CS 182/282A Deep Neural Networks **February 27th, Spring 2023**

Lecture 11: Graph Neural Networks

Lecturer: Anant Sahai *Scribe: Hiva Mohammadzadeh*

11.1 Graph Neural Networks (GNNs)

Graph Neural Networks are a type of neural network designed to work with graph-structured data, where the nodes represent entities, and the edges represent the relationships between them.

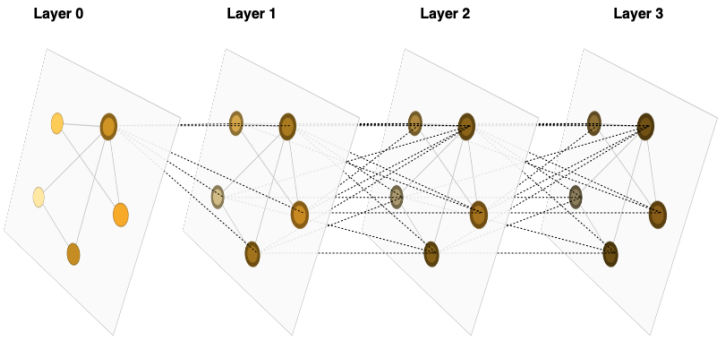


Figure 11.1: Shows an example of a GNN. This figure is taken from the interactive diagram in the Blog post on GNNs, illustrating how a node in a graph accumulates information from nodes around it through the layers of the network.

There are 2 perspectives in understanding Graph Neural Networks:

1. Generalizing Convolutional Neural Networks from images to graphs.
2. Generalizing Graph algorithms to be learnable via Neural Networks.

For the second perspective, there are many algorithms like graphical models that have been handcrafted by humans to extract information from graphs. A natural idea is to use neural networks to learn such algorithms on their own. We will focus on the first perspective to help us understand GNNs.

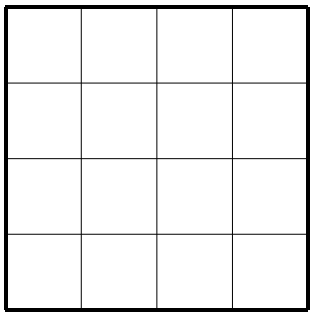
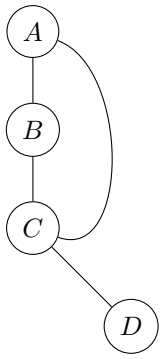
We will go from Convolutional Neural Nets on images to graphs.

11.1.1 Basic Idea

Going from images which are structured data with shape HxWxC (Height x Width x Number Of Channels) to the general idea of GNNs which are just graphs that have nodes, edges between them, and $V_n \in R^c$ which is the associated information or label for node n. Additionally, nodes and edges can have labels to provide further information about them.

The key difference between CNNs and GNNs is that while CNNs operate on a large amount of image data, GNNs typically operate on a single graph or multiple graphs for different examples. In other words, all the

data needed for a GNN is contained within one graph or a set of graphs, whereas in CNNs, the data is spread across many images.

Going from  to 

(a) Image (b) Graph

Motivation: make neural nets work for graph-like structure like molecules.

11.2 Convolutional Neural Networks (CNNs) key ideas and ingredients

Understanding and recalling the key ideas of Convolutional Neural Networks (CNNs) and how they are applied to images will help in understanding Graph Neural Networks (GNNs). Some of these key ideas include:

1. Convolutions (Local operations) on the local neighborhood of every pixel with weight-sharing across different positions in the image.
2. Depth to increase "receptive field" and allow "distant" pixel information to be used. Idea of CNNs having depth.
3. Residual connections to prevent dying gradients.
4. Normalizations to adaptively "speed bump" exploding gradients by bringing down growing activation values.
5. Pooling to downsample and speed up growth of the receptive fields, and "routes" gradients. This helps to reduce the dimensionality of the data and improve computational efficiency.
6. Data Augmentation to improve the generalization of the model and prevent overfitting. Data augmentation can be used to create additional training examples by transforming the existing data in various ways, such as flipping or rotating the images. We have learned some techniques so far including:
   (a) Dropout: Randomly drops out (sets to zero) some of the neurons in the network during training. This forces the network to learn more robust features by preventing it from relying too heavily on any one feature. Dropout can be applied to convolutional layers, fully connected layers, or both.
   (b) Label Smoothing: Replaces the true label of a training example with a smoothed label distribution. This can help prevent overfitting by encouraging the network to learn more generalizable features.

11.3 Understand Graph Neural Networks by adapting key ideas of CNNs

How do we adapt CNN's key ideas to GNNs? What kinds of problems are we trying to solve?

Convolutional Neural Networks (CNNs)	**Graph Neural Networks (GNNs)**
Image-level tasks (Classification or Regression Tasks) (One output / target for entire image. Ex: dog, cat, etc.)	**Graph-level tasks** (Classification Tasks. Ex: Graph of a particular molecule: deciding if it is poisonous or not? Or at what temperature will it melt?)
Pixel-level tasks (Ex: Semantic segmentation for classification of every pixel)	**Node/Edge-level tasks** (Ex: Graph of customers and products in commercial data deciding the pricing of products or how to give) recommendations for each customer)

In Graph-level tasks we want to learn a pattern that allows us to go from the graph to something that is true for the whole graph.

In Node-level tasks we want to say something about the graph from each node. In graphs, we have a duality of nodes and edges (any property or operation that can be defined on nodes can also be defined on edges, and vice versa). Therefore, we have the same kind of problem at edge-level as well.

The training data in CNNs are usually many images but in GNNs, in many cases we only have one big graph that contains all the data. For example, Facebook has a social network graph of all the people, who they are connected and how they've interacted on Facebook. They don't have a whole set of examples, but just one.

Another example on Homework 5 Question 5: Clustering the nodes where we only have data for some of the nodes.

Let's focus on graph-level tasks and see how a GNN works.

11.3.1 Convolutions and aggregation functions in GNNs

We will focus on a Graph level task to better adapt these ideas.

We built a Convolution network for a particular size of image and input size.

Easiest Case: We have a single graph topology like the one in the figure (b) on the last page. Dataset consists of different labelings. To find the counterpart of a Convolutional layer, we want to respect local topology and share learned weights across the graph. When doing CNN, we're fine with adding more channels. The output of a convolution layer is another image which can have more channels. So, we have to do this for GNNs as well. A graph that only has edge labels can be converted to a graph that only has node labels and has no edge labels.

In a 3x3 convolution:

NW	N	NE
W	Me	E
SW	S	SE

(a) 3x3 Convolution

W1	W2	W3
W4	W5	W6
W7	W8	W9

(b) 3x3 convolution weights

Figure 11.2: This figure which was taken from Professor Sahai's lecture notes shows a 3x3 convolution that is turned into learned weight matrix shared across all positions in image.

How do I combine the information from my neighbors to come up with a new value for this channel with this pixel? We can see from the example that the key challenge is that the neighbors in a graph are a set, not a list (like in CNNs) which means that there's no order to them. One other challenge is that they can have a different number of neighbors for different nodes meaning that the sets can be of different sizes.

To tackle these challenges, the first idea is to per channel use a single identical learnable weight $\vec{w}$ for all neighbors and possibly a second weight $\vec{W_{\text{me}}}$ for myself. The channels are ordered so we can use different weights per channel. We can also add some non-linearity to increase the expressive power.

We want functions that can produce an output based on the input of $f(\vec{me_{\text{info}}}, \{\text{neighbor info}\})$, where the $\vec{me_{\text{info}}}$ is a vector representing the data that is available at the current node, and the "neighbor info" is a set of vectors that includes the information from the neighbors of $\vec{me_{\text{info}}}$). To ensure that the graph is valid and respects its structure, we need to avoid dependency on the order of neighbor information. Therefore, we need permutation invariance on the neighbor information.

If we have edge labels, neighbor information are pairs which include information from the neighbor and the label.

These functions are commonly known as aggregation functions since they aggregate information from neighbors.

Some possible extensions include: adding some nonlinearity at the end, having different aggregation functions for different channels. GNNs use aggregation functions to combine the information from neighboring nodes when passing messages through the network. Aggregate functions take in a set of values and return a single value that summarizes the information from the set. They are used to summarize the feature information of neighboring nodes.

Some examples of aggregation functions:

1. Sum aggregation function:

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \sum_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

   where w_{me} is the learned weights.

2. Average/mean aggregation function: need to know the size of the set

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \frac{1}{NumberOfNeighbors} \sum_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

3. Other Permutation Invariant operations (the only principle is that it should only produce the same output regardless of the ordering of the input elements):

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \Pi_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \frac{1}{NumberOfNeighbors} \Pi_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \max_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \frac{1}{NumberOfNeighbors} \max_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \min_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

$$f(< \vec{w_{\text{me}}}, \vec{v_{\text{me}}} >) + \frac{1}{NumberOfNeighbors} \min_{i \in Neighbors} < \vec{w}, \vec{v_{\text{i}}} >$$

   others: norms (appropriately symmetric) and $softmax(i.e. in the form of \log \sum_i e^{z_i})$ which acts as a min or max when it is used as an aggregation function.

4. Can have functions with different thing for myself vs others:

$$f_{\text{w}}(\vec{v_{\text{me}}}, \sum S_{\text{w}}(\vec{v_{\text{me}}} \vec{v_{\text{i}}}) \vec{v_{\text{i}}})$$

   where $S_{\text{w}}(\vec{v_{\text{me}}} \vec{v_{\text{i}}})$ is the similarity function that could be learned.

Using GNN type operations, we can exactly recover a ConvNet because we have not lost any information.

Key note: The function is the same for every given node in graph at this layer which means that the weights are one set of weights that are learned. So, we will learn one set of weights for the aggregation operation and we apply it across the whole graph to get a new graph with more channels. Directed graphs have 2 sets of vectors.

Exercise: Let each pixel of an image be a node, how to recover a ConvNet with a graph by creating a graph representation of that CNN (having each layer be a node and the connections between layers as edges), and then assign the weights and biases of the CNN to the corresponding nodes in the graph. Learned weights can depend on the edge labels.

11.3.2 Depth to increase "receptive field" in GNNs

To achieve this, we can just copy the graph over and over again. Increasing the depth of the GNN can help to increase the receptive field, which refers to the range of neighborhood information that a node can capture during the message passing phase. As the number of layers increases, the receptive field of the network expands to cover a larger region of the graph.

Question: How can we make the analogy of changing the filter size in a CNN to a GNN? How can we make the graph bigger?

We can achieve this by taking 2 graph layers where we first take information from the neighbors and store them, and do it again. Then we can use our two-hop neighbors and aggregate information from them. However, we usually rely on having multiple layers to achieve this.

11.3.3 Residual Connections in GNNs

As long as we structure the layers such that the channel info adds, we don't get dying gradients. In a residual connection, the output of a layer is added to the input of the same layer, allowing the network to "skip" over layers and improve gradient flow. Formally, given an input feature vector x and a message passing function f, the output of the GNN layer with residual connections is:

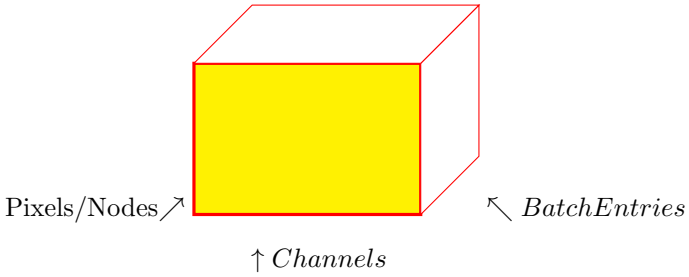
$$y = f(x, A) + x$$

where A is the adjacency matrix of the graph. The residual connection allows the gradient to flow through the layer more easily, as the gradient can be backpropagated directly to the input feature vector without being attenuated by the message passing function.

11.3.4 Normalizations in GNNs

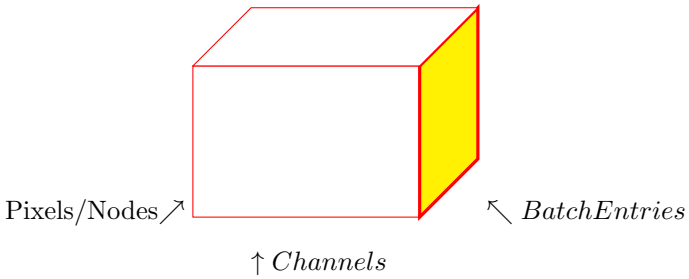
The easiest way to implement this is using **Layer Norm**. Layer normalization in CNNs normalizes the features of nodes across all nodes in a layer, instead of just a batch. It can help improve the stability of training and prevent overfitting. In GNNs, we usually can't do averaging across nodes because there are too many of them.

For CNNs: pixels × channels × batch entries. For GNNs: Nodes × channels × batch entries



Batch Normalization: Keep the channels separate and go across batches and pixels. Batch normalization helps to reduce the internal covariate shift and helps the GNN to learn more effectively. In CNNs, it is used to normalize the feature vectors of nodes in a batch during training. In GNNs we can't do it because we can't load the entire graph. Therefore, we have mini-batch training.

For CNNs: pixels × channels × batch entries For GNNs: Nodes × channels × Batch



The two figures above were taken from Professor Sahai's lecture notes to show the structure of layer normalization and batch normalization in CNNs vs GNNs.

11.3.5 Mini Batches in large GNNs

Mini-batch training is a common technique used in large-scale GNNs to handle the computational and memory constraints of training on large graphs. In mini-batch training, a small subset of nodes or edges are sampled from the graph and used to update the parameters of the network during each training iteration.

However, mini-batch training can also introduce some challenges, such as the loss of information between mini-batches and the possibility of bias in the sampling process. To address these issues, techniques such as adaptive sampling, importance sampling, and graph coarsening can be used to ensure that the mini-batches are representative of the overall graph and that important information is not lost during training.

We need to sample one subgraph to look at a time, but we can not do this the naive way by sampling at random because it will result in us losing all the edges and local connectivity information. Instead we sample targets and their neighbors by first sampling the node itself then sample their neighbors recursively to get a local graph that has the information that we want. When doing this, we try to always sample the same number of nodes if it all fits in memory but we're limited by GPU memory. We want the batch to be the most effective .

Key Note: Applying clustering on a graph is also tricky because we want to make sure that we're learning weights that are good and it can be ambiguous how we sample the clusters for our batches.

11.3.6 Pooling to downsample in GNNs

In CNNs, downsampling is typically performed using pooling layers. Pooling layers reduce the spatial size of the feature maps by applying a pooling function, such as max-pooling or average-pooling, to non-overlapping regions of the input which reduces the number of parameters in the network and helps prevent overfitting.

In GNNs, pooling is a technique used in GNNs to downsample the graph structure and reduce the computational and memory requirements of the network. However, pooling can also introduce some challenges, such as the loss of information and resolution during downsampling, as well as the possibility of introducing bias or instability in the network. So, downsampling is typically performed by aggregating information from neighboring nodes in the graph. This is often done using message-passing schemes, where each node receives information from its neighbors, aggregates this information, and updates its own representation. This process can be repeated multiple times to perform hierarchical feature extraction and downsampling.

Therefore, the choice of pooling operation and its parameters should be carefully considered and validated for each application. Therefore, pooling is very domain specific, hence we usually don't do it and just keep processing for the whole graph.

11.3.7 Some further comments on adapting ideas of CNN

1. So far we have focused on graphs without labels but it is important to note that if we do have edge labels in any context, we should just add them in to the graph.
2. If we have different graphs that are different from our initial assumptions: Eliminate the assumption that we have a single graph topology and that the dataset consists of different labelings: Nothing breaks because the local neighborhoods of our graph already had a diversity of neighborhoods in them. Our approach is fundamentally local and our assumptions are always true. This is because our graph does not change during the processing of the Graph Neural Net itself.

11.4 Exploring Graph Neural Networks Further

Some of these ideas were taken from the following lecture.

11.4.1 Holding out Nodes in node-level tasks: Have labels for each node

We either have labels on the entire graph for every node or we have a graph with labels and we want to keep some of those labels back for validation set.

In Node level problems where we only have one graph, what does it mean to have a Train/Test split? This will always require us to look at only labels of a subset of graphs. Therefore, we have to hold some parts of the graph back.

2 General approaches in doing so:

1. Naive approach (always try): Simply deleting the hold out and all associated edges during training. Delete the part of the graph that we're holding back just remove it from training data so that they're not in the graph. This has bad performance sometimes because we're destroying the connectivity of the graph.
2. Keeping the nodes (including labels) and associated edges but removing "the information" in the node. They will be in the graph but they will have dummy information associated with them which allows us to preserve the connectivity of the graph. We use this when the first approach ruins the topology of the graph and ruins the connectivity. Some comments:
   (a) Not in the loss.
   (b) "Removal" of node label by:
      i. Replacing the node with a zero vector if zero vector is not meaningful
      ii. Replacing it with the special label "MASK" (which means "No information" here. Domain specific: requires preprocessing. Learns to ignore information we don't need) Allows message to pass through that node. Also allows to just train on one graph.

11.5 What I wish this lecture also had to make things clearer?

11.5.1 More about Masking (including an example)

Masking nodes is the process of selectively ignoring or masking a subset of nodes during training or inference. For Example: The graph may contain a small subset of labeled nodes and a much larger subset of unlabeled nodes. In this scenario, we can mask out the unlabeled nodes during training and only use the labeled nodes to train the GNN. This approach can be particularly effective in cases where the number of labeled nodes is small relative to the number of unlabeled nodes. This can help improve the efficiency and accuracy of GNNs in node-level tasks by focusing the model's attention on the relevant subset of nodes while ignoring irrelevant or noisy information.

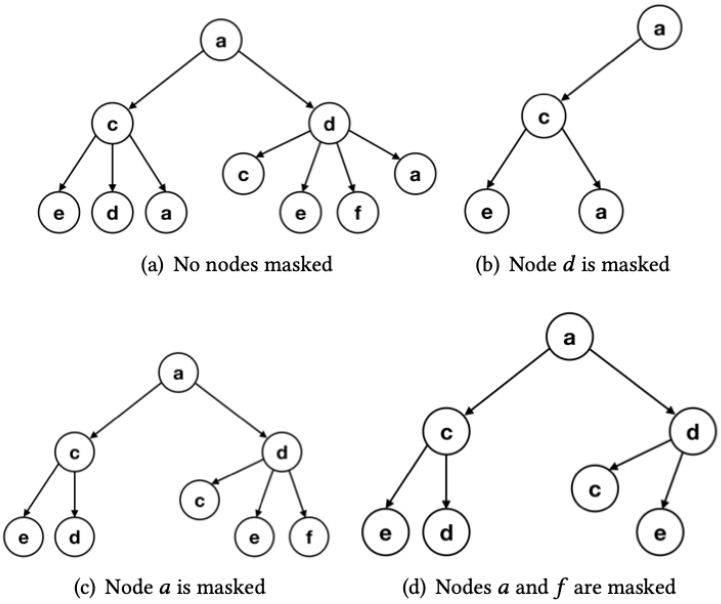


Figure 11.3: This figure shows that if a node is masked, it is excluded from the child set of other nodes but can still appear as the root. Figure from: Paper on Node Masking.

11.5.2 Practical Examples For GNNs

Graph Neural Networks have a wide range of practical applications across various domains, including computer vision, natural language processing, recommendation systems, chemistry and many more. Here are some practical examples of GNN applications:

1. Graph classification: GNNs can be used to classify the entire graph. Such as molecules or proteins in bio informatics, based on their structural and chemical properties.
2. Node classification: GNNs can be used to predict properties or labels for individual nodes in a graph, such as predicting the function of genes in biological networks or predicting the sentiment of social media posts in a social network.
3. Recommendation systems: GNNs can be used to recommend items to users based on their preferences and previous interactions, such as recommending movies or products to customers.
4. Natural language processing: GNNs can be used to model the structure and relationships between words and sentences in a text, such as identifying entities and relationships in a sentence or predicting the sentiment of a document. Take CS 288 at Berkeley to learn more Natural Language Processing.

11.6 Further Reading and resources:

To learn more about Node Masking: Paper on Node Masking: Making Graph Neural Networks Generalize and Scale Better https://arxiv.org/abs/2001.07524

To visualize more Graph Neural Networks visit: https://distill.pub/2021/gnn-intro/

For more standardized and classical examples: a library bulit on Pytorch for deep GNN: https://www.dgl.ai/

11.7 References

Sanchez-Lengeling, B., Reif, E., Pearce, A., amp; Wiltschko, A. B. (2021, September 8). A gentle introduction to graph neural networks. Distill. Retrieved March 9, 2023, from https://distill.pub/2021/gnn-intro/

Mishra, P., Piktus, A., Goossen, G., amp; Silvestri, F. (2021, May 16). Node masking: Making graph neural networks generalize and scale better. arXiv.org. Retrieved March 9, 2023, from https://arxiv.org/abs/2001.07524/