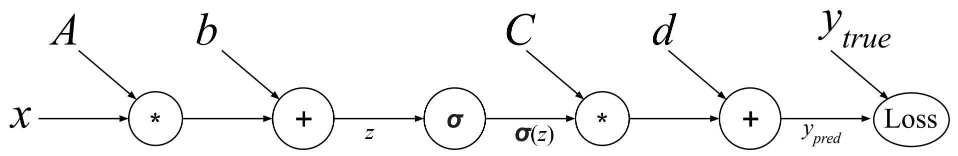
EECS 182 Deep Neural Networks — Fall 2022 Anant Sahai — Midterm Review: Basics

Consider the simple neural network that takes a scalar real input, has 1 hidden layer with k units in it and a sigmoid nonlinearity for those units, and an output linear (affine) layer to predict a scalar output. We can algebraically write any function that it represents as

$$y_{pred} = C\sigma(Ax + \mathbf{b}) + d$$

The $\sigma(.)$ represents an arbitrary nonlinearity, with derivative $\sigma'(.)$ Where $x \in \mathbb{R}$, $A \in \mathbb{R}^{k\times 1}$, $\mathbf{b} \in \mathbb{R}^{k\times 1}$, $C \in \mathbb{R}^{1\times k}$, $d \in \mathbb{R}$, and $y_{pred} \in \mathbb{R}$ We can write it as $y_{pred} = C\sigma(\mathbf{z}) + d$, where $z = Ax + \mathbf{b}$ and the nonlinearity is applied element-wise. We have the true label y_{true} for each x, and we use the L2 Loss $L(y_{true}, y_{pred}) = (y_{true} - y_{pred})^2$.



1. (a) Consider the sigmoid nonlinearity function $\sigma(z) = \frac{1}{1+e^{-z}}$. Show that $\frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z))$

Solution:

$$\sigma(z) = (1+\exp(-z))^{-1}$$
$$\sigma'(z) = -(1+\exp(-z))^{-2}(-\exp(-z))$$
$$\sigma'(z) = \frac{1}{1+\exp(-z)} \cdot \frac{\exp(-z)}{1+\exp(-z)}$$
$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

(b) Calculate $\frac{\partial L}{\partial d}$

Solution: $\frac{\partial L}{\partial d} = \frac{\partial L}{\partial y_{pred}} = 2 * (y_{pred} - y_{true})$

(c) Calculate $\frac{\partial L}{\partial C_i}$

Solution: $\frac{\partial L}{\partial C_i} = 2 * [\sigma(Ax + \mathbf{b})]_i * (y_{pred} - y_{true})$

(d) Calculate $\frac{\partial L}{\partial b_i}$

Solution: $\frac{\partial L}{\partial b_i} = 2 * C\sigma'(Ax + \mathbf{b}) * 1$

(e) Calculate $\frac{\partial L}{\partial A_i}$

Solution: $\frac{\partial L}{\partial A_i} = 2 * C\sigma'(Ax + \mathbf{b}) * x_i$

(f) Write the gradient-descent update rule for $\mathbf{b}_{t+1}$ with learning rate α.

Solution: $\mathbf{b}_{t+1} = \mathbf{b}_t - \alpha \frac{\partial L}{\partial b_t}^T$

2. Given the Regularized Objective function:

$$\underset{\mathbf{x}}{\operatorname{argmin}} \|A\mathbf{x} - \mathbf{b}\|^2 + \lambda\|\mathbf{x}\|^2$$

Use vector calculus to find the closed form solution for $\mathbf{x}$. Interpret what this means in terms of the singular values.

Solution: Expand out objective

$$f(\mathbf{x}) = \mathbf{x}^T A^T A\mathbf{x} - 2\mathbf{x}^T A\mathbf{b} + \mathbf{b}^T\mathbf{b} + \lambda\mathbf{x}^T\mathbf{x}$$

Take gradient wrt $\mathbf{x}$ and set it to zero:

$$\nabla_{\mathbf{x}} f = 2A^T A\mathbf{x} - 2A^T\mathbf{b} + 2\lambda\mathbf{x} = \mathbf{0}$$

Solve for $\mathbf{x}$:

$$(A^T A + \lambda I)\mathbf{x} = A^T\mathbf{b}$$

$$\mathbf{x} = (A^T A + \lambda I)^{-1} A^T\mathbf{b}$$

Interpretation: "Hack" of shifting the singular values of $A^T A$ away from zero, so that they don't blow up when the matrix is inverted.

3. Consider a simple neural network that spits out 1-dim values after a nonlinearity. These values for a batch are $\{1, 7, 7, 9\}$. What is the output of running batchnorm with this data and $\gamma = 1$ and $\beta = 0$. In other words, standardize the data to have mean 0 and variance 1.

Solution: We calculate the mean of the batch as $\mu = \frac{1+7+7+9}{4} = \frac{24}{4} = 6$

We calculate the variance as $\sigma^2 = \frac{(1-6)^2+(7-6)^2+(7-6)^2+(9-6)^2}{4} = \frac{5^2+1^2+1^2+3^2}{4} = \frac{36}{4} = 9.$

So the standard deviation is $\sigma = \sqrt{9} = 3$.

To standardize the batch, we subtract out the mean and divide by the standard deviation. so our batch becomes $\{\frac{1-6}{3}, \frac{7-6}{3}, \frac{7-6}{3}, \frac{9-6}{3}\} = \{\frac{-5}{3}, \frac{1}{3}, \frac{1}{3}, 1\}$

4. Consider a simplified batchnorm layer where we don't actually divide by standard deviation, instead we just de-mean our data before scaling it by γ and shifting it by β, then passing it to the next layer. That is, we calculate our mini-batch mean μ, then simply let $\hat{x}_i = x_i - \mu$, and $y_i = \gamma\hat{x}_i + \beta$ is passed onto the next layer. Assume batchsize of m. If our final loss function is L, Calculate $\frac{\partial L}{\partial x_i}$ in terms of $\frac{\partial L}{\partial y_i}$, γ, β, x_i, and m.

Solution:

$$\frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial y_i} * \frac{\partial y_i}{\partial \hat{x}_i} * \frac{\partial \hat{x}_i}{\partial x_i} = \frac{\partial L}{\partial y_i} * \gamma * (1 - \frac{1}{m})$$

Remember, the mean depends on x_i, so don't forget the $-\frac{1}{m}$.

EECS 182 Deep Neural Networks
Fall 2022 Anant Sahai

Discussion 8

1. Autoencoders (Conceptual)

State whether each of the statements below is True or False and explain why. If the type of autoencoder is not specified, you may consider all autoencoder types (vanilla, denoising, masked, etc.)

(a) There is no point in checking autoencoder reconstruction performance on a validation set because we will ultimately evaluate whether representations are useful by training on downstream tasks.

Solution: False: Checking performance on a validation set is still useful for sanity checking that the model is training correctly (val loss should go down), diagnosing overfitting, and early stopping.

(b) If you train two different autoencoder variants on the same dataset, the one which produces lower validation loss will perform better on the downstream task. **Solution:** This is not always true. For instance, if you add noise or masking or make the bottleneck smaller, reconstruction loss may go up, but these could still produce more useful representations.

(c) The autoencoder decoder is not used after pretraining. **Solution:** True, we only use the encoder for downstream training.

(d) Using autoencoder representations can can sometimes produce worse performance on a downstream task than using raw inputs. **Solution:** True. This may occur, for instance, if the pretraining dataset was too different from the task dataset, or if the autoencoder bottleneck is small enough that it throws away features which are important for that downstream task.

(e) Autoencoder representations can be useful even if the representation was trained on a very different dataset than the downstream task. **Solution:** True, often autoencoders are trained on large, diverse datasets. If these pretraining datasets are diverse enough that they are a superset of the task data, or at least are diverse enough that they learn some general patterns which are relevant to the task data, the AE representation may be useful.

(f) Using an autoencoder representation (rather than using raw inputs) is most useful when you have few labels for your downstream task. **Solution:** True: Unsupervised pretraining provides the largest gains when you have little task-specific data (and are therefore at risk of overfitting), though it can still help somewhat even when plenty of data is available.

(g) With images, it is often more effective to mask patches than to mask individual pixels. **Solution:** True. If you masked pixels, the model could get decent loss by just copying neighboring pixels. This representation would not learn much about the overall image structure

(h) We can think of masked and denoising autoencoders as vanilla autoencoders with data augmentation applied. **Solution:** True: like other data augmentations, these can help prevent overfitting and produce more robust representations. Masking, however, sometimes involves additional changes to the training procedure which are not used with other augmentations (e.g. some transformer architectures completely remove masked tokens from the encoder.)

(i) If you trained an autoencoder with noise or masking, you should also apply noise/masking to inputs when using the representations for downstream tasks. **Solution:** False: this is uncommon (though it is done occasionally when you want data augmentation on the downstream task). Like many regularizers (like data augmentation and dropout), the modification is applied during training only.

(j) Autoencoders always encode inputs into fixed-size lower-dimensional representations. **Solution:** False. Masked and denoising autoencoder representations may not be lower-dimensional, and representations can be variable-sized (e.g. a transformer's representation contains one vector for each input token.

2. Beam Search

This problem will also be covered on the homework.

When making predictions with an autoregressive sequence model, it can be intractable to decode the true most likely sequence of the model, as doing so would require exhaustively searching the tree of all $O(M^T)$ possible sequences, where M is the size of our vocabulary, and T is the max length of a sequence. We could decode our sequence by greedily decoding the most likely token each timestep, and this can work to some extent, but there are no guarantees that this sequence is the actual most likely sequence of our model.

Instead, we can use beam search to limit our search to only candidate sequences that are the most likely so far. In beam search, we keep track of the k most likely predictions of our model so far. At each timestep, we expand our predictions to all of the possible expansions of these sequences after one token, and then we keep only the top k of the most likely sequences out of these. In the end, we return the most likely sequence out of our final candidate sequences. This is also not guaranteed to be the true most likely sequence, but it is usually better than the result of just greedy decoding.

The beam search procedure can be written as the following pseudocode ($y_{t,i,k}$ is the kth token at timestep t generated from hypothesis i. $y_{1:t-1}, i$ means hypothesis i is a string of tokens for timesteps 1 through $t-1$.)

Algorithm 1 Beam Search

```
for each time step $t$ do
    for each hypothesis $y_{1:t-1,i}$ that we are tracking do
        find the top $k$ tokens $y_{t,i,1}$,...,$y_{t,i,k}$
    end for
    sort the resulting $k^2$ length $t$ sequences by their total log-probability
    store the top $k$
    advance each hypothesis to time $t+1$
end for
```

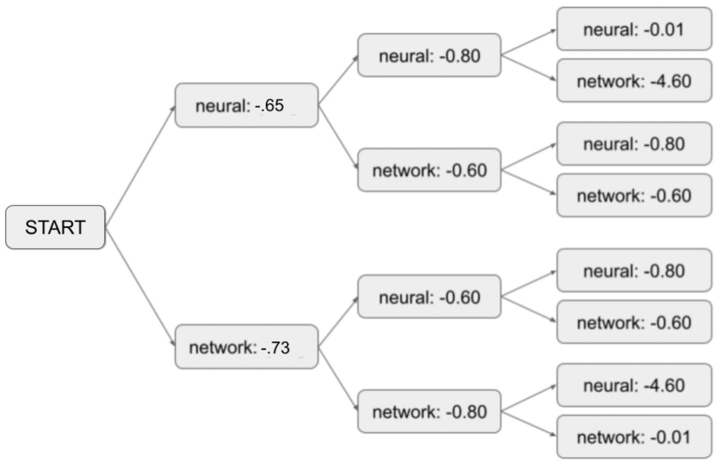


Figure 1: The numbers shown are the decoder's log probability prediction of the current token given previous tokens.

We are running beam search to decode a sequence of length 3 with $k = 2$. Consider predictions of a decoder in Figure 1, where each node in the tree represents the next token **log probability** prediction of one step of the decoder conditioned on previous tokens. The vocab consists of two words: "neural" and "network".

(a) **At timestep 1, which sequences is beam search storing? Solution:** There are only two options, so our beam search keeps them both: "neural" (log prob = -.65) and "network" (log prob = -.73).

(b) **At timestep 2, which sequences is beam search storing? Solution:** We consider all possible two word sequences, but we then keep only the top two, "neural network" (with log prob = -.65 - .6 = -1.25) and "network neural" (with log prob = -.73 - .6 = -1.33).

(c) **At timestep 3, which sequences is beam search storing? Solution:** We consider three word sequences that start with "neural network" and "network neural", and the top two are "neural network network" (with log prob = -.65 - .6 - .6 = -1.85) and "network neural network" (with log prob = -.73 - .6 - .6= -1.93).

(d) **Does beam search return the overall most-likely sequence? Explain why or why not. Solution:** No, the overall most-likely sequence is "neural neural neural" with log prob = -.65 - .8 - .01= -1.46). These sequences don't get returned since they get eliminated from consideration in step 2, since "neural neural" is not in the $k = 2$ most likely length-2 sequences.

(e) **What is the runtime complexity of generating a length-T sequence with beam size k with an RNN?** Answer in terms of T and k and M. You may assume that $M \geq K$, and that every time you need to compute the top K you do it by sorting the array and taking the last K. **Solution:**

- Step RNN forward one step for one hypothesis = $O(M)$ (since we compute one logit for each vocab item, and none of the other RNN operations rely on M, T, or K).
- Do the above, and select the top k tokens for one hypothesis. We do this by sorting the logits: $O(M \log M)$. (Note: there are more efficient ways to select the top K, for instance using a min heap. We just use this way since the code implementation is simple.) Combined with the previous step, this is $O(M \log M + M) = O(M \log M)$.
- Do the above for all k current hypotheses $O(KM \log M)$.
- Do all above + choose the top K of the K^2 hypotheses currently stored: we do this by sorting the array of K^2 items: $O(K^2 \log(K^2)) = O(K^2 \log(K))$ (since $\log(K^2) = 2\log(K)$). (Note: there are also more efficient ways to do this). Combining this with the previous steps, we get $O(KM \log M + K^2 \log(K))$. When one term is strictly larger than another we can take the max of the two. Since $M \geq K$, we could also write this as $O(KM \log M)$.
- Repeat this for T timesteps: $O(TKM \log M)$.

(f) **What information needs to be stored for each of the k hypotheses in beam search with an RNN? Solution:** We need to store the sequence of tokens $y_{1:t}$ and the final hidden state (and if applicable, cell state) so that we can continue unrolling the RNN.
(If you do not store the hidden states, then rolling out a hypothesis one more timestep involves re-running the RNN from the start, which increases the runtime. For the new runtime, the first timestep unrolls 1 timestep, the second unrolls 2, the third 3, etc. so the average unroll length is $T/2$. Therefore, the average RNN forward pass at each iteration of the algorithm switches from $O(1)$ to $O(T)$, giving $O(T^2KM \log M)$ overall.

Contributors:

- Olivia Watkins.
- CS 182 Staff from past semesters.

EECS 182 Deep Neural Networks
Fall 2022 Anant Sahai

Midterm Review: CNNs

1. CNN Design decisions

Consider a CNN for classification consisting of the layers shown below. "..." indicates appropriate arguments that have been passed in.

```
nn.Conv2d(...),
nn.ReLU(),
nn. MaxPool2d(...),
nn. Conv2d(...),
nn. ReLU(),
nn. MaxPool2d(...),
nn. Flatten(),
nn. Linear(...),
nn. Softmax(...)
```

In the questions below, explain how you would modify this design to achieve particular objectives.

(a) You would like to use fewer parameters in the fully connected layer at the end of the network. **Solution:** The parameters in the final fully connected layer are equal to (num classes) * (flattened output from the last conv block). We can reduce this flattened output by either reducing the number of channels in the last conv layer or by reducing the spatial dimension of the feature map. You can reduce the spatial dimension by increasing the stride in the conv/pooling layers, or by adding additional conv/pooling layers with stride > 1. Removing padding from the conv/pooling layers also slightly decreases the spatial dimension.

(b) You would like the model to be able to handle images of different sizes. **Solution:** Replace the Flatten and fully connected layers with a global pooling layer.

(c) You would like to increase the receptive field of each pixel in the feature map output by the second conv layer. **Solution:** Increase the stride of the first conv or max pool layer. Increasing the kernel size of either conv layer or the first max pool layer has a small effect too.

(d) You would like every conv layer to leave the height and width dimensions of its input unchanged. **Solution:** Use a stride of 1 and add padding to each conv layer. In Pytorch, this is 'SAME' padding. Manually, you would pad by $(K-1)/2$ on each side, where K is the kernel height/width.

(e) You would like to add 50 more layers to this network without suffering from vanishing gradients. **Solution:** Add Resblocks (blocks with residual connections) to the network rather than simple conv layers.

2. 3D Convolutions

We've seen CNNs in 1D and 2D settings. Now, let's consider a 3D setting: you are building a classifier to detect the activity shown in short video clips.

(a) One approach is to build a CNN which uses 3D convolutions. What is the size of the parameters in each convolutional layer? Write your answer in terms of F (number of filters), S (stride), C (number of input channels), H (input height), W (input width), T (input time length), K (kernel size). **Solution:** The weight will be (F, C, K, K, K) (Note: in some implementations, the C dimension comes last). The bias is F.

(b) Another approach is to process frames individually, then use an RNN to aggregate information over time. How would you combine a convolutional encoder with an RNN? **Solution:** At each timestep, encode the current frame with the same convolutional encoder. This encoder should include a flatten or global pooling layer which collapses the input into a single vector. This vector is then passed as the input at that timestep to the RNN. Make the final classification using a linear layer on top of the finall hidden state of the RNN.

EECS 182 Deep Neural Networks
Fall 2022 Anant Sahai Discussion 5

1. **Where Could We Use Graph Neural Network** In this problem, we aim to find out where using a graph is an appropriate way to solve the problems highlighted below. Answer with True/ False on where you think a GNN should be applied.

   A. We want to do image segmentation (i.e., predict what class of object is shown in each pixel.)

   Solution: CNN, since ConvNets are typically a good match for image processing. (You could technically use a GNN too since, as shown in Q3, you could think of images as graphs, but that's less common.)

   B. We want to predict what is the likely play of a football team. Your data consists of the coordinate position of each player at each timestep.

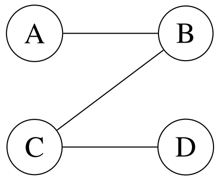
   Solution: probably a combination of RNN and GNN (i.e. a recurrent GNN); a GNN will be useful to model how team players interact, and the recurrent portion will capture how players move over time.

   C. We want to find influential papers by identifying citation patterns between different papers. Your data consists of a list of which papers cite which other papers.

   Solution: GNN, since we want to model connections between papers, and the connections have not fixed spatial structure.

2. 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 vectors v_A, v_B, v_C, v_D to nodes A, B, C, D respectively. Let:

$$v_A = \begin{bmatrix} 1 \\ 2 \end{bmatrix}, v_B = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, v_C = \begin{bmatrix} 2 \\ 2 \end{bmatrix}, v_D = \begin{bmatrix} -3 \\ 2 \end{bmatrix}$$

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

$$f_v(v_i) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} v_i$$

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 $v_A^{(1)}$ and $v_A^{(2)}$, we have:

$$v_A^{(1)} = f_v(v_A^{(0)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 13 \\ 6 \end{bmatrix}$$

$$v_A^{(2)} = f_v(v_A^{(1)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 13 \\ 6 \end{bmatrix} = \begin{bmatrix} 69 \\ 58 \end{bmatrix}$$

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) Using the updates rule above, **compute $\mathbf{G}^{(1)}$ and $\mathbf{G}^{(2)}$**.

Solution:

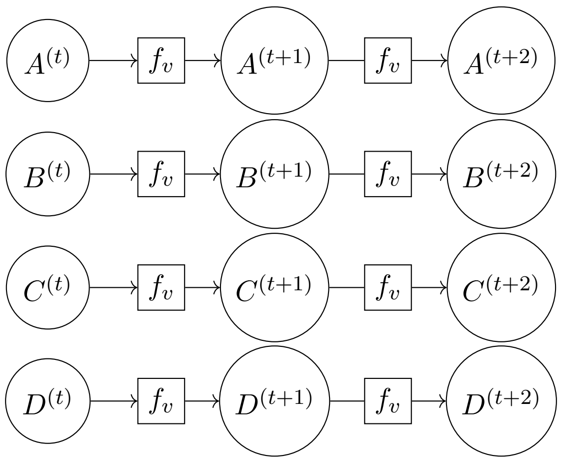
$$f_v(v_A^{(0)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 13 \\ 6 \end{bmatrix}, \qquad f_v(v_A^{(1)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 13 \\ 6 \end{bmatrix} = \begin{bmatrix} 69 \\ 58 \end{bmatrix}$$

$$f_v(v_B^{(0)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 7 \\ -2 \end{bmatrix}, \qquad f_v(v_B^{(1)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ -2 \end{bmatrix} = \begin{bmatrix} 11 \\ 26 \end{bmatrix}$$

$$f_v(v_C^{(0)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 16 \\ 10 \end{bmatrix}, \qquad f_v(v_C^{(1)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 16 \\ 10 \end{bmatrix} = \begin{bmatrix} 98 \\ 74 \end{bmatrix}$$

$$f_v(v_D^{(0)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -3 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ \text{-10} \end{bmatrix}, \qquad f_v(v_D^{(1)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -10 \end{bmatrix} = \begin{bmatrix} \text{-47} \\ -6 \end{bmatrix}$$

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



$$\begin{tikzcd}
A^{(t)} \arrow[r] & \boxed{f_v} \arrow[r] & A^{(t+1)} \arrow[r] & \boxed{f_v} \arrow[r] & A^{(t+2)} \\
B^{(t)} \arrow[r] & \boxed{f_v} \arrow[r] & B^{(t+1)} \arrow[r] & \boxed{f_v} \arrow[r] & B^{(t+2)} \\
C^{(t)} \arrow[r] & \boxed{f_v} \arrow[r] & C^{(t+1)} \arrow[r] & \boxed{f_v} \arrow[r] & C^{(t+2)} \\
D^{(t)} \arrow[r] & \boxed{f_v} \arrow[r] & D^{(t+1)} \arrow[r] & \boxed{f_v} \arrow[r] & D^{(t+2)}
\end{tikzcd}$$

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.

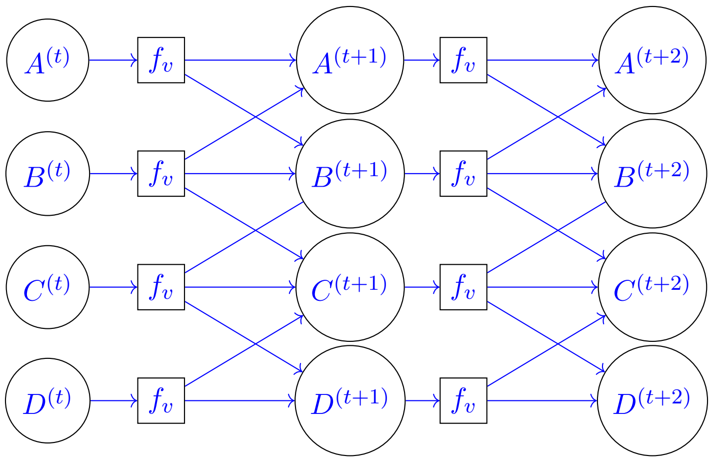
Find $\mathbf{G}^{(1)}$ under this new update rule.

Solution: (Only v_A is shown, but similar formulas can be applied to the other nodes. Depending on the node's number of neighbors, the sum will have 2 or 3 terms (# neighbors + 1)).

$$v_A^{(1)} = f_v(v_A^{(0)}) + f_v(v_B^{(0)}) = \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} + \begin{bmatrix} 3 & 5 \\ 4 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 20 \\ 4 \end{bmatrix},$$

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

Solution:

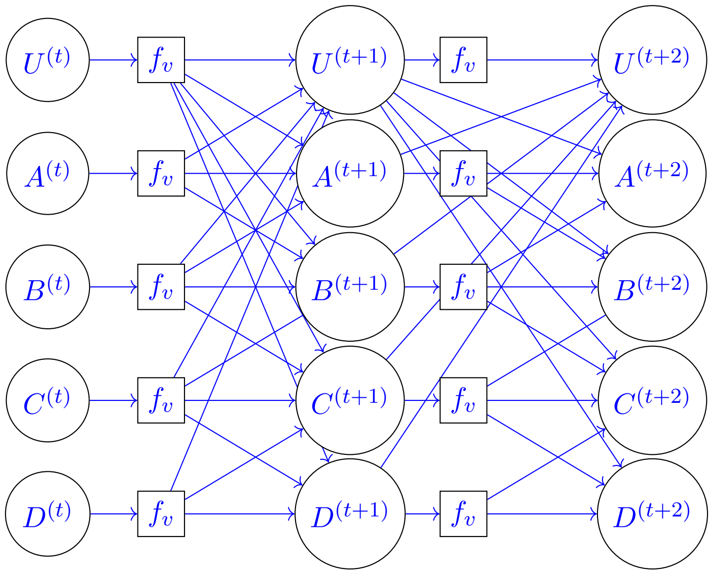


(d) Suppose the shortest path between nodes u and v in a graph traverses K edges. **How many iterations of updates must we do for information from node u to reach node v**? *(Hint: Consider the network diagram you made in part c)*

Solution: From looking at the network diagram, we can see that it will take K iterations for these nodes to communicate with eachother.

(e) If two nodes are connected, they will eventually be able to communicate with each other given enough layers. However, if we want to be able to apply the same network architecture to an arbitrary graph, then we have no guarantee that two connected nodes will be able to communicate with each other with our architecture (such as if we have L layers but we are now processing a graph which has two nodes separated by $L + 1$ edges). One approach to fixing this issue is to add a dummy node to our graph that is maximally connected to all of the original nodes in the graph. We can think of this node as representing the global state of the graph. **Draw a diagram like the one from part c reflecting the addition of this new dummy node. How does this solve our problem?**

Solution:



This solves our problem by allowing an arbitrary pair of nodes to communicate with each other in at most 2 layers.

(f) Although we are thinking about the global state as a "node", in practice we often think of it as a separate entity from the graph, and we allow it to have dimension different from that of our nodes. Correspondingly, we allow our GNN to have a separate update function for the global state, often written at f_U. Since we now have two separate update functions, our diagram from the last part will look different. To make things simpler, let us assume that the nodes have the same dimension as the global state ($U \in \mathbb{R}^2$). **Write an update rule for the global state.**

Solution: The actual update rule is a design choice. Some possibilities are:

$$U^{(t+1)} = f_U(U^{(t)}) + \sum_{v_i \in G} f_v(v_i^{(t)})$$

$$U^{(t+1)} = f_U(U^{(t)}) + f_v\left(\sum_{v_i \in G} v_i^{(t)}\right)$$

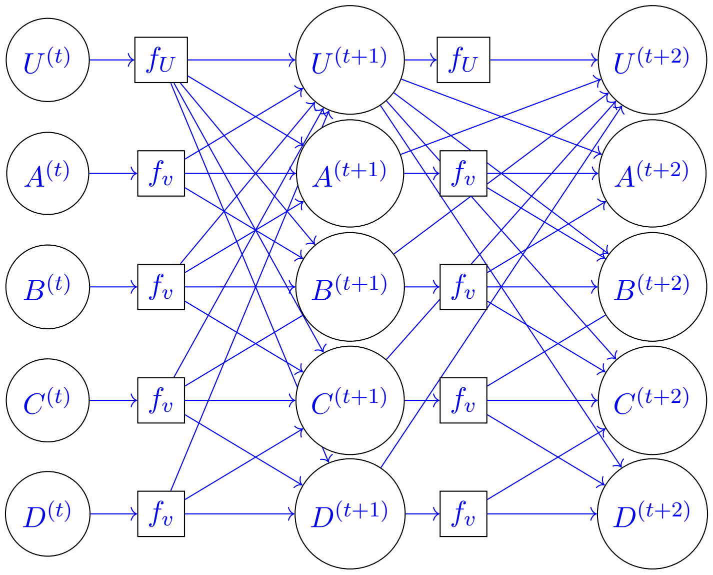
$$U^{(t+1)} = f_U(U^{(t)}) + \sum_{v_i \in G} v_i^{(t)}$$

$$U^{(t+1)} = f_U\left(U^{(t)} + \sum_{v_i \in G} v_i^{(t)}\right)$$

(g) **Draw a diagram like the one in part c reflecting two iterations of our network using f_v and the update rule you wrote in the previous part.**

Solution: Depending on the update rule in the previous part, the resulting diagram will look different. Here is the diagram for the update rule:

$$U^{(t+1)} = f_U(U^{(t)}) + \sum_{v_i \in G} f_v(v_i^{(t)})$$


$U^{(t)}$
$A^{(t)}$
$B^{(t)}$
$C^{(t)}$
$D^{(t)}$
f_U
f_v
$U^{(t+1)}$
$A^{(t+1)}$
$B^{(t+1)}$
$C^{(t+1)}$
$D^{(t+1)}$
$U^{(t+2)}$
$A^{(t+2)}$
$B^{(t+2)}$
$C^{(t+2)}$
$D^{(t+2)}$

3. **Graph Neural Network Conceptual Questions** 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:

   A. You are studying how organic molecules break down when heated. For each molecule, you know the element and weight of each atom, which other atoms its connected to, the length of the bond the atoms, 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.

      (a) How would you represent this as a graph? (What are the nodes, edges, and global state representations? Is it directed or undirected?) **Solution:** Each atom is a node with the element and weight as its value. There is an undirected edge between each pair of bonded atoms, with the bond length as the value. The global representation includes the molecule type.

      (b) How would you use the outputs of the last GNN layer to make the prediction? **Solution:** This is a classification problem across edges. One reasonable approach would be to softmax across logits produced by each edge as well as one logit produced by the global state which represents the option that no edges break.

      (c) How would you encode the node representation? **Solution:** Use a learned embedding for the element ID. Concatenate this with the atom weight. If atom weights are large, you should normalize the weights first.

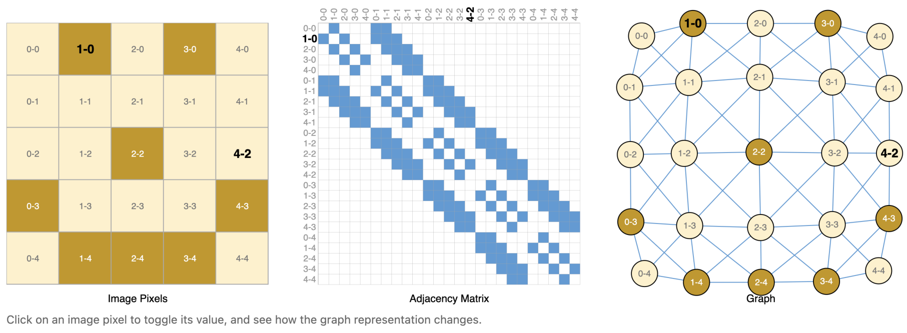


Image Pixels
Adjacency Matrix
Graph
Click on an image pixel to toggle its value, and see how the graph representation changes.

Figure 1: Images as Graphs

   B. There are analogs of many ConvNet operations which can be done with GNNs. As Figure 1 illustrates, we can think of pixels as nodes and pixel adjacencies as similar to edges. Graph-level classification tasks, for instance, are analogous to image classification, since both produce a single, global prediction. Fill out the rest of the table. (Not all rows have a perfect answer. The goal is to think about the role an operation serves in one architecture and whether you could use a technique which serves a similar role in the other architecture.)

CNN	GNN
Image classification	Graph-level prediction problem
Solution: Semantic segmentation (classifying what object is present at each pixel)	Node-level prediction problem
Color jitter data augmentation (adjusting the color or brightness of an image)	**Solution:** Jittering node values
Image flip data augmentation	**Solution:** A flip preserves graph values and connectivity, but changes its spatial orientation. Graphs don't have an orientation, so they don't need an analog. More generally, image flips augment the data by exploiting invariances in the task (i.e. an image's class shouldn't change if you flip it), and there might be similar invariances in graph problems.
Dropout	**Solution:** Zero some nodes (or edges) at one layer of the network
Zero padding edges	**Solution:** Zero padding addresses the fact that conv nets require that every pixel has the same number of neighbors. Graphs don't need an equivalent since they already handle variable numbers of neighbors
ResNet skip connections	**Solution:** Add a skip connection to the update - i.e. $v_i^{t+1} = v_i^t + UPDATE(v_i^t)$
Blurring an image	**Solution:** Averaging node values with neighbors
Solution: Image inpainting (filling in a missing section of an image)	Predicting missing values of nodes
Solution: Only using kernels with the same value for each index except the center (explanation: in GNNs, you typically run the same update function on each neighbor (or often first sum or average the neighbors, then run the update fn on the result). This means all neighbors are treated the same. CNN kernels, in contrast, have different values for at different spatial positions, so a left-side neighbor and a right-side neighbor don't get the same update. The CNN equivalent to edge-order invariance is to use the same value for each index in the kernel. The center pixel can still be different, since this corresponds to the current node value, which updated separately from the neighboring nodes.	Edge-order invariance - i.e. neighboring nodes/edges are a set with no ordering

C. 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? **Solution:** There are multiple strategies (including all of the strategies used in other ML problems with missing values), including creating a special 'missing' token or filling the node with the mean value, and training with dropout on the input. In graphs where individual nodes don't matter much (e.g. point clouds), it may be appropriate to remove the node and associated edges entirely.

D. 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 the updated by pooling all nodes and edges. For more details on the architecture setup, see `https://distill.pub/2021/gnn-intro/#passing-messages-between-parts-of-the-graph`.

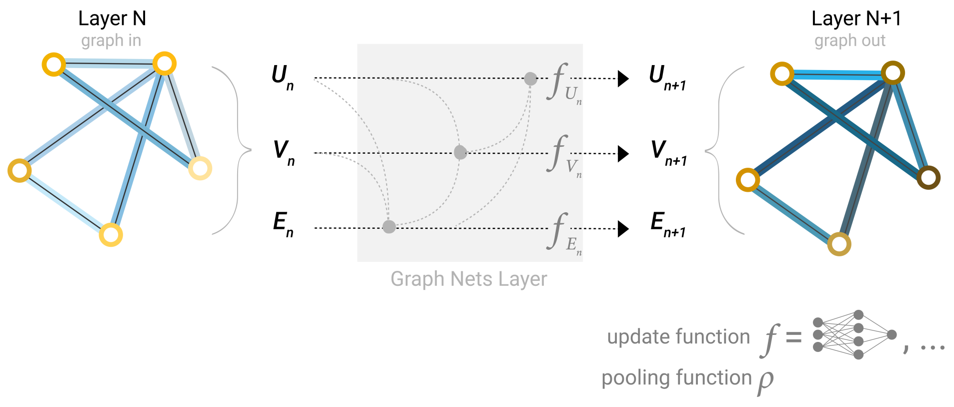


Figure 2: GNN architecture

(a) If we double the number of nodes in a graph which only has node representations, how does this change the number of learned weights in the graph? How does it change the amount of computation used for this graph if the average node degree remains the same? What if the graph if fully connected? (Assume you are not using a global state representation). **Solution:** The number of learned weights remains the same. If the node degree remains the same, computation will also double. If the graph is fully connected, computation scales by a factor of 4 (2x nodes * 2x neighbors to aggregate per timestep). (Here, we're thinking about how computational complexity scales with nodes and edges. There are some fixed costs which don't scale, so it won't be quite 2x or 4x in practice.)

(b) Where in this network are learned weights incorporated? **Solution:** The update functions include learned weights.

(c) The diagram provided shows undirected edges. How would you incorporate directed edges? **Solution:** There are multiple strategies here, but one reasonable option is that within a node's update function, use different weights to incorporate incoming and outgoing edges and nodes. (Incoming edges are edges pointing toward the node, and outgoing edges have the arrows away from the node.) In some cases, it might make sense to use only incoming or outgoing nodes/edges for the update.

(d) The differences between an MLP and a RNN include (a) weights are shared between timesteps and (b) data is fed in one timestep at a time. How would you make an MLP-like GNN? What about an RNN-like GNN? **Solution:** An MLP-like GNN uses separate weights at each layer. The RNN-like GNN uses the same weights at each timestep. If data is sequential, then at each timestep concatenate the node/edge values from your data at time t with the hidden node/edge values for that timestep.

E. Let's say you run a large social network and are trying to predict whether individuals in the network

like a particular ad. Each individual is represented as a node in the graph, and we are doing a node-level prediction problem. You have labels for around half of the people in the network and are trying to predict the other half. Unlike a traditional ML problem, where there there are many independent training points, here we have a single social network. How would you do prediction on this graph?
Solution: You can think about each node's classification as a separate training point. Split the nodes for which you have labels into train and validation sets, then train on the training nodes only. During training, we keep validation and unlabeled nodes in the graph (this is fine, since we aren't using their labels). Use the validation set to do model selection, then predict on the nodes without labels.

F. (Optional) Play around with different GNN design choices in the GNN playground at `https://distill.pub/2021/gnn-intro/#gnn-playground`. Which design choices lead to the best AUC?

Contributors:

- Matthew Lacayo.
- Olivia Watkins.
- Jerome Quenum.
- Anant Sahai.
- Anrui Gu.