

You asked:

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 (A^k) 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, ($A_{\{i,j\}} = 1$) if vertices (i) and (j) are connected in the graph and

* Claude Replied:

I'll work through this step-by-step to find the output of the j -th node at layer k .

Setup

Let me denote:

- $h_i^{(k)}$ = the n -dimensional output vector of node i at layer k
- The initial input: $h_i^{(0)} = e_i$ (the i -th standard basis vector, a one-hot encoding)

Layer-by-layer computation

Layer 0 (Input):

$$h_i^{(0)} = e_i = (0, \dots, 0, 1, 0, \dots, 0)^T$$

where the 1 is in the i -th position.

Layer 1: Node j receives weighted inputs from all nodes in layer 0. The weight from node i to node j is $A_{i,j}$.

$$h_j^{(1)} = \sum_{i=1}^n A_{i,j} \cdot h_i^{(0)} = \sum_{i=1}^n A_{i,j} \cdot e_i$$

Since e_i is the i -th standard basis vector, this sum picks out the i -th component being $A_{i,j}$ for each i . Therefore:

$$h_j^{(1)} = (A_{1,j}, A_{2,j}, \dots, A_{n,j})^T = A^T e_j = j\text{-th column of } A^T$$

Wait, let me reconsider the indexing. Actually:

$$h_j^{(1)} = \sum_{i=1}^n A_{i,j} \cdot e_i$$

This gives us a vector whose i -th component is $A_{i,j}$.

Pattern Recognition:

By the same logic, at layer 2:

$$h_j^{(2)} = \sum_{i=1}^n A_{i,j} \cdot h_i^{(1)}$$

Substituting the expression for $h_i^{(1)}$:

$$h_j^{(2)} = \sum_{i=1}^n A_{i,j} \sum_{\ell=1}^n A_{\ell,i} e_\ell = \sum_{\ell=1}^n \left(\sum_{i=1}^n A_{\ell,i} A_{i,j} \right) e_\ell$$

The term in parentheses is $(A^2)_{\ell,j}$, so:

$$h_j^{(2)} = \sum_{\ell=1}^n (A^2)_{\ell,j} e_\ell$$

General Formula

By induction, the output of node j at layer k is:

$$h_j^{(k)} = \sum_{i=1}^n (A^k)_{i,j} \cdot e_i$$

Correct

Or equivalently, $h_j^{(k)}$ is the **j-th column of $(A^k)^T$** , which can also be written as:

$$h_j^{(k)} = (A^k)^T e_j \text{ or } h_j^{(k)} = \text{vector with i-th component equal to } (A^k)_{i,j}$$

This beautifully connects to the interpretation that $(A^k)_{i,j}$ counts the number of walks of length k from vertex i to vertex j : the i-th component of $h_j^{(k)}$ tells us how many k-step walks end at node j starting from node i .

You asked:

(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 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. * $(L_k(i,j))$ 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, $(L_0(i,j) = 1_{\{i=j\}})$. 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).)

* Claude Replied:

I'll prove that the i-th component of $h_j^{(k)}$ equals $L_k(i,j)$, the number of paths of length k from vertex i to vertex j .

Proof by Induction

Claim: The i-th component of $h_j^{(k)}$ is $L_k(i,j)$.

From part (a), we know that $h_j^{(k)}$ is an n-dimensional vector whose i-th component is $(A^k)_{i,j}$.

So we need to prove: $(A^k)_{i,j} = L_k(i,j)$.

Base Case: $k = 0$

For $k = 0$:

- $(A^0)_{i,j} = I_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$
- By convention, $L_0(i,j) = 1_{i=j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$

Therefore, $(A^0)_{i,j} = L_0(i,j)$. ✓

Although the solution starts with $k = 1$, I think this is fine.

Inductive Step

Inductive Hypothesis: Assume that for some $k \geq 0$, we have $(A^k)_{i,j} = L_k(i,j)$ for all vertices i and j .

Goal: Prove that $(A^{k+1})_{i,j} = L_{k+1}(i,j)$.

Proof of Inductive Step

By matrix multiplication:

$$(A^{k+1})_{i,j} = (A^k \cdot A)_{i,j} = \sum_{\ell=1}^n (A^k)_{i,\ell} \cdot A_{\ell,j}$$

By the inductive hypothesis, $(A^k)_{i,\ell} = L_k(i,\ell)$, so:

$$(A^{k+1})_{i,j} = \sum_{\ell=1}^n L_k(i,\ell) \cdot A_{\ell,j}$$

Now, recall that $A_{\ell,j} = 1$ if and only if there is an edge from ℓ to j in the graph, i.e., $\ell \in V(j)$ (the set of neighbors of j). Otherwise, $A_{\ell,j} = 0$.

Therefore:

$$(A^{k+1})_{i,j} = \sum_{\ell \in V(j)} L_k(i, \ell)$$

Combinatorial Interpretation

Now let's count $L_{k+1}(i, j)$ directly:

A path of length $k + 1$ from i to j consists of:

1. A path of length k from i to some intermediate vertex ℓ
2. Followed by one edge from ℓ to j

For this to be possible, ℓ must be a neighbor of j , i.e., $\ell \in V(j)$.

The number of paths of length $k + 1$ from i to j that pass through ℓ as the second-to-last vertex is exactly $L_k(i, \ell)$.

Summing over all possible choices of ℓ :

$$L_{k+1}(i, j) = \sum_{\ell \in V(j)} L_k(i, \ell)$$

The proof looks fine to me.

Conclusion

We have shown:

$$(A^{k+1})_{i,j} = \sum_{\ell \in V(j)} L_k(i, \ell) = L_{k+1}(i, j)$$

By the principle of mathematical induction, $(A^k)_{i,j} = L_k(i, j)$ for all $k \geq 0$.

Therefore, **the i -th component of the output of node j at layer k equals $L_k(i, j)$** , the number of paths of length k from vertex i to vertex j . \square

You asked:

(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 activations 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?

* Claude Replied:

Great question! Let me work through the matrix representation of this GNN.

Matrix Representation

If we stack all node activations into a matrix where the i -th row contains the activation vector of node i , then:

Matrix H at layer k :

$$H^{(k)} \in \mathbb{R}^{n \times n}$$

where the i -th row is $(h_i^{(k)})^T$, the activation of node i at layer k .

From our initial setup:

- $H^{(0)} = I_n$ (the identity matrix, since node i starts with one-hot vector e_i)

The Update Function

From part (a), we found that each node j computes:

$$h_j^{(k+1)} = \sum_{i=1}^n A_{i,j} \cdot h_i^{(k)}$$

In matrix form, if we stack these column vectors as rows, we need to be careful about the transpose. Let's think about this more carefully:

The j-th row of $H^{(k+1)}$ is $(h_j^{(k+1)})^T$.

We have:

$$h_j^{(k+1)} = \sum_{i=1}^n A_{i,j} \cdot h_i^{(k)}$$

Taking the transpose:

$$(h_j^{(k+1)})^T = \sum_{i=1}^n A_{i,j} \cdot (h_i^{(k)})^T$$

This means the j-th row of $H^{(k+1)}$ is a weighted sum of rows of $H^{(k)}$, with weights from column j of A .

Therefore, the **update function** is:

$$H^{(k+1)} = A^T H^{(k)}$$

Again, this is correct.

Or equivalently (and more commonly written):

$$H^{(k+1)} = AH^{(k)}$$

if we use the convention that $A_{i,j}$ represents the weight from node j to node i (which is the more standard GNN convention).

The Two-Step Process

In GNN terminology, the update at each layer consists of:

1. Aggregation: Each node collects (aggregates) information from its neighbors

- Node j aggregates: $\text{agg}_j = \sum_{i \in V(j)} h_i^{(k)}$
- In our case with weights: $\text{agg}_j = \sum_{i \in V(j)} A_{i,j} \cdot h_i^{(k)}$

2. Update/Combine: Each node updates its representation based on the aggregated information

- In our simple linear case: $h_j^{(k+1)} = \text{agg}_j$
- More generally in GNNs: $h_j^{(k+1)} = \text{UPDATE}(h_j^{(k)}, \text{agg}_j)$

This is not exactly what is written in the solution, but this is very close, just need to state that $A_{\{i, j\}} = 1$. I think this is the right answer.

In Our Specific Case

The update function is simply the **identity function** applied to the aggregated inputs:

$$h_j^{(k+1)} = \text{AGGREGATE} \left(\{h_i^{(k)} : i \in V(j)\} \right) = \sum_{i \in V(j)} A_{i,j} \cdot h_i^{(k)}$$

There's no additional transformation—the output is exactly the weighted sum of neighbor activations. This is why it's a "deep linear network."

In more sophisticated GNNs, you might have:

$$h_j^{(k+1)} = \sigma \left(W \cdot \sum_{i \in V(j)} h_i^{(k)} + b \right)$$

where W is a learnable weight matrix, b is a bias, and σ is a nonlinear activation function.

You asked:

(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)?

* Claude Replied:

Great question! Let me think through what happens when we replace sum aggregation with max aggregation.

Setup with Max Aggregation

With max aggregation, the update rule becomes:

$$h_j^{(k+1)} = \max_{i \in V(j)} h_i^{(k)}$$

where the max is taken component-wise across the n-dimensional vectors.

More explicitly, the ℓ -th component of $h_j^{(k+1)}$ is:

$$[h_j^{(k+1)}]_\ell = \max_{i \in V(j)} [h_i^{(k)}]_\ell$$

Analysis

Layer 0:

- $h_i^{(0)} = e_i$ (one-hot encoding)

Layer 1:

- $h_j^{(1)} = \max_{i \in V(j)} e_i$
- The ℓ -th component: $[h_j^{(1)}]_\ell = \max_{i \in V(j)} [e_i]_\ell = \begin{cases} 1 & \text{if } \ell \in V(j) \\ 0 & \text{otherwise} \end{cases}$

So $[h_j^{(1)}]_\ell = 1$ if and only if there exists a path of length 1 from ℓ to j (i.e., ℓ is a neighbor of j).

Layer 2:

- $[h_j^{(2)}]_\ell = \max_{i \in V(j)} [h_i^{(1)}]_\ell$ This is correct
- This equals 1 if there exists some $i \in V(j)$ such that $[h_i^{(1)}]_\ell = 1$
- Which means there exists a neighbor i of j such that $\ell \in V(i)$
- This is equivalent to: there exists a path of length 2 from ℓ to j

General Pattern:

By induction, the i -th component of $h_j^{(k)}$ is:

$$[h_j^{(k)}]_i = \begin{cases} 1 & \text{if there exists at least one path of length } k \text{ from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Interpretation

With max aggregation, the output of node j at layer k is a binary indicator vector that tells us which nodes can reach node j in exactly k steps.

In other words:

- **Sum aggregation counts:** How many paths of length k exist from i to j ?
- **Max aggregation detects:** Does there exist any path of length k from i to j ?

This is the difference between counting paths (sum) and detecting reachability (max). Max aggregation computes a form of **k-hop reachability** in the graph—it answers the question "can I get from node i to node j in exactly k steps?" rather than "how many ways can I get there?"

This connection shows why different aggregation functions in GNNs capture different structural properties of graphs!

You asked:

(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?

Claude Replied:

Great application problem! Let me work through how to formulate this molecular bond-breaking prediction as a graph learning task.

(i) Graph Representation

Nodes: Each atom in the molecule is a node.

Edges: Each chemical bond between atoms is an edge.

- **Undirected graph:** Chemical bonds are symmetric relationships between atoms, so the graph should be undirected.
- **Edge features:** Each edge should have features representing:
 - Bond length (given in the problem)
 - Bond type (single, double, triple, aromatic, etc.)
 - Possibly bond angle information if available

Global state/Graph-level features:

This is correct.

- Molecule type (carbohydrate, protein, lipid, etc.) - can be one-hot encoded
- Overall molecular properties (total molecular weight, pH environment if relevant, temperature, etc.)
- Any experimental conditions

(ii) Using GNN Outputs for Prediction

After running the GNN for K layers, we have node embeddings $h_i^{(K)}$ for each atom i .

To predict which bond will break first, we need to make a **prediction for each edge** (bond). For each edge (i, j) :

Edge representation: Combine the representations of the two endpoint nodes:

$$e_{ij} = \text{COMBINE}(h_i^{(K)}, h_j^{(K)}, \text{edge_features}_{ij})$$

Common choices for COMBINE:

- Concatenation: $e_{ij} = [h_i^{(K)} \| h_j^{(K)} \| \text{edge_features}_{ij}]$
- Sum: $e_{ij} = h_i^{(K)} + h_j^{(K)}$
- Element-wise product
- More sophisticated: $e_{ij} = \text{MLP}([h_i^{(K)} \| h_j^{(K)} \| \text{edge_features}_{ij}])$

Not sure why node embeddings should be involved, but it can certainly be one method.

Prediction head: Apply a classifier to predict bond-breaking probability:

$$p_{ij} = \text{softmax over all edges } (\text{MLP}(e_{ij}))$$

or for each edge independently:

$$s_{ij} = \text{MLP}(e_{ij})$$

where s_{ij} is a score indicating the likelihood that bond (i, j) breaks first.

Then apply softmax across all edges to get a probability distribution over which bond breaks first:

$$p_{ij} = \frac{\exp(s_{ij})}{\sum_{(k,\ell) \in \text{Edges}} \exp(s_{k\ell})}$$

Alternative: Include global state information:

$$s_{ij} = \text{MLP}([e_{ij} \| g])$$

where g is a global graph representation obtained by aggregating all node features:

$$g = \text{READOUT}(\{h_i^{(K)} : i \in \text{nodes}\})$$

Common readout functions: mean, max, sum, or attention-based pooling.

(iii) Node Feature Encoding for Input

The initial node representation $h_i^{(0)}$ for each atom i should encode:

1. Atom identity (element type):

- One-hot encoding of element: [H, C, N, O, S, P, ...]
- Or learnable embedding for each element type
- Atomic number (can be normalized)

2. Atomic properties:

- Atomic mass
- Electronegativity
- Number of valence electrons
- Formal charge
- Hybridization state (sp , sp^2 , sp^3)
- Aromaticity (binary: is this atom part of an aromatic ring?)

I think this is thinking too much. It uses too much expertise knowledge and does not actually capture the spirit of the question. However, it could be an encoding.

3. Local structural features:

- Degree (number of bonds/neighbors)
- Number of hydrogen atoms attached
- Whether the atom is in a ring
- Ring size if applicable

4. Chemical context:

- Implicit hydrogen count
- Radical electrons if any

Example encoding:

$$h_i^{(0)} = [\text{one_hot(element)} \parallel \text{degree} \parallel \text{valence} \parallel \text{formal_charge} \parallel \text{hybridization} \parallel \text{aromatic} \parallel \text{in_ring}]$$

This creates a feature vector of dimension, say, 50-100 for each atom.

Complete Pipeline Summary

1. **Input:** Molecular graph with atom features at nodes, bond features at edges, molecule type as global feature
2. **GNN layers:** Message passing for K layers to aggregate neighborhood information
3. **Edge scoring:** For each bond, combine endpoint node embeddings and edge features
4. **Output:** Softmax over all edges to predict which bond breaks first

This formulation allows the GNN to learn:

- Local chemical environment around each bond
- How strain and electron distribution affect bond stability
- Which structural motifs are associated with bond breaking
- How global molecular properties (like molecule type) affect the prediction

You asked:

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.)

* Claude Replied:

Great exercise in drawing analogies between CNNs and GNNs! Let me fill out the table by thinking about what role each operation serves and finding the corresponding analogue.

CNN	
Image classification	Graph-level prediction problem
	Node-level prediction problem
Color jitter data augmentation (adjusting the color or brightness of an image)	Node feature perturbation (adjusting node features)
Image flip data augmentation	Graph augmentation: node/edge flipping
Channel dropout	Feature dropout (dropping node features)
Zero padding edges	Adding virtual nodes or self-loops
ResNet skip connections	Skip connections between GNN layers or jumping knowledge networks (preserving information across layers)
Blurring an image	Over-smoothing or adding more neighbors
	Predicting missing values of nodes

Reasoning for each analogy:

The Image flipping part and the zero padding part are incorrect

Color jitter perturbs pixel features → **Node feature perturbation** perturbs node features

Image flip creates a semantically equivalent but transformed input → **Graph augmentation** (edge/node dropping, permutation) creates structurally similar graphs since graphs have no fixed orientation

Channel dropout randomly zeros out feature channels in CNNs → **Feature dropout** does the same for node features, or **edge dropout** provides regularization by randomly removing graph structure

Zero padding handles boundary conditions in CNNs → **Virtual nodes** or **self-loops** help with boundary/connectivity issues in graphs (virtual nodes allow global information aggregation; self-loops prevent information loss during aggregation)

ResNet skip connections preserve information across layers → **Skip connections/Jumping knowledge** in GNNs preserve node information across message-passing layers, preventing over-smoothing

Blurring averages neighboring pixel values → **Over-smoothing** in deep GNNs makes neighboring node features increasingly similar, or explicit **graph diffusion** operations smooth features across the graph

Predicting missing values is a node-level task unique to GNNs, similar to dense prediction tasks in CNNs but operating on irregular graph structure

You asked:

Can you also fill out the blank in the CNN section?

* Claude Replied:

Ah yes, good catch! Let me fill in the CNN analogue for "Predicting missing values of nodes":

CNN	
Image classification	Graph-level prediction
Image segmentation or dense prediction tasks (predicting a label/value for each pixel)	Node-level prediction
Color jitter data augmentation (adjusting the color or brightness of an image)	Node feature perturbation
Image flip data augmentation	Graph augmentation
Channel dropout	Feature dropout (dropout)
Zero padding edges	Adding virtual nodes
ResNet skip connections	Skip connections between jumping knowledge blocks
Blurring an image	Over-smoothing or adversarial smoothing
Image inpainting or denoising (predicting missing/corrupted pixel values)	Predicting missing values

Reasoning:

Correct

Image inpainting/denoising involves predicting missing or corrupted pixel values based on surrounding context → **Predicting missing node values** uses the graph structure and neighboring nodes to infer missing node features

Both tasks involve filling in missing information using local context - in CNNs this is spatial neighborhoods of pixels, in GNNs this is graph neighborhoods of nodes. Examples include:

- CNN: filling in a masked region of an image, removing noise
- GNN: imputing missing node attributes, predicting unknown properties of molecules/users in a social network

You asked:

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?

* Claude Replied:

Great question! There are several strategies for handling missing node values in graph-level classification:

1. Imputation Before Graph-Level Prediction

First impute the missing node values, then use the complete graph for classification:

- **Mean/median imputation:** Replace missing values with the mean or median of that feature across all nodes (either in this graph or across the dataset)
- **GNN-based imputation:** Train a separate GNN to predict missing node values based on:
 - Features of neighboring nodes
 - Graph structure
 - Available features of the node itself
- **Multi-task learning:** Jointly train for both node-level imputation and graph-level classification

2. Masking/Indicator Features

Explicitly tell the GNN which values are missing:

- **Add indicator features:** Augment each node's feature vector with binary flags indicating which features are missing
 - e.g., $h_i^{(0)} = [\text{features} \parallel \text{missing_mask}]$
- **Set missing values to zero or a special value** and let the indicator features inform the GNN how to handle them
- The GNN can learn to weight information differently from nodes with missing data

3. Aggregation-Aware Approaches

Modify the GNN aggregation to handle missingness:

- **Attention mechanisms:** Learn to down-weight nodes with missing features during message passing

- **Robust aggregation functions:** Use aggregation functions that are less sensitive to missing data (e.g., median instead of mean)
- **Weighted aggregation:** Weight contributions based on how complete each node's features are

4. Architecture Modifications

- **Multiple imputation:** Create multiple versions of the graph with different imputations, run predictions on each, and ensemble the results
- **Uncertainty-aware models:** Model uncertainty in the missing values and propagate this uncertainty through to the final prediction
- **Variational approaches:** Use variational GNNs that can handle missing data by marginalizing over possible values

5. During Training: Leverage Missingness as Regularization

If missingness occurs during training:

- **Treat as data augmentation:** Randomly drop node features during training to make the model robust to missingness
- **Self-supervised pre-training:** Pre-train the GNN to predict masked node features (similar to BERT), then fine-tune for graph classification

Recommended Approach

Good enough

For most practical cases, I'd recommend:

1. **Add indicator features** to mark which values are missing
2. **Use a simple imputation strategy** (like mean imputation or zero-filling) combined with the indicators
3. **Let the GNN learn** how to handle these nodes during message passing

This is simple, interpretable, and allows the model to learn which features are critical vs. which can be inferred from graph structure.

The key insight is that GNNs are well-suited to handle missing node features because they can leverage **graph structure and neighboring nodes** to propagate information and compensate for missing values - this is actually one of the advantages of GNNs over methods that treat nodes independently!

You asked:

(h) Consider the graph neural net architecture shown in Figure 2. It includes representations of nodes (V_n), edges (E_n), and global state (U_n). At each timestep, each node and edge is updated by aggregating neighboring nodes/edges, as well as global state. The global state is updated by getting information from all nodes and edges. (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 is 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?

 Claude Replied:

Great questions about the computational and architectural properties of GNNs!

(i) Scaling with Number of Nodes

Let's say we have a graph with n nodes, and we double it to $2n$ nodes.

Number of Learned Weights

Answer: The number of learned weights **DOES NOT CHANGE**.

Reasoning:

- GNNs use **weight sharing** across all nodes - the same update function f_V is applied to every node
- The learned parameters are in the update functions (MLPs, linear layers, etc.), not specific to individual nodes
- Just like CNNs use the same convolutional filters across all spatial locations, GNNs use the same parameters for all nodes

- Doubling nodes doesn't add new parameters, it just applies the existing parameters more times

Amount of Computation

This depends on the graph connectivity:

Case 1: Average node degree remains constant (sparse graph)

If each node has degree d on average:

- Each node aggregates from d neighbors
- With n nodes: computation is $O(n \cdot d) = O(n)$ since d is constant
- With $2n$ nodes: computation is $O(2n \cdot d) = O(2n)$

Answer: Computation DOUBLES (scales linearly with number of nodes)

Case 2: Fully connected graph

If the graph is fully connected:

- Each node has degree $n - 1$ (connects to all other nodes)
- With n nodes: computation is $O(n \cdot n) = O(n^2)$
- With $2n$ nodes: computation is $O(2n \cdot 2n) = O(4n^2)$

Answer: Computation QUADRUPLES (scales quadratically with number of nodes)

Summary:

- **Learned weights:** No change (weight sharing)
- **Computation (sparse):** Doubles - $O(n) \rightarrow O(2n)$
- **Computation (fully connected):** Quadruples - $O(n^2) \rightarrow O(4n^2)$

Correct

(ii) Where are the Learned Weights/Parameters?

Looking at Figure 2, learned parameters appear in:

1. Node Update Function f_{V_n}

- Takes as input: aggregated neighbor information, current node features, possibly edge features and global state
- Typically an MLP or neural network
- Parameters: weight matrices and biases in this function

2. Edge Update Function f_{E_n}

- Takes as input: features of connected nodes, current edge features, possibly global state
- Typically an MLP
- Parameters: weight matrices and biases

Correct

3. Global State Update Function f_{U_n}

- Takes as input: aggregated node features, aggregated edge features, current global state
- Typically an MLP
- Parameters: weight matrices and biases

4. Aggregation Functions (sometimes)

- In simple GNNs: aggregation is parameter-free (sum, mean, max)
- In attention-based GNNs (like GAT): the **attention mechanism** has learned parameters to compute attention weights
 - e.g., $\alpha_{ij} = \text{softmax}(a^T [Wh_i \| Wh_j])$ where W and a are learned

5. Pooling Function ρ (if parameterized)

- Simple pooling (sum, mean, max): no parameters
- Attention-based pooling or learned pooling: has parameters

Key insight: The same set of parameters is shared across all nodes/edges of the same type - this is what makes GNNs scalable and permutation-invariant!

(iii) Incorporating Directed Edges

For **directed edges**, you need to distinguish between incoming and outgoing information flow. Here are several approaches:

Approach 1: Separate Aggregation for In-edges and Out-edges

For each node v :

- Aggregate separately from **incoming neighbors** $N_{in}(v)$ and **outgoing neighbors** $N_{out}(v)$
- Use different update functions or concatenate:

$$h_v^{(k+1)} = f_V \left(h_v^{(k)}, \text{AGG}_{in}\{h_u^{(k)} : u \in N_{in}(v)\}, \text{AGG}_{out}\{h_u^{(k)} : u \in N_{out}(v)\} \right)$$

Approach 2: Separate Weight Matrices

Use different learned transformations for incoming vs. outgoing messages:

$$h_v^{(k+1)} = f_V \left(h_v^{(k)}, \sum_{u \in N_{in}(v)} W_{in} h_u^{(k)}, \sum_{u \in N_{out}(v)} W_{out} h_u^{(k)} \right)$$

Approach 3: Edge Direction as Edge Feature

Treat edge direction as an edge feature:

Correct

- Add a binary or categorical feature to E_n indicating direction
- The edge update function f_{E_n} can learn to handle direction differently
- When aggregating to nodes, the direction feature helps distinguish message sources

Approach 4: Reverse Edges

Create a graph with both:

- Original directed edges with type "forward"
- Reverse edges with type "backward"
- Use edge types to learn different message functions

Example in the Figure 2 framework:

For directed edges, the edge update becomes:

$$E_{n+1}^{(i,j)} = f_{E_n}(E_n^{(i,j)}, V_n^{(i)}, V_n^{(j)}, U_n, \text{direction})$$

And node updates aggregate with awareness of edge direction:

$$V_{n+1}^{(j)} = f_{V_n} \left(V_n^{(j)}, \rho_{in}(\{E_n^{(i,j)} : i \in N_{in}(j)\}), \rho_{out}(\{E_n^{(j,k)} : k \in N_{out}(j)\}), U_n \right)$$

The key is to **break the symmetry** - make the model aware that information flow from node i to j is different from j to i .

You asked:

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 $(s_{-i}^{\ell-1})$ be the self-message (i.e. the state computed at the previous layer for this node) for this node from the preceding layer, while the preceding layer messages from the (n_i) neighbors of node (i) are denoted by $(m_{-i,j}^{\ell-1})$ where (j) ranges from 1 to (n_i) . 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_i^ℓ) . 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) $[s_i^\ell = w_1 s_i^{\ell-1} + w_2 \frac{1}{n_i} \sum_{j=1}^{n_i} m_{i,j}^{\ell-1}]$ (ii) $[s_i^\ell = \max(w_1 s_i^{\ell-1}, w_2 m_{i,1}^{\ell-1}, w_3 m_{i,2}^{\ell-1}, \dots, w_{n_i} m_{i,n_i}^{\ell-1}]$ where the max acts component-wise on the vectors. (iii) $[s_i^\ell = \max(w_1 s_i^{\ell-1}, w_2 m_{i,1}^{\ell-1}, w_2 m_{i,2}^{\ell-1}, \dots, w_{n_i} m_{i,n_i}^{\ell-1}]$ 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. Table 1 (see the attached image) 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 $\frac{1}{n} \sum_x \left(y(x) \log \hat{y}(x) + (1-y(x)) \log (1-\hat{y}(x)) \right)$ where the (x) in the sum ranges over the training points and $(\hat{y}(x))$ 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) . $s_i^\ell = s_{i-1}^\ell + W_1 \frac{1}{n_i} \sum_j m_{i,j}^{\ell-1}$ where the (\tanh) nonlinearity acts element-wise. For a given node (i) in the graph, let $(s_i^{\ell-1})$ be the self-message for this node from the preceding layer, while the preceding layer messages from the (n_i) neighbors of node (i) are denoted by $(m_{i,j}^{\ell-1})$ where (j) ranges from 1 to (n_i) . 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 (W_2) has (k) rows, what are the dimensions of the matrix (W_1) ? (iii) If we choose to use the state $(s_i^{\ell-1})$ itself as the message $(m_{i,j}^{\ell-1})$ going to all of node (i) 's neighbors, please write out the update rules corresponding to (1) giving (s_i^ℓ) for the graph in Figure 3 for nodes $(i=2)$ and $(i=3)$ in terms of information from earlier layers. Expand out all sums.

 Claude Replied:

Great problem! Let me work through each part systematically.

(a) Validity of Update Functions

For a GNN on an undirected graph to be valid, it must be:

- **Permutation invariant** with respect to the ordering of neighbors (since the graph structure doesn't define an ordering)
- Able to use **weight sharing** across all nodes regardless of their degree

(i) $s_i^\ell = w_1 s_i^{\ell-1} + w_2 \frac{1}{n_i} \sum_{j=1}^{n_i} m_{i,j}^{\ell-1}$

VALID ✓

- The sum over neighbors is permutation invariant (reordering neighbors doesn't change the sum)
- Uses fixed weights w_1, w_2 regardless of node degree
- The average aggregation $\frac{1}{n_i} \sum$ is permutation invariant and handles variable degree

(ii) $s_i^\ell = \max(w_1 s_i^{\ell-1}, w_2 m_{i,1}^{\ell-1}, w_3 m_{i,2}^{\ell-1}, \dots, w_{n_i+1} m_{i,n_i}^{\ell-1})$

INVALID ✗

Two major issues:

1. **Different weights for different neighbors:** Uses $w_2, w_3, \dots, w_{n_i+1}$ - different weights for each neighbor position. This breaks permutation invariance because it matters which neighbor is "first," "second," etc., but undirected graphs have no canonical ordering of neighbors.
2. **Number of parameters depends on degree:** Nodes with different degrees would require different numbers of weights, making weight sharing impossible across nodes with different degrees.

(iii) $s_i^\ell = \max(w_1 s_i^{\ell-1}, w_2 m_{i,1}^{\ell-1}, w_2 m_{i,2}^{\ell-1}, \dots, w_2 m_{i,n_i}^{\ell-1})$

Correct

VALID ✓

- All neighbor messages use the same weight w_2
- The max operation is permutation invariant (reordering the arguments doesn't change the result)
- Works for any node degree
- This is essentially: $s_i^\ell = \max(w_1 s_i^{\ell-1}, \max_{j=1}^{n_i} w_2 m_{i,j}^{\ell-1})$

(b) Training Loss Calculation

From Table 1 and Figure 3:

- **Training nodes:** 2, 3, 5 (holding back 1 and 4)
- Ground truth labels: $y_2 = 1, y_3 = 1, y_5 = 0$
- Predicted probabilities: $\hat{y}_2 = b, \hat{y}_3 = c, \hat{y}_5 = e$

The average binary cross-entropy loss with $n = 3$ training points:

$$\text{Loss} = \frac{1}{3} \sum_{i \in \{2,3,5\}} \left[y_i \log \frac{1}{\hat{y}_i} + (1 - y_i) \log \frac{1}{1 - \hat{y}_i} \right]$$

Calculating for each node:

Node 2: $y_2 = 1, \hat{y}_2 = b$

$$1 \cdot \log \frac{1}{b} + 0 \cdot \log \frac{1}{1-b} = -\log b$$

Node 3: $y_3 = 1, \hat{y}_3 = c$

$$1 \cdot \log \frac{1}{c} + 0 \cdot \log \frac{1}{1-c} = -\log c$$

Node 5: $y_5 = 0, \hat{y}_5 = e$

$$0 \cdot \log \frac{1}{e} + 1 \cdot \log \frac{1}{1-e} = -\log(1-e)$$

Correct

Training Loss:

$$\boxed{\text{Loss} = \frac{1}{3} [-\log b - \log c - \log(1-e)] = -\frac{1}{3} [\log b + \log c + \log(1-e)]}$$

Or equivalently:

$$\boxed{\text{Loss} = -\frac{1}{3} \log [bc(1-e)]}$$

(c) Update Rule Analysis

Given: $s_i^\ell = s_i^{\ell-1} + W_1 \frac{\sum_{j=1}^{n_i} \tanh(W_2 m_{i,j}^{\ell-1})}{n_i}$

(i) Design Pattern

Residual connection

Correct

The update has the form $s_i^\ell = s_i^{\ell-1} + f(\text{neighbors})$, which is exactly a residual/skip connection. The state from the previous layer is added to the processed neighbor information.

Batch normalization - Not present (no normalization across a batch)

(ii) Dimensions of W_1

- State $s_i^{\ell-1}$ is d -dimensional
- Messages $m_{i,j}^{\ell-1}$ are also d -dimensional (assuming messages have same dimension as states)
- W_2 has k rows, so $W_2 m_{i,j}^{\ell-1}$ produces a k -dimensional vector
- After \tanh and averaging, we have a k -dimensional vector
- W_1 must map this k -dimensional vector to a d -dimensional vector to add to $s_i^{\ell-1}$

Answer: W_1 has dimensions $d \times k$

(iii) Expanded Update Rules

Given $m_i^{\ell-1} = s_i^{\ell-1}$ (message equals state).

Looking at Figure 3:

- **Node 2** has neighbors: 1, 4, 5 (so $n_2 = 3$)
- **Node 3** has neighbors: 4 (so $n_3 = 1$)

For node 2:

$$\boxed{s_2^\ell = s_2^{\ell-1} + W_1 \frac{\tanh(W_2 s_1^{\ell-1}) + \tanh(W_2 s_4^{\ell-1}) + \tanh(W_2 s_5^{\ell-1})}{3}}$$

For node 3:

$$s_3^\ell = s_3^{\ell-1} + W_1 \frac{\tanh(W_2 s_4^{\ell-1})}{1} = s_3^{\ell-1} + W_1 \tanh(W_2 s_4^{\ell-1})$$

This is incorrect probably because Claude does not recognize image that well