

Graph neural networks homework help

Manan



File: hw06_question.pdf

I'm a student in EECS 182 (Deep Learning) at UC Berkeley.

I'm working through Homework 6, which covers Graph Neural Networks (GNNs), graph dynamics, and some modern optimizer concepts. I need help with the hw questions. I'll share them one at a time.

For each question:

1. Think through the problem step-by-step
2. Show all your reasoning and any relevant math/equations
3. State your final answer clearly
4. If you're uncertain about anything, tell me

Let's start with question 2a

Thought process: Traced network layer computations through matrix multiplication.

The user is asking about Question 2a from their EECS 182 homework. Let me look at the problem carefully.

The problem describes a simple deep linear network on a graph:

- Graph has n vertices with adjacency matrix A (where $A_{i,j} = 1$ if vertices i and j are connected)
- n vertices in each layer, corresponding to n vertices of the graph
- Each vertex has n channels
- Input to node i at layer 0 is a one-hot encoding: $(0, \dots, 0, 1 \text{ at position } i, 0, \dots, 0)$
- Weight connecting node i in layer k to node j in layer $k+1$ is $A_{i,j}$
- At each layer, each node sums up the weighted inputs from its neighbors

Question 2a asks: Write the output of the j -th node at layer k in this network in terms of the matrix A .

Let me think through this step by step.

Layer 0:

- Node i has input $x_{i^0} = e_i$ (the i -th standard basis vector, which is the one-hot encoding)

So the input at layer 0 can be thought of as a matrix X^0 where each row corresponds to a node's output vector. If we arrange it so that node i 's output is the i -th row, then $X^0 = I$ (the identity matrix).

Actually, let me think more carefully about the notation. The problem says:

- Each vertex has n channels
- The input to node i at layer 0 is a one-hot encoding of its own identity

So if we denote the output of node j at layer k as a vector o_{j^k} (which is n -dimensional), then:

- $o_{i^0} = e_i$ (the i -th standard basis vector)

Now, the weight connecting node i in layer k to node j in layer $k+1$ is $A_{i,j}$.

Wait, I need to be careful about the direction here.

The weight from node i (layer k) to node j (layer $k+1$) represents a connection where node j receives input from node i , weighted by $A_{i,j}$. At each layer, nodes compute a weighted sum of inputs from connected nodes.

Specifically, the output of node j at layer $k+1$ emerges from summing the weighted contributions of all nodes in the previous layer:

$$o_j^{k+1} = \text{sum over } i \text{ of } A_{i,j} * o_i^k$$

This reveals how the adjacency matrix A directly governs the network's information propagation across layers.

By arranging outputs as a matrix O^k , where each column represents a node's output, we discover a compact matrix representation: $O^{k+1} = O^k A$.

This formulation elegantly captures how network connections transform node states between layers, with the initial state O^0 being the identity matrix, representing individual node activations.

Tracing through successive layers reveals a powerful pattern: O^k becomes A^k , meaning each node's output at layer k is precisely the k -th power of the adjacency matrix applied to its initial basis vector.

Verifying this approach shows it correctly models network propagation, from initial single-node activations to complex multi-node interactions across network layers.

The key insight emerges: node j 's output at layer k is fundamentally represented by the j -th column of A^k , capturing its weighted connections and transformed state through successive network transformations.

I can express this compactly as a vector operation or break it down to individual component calculations, revealing the underlying computational mechanism of neural network layer propagation.

Question 2a: Output of Node j at Layer k

Setting Up the Problem

Let me denote the output of node j at layer k as \mathbf{o}_j^k , which is an n -dimensional vector (since each node has n channels).

Initial condition (Layer 0): Each node i receives a one-hot encoding of its own identity: $\mathbf{o}_i^0 = e_i = (0, \dots, 0, \underbrace{1}_{\text{i-th}}, 0, \dots, 0)^T$

Deriving the Recurrence

At each layer, node j computes the weighted sum of inputs from all nodes i , where the weight from node i to node j is $A_{i,j}$:

$$\mathbf{o}_j^{k+1} = \sum_{i=1}^n A_{i,j} \cdot \mathbf{o}_i^k$$

Matrix Form

If I arrange all node outputs as **columns** of a matrix $O^{(k)}$ (so the j -th column is $o_j^{(k)}$), then:

$$O^{(k+1)} = O^{(k)} A$$

At layer 0: Since $o_i^{(0)} = e_i$, we have: $O^{(0)} = [e_1 \mid e_2 \mid \dots \mid e_n] = I$

Unrolling the recurrence:

- $O^{(1)} = O^{(0)} \cdot A = I \cdot A = A$
- $O^{(2)} = O^{(1)} \cdot A = A^2$
- \vdots
- $O^{(k)} = A^k$

Final Answer

The output of the j -th node at layer k is:

$$\boxed{o_j^{(k)} = A^k e_j}$$

This is simply the **j-th column of A^k** .

Equivalently, the i -th component of this output vector is $(A^k)_{i,j}$ — the (i,j) entry of the matrix A^k .



Question 2(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. 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, $L_0(i, j) = 1$ if $i=j$, else 0. 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 .)

Thought process: Orchestrated inductive proof linking matrix powers to graph path counts.

The user wants me to prove that 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 .

From part (a), we established that the output of node j at layer k is the j -th column of A^k . So the i -th component of this output is $(A^k)_{\{i, j\}}$.

So I need to prove that $(A^k)_{\{i, j\}} = L_k(i, j)$ for all i, j , and $k \geq 0$.

This is a classic result about adjacency matrices. Let me prove it by induction on k .

**Base case: $k = 0$ **

- $A^0 = I$ (identity matrix)
- $(A^0)_{\{i, j\}} = 1$ if $i = j$, else 0
- $L_0(i, j) = 1$ if $i = j$, else 0 (by convention)
- So $(A^0)_{\{i, j\}} = L_0(i, j)$

Inductive step:

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

We want to show $(A^{k+1})_{\{i, j\}} = L_{k+1}(i, j)$.

Now, $A^{k+1} = A^k \cdot A$, so:

$$(A^{k+1})_{\{i, j\}} = \sum_m (A^k)_{\{i, m\}} \cdot A_{\{m, j\}}$$

By the inductive hypothesis, $(A^k)_{\{i, m\}} = L_k(i, m)$.

And $A_{\{m, j\}} = 1$ if m and j are connected by an edge, 0 otherwise.

So:

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

I'll explain how this relates to path counting. When constructing paths of length $k+1$ from i to j , we consider all possible intermediate vertices m . Each such path consists of a k -length path from i to m , followed by an edge from m to j . The total number of these paths is precisely the sum of k -length paths to each neighbor of j .

This means the number of paths of length $k+1$ from i to j is exactly the sum of paths from i to each

vertex connected to j , which matches our algebraic derivation.

By mathematical induction, this relationship holds for all non-negative integers k , confirming the connection between matrix powers and path counting.

Question 2b: Proving Path Counting via Induction

What We Need to Prove

From part (a), the i -th component of node j 's output at layer k is $(A^k)_{i,j}$.

Claim: $(A^k)_{i,j} = L_k(i, j)$ for all nodes i, j and all $k \geq 0$.

Proof by Induction on k

Base Case: $k = 0$

Left side: $A^0 = I$, so $(A^0)_{i,j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$

Right side: By the given convention, $L_0(i, j) = \mathbf{1}_{i=j} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases}$

✓ Base case holds.

Inductive Hypothesis

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

Inductive Step: Show it holds for $k + 1$

Computing $(A^{k+1})_{i,j}$:

Using matrix multiplication: $\$ A^{k+1} = A^k \cdot A \$$

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

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

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

Since $A_{m,j} = 1$ only when m is a neighbor of j (i.e., $\$ m \in V(j) \$$), and 0 otherwise:

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

Computing $L_{k+1}(i, j)$:

Any path of length $k+1$ from i to j can be decomposed as:

- A path of length k from i to some intermediate vertex m
- Followed by a single edge from m to j

For this to be valid, m must be a neighbor of j , so $m \in V(j)$.

The number of such paths (for a fixed m) is exactly $L_k(i, m)$.

Summing over all possible intermediate vertices:

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

Conclusion

We have shown: $\$(A^{k+1})\{i,j\} = |sum\{m \in V(j)\} L_k(i, m) = L_{k+1}(i, j)$$$

By the principle of mathematical induction:

$$(A^k)_{i,j} = L_k(i, j) \text{ for all } i, j, k \geq 0$$

In words: The i -th output of node j at layer k counts the number of paths of length k from vertex i to vertex j .

- ⌚ Question 2(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?

Thought process: Reconciled matrix representations and derived linear update equations.

The question is asking about two things:

1. If we represent the graph with activations of node i in row i (so we have a matrix where each row corresponds to a node), what is the update function at each layer?
2. Specifically for the GNN described in the previous parts, what is the update function that takes aggregated neighbor inputs and produces the output?

Let me think about this carefully.

In part (a), I had the outputs arranged as columns. Let me reconsider with rows.

If we have a matrix H^k where the i -th row is the activation/output of node i at layer k , then:

- H^k is an $n \times n$ matrix (n nodes, n channels each)
- The i -th row of H^k is the output vector of node i

At layer 0:

- Node i has input e_i (one-hot encoding)
- So the i -th row of H^0 is $e_i^T = (0, \dots, 0, 1, 0, \dots, 0)$ with 1 in position i
- This means $H^0 = I$ (identity matrix)

Now, at each layer, node j receives messages from all its neighbors i (where $A_{\{i,j\}} = 1$).

Wait, let me re-read the original problem setup. The weight connecting node i in layer k to node j in layer $k+1$ is $A_{\{i,j\}}$.

So node j at layer $k+1$ receives:

- From node 1: $A_{\{1,j\}} \times (\text{output of node 1 at layer } k)$
- From node 2: $A_{\{2,j\}} \times (\text{output of node 2 at layer } k)$
- etc.

The output of node j at layer $k+1$ = $\sum_i A_{\{i,j\}} \times (\text{output of node } i \text{ at layer } k)$

If the output of node i at layer k is the i -th row of H^k , denoted h_i^k (a row vector), then:

$$h_j^{k+1} = \sum_i A_{\{i,j\}} \times h_i^k$$

I see this as a matrix multiplication process where each row of the next layer is computed by aggregating neighbor information through the adjacency matrix. The computation captures how each node's output depends on its neighbors' previous layer outputs, weighted by the graph's connectivity.

The matrix representation reveals a elegant transformation: $H^{(k+1)} = A^T H^{(k)}$, which systematically propagates information across the graph's structure. By transposing the adjacency matrix, we enable a clean, vectorized update mechanism that captures local neighborhood interactions efficiently.

Iterating this process from an initial identity matrix $H^{(0)} = I$ generates progressively transformed representations, with each iteration capturing increasingly complex graph-based relationships through successive matrix multiplications.

For undirected graphs, the symmetry $A = A^T$ simplifies the computation to $H^{(k)} = A^k$, where each matrix entry directly represents path connectivity between nodes. The entries encode path lengths and structural relationships, revealing intricate network dynamics through pure linear algebraic operations.

The key insight emerges in distinguishing column versus row representations: output matrices can be viewed either as column-wise node outputs or row-wise node perspectives, with careful transposition maintaining computational consistency.

The update mechanism involves two critical components: matrix propagation through $A^T H^{(k)}$ and a straightforward identity function for local node transformations, ensuring minimal computational overhead while preserving network information flow.

For undirected graphs, the matrix update simplifies to $H^{(k+1)} = A H^{(k)}$, where node activations propagate directly through the adjacency matrix without additional transformation. At each node, the update process aggregates neighbor inputs through summation, then applies an identity function that preserves the aggregated values.

Mathematically, this means node representations are updated by summing neighboring node activations, with no additional nonlinear processing applied during the local update step.

Question 2c: Update Function for the GNN

Part 1: Matrix Form of the Update

Let's define $H^{(k)}$ as the matrix where the **i-th row** contains the activations of node i at layer k .

- $H^{(k)}$ is an $n \times n$ matrix (n nodes, each with n channels)
- $H^{(0)} = I$ (since each node starts with a one-hot encoding of its identity)

Deriving the update:

Node j at layer $k+1$ receives weighted inputs from all nodes i , with weight $A_{\{i,j\}}$:

$$\mathbf{h}_j^{(k+1)} = \sum_{i=1}^n A_{i,j} \cdot \mathbf{h}_i^{(k)}$$

In matrix form, this becomes:

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

(For undirected graphs where $A = A^T$. More generally, it would be $A^T H^{(k)}$.)

Part 2: The Local Update Function

The GNN operation can be decomposed into two steps:

Step 1 - Aggregation: Sum the features from all neighbors $\text{aggregated}_j = \sum_{i \in V(j)} \mathbf{h}_i^{(k)}$

Step 2 - Update: Transform the aggregated result into the output

In this simple GNN, the update function is:

$$f(\text{aggregated input}) = \text{aggregated input}$$

The update function is simply the identity function.

There is no learned transformation, no nonlinearity, and no additional processing—the node just outputs the sum of its neighbors' features directly. This is why the network is "essentially linear" as mentioned in the problem statement.

- ⌚ Question 2(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?

Thought process: Traced through max aggregation layers to reveal path existence indicators.

Let me think about what happens when we replace sum aggregation with max aggregation.

With sum aggregation, we showed that the i-th output of node j at layer k counts the number of paths of length k from i to j.

With max aggregation, instead of summing over neighbors, we take the component-wise maximum.

Let's trace through this:

Layer 0:

Each node i has output $o_i^{(0)} = e_i$ (one-hot vector)

Layer 1:

For node j, we take max over neighbors:

$$o_j^{(1)} = \max_{m \in V(j)} o_m^{(0)} = \max_{m \in V(j)} e_m$$

The i-th component of this is:

$$(o_j^{(1)})_i = \max_{m \in V(j)} (e_m)_i = \max_{m \in V(j)} 1_{\{m=i\}}$$

This equals 1 if $i \in V(j)$ (i.e., if i is a neighbor of j), and 0 otherwise.

So $(o_j^{(1)})_i = 1$ if there exists a path of length 1 from i to j, and 0 otherwise.

Layer 2:

$$o_j^{(2)} = \max_{m \in V(j)} o_m^{(1)}$$

The i-th component:

$$(o_j^{(2)})_i = \max_{m \in V(j)} (o_m^{(1)})_i$$

$(o_m^{(1)})_i = 1$ if there's a path of length 1 from i to m.

So $(o_j^{(2)})_i = 1$ if there exists some $m \in V(j)$ such that there's a path of length 1 from i to m.

This means: there's a path of length 1 from i to some neighbor m of j, then an edge from m to j.

This indicates a path of length 2 from i to j.

The max aggregation reveals a binary reachability pattern: for each layer k, the output shows whether a node is reachable in exactly k steps.

At each layer, we're essentially checking if a path exists between nodes, with the max operation creating a binary indicator of connectivity.

The key distinction emerges between sum and max aggregations: sum counts path occurrences, while max simply signals path existence.

Question 2d: Max Aggregation Interpretation

Tracing Through the Max Aggregation

Layer 0 (unchanged):

$$\mathbf{o}_i^{(0)} = \mathbf{e}_i \quad (\text{one-hot vectors, entries are 0 or 1})$$

Layer 1: For node j with max aggregation: $\max\{\mathbf{o}_j^{(1)}\}_i = \max\{m \in V(j) \mid \mathbf{o}_m^{(0)}_i = 1\}$

This equals:

- **1** if $i \in V(j)$ (i.e., i is a neighbor of j , meaning a path of length 1 exists)
- **0** otherwise

Layer 2:

$$(\mathbf{o}_j^{(2)})_i = \max_{m \in V(j)} (\mathbf{o}_m^{(1)})_i$$

This equals 1 if there exists *any* neighbor m of j such that there's a path of length 1 from i to m —which means a path of length 2 from i to j exists.

General Pattern (by induction)

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

Final Answer

The i -th output of node j at layer k is a binary indicator of whether node j is reachable from node i in exactly k steps.

Comparison:

Aggregation	Output Interpretation
Sum	Count of paths of length k from i to j
Max	Existence of at least one path of length k from i to j (0 or 1)

- Question 2(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?

Thought process: Architected graph-based molecular representation and edge-level prediction strategy.

This is a practical application question about representing molecules as graphs for GNN-based prediction. Let me think through each part carefully.

The problem setup:

- We have organic molecules
- We know: element type for each atom, connectivity (which atoms are bonded), bond lengths, molecule type
- Goal: predict which bond breaks first when heated

Let me address each part:

(i) Graph representation:

- Nodes: atoms
- Edges: chemical bonds between atoms
- Node features: element type
- Edge features: bond length
- Global state: molecule type
- Directed vs undirected: chemical bonds are typically undirected (symmetric), so undirected makes sense

(ii) Making the prediction:

- We're predicting which BOND breaks first
- This is an edge-level prediction task
- After the GNN processes, we get edge representations
- We can apply a classifier to each edge to predict probability of breaking first
- Use softmax over all edges to get the "first to break" prediction
- Could also have a "no bond breaks" option

(iii) Node encoding:

- Element type: could use one-hot encoding (C, H, O, N, etc.) or learned embeddings
- Could also include other atomic properties like atomic number, electronegativity, etc.

Question 2e: Molecular Bond Breaking Prediction

Part (i): Graph Representation

Component	Representation
Nodes	Atoms in the molecule
Edges	Chemical bonds between atoms

Component	Representation
Node features	Element type (e.g., C, H, O, N, S, etc.)
Edge features	Bond length (given in the problem)
Global state	Molecule type (carbohydrate, protein, etc.)
Directed?	Undirected — chemical bonds are symmetric; if atom A is bonded to atom B, then B is equally bonded to A

Part (ii): Using GNN Outputs for Prediction

Since we're predicting **which bond breaks first**, this is an **edge-level prediction task**.

Approach:

1. Run the GNN to compute final node embeddings \mathbf{h}_i for each atom i
2. For each edge (bond) between atoms i and j, construct an edge representation by combining the endpoint node embeddings: $\mathbf{e}_{ij} = f(\mathbf{h}_i, \mathbf{h}_j, \text{edge features})$ where f could be concatenation, sum, or a learned function
3. Pass each edge representation through an MLP to get a scalar score (logit) for "likelihood of breaking first"
4. Apply **softmax over all edges** (plus optionally a "no bond breaks" class) to get a probability distribution:

$$P(\text{bond } (i,j) \text{ breaks first}) = \frac{\exp(s_{ij})}{\sum_{(a,b) \in \text{edges}} \exp(s_{ab})}$$
5. Predict the edge with the highest probability

Part (iii): Encoding Node Representations

For the element type (categorical feature):

- **One-hot encoding:** Create a vector of length |elements| (e.g., 10-20 common elements), with a 1 in the position corresponding to the element
 - Example: C → [1,0,0,0,...], H → [0,1,0,0,...], O → [0,0,1,0,...]
- **Learned embedding:** Map each element to a learned d-dimensional dense vector (more expressive, especially with many element types)

Optional additional features to include:

- Atomic number
- Electronegativity
- Number of valence electrons
- Atomic mass

$$\mathbf{x}_i = [\text{one-hot}(\text{element}_i) \parallel \text{other atomic properties}]$$

where \parallel denotes concatenation.

- Question 2(f): There are analogs of many ConvNet operations which can be done with GNNs. We can think of pixels as nodes and pixel adjacencies as similar to edges. Graph-level classification tasks are analogous to image classification, since both produce a single, global prediction. Fill out the rest of the table (find the GNN equivalent for CNN operations, or CNN equivalent for GNN operations):

CNN	GNN
Image classification	Graph-level prediction problem
???	Node-level prediction problem
Color jitter data augmentation	???
Image flip data augmentation	???
Channel dropout	???
Zero padding edges	???
ResNet skip connections	???
Blurring an image	???
???	Predicting missing values of nodes

Thought process: Systematically mapped CNN operations to their GNN counterparts.

This is a conceptual question asking me to find analogous operations between CNNs and GNNs. Let me think through each one carefully.

1. **??? \leftrightarrow Node-level prediction problem**

- Image classification produces one prediction for the whole image
- Graph-level prediction produces one prediction for the whole graph
- Node-level prediction produces one prediction per node
- In CNNs, what produces one prediction per pixel? **Semantic segmentation** (pixel-wise classification)

2. **Color jitter data augmentation \leftrightarrow ???**

- Color jitter adjusts color/brightness of pixels (modifies node features without changing structure)
- GNN analog: **Node feature perturbation** - adding noise to node features or randomly modifying node attributes

3. **Image flip data augmentation \leftrightarrow ???**

- Flipping an image changes the spatial arrangement but preserves structure
- For graphs: This would be like permuting the node labels/ordering, but since GNNs are permutation invariant, this doesn't really help
- Better analog: **Graph isomorphism augmentation** or since flipping preserves adjacency structure, maybe there's no direct analog because GNNs are already permutation invariant
- Or we could think of it as permuting node indices, which GNNs handle naturally
- Actually, a better answer might be: **Random node/edge permutation** or "No direct analog needed (GNNs are permutation invariant)"

4. **Channel dropout \leftrightarrow ???**

- Dropping out entire channels in a CNN

- GNN analog: **Node feature dropout** or **dropping out dimensions of node embeddings**
5. **Zero padding edges \leftrightarrow ???**
- Zero padding adds "virtual" pixels around the border to preserve spatial dimensions
 - GNN analog: **Adding virtual/dummy nodes** connected to boundary or low-degree nodes to normalize neighborhood sizes
6. **ResNet skip connections \leftrightarrow ???**
- Skip connections that add input to output
 - GNN analog: **Residual connections in GNN layers** - same concept applies directly
 - This is used in many GNNs like GraphSAGE, GCN variants
7. **Blurring an image \leftrightarrow ???**
- Blurring averages pixel values with neighbors, smoothing the image
 - GNN analog: **Feature smoothing / averaging neighbor features** - this is essentially what a basic GNN aggregation does (like in GCN where you average neighbor features)

I'll focus on the skip connections and blurring analogies. Skip connections in neural networks allow direct information flow between layers, preventing vanishing gradients. In GNNs, residual connections work similarly, enabling deeper graph neural network architectures. Blurring's concept of neighborhood averaging directly parallels how GNNs aggregate information from neighboring nodes, smoothing features across the graph's structure.

Laplacian smoothing provides another mechanism for feature averaging, helping normalize node representations by considering local neighborhood characteristics. This technique bridges image processing and graph representation learning, showing how signal processing concepts translate across domains.

Question 2f: CNN \leftrightarrow GNN Analogies

CNN	GNN
Image classification	Graph-level prediction problem
Semantic segmentation (pixel-wise classification)	Node-level prediction problem
Color jitter data augmentation	Node feature perturbation (adding noise to or randomly scaling node attributes)
Image flip data augmentation	No direct analog needed (GNNs are permutation-invariant by design) OR random edge rewiring
Channel dropout	Node feature/embedding dimension dropout (dropping out channels of node representations)
Zero padding edges	Adding virtual/dummy nodes to low-degree nodes to normalize neighborhood sizes
ResNet skip connections	Residual GNN connections (same concept: $\mathbf{h}^{(k+1)} = \mathbf{h}^{(k)} + f(\mathbf{h}^{(k)})$)
Blurring an image	Laplacian smoothing / neighbor feature averaging (basic GNN aggregation inherently does this)
Image inpainting (predicting missing pixel values)	Predicting missing values of nodes



Brief Explanations

- **Semantic segmentation ↔ Node-level prediction:** Both produce a prediction for each local element (pixel or node) rather than a global prediction.
- **Color jitter ↔ Node feature perturbation:** Both modify the feature values without changing the underlying structure (adjacency/spatial layout).
- **Image flip ↔ No direct analog:** Image flips exploit spatial symmetry. GNNs are already invariant to node ordering, so permuting node indices has no effect.
- **Blurring ↔ Laplacian smoothing:** Blurring averages a pixel with its neighbors; GNN message passing naturally averages/aggregates neighbor features, achieving a similar smoothing effect.
- **Image inpainting ↔ Missing node values:** Both involve predicting unknown values at certain locations based on known surrounding context.



- ⌚ Question 2(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?

Thought process: Synthesized multiple strategies for handling incomplete node data.

The user is asking about how to handle missing node features in a graph-level classification problem. This is a practical question about dealing with incomplete data in GNNs.

Let me think through several approaches:

1. **Imputation before GNN**: Fill in missing values with some placeholder or estimated values before feeding to the GNN
 - Use zeros, mean values, or learned embeddings for missing nodes
 - Use a separate model to predict missing values first
2. **Masking approach**: Use a mask to indicate which nodes have missing values
 - Add a binary indicator feature to each node (1 = missing, 0 = observed)
 - Let the GNN learn to handle missing values
3. **Let GNN propagate information**: Since GNNs aggregate neighbor information, nodes with missing values can still receive useful information from their neighbors through message passing
4. **Two-stage approach**: First predict missing node values (like the node imputation task mentioned in 2f), then do graph classification
5. **Modified aggregation**: Modify the aggregation function to only consider nodes with known values

Let me structure a clear answer.

Question 2g: Handling Missing Node Values in Graph-Level Classification

Approaches to Handle Missing Node Features

Approach 1: Placeholder Imputation

Replace missing node features with a default value before running the GNN:

- **Zero vector:** Set missing features to $\mathbf{0}$
- **Mean imputation:** Use the average feature vector from nodes with known values
- **Learned "missing" embedding:** Learn a special embedding vector specifically for missing nodes

$$\mathbf{x}_i = \begin{cases} \mathbf{x}_i & \text{if observed} \\ \mathbf{x}_{\text{default}} & \text{if missing} \end{cases}$$

Approach 2: Add a "Missing" Indicator Feature

Concatenate a binary mask feature to each node's representation:

$$\mathbf{x}'_i = [\mathbf{x}_i \parallel m_i]$$

where $m_i = 1$ if node i's features are missing, $m_i = 0$ otherwise.

This allows the GNN to **learn to treat missing nodes differently** from observed nodes.

Approach 3: Let Message Passing Propagate Information

GNNs naturally help here! Even if a node's features are missing:

- It can **receive information from neighbors** with known features
 - After several layers, the missing node's embedding becomes informed by its local graph structure and neighbors' features
 - The node still contributes structural information (its connectivity pattern)
-

Approach 4: Two-Stage Prediction

1. **First**, train a GNN to predict the missing node values (node-level regression/classification)
 2. **Then**, use the completed graph for graph-level classification
-

Recommended Strategy

Combine placeholder values (e.g., zeros or learned embedding) with a missing indicator feature, then let GNN message pass

This works because:

- The mask tells the model which nodes are uncertain
- Message passing allows missing nodes to "borrow" information from their neighborhood
- The global pooling for graph classification aggregates all nodes, so partial information is still useful

Question 2(h): Consider the graph neural net architecture that 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 then 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?

Thought process: Dissected GNN architecture scaling, weight distribution, and directed edge incorporation.

Let me work through each part of question 2(h) about the GNN architecture with node, edge, and global state representations.

Part (i): Effect of doubling nodes

The question asks about a graph with only node representations (no edge features, no global state).

Number of learned weights:

In a GNN, the learned weights are in the update functions (MLPs) that:

- Transform node features
- Aggregate neighbor messages
- Potentially transform the output

These weight matrices have fixed dimensions determined by the feature dimensions (hidden sizes), NOT by the number of nodes. The same learned weights are applied to every node.

So if we double the number of nodes: **The number of learned weights stays the same.**

Computation with same average degree:

Let's say we have n nodes with average degree d .

- Each node aggregates from $\sim d$ neighbors
- Total aggregation operations: $O(n \times d) = O(n \times d)$
- If we double n to $2n$, and d stays the same: $O(2n \times d)$

So computation **doubles** (scales linearly with number of nodes).

Computation with fully connected graph:

If the graph is fully connected:

- Each node is connected to all other nodes
- Degree $d = n - 1 \approx n$
- Total computation: $O(n \times n) = O(n^2)$
- If we double n to $2n$: $O((2n)^2) = O(4n^2)$

So computation **quadruples** (scales quadratically).

Part (ii): Where are learned weights?

Looking at the architecture diagram described:

- $f_{\text{U$}}$: Update function for global state

- $\$f_V\$$: Update function for nodes
- $\$f_E\$$: Update function for edges
- $\$\\rho\$$: Pooling/aggregation function

The learned weights are in:

1. The update functions $\$f_U\$$, $\$f_V\$$, $\$f_E\$$ (typically MLPs)

2. Potentially in aggregation functions, though these are often simple operations like sum, mean, or max.

`## Part (iii): Incorporating directed edges`

For directed edges:

- Currently, undirected edges mean information flows both ways equally
- For directed edges, we need to distinguish between incoming and outgoing edges

Approaches:

1. **Separate aggregation for incoming vs outgoing:** Create distinct aggregation mechanisms for messages received along incoming and outgoing edges
2. **Edge direction feature:** Introduce a directional feature for each edge to capture its orientation
3. **Asymmetric message passing:** Implement unidirectional message transmission, strictly following the edge's direction
4. **Duplicate with direction labels:** Treat edges between nodes as unique based on their specific directional relationship, potentially with distinct edge characteristics

Question 2h: GNN Architecture with Nodes, Edges, and Global State

Part (i): Effect of Doubling the Number of Nodes

Number of Learned Weights

The number of learned weights does not change.

GNN weights are in the update functions (MLPs) that transform features. These have fixed dimensions based on feature/hidden sizes, not the number of nodes. The same weights are shared and applied to every node via weight sharing.

Computation: Average Degree Stays the Same

Let n = number of nodes, d = average degree.

Quantity	Before	After ($2n$ nodes)
Nodes	n	$2n$
Edges	$\\sim nd/2$	$\\sim 2nd/2 = nd$
Computation per node	$O(d)$	$O(d)$
Total computation	$O(nd)$	$O(2nd)$

Computation doubles (linear scaling with number of nodes)

Computation: Fully Connected Graph

In a fully connected graph, each node connects to all others, so degree $d = n - 1 \approx n$.

Quantity	Before	After (2n nodes)
Nodes	n	2n
Degree	$n - 1$	$2n - 1$
Edges	$\sim n^2/2$	$\sim (2n)^2/2 = 2n^2$
Total computation	$O(n^2)$	$O(4n^2)$

Computation quadruples (quadratic scaling with number of nodes)

Part (ii): Location of Learned Weights/Parameters

The learned weights are in the **update functions**:

Function	Role	Learned Parameters
f_V	Node update function	MLP weights that transform aggregated neighbor/edge/global information into new node embeddings
f_E	Edge update function	MLP weights that transform endpoint node features and global state into new edge embeddings
f_U	Global state update function	MLP weights that transform pooled node/edge information into new global state

The **pooling/aggregation functions** (ρ) are typically **not learned**—they're usually fixed operations like sum, mean, or max.

Learned weights are in f_V , f_E , and f_U (the update MLPs)

Part (iii): Incorporating Directed Edges

Several approaches to handle directed edges:

Approach 1: Asymmetric Message Passing

Only pass messages in the direction of the edge—from source node to target node. A node only aggregates from its **incoming** edges.

Approach 2: Separate Aggregations for In/Out Edges

Compute two separate aggregations for each node: $\text{h}_i^{in} = \text{Agg}_{in}(N_i)$ $\text{h}_i^{out} = \text{Agg}_{out}(N_i)$

Then combine them in the update function.

Approach 3: Edge Direction Feature

Add a direction indicator to the edge features:

- For edge $i \rightarrow j$: include feature indicating "outgoing from i " or "incoming to j "
- Could use a one-hot encoding: [1, 0] for forward, [0, 1] for backward

Approach 4: Treat as Two Separate Edge Types

Use different weight matrices for messages along forward vs. backward edges.

Use asymmetric message passing (source \rightarrow target only) or separate aggregation functions for incoming vs. outgoing neighbors.

Question 3(a): 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. 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 n_i neighbors of node i are denoted by $m^{l-1}_{j,i}$ where j ranges from 1 to n_i . We use w with subscripts to denote learnable scalar weights. If there's no superscript, the weights are shared across layers. 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. (i) $s^l_i = w_1 * s^{l-1}_i + w_2 * (1/n_i) * \sum_{j=1}^{n_i} m^{l-1}_{j,i}$ (ii) $s^l_i = \max(w^{l-1}_1 * s^{l-1}_i, w^{l-1}_2 * m^{l-1}_{i,1}, w^{l-1}_3 * m^{l-1}_{i,2}, \dots, w^{l-1}_{n_i+1} * m^{l-1}_{i,n_i})$ where the max acts component-wise on the vectors. (iii) $s^l_i = \max(w^{l-1}_1 * s^{l-1}_i, w^{l-1}_2 * m^{l-1}_{i,1}, w^{l-1}_2 * m^{l-1}_{i,2}, \dots, w^{l-1}_{n_i} * m^{l-1}_{i,n_i})$ where the max acts component-wise on the vectors.

Thought process: Evaluated GNN update functions for permutation invariance and weight sharing.

The question asks me to determine which update functions are valid for a GNN on an undirected graph with no edge labels. The key properties that must be respected are:

1. **Permutation invariance of neighbors**: Since the graph is undirected and has no edge labels, the ordering of neighbors is arbitrary. The function must produce the same output regardless of how we order/index the neighbors.
2. **Weight sharing**: The same function (with the same weights) must be applicable to all nodes, regardless of their degree.

Let me analyze each option:

```
## (i) $s^l_i = w_1 s^{l-1}_i + w_2 \frac{1}{n_i} \sum_{j=1}^{n_i} m^{l-1}_{j,i}$
```

- The self-message term $w_1 s^{l-1}_i$ is fine
- The neighbor aggregation uses mean pooling: $\frac{1}{n_i} \sum_{j=1}^{n_i} m^{l-1}_{j,i}$
- Mean pooling is permutation invariant - reordering the neighbors doesn't change the sum
- The weights w_1, w_2 are fixed (no superscript, so shared across layers)
- This works for any number of neighbors n_i

Valid ✓

```
## (ii) $s^l_i = \max(w^{l-1}_1 s^{l-1}_i, w^{l-1}_2 m^{l-1}_{i,1}, w^{l-1}_3 m^{l-1}_{i,2}, \dots, w^{l-1}_{n_i+1} m^{l-1}_{i,n_i})$
```

Issues:

- Each neighbor message gets a **different weight** ($w_2, w_3, \dots, w_{n_i+1}$)

- Neighbor ordering directly impacts the computation
- Swapping neighbor indices changes the applied weights
- This breaks fundamental requirements of permutation invariance
- Varying weights based on neighbor position creates inconsistent node representations

Invalid ✗ - Not permutation invariant;

The layer-specific weight w^{l-1}_1 introduces nuanced parameter handling across neural network layers. This suggests a more complex weight allocation strategy that adapts dynamically during network

processing, potentially allowing for more flexible feature extraction while maintaining a structured computational approach.

Question 3a: Valid GNN Update Functions

For a valid GNN update function on an undirected graph, we need:

1. **Permutation invariance:** The function must produce the same output regardless of how we order/index the neighbors (since the graph is undirected with no edge labels)
2. **Weight sharing compatibility:** The same weights must work for nodes with different degrees

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

Valid ✓

Why:

- The mean aggregation $\frac{1}{n_i} \sum_{j=1}^{n_i} m_{i,j}^{l-1}$ is **permutation invariant** — reordering the neighbors doesn't change the sum
- Only two fixed weights (w_1, w_2) are used regardless of node degree
- Works for any number of neighbors n_i

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

Invalid ✗

Why:

- Each neighbor gets a **different weight** ($w_2, w_3, \dots, w_{n_i+1}$)
- **Violates permutation invariance:** Swapping neighbor 1 and neighbor 2 would change the result since they're multiplied by different weights
- **Violates weight sharing:** The number of weights needed ($n_i + 1$) depends on node degree, so nodes with different degrees would require different numbers of parameters

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

Valid ✓

Why:

- All neighbor messages are multiplied by the **same weight** w_2
- The max operation is **permutation invariant** — the maximum value doesn't depend on ordering
- Only two weights are used (w_1 for self, w_2 for all neighbors), regardless of degree
- The layer-specific weight w_1^l is fine — each layer simply has its own learned value

Summary

Function	Valid?	Reason
(i) Mean aggregation	✓	Permutation invariant, fixed weights
(ii) Max with different weights per neighbor	✗	Not permutation invariant; weight count depends on degree
(iii) Max with same weight for all neighbors	✓	Permutation invariant, fixed weights

- Question 3(b): We are given a simple graph with 5 nodes (labeled 1-5) on which we want to train a GNN. The goal is binary node classification (classifying nodes as type 1 or 0). We hold back nodes 1 and 4 for testing while using the rest for training. The surrogate loss is average binary cross-entropy loss.

nodes	1	2	3	4	5
y_i (ground truth)	0	1	1	1	0
ŷ_i (predicted prob)	a	b	c	d	e

Compute the training loss at the end of training.

The formula for average binary cross-entropy loss is:

$$(1/n) * \sum_x [y(x) * \log(1/\hat{y}(x)) + (1-y(x)) * \log(1/(1-\hat{y}(x)))]$$

where x ranges over training points and $\hat{y}(x)$ is the predicted probability that the label is 1.

Thought process: Parsed training nodes, simplified cross-entropy formula, computed loss.

Let me carefully parse this problem.

We have 5 nodes, and we're doing binary node classification.

Training nodes: 2, 3, 5 (the rest, excluding held-back nodes 1 and 4)

Test nodes: 1, 4 (held back)

Ground truth labels:

- Node 1: $y = 0$ (test)
- Node 2: $y = 1$ (training)
- Node 3: $y = 1$ (training)
- Node 4: $y = 1$ (test)
- Node 5: $y = 0$ (training)

Predicted probabilities:

- Node 1: $\hat{y} = a$
- Node 2: $\hat{y} = b$
- Node 3: $\hat{y} = c$
- Node 4: $\hat{y} = d$
- Node 5: $\hat{y} = e$

The training loss only considers training nodes (2, 3, 5).

The formula given is:

$$(1/n) * \sum_x [y(x) * \log(1/\hat{y}(x)) + (1-y(x)) * \log(1/(1-\hat{y}(x)))]$$

Let me simplify this formula:

- $y(x) * \log(1/\hat{y}(x)) = y(x) * (-\log(\hat{y}(x))) = -y(x) * \log(\hat{y}(x))$
- $(1-y(x)) * \log(1/(1-\hat{y}(x))) = (1-y(x)) * (-\log(1-\hat{y}(x))) = -(1-y(x)) * \log(1-\hat{y}(x))$

So the total is:

$$(1/n) * \sum_x [-y(x) * \log(\hat{y}(x)) - (1-y(x)) * \log(1-\hat{y}(x))]$$

This is the standard binary cross-entropy loss calculation, which penalizes incorrect predictions differently based on the ground truth label.

I'll break down the loss contributions for each training node. For Node 2, with a true label of 1 and predicted probability b, the loss is $-\log(b)$. Similarly, for Node 3, with a true label of 1 and predicted probability c, the loss is $-\log(c)$.

For Node 5, I'll continue calculating its loss contribution in the next step.

The total training loss is computed by averaging these individual node losses, specifically $-(1/3) * \log(bc(1-e))$. This formulation elegantly captures the model's predictive performance across different nodes by logarithmically penalizing incorrect probability estimates.

The general formula confirms this approach, showing how each node's loss depends on its true label and predicted probability, scaled by the total number of training examples.

Multiple mathematically equivalent representations exist, each highlighting different aspects of the loss calculation. The key insight is maintaining a consistent logarithmic penalty that reflects the model's confidence and accuracy.

Question 3b: Computing the Training Loss

Identifying Training vs Test Nodes

- Training nodes:** 2, 3, 5 ($n = 3$)
- Test nodes:** 1, 4 (held back for evaluation)

Node	Ground Truth y_i	Predicted \hat{y}_i	Set
1	0	a	Test
2	1	b	Train
3	1	c	Train
4	1	d	Test
5	0	e	Train

Computing Loss for Each Training Node

The binary cross-entropy formula: $\mathcal{L} = \frac{1}{n} \sum_x \left[y(x) \log \frac{1}{\hat{y}(x)} + (1-y(x)) \log \frac{1}{1-\hat{y}(x)} \right]$

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

$$\mathcal{L}_2 = 1 \cdot \log \frac{1}{b} + 0 \cdot \log \frac{1}{1-b} = \log \frac{1}{b}$$

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

$$\mathcal{L}_3 = 1 \cdot \log \frac{1}{c} + 0 \cdot \log \frac{1}{1-c} = \log \frac{1}{c}$$

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

$$\mathcal{L}_5 = 0 \cdot \log \frac{1}{e} + 1 \cdot \log \frac{1}{1-e} = \log \frac{1}{1-e}$$

Final Training Loss

$$\mathcal{L}_{\text{train}} = \frac{1}{3} \left[\log \frac{1}{b} + \log \frac{1}{c} + \log \frac{1}{1-e} \right]$$

Equivalent forms:

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

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

Question 3(c): Suppose we use the following update rule for the internal state of nodes at layer l: $s^l_i = s^{l-1}_i + W_1 * [\sum_{j=1}^n n_j \tanh(W_2 * m^{l-1}_{i,j})] / n_i$ where \tanh acts element-wise. (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^{l-1}_i itself as the message m^{l-1} going to all of node i's neighbors, please write out the update rules for the graph (5 nodes where: node 1 connects to 2,4,5; node 2 connects to 1,4,5; node 3 connects to 5; node 4 connects to 1,2; node 5 connects to 1,2,3) for nodes $i=2$ and $i=3$ in terms of information from earlier layers. Expand out all sums.

Thought process: Analyzed GNN architecture, computed matrix dimensions, and deduced node update equations.

Let me work through each part of this question about the GNN update rule.

The update rule is:

$$s^l_i = s^{l-1}_i + W_1 \frac{[\sum_{j=1}^n n_j \tanh(W_2 m^{l-1}_{i,j})]}{n_i}$$

Part (i): Design Patterns

Looking at the equation:

- $s^l_i = s^{l-1}_i + (\text{something})$

This is clearly a **residual connection** - the output is the input plus some transformation. This is the classic ResNet skip connection pattern.

Is there batch normalization? Batch normalization involves normalizing activations across a batch using mean and variance statistics. The division by n_i is just mean aggregation over neighbors, not batch normalization here.

Part (ii): Dimensions of W_1

Let's trace through the dimensions:

- s is d-dimensional, so $s^l_i \in \mathbb{R}^d$
- $m^{l-1}_{i,j}$ is also d-dimensional (since messages are typically the same dimension as states)
- W_2 has k rows, and it multiplies $m^{l-1}_{i,j}$ which is d-dimensional
- So $W_2 \in \mathbb{R}^{k \times d}$
- $W_2 m^{l-1}_{i,j} \in \mathbb{R}^{k \times d}$
- $\tanh(W_2 m^{l-1}_{i,j}) \in \mathbb{R}^k$
- The sum and mean aggregation keep it at \mathbb{R}^k
- W_1 multiplies this k-dimensional vector to produce s^l_i which is d-dimensional
- $W_1 \in \mathbb{R}^{d \times k}$

Part (iii): Update Rules for Nodes 2 and 3

I'll analyze the graph's connectivity and how it impacts message passing. Node 2 has connections to nodes 1, 4, and 5, which means its message aggregation will involve states from these three

neighboring nodes. The update process will involve transforming and averaging messages from these connected nodes, then applying the residual connection to update its own state.

For node 3, the connections are more limited, with only a direct link to node 5. This means its message aggregation will be simpler, drawing information solely from node 5's current representation. The update mechanism will involve mapping node 5's state through the appropriate weight matrix and then integrating it into node 3's representation.

The graph structure reveals a complex interconnection pattern, with nodes having varying degrees of connectivity. Node 2 appears most connected, linking to multiple other nodes, which suggests it might play a central role in information propagation across the network.

Analyzing the node connections systematically shows a non-trivial graph topology where each node has a unique neighborhood configuration. This structural complexity will significantly impact how information flows and transforms during graph neural network computations.

The mathematical representation demonstrates how node states are updated through a weighted aggregation of neighboring node states, using hyperbolic tangent transformations to capture non-linear interactions between connected nodes.

For node 3, the state update involves averaging the transformed states of its two neighbors, weighted by a specific coefficient. The calculation reflects the graph's local connectivity and the node's specific neighborhood structure.

The graph's topology reveals a complex interconnection pattern, with each node having multiple connections that influence its state evolution through a carefully designed neural network mechanism.

Question 3c: GNN Update Rule Analysis

Part (i): Design Patterns

Looking at the update rule: $s_i^l = s_i^{l-1} + W_1 \frac{1}{n_i} \sum_{j=1}^{n_i} \tanh(W_2 m_{i,j}^{l-1})$

This has the form: **output = input + f(input)**

Residual connection ✓

The previous state s_i^{l-1} is added directly to the aggregated neighbor information, which is the classic skip connection pattern from ResNets.

Not batch normalization — the division by n_i is simply mean aggregation over neighbors, not normalization using batch statistics (mean and variance across a batch).

Part (ii): Dimensions of W_1

Let's trace dimensions through the computation:

Quantity	Dimension	Reasoning
s_i^l	d	Given: state is d -dimensional
$m_{i,j}^{l-1}$	d	Messages have same dimension as states

Quantity	Dimension	Reasoning
W_2	$k \times d$	Given: W_2 has k rows, multiplies d -dimensional vector
$W_2 m_{i,j}^{l-1}$	k	Matrix-vector product
$\tanh(W_2 m_{i,j}^{l-1})$	k	Element-wise operation preserves dimension
$\frac{1}{n_i} \sum_j \tanh(\cdot)$	k	Sum of k -dimensional vectors
$W_1 \cdot (\text{aggregated})$	d	Must match s_i^{l-1} for the addition
$W_1 \in \mathbb{R}^{d \times k}$		

Part (iii): Update Rules for Nodes 2 and 3

From Figure 3, the graph connectivity is:

Node	Neighbors
1	2, 5
2	1, 3, 4, 5
3	2, 4
4	2, 3, 5
5	1, 2, 4

Since $m_{i,j}^{l-1} = s_j^{l-1}$ (the message from neighbor j is its own state):

Node 2 (neighbors: 1, 3, 4, 5; degree $n_2 = 4$):

$$s_2^l = s_2^{l-1} + W_1 \frac{\tanh(W_2 s_1^{l-1}) + \tanh(W_2 s_3^{l-1}) + \tanh(W_2 s_4^{l-1}) + \tanh(W_2 s_5^{l-1})}{4}$$

Node 3 (neighbors: 2, 4; degree $n_3 = 2$):

$$s_3^l = s_3^{l-1} + W_1 \frac{\tanh(W_2 s_2^{l-1}) + \tanh(W_2 s_4^{l-1})}{2}$$

- Question 6(a): In the Newton-Schulz implementation for the Muon optimizer, we initially scale the Frobenius norm to be at most $\sqrt{3}$. Can you explain why we choose this particular scaling? Comment on why this is better than using 1. (Hint: Inspect the roots of the cubic polynomial. What is the connection between the roots and the convergence properties of the singular values?)

Thought process: Analyzed polynomial roots and convergence basin optimization.

This question is about the Newton-Schulz iteration used in the Muon optimizer to compute the matrix square root inverse (or orthogonalize matrices). Let me think through this carefully.

The Newton-Schulz iteration is used to compute $A^{-1/2}$ or to orthogonalize a matrix (find the closest orthogonal matrix). The iteration typically looks like:

$$X_{k+1} = X_k (aI + bX_k^T X_k + c(X_k^T X_k)^2)$$

or for computing the polar decomposition (orthogonalization):

$$X_{k+1} = X_k \cdot \frac{1}{2}(3I - X_k^T X_k)$$

This is the basic Newton-Schulz iteration for computing the polar factor.

For a more general cubic polynomial iteration:

$$X_{k+1} = X_k (aI + bX_k^T X_k + c(X_k^T X_k)^2)$$

The iteration acts on singular values. If the singular values of X_k are σ_i , then after one iteration, the singular values become:

$$\sigma'_i = \sigma_i \cdot p(\sigma_i^2)$$

where $p(t) = a + bt + ct^2$ is the polynomial.

For convergence to orthogonalization (singular values $\rightarrow 1$), we need:

$$f(\sigma) = \sigma \cdot p(\sigma^2) \rightarrow 1 \text{ as we iterate}$$

The fixed point should be $\sigma = 1$, so $f(1) = 1 \cdot p(1) = 1$, meaning $p(1) = 1$.

For convergence, we need $|f'(1)| < 1$, and faster convergence means smaller $|f'(1)|$.

I'm investigating the standard Newton-Schulz iteration's convergence properties. The specific polynomial transformation $f(\sigma) = \frac{3\sigma - \sigma^3}{2}$ suggests a nuanced behavior of singular values during the orthogonalization process. The iteration's convergence depends on carefully constraining the singular values within a specific range that ensures the transformation moves towards an orthogonal matrix.

The critical interval is $(0, \sqrt{3})$, which emerges from analyzing the iteration's behavior. When σ approaches $\sqrt{3}$, the transformation $f(\sigma)$ becomes zero, marking a key boundary condition. The roots of the iteration polynomial - zero and $\pm\sqrt{3}$ - provide insight into the dynamical system's structure.

The basin of attraction for the fixed point $\sigma = 1$ depends on scaling the matrix's Frobenius norm. By constraining $\|X\|_F \leq \sqrt{3}$, we ensure that singular values remain within the

convergence region. This scaling mechanism guarantees the Newton-Schulz iteration's stability, particularly when dealing with matrices having a single dominant singular value.

The relationship between Frobenius norm and singular values reveals an important constraint. When a matrix has only one singular value, the Frobenius norm directly corresponds to that singular value. This means scaling ensures all singular values stay below the critical boundary of $\sqrt{3}$, which is crucial for algorithmic convergence.

Critically, the Frobenius norm provides an upper bound that inherently captures the matrix's spectral characteristics. By maintaining this norm below $\sqrt{3}$, we create a robust condition that prevents divergence in iterative matrix computations.

The convergence dynamics become particularly interesting near $\sigma = 1$, where the iteration's derivative reaches zero, suggesting a higher-order convergence mechanism that transcends linear approximations.

For singular values less than 1, the iteration drives values upward, progressively approaching the fixed point. Conversely, for singular values between 1 and $\sqrt{3}$, the iteration pulls values downward, converging toward 1.

This behavior establishes a robust basin of attraction spanning $(0, \sqrt{3})$, with 1 serving as the stable equilibrium point. The symmetric convergence properties around 1 highlight the elegant mathematical structure of the Newton-Schulz method.

By permitting singular values up to $\sqrt{3}$, we gain flexibility in handling matrices with broader dynamic ranges. Scaling to $\sqrt{3}$ prevents unnecessary compression, allowing smaller singular values to maintain more meaningful relative magnitudes during the iterative refinement process.

Consider a matrix with hierarchically distributed singular values. Scaling to $\sqrt{3}$ ensures that smaller singular values retain more of their original proportional relationships, potentially accelerating overall convergence and preserving more nuanced structural information.

Critically, the iteration dynamics for small singular values exhibit an intriguing behavior: they grow approximately linearly with a rate greater than 1, which paradoxically supports faster convergence to the target value.

The key insight emerges from understanding how different scaling approaches impact the matrix's condition number and the relative dynamics of its singular values.

By choosing $\sqrt{3}$ as the scaling factor, we achieve multiple strategic advantages: ensuring all singular values remain bounded, maintaining more of the original matrix's characteristics, and providing an optimal convergence mechanism for matrices that are already close to orthogonal.

The polynomial iteration's roots and scaling strategy are carefully designed to balance preservation of matrix structure with efficient transformation, demonstrating a nuanced approach to numerical optimization.

The cubic polynomial's roots at 0 and $\pm\sqrt{3}$ reveal a precise mathematical framework for controlling matrix convergence, with each root representing a critical point in the transformation process.

I see three key advantages to using $\sqrt{3}$ as the convergence boundary:

1. Maximizing the convergence basin by utilizing the full range of singular values
2. Enabling more nuanced scaling that preserves computational efficiency
3. Accelerating convergence by allowing larger singular values to rapidly approach the fixed point while maintaining sensitivity for smaller values

This approach balances computational precision with adaptive scaling, ensuring optimal performance across diverse matrix transformations.

Question 6a: Why Scale Frobenius Norm to $\sqrt{3}$?

The Newton-Schulz Iteration

The Newton-Schulz iteration for matrix orthogonalization applies the following update: $X_{k+1} = X_k \cdot \frac{1}{\sigma^2} (3I - X_k^T X_k)$

This iteration acts on singular values. If σ is a singular value, it evolves as: $f(\sigma) = \sigma \cdot \frac{3 - \sigma^2}{\sigma^2} = \frac{3\sigma - \sigma^3}{\sigma^2}$

Finding the Roots of the Cubic

Setting $f(\sigma) = 0$: $\frac{\sigma(3 - \sigma^2)}{\sigma^2} = 0$

The roots are: $\boxed{\sigma = 0, \sigma = +\sqrt{3}, \sigma = -\sqrt{3}}$

Convergence Analysis

The fixed point where $f(\sigma) = \sigma$ is at $\sigma = 1$ (orthogonal matrices have all singular values equal to 1).

Basin of attraction: The iteration converges to 1 for initial singular values in $(0, \sqrt{3})$:

Region	Behavior
$0 < \sigma < 1$	$f(\sigma) > \sigma$ — singular values increase toward 1
$\sigma = 1$	$f(\sigma) = \sigma$ — fixed point
$1 < \sigma < \sqrt{3}$	$f(\sigma) < \sigma$ — singular values decrease toward 1
$\sigma \geq \sqrt{3}$	$f(\sigma) \leq 0$ — diverges/fails

Why $\sqrt{3}$ is the Right Choice

Scaling the Frobenius norm to at most $\sqrt{3}$ guarantees: $|X|F \leq \sqrt{3} \implies \sigma_{\max} \leq \sqrt{3}$

This ensures **all singular values lie within the convergence basin** $(0, \sqrt{3})$.



Why $\sqrt{3}$ is Better Than 1

Aspect	Scaling to 1	Scaling to $\sqrt{3}$
Uses convergence region	Only $(0, 1]$ — half the basin	Full $(0, \sqrt{3})$ — entire basin
Scaling aggressiveness	Over-compresses matrices	Minimal necessary scaling
Small singular values	Become even smaller, slower convergence	Remain larger, faster convergence
Already near-orthogonal matrices	Unnecessarily scaled down	Less distortion needed

Key insight: If we scale to 1, singular values that were slightly above 1 get compressed below 1, then must iterate back up to 1. With $\sqrt{3}$, values in $(1, \sqrt{3})$ naturally converge down to 1 without unnecessary scaling.

Summary

$\sqrt{3}$ is chosen because it is the positive root of the cubic polynomial, defining the boundary of the convergence basin. Using $\sqrt{3}$ instead of 1 results in better scaling and faster convergence.

Question 6(b): Note that Muon requires that parameters are 2D matrices of shape $d_{out} \times d_{in}$. However, convolutional kernels have shape $c_{out} \times c_{in} \times k \times k$, where c denotes number of channels and k is kernel size. Modern implementations of convolutional layers will (implicitly) transform an input image x of shape $c_{in} \times h \times w$ to x' such that each column of x' has size $c_{in} \cdot k \cdot k$ and corresponds to one flattened "receptive field" of the image. Given this fact, how do we modify the convolutional kernel into a $d_{out} \times d_{in}$ matrix C such that the output of the convolutional layer can be expressed as Cx' ?

Thought process: Deconstructed kernel dimensions and derived reshaping strategy.

The question is asking about how to reshape a convolutional kernel to work with the im2col operation.

Let me think about this:

1. The convolutional kernel has shape: $c_{out} \times c_{in} \times k \times k$
2. The im2col operation transforms the input image x of shape $c_{in} \times h \times w$ into x' where:
 - Each column of x' has size $c_{in} \cdot k \cdot k$ (a flattened receptive field/patch)
 - The number of columns depends on the output spatial dimensions (let's call it $h' \times w'$ for the output height and width)
 - So x' has shape $(c_{in} \cdot k \cdot k) \times (h' \times w')$
3. The output of the convolution should have shape $c_{out} \times h' \times w'$, which when flattened appropriately is $c_{out} \times (h' \cdot w')$
4. For the matrix multiplication $C \cdot x'$ to work:
 - x' is $(c_{in} \cdot k \cdot k) \times (h' \cdot w')$
 - We want output of shape $c_{out} \times (h' \cdot w')$
 - So C must have shape $c_{out} \times (c_{in} \cdot k \cdot k)$
5. To reshape the kernel from $c_{out} \times c_{in} \times k \times k$ to $c_{out} \times (c_{in} \cdot k \cdot k)$:
 - We simply flatten/reshape the last three dimensions
 - $d_{out} = c_{out}$
 - $d_{in} = c_{in} \cdot k \cdot k$

I'll transform the kernel by collapsing the channel, kernel height, and kernel width dimensions into a single dimension. This creates a 2D matrix ready for matrix multiplication with the im2col transformed input.

Question 6b: Reshaping Convolutional Kernels for Muon

Understanding the im2col Operation

Input transformation (im2col):

- Original input x : shape $c_{in} \times h \times w$
- Transformed input x' : shape $(c_{in} \cdot k \cdot k) \times n_{patches}$

where $n_{\text{patches}} = h' \times w'$ is the number of output spatial locations (patches).

Each column of x' is one flattened receptive field of size $c_{\text{in}} \cdot k \cdot k$.

Deriving the Required Matrix Shape

For the convolution to be expressed as Cx' :

Matrix	Shape
x'	$(c_{\text{in}} \cdot k \cdot k) \times n_{\text{patches}}$
Output	$c_{\text{out}} \times n_{\text{patches}}$
C	Must be $c_{\text{out}} \times (c_{\text{in}} \cdot k \cdot k)$

The Kernel Reshape

Original kernel shape: $c_{\text{out}} \times c_{\text{in}} \times k \times k$

Required matrix shape: $c_{\text{out}} \times (c_{\text{in}} \cdot k \cdot k)$

$$C = \text{reshape}(\text{kernel}, (c_{\text{out}}, c_{\text{in}} \cdot k \cdot k))$$

In other words: Flatten the last three dimensions (c_{in}, k, k) into a single dimension while keeping c_{out} as the first dimension.

Matching to Muon's Requirements

Muon Parameter	Convolutional Equivalent
d_{out}	c_{out}
d_{in}	$c_{\text{in}} \cdot k \cdot k$

In PyTorch:

```
# kernel has shape (c_out, c_in, k, k)
C = kernel.reshape(c_out, c_in * k * k)
# C has shape (c_out, c_in * k * k) = (d_out, d_in)
```

Why This Works

Each row of C corresponds to one output channel's filter, flattened into a vector of length $c_{\text{in}} \cdot k \cdot k$. When we compute Cx' :

- Each row of C dots with each column of x' (a flattened patch)

- This produces exactly the same result as sliding the 3D filter across the input
- The output has shape $c_{out} \times n_{patches}$, which can be reshaped back to $c_{out} \times h' \times w'$