Homework 6: Gemini Special Participation A

Arnav Dalal

EECS 182 - Fall 2025

2. Graph Dynamics and GNN Concepts

(a) **Output of j-th node at layer k:**
Let the input matrix at layer 0 be $X^{(0)} = I$ (the identity matrix), where the i-th row corresponds to the one-hot encoding of node i. The operation at each layer is a weighted sum of neighbors. In matrix form, with adjacency matrix A, the update is linear: $X^{(k+1)} = AX^{(k)}$ (assuming standard matrix multiplication propagation). The output of node j at layer k corresponds to the column j of the matrix A^k.

Notes: Gemini is able to explain this problem using a zero-shot approach. The model is able to reason to the recurrence relation for the graph at each timestep, and then extrapolate the answer from there. While the formatting for the answer is not consistent, this is due to a lack of context from my input most likely.

(b) **Proof: Output counts paths of length k:**
Base Case ($k = 0$): The matrix $A^0 = I$. $(A^0)_{ij} = 1$ if $i = j$ and 0 otherwise. This is correct because there is exactly 1 path of length 0 from a node to itself (staying put).

Inductive Step: Assume $(A^k)_{ij}$ equals the number of paths of length k from i to j. Consider layer $k + 1$. The value at node j is the sum of inputs from its neighbors l:

$$(A^{k+1})_{ij} = \sum_l (A^k)_{il} A_{lj}$$

Since $(A^k)_{il}$ is the number of paths of length k from i to l, and $A_{lj} = 1$ represents an edge from l to j, multiplying them extends those paths by one edge to reach j. Summing over all intermediate nodes l counts all possible paths of length $k + 1$.

Notes: Gemini is able to get most of this question correct zero-shot, defining our base case for the induction proof and performing the inductive step correctly as well. The recurrence relation that the model concludes on is correct and then comes to the right conclusion that this summation is the definition for the number of paths of length k + 1 that start at the node and return to the node.

(c) **Update Function:**
The GNN described is linear. If $h_v^{(k)}$ is the activation vector of node v at layer k, the update function is:

$$h_v^{(k)} = \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)}$$

This is simple summation aggregation without any non-linear activation or learnable weight matrices.

Notes: While Gemini got this question right, it misunderstood parts of the question, incorporating an activation vector at each node, when there is no activation within the scope of this question. Other than this inconsistency, the model is able to complete this problem.

(d) **Max Aggregation Interpretation:**
If we replace sum with max, the output $(A^k)_{ij}$ becomes boolean (assuming unweighted graphs). It will be 1 if there exists *at least one* path of length k from i to j, and 0 otherwise. It represents **reachability** via a walk of exactly length k, rather than the count of such walks.

Notes: Gemini brings up a very interesting point, that when we use max(.) as our aggregation function, we get a graph representing reachability at each time step. This is very valuable in the context of algorithms, so seeing the model bring up this insight is great.

(e) **Molecule Problem:**

(i) **Graph Representation:**

- **Nodes:** Atoms.
- **Edges:** Chemical bonds (undirected).
- **Global State:** Molecule type (carbohydrate, protein, etc.).

Notes: The model is able to capture the overall architecture of of the GNN, but misses a crucial detail for the edges. It is correct that the edges represent chemical bonds; however, the graph that Gemini output doesn't have any weights on the edges. The solution for this situation is to put the bond length as the weights for each of the edges since this is necessary for making our predictions.

(ii) **Prediction:** To predict which bond breaks, we need an output for each edge. We can use the final learned representations of the two nodes connected by a bond (h_u, h_v) and pass them through an MLP (e.g., MLP(h_u, h_v, e_{uv})) to predict a score.

Notes: Gemini proposes using an external MLP to pass the features of each edge into along with the bond length to make a prediction on each edge. This doesn't specify how this classification is being done (i.e. softmax), so this is incomplete in devising a scheme for this problem. Moreover, there are two case scenarios that we need to predict in this case: (1) whether any bonds break and (2) If there is a bond breakage, then which one is most likely to break first. The solution proposed handles finding which bond will break, but this is under the assumption that a bond does in fact break. Without further specification with the current design, it is unclear how this will be done without a threshold of sorts.

Attempt 2: For every edge (bond) in the graph, construct an edge representation by concatenating the final embeddings of its two connected nodes (h_u, h_v) with the bond's features (length). Pass each edge representation through an MLP to generate a scalar "breakability" score. Finally, apply a Softmax or Argmax over all edge scores (combined with a learnable "no-break" score or threshold) to select the single bond with the highest probability of breaking, or to predict that none break.

Notes: This new answer addresses two issues with the original answer: (1) Aggregating and comparing scores across different edges and (2) accounting for none of the bonds breaking. The model mentions the use of softmax in this case to see which bond has the highest likelihood for breaking since softmax will return everything as a probability instead of the MLP outputs from before. Second, the model mentions the option of none of the bonds breaking, but no clear implementation detail is provided for that.

(iii) **Node Input Encoding:** One-hot encoding of the atomic element (e.g., C, H, O, N).

Notes: Zero-Shot: The desired solution here is to have some sort of learned embedding for the input; however, Gemini opts to use a one-hot encoding for each element. This is very vague given the context of what the question is asking.

Attempt 2:

To fully capture the available data, the node representation should include:

- The Atomic Element: (e.g., via one-hot encoding).
- The Molecule Type: This global feature is often concatenated to every node's feature vector to provide necessary context for the prediction.

Notes: The model brings up an issue with its own initial answer about only dealing with the atomic element, but doesn't mention a learnable embedding still, but now incorporating the global molecule type.

(f) **CNN vs GNN Analogies:**

CNN Operation	**GNN Analogy**
Image classification	Graph-level classification (e.g., molecule property)
Color jitter augmentation	Node feature perturbation / noise injection
Image flip augmentation	Graph rotation/isomorphism or permutation
Channel dropout	Node feature dropout
Zero padding	Virtual nodes or Zero-padding neighbors
ResNet skip connections	Residual connections ($h^{(l)} = h^{(l-1)} + f(...)$)
Blurring an image	Message passing (smoothing)
Pixel-wise segmentation	Node-level prediction

Notes: From this entire table, we can see that Gemini even hallucinated the entries in the table, leading to a different problem as a whole. However, even with this, the model is able to get the problems correct. For the image flip augmentation, a graph doesn't have orientation, so the answer that the model gave with graph rotation is a little vague. For the zero padding case, graphs don't need virtual nodes even if that would be the equivalent. Also the last question for predicting missing values of nodes isn't answered, but should be image inpainting.

(g) **Missing Node Values:**
If node values are missing: (1) Mask the missing nodes during loss calculation (if targets). (2) If inputs, replace with a learnable 'missing' token embedding. (3) Use the graph structure to impute values via message passing.

Notes: The solution includes both of the answers from the key, mentioning that we can include a learnable token embedding or imputing with mean values. The model also adds an extra answer in masking the missing nodes, which would then have to be reflected in the update step.

(h) **GNN Architecture Scaling:**

(i) **Doubling Nodes:**

- **Learned Weights:** No change. Weights are shared across all nodes.
- **Computation:** Scales linearly with nodes ($O(|V| + |E|)$).

Notes: The model gets both points right about the number of learned weights not changing and the computation doubling, but saying that it scales linearly with the number of nodes added. The model ignores the part of the question asking about a fully connected neural network example and doesn't provide an idea of how the computation scales with the number of nodes.

(ii) **Where are weights?** In the update functions (MLPs), shared across the graph. **Notes**: The model is able to get this correct fully, providing the specific example that the MLPs that are assigned for the updates functions inlcude the learnable weights.

(iii) **Directed Edges:** Split aggregation into two parts: one for incoming edges and one for outgoing edges, with separate weight matrices. **Notes**: Here, the model is able to provide an implementation detail that wasn't mentioned in the answer key. Two matrices for the split aggregation functions provides more detail to the handling of both incoming and outgoing edges and what the motivation for that setup is.

3. Graph Neural Networks

(a) **Validity of Update Functions:**

(i) **Invalid/Unclear:** If the indices imply specific ordering, it is invalid. If it's a fixed sum, it is valid.

(ii) **Invalid:** Uses weights $w_2, \ldots, w_{n+1}$ based on neighbor indices. Violates permutation invariance.

(iii) **Valid:** Uses the same weight w_2 for all neighbors inside a max aggregation. Respects permutation invariance.

Notes: The model gets the first part incorrect since the provided question is the standard averaging update rule. This rule is permutation invariant to the neighbors and can be dependent on the number of neighbors. The second part is correct since we can't assign a unique weight to each indexed neighbor. The third one fixes this issue and is explained by the model.

Attempt 2:

(i) **Valid:** This function uses a summation operation which is permutation invariant (the order of neighbors does not affect the sum), and it employs a fixed set of weights (w_1, w_2) allowing for parameter sharing regardless of node degree.

(ii) **Invalid:** This function violates permutation invariance by assigning specific weights ($w_2, w_3, \ldots$) to specific neighbor positions, meaning the output changes if neighbor indices are swapped. It also requires a variable number of parameters ($n_i + 1$) dependent on node degree, preventing weight sharing.

(iii) **Valid:** This function applies the same shared weight (w_2) to all neighbor messages before the aggregation, ensuring the set of values is independent of neighbor order. The max operator is symmetric (permutation invariant), and the parameter count remains fixed.

Notes: This time the model got this completely correct and provided significantly more detail for each of the answers. This is likely due to the fact that we are independently prompting about this questions instead of the whole document

(b) **Training Loss Calculation:**
Nodes 1 and 4 are held out. Training nodes are 2, 3, and 5.
From Table 1: Node 2 ($y = 1, \hat{y} = b$), Node 3 ($y = 1, \hat{y} = c$), Node 5 ($y = 0, \hat{y} = e$).

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

Notes: The model calculates the correct loss with the same steps as the solution, but didn't show work for how it got to the final loss equation. This is likely due to the difference between prompting each individual question and performing zero-shot inference on the whole problem set.

(c) **Update Rule Analysis:**
Rule: $s_i^l = s_i^{l-1} + W_1 \frac{\sum \tanh(W_2 m_{i,j}^{l-1})}{n_i}$

(i) **Design Pattern:** Residual Connection (skip connection).

(ii) **Matrix Dimensions:** W_2 projects $d \to k$. The sum is size k. To add to s_i^{l-1} (size d), W_1 must be $d \times k$.

(iii) **Manual Update Calculation (Nodes 2 & 3):**
Node 2 (Neighbors 1, 4, 5):

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

Node 3 (Neighbor 5):

$$s_3^l = s_3^{l-1} + W_1 (\tanh(W_2 s_5))$$

Notes: Looking at each of the individual parts, the model is able to correctly answer each one correctly, providing concise details necessary for understanding the line of reasoning. For part (i), because it is a multi-select question, the model doesn't provide further explanation beyond the fact that there is a skip connection. For part (ii) and (iii), the model provides a little more detail, but still is very concise overall.

4. Exploring Deep Learning Tooling

(a) **TensorBoard:** Good for local debugging and real-time visualization. Pros: Free, private, no setup. Cons: Hard to share, manages large experiments poorly.

(b) **Weights & Biases (WandB):** Cloud-based tracking. Pros: easy sharing, persistent storage, advanced hyperparameter comparison, team collaboration. Cons: Requires account/setup, internet access.

Notes: The model is able to analyze the pros and cons of each of the visualization tools, especially focusing on the aspects of tool setup, collaboration, and hyperparameter integration. Didn't mention searching by configurations or the ease of searching through runs on WandB compared to Tensorbaord.