5-2021

Improving Bayesian Graph Convolutional Networks using Markov Chain Monte Carlo Graph Sampling

Aneesh Komanduri

Improving Bayesian Graph Convolutional Networks using Markov Chain Monte Carlo Graph Sampling

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering

College of Engineering

University of Arkansas

Fayetteville, AR

May 2021

by

Aneesh Komanduri

akomandu@uark.edu

Abstract

In the modern age of social media and networks, graph representations of real-world phenomena have become incredibly crucial. Often, we are interested in understanding how entities in a graph are interconnected. Graph Neural Networks (GNNs) have proven to be a very useful tool in a variety of graph learning tasks including node classification, link prediction, and edge classification. However, in most of these tasks, the graph data we are working with may be noisy and may contain spurious edges. That is, there is a lot of uncertainty associated with the underlying graph structure. Recent approaches to modeling uncertainty have been to use a Bayesian framework and view the graph as a random variable with probabilities associated with model parameters. Introducing the Bayesian paradigm to graph-based models, specifically for semi-supervised node classification, has been shown to yield higher classification accuracies. However, the method of graph inference proposed in recent work does not take into account the structure of the graph. In this paper, we propose Neighborhood Random Walk Sampling (NRWS), a Markov Chain Monte Carlo (MCMC) based graph sampling algorithm that utilizes graph structure, improves diversity among connections, and yields consistently competitive classification results compared to the state-of-the-art in semi-supervised node classification.

Keywords: Semi-Supervised Learning, Graph Neural Networks, Bayesian Inference, Node Classification, Markov Chain Monte Carlo

Contents

1 Introduction

Many phenomena in the world are represented through graphs since they can contain rich relation information among the entities. From modeling molecular structures to social networks, graphs have been a source of tremendous information. Graphs, however, are more useful than for just modeling purposes. We can mine graphs and retrieve insights that could be beneficial in understanding the behavior among connections in the graph. For example, we could analyze a friend network of Facebook users. We can use features such as posts that people like and sources that people follow to make predictions about the extent to which two Facebook friends may enjoy the same content.

A modern application of graph theory is in the field of deep learning. Specifically, analyzing graphs to retrieve useful insights. Graph Neural Networks proposed by Gori et al. [1] have become a useful method for predicting certain behaviors in graphs. Examples include node classification, edge classification, and recommendation systems, among others. This paper focuses on the node classification task. This is a classic classification problem where we are given an observed graph with partially labeled nodes and our task is to train our model on this data so that given a node and its features, the model will be able to predict the correct label of an unlabeled node up to some degree of accuracy. This is an incredibly important task because, in the real world, true labels are expensive to obtain, so we rely on techniques such as random sampling to get these labels and use these labels to develop label prediction models. Additionally, if we are considering demographics, a node connected to another node with a specific label can tell us that the two nodes are likely to share the same node label. For example, if one's friends are all Australian, they are more likely to

be Australian as well based on the correlations in their friend network. Going back to our Facebook example, we are able to monitor a graph at a certain timestep (the observed graph) with nodes representing users and the edges being connections among users. Each user will have some features associated with them. Now, given a certain number of categories that each user can fall into, our task would be to classify an uncategorized user into a class.

To improve on classification accuracies from the GNN, Defferrard et al proposed a spatial variant, [2] the Graph Convolutional Neural Network (GCN). Due to the fact that convolution operations are a spatial operation, meaning that they take into account contextual information, it helps a node in a graph to find its place in the graph with respect to its neighboring nodes. This method has proven to be among the best performing architectures for a variety of graph learning tasks. One caveat with GCNs is that they fail to account for the uncertainty in the underlying structure of a graph such as noisy data with spurious edges. We can tackle this issue by simply viewing the observed graph as a sample from a parametric random variable and target joint inference of the graph and GCN weights using a Bayesian scheme called a Bayesian Graph Convolutional Neural Network (BGCN) [3]. The framework of a BGCN allows graphs to be sampled randomly, which helps to account for uncertainty in the graph thereby increasing accuracy in node classification.

Previous works use an effective scheme called node copying, from Pal et al. [4], to sample graphs from the observed graph under some probability distribution with a fixed hyperparameter. However, a more effective technique could be to use the notion of rejection sampling and Markov Chain Monte Carlo methods such as Metropolis-Hastings to sample graphs based on graph structure. Our method, Neighborhood Random Walk Sampling, allows the sampling to occur with respect to the structure of the observed graph with a random

walk simulating a Markov chain and allows for more diversity in connections among loosely connected nodes. Additionally, our graph inference model yields competitive performance results compared to previous graph inference schemes for BGCNs.

The paper is organized as follows. Section II talks about some of the related work in the field of GCNs and BGCNs and the methods used. Section III explains the Preliminaries and fundamentals of Graphs and Neural Networks. Section IV dives into the mathematics behind sampling from probability distributions. Section V is where the workings of the BGCN are explained and the Metropolis-Hastings graph sampling variant is introduced. Section VI describes the datasets used and outlines the experimentation procedure and summarizes the results from the implementation. Finally, Section VII concludes the paper and discusses potential future work.

2 Related Work

Recent research in the area of graph-based models focuses on leveraging neural networks and deep learning to analyze structured data when there is a graph describing the relationships among the data. Work by Gori et al [1] led to the development of the *graph neural network*, which relies on recursive processing and propagation of information across the graph. However, as the sheer quantity of data has increased in recent times, standard graph neural networks have become undesirable due to long training times and computational inefficiency.

Due to this, the *graph convolutional network (GCN)* was proposed by LeCun et al and spectral methods by Kipf et al [5]. Starting from the framework of spectral graph convolutions introduced in Defferrard et al [2], Kipf and Welling introduce simplifications that allow

for faster training time and higher prediction accuracy.

The GCN, although yielding higher classification accuracies, requires costly matrix operations. To address the key challenges of spectral-based graph neural networks, the *graph attention network (GAT)* [6] was introduced. The GAT leverages masked self-attention layers and stacks layers in which nodes are able to attend over their neighborhood's features.

The original *Bayesian Graph Convolutional Network* [3] was proposed by Zhang et al. One problem with GCNs is that they assume the graph is ground truth. In other words, GCNs don't take into account noise in the data such as spurious edges and weakly tied links. The BGCN is designed to address the problem of uncertainty. Using a Bayesian paradigm, we can model uncertainty in graph data very well. In BGCNs, the graph is viewed as a random variable. So, the parameters in the model are all random variables with probability distributions. Experiments show that BGCNs perform better than GCNs not only in general classification accuracy but also under stricter semi-supervised constraints.

The *Bayesian Graph Convolutional Network with a node copying scheme* [4] was proposed by Pal et al. A graph sampling technique called node copying is proposed, from which the BGCN model is able to perform much more effective graph inference. The node copying scheme is designed to sample from the original graph in a way such that all the nodes are randomly chosen from the same class. The neighbors of a node with the same label are copied to the original node under some fixed probability. The goal of this approach is to provide more dense representations of nodes so the model can yield higher classification results.

3 Preliminaries

3.1 Fundamentals of Graph Representation

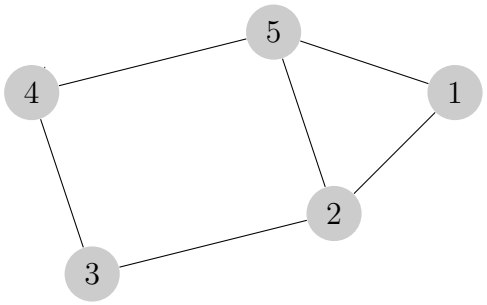
Computationally speaking, a graph is a data structure that models a set of objects (nodes) and their relationships (edges). A graph can be represented as follows

$$\mathcal{G} = (V, E)$$

where V represents the set of nodes and $E = \{(u, v) \mid u, v \in V\}$ represents the set of edges (unordered pairs) in graph $\mathcal{G}$. These edges may be **weighted**, which means that each edge has an associated numeric quantity that may represent the strength of the two entities interlinked, or **unweighted**, which would be just a connection between two nodes. The edges in a graph are often represented with an **adjacency matrix**, **A**. If the edges are weighted, $A \in \mathbb{R}^{V \times V}$ and if the edges are unweighted, $A \in \{0, 1\}^{V \times V}$, where a 1 indicates a connection between two nodes and 0 indicates no connection. In either case, A is a symmetric matrix.

Graphs can be either **directed** or **undirected**. In directed graphs, elements of E are **ordered pairs** $\{(u, v) \mid u, v \in V\}$. That is, (u, v) is distinct from (v, u). By convention, (u, v) points to node v. If both (u, v) and (v, u) are in E, then edges are said to be **mutual**. The following is an example of an unweighted graph and its adjacency matrix.

Observed Graph

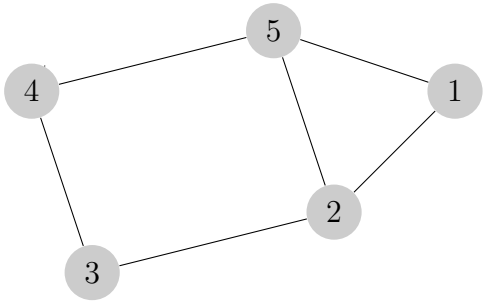


Adjacency Matrix

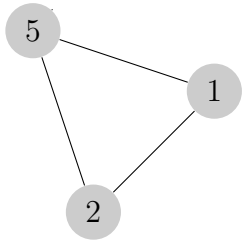
$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

A **subgraph** or an **induced graph** of a graph $\mathcal{G}$ is denoted $\mathcal{G}' = (V', E')$ such that $E' \subseteq E$ and $V' \subseteq V$. That is, any subset of the graph's edges and vertices is considered an induced graph. This concept is crucial for understanding techniques that are discussed later on. Lets take an example:

Observed Graph



Induced Graph



We can see that the observed graph is $\mathcal{G} = (V, E)$, where $V = \{1, 2, 3, 4, 5\}$. The induced subgraph is $\mathcal{G}' = (V', E')$, where $V' = \{1, 2, 5\} \subset V$ and $E' \subset E$.

Definition 3.1.1 An edge (u, v) is incident with the nodes u and v. The **degree**(d_v) of node v is its number of incident edges.

Definition 3.1.2 The **neighborhood** $\mathcal{N}_i$ of a node i is the set of all of its adjacent nodes. Further, $|\mathcal{N}_i| = d_i$

For example, the degree of node 1 in the graph above is $d_1 = 2$ and its neighborhood is $\mathcal{N}_1 = \{2, 5\}$. The value of the degree will range from 0 to $|V| - 1$. The sum of the degree sequence among all nodes is twice the size of the graph $\mathcal{G}$. That is,

$$\sum_{v=1}^{|V|} d_v = 2|E|$$

Oftentimes, we want to traverse through a graph in a certain way. A **path** of length l from v_0 to v_l is a consecutive sequence of distinct vertices such that v_i and v_{i+1} are adjacent for $i = v_0, v_1, \ldots, v_l$. A **walk** is simply a path where vertices do not have to be distinct. This concept will be used later on in the thesis when we talk about **random walks**.

3.2 Neural Networks

The foundation of deep learning is the artificial neural network. A node in a neural network is a structure, emulating a neuron in the brain. Analogous to neurons, we can think about each node as a neuron and the links connecting nodes as the stimuli fired from neuron to neuron, where we have nodes that are connected from one layer to the next. The simplest neural network is known as a perceptron, which is a single-layer neural network. A deep neural network is simply a neural network with multiple hidden layers. Generally, multiple hidden layers are able to learn more complex representations of features and can provide higher classification accuracy as a result. An L-layer neural network is one with an input layer and $L - 1$ hidden layers. Figure 1 shows a fully connected deep neural network. The

goal of a neural network is: given a vector of input features and output classes, learn a function that can best represent the mapping. This is known as supervised learning, where we are given our inputs and output classes ahead of time to train the network. In general machine learning system design, we split the dataset we are given into three partitions, one for training, one for cross-validation, and one for testing purposes. Usually, the majority of the dataset is assigned as training and the other two are split up about proportionally. For example, a common split is 60% training, 20% validation, and 20% testing. The validation set is used to evaluate the performance of the trained model and tune hyperparameters in the model for optimal performance. The test set is to truly evaluate the model on data that the model has never seen before.

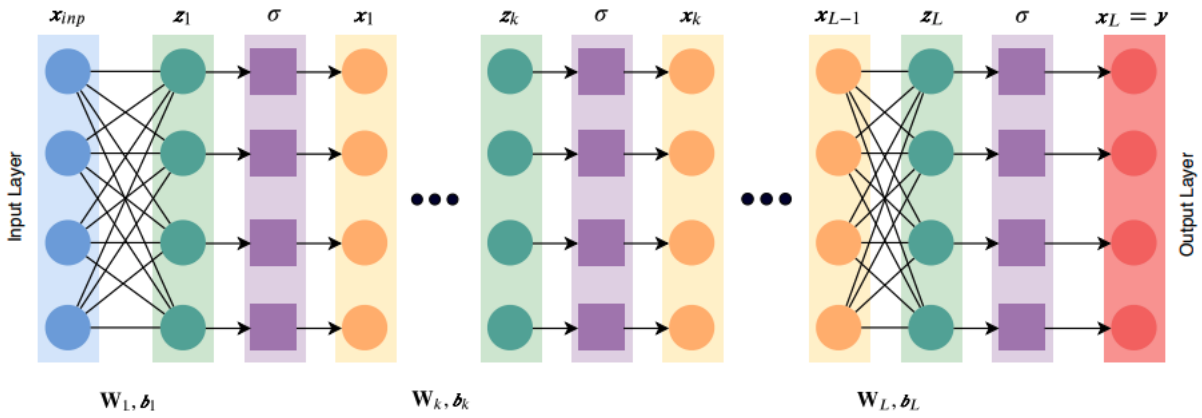


Figure 1: An L-layer deep neural network [7]

With a neural network, each layer has a weight matrix, an input vector, and an activation function to take inputs from one layer to outputs to the next layer. That is, we have that $a_1 = x_{\text{inp}}$ and $a_i = \sigma(Wa_{i-1} + b_i)$, where b_i is some bias unit and a_i is the input vector from layer i. We also apply a non-linear *activation function*, σ to the output to bound

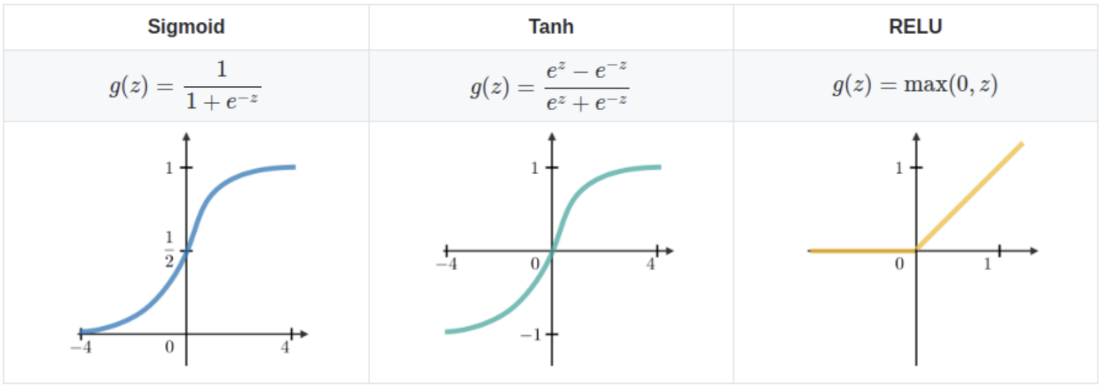


Figure 2: Popular activation functions [8]

the outputs so that they are not too large. Some common activation functions are *ReLU (rectified linear unit)*, *tanh (hyperbolic tangent)*, and *sigmoid* as shown in Figure 2. This process of computing outputs for the subsequent layer given weights and inputs is called a *forward pass*.

Now, we need a method to tell the network about its flaws so that it can learn from the mistakes. This is where the loss function comes in. A *loss function* is a function $\mathcal{L}$ that computes the difference between the ground-truth outputs and the predicted outputs and gives us a metric to represent the amount of loss between the predicted and actual output. The goal of a neural network is to minimize this loss to compute as accurate predictions as possible. Generally, in multi-class classification problems, we use the categorical cross-entropy loss as follows:

$$\mathcal{L}_{cross\_entropy}(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

where y_i is the ground truth class and $\hat{y}_i$ is the predicted class label. To optimize the values of all parameters, we differentiate the loss function with respect to each of the parameters in the model and compute the *gradient*, which tells us how much we should adjust that parameter

to get to a global minimum in the loss. Then, each parameter is adjusted according to the gradient. This process is known as *backpropagation*. It is an incredibly powerful algorithm that is able to optimize loss in neural networks. This optimization procedure occurs a number of times or *epochs*, which is usually set as a hyperparameter. This is the training process with a neural network.

Once the loss is optimized in the training stage, we use the validation set to evaluate the performance of the model while tuning hyperparameters such as the number of epochs or regularization to optimize the loss even further. Finally, the test set is used to evaluate the performance of the model on new data. A common evaluation metric is *accuracy*, which is simply the number of correct predictions out of all predictions.

3.3 Bayesian Inference

Before introducing the Bayesian neural network, we need to understand what bayesian inference really is and how it is applicable. Generally, we are given a dataset, D, that we want to use to make inferences about the world, where we are given a model M that we use to make predictions about the data which is defined as a function of some parameters Θ_M. Now, we are interested in the probability of seeing the data conditioned on a specific choice of parameters from the model. That is, we want to estimate $p(D \mid \Theta_M, M)$. In Bayesian inference, we are interested in the flipped quantity. That is, we are wanting to estimate $p(\Theta_M \mid D, M)$, the probability that the underlying parameters of the model are actually Θ_M given our data and assuming we are using our model, M. Now, **Bayes' Theorem** can help

us compute the posterior as follows:

$$p(\Theta_M \mid D, M) = \frac{p(D \mid \Theta_M, M)p(\Theta_M \mid M)}{p(D \mid M)} \tag{1}$$

The $p(\Theta_M \mid M)$ term is referred to as the **prior** distribution. This probability represents the preconceived notions about the data. The $p(D \mid \Theta_M, M)$ term is called the **likelihood** of the data given the model and its parameters. Lastly, the

$$p(D \mid M) = \int p(D \mid \Theta_M, M)p(\Theta_M \mid M)\, d\Theta_M \tag{2}$$

is the **evidence** or marginal likelihood for our model marginalized over all possible parameter values Θ_M. This value is simply trying to quantify how well our model explains the data after averaging over all possible values of Θ_M of the true underlying parameters [9].

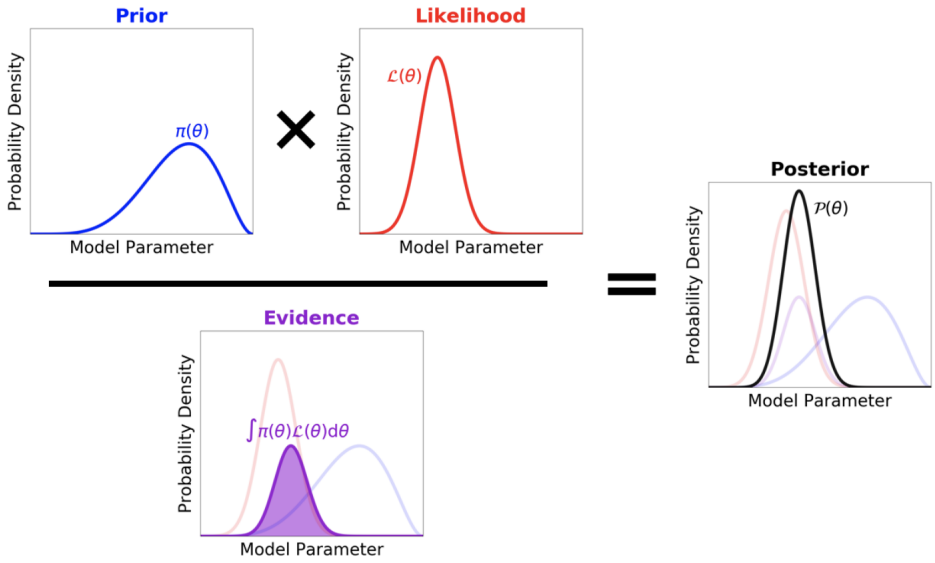


Figure 3: An illustration of Bayes' Theorem [9]

The $p(\Theta_M \mid D, M)$ value is known as the **posterior**, which quantifies our belief in

Θ_M after taking into account our prior intuition along with the current observations $p(D \mid \Theta_M, M)$ and normalizing by the overall evidence $p(D \mid M)$. The posterior probability can be thought of as a compromise between the prior and the likelihood.

3.4 Bayesian Neural Networks

A common definition of a Bayesian neural network is a stochastic artificial neural network that is trained using Bayesian inference [10]. Generally, in regular neural networks, the network weights are not treated as random variables. Rather, they are assumed to have a true value that may not be known yet. Bayesian Neural Networks (BNNs) treat weights as a random variable and learn the model weights based off information we have at hand. The goal is to learn a distribution of the weights (or any relevant parameters) conditional on what we observe in the data [11]. During the process of training and learning of Bayesian neural networks, unknown model weights are inferred based on what we can observe or know already. This is known as *inverse* probability and can be solved by the use of Bayes' Theorem. Let $\mathcal{W}$ represent the weights of our model. Since we can't immediately assume a true distribution for these weights, we can use Bayes' theorem to introduce a distribution over the weights conditional on the data we observe $p(\mathcal{W}|\mathcal{D})$ (the posterior distribution). We can initially observe the joint distribution between the model weights and the data $p(\mathcal{W}, \mathcal{D})$, which is defined by our prior beliefs and our choice of model (likelihood). That is,

$$p(\mathcal{W}, \mathcal{D}) = p(\mathcal{W})p(\mathcal{D}|\mathcal{W}) \tag{3}$$

With the prior and likelihood, we can now use Bayes' Theorem to specify how the pos-

terior distribution may be obtained. Specifically,

$$p(\mathcal{W}|\mathcal{D}) = \frac{p(\mathcal{W})p(\mathcal{D}|\mathcal{W})}{\int p(\mathcal{W})p(\mathcal{D}|\mathcal{W})\, d\mathcal{W}} = \frac{p(\mathcal{W})p(\mathcal{D}|\mathcal{W})}{p(\mathcal{D})} \tag{4}$$

The denominator in the posterior distribution is known as the marginal likelihood or the evidence. The reason for this quantity is to act as normalization to ensure that the posterior is a valid distribution.

Now, we can apply this process to a supervised learning classification setting. Let $\mathbf{X} = \{x_1, \ldots, x_n\}$ be the training inputs and $\mathbf{Y} = \{y_1, \ldots, y_n\}$ be the corresponding outputs. The goal is to learn a function $y = f(x)$ using a neural network to find a relationship between x and y. Using the Bayesian framework, we model the weights of our network $\mathcal{W}$ as a random variable with a prior distribution introduced over them. The weights are not a deterministic parameter, so the outputs of the neural network will also be a random variable. So, we compute the marginal likelihood of the outputs conditioned on the set of inputs and outputs as follows

$$p(y|x, \mathbf{X}, \mathbf{Y}) = \int p(y|x, \mathcal{W})p(\mathcal{W}|\mathbf{X}, \mathbf{Y})\, d\mathcal{W} \tag{5}$$

where $p(y|x, \mathcal{W})$ is the likelihood that can be computed by applying a softmax function to the output of the network. This integral is intractable, so we can use Monte Carlo techniques, which are discussed later in Section 4.2, to approximate the integral. Namely,

we can approximate the posterior as follows

$$p(y|x, \mathbf{X}, \mathbf{Y}) \approx \frac{1}{T} \sum_{i=1}^{S} p(y|x, \mathcal{W}_i) \tag{6}$$

where T is the number of samples drawn, S is the number of weights $\mathcal{W}_i$ obtained using Monte Carlo dropout [3].

3.5 Geometric Deep Learning

3.5.1 Graph Neural Networks

Graph reasoning models have become immensely useful in research topics pertaining to social networks. The Graph Neural Network (GNN), proposed by Gori et al. [1], is a connectionist model that can capture the dependence of graphs using a technique known as message passing. Unlike the regular counterparts, Graph Neural Networks are able to aggregate information from neighboring nodes and pass it along as a hidden state into the node being observed. This allows GNNs to model inputs and outputs that have dependencies in a graph [12]. Graphs are a complex data structure so it is difficult for standard neural networks such as Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN) to model them effectively. GNNs disregard the input order of nodes. Rather, they propagate on each node respectively. So, the output of a GNN has nothing to do with the ordering of the nodes in the graph. Edges in a graph represent dependency between the interlinked nodes. Instead of using these dependencies as features, GNNs are able to propagate based on the structure of the graph. Perhaps the most impressive, it has been proven that untrained

GNNs can perform relatively well on graph datasets.

In a graph, each node is defined by its features and its connections to neighboring nodes. The goal of a GNN is to attempt to learn a state embedding $h_v \in \mathbb{R}^s$ of the neighborhood of a node v. This embedding can be used to generate an output o_v, which in this case can represent the class of the node. Suppose f is a local transition function shared among all the nodes in the graph and updates the state of the node according to the input neighborhood. Let g be a local output function that generates the output. Then, we define the embedding and output as follows [1]:

$$h_v = f(x_v, x_{co[v]}, h_{ne[v]}, x_{ne[v]})$$
$$o_v = g(h_v, x_v)$$

where $x_v, x_{co[v]}, h_{ne[v]}, x_{ne[v]}$ are the node features of node v, the edge features, the hidden states, and the node features of the neighboring nodes, respectively. More generally, for all nodes in the graph, we have the following compact representation

$$H = F(H, X)$$
$$O = G(H, X_N)$$

where H, O, X, and X_N are vectors generated by stacking all the values of the respective quantities for each node, and F and G are global functions. In GNNs, all nodes in the graph are updated in parallel at the same time. So, we can run it for a certain number of iterations, where each iteration is a timestep of the whole graph's state at that point. That is,

$$H^{t+1} = F(H^t, X)$$

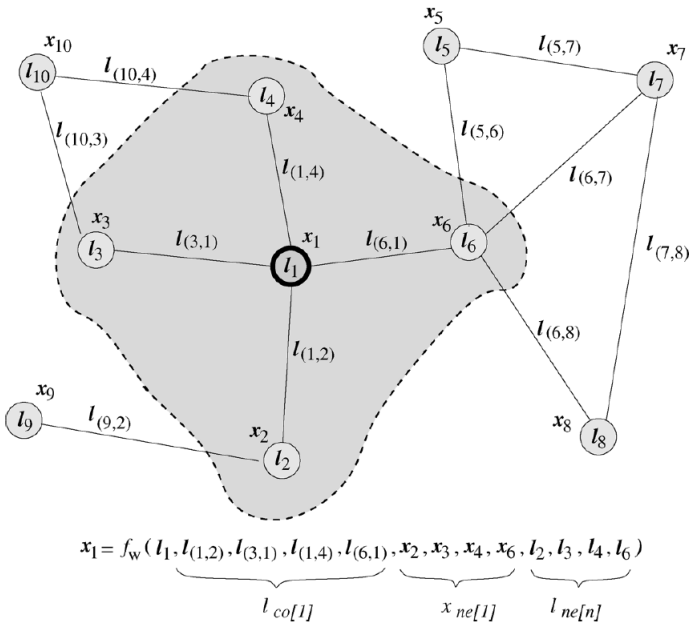


Figure 4: Graph and neighborhood of a node [1]

where H^t represents the t-th iteration of H. Now, given a target t_v (perhaps a class label) for node v, the generic loss function is defined as follows:

$$\mathcal{L} = \sum_{i=1}^{p} (t_i - o_i)$$

where p is the number of nodes that are labeled through supervision. The loss and weights are optimized via standard gradient descent. Figure 4 shows an illustration of how the GNN operates on graph structure.

Regardless of all the benefits that GNNs provide when modeling a complex structure such as a graph, they still have their limitations. For instance, the iterative update of the hidden states of the node is quite inefficient. Also, GNNs lack representation of nodes since the representation at a fixed point in time is not as informative when it comes to distinguishing nodes.

3.5.2 Graph Convolutional Networks

Due to the problems outlined above, the Convolution operation was introduced in graphs to boost the GNNs efficacy. The Graph Convolutional Neural Network, proposed by Defferrard et al [2] and Kipf et al [5], is a variant of the GNN. For the sake of comparison, we will compare graph convolutions with image convolutions. In the context of image data, a convolution operation aggregates the features of local pixels sequentially while reducing the spatial dimension by clustering features of nearby pixels. A convolution on a graph is similar, except the pixels are instead a varying number of neighboring nodes. Images have a fixed structure, but graphs have a variable number of neighbors, so kernels are generally more difficult to define.

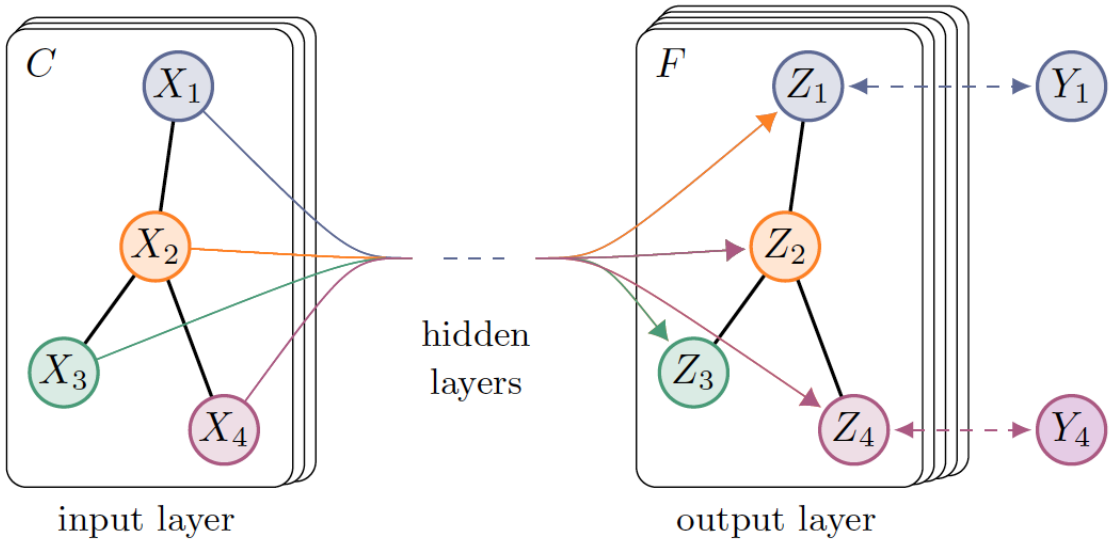


Figure 5: GCN Architecture [5]

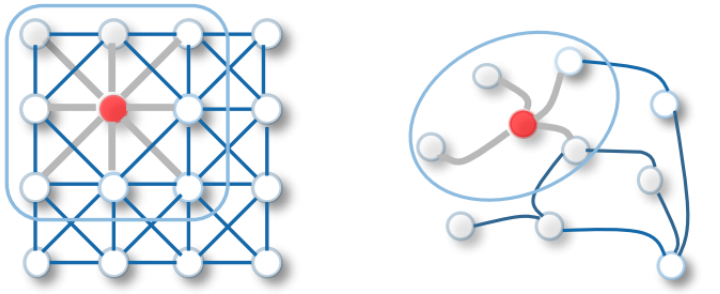
As shown in Figure 5, the convolution operation takes the input graph, propagates through hidden layers, and flattens the representation by aggregating features and ultimately condensing it down to a layer with labels. Once this is done, a simple softmax can

be taken to determine the probabilities of each label.

The Propagation Rule of the GCN is as follows:

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

where $H^{(l+1)}$ represents the hidden features (activations) at the next hidden layer, $\tilde{A}$ is the adjacency matrix with self-loops accounted for so that each node includes its own features at its next representation, and $\tilde{D}$ is the Degree Matrix of $\tilde{A}$ which is used to normalize nodes with large degrees because otherwise, nodes with a large number of neighbors can be computationally demanding. We have that $(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}) \in \mathbb{R}^{n\times n}$, $H^{(l)} \in \mathbb{R}^{n\times d}$, and $W^{(l)} \in \mathbb{R}^{d\times n}$ [5]. Intuitively, we can see the difference between a 2D image convolution and a graph convolution in Figure 6.



(a) 2D Convolution. (b) Graph Convolution.

Figure 6: Graph convolution analogous to image convolution [12]

We can see that (a) in the figure is the aggregation of pixels in a neighborhood to perform a convolution and reduce the spatial dimension. In (b), we can see that the aggregation is from neighboring nodes that may differ in size.

4 Sampling Theory

Before diving into our approach, we must define some important results in probability sampling. Often, in the bayesian paradigms, we are interested in sampling from some posterior distribution that we may not know. That is, we are interested in sampling from some proposal distribution that is close to the posterior in an attempt to sample from the posterior distribution. We can estimate sampling from a posterior by building a Markov chain that converges to some target distribution. The following subsections explain each of these methods and how they are used together to perform effective sampling.

4.1 Markov Chains

Markov Chains are stochastic random processes that can describe a sequence of events where the probability of each event depends only on the state of the previous event. This property is powerful and is referred to as the **Markov Property**. More formally, we look at the following definitions.

Definition 4.1.1 *Let I be a countable set. Then, each element $i \in I$ is called a **state** and I is called the **state-space**.*

Definition 4.1.2 *Suppose $(X_t)_{t\geq 0}$ are all random variables that map an event to a state in the state-space I. We say that $(X_t)_{t\geq 0}$ is a Markov chain with initial distribution π if for all $t \geq 0$ and $i_0, \ldots, i_t$,*

- $\mathbb{P}(X_0 = i_0) = \pi_{i_0}$
- $\mathbb{P}(X_t = i_t \mid X_{t-1} = i_{t-1}, X_{t-2} = i_{t-2}, \ldots, X_0 = i_0) = \mathbb{P}(X_t = i_t \mid X_{t-1} = i_{t-1})$

for some time t [13].

This implies that the probability distribution at time t, given all the previous events of the chain, is equal to the probability distribution given only the previous event. That is, we can determine the next state solely based on our current state in the chain. But on what basis do we decide to go to the next state? The **transition matrix**

$$P = (p_{ij} : i, j \in I, p_{ij} \geq 0 \text{ for all } i, j)$$

defines the probability of transitioning from state i to state j. This matrix is a **stochastic matrix**, which means that $\sum_{j \in I} p_{ij} = 1$. We can see a simple Markov chain in Figure 3.

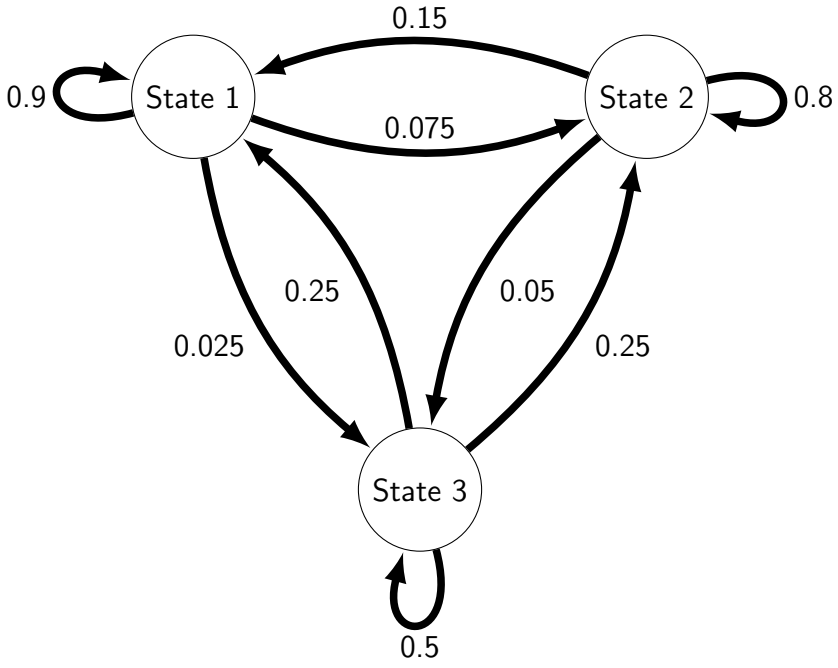


Figure 7: An example of a Markov Chain

From this state machine, we can construct a transition matrix with each row and column representing the respective state. The weighted arrows will be entries in the matrix. For the Markov chain in Figure 7, our transition matrix would be

$$\begin{pmatrix} 0.9 & 0.075 & 0.025 \\ 0.15 & 0.8 & 0.05 \\ 0.25 & 0.25 & 0.5 \end{pmatrix}$$

The matrix shows that the probability of transitioning from state 1 to state 2 is 0.8 since the $(1, 2)$ entry in the matrix is 0.8. If we are on iteration t of the chain, then

$$\mathbb{P}(X_{t+1} = j \mid X_t = i) = P_{ij}$$

Suppose we start the chain at state 2 with a probability of 1. So, we have an initial probability distribution $\pi_0 = (0, 1, 0)$. After one iteration, the probability distribution of the states is

$$\pi_1 = \pi_0 P = (0.15, 0.8, 0.05)$$

That is, after an iteration, we can be in state 1 with 0.15 probability, stay at state 2 with 0.8 probability, or be in state 3 with 0.05 probability. We can actually run the chain for n iterations so that we have the probability distribution after n iterations as follows

$$\pi_n = \pi_0 P^n = P^{(n)}$$

Now that we have defined some terminology when working with Markov chains, we can

explore some of the important properties that make Markov chains so useful in sampling from distributions. Firstly, we define a basic limit theorem.

To understand this theorem, we start with an example. Suppose we have a clock with 6 numbers on it: $0, 1, 2, 3, 4, 5$. We perform a random walk by moving either clockwise, counterclockwise, or staying in place all with probability $\frac{1}{3}$ at time n. That is,

$$P(i,j) = \begin{cases} 1/3 & \text{if } j = (i-1) \mod 6 \\ 1/3 & \text{if } j = i \\ 1/3 & \text{if } j = (i+1) \mod 6 \end{cases}$$

Suppose we start at state $X_0 = 2$. This implies that

$$\begin{aligned} \pi_0 &= (0,0,1,0,0) \\ \pi_1 &= \pi_0 P = (0, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, 0, 0) \\ \pi_2 &= \pi_1 P = (\frac{1}{9}, \frac{2}{9}, \frac{1}{3}, \frac{2}{9}, \frac{1}{9}, 0) \end{aligned}$$

We can see that the initial probability, concentrated at state 2 is now spreading out to the other states. The probability will continue spreading until π_n approaches the following uniform distribution:

$$\pi_n \to (\frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6})$$

as $n \to \infty$. We could have started anywhere on the chain and we would end up with the same uniform distribution. That is, what the distribution converges to does not depend upon the

initial distribution. The Basic Limit Theorem states this in more general terms [14].

Before we state the theorem, we must define what it means for a markov chain to be **irreducible** and **aperiodic** having a **stationary distribution**.

Definition 4.1.3 [14] Suppose a distribution π on I such that, if the Markov chain starts out with initial distribution $\pi_0 = \pi$, then we also have $\pi_1 = \pi$, and generally, $\pi_n = \pi$ for all n. Then, we call π the **stationary distribution** for the Markov chain. That is, for all $j \in I$,

$$\pi(j) = \sum_{i \in I} \pi(i) P(i, j)$$

Definition 4.1.4 [14] Let i and j be two states. We say that j is **accessible** from i if it is possible, with a positive probability, for the chain to visit state j if the chain starts in state i. That is,

$$\mathbb{P}_i \left\{ \bigcup_{n=0}^{\infty} \{X_n = j\} \right\} > 0$$

We say that i **communicates** with j if j is accessible from i and i is accessible from j. We say that a Markov chain is **irreducible** if all pairs of states communicate.

Definition 4.1.5 [14] Given a Markov chain $\{X_0, X_1, \dots\}$, define the **period** of a state i to be the greatest common divisor

$$d_i = gcd(n \mid P^n(i, i) > 0)$$

An irreducible markov chain is said to be **aperiodic** if its period is 1, and **periodic** otherwise.

We now state the Basic Limit Theorem, but due to the level of involvement in proving

the Basic Limit Theorem, the proof is omitted.

Theorem 1 (Basic Limit Theorem) Let $X_0, X_1, \ldots$ be an **irreducible**, **aperiodic** Markov chain having a **stationary** distribution $\pi(\cdot)$. Let X_0 have the distribution π_0, an arbitrary distribution. Then,

$$\lim_{n\to\infty} \pi_n(i) = \pi(i)$$

for states $i \in I$.

Now, we define another important property of Markov chains that we should be aware of. A Markov chain is **time reversible** if

$$(X_0, X_1, \ldots, X_n) = (X_n, X_{n-1}, \ldots, X_0)$$

That is, the sequence of states that are moving in the "forward" direction is equal in distribution to the sequence of states moving in the "backward" direction. This means that the element-wise states are equal in distribution as well, which implies that the distribution must be stationary. The property of time reversibility is important because it is relevant in Markov Chain Monte Carlo (MCMC) methods since it allows for the construction of a proper Markov chain from which to simulate a posterior distribution. A Markov chain's stationary distribution can well simulate the posterior distribution that we have trouble sampling from [15].

4.2 Monte Carlo Sampling

In many high-dimensional sampling problems, a technique known as Monte Carlo Sampling can be used. This technique is specifically used in two kinds of problems [16]:

- To sample from a posterior distribution $p(x)$
- Approximating integrals of the form $\int f(x)p(x)\,dx$

First, we introduce a couple of definitions to lead into Monte Carlo sampling and then we introduce two important techniques of sampling.

Definition 4.2.1 [17] Suppose we have some d-dimensional data $x \in \mathbb{R}^d$. Let $f(x)$ be the Probability Density Function (PDF) of the data. Let $h(x)$ be another function over the data x. Then, the expectation of $h(x)$ over the distribution $f(x)$ is:

$$\mathbb{E}[h(x)] = \int h(x)f(x)\,dx \tag{7}$$

Definition 4.2.2 [17] Using a sample of size n from the distribution $f(x)$ ($\{x_1, \ldots, x_n\} \sim f(x)$), we can approximate the expectation in the previous definition as follows:

$$\mathbb{E}[h(x)] \approx \frac{1}{n}\sum_{i=1}^{n} h(x_i) \tag{8}$$

This is known as a Monte Carlo approximation.

The MC methods are iterative methods that generate samples from a distribution. Some of the MC methods are simple MC methods. These methods draw samples blindly like a

blindfolded person because every step or iteration does not depend on the previous iteration. Therefore, the iterations are independent and are performed blindly in the space of the data distribution. There are two significant sampling techniques in this category: importance sampling and rejection sampling [17].

4.2.1 Importance Sampling

Suppose we have a distribution that may be complicated and isn't easy to sample from. Let $f(X)$ be the probability density function (PDF) of this distribution such that

$$f(X) = \frac{P^*(X)}{Z} \tag{9}$$

where Z is a marginalizing distribution to normalize the distribution (intractable to compute). The $P^*(X)$ term is a scaled PDF of the distribution with the shape of the distribution. Importance sampling is defined as follows. Consider a function of interest, denoted by $h(X)$. We want to calculate the expectation of this function $h(X)$ on the data, over the distribution $f(X)$ or $P^*(X)$. However, since the distribution is hard to compute, we can estimate this expectation using another simple distribution $Q(X)$. This simple distribution, which we can easily draw samples from, can be any distribution such as uniform or Gaussian [17].

4.2.2 Rejection Sampling

If we want to draw samples from some complicated distribution $f(X)$, we can use the idea of *rejection sampling* to sample from a simple distribution $Q(X)$ instead and use those samples to generate samples drawn from $P^*(X)$. The procedure is as follows. In rejection sampling,

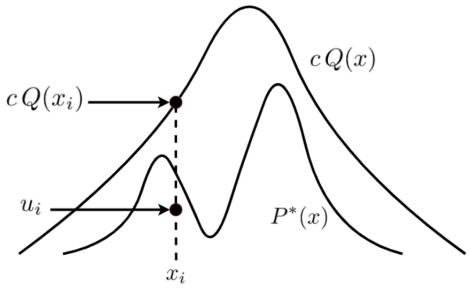


Figure 8: Rejection sampling [17]

we consider a distribution $Q(X)$, where for a positive number c, we have that

$$c \cdot Q(X) \geq P^*(X), \qquad \forall x \in \text{domain}(X) \tag{10}$$

For sampling x_i from the complicated distribution $P^*(X)$, we draw a sample from a simple distribution $Q(X)$ ($x_i \sim Q(X)$). Then, we uniformly sample a random number $u_i \sim U(0, c \cdot Q(x_i))$. Now, if u_i is smaller than $P^*(x_i)$, the sample x_i is accepted. Otherwise, we reject the sample and repeat the procedure. An illustration of rejection sampling can be seen in Figure 8 [17]. The rejection sampling algorithm is described in Algorithm 1.

Algorithm 1: Rejection Sampling

Input: Some distribution $Q(x)$ and n the number of samples
Output: Set of accepted samples
$i = 0$
while $i \neq n$ **do**
  $x^{(i)} \sim Q(X)$
  $u \sim U(0, 1)$
  if $u < \frac{P^*(x^{(i)})}{cQ(x^{(i)})}$ **then**
    accept $x^{(i)}$
    $i = i + 1$
  else
    reject $x^{(i)}$
  end
end

4.3 Markov Chain Monte Carlo Methods

Markov Chain Monte Carlo (MCMC) methods are a set of methods that help in approximating posterior distributions in Bayesian inference. MCMC allows one to characterize a distribution without actually knowing everything about the distribution's mathematical properties. What makes this method so powerful is that it can be used to draw samples from distributions even when all we know is how to evaluate the probability density for samples. MCMC provides a straightforward approach to numerically estimate uncertainties in the parameters of a model using a sequence of random samples. The Monte Carlo part of the algorithm is used to generate random samples from a known distribution and use these samples to simulate sampling from the posterior distribution. The Markov chain part of the method is the idea that random samples are generated in a sequential process. That is, each sample is used as a sort of stepping stone for the next sample due to the Markov property. This forms a chain of randomly sampled points. There are many MCMC methods, but we will focus on one specific method that will be used as a sampling technique for our approach: Metropolis-Hastings.

4.3.1 Metropolis-Hastings Algorithm

The Metropolis-Hastings (MH) algorithm simulates a set of samples from a probability distribution by making use of the full joint density function and the proposal distribution for each random variable in order to simulate sampling from the posterior distribution. The algorithm is described in Algorithm 2. The steps and details are as follows [18]:

Overview: We first initialize the sample value for each random variable (perhaps through a prior distribution). We choose a candidate sample x' from a proposal distribution q. Then, we compute an acceptance probability using some acceptance function $\alpha(x')$ which depends on the proposal distribution and joint density $\pi(\cdot)$. Finally, we either accept the candidate sample with probability α or reject the candidate sample with probability $1 - \alpha$. The algorithm is as follows:

The Proposal Distribution: We start the algorithm by sampling a candidate value x' from the proposal distribution $q(\cdot)$. But, firstly, what are proposal distributions? When we talk about proposal distributions, there are two types we refer to: symmetric and asymmetric. A proposal distribution is symmetric[1] if $q(x^{(i)} \mid x^{(i-1)}) = q(x^{(i-1)} \mid x^{(i)})$. The proposal distribution changes the state of the Markov chain at random, and then either accepts or rejects the state change under some probability. Algorithms that use symmetric proposal distributions in this fashion are called Random Walk Metropolis algorithm [18].

The Acceptance Function: There is a special condition that the Metropolis-Hastings algorithm needs to satisfy: *detailed balance*. This condition simply guarantees that the stationary distribution of the markov chain converges to the target posterior distribution that we are interested in estimating. It is based on this criteria that the acceptance function is chosen. Consider the following acceptance function:

$$\alpha(x^{(i)} \mid x^{(i-1)}) = \min\left\{1, \frac{q(x^{(i-1)} \mid x^{(i)})\pi(x^{(i)})}{q(x^{(i)} \mid x^{(i-1)})\pi(x^{(i-1)})}\right\} \tag{11}$$

[1]For the sake of this paper, we will only focus on symmetric distributions.

where $\pi(\cdot)$ is the full joint density function. If the proposal distribution is symmetric, we have that $q(x^{(i)} \mid x^{(i-1)}) = q(x^{(i-1)} \mid x^{(i)})$. This implies that the acceptance probability simply becomes the ratio of the probability of the current state and the next state under the joint probability density.

Accepting/Rejecting a candidate: The last step in the algorithm is to accept a given candidate with the acceptance probability α described above. The minimum operator ensures that the acceptance probability is never larger than 1. Due to the time-reversibility of the Markov chain and the $q(\cdot)$ term in the acceptance function, we can ensure that the chain doesn't get stuck in one place.

Algorithm 2: Metropolis-Hastings Random Walk

Input: Some distribution $q(x)$ and n the number of samples
Output: Set of accepted samples
Initialize $x^{(0)} \sim q(x)$
for i = *1,2,...,n* **do**
  $x' \sim q(x^{(i)} \mid x^{(i-1)})$
  $\alpha(x^{(i)} \mid x^{(i-1)}) = \min\left\{1, \left(q(x^{(i-1)} \mid x^{(i)})\pi(x^{(i)})\right) / \left(q(x^{(i)} \mid x^{(i-1)})\pi(x^{(i-1)})\right)\right\}$
  $u \sim \mathcal{U}(0,1)$
  if $u < \alpha$ **then**
    $\mathrm{x}^{(i)} \leftarrow x'$
  else
    $\mathrm{x}^{(i)} \leftarrow x^{(i-1)}$
  end
end

5 Approach

5.1 Overview

Generally in the Bayesian paradigm, as we described in the graph setting, the observed graph is viewed as a random variable and we are trying to use Bayesian methods to infer the posterior for the underlying graph [19] [4]. Based on this approach, we want to propose a graph sampling method that can be effective on three fronts. Our approach: utilizes graph structure to determine acceptance probability during sampling, proposes an MCMC random walk-based algorithm that allows for diverse connections, and allows the observed graph to learn connections between weakly linked nodes and potentially missing connections. To accomplish this, we propose a unique MCMC based random walk technique to sample nodes from the graph. Previous methods, specifically [4], utilize a version of node copying for graph inference, shown in Figure 9, which copies the neighbors of a node with the same label as the current node to the current node's neighbors. The problem with this approach is that high node densities can lead to overfitting of the GCN when applied to the graph. Additionally, unlike our approach, their node copying approach uses a fixed hyperparameter ϵ to determine the acceptance probability, which does not use the structure of the graph in any form. The arbitrary nature of this proposal can prove to be problematic when generalizing to larger graph datasets.

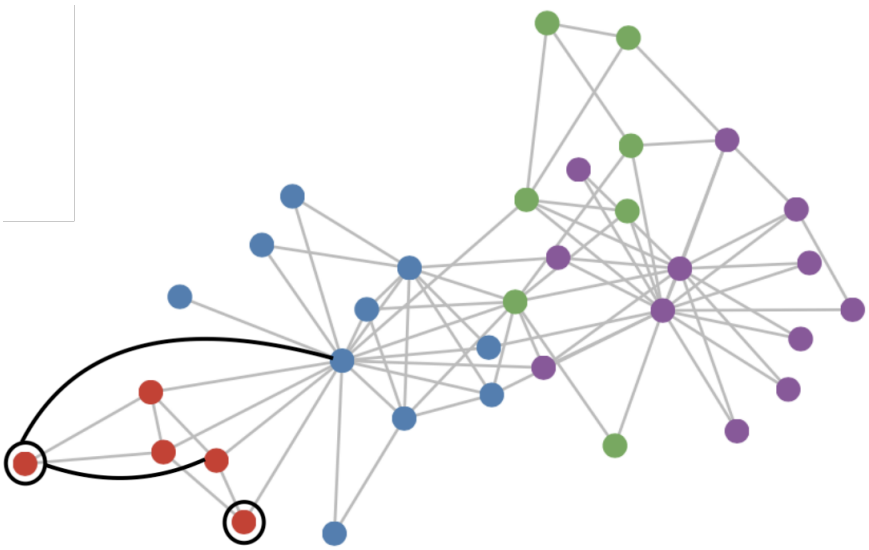


Figure 9: Node Copying Scheme (Graph image from [20])

5.2 Neighborhood Random Walk Sampling

Our approach, the Neighborhood Random Walk Sampling, utilizes MCMC methods to sample nodes from the observed graph and copy neighborhoods of nodes from the random walk. In order to sample graph $\mathcal{G}$ from the proposed model, we utilize the Metropolis-Hastings random walk. Recall the Metropolis-Hastings algorithm from the previous section. We will reformulate this algorithm and apply it to graph sampling.

Suppose we want to generate a random variable V taking values $\{1, 2, \ldots, n\}$, representing nodes in our graph, according to some target distribution $\{\pi_i\}$ where $i \in V$. We have that $\pi_i = \frac{b_i}{C}$ where $b_i > 0$ and the normalizing constant C is difficult to estimate. We simulate a markov chain such that the stationary distribution of the chain will converge to the target distribution of the posterior that we have sought after. Let $\{X_t \mid t = 0, 1, \ldots\}$ be the state-space of the markov chain M using a transition probability matrix Q, which we will assume is the proposal distribution for our method [21]. Now, if $X_t = i$, then we generate a

candidate sample Y such that $P(Y = j) = q_{ij}$ for all $i, j \in V$. Then, we have the following acceptance function:

$$\alpha(j \mid i) = \min\left\{1, \frac{q(j \mid i)\pi_j}{q(i \mid j)\pi_i}\right\} \tag{12}$$

Now, to accomplish graph sampling, we want to treat the nodes in the observed graph $\mathcal{G}$ as states in a Markov chain. We uniformly sample a neighbor of node i from the proposal distribution Q, say j. Since we are going to perform a random walk, the transition probability of moving from state (or node) i to state j is simply $1/d_i$, the degree of the current node i. The transition matrix Q is a stochastic symmetric positive semi-definite matrix and therefore is a symmetric distribution. For a uniform target, we have that $\pi_i = \pi_j$. Further, $q(j \mid i) = 1/d_j$ and $q(i \mid j) = 1/d_i$. So, our new acceptance function for sampling from graphs is

$$\alpha(j \mid i) = \min\left\{1, \frac{d_i}{d_j}\right\} \tag{13}$$

Let $\xi \in \{1, 2, \ldots, n\}^n$ be a random vector where n denotes the total number of nodes and the jth entry ξ^j denotes the node whose edges are to be copied to the jth node in the observed graph $\mathcal{G}_{obs}$ [4]. This random vector is similar to the random vector used for the node copying scheme in Pal et al. Let $X \in \mathbb{R}^{n \times d}$ represent the node features and $Y_L \in \mathbb{R}^n$ represent the training labels. That is, each row of X is the d-dimensional feature vector of the corresponding node. We want to predict one of K class labels $\hat{c}_m \in \{1, 2, \ldots, K\}$ for each node m in the graph. Define the posterior distribution of the random vector ξ as follows:

$$p(\xi \mid \mathcal{G}_{obs}, X, Y_L) = \prod_{j=1}^{n} p(\xi^j \mid \mathcal{G}_{obs}, X, Y_L)$$

$$p(\xi^j = i \mid \mathcal{G}_{obs}, X, Y_L) = \begin{cases} 1/d_i, & \text{if } i \in \mathcal{N}(j) \\ 0, & \text{otherwise} \end{cases} \tag{14}$$

Sampling a node ξ^j is done by selecting a node at random from a random walk generated by a Markov chain and copying the neighbors of that node. As stated before, the nodes in the observed graph can be viewed as states of the Markov chain. A random walk is generated from the current node and a random neighbor of the terminating node is chosen as the candidate to sample. This candidate is accepted based on an acceptance function calculated based on the ratio of the degree of the current node and the degree of the candidate node. The sampling is carried out by simply copying the ξ^j'th node of $\mathcal{G}_{obs}$ in the place of the jth node of G independently for all $1 \leq j \leq n$ with probability given by the acceptance function.

Now, assuming that the events (the states), namely ξ, are independently and identically distributed, our generative model is as follows:

$$p(\mathcal{G} \mid \mathcal{G}_{obs}, \xi) = \prod_{j=1}^{n} \left(\min\left\{1, \frac{d_i}{d_j}\right\} \right)^{\mathbb{1}_{\mathcal{G}_j = \mathcal{G}_{obs,j}}} \left(1 - \min\left\{1, \frac{d_i}{d_j}\right\}\right)^{\mathbb{1}_{\mathcal{G}_j = \mathcal{G}_{obs,\xi^j}}} \tag{15}$$

where $\mathbb{1}_{\mathcal{G}_q = \mathcal{G}_{obs,j}}$ is the indicator function of copying node q in the observed graph $\mathcal{G}_{obs}$ in place of node j of sampled graph $\mathcal{G}$. This model changes the neighbors of the jth node with the neighbors of the node reached by the random walk (shown in Figure 10).

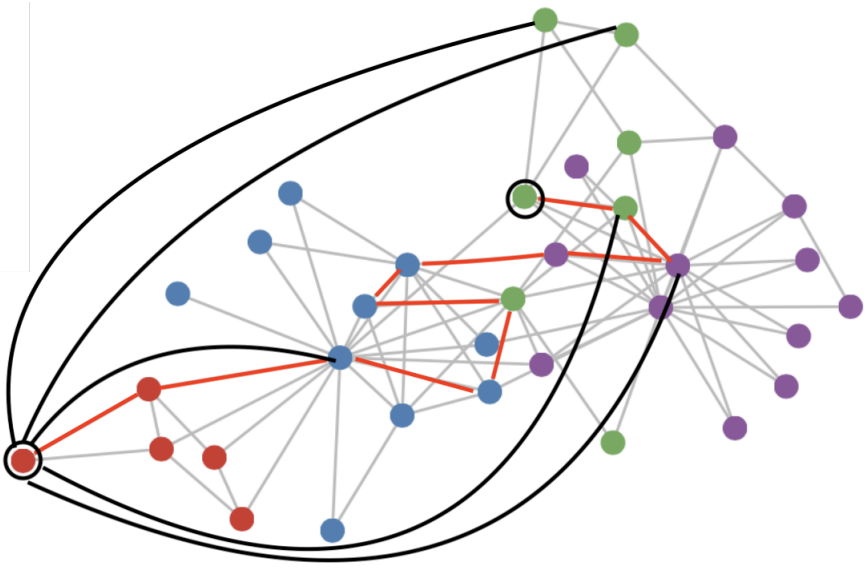


Figure 10: Neighborhood Random Walk Sampling

5.3 Bayesian Convolutional Networks

Given that Z is the final output collected from the last layer of the network (the prediction), we compute the posterior probability of the node labels by marginalizing with respect to the graph and the GCN weights [3].

$$p(Z \mid Y_L, X, \mathcal{G}_{obs}) = \int p(Z \mid W, \mathcal{G}_{obs}, X) p(W \mid Y_L, X, \mathcal{G}) \\ p(\mathcal{G} \mid \mathcal{G}_{obs}, \xi) p(\xi \mid \mathcal{G}_{obs}, Y_L, X) \; dW \; d\mathcal{G} \; d\xi$$

where W is the random weight matrix of the BGCN over the graph $\mathcal{G}$ and ξ is an n-dimensional random vector that represents the neighborhood random walk sampling model. Simply put, this continuous probability consists of the softmax probability, prior distribution on the weights, generative model for graph sampling, and our model for the posterior for the random variable ξ. Similar to Pal et al., our approach models the marginal posterior

distribution of the graph $\mathcal{G}$ as $p(\mathcal{G} \mid \mathcal{G}_{obs}, X, Y_L)$. This allows the features X to play a role in the graph inference process. This posterior is intractable, so Monte Carlo techniques are required to approximate the value.

$$p(Z \mid Y_L, X, \mathcal{G}_{obs}) \approx \frac{1}{V} \sum_{v=1}^{V} \frac{1}{N_G S} \sum_{i=1}^{N_G} \sum_{s=1}^{S} p(Z \mid W_{s,i,v}, \mathcal{G}_{obs}, X)$$

where V samples ξ_v are drawn from the $p(\xi \mid \mathcal{G}_{obs}, Y_L, X)$ distribution. The N_G sampled graphs are sampled from $p(\mathcal{G} \mid \mathcal{G}_{obs}, \xi_v)$ and S weight matrices are sampled from $p(W \mid Y_L, X, \mathcal{G}_{i,v})$ from the Bayesian GCN corresponding to the sampled graph. The complete algorithm is outlined in Algorithm 3.

Algorithm 3: Bayesian GCN using Neighborhood Random Walk Graph Sampling

Input: $\mathcal{G}_{obs}$, X, $Y_{\mathcal{L}}$
Output: $p(Z \mid Y_{\mathcal{L}}, X, \mathcal{G}_{obs})$
Initialization: Execute a pre-training step for a classifier to obtain predicted labels
for $i = 1$ **to** V **do**
  Sample $\xi_v \sim p(\xi \mid \mathcal{G}_{obs}, X, Y_{\mathcal{L}})$
  for $k = 1$ **to** N_G **do**
    for $t = 1$ **to** M **do**
      Sample $j \in \mathcal{N}(i)$
      Uniformly sample $u \sim U(0, 1)$
      if $u \leq \min\{1, d_i/d_j\}$ **then**
        accept node j
      else
        reject j and stay at node i
      end
    end
    Copy neighborhood of accepted node to the neighbors of candidate node
  end
end
Run GCN classifier with new inferred graph

The GCN weights for Bayesian inference can be obtained by performing a variety of techniques such as expectation propagation [22], variational inference [23] [24] [25], and Markov chain Monte Carlo methods [26]. We have found that our neighborhood random walk sampling is quite effective for a multitude of reasons. Firstly, a flaw of the node copying scheme proposed by Pal et al. is that the sampling occurs under a fixed probability parameter ϵ. Instead of fixing a parameter, our model uses the structure of the graph and an acceptance function to perform rejection sampling. Secondly, the node copying scheme copies the neighborhood of nodes that have the same label as the current node, which can lead to overfitting of the model. Nodes can be overly dense, which can cause the model classification accuracy to suffer. Rather, performing a Markov chain-based random walk throughout the graph over a number of iterations (set as a hyperparameter) can be beneficial for diversity in connections. Many connections in a noisy graph can be loose connections. A random walk can help diversify connection to the current node thereby broadening information when aggregated together from neighbors when being propagated through the graph neural network. Lastly, our proposed model works well as the dimension of the dataset becomes larger. That is, Monte Carlo techniques are generally high dimensional techniques and thus have better performance on higher dimensional computations.

6 Experiments and Results

We performed multiple experiments to validate our findings. Specifically, our model increases the node classification accuracy in a graph.

6.1 Datasets

Cora: The Cora dataset consists of 2708 scientific publications classified into one of 7 classes. The citation network consists of 5429 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 1433 unique words [27] [28].

Citeseer: The CiteSeer dataset consists of 3312 scientific publications classified into one of 6 classes. The citation network consists of 4732 links. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from the dictionary. The dictionary consists of 3703 unique words [29] [28].

Pubmed: The Pubmed Diabetes dataset consists of 19717 scientific publications from the PubMed database pertaining to diabetes classified into one of 3 classes. The citation network consists of 44338 links. Each publication in the dataset is described by a TF/IDF weighted word vector from a dictionary that consists of 500 unique words [30] [28].

6.2 Implementation

6.2.1 Running base classifier

The Neighborhood Random Walk Sampling implementation of the BGCN was implemented using PyTorch Geometric with an altered implementation modified to use the Metropolis-Hastings Random Walk algorithm for graph inference[2] as opposed to the node copying scheme from [4]. To implement our model, we used a Graph Convolutional Network as our base classifier. We ran the model for 300 epochs and collected the weights after 240 epochs. The weights in each layer are normalized according to a Gaussian distribution.

6.2.2 Graph Inference

The graph inference step utilized the Metropolis-Hastings Random walk algorithm to traverse through random paths in the graph to get to a node and copy the neighbors of that node to the original node. The sampling was run for 10 iterations to simulate a Markov chain. The graph inference began roughly after 200 epochs. The graph inference samples nodes based on the MCMC scheme and generates an "inferred" graph that is used in place of the original graph when training the model. So, sample graphs are fed into the GCN to compute more robust representations.

6.2.3 Hyperparameters

The hyperparameters used in our experiments are the same as those used for the GCN. We ran the model for 300 epochs with 200 epochs used to pre-train a base GCN classifier.

[2]Modified Implementation: `https://github.com/Akomand/BGCN_TORCH`

The rest of the 100 epochs use our MCMC graph sampling to minimize loss and increase accuracy. Our base GCN is a 2-layer network with input dimension of size the number of nodes, N, in the dataset, a hidden layer of size 16, and an output layer of size equal to the number of classes, K. We set the learning rate α to a standard 0.01. To prevent overfitting of our model, we employed dropout regularization with the keep probability set to 0.5. We set the number of iterations of the Markov chain to $M = 10$.

6.3 Results

We ran a series of experiments with a random split. For the random partition, we randomly sample 5, 10, or 20 labels per class to severely limit the number of classes known for supervision. This is designed to test the limits of the approach under semi-supervision. The random split can give us a more robust measure of the performance of our model. We compare our model's (BGCN-NRWS) performance with that of CheybyNet [2], GCN [5], GAT [6], BGCN [3], and BGCN using Node Copying [4] for the node classification task. The base hyperparameters set for the BGCNs and the GCN are identical. Each of the BGCN models is based on a GCN base classifier. The results of each algorithm are based on an average of 50 trial runs with Xavier random weight initialization. We use the accuracy measure, which is simply the number of correctly labeled nodes over the total number of nodes, as our evaluation metric. All the average accuracies for the three datasets are shown in Figure 11, 12, and 13.

We can see that our BGCN model clearly showcases consistently higher classification accuracies for both larger datasets and across most levels of supervision (5, 10, and 20

Random Split	5 labels	10 labels	20 labels
ChebyNet	61.7 ± 6.8	72.5 ± 3.5	78.8 ± 1.6
GCNN	70.0 ± 3.7	76.0 ± 2.2	79.8 ± 1.8
GAT	70.4 ± 3.7	76.6 ± 2.8	79.9 ± 1.8
BGCN	$\mathbf{74.6 \pm 2.8}$	77.5 ± 2.6	80.2 ± 1.5
BGCN-NC	73.8 ± 2.7	77.6 ± 2.6	80.3 ± 1.6
BGCN-NRWS	72.2 ± 2.8	$\mathbf{80.6 \pm 2.4}$	$\mathbf{80.6 \pm 1.6}$

Figure 11: Classification Accuracy for Cora Dataset

Random Split	5 labels	10 labels	20 labels
ChebyNet	58.5 ± 4.8	65.8 ± 2.8	67.5 ± 1.9
GCNN	58.5 ± 4.7	65.4 ± 2.6	67.8 ± 2.3
GAT	56.7 ± 5.1	64.1 ± 3.3	67.6 ± 2.3
BGCN	63.0 ± 4.8	$\mathbf{69.9 \pm 2.3}$	$\mathbf{71.1 \pm 1.8}$
BGCN-NC	63.9 ± 4.2	68.5 ± 2.3	70.2 ± 2.0
BGCN-NRWS	$\mathbf{65.2 \pm 2.8}$	66.0 ± 2.4	70.0 ± 1.6

Figure 12: Classification Accuracy for Citeseet Dataset

Random Split	5 labels	10 labels	20 labels
ChebyNet	62.7 ± 6.9	68.6 ± 5.0	74.3 ± 3.0
GCNN	69.7 ± 4.5	73.9 ± 3.4	77.5 ± 2.5
GAT	68.0 ± 4.8	72.6 ± 3.6	76.4 ± 3.0
BGCN	70.2 ± 4.5	73.3 ± 3.1	76.0 ± 2.6
BGCN-NC	$\mathbf{71.0 \pm 4.2}$	$\mathbf{74.6 \pm 3.3}$	77.5 ± 2.4
BGCN-NRWS	68.3 ± 2.8	71.5 ± 2.4	$\mathbf{78.0 \pm 1.1}$

Figure 13: Classification Accuracy for Pubmed Dataset

labels). We can see that our model performs better for higher levels of supervision on the Cora dataset, lower levels of supervision on the Citeseer dataset, and higher levels of supervision on the Pubmed dataset. With more hyperparameter tuning, we believe that our model has the potential to outperform across all levels of supervision. For instance, if the number of iterations of the Markov chain is increased, despite time complexity, our model may be able to perform better due to the ability to generate a longer chain. Compared to previous BGCN models, our model utilizes the structure of the graph by generating a Markov

chain random walk to account for diverse connections. This allows us to see consistently close quantitative improvements in addition to the qualitative enhancements.

7 Conclusion

In this thesis, we present a Bayesian Graph Convolutional Network using a neighborhood random walk-based graph sampling method to utilize graph structure, improve diversity among connections, and enhance the accuracy of the model in semi-supervised node classification. We put heavy constraints on the number of labels randomly sampled per training sample so that we can observe how the model performs under such constraints for semi-supervised learning. Future work includes performing statistical significance tests to determine the level of diversity our model provides. Additionally, to optimize our generative model, tuning hyperparameters such as the number of iterations of the Markov chain can help to obtain higher classification accuracies. Similar to Graph Convolutional Networks, there are other graph-based models such as the Graph Attention Network that defines specific weights to give more importance to certain aspects of the network than others. An interesting experiment would be to introduce a Bayesian framework on these attention weights and observe the effect on a variety of tasks including node classification. To improve our model, we could also utilize the Graph Attention Layer as one of the layers in the deep neural network. This could further optimize classification results by having weights for different nodes in a neighborhood to capture more robust representations.

References

[1] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[2] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *Advances in Neural Information Processing Systems*, 2017.

[3] Y. Zhang, S. Pal, M. Coates, and D. Üstebay, "Bayesian graph convolutional neural networks for semi-supervised classification," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2018.

[4] S. Pal, F. Regol, and M. Coates, "Bayesian graph convolutional neural networks using node copying," in *Proc. Learning and Reasoning with Graph-Structured Representations Workshop (ICML)*, 2019.

[5] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *Proceedings of the 5th International Conference on Learning Representations*, 2017.

[6] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," *6th International Conference on Learning Representations*, 2018.

[7] H. Robinson, A. Rasheed, and O. San, "Dissecting deep neural networks," *CoRR*, vol. abs/1910.03879, 2019. [Online]. Available: http://arxiv.org/abs/1910.03879

[8] B. Kanani, "Activation functions in neural network," https://studymachinelearning.com/activation-functions-in-neural-network/, Last accessed on April 19, 2021.

[9] J. Speagle, "A conceptual introduction to markov chain monte carlo methods," *arXiv: Other Statistics*, 2019.

[10] L. V. Jospin, W. Buntine, F. Boussaid, H. Laga, and M. Bennamoun, "Hands-on bayesian neural networks – a tutorial for deep learning users," 2020.

[11] E. Goan and C. Fookes, "Bayesian neural networks: An introduction and survey," *Lecture Notes in Mathematics*, p. 45–87, 2020. [Online]. Available: http://dx.doi.org/10.1007/978-3-030-42553-1_3

[12] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, no. 1, p. 4–24, Jan 2021. [Online]. Available: http://dx.doi.org/10.1109/TNNLS.2020.2978386

[13] J. Norris, "Markov chains," 2011, http://www.statslab.cam.ac.uk/~rrw1/markov/M.pdf, Last accessed on March 13, 2021.

[14] J. Chang, "Markov chains," 2013, http://www.stat.yale.edu/~pollard/Courses/251.spring2013/Handouts/Chang-MarkovChains.pdf, Last accessed on March 15, 2021.

[15] R. D. Peng. (2020) Advanced statistical computing. https://bookdown.org/rdpeng/advstatcomp/, Last accessed on March 3, 2021.

[16] M. Jordan and S. Jain. (2010) Monte carlo sampling. https://people.eecs.berkeley.edu/~jordan/courses/260-spring10/lectures/lecture17.pdf, Last accessed on March 18, 2021.

[17] B. Ghojogh, H. Nekoei, A. Ghojogh, F. Karray, and M. Crowley, "Sampling algorithms, from survey sampling to monte carlo methods: Tutorial and literature review," 2020.

[18] I. Yildirim, "Bayesian inference: Metropolis-hastings sampling," http://www.mit.edu/~ilkery/papers/MetropolisHastingsSampling.pdf, 2012.

[19] Y. Gal and Z. Ghahramani, "Bayesian convolutional neural networks with bernoulli approximate variational inference," in *Proc. Int. Conf. Learning Representations*, 2016.

[20] T. Kipf, "Graph neural networks," https://www.cs.ubc.ca/~lsigal/532S_2018W2/Lecture18a.pdf, Last accessed on March 29, 2021.

[21] M. A. Hasan, N. K. Ahmed, and J. Neville. (2013) Network sampling: Methods and applications. https://www.cs.purdue.edu/homes/neville/courses/NetworkSampling-KDD13-final.pdf, Last accessed on February 26, 2021.

[22] J. M. Hernández-Lobato and R. P. Adams, "Probabilistic backpropagation for scalable learning of bayesian neural networks," in *Proc. Int. Conf. Machine Learning*, 2015.

[23] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *Proc. Int. Conf. Machine Learning*, 2016.

[24] S. Sun, C. Chen, and L. Carin, "Learning Structured Weight Uncertainty in Bayesian Neural Networks," in *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, A. Singh

and J. Zhu, Eds., vol. 54. Fort Lauderdale, FL, USA: PMLR, 20–22 Apr 2017, pp. 1283–1292. [Online]. Available: http://proceedings.mlr.press/v54/sun17b.html

[25] C. Louizos and M. Welling, "Multiplicative normalizing flows for variational bayesian neural networks," 2017.

[26] R. Neal, "Bayesian learning via stochastic dynamics," in *Advances in Neural Information Processing Systems*, S. Hanson, J. Cowan, and C. Giles, Eds., vol. 5. Morgan-Kaufmann, 1993.

[27] J. Motl and O. Schulte, "The CTU prague relational learning repository," *CoRR*, vol. abs/1511.03086, 2015. [Online]. Available: http://arxiv.org/abs/1511.03086

[28] S. L. Group. (2021) https://linqs.soe.ucsc.edu/data, Last accessed on March 30, 2021.

[29] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: http://networkrepository.com

[30] T. Pham, T. Tran, D. Q. Phung, and S. Venkatesh, "Column networks for collective classification," *CoRR*, vol. abs/1609.04508, 2016. [Online]. Available: http://arxiv.org/abs/1609.04508