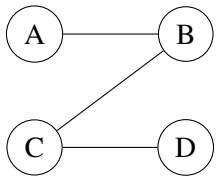
EECS 182 Deep Neural Networks
Fall 2025 Anant Sahai and Gireeja Ranade

Discussion 7

1. Graph Neural Network Forward Pass

Consider the following undirected graph G:



In this problem, we are going to work with an undirected graph without edge weights. We are imposing these limitations to make the problem simpler and to let us focus on the core ideas behind the forward pass. In practice, edges can be directed and have weights.

When a GNN layer is applied to a graph, it produces a new graph with the same topology as the original graph but with (potentially) different values in the nodes and the edges. After passing a graph through a number of these GNN layers, we can use the graph embedding for a variety of downstream tasks. In this problem, we will walk through a simplified forward pass of graph neural networks to help build concrete intuition for how GNNs operate (and so we will not be thinking about the downstream tasks). We will gradually add layers of complexity to the forward pass to allow GNNs to become more expressive.

To begin with, let us assign values v_A, v_B, v_C, v_D to nodes A, B, C, D respectively. Let:

$$v_A = 1, v_B = -2, v_C = -1, v_D = 2$$

We will define our update function for our nodes as follows:

$$f_v(v_i) = \begin{bmatrix} 1 & 2 \end{bmatrix} \text{ReLU}\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} v_i \right)$$

In practice, we could have different update functions at different layers of our network (and more generally, these update functions are learnable). For the sake of this problem, we will reuse the same update function at every layer of the network.

For example, to produce the node value at timestep $t+1$, we must apply the update rule to the node from timestep t, and so we have:

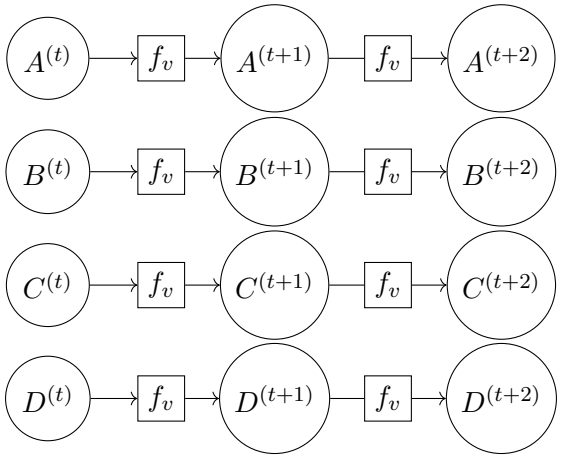
$$v_i^{(t+1)} = f_v(v_i^{(t)})$$

Thus, to compute $v_A^{(1)}$, we have:

$$v_A^{(1)} = f_v(v_A^{(0)}) = \begin{bmatrix} 1 & 2 \end{bmatrix} \text{ReLU}\left(\begin{bmatrix} -1 \\ 1 \end{bmatrix} v_A^{(0)} \right) = \begin{bmatrix} 1 & 2 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 2$$

In general, to produce the graph at timestep $t+1$, we will apply the update rules to each node in our graph from timestep t. Suppose that at timestep 0, the graph G is as above. Let us denote the state of G at timestep t by $G^{(t)}$.

(a) We can visualize two iterations of our current update rule with a diagram such as the following:



For what value of k would the update function defined for each node above be a counterpart to a $k \times k$ convolutional net?

(b) From this diagram, it is clear to see that our GNN is not leveraging the topology of our graph in its forward pass. The way we overcome this is through message passing. In practice, we can apply message passing to both nodes and edges. In this problem, we will only consider applying message passing to nodes to simplify things. Let us consider the new update rule for nodes:

$$v_i^{(t+1)} = f_v(v_i^{(t)}) + \sum_{v_j \in N(v_i)} f_v(v_j^{(t)})$$

where $N(v_i)$ is the set of neighbors of node v_i.
Compute $\mathbf{G^{(1)}}$ under this new update rule.

(c) **Draw a diagram like the one in part (a) reflecting two iterations of our new update rule.**

(d) Suppose the shortest path between nodes u and v in a graph traverses K edges. **How many iterations of updates must we do for information from node $\mathbf{u}$ to reach node $\mathbf{v}$?** *(Hint: Consider the network diagram you made in part (c))*

(e) Given a graph where the diameter (the longest shortest path between any two nodes) is L edges, **how many layers at least should our GNN have to ensure that the receptive field of each node is the entire graph?**

(f) To speed up information flow between nodes, once can add a dummy node to our graph that is maximally connected to all of the original nodes in the graph. **How many layers are needed now to ensure that the receptive field of each node is the entire graph?**

(g) **Draw a diagram like the one from part (c) reflecting the addition of this new dummy node.**

2. Understand GNNs Through Graph Theory

You've seen in lecture how Graph Neural Nets (GNN) naturally generalize the mechanisms of ConvNets. While it is indeed helpful to understand from a ConvNet perspective, the point of GNNs is actually to solve problems with graphs. This question is designed to teach you how the concepts in GNNs could emerge from just solving traditional graph problems.

Consider finding a *shortest path*. This kind of problem, to predict a function of a graph, is common in multiple applications such as in social network studies and molecular biology. Each instance of this problem:

- Has a given graph $G(V, E, W)$ where V is the set of vertices and E is the set of edges. The weight matrix $\mathbf{W}$ defines real-valued labels on the edges and is defined as

$$\mathbf{W}_{ij} = \begin{cases} \text{non-negative weight of edge from vertex } i \text{ to } j, & \text{if edge exists;} \\ \infty, & \text{otherwise.} \end{cases}$$

- Has an input $\mathbf{X}$ that represents the special *source vertex*.
- Has as desired output a vector $\mathbf{Y} \in [0, +\infty]^{|V|}$, where the i-th entry is is the shortest distance from the i-th vertex to specified source vertex.

Suppose we decided to frame this problem as a graph neural net. The network needs to learn a *mapping function* $f_\theta(\mathbf{X}) = \mathbf{Y}$, where the underlying function f is the shortest path.

The first question is whether or not this function is even representable by a GNN. Consider the pseudocode below for Dijkstra's algorithm, a well-known way of computing shortest paths in an iterative manner (notice the while loop). You can also look at the animation on Wikipedia.

```
def dijkstras(source):
    PQ.add(source, 0)
    For all other vertices, v, PQ.add(v, infinity)
    while PQ is not empty:
        p = PQ.removeSmallest()
        relax(all edges from p)
```

```
def relax(edge p,q):
   if q is visited (i.e., q is not in PQ):
       return

   if distTo[p] + weight(edge) < distTo[q]:
       distTo[q] = distTo[p] + w
       edgeTo[q] = p
       PQ.changePriority(q, distTo[q])
```

Figure 1: Pesudocode from https://joshhug.gitbooks.io/hug61b

(a) **Is this a node-level, edge-level, or graph-level prediction?**

(b) In GNN, a key property is message passing along edges of the graph. **Which part of the pseudocode above resembles the idea of message passing?**

How many iterations (forward pass) does it at most take to propagate the required information through the graph?

(c) In the *relax* function we need a $\min$ function, which is not differentiable. **Can you think of a way to approximate this with non-linearities in a neural net?**

3. Backprop through a Simple RNN

Consider the following 1D RNN with no nonlinearities, a 1D hidden state, and 1D inputs u_t at each timestep. (Note: There is only a single parameter w, no bias). This RNN expresses unrolling the following recurrence relation, with hidden state h_t at unrolling step t given by:

$$h_t = w \cdot (u_t + h_{t-1}) \tag{1}$$

The computational graph of unrolling the RNN for three timesteps is shown below:

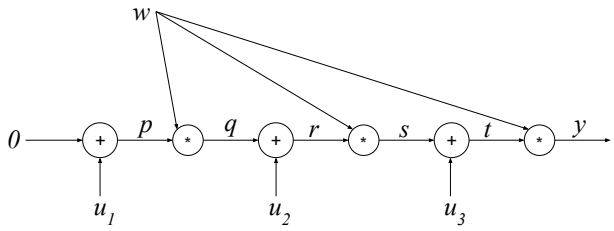


Figure 2: Illustrating the weight-sharing and intermediate results in the RNN.

where w is the learnable weight, u_1, u_2, and u_3 are sequential inputs, and p, q, r, s, and t are intermediate values.

(a) **Fill in the blanks for the intermediate values during the forward pass, in terms of w and the u_i's:**

$$t = \_\_\_\_\_$$

$$y = \_\_\_\_\_$$

(b) **Using the expression for y from the previous subpart, compute $\frac{dy}{dw}$.**

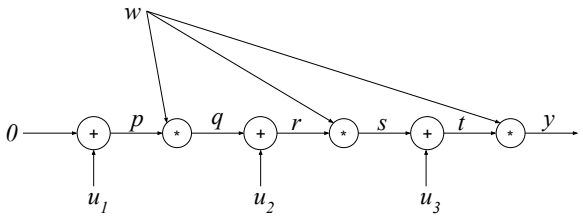
(c) **Fill in the blank for the missing partial derivative of y with respect to the nodes on the backward pass.** You may use values for p, q, r, s, t, y computed in the forward pass and downstream derivatives already computed.

$$\frac{\partial y}{\partial p} = \_\_\_\_\_$$

(d) **Calculate the partial derivatives along each of the three outgoing edges from the learnable w in Figure 2, replicated below.** (e.g., the right-most edge has a relevant partial derivative of t in terms of how much the output y is effected by a small change in w as it influences y through this edge. You need to compute the partial derivatives for the other two edges yourself.)

You can write your answers in terms of the p, q, r, s, t and the partial derivatives of y with respect to them.

Use these three terms to find the total derivative $\frac{dy}{dw}$.



(HINT: You can use your answer to part (b) to check your work.)

(e) **What is the number of computations performed during one forward step of the RNN? Assume that you already have access to the current hidden state.**

(f) **What is the number of computations performed during T forward steps of the RNN? Assume that you already have access to the current hidden state.**

(g) **Now, given a T timestep sequence of inputs, what is the number of computations performed during the forward pass of the RNN? How many of these computations can be done in parallel? Do know worry about exact implementation details, just give a high-level overview.**

Contributors:

- Olivia Watkins.
- Anrui Gu.
- Kevin Li.
- Suhong Moon.
- Anant Sahai.
- Saagar Sanghavi.
- Naman Jain.

