anonymous

You are an expert at math, linear algebra, machine learning, and deep learning. Your job is to solve the problem given to you by the user. Use LaTex to format mathematical expressions. Await for the user's response.

Some context to improve responses

GPT

Sure! I'm ready to help with any math, linear algebra, machine learning, or deep-learning problem you have. Please go ahead and describe the problem, and I'll provide a detailed solution (with LaTeX-formatted equations where appropriate).

anonymous

2. Graph Dynamics and GNN Concepts In this problem, we will explore connections between graph dynamics and graph neural networks (GNNs). Diagrams in this question were taken from https://distill.pub/2021/gnn-intro. This blog post is an excellent resource for understanding GNNs and contains interactive diagrams. Some graph neural network methods operate on the full adjacency matrix. Others, such as those discussed in https:// distill.pub/2021/gnn-intro/, at each layer apply the same local operation to each node based on inputs from its neighbors. This problem is designed to: • Show connections between these methods. • Show that for a positive integer k, the matrix Ak has an interesting interpretation. That is, the entry in row i and column j gives the number of walks of length k (i.e., a collection of k edges) leading from vertex i to vertex j. To do this, let's consider a very simple deep linear network, defined as follows: • Its underlying graph has n vertices, with adjacency matrix A. That is, Ai,j = 1 if vertices i and j are connected in the graph and 0 otherwise. • It has n vertices in each layer, corresponding to the n vertices of the underlying graph. • Each vertex has n channels. • The input to each node in the 0-th layer is a one-hot encoding of own identity. That is, the node i in the graph has input (0, · · · , 0, 1 |{z} i-th entry , 0, · · · , 0). • The weight connecting node i in layer k to node j in layer k + 1 is Ai,j . • At each layer, the operation at each node is simply to sum up the weighted sum of its inputs and to output the resulting n-dim vector to the next layer. You can think of these as being depth-wise operations if you'd like.

(a) Write the output of the j-th node at layer k in this network in terms of the matrix A. (Hint: This output is an n-dimensional vector since there are n output channels at each layer.)

(b) Recall that a path from i to j in a graph is a sequence of vertices that starts with i, ends with j, and
every successive vertex in the sequence is connected by an edge in the graph. The length of a path is

length of a path is
the number of edges in it.
Here is some helpful notation:

- V (i) is the set of vertices that are connected to vertex i in the graph.
- Lk(i, j) is the number of distinct paths that go from vertex i to vertex j in the graph where the number of edges traversed in the path is exactly k.
- By convention, there is exactly 1 path of length 0 that starts at each node and ends up at itself.

That is, L0(i, j) = 1i=j .
Prove that the i-th output of node j at layer k in the network above is the count of how many paths there are from i to j of length k.
(Hint: Induct on k.)

(c) The GNN we have worked on so far is essentially linear, since the operations performed at each layer are permutation-invariant locally at each node, and can be viewed as essentially doing the exact same thing at each vertex in the graph based on inputs coming from its neighbors. This is called "aggregation" in the language of graph neural nets.
If we represent the graph as a matrix, with the activatios of the i-th node in the i-th row, what is the update function?
In the case of the computations in previous parts, what is the update function that takes the aggregated inputs from neighbors and results in the output for this node?

(d) The simple GNN described in the previous parts counts paths in the graph. If we were to replace sum aggregation with max aggregation, what is the interpretation of the outputs of node j at layer k?

(e) You are studying how organic molecules break down when heated. For each molecule, you know the element for each atom, which other atoms it is connected to, the length of the bonds, and the type of molecule it is (carbohydrate, protein, etc.) You are trying to predict which bond, if any, will break first if the molecule is heated.
(i) How would you represent this as a graph? (What are the nodes, edges, and global state representations? Is it directed or undirected?)
(ii) How would you use the outputs of the last GNN layer to make the prediction?
(iii) How would you encode the node representation for the input to the GNN?

(f) There are analogs of many ConvNet operations which can be done with GNNs. As Figure 1 illustrates, we can think of pixels as nodes and pixel adjacencies as similar to edges. Graph-level classification tasks, for instance, are analogous to image classification, since both produce a single, global prediction.

Fill out the rest of the table. (Not all rows have a perfect answer. The goal is to think about the role an operation serves in one architecture and whether you could use a technique which serves a similar role in the other architecture.)

CNN GNN
Image classification Graph-level prediction problem
Node-level prediction problem
Color jitter data augmentation (adjusting the color or brightness of an image)
Image flip data augmentation
Channel dropout
Zero padding edges
ResNet skip connections
Blurring an image
Predicting missing values of nodes

(g) If you’re doing a graph-level classification problem, but node values are missing for some of your graph nodes, how would you use this graph for prediction?

(h) Consider the graph neural net architecture shown in Figure 2. It includes representations of nodes (Vn), edges (En), and global state (Un). At each timestep, each node and edge is updated by aggregating neighboring nodes/edges, as well as global state. The global state is the updated by getting information from all nodes and edges. For more details on the architecture setup, see **https://distill.pub/**2021/gnn-intro/#passing-messages-between-parts-of-the-graph.

(i) If we double the number of nodes in a graph which only has node representations, how does this change the number of learned weights in the graph? How does it change the amount of computation used for this graph if the average node degree remains the same? What if the graph if fully connected? (Assume you are not using a global state representation).

(ii) Where in this network are learned weights/parameters?
(iii) The diagram provided shows undirected edges. How would you incorporate directed edges?

9 **GPT**

Below each sub-question I give a concise answer together with the necessary mathematical detail (LaTeX-formatted).

(a) Output of node j at layer k

Let

Fully correct with good explaination

- $\mathbf{A} \in \{0,1\}^{n\times n}$ be the adjacency matrix,
- $\mathbf{h}_i^{(0)} = \mathbf{e}_i \in \mathbb{R}^n$ be the one-hot input of node i (the i-th canonical basis vector),
- $\mathbf{H}^{(k)} \in \mathbb{R}^{n\times n}$ be the matrix whose j-th **row** (or **node-wise vector**) contains the output of node j at layer k.

The weight from node i in layer k to node j in layer $k+1$ is A_{ij}.

Therefore the linear update performed at every layer is simply a matrix multiplication:

$$\boxed{\mathbf{H}^{(k+1)} = \mathbf{A}^\top \mathbf{H}^{(k)}} \qquad (\text{or } \mathbf{H}^{(k+1)} = \mathbf{A}\mathbf{H}^{(k)} \text{ if } \mathbf{A} \text{ is symmetric}).$$

Starting from $\mathbf{H}^{(0)} = \mathbf{I}_n$ (the identity, because each node receives its own one-hot vector), we obtain by induction

$$\boxed{\mathbf{H}^{(k)} = \mathbf{A}^k}.$$

Consequently the **j-th node** at layer k outputs the j-th **column** of $\mathbf{A}^k$:

$$\boxed{\mathbf{h}_j^{(k)} = \left(\mathbf{A}^k\right)_{:j} \in \mathbb{R}^n}$$

i.e. the i-th component of that vector is $(\mathbf{A}^k)_{ij}$.

(b) Interpretation of the components – walks of length k

Define

- $L_k(i,j)$ = number of distinct walks (allowing repeated vertices) that start at vertex i and end at vertex j using exactly k edges,
- $L_0(i,j) = \mathbf{1}_{\{i=j\}}$.

We prove by induction on k that

$$(\mathbf{A}^k)_{ij} = L_k(i,j) \qquad \forall i,j.$$

Base case $k = 0$

$$\mathbf{A}^0 = \mathbf{I}_n \implies (\mathbf{A}^0)_{ij} = \mathbf{1}_{\{i=j\}} = L_0(i,j).$$

Inductive step

Assume $(\mathbf{A}^k)_{pj} = L_k(p,j)$ for all p.

For $k+1$,

$$(\mathbf{A}^{k+1})_{ij} = \left(\mathbf{A}\mathbf{A}^k\right)_{ij} = \sum_{p=1}^{n} A_{ip}\,(\mathbf{A}^k)_{pj}.$$

Because $A_{ip} = 1$ exactly when there is an edge (i,p), the term A_{ip} selects those intermediate vertices p that can be reached from i in **one** step.

Each such p contributes $L_k(p,j)$ walks of length k from p to j.

Appending the first edge (i,p) to each of those walks yields a distinct walk of length $k+1$ from i to j.

Summing over all possible p counts **all** walks of length $k+1$:

$$\sum_p A_{ip}\,L_k(p,j) = L_{k+1}(i,j).$$

Thus $(\mathbf{A}^{k+1})_{ij} = L_{k+1}(i,j)$. ▮

Hence the i-th component of the vector output by node j at layer k is exactly the number of length-k walks from vertex i to vertex j.

(c) Linear "aggregation" update written as a matrix operation

Let

- $\mathbf{H}^{(k)} \in \mathbb{R}^{n\times n}$ be the node-feature matrix at layer k (row i = feature vector of node i),
- $\mathbf{A}$ the adjacency matrix (assume undirected, i.e. $\mathbf{A} = \mathbf{A}^{\top}$).

The **aggregation** performed at each node is the sum of the feature vectors of its neighbours (including possibly itself). In matrix form this is

$$\boxed{\mathbf{H}^{(k+1)} = \mathbf{A}\,\mathbf{H}^{(k)}}.$$

If we wish to keep the node's own previous representation (as is common) we would write a *residual* version

$$\mathbf{H}^{(k+1)} = \mathbf{H}^{(k)} + \mathbf{A}\,\mathbf{H}^{(k)}.$$

The update function that maps the aggregated neighbour information to the new node representation is therefore the linear map "multiply by $\mathbf{A}$" (or, more generally, an MLP that receives $\mathbf{A}\mathbf{H}^{(k)}$ as input).

(d) Replacing sum aggregation by max aggregation

If at each layer a node takes the element-wise maximum over the feature vectors of its neighbours, the i-th component of the output of node j at layer k becomes

$$\left[\mathbf{h}_j^{(k)}\right]_i = \max_{\text{walk } (i\to\cdots\to j) \text{ of length } k} 1.$$

Since the entries are binary (the initial one-hots contain only 0/1), the max is **1** iff there exists **at least one** walk of length k from i to j, and **0** otherwise exists **at least one** walk of length k from i to j, and **0** otherwise.

> **“Interpretation: the max-aggregating GNN computes the *reachability* indicator for walks of exactly k hops, i.e. the Boolean matrix $\mathbf{A}^k$ with entries thresholded to $\{0,1\}$.”**

(e) Predicting the first bond to break in a heated molecule

(i) Graph representation

Element	Choice
Nodes	Atoms. Feature vector may contain a one-hot for element type (C, O, N, ...), atomic number, hybridisation, partial charge, etc.
Edges	Chemical bonds. Edge attributes: bond order (single/double/triple), bond length, bond type (aromatic, hydrogen-bond, etc.).
Global state	Information that is the same for the whole molecule: temperature, pressure, molecule class (carbohydrate, protein, ...).
Directionality	Bonds are physically undirected, so we use an **undirected** graph. If we want to encode polarity we can add a directed edge type (e.g. donor→acceptor) but the basic topology is undirected.

(ii) Using the last GNN layer to make a prediction

1. Run a (possibly multi-hop) GNN to obtain a final node embedding $\mathbf{h}_i^{(K)}$ for every atom.
2. Form an **edge representation** for each bond by concatenating (or otherwise combining) the embeddings of its two incident atoms and the edge attributes:

   $$\mathbf{e}_{uv} = \phi_{\text{edge}}\big([\mathbf{h}_u^{(K)} \,\|\, \mathbf{h}_v^{(K)} \,\|\, \text{bond\_features}_{uv}]\big),$$

   Different solution, but since multiple solutions are possible, I believe this is resonable

   where ϕ_{edge} is a small MLP.
3. Feed all edge representations through a classifier (e.g. a soft-max over all edges) that outputs a probability distribution over bonds.
4. The bond with the highest probability is taken as the predicted first-to-break bond.

Alternatively, a **global read-out** (sum/mean of node embeddings) can be concatenated with the edge-level scores to incorporate molecule-wide context.

(iii) Encoding the node representation (input features)

(iii) Encoding the node representation (input features)

A practical encoding is a concatenation of several one-hot / scalar fields:

$$\mathbf{x}_i = \underset{\text{size } E}{\text{One-hot}(\text{element}_i)} \quad \underset{\text{size } H}{\text{One-hot}(\text{hybridisation}_i)} \quad \underset{1}{\text{Scalar}(\text{partial charge}_i)}$$

The resulting vector is fed directly into the first GNN layer (or first passed through a linear projection to a desired hidden dimension).

(f) Analogy table – CNN ↔ GNN

There are some issues to this answer, and I did try to fix them, but was unable to after a few attempts

CNN operation / task	Corresponding GNN analogue
Image classification	**Graph-level classification** (global read-out such as $\text{readout}(\{\mathbf{h}_i\}_{i=1}^{n}) = \sum_i \mathbf{h}_i$ followed by an MLP)
Node-level prediction	**Node classification / regression** (each node's final embedding is fed to a per-node MLP)
Color-jitter data augmentation	**Feature-noise augmentation** – add Gaussian noise or randomly mask/ perturb node feature dimensions
Image-flip data augmentation	**Graph isomorphism augmentation** – randomly permute node ordering or apply graph-automorphism-preserving transformations (e.g. reflect a planar molecular graph)
Channel dropout	**Dropout on feature channels** – zero-out random dimensions of node feature vectors (or drop whole node feature channels)
Zero-padding edges	**Adding dummy isolated nodes** (or self-loops with zero features) to obtain a fixed-size adjacency matrix for batching
ResNet skip connections	**Residual GNN layers** – $\mathbf{h}^{(k+1)} = \mathbf{h}^{(k)} + \text{AGG}(\mathbf{h}^{(k)})$ (or using $\mathbf{h}^{(k+1)} = \text{MLP}(\mathbf{h}^{(k)} \parallel \text{AGG}(\mathbf{h}^{(k)}))$)
Blurring an image	**Diffusion / low-pass filtering** – repeated aggregation (e.g. k hops) or applying the normalized Laplacian $\mathbf{L} = \mathbf{I} - \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$
Predicting missing values of nodes	**Graph-based imputation** – treat missing node features as learnable variables, run a GNN to propagate information from observed nodes, then read out the filled-in values

---

(g) Missing node values in a graph-level classification task

When some nodes have *missing* feature vectors:

1. **Masking:** Create a binary mask $\mathbf{m}_i \in \{0, 1\}$ indicating whether node i is observed

1. **Masking:** Create a binary mask $\mathbf{m}_i \in \{0, 1\}$ indicating whether node i is observed. During aggregation, multiply each neighbour's contribution by its mask, i.e.

$$\tilde{\mathbf{h}}_j^{(k+1)} = \sum_{i \in \mathcal{N}(j)} m_i \, \mathbf{h}_i^{(k)}.$$

2. **Learned imputation:** Initialise missing node features with a learnable "null'' vector $\mathbf{h}_{\text{null}}$ (or with random noise) and let the GNN propagate information from observed neighbours; the model will implicitly infer plausible values.

3. **Read-out on observed nodes only:** Use a permutation-invariant pooling (sum / mean) that ignores masked nodes:

$$\mathbf{g} = \frac{1}{\sum_i m_i} \sum_i m_i \, \mathbf{h}_i^{(K)}.$$

The pooled representation $\mathbf{g}$ is then fed to the final classifier. Because the GNN aggregates from neighbours, information can still flow through the missing nodes, effectively "filling-in'' them during message passing.

(h) Figure-2 style message-passing architecture

(i) Effect of doubling the number of nodes

Scenario	Number of learned parameters	Computational cost (per forward pass)
Only node representations, shared weights (typical GNN)	**Unchanged –** parameters belong to the *shared* message/update functions (e.g. MLPs) and do **not** depend on the number of nodes.	**Linear** in the number of edges. If average degree stays constant, the number of edges doubles, so the cost roughly doubles: $\mathcal{O}(N \cdot d_{\text{avg}})$.
Fully-connected graph	Still **unchanged** (weights are shared).	Edge count grows as $\frac{N(N-1)}{2}$; cost becomes quadratic: $\mathcal{O}(N^2)$. Doubling N multiplies the cost by roughly four.

Thus, in standard GNNs the parameter count is *independent* of graph size, while the runtime scales with the number of edges (linear for sparse graphs, quadratic for dense graphs).

(ii) Where are the learned weights?

In the architecture of Figure 2 the trainable components are the functions that compute messages and updates:

- **Edge update** ϕ^e: an MLP (or linear layer) that takes the sender node feature, receiver node feature, current edge feature, and possibly the global state, and outputs the new edge feature.
- **Node update** ϕ^v: an MLP that receives the aggregated incoming edge messages (often summed) together with the node's previous feature and the global state.
- **Global (graph) update** ϕ^u: an MLP that aggregates information from all nodes and edges (e.g., by sum/mean) and updates the global vector.

All three ϕ's share parameters across *all* edges, *all* nodes, and *all* graphs (respectively), which gives the permutation-invariance property.

(iii) Handling directed edges

To incorporate directionality one can:

1. **Separate adjacency matrices** for incoming and outgoing edges, $\mathbf{A}^{\text{in}}$ and $\mathbf{A}^{\text{out}}$, and use distinct message functions ϕ^e_{in} and ϕ^e_{out}.

2. **Encode edge direction as a categorical feature** (e.g., "forward" vs "backward") that is fed to the edge-update MLP, allowing the same function to treat the two cases differently.

3. **Treat each directed edge as a distinct edge object** in the message-passing loop: for an ordered pair $(i \to j)$ the message is computed from the source node i to the target node j. If the underlying graph is undirected but you need both directions, simply create two directed edges $(i \to j)$ and $(j \to i)$.

All of the above preserve the overall architecture; the only change is that the aggregation at a node now distinguishes between incoming and outgoing messages (or uses a direction-aware message function).

In summary, the deep linear network described in parts (a)–(d) is precisely the matrix power $\mathbf{A}^k$; its components count walks of length k, and a max-aggregation variant counts *existence* of such walks. Extending these ideas yields concrete designs for molecular-bond-breakage prediction, analogies between CNN and GNN operations, strategies for missing data, and a clear view of where parameters live and how they scale in modern message-passing GNNs.

Overall it does pretty well on
this question, especially since
it is a very long question.

anonymous

You are an expert at math, linear algebra, machine learning, and deep learning. Your job is to solve the problem given to you by the user. Use LaTex to format mathematical expressions. Await for the user's response.

New chat to improve responses

GPT

Sure! I'm ready to help with any mathematics, linear algebra, machine learning, or deep-learning problem you have. Please go ahead and share the details of the problem, and I'll work through it step by step (using LaTeX for any formulas).

anonymous

3. Graph Neural Networks For an undirected graph with no labels on edges, the function that we compute at each layer of a Graph Neural Network must respect certain properties so that the same function (with weight-sharing) can be used at different nodes in the graph. Let's focus on a single particular "layer" ℓ. For a given node i in the graph, let s ℓ−1 i be the self-message (i.e. the state computed at the previous layer for this node) for this node from the preceeding layer, while the preceeding layer messages from the ni neighbors of node i are denoted by mℓ−1 i,j where j ranges from 1 to ni . We will use w with subscripts and superscripts to denote learnable scalar weights. If there's no superscript, the weights are shared across layers. Assume that all dimensions work out.

(a) Tell which of these are valid functions for this node's computation of the next self-message s
ℓ
i
.
For any choices that are not valid, briefly point out why.
Note: we are not asking you to judge whether these are useful or will have well behaved gradients.
Validity means that they respect the invariances and equivariances that we need to be able to deploy as
a GNN on an undirected graph.
(i) s
ℓ
i = w1s
ℓ−1
i + w2
1
ni

Pni
j=1 mℓ−1
i,j
(ii) s
ℓ
i = max(w
ℓ
1
s
ℓ−1
i
, w2mℓ−1
i,1
, w3mℓ−1
i,2
, . . . , wni+1mℓ−1
i,ni
) where the max acts component-wise on
the vectors.
(iii) s
ℓ
i = max(w
ℓ
1
s
ℓ−1
i
, w2mℓ−1
i,1
, w2mℓ−1
i,2
, . . . , w2mℓ−1
i,ni
) where the max acts component-wise on the
vectors.

(b) We are given the following simple graph on which we want to train a GNN. The goal is binary node
classification (i.e. classifying the nodes as belonging to type 1 or 0) and we want to hold back nodes 1
and 4 to evaluate performance at the end while using the rest for training. We decide that the surrogate
loss to be used for training is the average binary cross-entropy loss

loss to be used for training is the average binary cross-entropy loss.

Figure 3: Simple Undirected Graph (5 nodes labeled 1-5; 1 has edges to 2,4,5; 2 has 1,4,5; 3 has 5; 4 has 1,2; 5 has 1,2,4)

nodes	1	2	3	4	5
yi	0	1	1	1	0
yˆi	a	b	c	d	e

Table 1: yi
is the ground truth label, while yˆi
is the predicted probability of node i belonging to class 1 after training.

Table 1 gives you relevant information about the situation.

Compute the training loss at the end of training.

Remember that with n training points, the formula for average binary cross-entropy loss is

1

n

X

x

y(x) log 1
yˆ(x)

- (1 − y(x)) log 1 1 − yˆ(x) ! where the x in the sum ranges over the training points and yˆ(x) is the network's predicted probability that the label for point x is 1.

(c) Suppose we decide to use the following update rule for the internal state of the nodes at layer ℓ.

s
ℓ
i = s
ℓ−1
i + W1
Pni
j=1 tanh
W2mℓ−1
i,j
ni
(1)

where the tanh nonlinearity acts element-wise.

For a given node i in the graph, let s
ℓ−1
i
be the self-message for this node from the preceding layer,
while the preceding layer messages from the ni neighbors of node i are denoted by mℓ−1
i,j where j

ranges from 1 to ni
. We will use W with subscripts and superscripts to denote learnable weights in
matrix form. If there's no superscript, the weights are shared across layers.
(i) Which of the following design patterns does this update rule have?
☐ Residual connection
☐ Batch normalization
(ii) If the dimension of the state s is d-dimensional and W2 has k rows, what are the
dimensions
of the matrix W1?
(iii) If we choose to use the state s
ℓ−1
i
itself as the message mℓ−1 going to all of node i's neighbors,
please write out the update rules corresponding to (1) giving s
ℓ
i
for the graph in Figure 3 for
nodes i = 2 and i = 3 in terms of information from earlier layers. Expand out all sums.

GPT

Below is a complete answer to the three sub-questions. All formulas are written in LaTeX for clarity.

(a) Validity of the proposed update functions

For an **undirected** graph the aggregation over the neighbour set must be **permutation-invariant** (or permutation-equivariant when combined with the self-message). In practice this means that the operation applied to the multiset $\{m_{i,j}^{\ell-1}\}_{j=1}^{n_i}$ cannot depend on the ordering of the neighbours and must use the same parameters for every neighbour.

#	Update rule	Permutation-invariance / weight sharing?	Verdict
(i)	$s_i^\ell = w_1\, s_i^{\ell-1} + \dfrac{w_2}{n_i}\displaystyle\sum_{j=1}^{n_i} m_{i,j}^{\ell-1}$	The neighbour messages are **averaged** (a sum divided by a constant). The sum is invariant to any permutation of the neighbours, and the same scalar weight w_2	**Valid**

		is used for all of them.	
(ii)	$s_i^\ell = \max\Big(w_\ell^{(1)} s_i^{\ell-1},\ w_2 m_{i,1}^{\ell-1},\ w_3 m_{i,2}^{\ell-1}, \ldots, w_{n_i+1} m_{i,n_i}^{\ell-1}\Big)$	Each neighbour receives a *different* scalar weight ($w_2, w_3, \ldots$). The result therefore depends on the *order* in which the neighbours are listed, which violates the required permutation invariance.	**Invalid –** invarianc because neighbou weights a shared.
(iii)	$s_i^\ell = \max\Big(w_\ell^{(1)} s_i^{\ell-1},\ w_2 m_{i,1}^{\ell-1},\ w_2 m_{i,2}^{\ell-1}, \ldots, w_2 m_{i,n_i}^{\ell-1}\Big)$	All neighbours are multiplied by the **same** scalar w_2 and then a component-wise max is taken. The max operator is symmetric (i.e. permutation-invariant), so the whole expression is invariant to neighbour ordering.	**Valid**

(b) Training loss for the simple graph

The graph has five nodes, but nodes **1** and **4** are held-out for testing.

Hence the training set consists of the three nodes $\{2, 3, 5\}$.

node	true label y_i	predicted probability $\hat{y}_i$
2	1	b
3	1	c
5	0	e

The binary-cross-entropy (BCE) loss for a single example is

$$\ell_{\text{BCE}}(y, \hat{y}) = -\Big[y \log \hat{y} + (1-y)\log(1-\hat{y})\Big].$$

The **average** loss over the three training points is therefore

$$\mathcal{L}_{\text{train}} = \frac{1}{\;}\Big[-[1 \cdot \log b + 0 \cdot \log(1-b)] \ + \ -[1 \cdot \log c + 0 \cdot \log(1-c)] \ + \ -[$$

$$\text{node 2} \qquad\qquad\qquad\qquad \text{node 3}$$

$$= \frac{-\log b \ - \log c \ - \log(1-e)}{3}.$$

That is the required training loss expressed in terms of the three learned probabilities b, c, e.

(c) Analysis of the update rule

The update that is given for layer ℓ is

$$\boxed{s_i^{\ell} = s_i^{\ell-1} + \ W_1 \sum_{j=1}^{n_i} \tanh\!\big(W_2\, m_{i,j}^{\ell-1}\big)} \qquad (1)$$

where the $\tanh$ is applied element-wise.

(i) Design pattern(s)

- **Residual connection** – the term $s_i^{\ell-1}$ is added directly to the transformed neighbour aggregation, which is exactly the definition of a residual (skip) connection.
- **Batch-normalisation** – no normalisation of activations appears in (1); therefore this pattern is **not** present.

Hence the correct choice is:

$$\boxed{\text{Residual connection ✓} \qquad \text{Batch normalisation ✗}}$$

(ii) Dimensions of W_1

- The node state $s_i^{\ell-1}$ (and therefore s_i^{ℓ}) lives in $\mathbb{R}^d$.
- The neighbour message $m_{i,j}^{\ell-1}$ has the same dimension d (it will be the previous state of the neighbour, see part (iii)).
- W_2 is said to have k rows, i.e.

$$W_2 \in \mathbb{R}^{k\times d}.$$

After multiplication and $\tanh$, each term $\tanh\!\left(W_2 m_{i,j}^{\ell-1}\right)$ is a **k-dimensional vector**.

- The sum over neighbours therefore yields a vector in $\mathbb{R}^k$.

  To add this to the d-dimensional self-message, W_1 must map $\mathbb{R}^k \to \mathbb{R}^d$:

$$\boxed{W_1 \in \mathbb{R}^{d\times k}}.$$

(iii) Explicit update for nodes 2 and 3 when the message is the neighbour's own state

We now set

$$m_{i,j}^{\ell-1} = s_j^{\ell-1},$$

i.e. each neighbour sends its **previous self-state** as its message.

Very close as is misses the /3 here. Could be because of the formatting since it read the update rule (8) incorrectly above. This shows that proper input formatting matters.

Node 2

Neighbour set of node 2 (from the figure): $\mathcal{N}(2) = \{1, 4, 5\}$; thus $n_2 = 3$.

$$s_2^{\ell} = s_2^{\ell-1} + W_1\left[\tanh\!\left(W_2 s_1^{\ell-1}\right) + \tanh\!\left(W_2 s_4^{\ell-1}\right) + \tanh\!\left(W_2 s_5^{\ell-1}\right)\right].$$

Node 3

Neighbour set of node 3: $\mathcal{N}(3) = \{5\}$; hence $n_3 = 1$.

$$s_3^{\ell} = s_3^{\ell-1} + W_1 \tanh\!\left(W_2 s_5^{\ell-1}\right).$$

These two equations give the full, expanded update for the requested nodes, expressed only in terms of the previous-layer states $\{s_1^{\ell-1}, s_2^{\ell-1}, s_3^{\ell-1}, s_4^{\ell-1}, s_5^{\ell-1}\}$ and the learnable matrices W_1, W_2.

Summary

- (a) (i) and (iii) are valid; (ii) is invalid because it uses neighbour-specific weights.

$$-\log b - \log c - \log(1-e)$$

- (b) Training loss $= \frac{}{3}$.
- (c) The rule contains a residual connection; W_1 has shape $d \times k$; the explicit updates for nodes 2 and 3 are given above.

Executive Summary:

Overall, gpt-oss-120b was really good at one-shotting the problems. However it seems like its biggest weakness is reading input. Inputs need to be formatted properly or else it can miss critical information. I did not face hallucinations, but is probably because I cleaned the context after every problem and provided a good initial context for the LLM.