An Overview of Graph-based Data and Learning Methods

Barry Xue
University of California, San Diego
zex001@ucsd.edu

Abstract

Graphs capture the relationship of data within a large complex system. As the world accumulates more data, we start seeing complex systems with graph structures appear across many fields. Some notable examples include giant online social networks, point clouds and 3D models in computer vision, and molecular structures. In this report, we will summarize the terminology and mathematical foundations related to the study of graph structures (degree distribution, graph Laplacian matrix), identify and present graph-related machine learning tasks (node classification, link prediction), and explain some notable concepts that help to solve those tasks (node2vec, GCN, GCN-AE, and GraphSage). We will also evaluate the performance of these models introduced, and report their performance on the Cora and CiteSeer datasets. At the end, we will examine some practical applications of GCN and GraphSage. The goal of this report is to provide a theoretical foundation to understand our work for the quarter 2 project.

1. Introduction

Today's world is driven by algorithms that feed on data of various forms, including images, text, audio, and much more. Large amounts of data forms complex systems such as video feeds, social networks, molecular structures, etc. Each system has its unique structure and features. As a result, to extract utility from the data, we have considered the underlying structure corresponding to each data type. In this report, we are going to introduce two machine learning tasks on graph type data: node classification and link prediction. In addition to identifying the two graph ML tasks, we will explain the theoretical fundamentals and state-of-the-art techniques for completing these tasks.

1.1 Graph Theory Overview

Graph structures can help capture the structure of data in different complex systems with generalizable features and statistics. This type of characterization allows people to gain a better understanding of each network, compare systems despite their corresponding data types, and enable different machine learning tasks. Figure 1.1 has a list of popular networks and their graph statistics. This is an example of understanding the networks and comparison between networks through their graph statistics.

Network	Nodes	Links	Directed / Undirected	N	L	‹K›
Internet	Routers	Internet connections	Undirected	192,244	609,066	6.34
WWW	Webpages	Links	Directed	325,729	1,497,134	4.60
Power Grid	Power plants, transformers	Cables	Undirected	4,941	6,594	2.67
Mobile-Phone Calls	Subscribers	Calls	Directed	36,595	91,826	2.51
Email	Email addresses	Emails	Directed	57,194	103,731	1.81
Science Collaboration	Scientists	Co-authorships	Undirected	23,133	93,437	8.08
Actor Network	Actors	Co-acting	Undirected	702,388	29,397,908	83.71
Citation Network	Papers	Citations	Directed	449,673	4,689,479	10.43
E. Coli Metabolism	Metabolites	Chemical reactions	Directed	1,039	5,802	5.58
Protein Interactions	Proteins	Binding interactions	Undirected	2,018	2,930	2.90

Figure 1.1: A list of networks with different data types with their corresponding graph statistics (number of nodes, number of links, average degree). [1]

The Network Science by Albert-László Barabási provides a detailed description of different measurements of graphs, such as degree distribution, connectedness measurements, and clustering coefficient. These metrics are useful for the evaluation of different graphs as early analysis, and the insight we learned can then help us diagnose issues related to specific tasks when visualization becomes unreliable [1]. To better explain graph machine learning models, such as Graph Neural Networks (GNNs), we need to understand the degree matrix and the Laplacian matrix.

$$[\mathbf{D}]_{ii} = a_i = \sum_{\mathrm{j}=1}^{\mathrm{n}} a_{ij}$$

$$\mathrm{L} = \mathrm{A}_w - \mathrm{D}$$

The degree matrix (D) is a diagonal matrix with the degree of each node at its diagonal. The Laplacian matrix (L) is the difference between the adjacency matrix and the degree matrix.

Another important concept is random walks on graphs, from which more advanced feature encoding techniques like node2vec and graph convolution are developed. In essence, the random walk procedure correlates the edges of connected nodes with a local probability distribution; by taking steps to traverse the graph according to the local distribution, we generate a larger distribution with the state of the graph at each time step. [2]. With the random walk, we can sample nodes and form subgraphs, and collect the structural context of nodes, all while maintaining control of the computational complexity thanks to its iterative nature. In addition, the linear algebra tools introduced previously enable the walking procedure to be applied on large-scale graph networks, which is ideal when we want to train a model on massive datasets.

1.2 Node2Vec

Node2vec is an algorithm inspired by the ideas of sampling local structure from a graph with the random walk and creating embedding from structural context. Intuitively, node2vec is very similar to word2vec: they both generate embedding for the target based on their context.

There are two benefits of using node2vec embeddings for graph related tasks. First, node2vec transforms the raw connection to a low-dimensional representation; it can be viewed as a type of dimensionality reduction. Second, the node2vec algorithm considers the 2nd-order neighborhood (the immediate connections and their connections) of a node with user-defined bias terms.

Formally, the node2vec algorithm aims to minimize the same skip-gram objective as word2vec:

$$\max_{f} \sum_{u \in V} \log \Pr(N_S(u) \mid f(u))$$

Where u is the target node, $f(u)$ is the node's feature embedding, and $N_s(u)$ is the neighboring nodes, which is the 'context'.

To further formalize the objective, the paper [3] outlined two assumptions: conditional independence, and symmetry in feature space. These two conditions ensure (1) the likelihood to observe each node in the neighborhood is independent, and (2) connected node features have symmetric effect on each other. With these assumptions, the objective is now:

$$\max_f \sum_{u \in V} \left[-\log Z_u + \sum_{n_i \in N_S(u)} f(n_i) \cdot f(u) \right]$$

where $Z_u = \sum_{v \in V} \exp(f(u) \cdot f(v))$ and this can be approximated with negative sampling to save computation time.

Looking at the objective function, it is obvious that we have to define the neighborhood of a given node. And because graphs are not sequential as text, we can't simply borrow the sliding window approach from word2vec. All the algorithms that traverse the graph lie between Breadth First Sampling and Depth First Sampling. The paper proposed a biased random walk procedure to sample local neighbors.

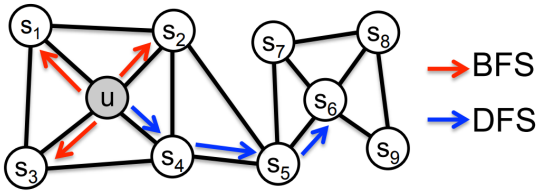


Figure 1.2: BFS and DFS for node u

The original node2vec paper [3] highlights the purpose of the bias terms in their version of the random walk; there are two terms, the in-out parameter and the return parameter, both are defined by a probability provided as hyperparameters.
Normally, the random walk follow:

$$P(c_i = x \mid c_{i-1} = v) = \begin{cases} \frac{w_{vx}}{Z} & \text{if } (v, x) \in E \\ 0 & \text{otherwise} \end{cases}$$

where w is the edge weights. However, w is replaced by the bias term α, which results from the in-out parameter and the return parameter:

$$\alpha_{pq}(t, x) = \begin{cases} \frac{1}{p} & \text{if } d_{tx} = 0 \\ 1 & \text{if } d_{tx} = 1 \\ \frac{1}{q} & \text{if } d_{tx} = 2 \end{cases}$$

The in-out parameter defines the probability for the walk to traverse out of the 1st-order neighbors to the 2nd-order neighbors. The return parameter defines the probability for the walk to return to the original node. With these two parameters tuned, this becomes a biased random walk that lies between complete Breath First Samping, which only considers the 1st-order neighborhood, and Depth First Sampling, which samples across the entire graph.
In summary, node2vec is a scalable, effective method to generate node feature encoding for all types of tasks.

1.3 Graph Convolutional Neural Network (GCN)

Graph Convolutional Neural Network (GCN) [5] is an supervised approach to generating node's structure embedding through convolution of on the graph. The idea remains similar, utilizing the 1st-order connections to make an educated description of the node's role in the local neighborhood of the graph. However, GCNs don't need to generate the node embedding beforehand, which is a separate learning task by itself. Instead, GCNs generate a new embedding and train for a specific task as a part of the model. GCNs do this by convolving the graph's adjacency matrix with the original feature representation of the nodes in an iterative manner. Each iteration of convolution is one convolutional layer; the result of the connected convolutional layers will be fed directly to generate the result.

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$

Formula for a Graph Convolution Layer's output

For example, if we are using a GCN network for node classification, we will input the node connect feature from the adjacency matrix and train that network with the nodes' target labels. This makes GCNs more intuitive to use compared to other models because it doesn't require complex feature engineering and several pipelines to complete one task. In addition, GCNs are highly customizable. We can change the propagation rule of each of the convolution layers to change the network's behavior. More specifically, different propagation rules can be created by changing the normalization factor. And GCNs can adapt to different tasks because we can fit different types of output layers for training. In the experiment section, we will test the performance of GCN networks with node classification tasks.

1.4 Graph Convolutional Autoencoder (GCN-AE)

An Autoencoder is a commonplace neural network architecture consisting of an encoder and a decoder. The encoder will take in an embedding, passing it through a set amount of fully connected layers to generate a lower dimension latent representation of that embedding. The encoder thus can be viewed as a non-linear dimension reduction procedure, which will eliminate unwanted noise in the data to extract useful information. The decoder will use the encoded embedding, and attempt to reconstruct the target objective with the core information.

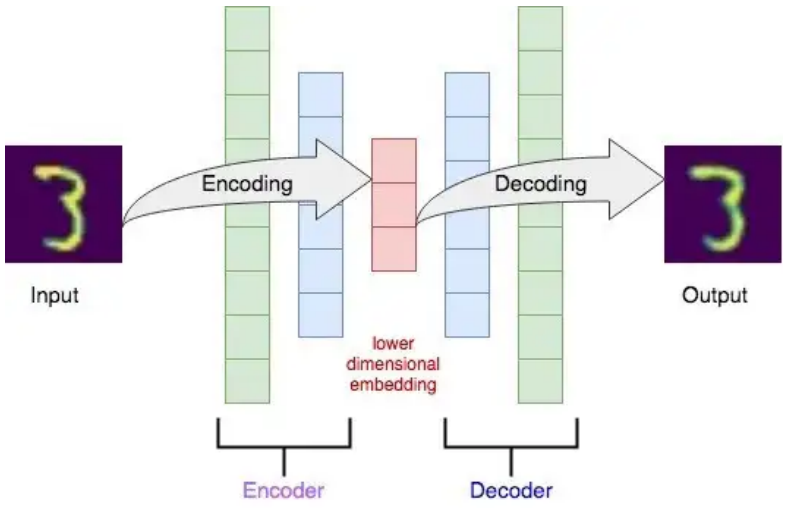


Figure 1.4: Autoencoder Architecture [11]

A *Graph Convolutional Autoencoder* (GCN-AE) follows the same structure. The encoder for a GCN-AE uses one or more graph convolutional layers to create node embeddings at a lower dimension, using the graph's structure.

The decoder's structure depends on the specific tasks we are using GCN-AE for. In the experiment section, we applied GCN-AE for the edge prediction task; to determine whether an edge exists between two nodes, we calculate the dot product of the encoded node embeddings of the corresponding nodes, and apply a sigmoid activation function. Here is a decoder structure that generate the adjacency matrix [8]:

$$\hat{\mathbf{A}} = \sigma\left(\mathbf{Z}\mathbf{Z}^{\top}\right), \quad \text{with} \quad \mathbf{Z} = \mathrm{GCN}(\mathbf{X}, \mathbf{A})$$

1.5 GraphSage

GraphSage stands out compared to the conventional GCNs because it is not a *transductive* learning method, it is an *inductive* learning algorithm. A transductive model requires the adjacency matrix of the graph, i.e. the model needs the entire graph structure to learn node embeddings. This is very computationally expensive because everytime the graph structure is updated, the model has to re-train. This is the case for conventional GCN-based models. GraphSage, as an inductive algorithm, can learn node embedding with an aggregation process. The aggregation is applied on a sampled local neighborhood, so there is no need for retraining everytime the graph structure changes.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Figure 1.5: Pseudocode for GraphSage node embedding learning

The main step in the algorithm is the aggregate-concatenate steps. For any sampled local neighboring nodes, GraphSage aggregates their embeddings and concatenates the aggregated result with the previous node embedding of the target node to generate the new embedding. There are several aggregators for the aggregate step, including min-aggregator, max-aggregator, or even LSTM-aggregator. Most common of all is the mean-aggregator. During the discussion in [10], mean-aggregator is concluded to have the most expressive power and thus is the best choice.

$$\mathbf{h}_v^k \leftarrow \sigma(\mathbf{W} \cdot \text{MEAN}(\{\mathbf{h}_v^{k-1}\} \cup \{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$$

Figure 1.5.2: Formal for Mean Aggregator

The target loss function of learning an embedding is as follow:

$$J_{\mathcal{G}}(\mathbf{z}_u) = -\log\left(\sigma(\mathbf{z}_u^\top \mathbf{z}_v)\right) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} \log\left(\sigma(-\mathbf{z}_u^\top \mathbf{z}_{v_n})\right)$$

The goal of this loss function is to make sure the embedding is similar for nearby nodes and distinct for distanced nodes. The inner product of embedding is passed through a non-linear activation function and then log scaled so the output value is controllable. The second term multiplied by Q signifies negative samples.

2.1 Datasets

To present the aforementioned concepts, we will be using a couple of datasets from CORA [5], PubMed [5], and CiteSeer [7]. CORA is a collection of scientific publications and their citations. There are 2708 publications included with 5429 citations. The CORA dataset also includes the top 1433 unique words for each publication along with their category label. CiteSeer contains 3312 scientific publications. There are a total of 4732 links. For the node features, CiteSeer contains a 3703 word vector for every publication along with their category label. We will be relying on CORA and CiteSeer to evaluate the models'

performance. The PubMed dataset is the biggest among the three. It consists of 19717 nodes and 44338 links. Each of the nodes has 500 word vector features.

2.2 Experiments

We will evaluate the model with respective tasks. We will focus on two tasks for this report: (1) Node classification; (2) Link prediction. For each task, we picked two datasets to evaluate the models against.

Task	**Models**	**Datasets**
Node Classification	Fully Connected Neural Net	1. Cora 2. CiteSeer
	Graph Convolution Neural Net	
	GraphSage	
Edge Prediction	Graph Convolutional Autoencoder	1. Cora 2. PubMed
	GraphSage	

2.3. Experiment Setups

The model setups are listed below:

Models	**Epochs**	**Layer Sizes**
Fully Connected Neural Net	100	{input → 100 → 50}
Graph Convolution Neural Net	300	{input → 1000 → 100 → 50}
GraphSage (Node Prediction)	200	{20, 20}, with sample size {1-hop: 20, 2-hop:10}
Graph Convolutional Autoencoder	100	{input → 1000 → 100 → 50}
GraphSage (Edge Prediction)	50	{20, 20}, with sample size {1-hop: 20, 2-hop:10}

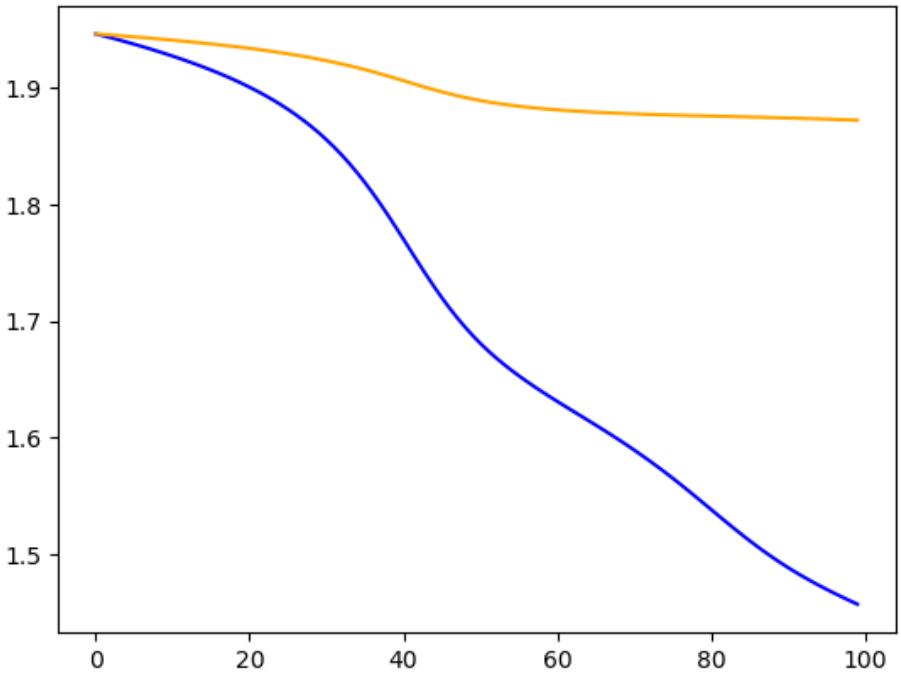
The dataset setup for train, validation, and testing:

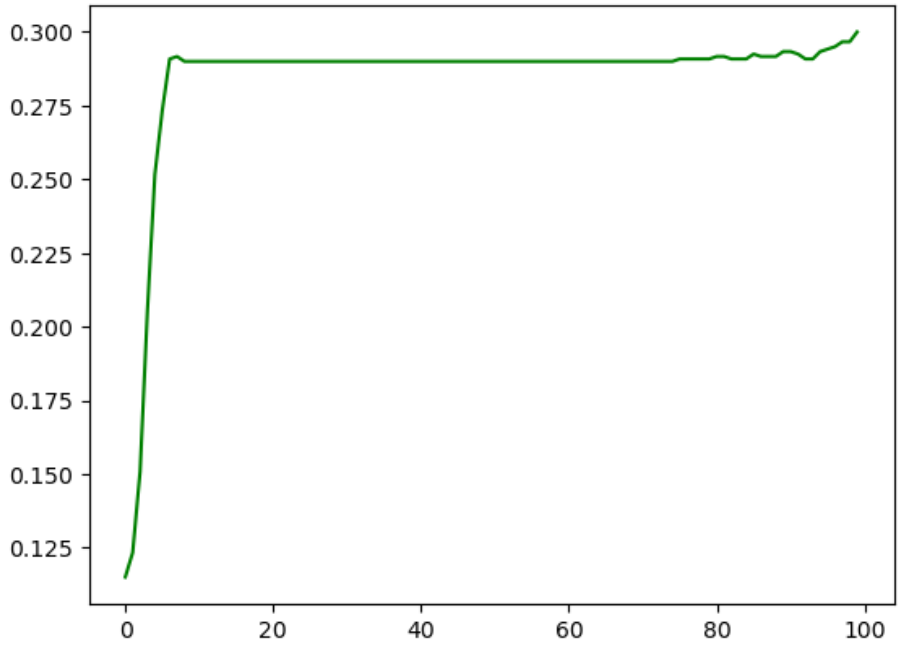
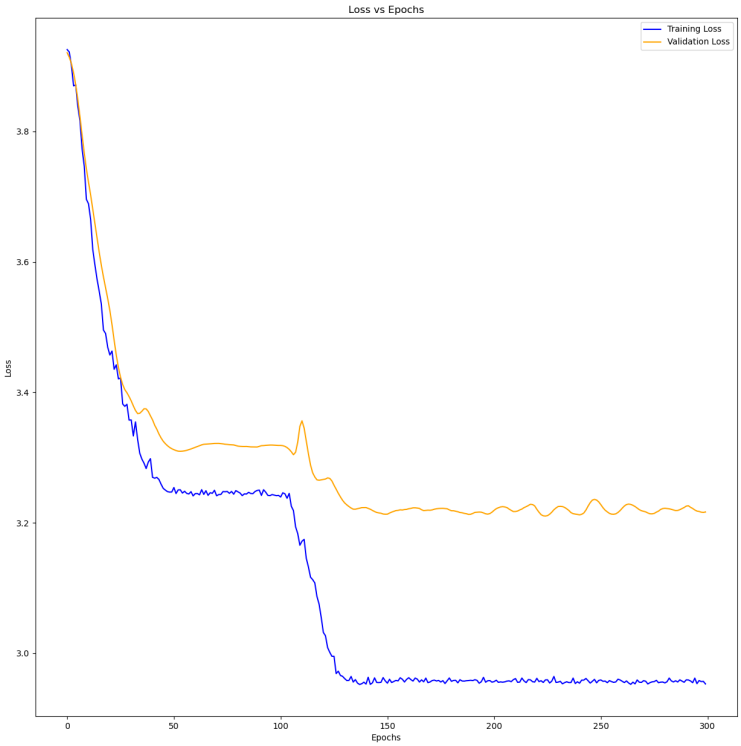
Datasets	**Train**	**Validation**	**Test**
Cora	140	500	1000
CiteSeer	140	500	1000
PubMed	7980	2838	8899

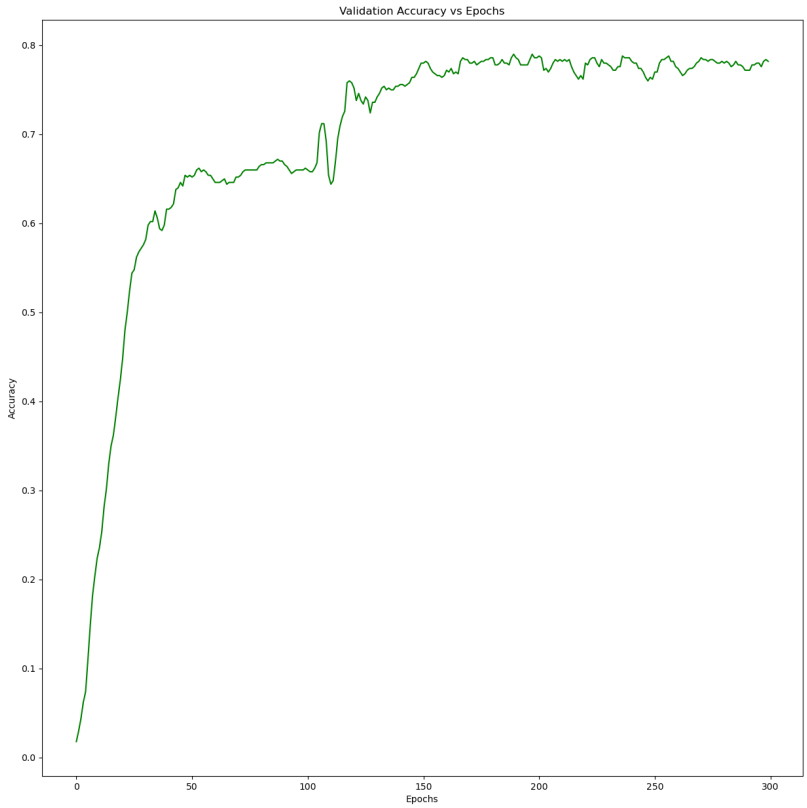
2.4. Results

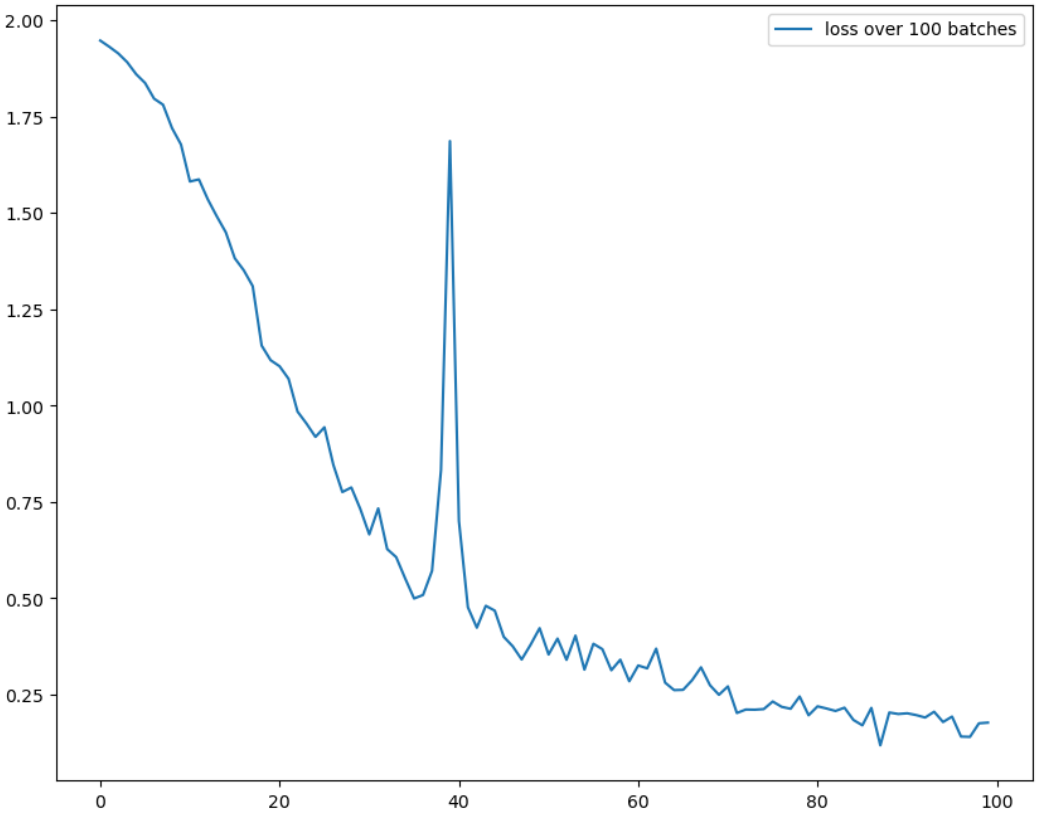
Node Classification:

Model	Test Accuracy - Cora	Test Accuracy - CiteSeer
FCN	0.299	0.335
GCN	0.701	0.657
GraphSage	0.845	0.816

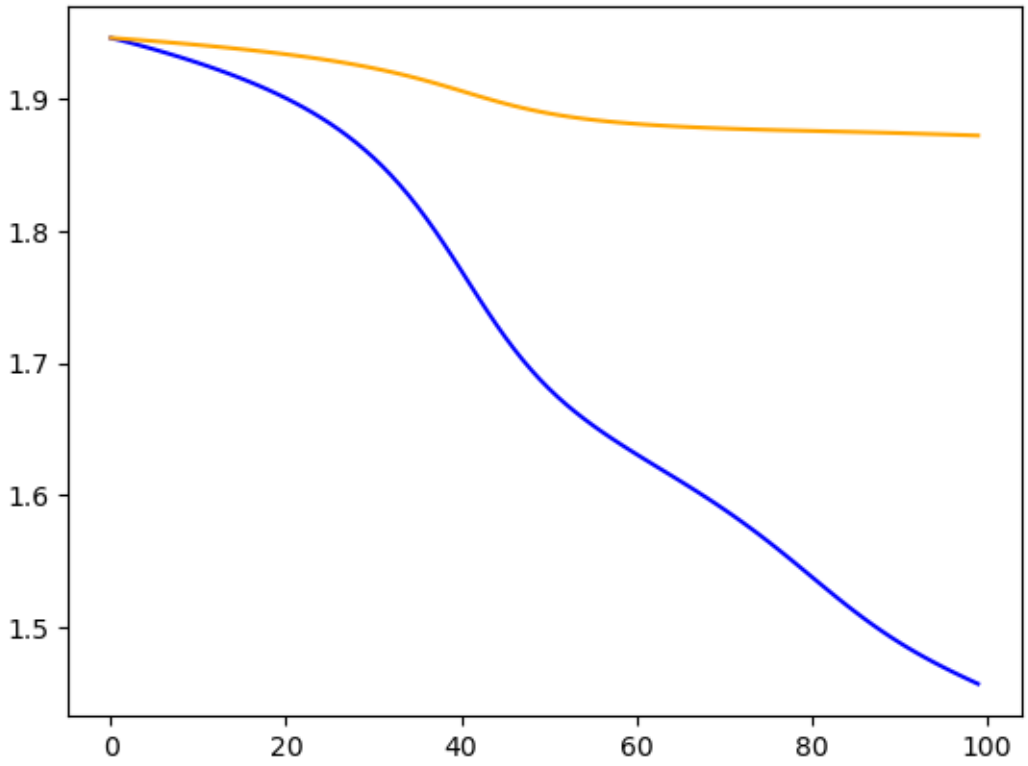
Model	**Training/Validation Trend - Cora**
FCN	Training + Validation Loss  Validation Accuracy

	 0.300 0.275 0.250 0.225 0.200 0.175 0.150 0.125 0 20 40 60 80 100
GCN	 Loss vs Epochs Training Loss Validation Loss Loss 3.8 3.6 3.4 3.2 3.0 0 50 100 150 200 250 300 Epochs

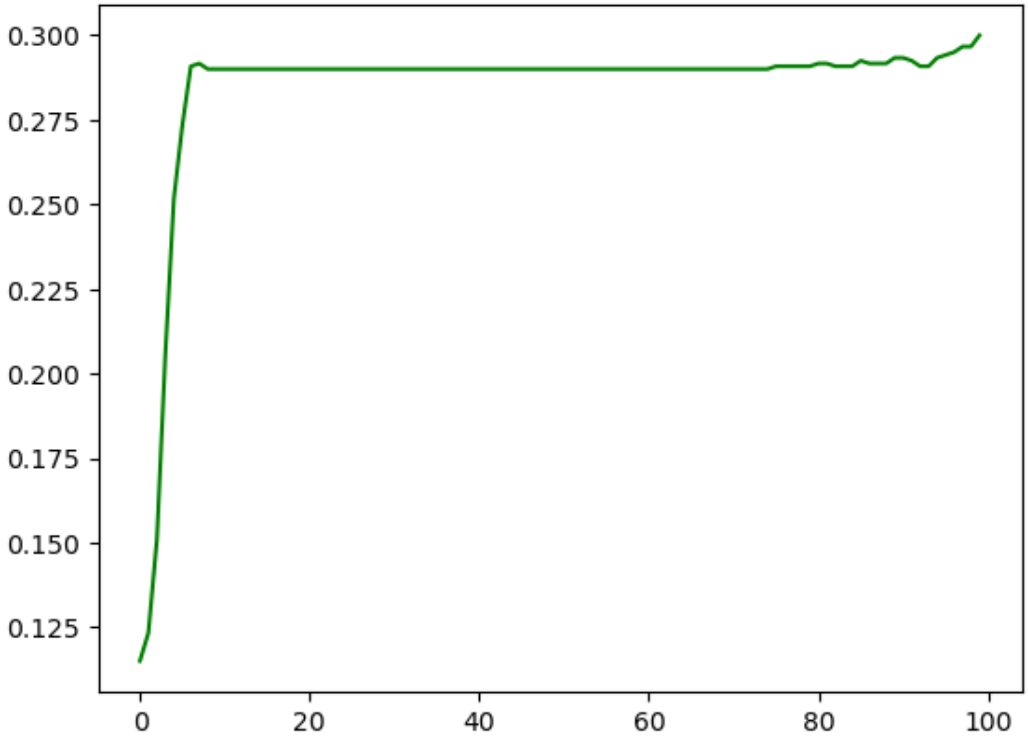

Validation Accuracy vs Epochs
Accuracy
Epochs
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0
50
100
150
200
250
300

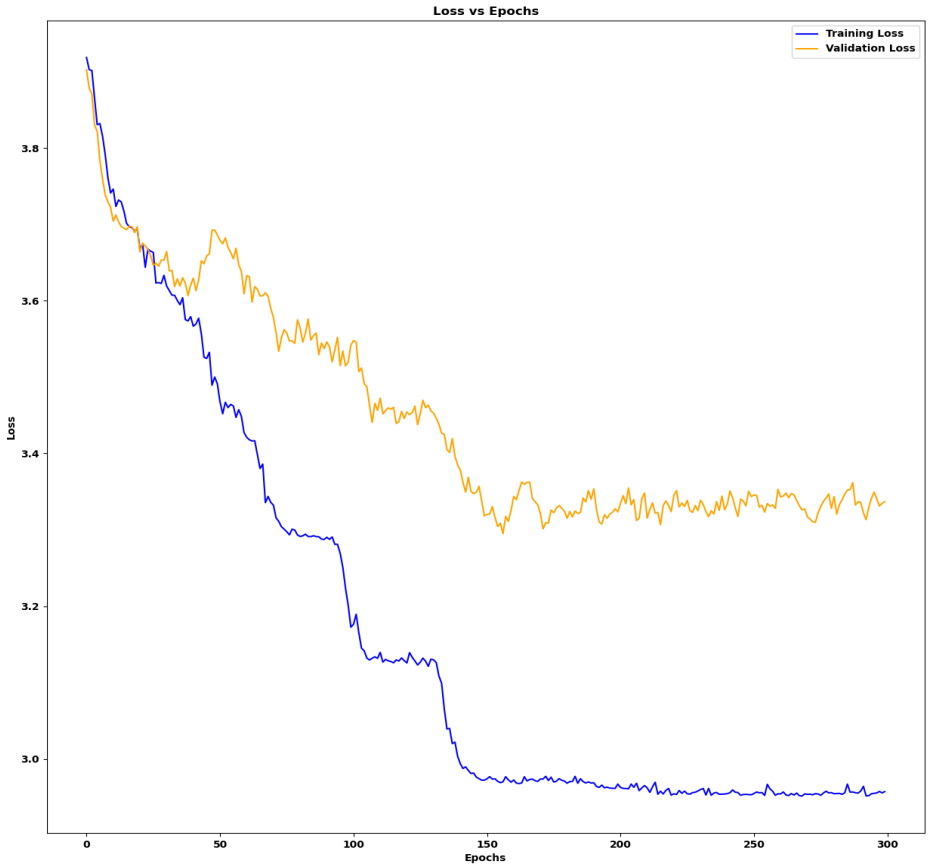
GraphSage	

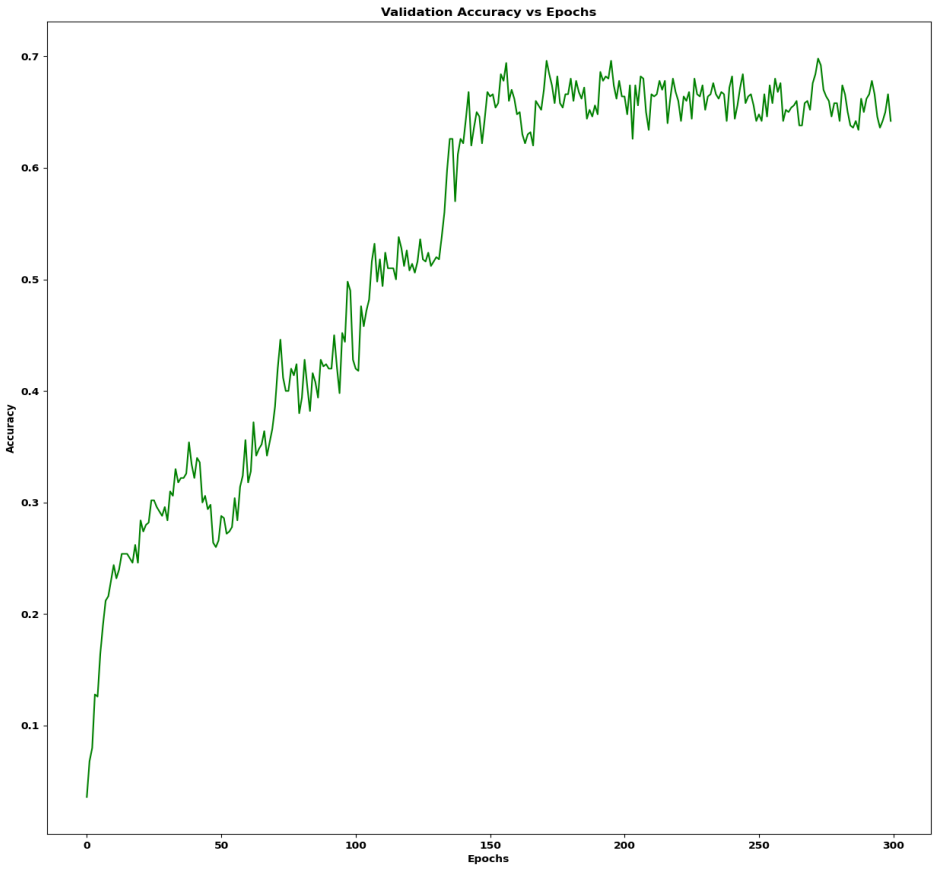
Model	**Training/Validation Trend - CiteSeer**
FCN	Training + Validation Loss

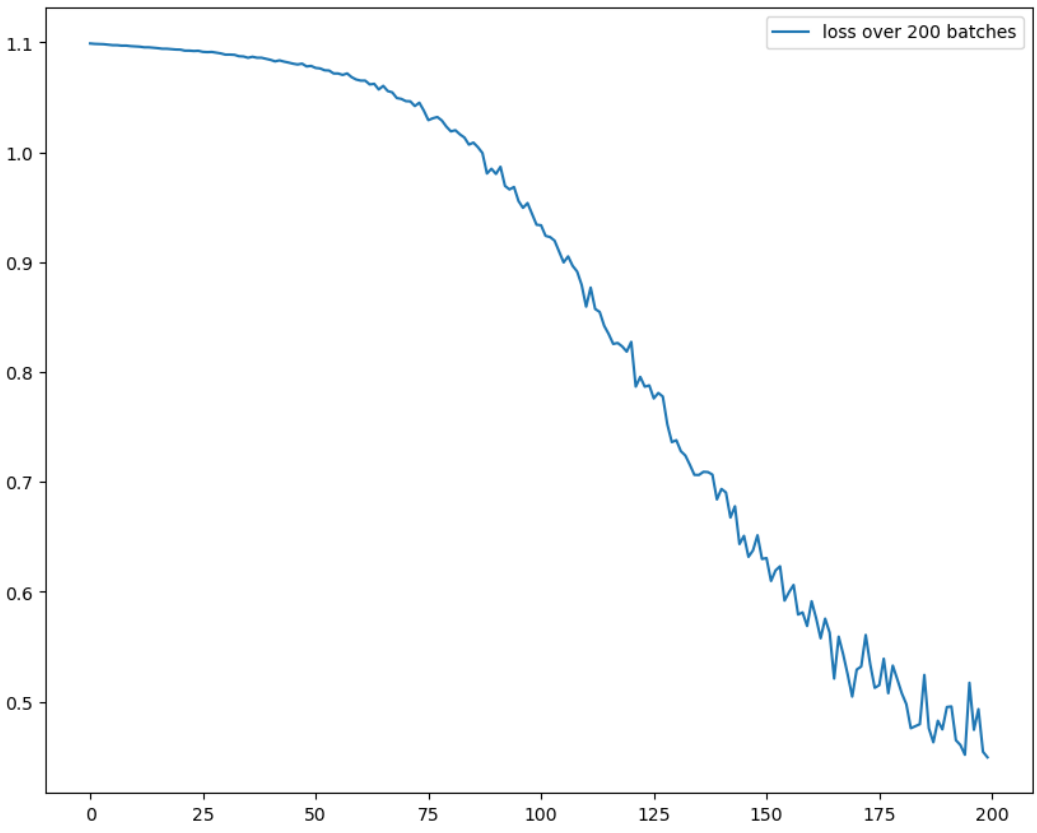


Validation Accuracy



GCN	 Loss vs Epochs Training Loss Validation Loss Loss 3.8 3.6 3.4 3.2 3.0 0 50 100 150 200 250 300 Epochs


Validation Accuracy vs Epochs
Accuracy
Epochs
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
50
100
150
200
250
300

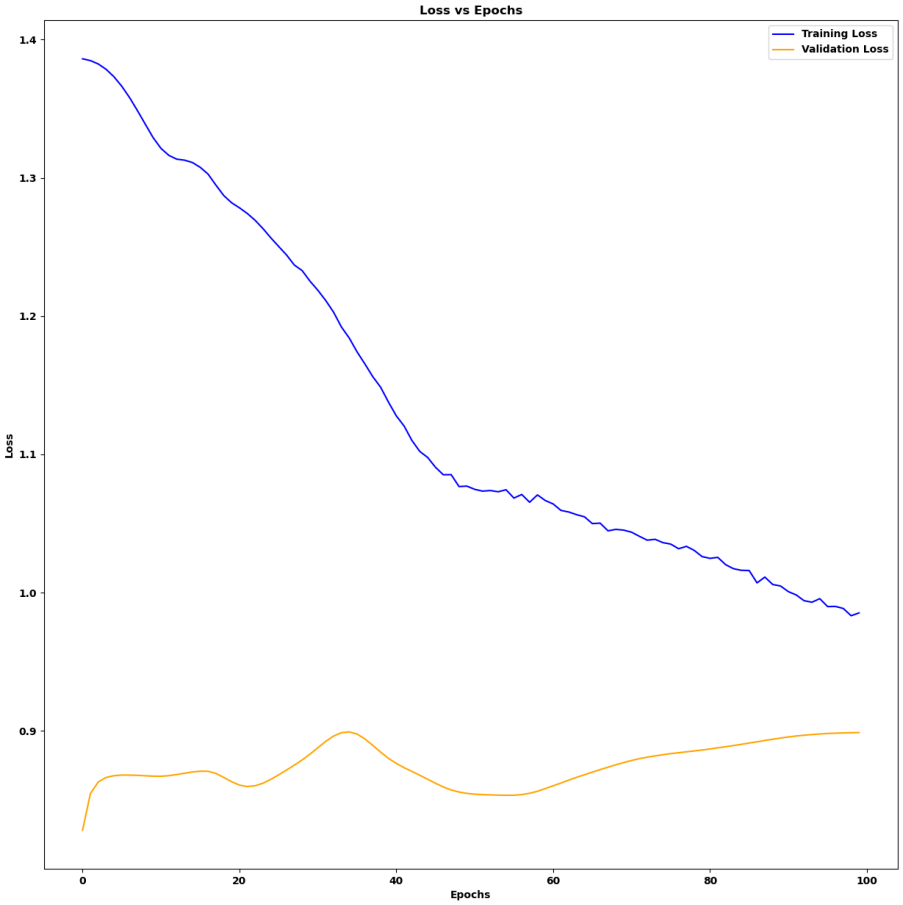
GraphSage	

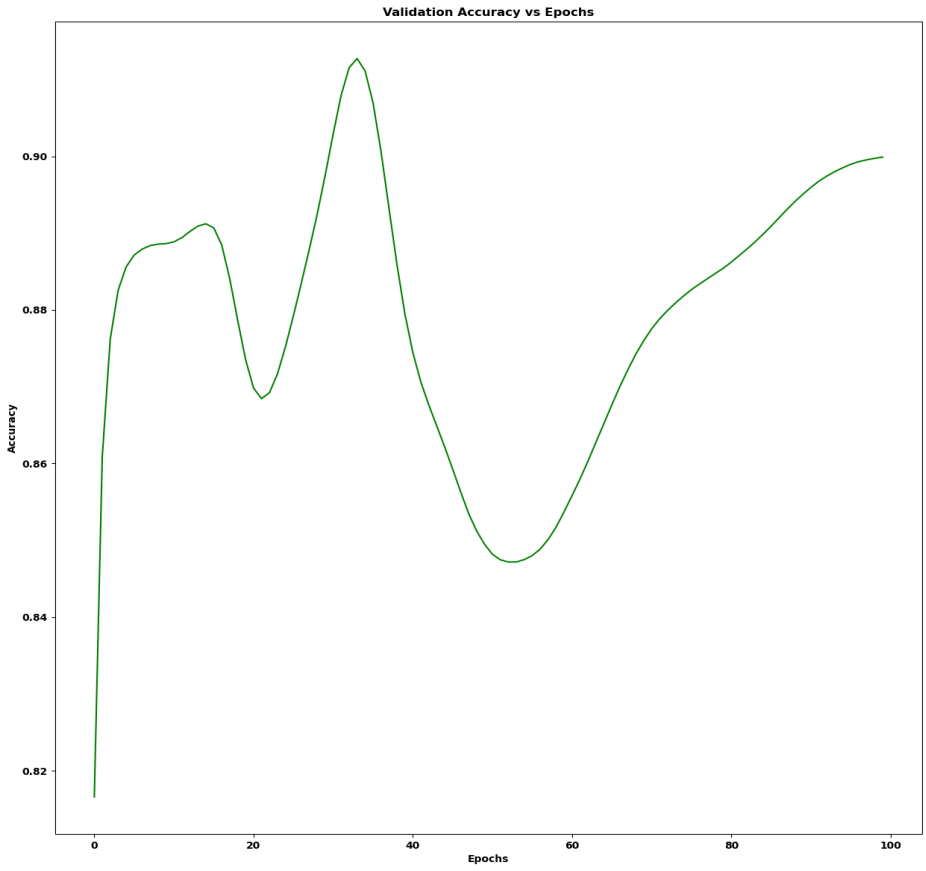
As shown by the result table, GraphSage out performs GCN and FCN on both Cora and CiteSeer dataset. In addition, on node classification tasks, GraphSage's only need 200 epochs of training to perform better than GCN training for 300 epochs.

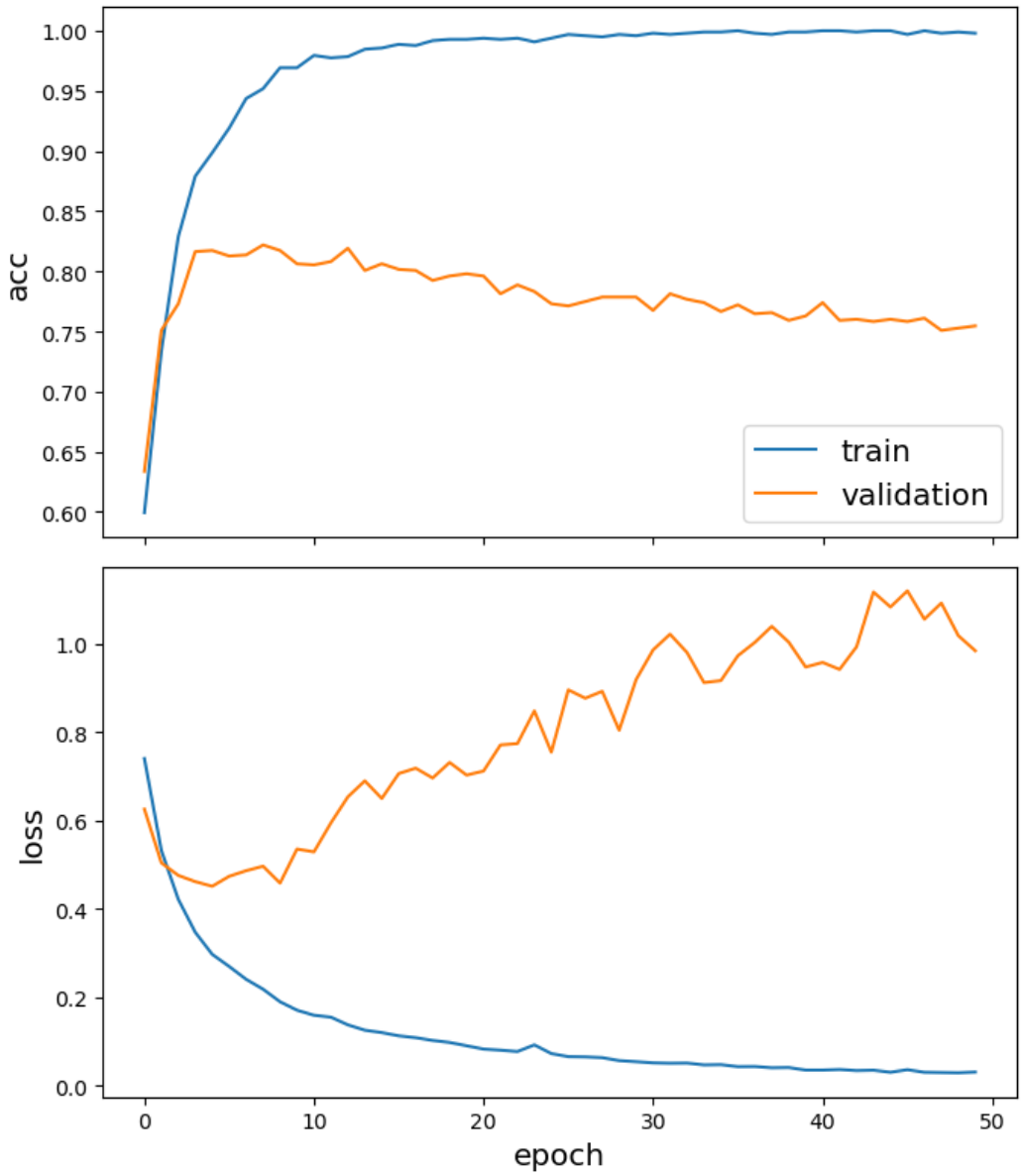
Edge Prediction:

Model	Test Accuracy - Cora	Test Accuracy - PubMed
GCN-AE	0.7943	0.8872
GraphSage	0.7500	0.8728

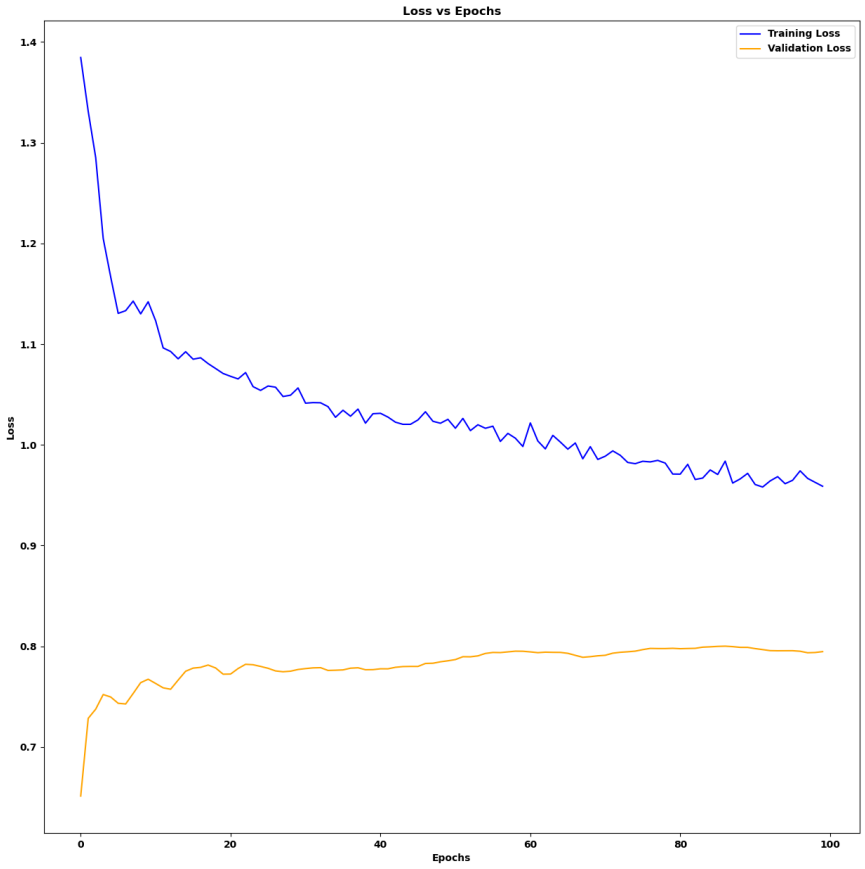
Model	**Training/Validation Trend - Cora**

GCN	 Loss vs Epochs Training Loss Validation Loss Loss Epochs 1.4 1.3 1.2 1.1 1.0 0.9 0 20 40 60 80 100

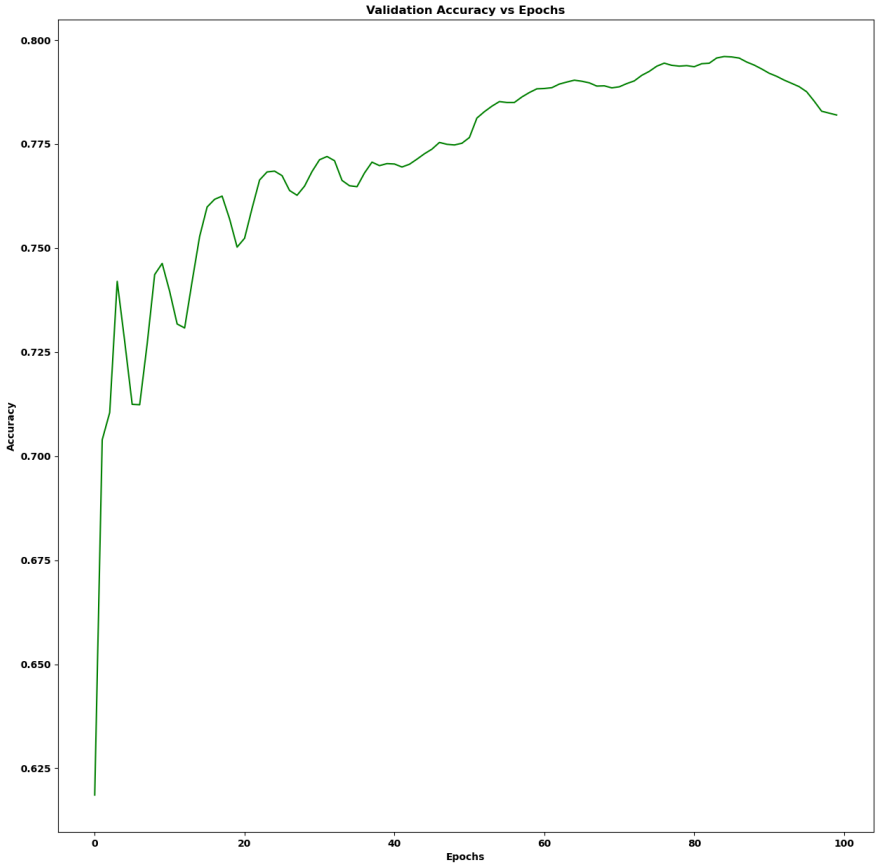

Validation Accuracy vs Epochs
Accuracy
0.90
0.88
0.86
0.84
0.82
0
20
40
60
80
100
Epochs

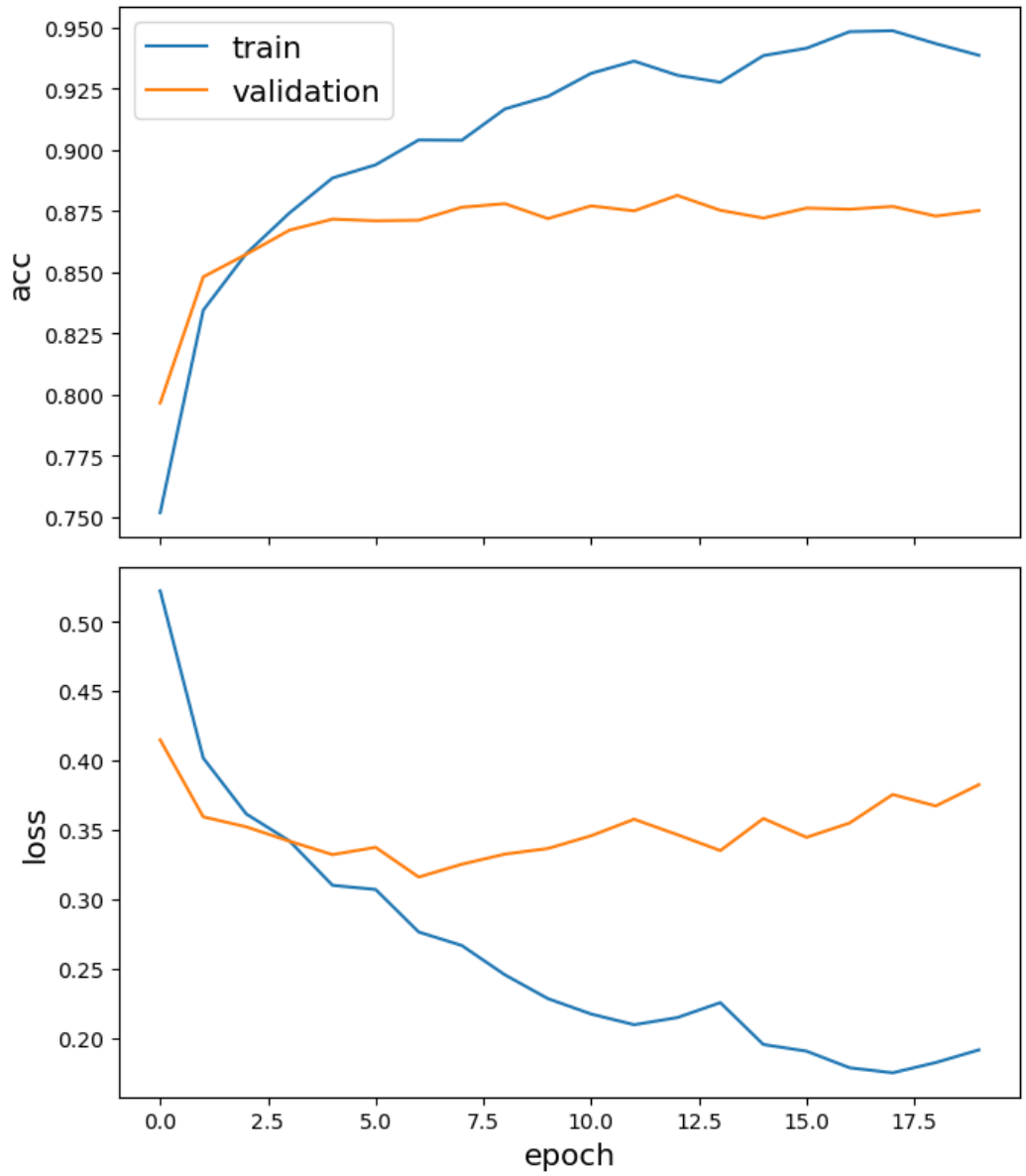
GraphSage	

Model	**Training/Validation Trend - PubMed**

GCN	

Loss vs Epochs
Training Loss
Validation Loss
Loss
Epochs
1.4
1.3
1.2
1.1
1.0
0.9
0.8
0.7
0
20
40
60
80
100


Validation Accuracy vs Epochs
Accuracy
Epochs
0.800
0.775
0.750
0.725
0.700
0.675
0.650
0.625
0
20
40
60
80
100

GraphSage	 Note: this is run for 50 epochs, not 20; there is a mistake in the plot.

GCN-AE and GraphSage have similar performance with edge prediction tasks on both Cora and PubMed datasets.

4. Conclusion

In conclusion, there is clear improvement compared to the models introduced in this report, GCN, GCN-AE, and GraphSage, with the latest approach working the best for both node classification and edge prediction tasks. The main reason for the success of GraphSage is because of its aggregator and propagation approach to working on graphs. It is more modular and versatile than the traditional graph convolution method. In the paper, “How powerful are graph neural networks?”, they outlined a

framework to evaluate Graph Neural Networks' expressiveness, and they are able to prove GINs, similar to GraphSage, has the best possible performance on graph related tasks, because the aggregation and propagation approach with mean aggregator results has the best expressiveness. We hope to explore more of the graph neural network in the upcoming quarter.

Reference

[1] Barabási, A.-L. Network science by Albert-László Barabási. *BarabásiLab*. http://networksciencebook.com/chapter/2#networks-graphs.

[2] Spielman, D. A. *Random Walks on Graphs*. *Spectral Graph Theory*. lecture. http://cs.yale.edu/homes/spielman/561/lect10-18.pdf.

[3] Grover, A., & Leskovec, J. (2016, July 3). Node2vec: Scalable Feature Learning for Networks. *arXiv*. arXiv:1607.00653v1 [cs.SI]. https://arxiv.org/pdf/1607.00653.pdf.

[4] Spielman, D. A. *Random Walks on Graphs*. *Spectral Graph Theory*. lecture. http://cs.yale.edu/homes/spielman/561/lect10-18.pdf.

[5] Kipf, Thomas N., and Max Welling. "Semi-supervised classification with graph convolutional networks." arXiv preprint arXiv:1609.02907 (2016), https://arxiv.org/abs/1609.02907.

[6] Motl, J., & Schulte, O. (2015). The CTU Prague Relational Learning Repository. doi:10.48550/ARXIV.1511.03086. https://arxiv.org/abs/1511.03086.

[7] Yang, Zhilin, et al. Revisiting Semi-Supervised Learning with Graph Embeddings. arXiv, 26 May 2016. arXiv.org, https://doi.org/10.48550/arXiv.1603.08861.

[8] Kipf, Thomas N., and Max Welling. "Variational graph auto-encoders." arXiv preprint arXiv:1611.07308 (2016), https://arxiv.org/abs/1611.07308.

[9] Hamilton, Will, Zhitao Ying, and Jure Leskovec. "Inductive representation learning on large graphs." Advances in neural information processing systems 30 (2017), https://arxiv.org/abs/1706.02216.

[10] Xu, Keyulu, et al. "How powerful are graph neural networks?." arXiv preprint arXiv:1810.00826 (2018), https://arxiv.org/abs/1810.00826.

[11] Han, Fanghao. "Tutorial on Variational Graph Auto-Encoders". Towards Data Science. https://towardsdatascience.com/tutorial-on-variational-graph-auto-encoders-da9333281129.