements
3) 0 for others.

12) $L = U \Lambda U^{T}$, $\Lambda = diag([\lambda_0, \cdots \lambda_{N-1}])$, U = matrix of Eigen ... as column

5.8

GRAPH PARTITIONING

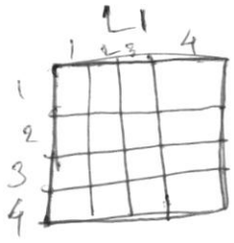
~~We In the previ~~ L matrix of ~~the~~ previous ~~Page~~),

~~we can treat e~~

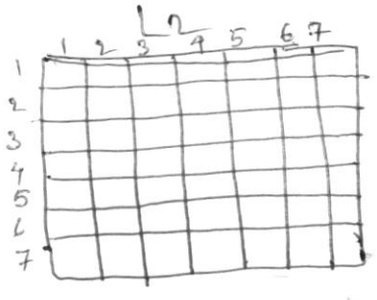
In the graph of the previous page, we can treat each of the two components as separate entities (graph) and create separate Laplacians for them. We can call them L_1 and L_2. L_1 forms the upper left half of the L matrix in the previous page. L_2 forms its lower right half.

No. of eigen-~~values~~ vectors (of a Laplacian) with eigen-value = 0 is the no. of disconnected components in a graph

If you don't know how many disconnected components there are in a graph, take the laplacian of the entire graph and figure out how many Eigen vectors have Eigen values = 0 and the value of those Eigen vectors will tell you about the nodes of each of the disconnected components.



$L_1 1 = \lambda_1 1$ where $\lambda_1 = 0$, Here $1 = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ a vecto of 4 1s



$L_2 1 = \lambda_2 1$ where $\lambda_2 = 0$

Here $1 = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$ A vector of 7 1s

VS.9

GRAPH PARTITIONING

① $X_{L1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$ (4 1s) and $X_{L2} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}$ (7 1s) were Eigen-vectors for L_1 and L_2 respectively.

② Eigen vectors for the whole Laplacian 'L' can be derived from the above two Eigen-vectors.

⇓

By padding the above vectors of 1s with 0s. No. of 0s could be the remaining no. of nodes. For example, the 1st Eigen-vector for L_1 has 4 1s while the total no. of nodes in the graph is 11. So, we will add 7 (=11−4) 0s to X_{L1}. Similarly, to get eigen-vector for L from the eigen-vector of L_2, we would add 4 (=11−7) 0s to it.

② $X_{L1} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \rightarrow X_L = \begin{bmatrix} 1 \\ \vdots \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$ 11 elements

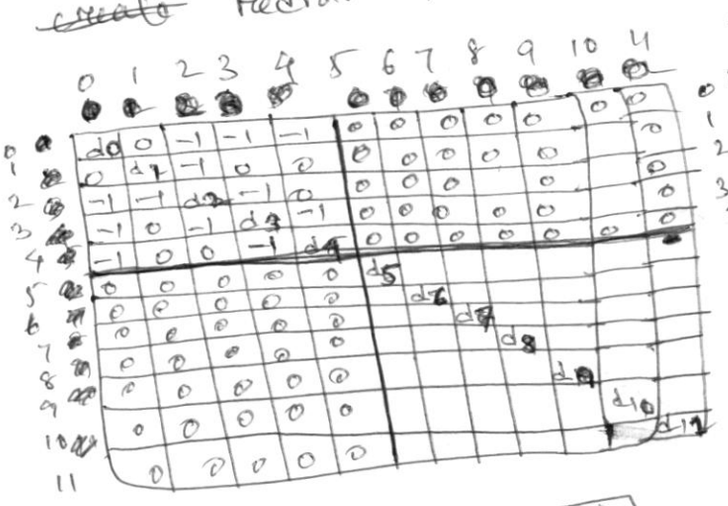
$X_{L2} = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \rightarrow X_L = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$ 11 elements

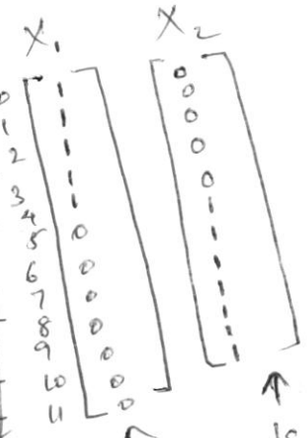
GRAPH PARTITIONING

In So, we concatenate 0s at the end of X_{L1} and at the beginning of X_{L2} to obtain the Eigen vectors of L.

⇓

This can be generalised as :→ To create Eigen vectors for the Laplacian 'L' of a graph, we put 1s for nodes of one component and 0s for nodes of another component of the graph. This way we create multiple eigen-vectors - each one corresponding to nodes of one component and 0s corresponding to nodes of another component. To illustrate, we redraw the L matrix for the previous graph





$LX_1 = \lambda_1 X_1 = 0$ ($\because L_1 1_1 = 0$)

$LX_2 = \lambda_2 X_2 = 0$ ($\because L_2 1_2 = 0$)

Eigen vectors created this way have Eigen-Value 0

Eigen Vector created from the vector '1' of Laplacians of component graphs

VS-11

GRAPH PARTITIONING.

Even if the ordering of the nodes of the graph in VS-7 is shuffled, the L will be modified.

⇓

To create Eigen vectors with Eigen values = 0, just put 1s for nodes (as per new ordering) of one component of graph and 0s for the other components for the reshuffled graph.

For example, the new Eigen vectors will look like:-

$$x_1 = \frac{1}{\sqrt{7}} \begin{bmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \qquad x_2 = \frac{1}{\sqrt{5}} \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$\lambda_0 = \lambda_1 = 0$

To figure out the disconnected components of a graph, i.e. nodes of the

to figure out the partitions of a graph, :-

1. Find the graph Laplacian by ordering nodes
2. Find out all the Eigen vectors of the graph
3. Find out the set of Eigen vectors with eigen values of 0
4. The 1s in each of these eigen vectors correspond to each disconnected component. The nodes of a component.

5.12

Positive Semi-Definite

$$x^T L x = \sum_{ij} L_{ij} x_i x_j$$

Here, x_i and x_j represent diff. values of variable x

And not, different dimensions

Laplacian matrix

arbitrary vector

across all 'i's and 'j's.

$$= \sum_{ij} (\delta_{ij} d_i - A_{ij}) x_i x_j$$

$$= \frac{1}{2} \sum_i d_i x_i^2 + \frac{1}{2} \sum_j d_j x_j^2 - \sum_{ij} A_{ij} x_i x_j$$

$$\left(\sum_j A_{ij} = d_i \right)$$

$$= \frac{1}{2} \sum_{ij} \left[A_{ij} x_i^2 - 2 A_{ij} x_i x_j + A_{ij} x_j^2 \right]$$

$$= \frac{1}{2} \sum_{ij} A_{ij} (x_i - x_j)^2 \geq 0$$

always ≥ 0

1/0

$\therefore x^T L x \geq 0$

$x^T L x \geq 0$

Application

Application of $x^T L x \geq 0$:

CLUSTERING NODES OF A GRAPH

Let's say, we want to cluster a graph into two groups based on similarity.

Note: - graph clustering is not graph partitioning

(grouping nodes based on similarities and not based on edges)

(dividing a graph into components based on connections)

How can $x^T L x$ be useful in graph Clustering??

→ In $x^T L x$, let 'x' be a vector that is (+1) if part of cluster A and (-1) if part of cluster B ~~and 0 if x_i and x_j are in same cluster and 2 if they are in different clusters~~

v5.13

x_i	x_j	$x^T L x$	
—	—		
+1	−1	2	→ Diff. clusters
+1	+1	0	→ Same cluster
−1	+1	2	→ Diff. clusters
−1	−1	0	→ Same cluster.

If we find the 'x' that minimizes $x^T L x$, we will find cluster assignments that minimize cross cluster edges

x_i	x_j	A_{ij}	$(x^T L x)_{ij}$
—	—		2
+1	−1	1	
+1	+1	1	0
−1	+1	1	2
−1	−1	1	0
+1	−1	0	0
+1	+1	0	0
−1	+1	0	0
−1	−1	0	0

Two nodes in the same cluster if connected

Or No connection b/w nodes

How To Cluster Nodes

(1) Find x that minimizes $x^T L x$.

'x' will contain the mapping of nodes to clusters.

Max^m. value of $(x^T L x)_{ij} = 2$

Min^m. value of $(x^T L x)_{ij} = 0$

if x_i and x_j are real, continuous values, $(x_i - x_j)^2$ represents the distance b/w them

$x^T L x$ will be minimized if 2 nodes that are connected lie in the same cluster or if there is no connection at all.

→ Instead of +1's and −1's, the values in x can be real, continuous values — embeddings of nodes. Then, minimizing $x^T L x$ could be minimizing the distance b/w connected nodes. If there is no connection b/w x_i and x_j, we don't care about the distance as a constraint

(V5.14) ~~We need additional constraints because~~

A constant vector x $(c, c, \ldots c)$ is an eigen vector (of L) with eigen value 0, thereby, making $x^T L x = 0$ (minimum value)

∴ A constant vector $(c, c, \ldots c)$ minimizes $x^T L x$ to 0

(∵ its an eigen-vector (of L) with eigen value 0
So, $Lx = 0 \cdot x = 0$)

To avoid getting 'x' that is an all-equal-constant vector, we need to put additional constraints.

Additional Constraints:—

1) Centre of Mass about the origin: $\sum_i x_i = 0 \Rightarrow x \cdot 1 = 0$
$\Rightarrow x \perp 1$ (vector of 1's)

(could not get this point clearly)

Corner case: All values in 'x' being 0 is also a trivial soln. inspite of this constraint.

THIS PART IS NOT IMPORTANT. CAN SKIP IT.

2) ~~$x^* I^* x$~~ $x^T * x = 1 \Rightarrow$ 'x' should be normalizable to 1

It's will avoid the corner case in which all values of 'x' are 0. i.e. it avoids all-0 vector.

~~Rayleigh~~ Rayleigh–Ritz Theorem:—

For a graph (Laplacian L), eigen vector with smallest eigen-value minimizes $x^T L x$ the most.

Smaller the eigen-value, more the compression along eigen vector. And, we count the max^m. compression by L.

v5.15

But, the ~~smallest eigen~~ eigen vector corresponding to the ~~smallest~~ eigen ~~vector~~ value '0' is an ~~all-equi-const vector~~ an all-equi-constant (1) vector ← We are trying to avoid this case (v5.14)

$\lambda = 0 \Rightarrow x = 1$

Laplacian (L) Properties

1) Real-Valued elements
2) Symmetric: i^{th} row = i^{th} column $\Rightarrow L = L^T$
3) Non-negative Eigen values (2)
4) Orthogonal Eigen vectors (x)
5) ~~$\lambda = 0$~~ Sum of ~~xx~~ elements of a row = 0

$\Rightarrow L1 = 0.1$

$\therefore$ 1 is an Eigen vector of L with $\lambda = 0$

6) $x^T L x \geq 0$ 7) Multiplying 'L' by the embeddings ~~of~~ of a node is eqv. to

So, we use the ~~eigh~~ Eigen vector with the 2nd smallest Eigen-value. This soln. meets the 2 additional constraints (1) $x \perp 1$ and ~~normalizable~~

(2) Normalizable to 1 i.e. not an all-zero constant vector.

This Eigen vector is called "Fiedler Vector"

For example, the following Fiedler vector can be converted to cluster assignment as follows,

$$x = \begin{bmatrix} -0.5 \\ 0.2 \\ -0.1 \\ 0.3 \end{bmatrix} \rightarrow \begin{bmatrix} -1 \\ +1 \\ -1 \\ +1 \end{bmatrix}$$

Embeddings > 0 are assigned $+1$ cluster
" < 0 " " -1 "

How to cluster nodes of a graph?

1) Construct Laplacian of a graph.
2) Get all its Eigen vectors and Eigen values
3) Get the Eigen vector(x) corresponding to 2nd smallest Eigen value(x) → minimizes $x^T L x$ s.t. $(x, 0)$
4) ~~Use this vector to cluster~~
5) Use the elements of this vector to cluster the corresponding nodes of the graph.

→ So, the 2nd smallest Eigen-value's Eigen Vector is the x in $x^T L x$ and its elements act as the embeddings of the corresponding nodes

** → { ith element of the Fiedler vector is the embedding of the ith node of the graph.

→ k-dimensional embedding for a node can be constructed from the corresponding element of the 1st k (and smallest to the $(k+1)^{th}$ smallest eigen-value's ~~eigen vector~~) eigen vectors.

V5.17

→ Just ~~like~~ as the 2nd smallest ~~Eigen vector~~ eigen-va
Eigen-vectors ~~can~~ can be used to assign clusters
to the nodes of ~~the~~ a graph, ~~larger~~ successively
larger eigen-valued Eigen vectors can also be used
to assign clusters to the nodes of the graph.

→ The cluster assigned by using the 2nd smallest λ is based
on coarse properties

→ ~~As we go on using larger λ for,~~

→ ~~(clustering using larger λ is o~~

→ As we ~~go on~~ go on to use larger and larger
λ, the clustering is based on ~~more~~ finer and
finer properties.

→ As we saw that the elements of an Eigen vector
can be ~~used~~ assigned as embeddings to the corresponding
~~eleme~~ nodes of the graph.

→ Elements of larger eigen-valued Eigen
vectors represent ~~properties~~ node features
or embeddings that are more
~~fine~~ fine-grained.

~~V5.17~~ V5.18

In $x^T Lx$, if x be a normalized Eigen vector,

then, $x^T Lx = x^T \lambda x = \lambda x^T x = \lambda$ ('x' is normali

An Eigen value of 'L'

$$\begin{bmatrix} 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

$= 1.1 + 1.1 + 1.1$

→ Because a Laplacian always has non-negat Eigen values, $\boxed{x^T Lx = \lambda \geq 0}$ for normalized

~~Eig~~ Since Eigen values of a Laplacian is always non-negative, they can be treated as frequencies because frequencies are always non-negative.

Small eigen value ⇒ low frequency.

Large eigen value ⇒ ~~to~~ high frequency.

→ Elements of larger Eigen valued - Eigen vectors represent node-features that have higher frequency. Larger is the Eigen value, higher is the frequency of node features

~~V5.17~~ V5.19

Properties of Laplacian (L)

1) Real-valued elements

2) Symmetric : ith row = ith column $\Rightarrow L = L^T$

3) Non-negative Eigen values (λ)

4) Orthogonal Eigen vectors $\Rightarrow$ $x_1 \cdot x_2 = 0$

5) Sum of elements of a row = 0 $\Rightarrow L1 = 0.1$
$\Rightarrow$ 1 is an Eigen-ve of L

6) 1 is the Eigen vector with the smallest λ (=0)

7) $x^T L x \geq 0$ (semi-Positive definite)

8) Multiplying 'L' with the features/embeddings of a node has the effect of getting the difference of a node's signal with its neighbours.

(14) Each Eigenvector of L has N elements
N = no. of nodes

9) Eigen vectors of L give us useful embedding of the nodes

(13) No. of Eigen vectors of L = No. of nodes in the graph

10) ~~Smaller~~ Eigen vectors with smaller λ give us coarse properties/features/embeddings of the nodes.

11) Eigen vectors with larger λ give us finer properties/features/embeddings of the node

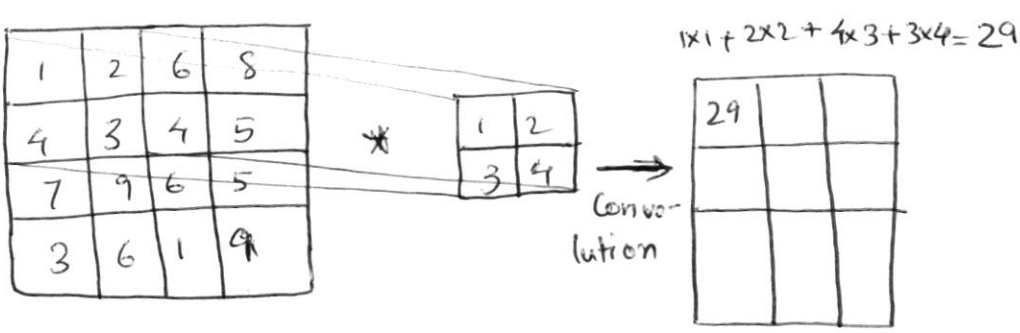
12) $L = U \Lambda U^T$, $\Lambda = \text{diag}([\lambda_0, \cdots \lambda_{N-1}])$, U = matrix of Eigen-vectors of L sta as colum

~~V6.1~~

Fourier Transforms, Wavelets and Spectral Convolu

→ In Convolution Neural Networks (CNNs), the convolution operation is done by scanning the image - grid of pixels with a ~~small~~ filter - a small grid of values. The ~~it~~ filter scans the entirety of the image.

→ A significant amount of works was done ~~in the~~ on how to perform convolution on graphs - like, how to even define a filter ~~if~~ ~~while~~ since the no. of neighbours is differen for different nodes unlike in the case of image



→ Spectral methods ~~use~~ deal with frequencies. Frequencies come from Fourier Transform (FT). ~~Fourier~~ FT decomposes a ~~thing~~ signal into its component frequencies

→ Spectral filters are a ~~of~~ scalar function ~~in~~ that operate on signals and diminish/amplify ~~certain~~ different component frequencies of the signal ~~of dif~~ by different ~~factors~~ (co-efficients)

$f(x)$ —FT→ $\hat{f}(\omega)$ —Apply Spectral Filter→ $\hat{f}^*(\omega)$ —FT^{-1} (Inverse FT)→ $f^*(\omega)$

($f(x)$: signal; $\hat{f}(\omega)$: composed of constituent frequencies)

~~v6.1~~ v6.2

Continuous Fourier Transform

$$FT\{f(x)\} = \hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x)\, e^{-i\omega x}\, dx$$

- $f(x)$: original fn. in space or time domain
- ω: (frequency)
- $\hat{f}(\omega)$: A scalar coefficient for the frequency ω.

FT gives a set a coefficients that say how much of each of ~~these~~ the frequencies exist in a signal.

Dot product of two vectors ⇒ How much one vector lies along another vector

Analogy: Let's imagine one vector to be a component frequency and the other vector to be the signal. If we take a dot product (or inner product) b/n the signal and the frequency, the result will be a scalar (a coefficient) that tells how much of the frequency is present in the signal.

In Dot product, corresponding elements of the 2 vectors are multiplied and ~~summed~~ summed.

Analogy: In continous FT, $f(x)$ is the signal and $e^{-i\omega x}$ is basis fn. corresponding to a frequency ω and represents the frequency and since they are continuous, they have infinite elements.

V6.3

Dot product is the sum of the products of the corresponding elements of 2 vectors X_1 and X_2

$$X_1 \cdot X_2 = \sum_i x_{1i} x_{2i}$$

Analogy :— Discrete domain / Continuous Domain

$X_1 \longrightarrow f(x)$

$X_2 \longrightarrow e^{-i\omega x}$

$i \to \infty$ (infinite elements)

$$\sum_i x_{1i} x_{2i} \longrightarrow \int_{-\infty}^{\infty} f(x)\, e^{-i\omega x}\, dx$$

$$\therefore FT\{f(x)\} = \hat{f}(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(x) e^{-i\omega x}\, dx$$

gives a co-efficient for a frequency ω.

Continuous Inverse FT :—

$$FT^{-1}\{\hat{f}(\omega)\} = f(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\omega)\, e^{i\omega x}\, d\omega$$

adds up all the ~~fre~~ component corresponding to a frequency to obtain the ~~signal~~ original signal.

basis fns : ~~frequencies~~

i.e. $\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} \hat{f}(\omega)\, e^{i\omega x}\, d\omega$, '$f(x)$' is projected onto a new set of functions that ~~are~~ are parameterized by 'ω' : $e^{i\omega x}$ $(= \cos\omega x + i \sin\omega x)$

(V6.4)

$e^{i\omega n}$ is the eigen fn. of $\partial^2/\partial n^2$

$$\therefore \frac{\partial^2}{\partial n^2}(e^{i\omega n}) = -\omega^2 e^{i\omega n}$$

$$(\because \omega, \; Lx = \lambda x)$$

Here, $\partial^2/\partial x^2$ is 1-D Laplacian operator of $e^{i\omega n}$ because when you plug in the fn. in the operator, you get the same fn. scaled by a constant.

(*) The basis fns. of the FT are eigen functions of 1-D laplacian operator

(1) FT → Expansion Expressing a signal in terms of its component frequencies.

(2) Continuous FT → Expressing a function in terms of eigen functions of the laplacian operator

(3) Graph FT → Expressing in terms of eigen vectors of the Graph Laplacian, L

Continuous → Discrete

Operators → $\partial^2/\partial x^2$ is → L

$e^{i\omega x}$ → x

$$\frac{\partial^2}{\partial n^2}(e^{i\omega n}) \rightarrow -\omega^2(e^{i\omega n})$$

$$Lx \rightarrow \lambda x$$

~~16.5~~ v6.5

Graph Fourier Transform

GRAPH-LEVEL → $L x_\ell = \lambda_\ell x_\ell$ ($\ell : 1, 2, 3, \ldots$; $\lambda_\ell : \lambda_1, \lambda_2, \lambda_3, \ldots$; $\lambda_0 = 0$; $\lambda_1 < \lambda_2 < \lambda_3 < \ldots$)

Graph FT: $\hat{f}(\ell) = \langle x_\ell, f \rangle = \sum_{n=1}^{N} x_\ell^*(n) f(n)$

- $\hat{f}(\ell)$ is the scalar co-efficient for the eigen vector x_ℓ of 'ℓ'
- A scalar representation at the graph level
- $\langle , \rangle$ ⇒ inner product
- Summed across all the nodes of the graph
- eigen vector of 'ℓ' of the graph
- features of a node

Inverse Graph FT: $f(n) = \langle \hat{f}, x_\ell \rangle = \sum_{\ell=0}^{N-1} \hat{f}(\ell)\, x_\ell(n)$

- $\hat{f}(\ell)$ tells us how much of the frequency / Eigen vector 'x_ℓ' is present in the entire graph
- A scalar Graph-level
- Summed across all Eigen vectors (x_ℓ) and FTs ($\hat{f}(\ell)$) of the graph
- NODE-LEVEL
- Similar to a signal, a graph can be expressed as the sum of the product of Eigen vectors and corresponding co-efficients ($\hat{f}(\ell)$)

$x_\ell(n)$: - n^{th} element of the Eigen-vector $x_\ell(n)$. It corresponds to the n^{th} node of the Graph.

- No. of elements in it = no. of nodes

$f(n)$: - n-th element of Feature vector of nth node ← This is our signal

$f(n)$ = a scalar

$\hat{f}(\ell)$ = a scalar

But, a node has a vector of features

v6.6

X_l = Eigen vetor : Constituent frequency

nth element of X_l corresponds to feature of nth node

Feature Vector of n-th node

$$\begin{bmatrix} x \\ \\ \\ \\ \\ \\ \end{bmatrix}$$

nth element of each Eigen vector

$$\begin{bmatrix} X_2(n) \\ X_3(n) \\ X_4(n) \\ \vdots \\ X_m(n) \end{bmatrix}$$

$\hat{f}(l)$ tells us how much of the nth element of the lTh Eigen Vector X_l is present in the feature of the nth node

To get the total feature of the nth node, sum up the products of the nth elements of all the Eigen vectors and their corresponding Fourier co-efficients. That is an inner product.

v6.7

Graph Fourier Transform in Matrix Form.

$L = U \Lambda U^T$, $\Lambda = \text{diag}([\lambda_0, \cdots \lambda_{N-1}])$

U = Matrix of Eigen vectors of L, stacked as columns.

x = feature vector

$\hat{x} = U^T x$ performs the same op as $\langle x_\ell, f \rangle = \sum_{n=1}^{N} x_\ell^*(n) f(n)$

$x = U \hat{x}$ performs same op as $\langle \hat{f}, x_\ell \rangle = \sum_{\ell=0}^{N-1} \hat{f}(\ell) x_\ell(n)$

$Lx = \lambda x$

computes just one element of the vector $\hat{x}$

Computes just one element of vector x

In v6.1, $f(n) \xrightarrow{FT} \hat{f}(\omega) \xrightarrow[\text{Spectral Filter}]{\text{Apply}} \hat{f}^*(\omega) \xrightarrow{FT^{-1}} f^*(\omega)$

In case of graphs

$f(n) \xrightarrow[(FT)]{U^T x} \hat{f}(\omega) \xrightarrow[\text{filter } g_\theta(\Lambda)]{\text{Apply}} \hat{f}^*(\omega) \xrightarrow[(FT^{-1})]{U\hat{x}} f^*(n)$

$g_\theta(\Lambda)$ is the ~~filter~~ spectral filter

We apply this filter performing spectral for convolution

~~$x^* = U g_\theta$~~

~~Step 1:~~

~~$x^* = U g_\theta(\Lambda) U^T x$~~ (FT)

This is how convolution is performed on graphs

V6.8

$$x^* = U g_\theta(\Lambda) U^T x$$

$U^T x$: FT

$g_\theta(\Lambda)$: Apply filter

U : FT^{-1}

It's expensive to compute / use all the eigen vectors of L

FT is mapping a fn. onto a set of basis fns.

Issues with Graph Spectral Filtering Problems:→

Problem 1

→ If N = no. of nodes in the graph, no. of Eigen vectors = N, length of each Eigen vector = N. So, U is NxN big. For large graphs, U will be very big and hence computationally expensive to use

→ In calculating x^* by using the above formula the FT co-efficient calculation ($= U^T x$, $= \langle \chi_l, f \rangle = \sum_{n=1}^{N} \chi_l^*(n) f(n)$) involves all the nodes of the graph. When these co-efficients are used to compute FT^{-1} post filter application, all the other nodes of the graph influence the re-construction of the feature vector of a node. So, this process of taking FT, applying Spectral filter and then taking FT^{-1} is highly non-local. We prefer only the immediate neighbourhood nodes to influence the node's feature computation.

Problem 2

These 2 problems get solved by using graph wavelets in FT computation. Graph wavelets can be approximated by low order polynomials of the laplacian and as an added benefit, it avoids the expensive eigen decomposition computation.

V 6.9

~~The problems~~ filters

Graph wavelets can be approximated by Lao-order (Chebyshev polynomials of the Laplacian to :-

① Localize filter — avoiding the influence of distant nodes (Problem 1)

② avoid expensive eigen decomposition computation (Problem 2)

(Eigen decomposition of L : $L = U \Lambda U^T$)

$$y = g_\theta(L)x = g_\theta(U \Lambda U^T)x = U g_\theta(\Lambda) U^T x$$

Instead of doing this, $g_\theta(L) = \sum_k \theta_k T_k(L)$

~~$x^* = g_\theta * g$~~

① localizes the operation to 1-hop neighbourhood

② Simplifies computation by avoiding U

$$x^* = g_\theta(L)x \approx \sum_{k=0}^{K} \theta_k T_k(\tilde{L})x \quad \text{(V6.8)}$$

Filter

$$\approx (\theta_0 + \theta_1 \tilde{L})x \text{ for } K=1$$

θ_0 and θ_1 are filter parameters that need to be learnt. They are learnt via BackPropagation

Final Graph Convolution Operation

$$x^* \approx (\theta_0 I + \theta_1 \tilde{L})x$$

Computationally Simple

I = Identity matrix

∴ $K=1$
∴ It's 1st order approximation

~~So, GCN is~~ So, the source of GCN lies in :-

A 1st order approximation of wavelet filters on graphs ($K=1$)

~~v7~~ (v7.1) Some more approximations :-

① $\tilde{L}$ represents scaled, normalized L

$\Rightarrow L = I - D^{-1/2} A D^{-1/2}$ (D - Degree matrix, A - Adjacency matrix)
(Normalized)

$$\tilde{L} = \frac{2}{\lambda_{max}} L - I \quad (\lambda_{max} \text{ is largest } \lambda)$$

$$= I - D^{-1/2} A D^{-1/2} - I \Big\} \text{ for } \lambda_{max} = 2$$

$$\Rightarrow \tilde{L} = -D^{-1/2} A D^{-1/2}$$

So, $x^* = g_\theta x \approx (\theta_0 I + \theta_1 \tilde{L}) x$

$$\approx (\theta_0 I - \theta_1 D^{-1/2} A D^{-1/2}) x$$

② $\theta_0 = -\theta_1 = \theta$

~~This~~ This allows reduction ~~of~~ in no. of parameters.

Also, it ~~avoids~~ simplifies / beautifies the expression of x^* into 3 distinct factors :-

~~x^*~~

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

parameter — info regarding graph — node feature vector

③ The factor $(I + D^{-1/2} A D^{-1/2})$ ~~was~~ which is a matrix has Eigen values ~~of~~ in $[0, 2]$. So, repeated applications of this matrix will blow up the scale. [WHY WE NEED REPEATED APPLICATIONS 2026]

So, the following "renormalization" trick is performed.

$\tilde{A} = A + I$ (= Adding a self-loop ~~to~~ / edge to each node)

$D_{ii} = \sum_j \tilde{A}_{ij} = D_{ii} + 1$

$(I + D^{-1/2} A D^{-1/2}) \longrightarrow \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}$

So, $x^* = g_\theta * x \approx \theta (\tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2}) x$

Filter — Graph — Feature

v72

GCN Final Form

$$Z = \tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2} X \Theta$$

$$= \hat{A} X \Theta$$

- $\hat{A}$ ↑ Graph
- X ↑ Features
- Θ ↑ Filter

Message Passing Operation

$\hat{A}X$:-

When we multiply a set of features with the adjacency matrix A, what we do essentially is summing up the features of all its connections (nodes).

In $Z = \hat{A}X\Theta$, we are multiplying the features with the normalized adjacency matrix $\hat{A}$ (which includes self-edges to every node). So, $\hat{A}X$ is equivalent to summing up the features of its neighbours as well as its own features (because of the self-edge).

Multi-Layered GCN