HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

MASTER THESIS

Ms.
Subhashree Panda

RECURRENT UNIT TOGETHER WITH REINFORCEMENT LEARNING FOR GRAPH NETWORK

2023

Faculty of **Applied Computer and Biosciences**

MASTER THESIS

RECURRENT UNIT TOGETHER WITH REINFORCEMENT LEARNING FOR GRAPH NETWORK

Author:
Subhashree Panda

Study Programme:
Applied Mathematics for Network and Data Science

Seminar Group:
MA18W1-M

First Referee:
Prof. Dr. rer. nat. habil. Thomas Villmann

Second Referee:
Dr. Marika Kaden

Mittweida, February 2023

Acknowledgement

First and foremost, my deepest gratitude goes to Prof. Dr. rer. nat. habil. Thomas Villmann and Dr. Marika Kaden of the University of Applied Sciences Mittweida for their persistent assistance, elaborated suggestions, invaluable counsel, and lastly for believing in my ability to work on the topic.

I would also like to thank my parents and friends for supporting and encouraging me throughout my study years. This accomplishment would not have been possible without them.

Bibliographic Information

Panda, Subhashree: RECURRENT UNIT TOGETHER WITH REINFORCEMENT LEARNING FOR GRAPH NETWORK, 47 pages, 40 figures, Hochschule Mittweida, University of Applied Sciences, Faculty of Applied Computer and Biosciences

Master Thesis, 2023

Abstract

Recently a deep neural network architecture designed to work on graph-structured data have been capturing notice as well as getting implemented in various domains and application. However, learning representation (feature embedding) from graphical data picking pace in research and constructing graph(s) from dataset remains a challenge. The ability to map the data to lower dimensions further makes the task easier while providing comfort in applying many operations. Graph neural network (GNN) is one of the novel neural network models that is catching attention as it is outperforming in various applications like recommender systems, social networks, chemical synthesis, and many more. This thesis discusses a unique approach for a fundamental task on graphs; node classification. The feature embedding for a node is aggregated by applying a Recurrent neural network (RNN), then a GNN model is trained to classify a node with the help of aggregated features and Q learning supports in optimizing the shape of neural networks. This thesis starts with the working principles of the Feedforward neural network, recurrent units like simple RNN, Long short-term memory (LSTM), and Gated recurrent unit (GRU), followed by concepts of Reinforcement learning (RL) and the Q learning algorithm. An overview of the fundamentals of graphs, followed by the GNN architecture and workflow, is discussed subsequently. Some basic GNN models are discussed in brief later before it approaches the technical implementation details, the output of the model, and a comparison with a few other models such as GraphSage and Graph attention network (GAN).

I. Contents

II. List of Figures

III. List of Tables

Abbreviations

ML	Machine learning
AI	Artificial Intelligence
DL	Deep learning
RL	Reinforcement learning
GNN	Graph neural network
DNN	Deep neural network
MLP	Multi-layer perceptron
NMP	Neural message passing
DiGraph	Directed graph
RNNs	Recurrent neural networks
GRU	Gated recurrent units
LSTM	Long short-term memory
GCN	Graph convolutional network
GAN	Graph attention network
MDP	Markov decision process
SL	Supervised learning
UL	Unsupervised learning
SSL	Semi-supervised learning
FFNN	Feedforward neural network
ReLU	Rectified linear unit
SGDL	Stochastic gradient descent learning
ANN	Artificial neural network
TD	Temporal-difference
RNN-GCN-RL	Suggested model
Tx	Tasks
Ex	Experience
P	Performance measure

Mathematical Symbols

$G = G(V,E)$	A graph
V	A finite nonempty set of vertex or node
E	A finite nonempty set of edge
$\|V\| = n$	Number of vertices in a graph
$\|E\|$	Number of edges in a graph
e_{ij}	Edge connecting vertices i and j
$W(u,v)$	Walk in a graph
w	Weight of nodes or edges in a graph
$deg(v)$	Degree of a node v in a graph
$N(v)$	Set of neighbor nodes v in a graph
ϕ	One to one mapping
$\mathbb{R}$	Real number
$A = [a_{ij}]$	Adjacency matrix
$B = [b_{ij}]$	Incidence matrix
$a_{ij}^{\{k\}}$	Number of walks of length k from vertex i to vertex j
${h_v}^{\{k\}}$	Node or hidden or feature embedding of a node
${agg_v}^{\{k\}}$	Aggregated feature vector of neighbor nodes
X	Set of feature vector of a nodes
l	Set of length of node features
x_v	Feature vector of node v
x	input vector
k	Time step or iteration number
K	Number of layers in GNN
$S(x)$	Binary step function
θ	Slope parameter
$\sigma = sgd_\theta(x)$	Sigmoid activation function
$e = exp$	Exponent
$\hat{y}_k$	Resulted output
y_k	Expected output
net_i	Activation
l_k	Local loss
L	Loss function
L_G	Loss function in graph

Table 1: Mathematical symbols & notations

Mathematical symbols & notations contd.

$Q(s,a)$	Q value for state s and action a
η, ψ	Learning rate and step size
α_{ij}	Attention coefficient
γ	Discount factor
π_t	Policy at time step t
λ	Eigenvalue
D	Diagonal matrix
I	Identity matrix
Z	Output embedding of a layer
h	Hidden embedding or output of a hidden layer
H	Number of attention head in multiattention GAT
H_a	Attention head layer number
T	Matrix transpose operation
$\|\|.\|\|$	Dissimilarity measure
$\|\|$	Concatenation
$[,]$	Closed set, inclusive of boundary values
$(,)$	open set, exclusive of boundary values

1 Introduction

Current developments highlight the success of deep neural networks in numerous domains, showing superior performance in image processing [14], natural language processing [15], speech recognition [16], etc which motivated research further. The power of deep learning to exploit the hidden relationship in data to extract the underlying pattern is well recognized. However, with the increasing complexity of data, the representation ability of a graph provides the benefit of complete data structure representation and the relationship of a node with the graph structure [12].

The Graph neural network (GNN) model is a deep neural network framework primarily designed to handle graph structure data. The key idea here is to learn the representation vector of a node, considering the graph structure along with the feature information available. The hidden embedding of a node is learned by aggregating the feature of the neighbor nodes and merging with the node feature, leading to a simpler representation of the graph. [12]

Graph-structured data are omnipresent and have a wide range of uses. The task in the graph domain can be categorized based on elementary levels: node-level embedding, edge-level embedding, and graph-level embedding [17]. This thesis considers the node classification problem in a graph, where all the nodes are available for training and labels are available for only a small subset of nodes. For making effective predictions, a crucial problem is to have an efficient node representation. Another part of this thesis implementation also tries to test the model with a new graph that is not exposed to the model during training.

A method proposed for node classification by Kipf et al. [18] is a graph convolution model which scales the number of edges linearly and learns the hidden embedding that represents local graph and node features. However, this model works only for the transductive learning task. This model is further studied by Esmaeili et al. [19] with a modification to add a piece of noisy side information extracted from the adjacency matrix or feature matrix to the available data labels for training the model.

An inductive learning model using GraphSage for unseen data is proposed by William et al. [20] for node classification. This method samples and aggregates neighbor nodes and features to generate node embedding. This facilitates different complex aggregation functions.

A study on graph attention networks proposed by Veličković et al. [21] is for node classification by performing self-attention on the nodes. Here, each neighbor node is assigned a different weight, attention layers are stacked and node embedding is calculated, resulting in a promising outcome in both transductive and inductive learning tasks.

Numerous models and further enhanced studies have been proposed for smoothening the turnaround time and enhancing the efficacy of data learning over various data domains. Recurrent neural networks (RNNs) like Gated recurrent units (GRU) and Long short-term memory (LSTM) can act as an efficient aggregator as well as predict the output. Initially, Gori et al. [22] suggested the use of a recursive neural network on various types of graphs, [23] studied further using RNN with a gated mechanism. Weiping et al. [24] and Wenhan et al. [25] modeled the learning environment using the Markov decision process (MDP) and presented the efficacy of the model by integrating the LSTM network and Reinforcement learning(RL) reward technique for an agent for prediction.

This thesis focuses on the problem of learning node embedding for the classification of nodes and implementing the GNN model for graph data. A graph G(V, E), V is the set of labeled nodes (data point with feature vector), and E is the set of pairs of vertices representing edges (relationship between data points.). a_{ij} is a $N \times N$ adjacent matrix of the graph G where N is the number of vertices. Node attribute matrix X is a $N \times C$ matrix with C number of features representing each node. The proposed neural model aims to learn embedding ${h_v}^{\{k\}}$ for vertex v, considering the k layer neighborhood. This model utilizes the node attribute and graph structure information to learn effectively and predict classification results with minimum loss. [12]

To implement the above-stated goal, this thesis deeply focuses on the underlying concepts, and the techniques followed in [24] and [25] and enhances it further. To begin with the GNN model, this thesis attempts to arm the readers with the required concepts and contexts to understand the following sections. In particular, graph theory is essential to represent and compute data in a graphical structure. However, it will not delve too deeply. Prior to that, this thesis briefs on the fundamental concepts of deep learning, and the subsequent subsection provides an outlook on various recurrent neural networks and respective functionalities followed by the basic knowledge of reinforcement learning and the selected algorithm for implementation. Up next, it will discuss the general framework of GNN and its phases, and later it explores further some of the benchmarked models like Graph convolutional network (GCN), GraphSage, and Graph attention networks (GAN).

Next, this thesis will elaborate on the proposed model implementation in detail, environment setup, and opted dataset for trial. Further explains a few experiments on several datasets to assess the efficacy and turnaround time of the proposed GNN model setup. It demonstrates two different approaches to model training and testing. In one method, the model is trained by complete exposure to the graph dataset features and tested for the hidden nodes. In the other one, two subgraphs are induced from the graph for training and testing. The results of both the techniques applied over all the desired datasets are plotted and discussed in detail before concluding with the outcomes and possible future works to extend.

2 Fundamentals of Deep Learning and Neural Networks

Machine learning (ML) is a field of Artificial intelligence (AI), that indulge in learning the process of designing computer algorithms that can mimic human intelligence. The formal definition of ML is coined as "a field of study that gives computers the ability to learn without being explicitly programmed". More explicitly, "A computer program is said to learn from experience (Ex) with respect to some class of tasks (Tx) and performance measure (P), if its performance at tasks in Tx, as measured by P, improves with experience Ex". [1]

The critical part of the process of enabling the machines to learn is to make the machine equipped with the right set of features. However, it can be tricky and complicated to identify the ideal set of features manually and comprehend which traits should be extracted when taking into account all potential scenarios. The challenges encountered by hard-coded knowledge-based systems indicate that AI systems must be able to learn on their own by identifying patterns in unstructured data. Hence, using the machine learning algorithm to learn feature representation along with delivering output from the representation will yield lesser human effort and training time. However, abstracting highly precise features from raw data with negligible variance turns out to be extremely difficult. Here, Deep learning (DL) facilitates the flexibility to represent the complex feature in terms of simpler representations by composing a Deep neural network (DNN). DNN has a complex architecture of strongly interconnected neurons with at least one hidden layer. Further, most of the ML or DL algorithms can be categorized into supervised, unsupervised, and semi-supervised learning according to the nature of the labeling of data [10]. [26]

2.1 Deep Learning Approaches

Supervised learning (SL) is the process of training a model to predict the mapping of unknown samples from a set of given pair <x,y> of input x and respective expected label y for output. Based on the value types of data labels, SL can be divided into classification and regression. When the target label is a discrete value type, classification is performed. The regression method is applied for the continuous value of desired output. Given the dataset contains only the input values without the expected labels, the learning system follows the Unsupervised learning (UL) technique. As a result, the objective of UL is to infer structures and patterns while looking for similarities in data. Clustering and representation of data in higher dimensional space are a few examples of UL procedure. In Semi-supervised learning (SSL), the algorithm learns with the dataset which is partially labeled. A portion of the labeled data is used to infer the learning (hidden

structure) of the unlabeled part of the data. [10] [1]

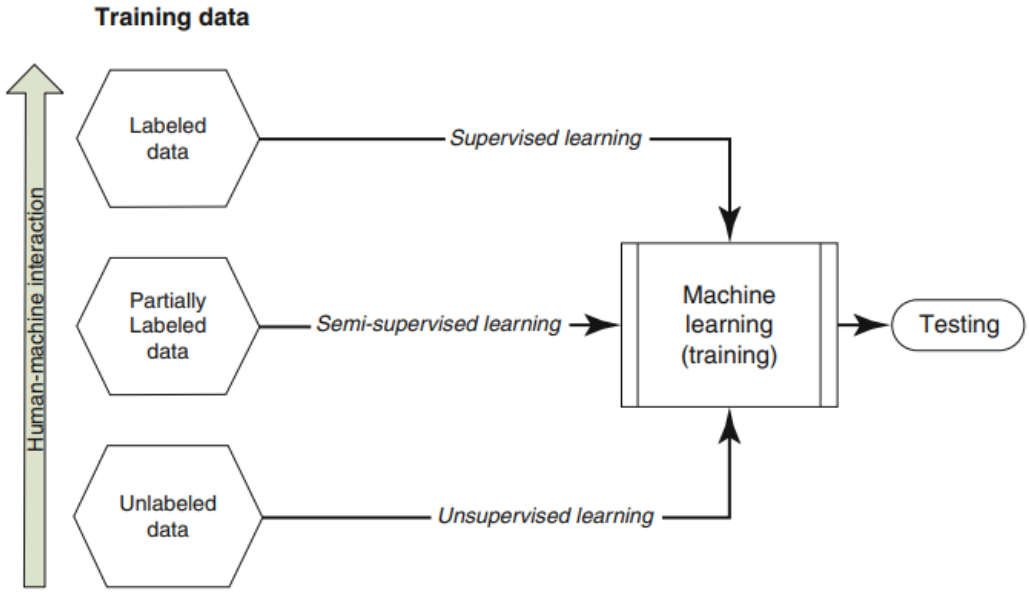


Figure 2.1: Deep learning algorithm types based on depending on the training data [1]

Another significant approach of DL is Reinforcement learning (RL). RL enables the learning system to make a sequence of decisions by a penalty-reward technique based on the actions chosen with a goal to attain maximum rewards and least penalties [10]. This training mechanism differs from SL as to infer the output for unaware scenarios, a machine needs to learn from experience. It also can not be referred to as UL, as it tries to obtain maximum rewards by selecting a sequence of appropriate decisions instead of learning the hidden mapping of data. In order to learn the process of effective decision-making for optimal reward, the agent needs to exploit the best rewarding options from experience and/or by exploring an appropriate number of available options. [7]

2.2 Feedforward Neural Network: A neuron model

A neuron is the simplest and linear neural network model, also referred to as an information processing unit. This model f(x,w), provided with a set of inputs x and learning parameter weight as w, can identify linearly separable inputs into a positive or negative outcome. The shortcoming of this model (non-linearly separable data) leads to Multilayer perceptron (MLP) or Feedforward neural network (FFNN), one of the most effective approaches. MLP, a Deep neural network (DNN), consists of several neurons interconnected through at least one hidden layer apart from the input and output layers. The information propagates through each layer forward, processed together with weights and biases assigned per layer and an activation function applied on it. The accuracy of a model is attained by comparing the result with the expected output and optimizing the

trainable parameters accordingly. The below image illustrates data processing from the input layer to the output layer in a feedforward neural network. [26]

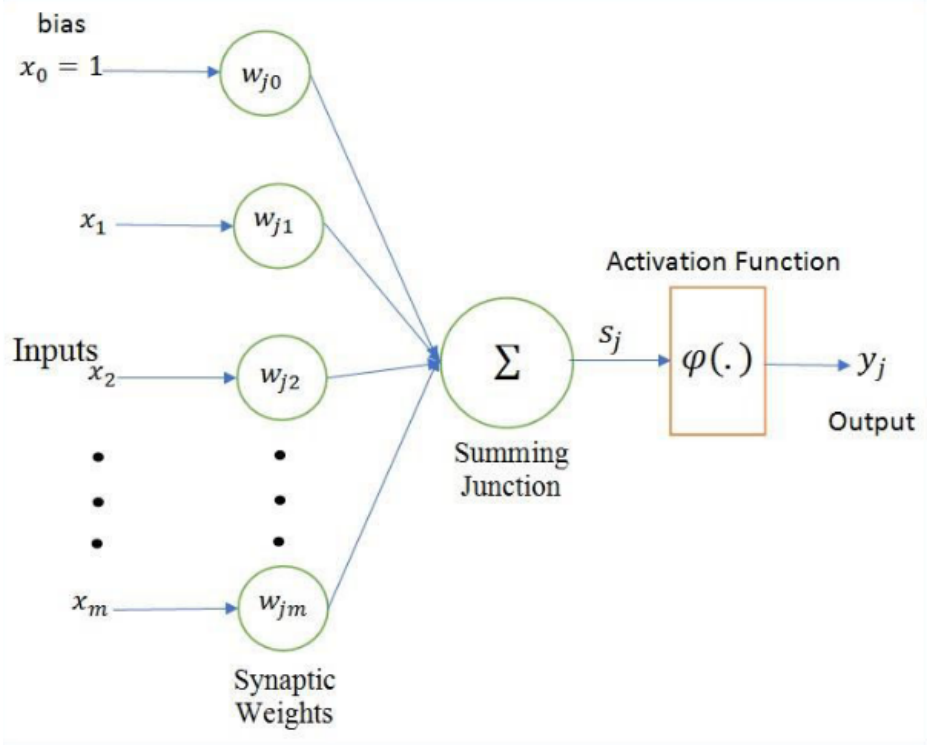


Figure 2.2: Single unit representation of information processing in MLP [2]

2.2.1 Activation Functions

An activation function is an element-wise applied differentiable operation that is required to calculate hidden layer values, helping MLP to excel in performing complex tasks. A few of the commonly used activation functions are discussed below [26].

Binary step function: A threshold-based non-differentiable binary classifiable step function that activates the neuron (output = 1) if the input value exceeds the threshold, otherwise deactivates (output = 0). [26]

$$S(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Sigmoid function: The sigmoid activation function is a continuously differentiable and non-linearly separable function that squeezes the output values in the range $[0,1]$.

$$sgd_\theta(x) = \frac{1}{1+e^{-\theta x}}$$

where $\theta > 0$ controls the sharpness of the slope. The derivative of the function is

$$sgd_\theta{}'(x) = sgd_\theta(x)(1 - sgd_\theta(x))$$

when θ tends to larger values (∞), MLP suffers from vanishing gradient problem. [26]

Rectified Linear Unit (ReLU): One of the most popular non-linear squashing functions, denoted as ReLU generates the maximum between 0 and input x as output, max(0,x). [26]

$$ReLU(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise} \end{cases}$$

SoftMax function: The often used final layer activation function predicts the categorical probability distribution of the set of input provided and the output being discrete probability values range in $(0,1)$. [26]

$$SoftMax(x_i) = \frac{e^{(x_i)}}{\sum_{j=1}^{n} e^{(x_j)}}$$

2.2.2 Gradient Descent Optimization Method:

In order to obtain the highest level of accuracy and minimum loss in a machine learning task, one of the most commonly used optimizers is **gradient descent** method, which tries to minimize the error function by iteratively updating the learning parameters based on gradient computed through partial derivatives. In each iteration, the parameters (weight and bias) are updated and the step size reduces tending to a stagnant state of the cost function, which indicates saturation of learning. **Stochastic gradient descent learning** (SGDL) is one of the variants of gradient descent optimization techniques. The necessity of a gradient-based optimization function in a non-linear DNN model is due to its non-convex cost function. A convex optimization algorithm assures global convergence, whereas a non-convex function is highly dependent on parameter initialization. [26]

2.2.3 Working of Feedforward Neural Network

By incorporating all the concepts mentioned above, the functionality of MLP is elaborated with an illustration using figure 2.2. As the name feedforward conveys, the input is processed forward to produce an output, and the method is called **forward-pass**, whereas training the learning parameters to attain an optimal desired outcome by propagating the error backward from the output layer to the input layer is known as **backward-**

pass, alternatively **back-propagation** [26]. The significant variables of an MLP are a set of inputs x $(x_1, x_2, ..., x_n)$, weight matrix w for each layer, bias b, activation function f per neuron in a layer, an optimizer δ and learning rate η. [26]

Steps: [3] [26]
1. Initialize all the weights w with random small numbers, preferably in the range (-1,1).
2. Process a sample from training data and obtain a result $\hat{y}_k$.
3. Compare the result with the expected outcome y_k.

$$net_1 = w_1 x + b_1, \quad net_2 = w_2 o_1 + b_2,$$
$$o_1 = f(w_1 x + b_1), \quad o_2 = f(net_2)$$

net_1 is the weighted sum or the activation in the first hidden layer, and f is the activation function applied to activation. o_1 is the output from the 1st layer of neurons, provided x as input. net_1 is the input for the next hidden layer and continues further. Hence, aggregating all the terms together:

$$net_i = \sum_{j->i} w_{ji} o_j + b_i$$

net_i is the excitation of neurons for layer i, w_{ji} is the weight matrix applied on the connection from layer j to i, and b_i is the bias for i^{th} layer, and $o_j = x$ is the data presented to the input layer.

$$\hat{y}_k = f(net_i)$$

$\hat{y}_k$ is the output from the final layer, where k is the number of neurons in the output layer after i^{th} layer.

$$l_k = (y_k - \hat{y}_k)$$

local error l_k is the difference between the expected (y_k) and predicted result $(\hat{y}_k)$.

$$L = \frac{1}{2} \sum_k l_k^2 = \frac{1}{2} \sum_k (y_k - \hat{y}_k)^2$$

by using mean-square error here, the total loss L of the model is calculated.

4. Using the gradient descent optimization technique, propagate the error backward. Calculate the gradient, update the final layer associated weight $w_j o$ and bias accordingly, then navigate to the previous layer and continue the same operation until the initial layer. The process continues iteratively to attain minimum loss and maximum expected outcome. Calculating gradient:

$$\Delta w_{ji} = -\eta \frac{\partial L}{\partial w_{ji}},$$
$$w'_{ji} = w_{ji} + \Delta w_{ji}$$

$$b_i' = b_i + \Delta b_i$$

$$\Delta w_{jo} = -\eta(y_k - \hat{y}_k)(1 - \hat{y}_k)\hat{y}_k o_h$$

$\Delta w_{ji} = \Delta w_{jo}$ is the weight update in the output layer, calculated by taking the gradient, which is derivative of error L with respect to weight w linked from the last hidden layer to final layer and updating the weight and bias accordingly. Bias can also be inserted to input matrix x as x_0 as figure 2.2 demonstrates, avoiding the update effort during back-propagation. Δb_i is the change in bias and b_i' is the updated bias value. [26]

Learning rate η helps in controlling the updated value or the step size, avoiding the gradient turning out to be 0. The learning rate value ranges in (0,1), a very small positive value. The state of gradient value close to zero and the change in weight and bias is negligible is called **vanishing gradient**. In DNN, while updating the parameters in back-propagation, earlier layers toward output get updated successfully. However, due to the dense network containing a large number of hidden layers, the gradient tends to zero after parameter updates for the last few layers, leading to stagnant values in the initial layers. Hence, the model stops learning after reaching saturation for activation function on extreme values. Artificial neural network (ANN) treats each data fed to a neuron individually, so the connection or order of the data is lost. Sequential data processing is one of the major drawbacks of an Artificial neural network (ANN), as the neurons in a DNN do not contain a memory to hold and flexibility to forget irrelevant data. To overcome the deficit of ANN, RNN and its various types stands out with the advantages. [26]

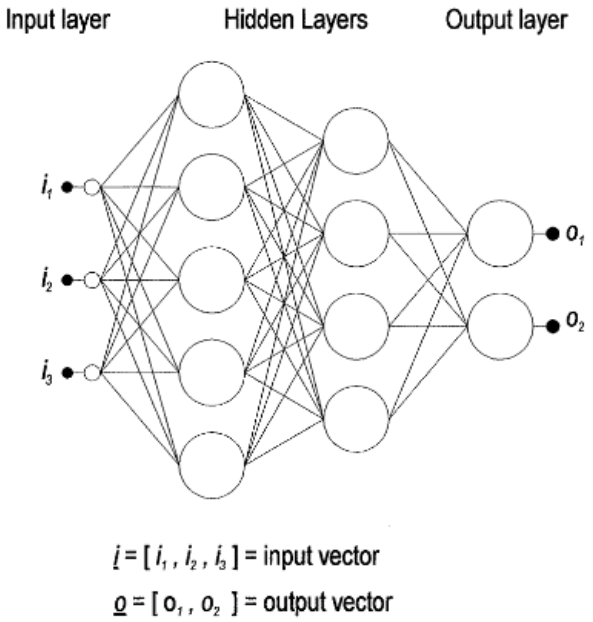


Figure 2.3: Example of a dense MLP with 2 hidden layer [3]

2.3 Recursive Neural Networks: Sequence modeling

Recurrent neural networks (RNNs) are a class of neural networks that are particularly well-suited to handling sequential values as input. As the name recursive or recurrent suggests, repetition of a step multiple times results in sharing parameters across the DNN. The output at a time step refers to the output produced in the previous steps and is generated using the same update rule as applied to the previous. A general structure of an RNN and its comparison with respect to an FFNN is portrayed in the below figure. [26]

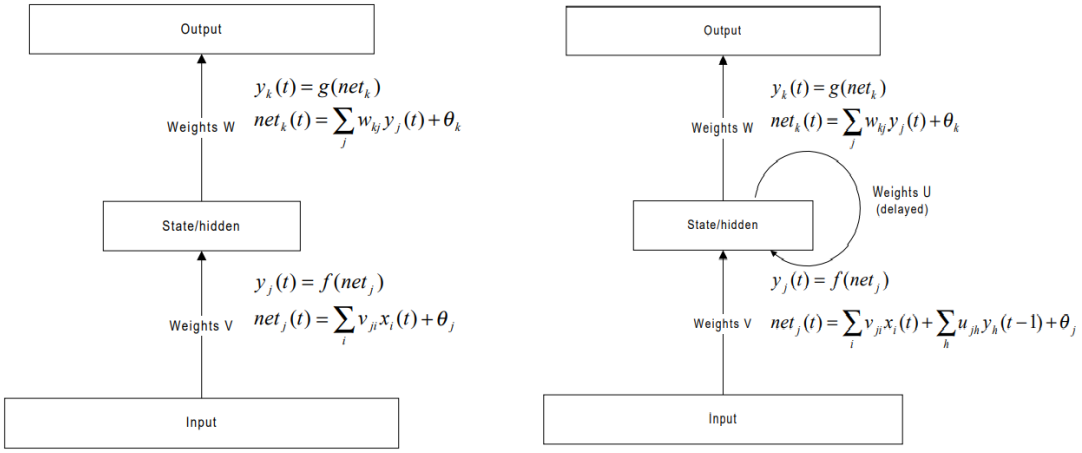


Figure 2.4: Workflow of a FFNN and RNN [4]

The left part of the image is the basic architecture of an FFNN, and the right side of the figure is the RNN. The activation $net_j(t)$ at time step t is the total sum of weighted (v_{ji}) inputs (x_i), and bias θ_j for the j^{th} layer neuron contributed from all the neurons from layer i. An activation function (f or g) is applied to the activation to generate an output of a layer. However, in the case of RNN, the output at time step t depends on the weighted input at time step $t-1$, and the result of $t-1$ step relies on $t-2$ step, and so on. This dependency is attained by a self-loop in the hidden state, which says the output of the previous time step is fed as input to the hidden state for calculating result for the next time step. The outcome of the final layer remains the same for both neural networks as the change remains in the hidden layers. [26]

The result obtained at a time step t, acquired from partially observing the data that can influence the output at a much later time step $\mathbb{T}$+t. As the network iteratively impacts the future outcome, it can process any length of data, even the much larger ones. The historical data of the sequence up to the present time step can be reserved using the hidden variables, indicating that the neural network has the potential to memorize data. [6]

The hidden state, which is the key component of RNN, is further explored in detail. The

hidden variable at time step t is determined by the current time step input x(t) and the hidden variable of the previous time step. The implicit layer expression is given by [6]

$$o_t = y_k(t) = g(Wh_t + \theta_k)$$

$$h_t = y_j(t) = f(VX(t) + Uy_{t-1} + \theta_j)$$

Expanding the hidden variable h_t and the output variable o_t, [6]

$$h_t = f(VX(t) + Uf(VX(t-1) + Uf(VX(t-2) + Uy_{t-3} + \theta_j) + \theta_j) + \theta_j)....$$

$$o_t = g(Wf(VX(t) + Uy_{t-1} + \theta_j) + \theta_k)....$$

A pictorial representation of unrolling the hidden state up to three layers is conveyed in the below figure.

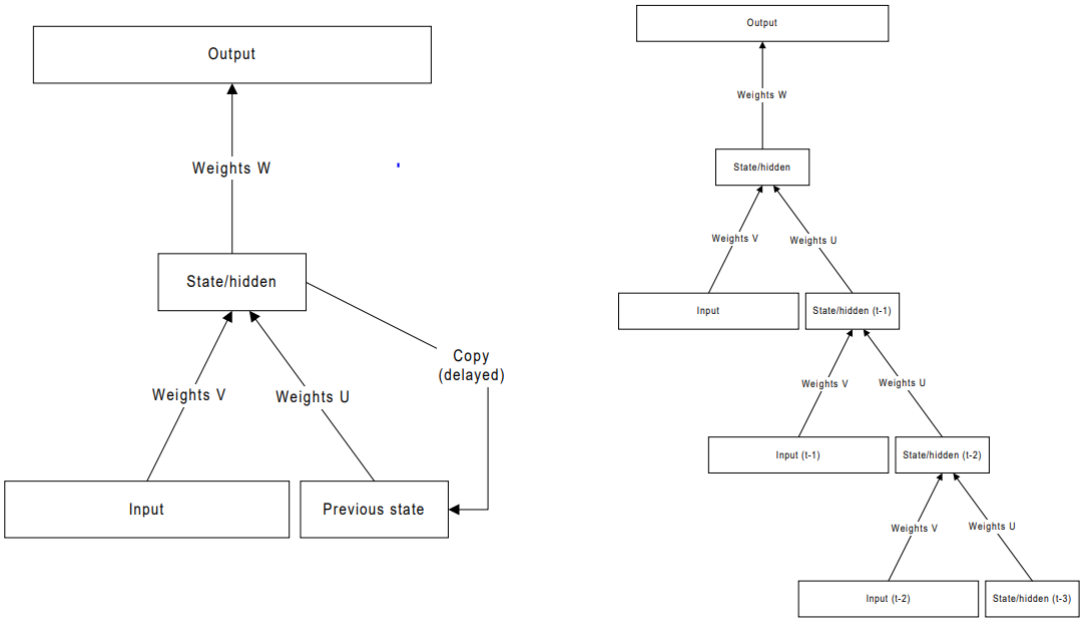


Figure 2.5: Unrolling the self-loop at hidden state in RNN [4]

As the above two images convey, the weight assigned for all the hidden layers are same (U, here). The update in the parameters (weight and bias) during back-propagation is a big step for each value. Therefore, the gradient calculated using the loss either diminishes very fast or grows large leading to a vanishing gradient or gradient explosion. Hence, a dense hidden layer is not contributing to model training. These problems can be overcome by replacing the hidden unit with a gated memory block. Larger gradients values are typically stored in memory to avoid gradient disappearance and setting a threshold value can prevent gradient explosion. [6]

2.4 Long Short-Term Memory: A gated RNN

The issue of vanishing and exploding gradients prevents standard RNNs from learning long-term dependencies. Hence, an LSTM unit was developed by replacing the hidden unit with a recurrent network cell. LSTM network structure consists of three gates (logical units): **input gate, output gate, and forget gate** along with a **memory cell**. These gates adjust the weights at the edges of the other neural network components connected to the memory unit to improve the error function of the selective memory feedback with the gradient. However, they do not take part in sending their output to other neurons. An image of the architecture of an LSTM cell unit is given below. [6] [5]

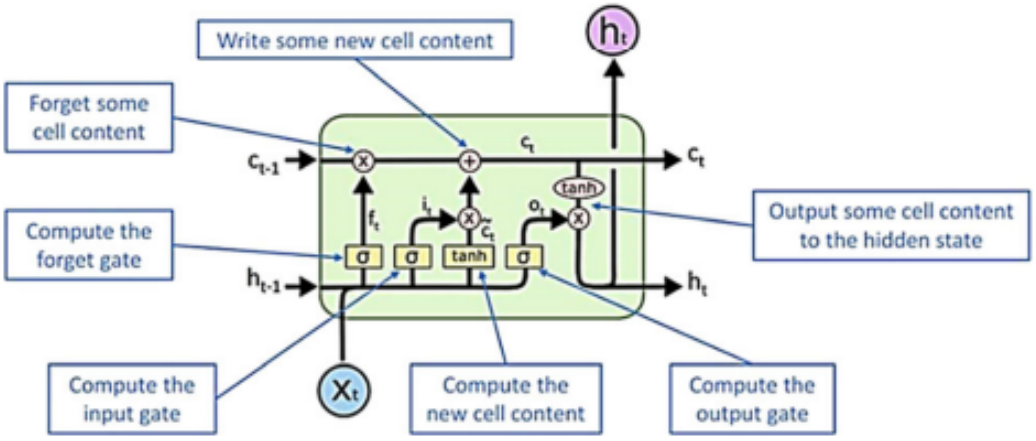


Figure 2.6: Exploration of LSTM memory cell replacing hidden state [5]

Each gate is regulated by four factors: current input, previous cell state, sigmoid activation function σ, and a multiplication operator, as in the figure above, resulting in output between (0,1). An input unit is processed with a regular artificial neuron unit and can also have a nonlinear squashing function for the same. The sigmoid layer output of **input gate** decides the need for input data to be added to the cell state, and the tanh function processes previous cell state contributing to the current cell state. The weight for the self-loop of the cell state is decided by the **forget gate** sigmoid value, deciding whether to retain or discard the previous cell state information. The updated cell state information is stored in the **memory unit**. The **output gate** determines which cell information to be carried as output for the current time step. [26] [6]

The vanishing and exploding gradient problem is handled by replacing the concatenated multiplication of cell state and input value with concatenated addition. The three gates and memory cell state f_t of LSTM can be calculated by [27]

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.1}$$

$$\begin{cases} i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \\ \tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c) \end{cases} \tag{2.2}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.3}$$

$$\begin{cases} o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \\ h_t = o_t * \tanh(C_t) \end{cases} \tag{2.4}$$

$W_f, W_i, W_o, W_c, b_f, b_i, b_o, b_c$ are the weight matrix and bias for forget, input, output gates, and cell state. $[h_{t-1}, x_t]$ denotes summation of input data and previous cell state. C_t, C_{t-1}, $\tilde{C}_t$ are new cell state value, previous cell information, and new memory content respectively. i_t, o_t are information collected from the input gate and output gate, whereas h_t is the hidden layer output for the current time step. [27]

2.5 Gated Recurrent Units

Gated Recurrent Units (GRU) is a modified version of RNN, extending the feature of LSTM by removing the cell state and replacing the input and forget gate with an update gate with combined features for an improved efficiency up to a large time step [27]. GRU contains three gates: update gate, reset gate, and learning gate [6]. A pictorial representation of a GRU with explicit details on gates and operations is given below.

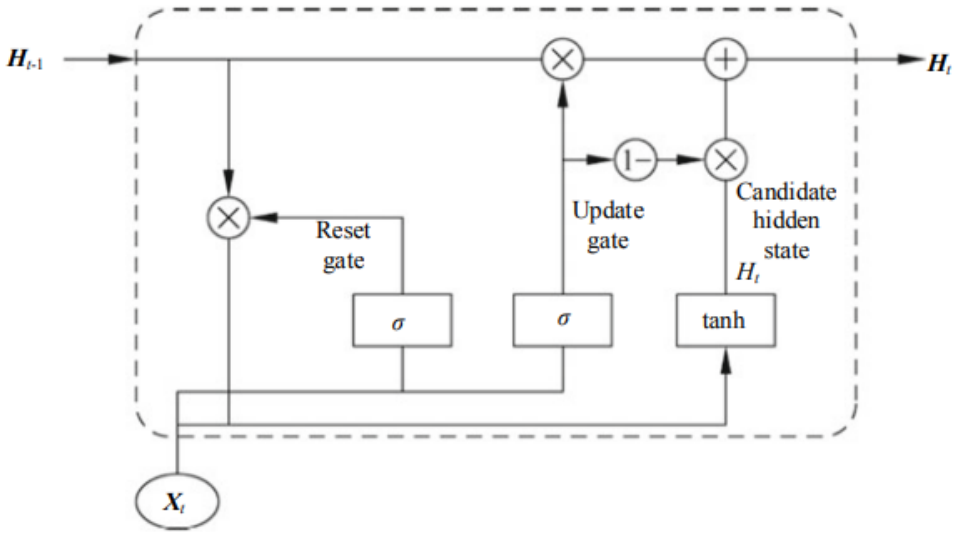


Figure 2.7: Exploration of GRU memory cell [6]

The amount to which the past and current information is kept and ignored depends on the update gate given by z_t [27]

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \tag{2.5}$$

The reset gate r_t determines which of the previous data to be carried forward and which

is to be discarded, given by expression [27]

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \tag{2.6}$$

The hidden state output h_t depends on the current reset gate, and input gate result along with previously hidden state information (h_{t-1}) and input data [27]

$$\begin{cases} \tilde{h_t} = \tanh(W_h \cdot [r_t * h_{t-1}, x_t]) \\ h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t} \end{cases} \tag{2.7}$$

w_z, w_r, w_h, b_z, b_r are the weight matrix and bias for the update, reset gate, and weight for the hidden state. σ and $\tanh$ are sigmoid and tanh activation functions. [27]

2.6 Reinforcement Learning:

Reinforcement learning (RL) is a goal-directed learning mechanism with an interactive agent to learn while interacting with an uncertain environment by sequential actions, figuring out the best action to take on a step considering the consecutive steps ahead to optimize the numeric reward value, with an intention to attain minimum loss in terms of penalty [7]. The reward retained here is a cumulative value of outcomes during an infinite number of interactions. This interactive process of state-action-state is a state transition methodology, also known as a Markov decision process (MDP). [28]

The major elements of Reinforcement learning are an agent and an environment, action/s, state, and reward and policy, reward system, value function, and/or a model. An **agent** is the learner and the decision maker. The **Environment** E is everything that surrounds the agent and with which it interacts, which defines the task. An agent at time step t and state (situation) S_t, performs an action A_t, and the environment reacts by presenting the agent with a new state S_{t+1} and a numerical value as reward/penalty R_{t+1}. This workflow summary of an RL problem is given below. [7]

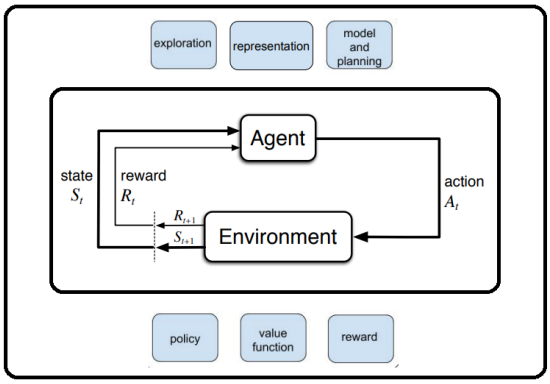


Figure 2.8: Overview of an RL problem in terms of MDP [7] [8]

The mapping implemented by the agent from states to probabilities $\pi_t(a|s)$ of choosing each potential action at each time step is called **Policy** π_t. Policy changes are based on the agent's learning from experiences. A **reward** system helps the agent in identifying good and bad events by providing a positive or negative number corresponding to the selected action at a state. The reward system can act as a stochastic function of the state, and the actions opted for. The reward for a task is a collection of all the step-wise rewards attained throughout the state transition from start to goal. Hence, accumulating long-term rewards is more important than collecting immediate larger rewards. A **value** function indicated the usefulness of a state after accounting for the states that are anticipated to follow and the benefits offered by those states. An RL system can have a defined model to plan and learn simultaneously by inferring the environment's reaction in the future based on the present state and actions. Otherwise, it learns from trial and error as a model-free approach. In an RL task, if the agent-environment interaction ends naturally due to an identified end state, then the task is known as episodic. Else if it continues learning infinitely, then it is a continuous task. [7]

An RL algorithm aims at attaining the policy which accumulates maximum rewards (value function). To attain maximum approximation, certain algorithms like the Monte-Carlo method try to leverage their experiences and navigate to the future by exploiting already-known information to obtain rewards. However, there are certain other algorithms like dynamic programming, that estimate future rewards by exploring new options. Temporal-difference (TD) learning algorithm uses a combination of both processes by attempting to balance the exploration and exploitation technique. On-policy methods aim to assess or improve the policy that is used to make decisions as well as obtain the data. On the contrary, off-policy methods aim to improve the policy-generating value function, different from the policy controlling the agent and behavior. [7]

Q-Learning: A model-free approach

The Q-Learning method of RL is an off-policy (episodic) TD learning which follows a model-free approach [7]. The steps followed to obtain optimal action-value function Q(s, a) in the Q-learning algorithm are stated below. ψ and γ are small positive values, where ψ is step-size influencing learning rate and γ is discount rate for maximal reward. [7]

Algorithm 1 : One-step Q-learning algorithm [7]

Initialize: $Q(s,a), \forall s \in S, a \in A(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
  Initialize s_t
  Repeat (for each step of episode):
    Choose a_t from $A(s)$ using policy derived from $Q(e.g. \varepsilon - greedy)$
    Take action a_t, observe r_t, s_{t+1}
    $Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \psi[r_t + \gamma \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$
    $s \leftarrow s_{t+1}$
  until s is terminal

In the above algorithm, $Q(s,a)$ is a table storing Q values for each state $'s'$ and action $'a'$ pair. It is initialized to zero before proceeding with actions and state transitions. $A(s)$ is the action space containing the possible actions available for each state. Select a start state s_t, to begin the learning process. The start state is picked arbitrarily every time in the initiation of an episode. After selecting a start node for the episode, evaluate and opt for action a_t from action space $A(s)$ to perform on s_t based on $\varepsilon - greedy$ policy to obtain a maximal reward. $\varepsilon - greedy$ policy algorithm says to select the action ($max_{a_{t+1}}$) which provides the maximum reward at the current state. Once the action a_t is taken, observe the response of the environment for the action a_t at state s_t. The environment reacts by providing a next state s_{t+1} and a reward r_t. The Q-table update rule can be rearranged and rewritten as [7]:

$$\begin{aligned} Q(s_t,a_t) \leftarrow \quad & \underbrace{(1-\psi)Q(s_t,a_t)}_{\text{Q value for the pair } (s_t,a_t)} + \underbrace{\psi[r_t+\gamma \ \max_{a_{t+1}} Q(s_{t+1},a_{t+1})]}_{r_t \text{ and estimated future reward}} \\ Q(s_t,a_t) \leftarrow \ & (1-\psi)Q(s_t,a_t) + \psi r_t + \psi\gamma \ \max_{a_{t+1}} Q(s_{t+1},a_{t+1}) \end{aligned} \tag{2.8}$$

The first term in equation 2.8 (below equation) is the discounted Q value, the second term is the reward obtained from the environment for the action performed at state s_t, and the last term is the bootstrapped value for the best future action. [7]

Then update the present state value to s_{t+1} and repeat the process of selecting an action, observing the environment response, and moving to the next state. Repeat the process until it reaches a terminal state. [7]

Later in this thesis, the Q-learning algorithm is used to optimize the size of a neural network for an efficient result. GNN struggles with the problem of neighborhood expansion, which is the endless continuation of the selection of neighbor nodes for aggregating features. This optimization technique helps in finding an efficient size of the hidden layer. It can also provide immense help in optimization if the scenario is studied further by setting the neighborhood details stored in the adjacency matrix as environment and action space as sampled neighbors of selective lengths. However, this study is not a part of this thesis.

3 Graph Neural Network Model

Data being the foundation of machine learning, can be represented in a variety of ways. Graphs are one of the potentially effective formats for data representation. Graphs are constructed by leveraging the structural relationships between data points. It has been studied that graphs can be used to address problems in a range of academic domains. [9] [29]

Graph G is denoted by an ordered pair of two finite nonempty sets (V, E), where V represents a set of vertices or nodes, and the set of edges or connections between the data points is denoted as E. In this thesis, the term node is used to explain the set of data points V. Each element of V is symbolized as v and a pair of nodes (v1, v2) denotes an edge. A connectivity that is having an edge between a pair of nodes indicates the two nodes are adjacent to each other. A single-order graph is the simplest and most trivial structure of a graph. However, numerous complex graphs are categorized into a variety of groups, including directed, undirected, cyclic, simple (free of cycles and self-loops), labeled, unlabeled, trees, and others. Based on the connectivity of the graph, it can also be divided into disconnected, partially connected, bipartite, and fully connected graphs. [30]

While talking about representing data in a graphical structure intuitively, an imagination can be popped, with some of the easily apprehensible examples such as chemical bonding among the elements and social networking among people. Datasets such as images as pixels, relationships between words in a text, connections between published papers, and many such datasets can be depicted as graphs. Below are some examples of data representation in terms of a graph. [9]

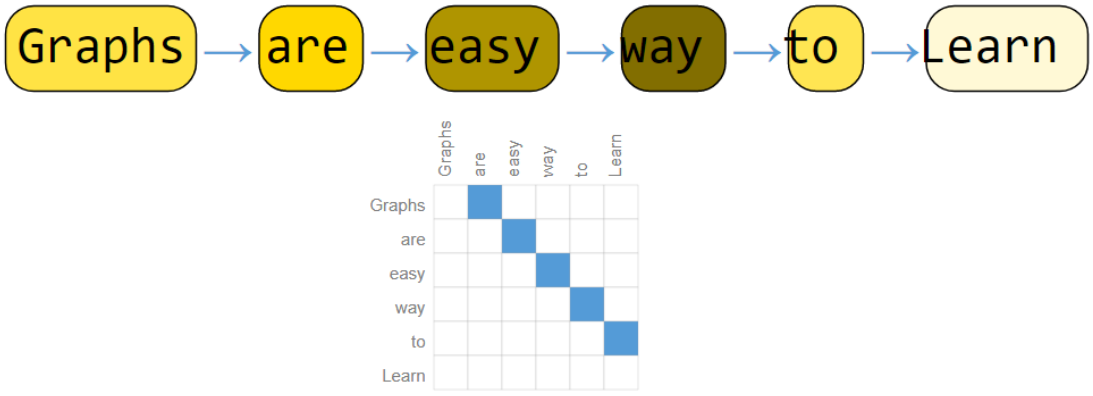
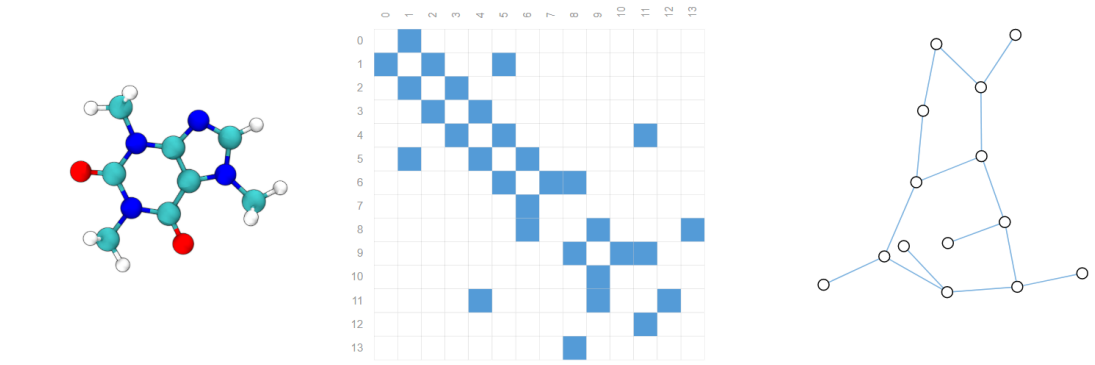


Figure 3.1: A sentence represented in adjacency matrix [9]

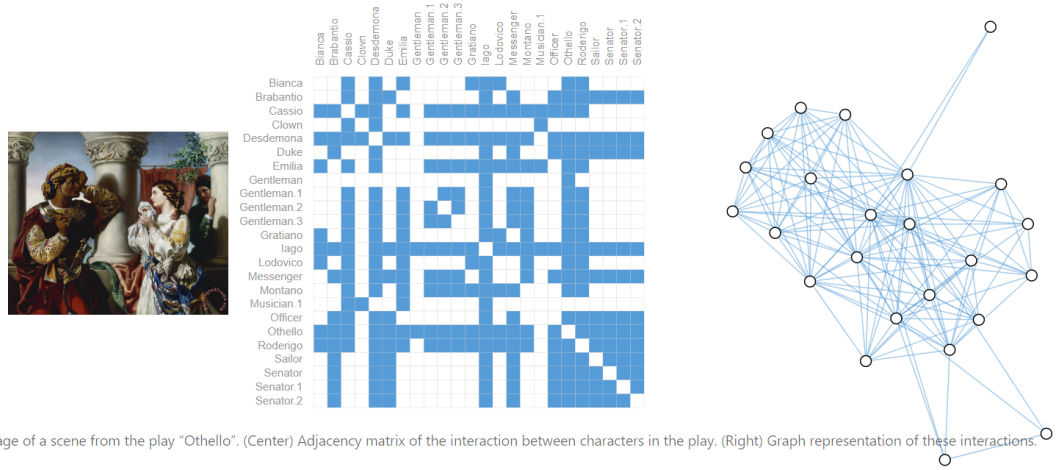
The aforementioned example illustrates how an adjacency matrix is produced from avail-

able data (here, it is a statement). The link between the data points (words) is maintained by the matrix. Two examples of molecules and social network data, their adjacency matrices, and graphical structures are shown below.



(Left) 3d representation of the Caffeine molecule (Center) Adjacency matrix of the bonds in the molecule (Right) Graph representation of the molecule.

Figure 3.2: Graphical representation, the adjacency matrix of molecules [9]



(Left) Image of a scene from the play "Othello". (Center) Adjacency matrix of the interaction between characters in the play. (Right) Graph representation of these interactions.

Figure 3.3: Graphical representation of the interaction between people enacting a play [9]

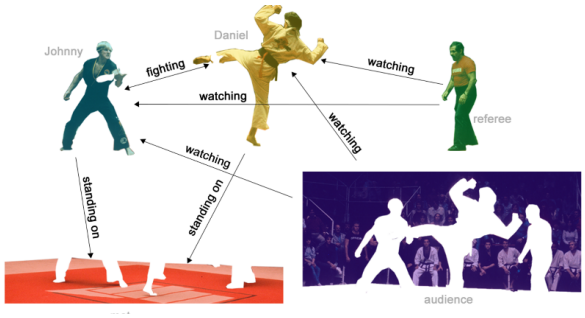


Figure 3.4: Interaction between players in a karate club displayed as a graph [9]

The karate club example shows how data points are presented as nodes with edges defining their connections. Labels for the nodes are distinguished by their colors, while edge labels are indicated by the texts highlighted over the edges.

3.1 Graph Theory Terminologies

This section discusses the fundamental concepts of graph theory and provides definitions for the helpful keywords. A graph is one of the simplest techniques for expressing information clearly. It offers a variety of approaches to perform actions on the data.

- A **graph** is an ordered pair G = (V, E), where V is a finite nonempty set of vertices and E is a 2-element subset of unordered pair of vertices. Vertex is also referred to as a point or node, similarly lines or links for edges. When E is a set of ordered pairs of distinct vertices, graph G is called **DiGraph** or **Directed Graph**. [30]
- A **loop** in graph G is an edge e(v,v) at vertex v if it connects a vertex to itself. If multiple numbers of edges are connected between a single pair of vertices (u, v), then the edges are called **parallel edges**, and such a graph G is defined as **Multigraph**. [30]
- A **walk** W(u,v) is a sequence of vertices, a subset of V in graph G of length 0 or more, which starts from u and ends at v. The consecutive pair of vertices in the sequence are **neighbor vertices**. It can visit a single node multiple times. A **cycle** is a walk of length two or more that starts and ends with the same node. A walk is a **path** if all the vertices covered in the sequence are distinct. The **length** of a walk or path, or cycle is the number of edge covers between the start and end vertex. [30]
- A pair of nodes (u, v) can have zero edges connecting them, the graph G is known as a **disconnected graph**. There does not exist a path between any two nodes of a graph. Such graphs contain multiple components. A **fully connected graph** is a graph G having a path of length at least one between each pairs of nodes. [30]
- The **size** of graph G (V, E) is the number of edges in a graph |E|, and the **order** of the graph is given by the number of nodes |V|. The trivial order of graph G is one, its size is zero, and G is a single node graph or a single isolated vertex. [30]
- A pair of vertices (u, v) in graph G are adjacent or neighbor to each other if there exists at least one edge connecting them. An edge in graph G is said to be incident on vertex v if one end of the edge starts or ends on v. [30]
- The **degree** of a vertex v in graph G denoted by deg(v) is the number of edges incident on vertex v or the number of vertices adjacent to vertex v. In a directed graph G the number of edges coming in and out of a vertex v are denoted by **indegree** and **outdegree**, respectively. [30]
- The **neighbors** N(v) of a vertex v in a graph G is a subset of V, induced by all vertices adjacent to v, which is equal to the degree of a vertex v. The **neighborhood graph** of a vertex v in a graph G is an induced subgraph of G composed of all the neighbor vertices of v and only the edges connecting the vertices in the set N(v). [30]

- A graph G1 is a **subgraph** of graph G(V, E) if the vertex set of G1 is a subset of V and the edge set of G1 is a subset of E. If both graphs G and G1 have the same set of vertices, then Graph G1 is called the **spanning tree** of graph G. When the edge set of graph G1 contains all the edges from graph G which has endpoints in the vertex set of G1, the graph is called **induced subgraph** of G. [30]
- Graph G, is called **labeled** if the vertices of the graph are labeled. Otherwise, it is known as an **unlabeled** graph. Graph G, also facilitates edge labeling based on various parameters like distance from the source node or priority. [30]
- Two labeled graphs G1(V1,E1) and G2(V2,E2), having an one to one mapping $\phi : G1 -> G2$ from V1 -> V2, such that an edge (u,v) $\in$ E1 if and only if edge mapping $(\phi(u), \phi(v)) \in$ E2, then G1 is **isomorphic** to G2. [30]
- A graph G(V, E,w) is an **edge-weighted graph** if there is a mapping w: E -> $\mathbb{R}$, each edge is assigned with a real value, and **vertex-weighted graph** if a real number is assigned to each vertex; the weight mapping w: V -> $\mathbb{R}$. [30]
- A **Bi-Partite** or **K-Partite** graph is graph G(V, E) which can be partitioned into Bi or K parts such that each subset of vertices contains distinct values and both the ends of edges do not belong to a single subset. A bipartite graph is a cycle-free graph. Acyclic (cycle-free) graphs are **forest**. A connected acyclic graph G is a **tree** if there exists a unique path between every pair of vertices. [30]
- A graph G(V,E) of order n, |V| = n and size m, |E| = m, the **adjacency matrix** of G is A = $[a_{ij}]$, a $n \times n$ matrix defined by [30]

$$a_{ij} = \begin{cases} 1, & \text{if (i,j)} \in \text{E} \\ 0, & \text{otherwise} \end{cases}$$

- A graph G(V,E) of order n, |V| = n and size m, |E| = m, the **incidence matrix** of G is B = $[b_{ij}]$, a $n \times m$ matrix defined by [30]

$$b_{ij} = \begin{cases} 1, & \text{if } v_i \text{ is incident to } e_j;\ e_j = (v_i, v_r) \\ -1, & \text{if G is DiGraph and } e_j = (v_r, v_i) \\ 0, & \text{otherwise} \end{cases}$$

- A graph G(V, E) of order n, |V| = n, and a_{ij} is the associated adjacency matrix, for any positive integer k, $a_{ij}^{\{k\}}$ = the number of walks of length k from vertex i to vertex j. [30]

	Milan	Paris	Dublin	Rome
Milan	0	0	19	5
Paris	8	0	11	0
Dublin	0	0	0	0
Rome	0	0	0	0

Figure 3.5: Example of an edge-weighted, labeled DiGraph and its adjacency matrix [10]

3.2 Graph Neural Network

Graph neural network (GNN) is a generalized methodology to implement Deep neural network (DNN) on graph data. The power of the graph is it elaborates on the relationship between the points as well as its general structure. Adjacency matrices, which preserve the connection between data points, can be used to perform DNN on graphs. The graph can be flattened into an adjacency matrix and feed the matrix as input to the neuron for Multi-layer perceptron (MLP). However, the output produced is limited to the order in which nodes are chosen while generating an adjacency matrix. The produced output is thus order-dependent which can not support the graph isomorphism problem. [11]

One of the advantages of applying DNN on graph data over regular data (array or matrix format) is GNN model satisfies permutation invariant, variable data size, and non-euclidean data space. Even though there has been much research in progress on GNN, the generation of graphs from datasets remains a challenge. However, this thesis does not involve graph generation from datasets. It only focuses on feature aggregation and task prediction on graphs. The fundamental idea of graph neural networks is to continuously update the node feature embedding by integrating its representation with those of its neighbors. This strategy serves as the framework for a GNN model known as the message-passing technique. The Neural message passing (NMP) phase and the readout phase are the two stages that make up the GNN model architecture. [11] [12]

Message Passing: Feature information associated with a graph often is node-level attributes and edge-level attributes. Neural message Passing (NMP) is a framework in which node features (vector messages) are exchanged between the nodes to figure out the hidden embeddings which is the representation vector corresponding to each node. A hidden embedding ${h_v}^{\{k\}}$ is obtained from the aggregated information from neighborhood N(v) of node v, for the time step k. The NMP approach is also known as message passing update due to its two-step procedure, the first step is the aggregate message, and the next is to update node embedding. The NMP update process leverages the graph structure that can be obtained from the adjacency matrix, and node features are indicated as X. Starting with initializing node embedding $h^{\{0\}} = X$, the two steps are performed in each layer of the model. [11] [12]

- Aggregate: The aggregated embedding for a node is obtained by collecting the feature vectors of all immediate neighbor nodes. [12]
- Update: The new embedding vector for the node is updated by considering the existing feature of the node and aggregated representation vector from neighbor nodes. [12]

The equations below show a step-by-step mathematical depiction of the two stages, which can be defined as the general framework of a GNN model [12].

Algorithm 2 : Message Passing Steps [12]

Initialize: $h^{\{0\}} = X$, For k = 1,2,$\cdots$,K,

$$agg_v{}^{\{k\}} = Aggregate^{\{k\}}\{h_u{}^{\{k-1\}} : u \in N(v)\} \tag{3.1}$$

$$h_v{}^{\{k\}} = Update^{\{k\}}\{h_v{}^{\{k-1\}}, agg_v{}^{\{k\}}\} \tag{3.2}$$

$agg_v{}^{\{k\}}$ is the aggregated information for node v, from neighborhood N(v). N(v) contains all the vertices which share an edge with v. u is known as a neighbor vertex of v if there exists an edge e_{uv} or a path of length k between the nodes, u and v. When k=0, each node v is initialized with its feature vector x_v. With each increment of k, k=1, the updated embedding considers the representation vector of neighbor nodes with path length one and similarly K-hop neighbor for k=K. The k^{th} update $h_v{}^{\{k\}}$ for the node v and layer k is obtained by considering aggregated message $agg_v{}^{\{k\}}$ and feature embedding from previous layer k-1 for node v, $h_u{}^{\{k-1\}}$. [12]

The fore mentioned *Aggregate* and *Update* methods satisfy the fundamental requirement for designing neural networks since they are differentiable functions. Therefore, an activation function can be applied to these methods to train the model, and most importantly, backpropagation and forward pass, can be performed with no hindrance. [29]

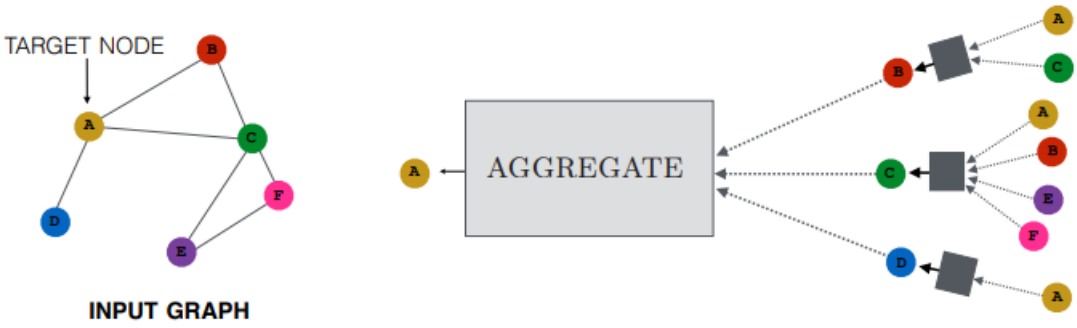


Figure 3.6: Unfolding neural message passing technique [11]

The above figure illustrates the message aggregation process implemented by a single node in its neighborhood. The image is a depiction of a two-layer message-passing framework. The embedding for node A is associated with the feature vectors of N(A) = {B, C, D}, subsequently the aggregation of each node in N(A) from its neighbor node embedding. Therefore, the node embedding update for the node v and k =2 depends on the feature vector of nodes in N(A)={B, C, D} = N(N(B), N(C), N(D)) = {B, {A, C}, C, {A, B, E, F}, D, {A}}. The message-passing approach in GNN unrolls the graph structure into a tree of depth k by unraveling the neighbor nodes of the selected node.

Readout Phase: Referring to equations mentioned above in Algorithm 1, $N(v)$ is the collection of neighbor nodes of node v, and $N(v)^k$ is a set of all the neighbor nodes

with path length k, starting from node v. The node representation ${h_v}^{\{k\}}$ is the final node embedding where $k = K$, which is updated by considering the previous layer embedding ${h_v}^{\{k-1\}}$ and feature embedding aggregated considering attributes of neighbors of node v of length up to k, ${agg_v}^{\{k\}}$. [12]

Once the final layer K is reached and representation embedding for K^{th} layer is obtained ${h_v}^{\{K\}}$, the subsequent task can be processed to train the model by applying weight and bias on the final layer feature embedding followed by an appropriate activation function suitable for the expected outcome. For example, as this thesis focuses on node classification, a SoftMax activation function will help in predicting the result, the loss can be calculated using cross entropy and to achieve minimum loss, the NN is trained using backpropagation. This process of training the model to attain the result using the feature embedding obtained from the previous step is defined as the readout phase. [12]

3.2.1 A Basic GNN Model:

So far, the basic components of GNN are discussed abstractly as a sequence of message-passing steps iteratively by aggregate and update functions followed by a readout layer at the end for the conclusion. Putting them all together into implementation and building a basic GNN by instantiating the aggregate and update methods with the parameters, GNN representation learning is given by, [11]

$$ {h_v}^{\{k\}} = \sigma\left({W_{self}}^{\{k\}} {h_v}^{\{k-1\}} + {W_{neigh}}^{\{k\}} \sum_{u \in N(v)} {h_u}^{\{k-1\}} + b^{\{k\}} \right) \tag{3.3} $$

${h_v}^{\{k\}}$ is the output of the message passing layer and input to the final layer for output prediction. ${h_v}^{\{k\}}$ is calculated by applying the sigmoid activation function (σ) to the weighted aggregated value with previous embedding. The first term in the equation is the product of the weight matrix for node v at k^{th} layer and embedding of node v derived from $(k-1)^{th}$ layer. The second term is the weighted sum of the weight matrix for all the neighbor nodes of v at k^{th} layer, the hidden embedding of each neighbor node from $(k-1)^{th}$ layer ${h_u}^{\{k-1\}}$ and bias b at k^{th} layer. Considering the example of node classification, the output of the readout layer can be defined as [12]

$$ \hat{y}_v = SoftMax(W^{\{k+1\}} {h_v}^{\{k\}}) \tag{3.4} $$

$\hat{y}_v$ is the estimated output of the model by applying the SoftMax activation function on $W^{\{k+1\}}$ weight matrix for the last (K+1) layer and embedding estimated from the previous layer. To train the GNN model, given expected labels y_v, n_l be the number of labeled nodes, the loss function L can be calculated from [12]

$$ L = \frac{1}{n_l} \sum_{v=1}^{n_l} crossentropyloss(\hat{y}_v, y_v) \tag{3.5} $$

The output function $\hat{y}_v$ is derived using the SoftMax activation function and loss L is calculated using the cross-entropy loss function is an example considering the task as node classification. Machine Learning models are designed to learn from data and produce the desired result based on the task to be solved. Therefore, the above equations vary based on the problem to be solved. [12]

3.2.2 Graph Learning:

The prominent difference which makes GNN stand out is the data (node here) are not IID (independent and identically distributed) which is the primary assumption for the existing models. On the contrary, in the GNN model, the neighbor nodes share a correlation of feature attributes. Another advantage of graph data is during the training phase of the model, the complete graph is provided as input, keeping the test labels hidden. So, the node features for test data available during the training phase enhance the graph learning credibility. However, the consideration of the data for observation is decided based on the selected learning type. The transductive learning method is exposed to the datasets for observation, while the inductive learning process gets exposed only to the training node features and labels. [11]

Even though this thesis has already addressed a wide range of theories and technical aspects, one crucial aspect of machine learning has not yet been established. In machine learning, a model is designed to solve a particular type of task. Graph learning tasks can be three kinds of problems as graphs can be analyzed at various component levels: node, edge, and graph. For node-level embedding, the tasks can be segregated in classification, clustering, regression, etc similar to existing neural network models. For edge embedding, the possible problem to solve is edge classification and edge prediction. At the graph level, community detection, and the relationship between graph components are new directions along with graph classification and graph clustering. [17] [11]

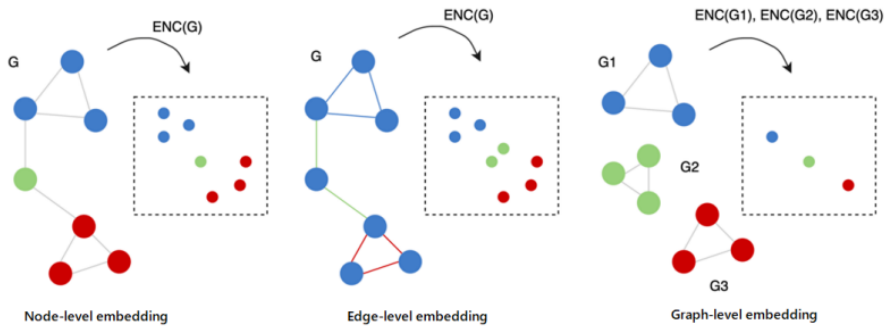


Figure 3.7: Types of graph embedding (classification problem) [10]

The figure above explains the classification task on each granular level of a graph. In the left segment, node level embedding image, the nodes are classified into three classes

distinguished by three different colors. Similarly in the middle section, the edges are classified into three different classes based on edge-level embedding. The final image in the right corner displays how the graphs are categorized into different classes.

Problem Statement: Formally defining the problem to be explored in this thesis, the challenge is to learn node-level embedding for node classification using GNN. Let G = (V, E) a graph with V as the node set and E as the edge set. With N number of nodes, C number of features of each node, adjacency matrix $A \in R^{N \times N}$ and node attribute matrix $X \in R^{N \times C}$, the goal is to learn effective hidden embedding $h \in R^{N \times F}$ for nodes by taking into account node features and graph structure where F is the size of hidden embedding. Then, the node embedding is further used for node classification. [12]

Concept	Notation
Graph	$\mathcal{G} = (\mathcal{V}, \mathcal{E})$
Adjacency matrix	$A \in \mathbb{R}^{N \times N}$
Node attributes	$X \in \mathbb{R}^{N \times C}$
Total number of GNN layers	K
Node representations at the k-th layer	$h^k \in \mathbb{R}^{N \times F}, k \in \{1, 2, \cdots, K\}$

Figure 3.8: Overview of notations used to define problem statement [12]

Further, this thesis discusses a few of the most popular GNN models, such as GCN, GAT, and GraphSage, before proceeding with the proposed model approach for the defined problem statement.

3.2.3 Graph Convolutional Network (GCN):

GCN is one of the simplest architecture-bearing GNN models with higher effectiveness in various tasks, domains, and applications. This model also follows the above-highlighted stages of GNN in each layer. Hence, a hidden embedding is calculated for each node in each layer. The update propagation rule for learning node representation learning in each layer is [12] [18]

$$h^{\{k+1\}} = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}h^K W^K) \tag{3.6}$$

$$\hat{A} = A + I, \qquad \hat{D}_{ii} = \sum_j \hat{A}_{ij} \tag{3.7}$$

Adjacency matrix A is regularized by adding self-connections to the nodes resulting in $\hat{A}$ and $\hat{D}$ is the diagonal matrix where $\hat{D}_{ii}$ is the degree of a node considering the self connections. Self-connection is critical to consider as the aggregated feature always considers features of the node along with the neighbor node features. Layer-specific parameters are W^K is the weight matrix, $\sigma(.)$ is a non-linear activation function such as

ReLU, and h^K is the input to the layer K+1 that is the hidden embedding of the previous layer. h^0 is X, the input graph. The above equation can be reformulated considering the aggregate and update methods of the readout phase as [12] [18]

$$h_i^{\{k\}} = \sigma\left(\sum_{j \in N(i) \cup i} \frac{\hat{A}_{ij}}{\sqrt{\hat{D}_{ii}\hat{D}_{jj}}} h_j^{\{k-1\}} W^{\{k\}}\right) \tag{3.8}$$

$$h_i^{\{k\}} = \sigma\Bigg(\underbrace{\sum_{j \in N(i)} \frac{\hat{A}_{ij}}{\sqrt{\hat{D}_{ii}\hat{D}_{jj}}} h_j^{\{k-1\}} W^{\{k\}}}_{\text{Aggregated neighbor nodes features}} + \underbrace{\frac{1}{\hat{D}_i} h_i^{\{k-1\}} W^{\{k\}}}_{\text{Node features through self connection}}\Bigg) \tag{3.9}$$

The preceding equation 3.8 can be expanded to equation 3.9 by considering the adjacency matrix split. With the two terms in the later equation, it is evident that the first term aggregates the neighbor node features, and the second term helps in updating the node representation considering its previous layer representation. Here, i is the node for which hidden embedding is processed, whereas j is the neighbor considered for aggregating features. [12] [18]

The **spectral convolutions** on graphs can be defined as the multiplication of input signal x of a node with a convolutional filter $g_\theta = diag(\theta)$, (θ is the parameter of the filter) in the Fourier domain. [12] [18]

Normalized graph Laplacian matrix, $L = I_N - D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$

Laplace matrix in terms of eigenvectors U, $L = U \Lambda U^T$

Graph convolution, $g_\theta * x = U g_\theta U^T x$

λ is the diagonal matrix of eigenvalues for the eigenvector U of L, $U^T x$ is the graph Fourier transform of x, and T is the matrix transpose. Therefore, g_θ is a function of eigenvalues (λ) of L. Computing eigenvectors and eigenvalues of L is a quadratic polynomial problem. To smoothen the quadratic problem, the laplacian matrix is rescaled to $\hat{L} = \frac{2}{\lambda_{max}} L - I_N$, λ_{max} is the largest eigenvalue of L. [12] [18]

A **layer wise** convolutional, non-linear model, with the fundamental notion to have at least one hidden layer, K= 2. By considering the above-defined terms L, $g_\theta * x$, and equation 3.9, [12] [18]

$$g_{\theta'} * x \approx \theta_0' x + \theta_1' (L - I_N) x = \theta_0' x - \theta_1' D^{-\frac{1}{2}} A D^{-\frac{1}{2}} x \tag{3.10}$$

$$g_\theta * x \approx \theta (I_N + D^{-\frac{1}{2}} A D^{-\frac{1}{2}}) x \tag{3.11}$$

The filter parameters in equation 3.10, θ_0', θ_1' are shared across the graph to effectively convolve k^{th} neighborhood of the graph, where k is the number of the convolutional

layer in the neural network model. To reduce the filter parameters and operations to be performed, it can be further processed as can be seen in equation 3.11, with $\theta = \theta_0^{'} = -\theta_1^{'}$. Due to the rescaled value of L for normalization, the eigenvalue now ranges in [0,2]. Hence, while optimizing the learning parameters and training the model, it bears a risk of vanishing or exploding gradients. Therefore, another renormalization trick is introduced by adding self-connections (I_N) to adjacency matrix A and replacing the diagonal matrix (D) with the diagonal matrix ($\hat{D}$) of the new adjacency matrix ($\hat{A}$) as seen before from equations 3.6 and 3.7. The new equation for the convolved signal is [12] [18]

$$Z = f(X,A) = \hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}X\Theta \tag{3.12}$$

The loss function (L) with Laplacian regularization (L_{reg}), $L = L_0 + \lambda L_{reg}$.

where, $L_{reg} = \sum_{i,j} A_{ij}||f(X_i) - f(X_j)||^2 = f(X)^T \Delta f(X)$, $\Delta = D - A$. L_0 is the supervised loss for known labels of the graph, Δ is the unnormalized graph Laplacian matrix, X is the input matrix, T is matrix transpose, f(.) is a differentiable function like an MLP or a NN or mean, and ||.|| is used to represent a dissimilarity measure. [18]

3.2.4 Graph Attention Networks

In the process of learning node representation, the node features are processed from one layer to the next layer until the resulting layer. In this process of learning the number of features carried further in each layer reduces keeping the defining features intact. One of the major challenges encountered in the GCN model for graph networks is the missing importance of each node to all other nodes. To overcome the challenge and maintain the efficacy throughout the layers of NN, the GAT network is built and tries to learn the priority of each neighbor of a node based on **attention mechanism** by introducing a graph attention layer. Graph attention layer leverages effective transfer of expressive power by transforming the lower level node representation to higher level node representation by applying a shared linear transformation on every node, which leads to more appropriate hidden embedding transfer from $(k-1)^{th}$ layer to k^{th} layer and higher accuracy at the end. The importance of node j to node i is defined as the attention coefficient given by [12] [21]

$$e_{ij} = a(Wh_i^{\{k-1\}}, Wh_j^{\{k-1\}}) \tag{3.13}$$

W is the shared linear transformation applied to node features, and a is the shared self-attention mechanism applied to the nodes. To implement this self-attention mechanism in a graph network, it is important to consider the graph structure to calculate the attention coefficient e_{ij}. Therefore, e_{ij} is calculated for all the nodes j in neighbor N(i) of node i, which are the first-order neighbors of i. The relationship of other nodes, not in N(i) to node i, is ignored here. The attention coefficients are typically normalized with

the SoftMax function to make them comparable across different nodes: [12] [21]

$$\alpha_{ij} = SoftMax_j(\{e_{ij}\}) = \frac{exp(e_{ij})}{\sum_{l \in N(i)} exp(e_{il})} \tag{3.14}$$

The shared attention mechanism a is a unit layer of a feedforward neural network, including a linear transformation with a non-linear activation function (LeakyReLU), weight vector for the layer W_2. The attention coefficient in equation 3.14 is redefined as [21] [12]

$$\alpha_{ij} = \frac{exp(LeakyReLU(W_2[Wh_i{}^{\{k-1\}} || Wh_j{}^{\{k-1\}}]))}{\sum_{l \in N(i)} exp(LeakyReLU(W_2[Wh_i{}^{\{k-1\}} || Wh_l{}^{\{k-1\}}]))} \tag{3.15}$$

where || represents the concatenation of two vectors. Therefore, the new hidden embedding of a node is defined composedly by weights determined by the attention coefficient attained from non-linear transformation (equation 3.15) and the neighbor node representation: [21] [12]

$$h_i^{\{k\}} = \sigma\left(\sum_{j \in N(i)} \alpha_{ij} W h_j{}^{\{k-1\}} \right) \tag{3.16}$$

Instead of a shared single attention mechanism, a multi-head attention mechanism can be applied to nodes, and the final representation for a node is finalized by concatenating all the node representations learned by different attention heads. Equation 3.16 is reformulated for a multi-head attention mechanism with H attention heads as [12] [21]

$$h_i^{\{k\}} = \Big|\Big|_{H_a=1}^{H} \sigma\left(\sum_{j \in N(i)} \alpha_{ij}^{H_a} W^{H_a} h_j{}^{\{k-1\}} \right) \tag{3.17}$$

$\alpha_{ij}^{H_a}$ is the attention coefficient and W^{H_a} is the linear transformation matrix attained for H_a^{th} head. The hidden embedding for a node can be concluded by using any of the other pooling techniques(mean, max) instead of concatenation feature vectors. The node embedding can be calculated by averaging the feature vectors as: [21] [12]

$$h_i^{\{k\}} = \sigma\left(\frac{1}{H} \sum_{H_a=1}^{H} \sum_{j \in N(i)} \alpha_{ij}^{H_a} W^{H_a} h_j{}^{\{k-1\}} \right) \tag{3.18}$$

3.2.5 GraphSage:

The node representation learning dwells on the underlying principle behind node embedding approaches, that is to use dimensionality reduction techniques to extract the high-dimensional information about a node's neighborhood into a dense vector embedding. Attaining higher efficacy in performing the task using this dense embedding falls easier when the features and the labels are known. However, it becomes challenging to generate efficient node embedding for unseen data, like in inductive learning tasks. [20]

Algorithm 3 GraphSage algorithm [20]

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
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

GraphSage exploits the available node features and graph structure information to acquire an effectively hidden embedding. GraphSage can be stated as a 2-stage extension of GCN. GraphSage extends the aggregate method from the mean (in GCN) operator to pooling operators and LSTM. GraphSage replaces the sum operator by concatenation in the update phase. This concatenated result is fed through a fully connected layer with a non-linear activation function which transforms the vector to be used for the next layer. To ease the computation for larger graphs, GraphSage facilitates mini-batch processing, which can have one or more elements. It also fixes the size of neighbor nodes to be sampled to reduce the neighborhood expansion, which might result in sampling quality. To attain an effective representation in unsupervised conditions, a graph-based loss function L_G is applied to the output embedding z_u for node u, trains the weight matrix, and aggregator function parameters through SGDL. This training method leads to similar representations for nearby nodes and higher disparity for distant nodes. [20]

$$L_G(z_u) = -log(\sigma(z_u^T z_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} log(\sigma(-z_u^T z_{v_n})) \tag{3.19}$$

where v is the node occurring on fixed length random walk from u, T is the matrix transpose operation, P_n is a negative sampling distribution and Q is the number of negative samples. The loss function is applied on attained node embedding instead original embedding of the node. In the GraphSage algorithm, by setting $K = |V|$, W = I, and an appropriate hash function as an aggregator function, the algorithm behaves as the Weisfeiler-Lehman isomorphism test. If the resulting embedding for two graphs turns out to be the same, then the graphs are stated as isomorphic. [20]

4 Experiments

Along with the increase in research work in the field of GNN, the application spreads across various domains. Amongst the wide range of GNN applications, this thesis focuses on a specific type of dataset and learning hidden embedding of the same to perform node classification. The following experimental setup and the performance evaluation are conducted using a citation networks and recommendation networks.

Technical specification to establish:

- Python
- PyTorch
- PyTorch geometric
- Matplotlib
- Keras

4.1 Datasets:

This thesis considers citation network datasets such as Cora, Citeseer, and PubMed datasets for classifying academic papers into predefined categories of subjects. The thesis also takes into account the recommendation networks such as Amazon Photos and Amazon Computers. All of these discussed datasets contain a set of the bag of words playing the key role in performing the task of classification. Below, the datasets are elaborated further in detail concentrating on the important features. However, a point to be kept in mind is the datasets are not necessarily a single component graph. For an incident to explain, the Cora dataset consists of only one graph, a single connected component whereas the Amazon Photo dataset is a disconnected multicomponent graph with a single graph along with some of the isolated nodes. All the graphs here are treated as undirected graphs.

	Cora	**Citeseer**	**PubMed**	**Computers**	**Photo**
Number of Nodes	2708	3327	19717	13752	7650
Number of Edges	10556	9104	88648	491722	238162
Number of Features	1433	3703	500	767	745
Number of Classes	7	6	3	10	8

Table 4.1: Graph datasets details

A Brief on Datasets Based on Learning Tasks:

The proposed model performs the classification task using both transductive learning as well as inductive learning technique individually. **Transductive learning** technique is applied by exposing a complete set of nodes and node features to train the model. The node features of the validation set and test set nodes are available for the model to learn during the training phase. However, in **Inductive learning** methodology, a set of nodes and respective node features are fed to the model during the training phase where as other nodes are not opened up to the model before the phase. This thesis has tried to implement inductive learning by segregating the graph into 3 subgraphs based on the node splits attained for the train-validation-test phases.

Details Of Transductive Learning On Graph Dataset

```
Dataset: Cora()
-------------------
Number of graphs: 1
Number of nodes: 2708
Number of features: 1433
Number of classes: 7

Graph:
------
Training nodes: 1208
Evaluation nodes: 500
Test nodes: 1000
Edges are directed: False
Graph has isolated nodes: False
Graph has loops: False
```

```
Dataset: Citeseer()
-------------------
Number of graphs: 1
Number of nodes: 3327
Number of features: 3703
Number of classes: 6

Graph:
------
Training nodes: 1827
Evaluation nodes: 500
Test nodes: 1000
Edges are directed: False
Graph has isolated nodes: True
Graph has loops: False
```

```
Dataset: Pubmed()
-------------------
Number of graphs: 1
Number of nodes: 19717
Number of features: 500
Number of classes: 3

Graph:
------
Training nodes: 18217
Evaluation nodes: 500
Test nodes: 1000
Edges are directed: False
Graph has isolated nodes: False
Graph has loops: False
```

```
Dataset: AmazonComputers()
-------------------
Number of graphs: 1
Number of nodes: 13752
Number of features: 767
Number of classes: 10

Graph:
------
Training nodes: 12252
Evaluation nodes: 500
Test nodes: 1000
Edges are directed: False
Graph has isolated nodes: True
Graph has loops: False
```

```
Dataset: AmazonPhoto()
-------------------
Number of graphs: 1
Number of nodes: 7650
Number of features: 745
Number of classes: 8

Graph:
------
Training nodes: 6150
Evaluation nodes: 500
Test nodes: 1000
Edges are directed: False
Graph has isolated nodes: True
Graph has loops: False
```

Details Of Inductive Learning On Graph Dataset

```
Graph:Cora()
------
Training nodes: 1208
Number of Edges in Training Set: 2228
Evaluation nodes: 500
Number of Edges in Validation Set: 316
Test nodes: 1000
Number of Edges in Test Set: 1566
```

```
Graph:Citeseer()
------
Training nodes: 1827
Number of Edges in Training Set: 2602
Evaluation nodes: 500
Number of Edges in Validation Set: 258
Test nodes: 1000
Number of Edges in Test Set: 824
```

```
Graph:Pubmed()
------
Training nodes: 18217
Number of Edges in Training Set: 75966
Evaluation nodes: 500
Number of Edges in Validation Set: 64
Test nodes: 1000
Number of Edges in Test Set: 202
```

```
Graph:AmazonComputers()
------
Training nodes: 12252
Number of Edges in Training Set: 380964
Evaluation nodes: 500
Number of Edges in Validation Set: 808
Test nodes: 1000
Number of Edges in Test Set: 3020
```

```
Graph:AmazonPhoto()
------
Training nodes: 6150
Number of Edges in Training Set: 152006
Evaluation nodes: 500
Number of Edges in Validation Set: 1184
Test nodes: 1000
Number of Edges in Test Set: 3852
```

4.2 Architecture:

The challenges this thesis has tried to concentrate on are feature aggregation using a recurrent neural network before feeding it to GNN and optimizing the size of the hidden layer using Q-learning. The proposed thesis model architecture is modeled in numerous steps, which are elaborated in steps in the following sections.

4.2.1 Setting up GNN models:

Setting Up The Base Model: An initial base setup is established for the three outperforming models Graph convolutional networks (GCN), GraphSage (Sampling and Aggregate), and Graph attention networks (GAT), and their test accuracy percentage along with the validation cross-entropy loss are noted. This training of this base model is performed using the ReLU activation function for the inner layers, SoftMax for the final layer, ADAM as an optimizer, and a constant dropout in each layer for regularization. Two layered convolutional networks (GCN and GAT) and a three-layered convolutional sage network (GraphSage) outputs are outlined below, followed by a tabular report of the excellence of these models researched on the selected datasets [13]. GraphSage is denoted as GS in the table from the referred paper.

	CORA	CiteSeer	PubMed	CORA Full	Amazon Computer	Amazon Photo
GCN	81.5 ± 1.3	$\mathbf{71.9} \pm 1.9$	77.8 ± 2.9	$\mathbf{62.2} \pm 0.6$	82.6 ± 2.4	91.2 ± 1.2
GAT	$\mathbf{81.8} \pm 1.3$	71.4 ± 1.9	$\mathbf{78.7} \pm 2.3$	51.9 ± 1.5	78.0 ± 19.0	85.7 ± 20.3
GS-mean	79.2 ± 7.7	71.6 ± 1.9	77.4 ± 2.2	58.6 ± 1.6	82.4 ± 1.8	$\mathbf{91.4} \pm 1.3$
GS-maxpool	76.6 ± 1.9	67.5 ± 2.3	76.1 ± 2.3	40.7 ± 1.5	N/A	90.4 ± 1.3
GS-meanpool	77.9 ± 2.4	68.6 ± 2.4	76.5 ± 2.4	40.5 ± 1.5	79.9 ± 2.3	90.7 ± 1.6

Figure 4.1: Overview of the performances of selected models on preferred datasets [13]

The base models tried to set up, trained in mini-batches, and recorded the outcomes for GCN, GAT, and GraphSage-mean. The preferred datasets for the execution of these models are only the citation network Cora, Citeseer, and PubMed, the result for which are projected below.

	Cora					Citeseer					Pubmed				
	Tr Acc	Val Acc	Test Acc	Tr Loss	VLoss	Tr Acc	Val Acc	Test Acc	Tr Loss	VLoss	Tr Acc	Val Acc	Test Acc	Tr Loss	VLoss
GAT	69.34%	42.28%	79.20%	0.795	3.04	76.42%	32.3	68.00%	0.624	4.35	87.87%	50.17%	77.30%	0.349	0.89
GCN	96.62%	64.55%	79.20%	0.093	1.6	97.67%	53.38%	68.00%	0.052	2.94	100%	76.71%	75.30%	0.048	0.64
Graphsage-Mean	100%	59.29%	76.80%	0.002	1.99	98.75%	54.17%	63.60%	0.041	6.66	98.53%	77.50%	74.70%	0.013	0.8

Figure 4.2: Performances of selected models attained on chosen datasets

4.2.2 Introducing recursive unit to GNN:

GraphSage model has the advantage of extending its aggregator as LSTM. This thesis tried to set up a GraphSage model with an LSTM aggregator. The established model

could not provide a promising result for the simpler networks and performed slightly well in the moderately complex network. However, the execution time remained a challenge for batch training. This thesis primarily focuses on extending GCN with recursive units and optimizing the hidden layer size with Q-learning.

setting up proposed model: The model can be segregated into three parts, the first part comprises a two-layered RNN (simpleRNN or LSTM or GRU), the second part consist of a two convolutional layer GCN, and the last part is a Q-table optimizing the size of the hidden layer for both RNN and GCN. The activation function used for the inner layers is ReLU, for the final layer is SoftMax and ADAM is the optimizer for both networks. The learning rate is provided as 0.1, the discount factor for Q-Learning is 0.9, the dropout value for GCN is settled in [0.4, 0.6], the Q-table is initialized to zero matrix, and the epoch count is set to 200. The model is approached in steps stated below.

Step 1: Node features are provided as input to RNN for aggregating the crucial node features.
Step 2: Features attained in step 1 are provided to GCN as input. GCN works in two folds. First, it aggregates node embedding by considering a subset of the neighbor node, and then it uses the attained hidden embedding for a node classification task.
Step 3: Q-table samples the hidden layer with different sizes, and the matrix value gets updated based on the accuracy of the network.

The model is run in two phases, one enabling all the nodes and connections in the training stage. While in the next phase, the graph is divided into two induced-subgraphs as train and test. The model is trained using a train graph and tested with a test graph. The nodes and links of the test graph are hidden from the model in the training stage. The result of both stages is recorded in terms of loss and accuracy plots along with an analysis in the confusion matrix. This model training and testing is performed using all the fore mentioned datasets, Cora, Citeseer, PubMed , AmazonPhoto, and AmazonComputers. RNN-GCN-RL notation used for suggested model, 'Tr Acc' for training accuracy, Val for validation. Trans- transductive, Induct- Inductive.

	Cora				Citeseer				Pubmed			
	Tr Acc	Val Acc	Test Acc	Tr Loss	Tr Acc	Val Acc	Test Acc	Tr Loss	Tr Acc	Val Acc	Test Acc	Tr Loss
RNN-GCN-RL-Trans	89.16%	83%	84%	0.6691	72.25%	76.60%	76.60%	1.1513	85.12%	85%	84.50%	0.4951
RNN-GCN-RL-Induct	82.04%	50.60%	55.20%	1.064	70.33%	50%	53.50%	1.2059	85.24%	78.60%	77.40%	0.4899

Figure 4.3: Transductive-Inductive learning results - **Planetoid (Cora, Citeseer and PubMed)** datasets

	AmazonComputers				AmazonPhoto			
	Tr Acc	Val Acc	Test Acc	Tr Loss	Tr Acc	Val Acc	Test Acc	Tr Loss
RNN-GCN-RL-Transductive	79.82%	79.60%	79.30%	0.8032	88.98%	87.40%	89.10%	0.6084
RNN-GCN-RL-Inductive	80.60%	58%	59.90%	0.8082	81.11%	58%	55.50%	0.8234

Figure 4.4: Transductive-Inductive learning results - **Amazon (Computers and Photo)**

Discussing Results: This section will enable the user to understand the charts of results flooded here. Before discussing and analyzing the plots, a few factors to consider are: a) the loss result plotted are cross-entropy values, b) the accuracy value is normalized in the range [0,1] to plot, but noted in percentage for tabular representation (figure 4.3 4.4). Accuracy charts are plotted on a plane with the number of epochs on the x-axis and a range from zero to maximum accuracy attained on the y-axis. Training loss is charted for loss in the y-axis attained for the number of epochs on the x-axis. The confusion matrix is drawn by comparing the actual labels on the y-axis and the predicted labels on the x-axis.

It can be seen from the model's results, which are shown in tables from figures 4.3 and 4.4, that the transductive learning setup of the model performed competitively well. The transductive learning process managed to achieve an average test accuracy of 80% with a deviation of 1%. However, the inductive model could not perform similarly and managed to attain an average of 55% with a deviation of 1%. The loss during training remained consistent in both processes.

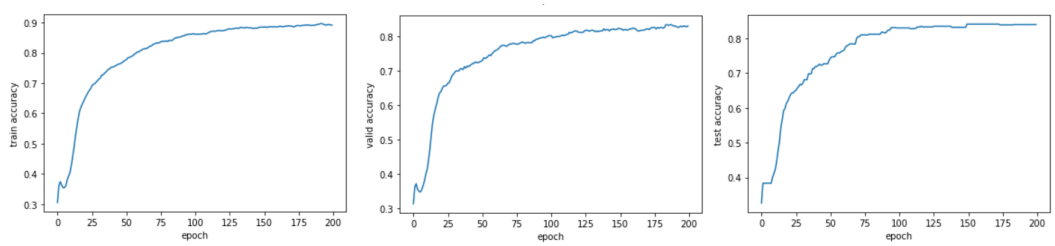


Figure 4.5: Transductive learning accuracy result- **Cora** dataset

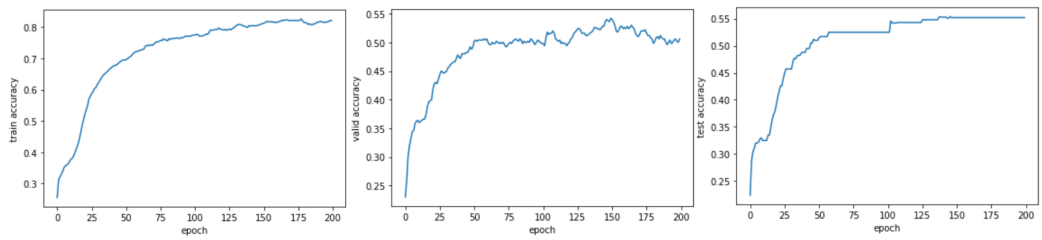


Figure 4.6: Inductive learning accuracy result- **Cora** dataset

Cora Dataset Results: The model ran on the Cora dataset, attained a training accuracy closer to 90%, and performed quite well with an accuracy of $83-84\%$ which can be seen in figure 4.5. The learning growth and reduction in the loss was exponential in the initial phase of training until the 100-125 epoch, post which the gradient became smaller and the learning of the model became stagnant, which can be seen in figures 4.5 - 4.7. As can be seen in figure 4.6, the inductive learning model resulted in an accuracy of 55%, the learning growth was reduced after 100 epochs. The confusion matrix shows, the count of correctly classified data and the deviation in wrongly classified data in various categories. Figure 4.8, the right side image significantly displays an inclination of test results towards the higher number of training samples based on categories.

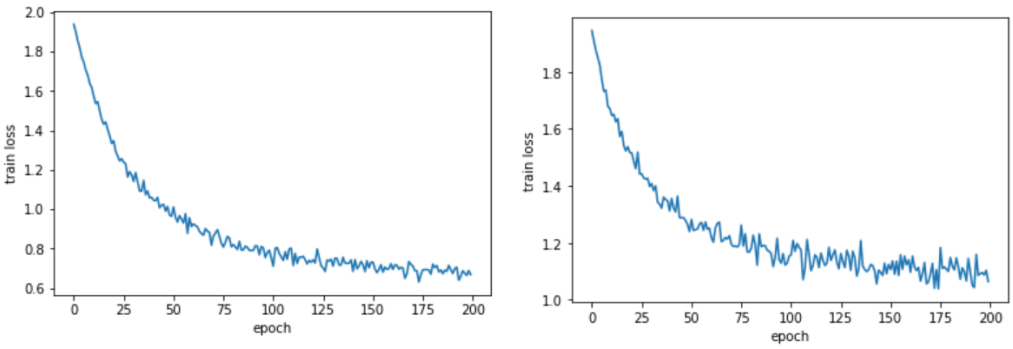


Figure 4.7: Training Loss- **Cora** dataset (L-Transductive, R-Inductive)

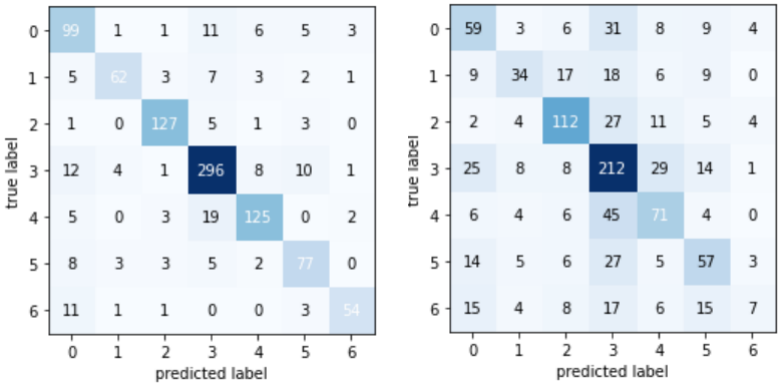


Figure 4.8: Confusion matrix result- **Cora** dataset (L-Transductive, R-Inductive)

That conveys, if a class has higher training samples compared to other classes, the majority of the incorrectly classified data are identified to that label. The Cora dataset has seven predefined categories of data, the numeric values representing the categories labeled in the range [0,6] (axis values of confusion matrix).

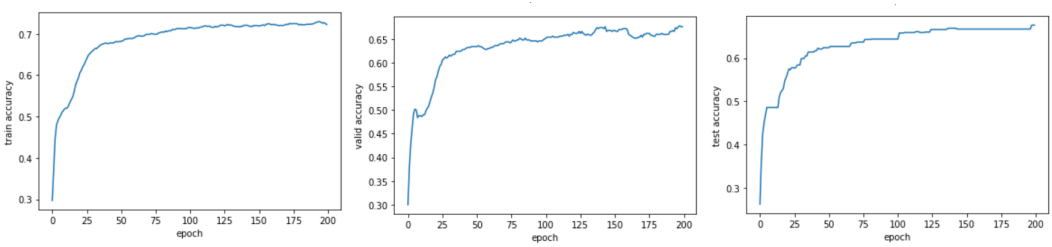


Figure 4.9: Transductive learning accuracy result- **Citeseer** dataset

Citeseer Dataset Results: The model executed for the Citeseer dataset resulted almost similar to the Cora dataset, attained a transductive learning test accuracy closer to 76% and inductive learning test accuracy closer to 55%, which can be seen in figures 4.9 4.10. The stagnant phase is attained after training of 100-125 epochs for inductive learning, whereas for the transductive process, it is attained in a lesser number of epochs as seen in figure 4.11. The confusion matrix in figure 4.12 shows a similar deviation to the result attained for Cora.

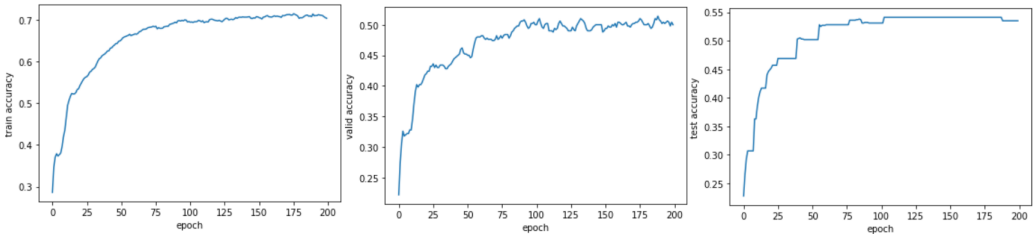


Figure 4.10: Inductive learning accuracy result- **Citeseer** dataset

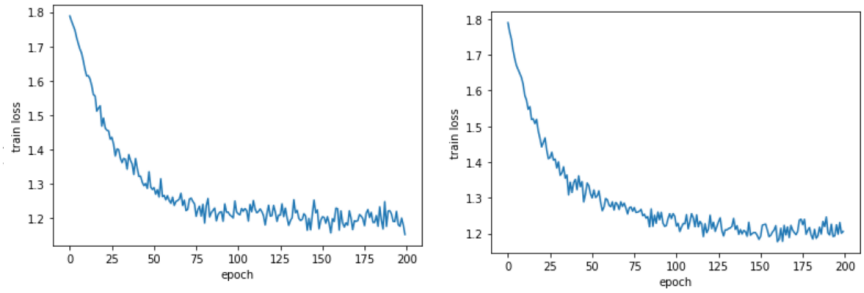


Figure 4.11: Training Loss- **Citeseer** dataset (L-Transductive, R-Inductive)

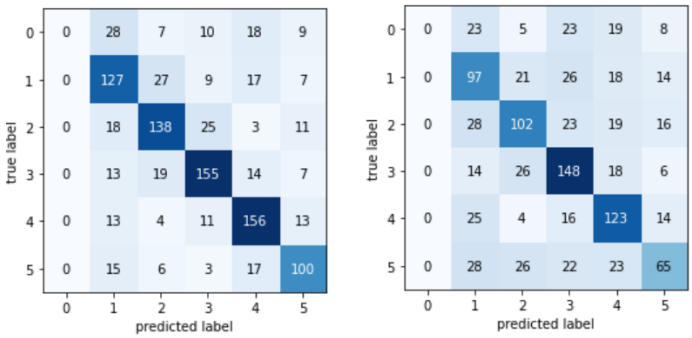


Figure 4.12: Confusion matrix result- **Citeseer** dataset (L-Transductive, R-Inductive)

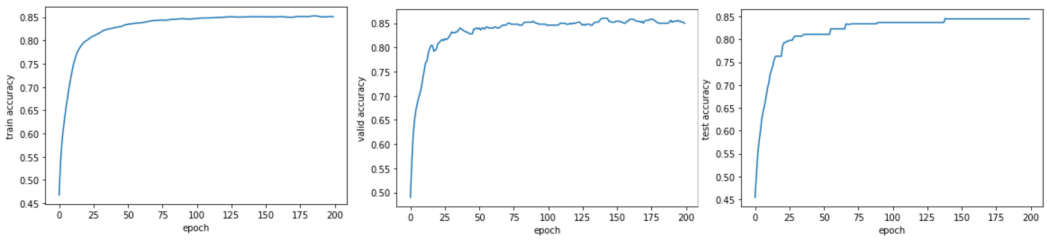


Figure 4.13: Transductive learning accuracy result- **PubMed** dataset

PubMed Dataset Results: Although the PubMed dataset is sufficiently large and complex with comparatively more node features, due to a lesser number of categories in data, the model performed distinctively well in both transductive learning with test accuracy closer to 85% and inductive learning test accuracy closer to 77% which can be seen in figures 4.13 4.14. The stagnant phase is attained (loss = 0.49) after training of 50-75 epochs for inductive learning, whereas for the transductive process, it is attained

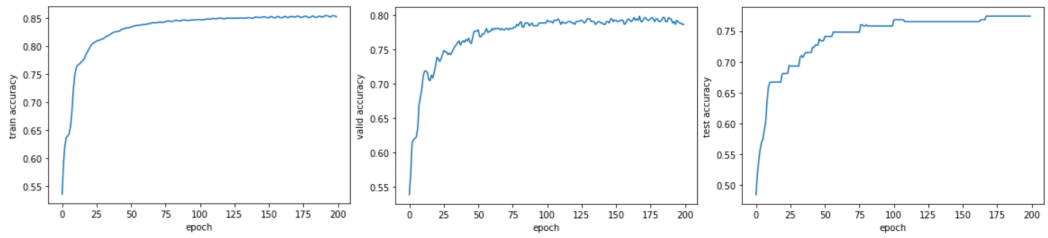


Figure 4.14: Inductive learning accuracy result- **PubMed** dataset

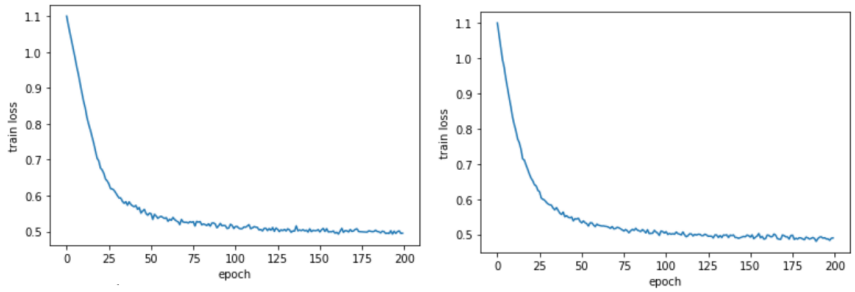


Figure 4.15: Training Loss- **PubMed** dataset (L-Transductive, R-Inductive)

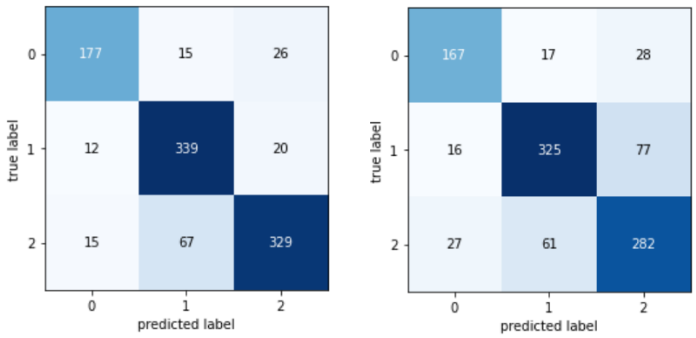


Figure 4.16: Confusion matrix result- **PubMed** dataset (L-Transductive, R-Inductive)

earlier as seen in figure 4.15. The confusion matrix in figure 4.16 shows a lesser deviation in the inductive phase compared to the results attained earlier, conveying that with lesser categories the model performs equally well for unseen data.

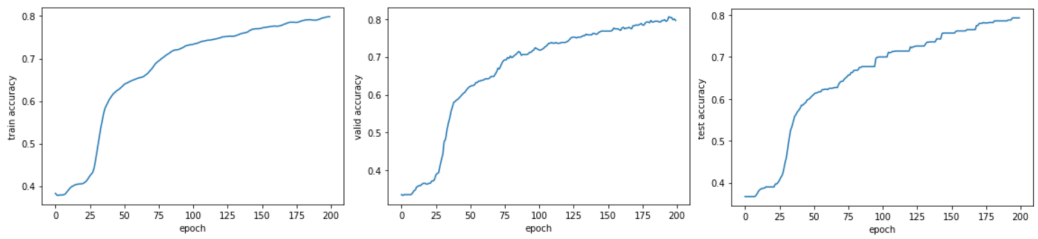


Figure 4.17: Transductive learning accuracy result- **AmazonComputers** dataset

AmazonComputers Dataset Results: AmazonComputers dataset with the increased complexity of graph, large and strongly connected nodes with higher node features, the model performed admirably in transductive learning with test accuracy closer to 79%

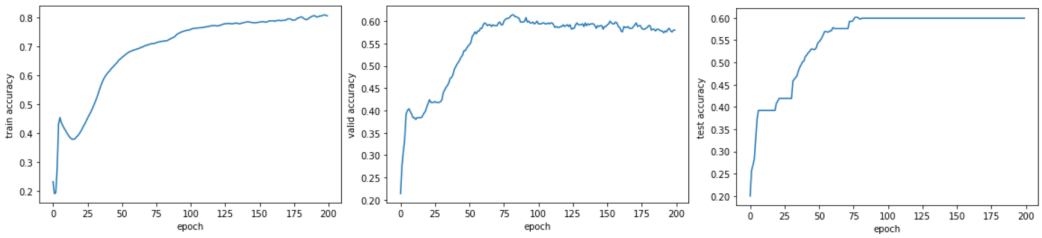


Figure 4.18: Inductive learning accuracy result- **AmazonComputers** dataset

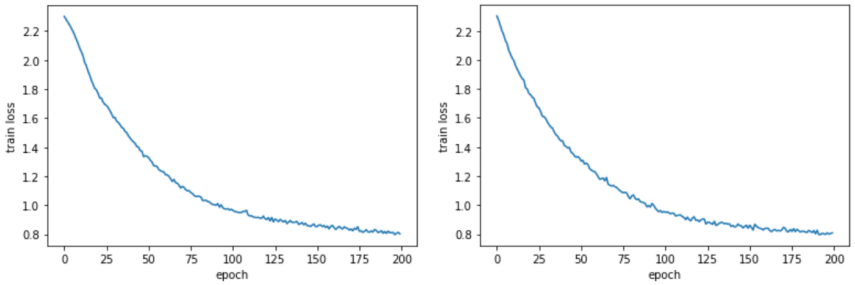


Figure 4.19: Training Loss- **AmazonComputers** dataset (L-Transductive, R-Inductive)

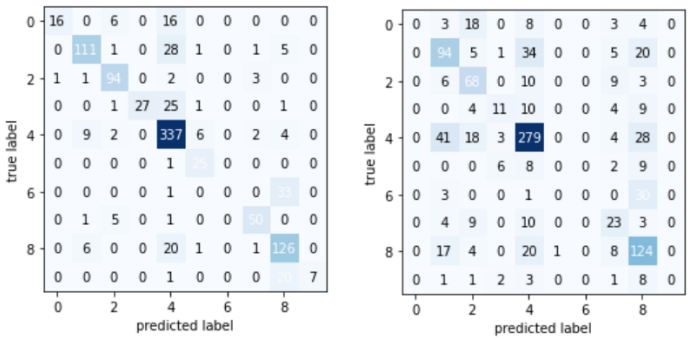


Figure 4.20: Confusion matrix - **AmazonComputers** (L-Transductive, R-Inductive)

and inductive learning test accuracy closer to 60% which can be seen in figures 4.17 4.18. The stagnant phase is attained after training of 100-125 epochs for inductive learning, whereas for the transductive process, the learning remained in progress until the last epoch. The slope of the loss in inductive learning was initially higher and flat later, as seen in figure 4.19. The confusion matrix in figure 4.20, the left image for the transductive process shows increased accuracy and lesser variation.

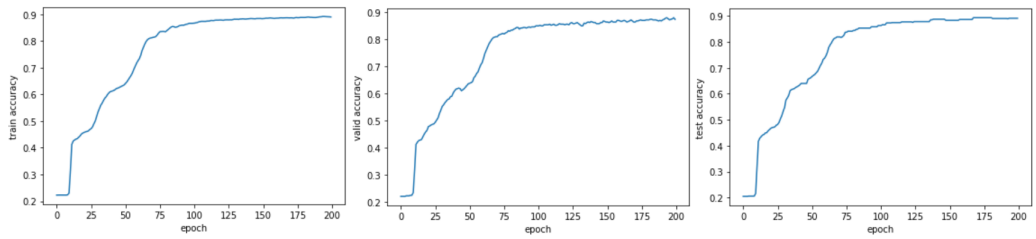


Figure 4.21: Transductive learning accuracy result- **AmazonPhoto** dataset

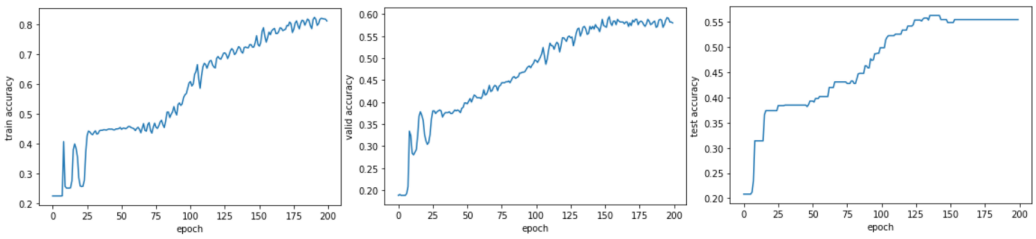


Figure 4.22: Inductive learning accuracy result- **AmazonPhoto** dataset

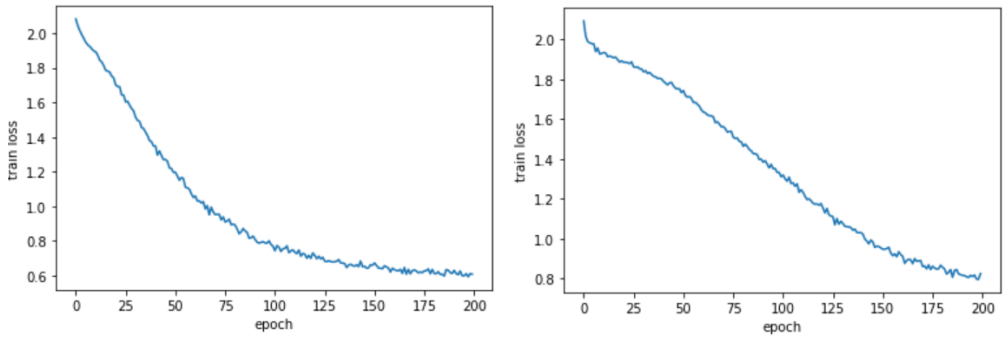


Figure 4.23: Training Loss- **AmazonPhoto** dataset (L-Transductive, R-Inductive)

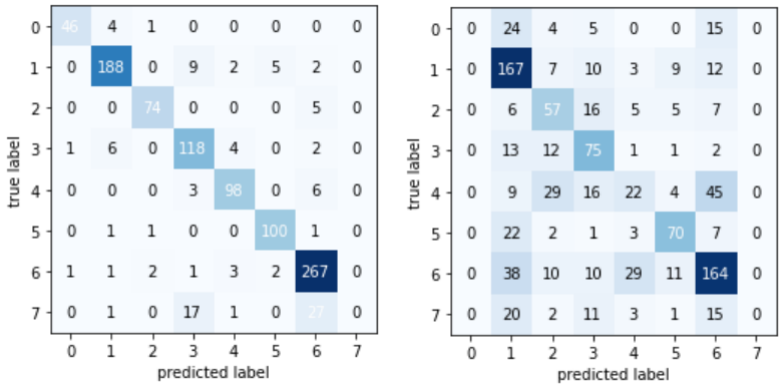


Figure 4.24: Confusion matrix - **AmazonPhoto** dataset (L-Transductive, R-Inductive)

AmazonPhoto Dataset Results: AmazonPhoto, another complex dataset with the increased complexity of graph, large number of nodes, and strongly connected graph with higher node features, the model displayed excellent performance in transductive learning with test accuracy closer to 79% and managed to attain test accuracy closer to 60% in inductive learning, which can be seen in figures 4.21 4.22. The learning task for the model took longer time for the inductive model compared to the transductive model as per figures 4.21 4.22. The slope of the loss in inductive learning was small in the initial phase and higher throughout training later, as seen in figure 4.23. The confusion matrix in figure 4.24, the left image for the transductive process displays an excellent performance of the model with minute deviation in actual and predicted values.

5 Conclusion

Despite the suggested model implementation not being as efficient as the expert models and papers referred to, the initial setup performed similarly to the state-of-the-art model's output. However, the enhancement performed on GCN together with recurrent units and Q-learning produced comparable outcomes. The results achieved for the transductive learning method are impressive compared to the inductive learning approach. Nevertheless, the inductive learning tests could manage to exceed the result for the PubMed dataset. The model performance was evaluated considering five different datasets, namely Cora, Citeseer, PubMed, AmazonComputers, and AmazonPhoto. With a limitation in technical setup and established setup with considerable parameters, the suggested model performance for Cora, Citeseer, and PubMed datasets exceeded the expectation of competent results for transductive learning. The result attained for AmazonComputers, and AmazonPhoto datasets in the transductive learning technique are also remarkable. The inductive learning method test results could not attain a commendable result which could be worked further in broader aspects in future work.

5.1 Expanding Ideas For Further Study

Although the model achieved competitive results, it can be further studied to check if the model satisfies Weisfeiler-Lehman isomorphism test. This thesis does not evaluate the model's ability to handle permutation invariance, which is one of the crucial characteristics of graph learning. This aspect can also be selected for study further. To enhance the efficacy of the model, it can also be expanded further by experimenting with the feasibility of LSTM as an aggregator for GCN and fixing the size of the neighbor loader for LSTM. Another exciting scenario to study further is applying a Q-learning algorithm for attaining appropriately hidden embedding by setting the environment to an adjacency matrix and action to select the neighbors randomly but keeping a tracker not to revisit the visited nodes, keeping the cycle of smaller diameter as one entity and updating the Q-table accordingly. This thesis work does not consider the direction of the graphs, which is also one significant trait to consider further. Although the implementation includes the edge weight to evaluate the importance of the connection between nodes, this can be enhanced further by adding a weight matrix for the critical features and node importance.

Bibliography

[1] Issam El Naqa and Martin J. Murphy. *What Is Machine Learning?*, pages 3–11. Springer International Publishing, Cham, 2015.

[2] Abdel Nasser Sharkawy. Principle of neural network and its main types: Review. *Journal of Advances in Applied & Computational Mathematics*, 7, 2020.

[3] M.W Gardner and S.R Dorling. Artificial neural networks (the multilayer perceptron)—a review of applications in the atmospheric sciences. *Atmospheric Environment*, 32(14):2627–2636, 1998.

[4] Mikael Bodén. A guide to recurrent neural networks and backpropagation. 12 2001.

[5] Xue Yan, Wang Weihan, and Miao Chang. Research on financial assets transaction prediction model based on lstm neural network. 06 2020.

[6] Chen Lei. *RNN*, pages 83–93. Springer Singapore, Singapore, 2021.

[7] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[8] Yuxi Li. Deep reinforcement learning. *CoRR*, abs/1810.06339, 2018.

[9] Benjamin Sanchez Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. *Google Research*, 2021.

[10] Claudio Stamile, Aldo Marzullo, and Enrico Deusebio. *Graph Machine Learning: Take Graph Data to the Next Level by Applying Machine Learning Techniques and Algorithms*, page 330. Packt Publishing, 2021.

[11] William L. Hamilton. Graph representation learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 14(3):1–159.

[12] Lingfei Wu, Peng Cui, Jian Pei, and Liang Zhao. *Graph Neural Networks: Foundations,Frontiers, and Applications*. Springer Singapore, 2022.

[13] Oleksandr Shchur, Maximilian Mumme, Aleksandar Bojchevski, and Stephan Günnemann. Pitfalls of graph neural network evaluation. *CoRR*, abs/1811.05868, 2018.

[14] E Francesconi, P Frasconi, M Gori, S Marinai, Q Sheng, J, G Soda, and Sperduti A. Logo recognition by recursive neural networks. page 104–117, 07 2005.

[15] Antonella Bua, Marco Gori, and Fabrizio Santini. Recursive neural networks applied to discourse representation theory. page 290–295, 08 2002.

[16] Geoffrey Hinton, Li Deng, Dong Yu, George E. Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N. Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, 2012.

[17] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. 1:57–81, 2020.

[18] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

[19] Mohammad Esmaeili and Aria Nosratinia. Semi-supervised node classification by graph convolutional networks and extracted side information, 2020.

[20] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017.

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2017.

[22] Marco Gori, Gabriele Monfardini, and Franco Scarselli. A new model for earning in raph domains. volume 2, pages 729 – 734 vol. 2, 01 2005.

[23] Luana Ruiz, Fernando Gama, and Alejandro Ribeiro. Gated graph recurrent neural networks. *IEEE Transactions on Signal Processing*, 68:6303–6318, 2020.

[24] Song Weiping, Duan Zhijian, Yang Ziqing, Zhu Hao, Zhang Ming, and Tang Jian. Explainable knowledge graph-based recommendation via deep reinforcement learning. *CoRR*, abs/1906.09506, 2019.

[25] Xiong Wenhan, Hoang Thien, and Yang Wang William. Deeppath: A reinforcement learning method for knowledge graph reasoning. *CoRR*, abs/1707.06690, 2017.

[26] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016. `http://www.deeplearningbook.org`.

[27] Fengsheng Zeng and Qin Wang. Intelligent recommendation algorithm combining rnn and knowledge graph. page 11, 08 2022.

[28] Roohollah Amiri, Hani Mehrpouyan, Lex Fridman, Ranjan Mallik, Arumugam Nallanathan, and David Matolak. A machine learning approach for power allocation in hetnets considering qos. 03 2018.

[29] Zhiyuan Liu and Jie Zhou. *Introduction to Graph Neural Networks*. Morgan Claypool, 2020.

[30] Gary Chartrand and Ping Zhang. *A First Course in Graph Theory*. Dover books on mathematics. Dover Publications, 2012.

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, im February 2023