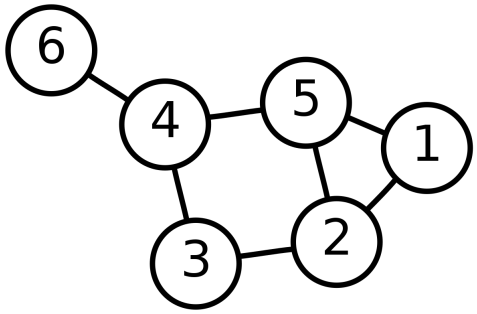
Graph Neural Networks

Saahith Janapati*

November 2020



1 Primer on Graphs

A graph is a collection of nodes and the edges that connect each node. Graphs can be used to represent a variety of structures from social media networks to chemical compounds. For more information on graphs, see the first page of this SCT lecture.

2 Overview of Graph Neural Networks

Graph Neural Networks (GNNs) aim to create embeddings for each node in a graph. Essentially, we want to encode the spatial structure and neighborhood information of some node into a low-dimensional representation. This is similar to the convolution operation, which condenses information about individual pixels and their neighbors into lower dimensions.

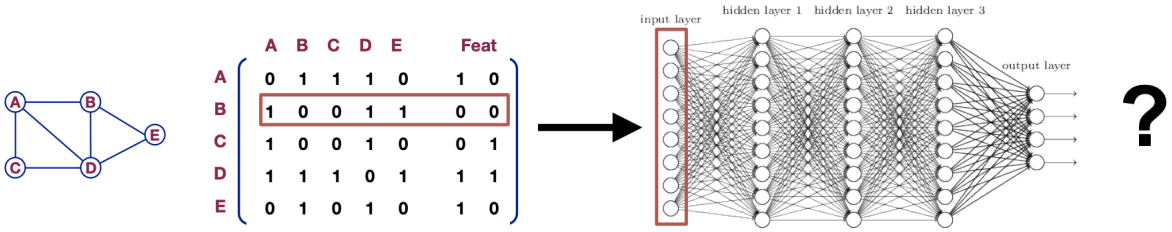
These embeddings can then be used to perform tasks like node classfication, node similarity, and graph classification.

In this lecture, I discuss a simple version of GNNs and then discuss Graph Convolutional Networks which are slightly different. Note that there a wide variety of GNN architectures that are suited for a variety of graph-based tasks. Furthermore, graph-based deep learning is an active area of research, so there will likely be many more proposed architectures in the future. To learn of all the variants of GNNs today, check out this paper.

3 Traditional Deep Learning Approaches on Graphs

It is worth asking why we can't just obtain traditional graph representations (like an adjacency matrix) and just feed them into a neural network that could then learn to perform graph classification. The problem is that representations like adjacency matrices are quite sparse and large graph will lead to large input matrices. Another problem is that the adjacency matrix is not invariant to node ordering; switching up the order will lead to a completely different matrix. Lastly, a neural network trained on graphs of one size can't be used for graphs of other sizes because the input size to the network must be constant.

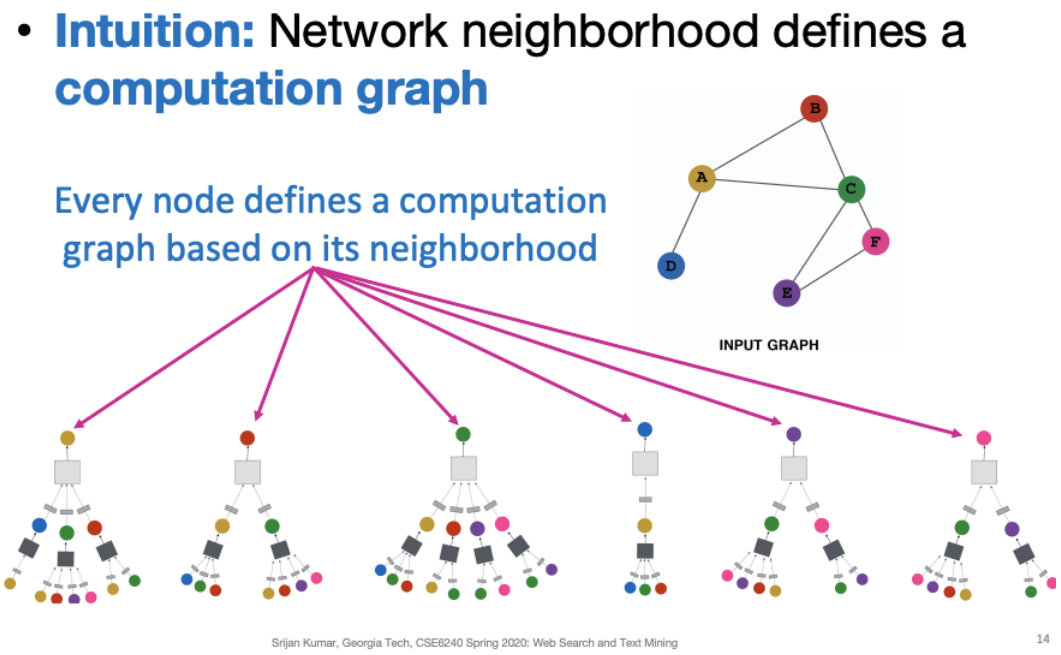
*Edited and presented by Tarushii Goel



4 Graph Neural Networks

As previously mentioned, the goal of graph neural networks is to output low-dimensional embeddings for each node that capture information regarding the neighboring nodes.

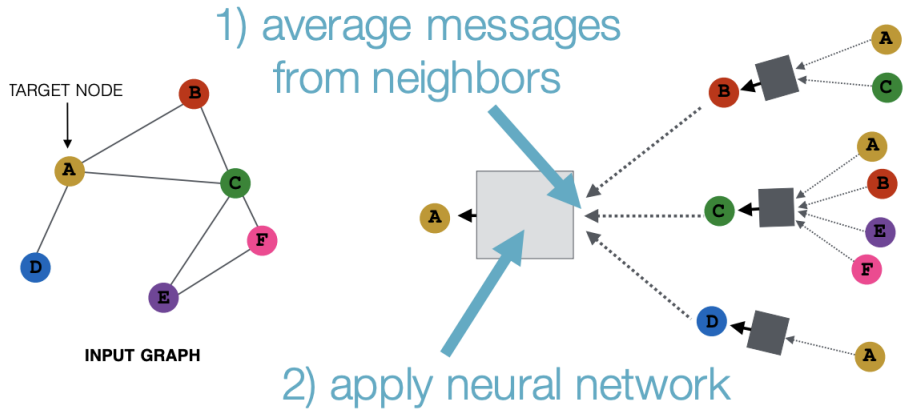
Before discussing the exact mechanics of the generation of this embedding, it's important to notice that the surrounding neighborhood of a node defines a computational graph for every node in the graph, as shown in the picture below.



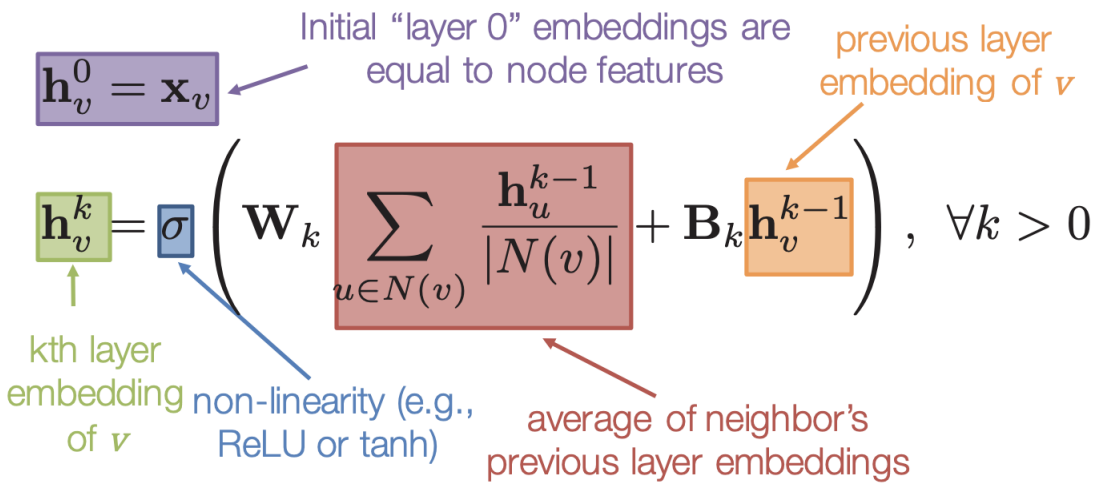
Here the height of each node's neighborhood computation graph is 2, although we can set it to whatever we like. To a certain extent, the higher this hyperparameter is, the more accurate our node embeddings will be.

The basic idea behind GNNs is to use this computation graph to aggregate the embeddings of the neighboring nodes together and apply a neural network to the combined embeddings, as outlined in the below diagram.

- **Basic approach:** Average neighbor information and apply a neural network.



Formally, we can describe this process as the equation shown below.



The embedding of each node is initially set to the feature vector of each node. For a graph representing a social network, this could be a vector representing characteristics of the user, (such as their age, gender, interests, etc.). At each layer of a single node's computation graph, we simply average the embeddings of all the incoming neighbor nodes and multiply it by weight matrix W_k. We also add to this value the product of a secondary wight matrix that is multiplied with the current node's embedding on the previous layer. We then run this sum through a non-linearity function such as ReLU, resulting in the embedding for our node.

Note that the matrix written as B_k is not a bias matrix, but is instead a secondary weight matrix that is multiplied the node's own previous embedding.

4.1 Loss Functions

Note that all the operations depicted in the previous equation are all differentiable. Thus, we can easily conduct back propagation on the weight matrices to optimize them for some loss function. The learning process for GNNs can either be unsupervised or supervised. There a variety of unsupervised approaches to calculating loss (including random walks, graph factorization, or some similarity measure that incentives our network to output similar embeddings for nodes that are close together in the graph).

If we have a task such as node classification, we can define our own loss function. Note that for node classification, in addition to the weight matrices previously described, we must create a final set of weights

to transform the embeddings generated in the last layer into a discrete classification. Once we have all these parameters, we can perform traditional gradient descent to minimize some categorical loss function.

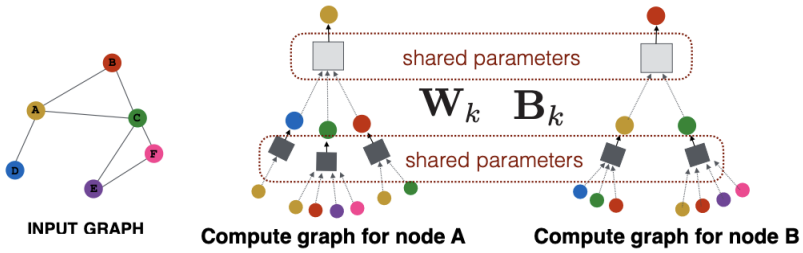
For classification specifically, in addition to the weight matrices, we must also define some classification weights that transform our last embedding to a discrete classification (in the image below, this classification is whether or not a node represents a human or bot in a social network).

4.2 Training

In order to train the parameters, you must select several nodes, generate computation graphs for those nodes, and then conduct gradient descent on each graph. The collection of computation graphs is akin to a batch in traditional deep learning.

4.3 Inductive Capabilities

The key thing to note about GNNs is that all the weight matrices of each layer are shared parameters. We do not train separate weight matrices for the computation graph of each node.



The beauty of this fact is that if a new node is added to our graph, we do not have to retrain our weights at all. We simply generate the computation graph of the newly added node to the appropriate depth and then use the weights we have already learned. We can also use the same weights on a completely different graph (and it does not need to be the same size).

4.4 Graph Convolutional Networks

Graph Convolutional Networks (GCNs), introduced in this paper in 2017, are a slight variant of the simpler GNNs we have described so far. GCNs use a slightly different formula to aggregate the embeddings of neighboring nodes. These differences are outlined in the image below. The basic neighborhood aggregation technique is what we have discussed previously.

Basic Neighborhood Aggregation

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v)} \frac{\mathbf{h}_u^{k-1}}{|N(v)|} + \mathbf{B}_k \mathbf{h}_v^{k-1}\right)$$

VS.

GCN Neighborhood Aggregation

$$\mathbf{h}_v^k = \sigma\left(\mathbf{W}_k \sum_{u \in N(v) \cup v} \frac{\mathbf{h}_u^{k-1}}{\sqrt{|N(u)||N(v)|}}\right)$$

same matrix for self and neighbor embeddings

per-neighbor normalization

The key differences are that 1) there is no secondary weight matrix for the nodes own previous embedding and 2) the normalization over the neighbors is not a simple average. This new normalizing method lessens the impact of neighbors with embeddings that have a high magnitude.

Empirical results have shown that these networks perform better on a variety of tasks.

Note that this version of Graph Neural Networks is just one of many. We can create many more GNN architectures simply by using different neighborhood aggregation functions. We just need to make sure that the function we use is differentiable so we can conduct backpropagation.

4.5 Other Graph Layers

- GATConv: GATConv applies attention to graphs. Instead of the normalization factor, each feature vector is weighted by an embedding similarity metric. You use some similarity metric (e.g., the dot product) so computer the similarity between feature vectors: $a_{ij} = attention(h_i, h_j)$. Then, as always, you pass the attention scores through a softmax to scale the scores.

$$a_{ij} = \frac{e^{a_{ij}}}{\sum_{k \epsilon N(i)} e^{a_{ik}}} \tag{1}$$

  Replace the normalization factor with the attention score to get your updated layer formula.

- GraphSAGE: Employs random sampling of the nodes, with increased depth, for densely connected graphs where it would be computationally inefficient to factor in all the nodes in every layer.

5 Applications

As we have mentioned before, Graph Neural Networks can bee used to perform tasks such as calculating node similarity, node classification, and entire graph classification.

Graphs are widely used to model social media networks. By training a GNN to output embeddings for each account on a network, we may be able to train a classifier to use these embeddings to determine if a certain account is controlled by a human or a bot.

Graphs can also be used to model traffic systems. Google recently used GNNs to improve ETA prediction on Google Maps. You can read about it in this blog post.

GNNs can also be used to model physical systems. Check out this blog post detailing a project that used Graph Neural Networks to model and understand the mobility of glass particles.

6 Implementing GNNs

In order to create GNNs yourself, it's in your best interest to use a library that makes the entire process easier. Two widely used frameworks for graph-based deep learning are PyTorch Geometric (built off of the PyTorch framework), Graph Nets (built off of Tensorflow), and DeepGraphLibrary, or DGL, which can be used with any deep learning framework.

7 Sources and External Links

- Stanford SNAP Lecture Slides (from which most images were taken)
- Stanford CS224w Lecture Video
- Interesting Article on GNNs' connection with Transformers