ETH
Swiss Federal Institute of Technology Zurich

Seminar for
Statistics

Department of Mathematics

Master Thesis Summer 2019

Leslie O'Bray

Learning Vector Representations of Graphs Using Recurrent Neural Network Autoencoders

Submission Date: September 6, 2019

Co-Adviser: Prof. Dr. Karsten Borgwardt
Co-Adviser: Dr. Bastian Rieck
Adviser: Prof. Dr. Marloes Maathuis

To ETH Zürich and the Swiss Federation, for providing the opportunity to get a great education at an affordable price.

Abstract

Increasingly, more and more data is stored as graph-structured data, across a wide range of fields such as bioinformatics, social network analysis, telecommunications, and others. Traditional classification methods are often unavailable for graph-structured data, since many operate on vector data, and it isn't immediately clear how to best compress the rich information in the adjacency matrix into a vector, particularly when the ordering of the nodes in the adjacency matrix is random. Nevertheless, finding a way to compare and classify graphs would be useful in many domains, and is an ongoing area of research. This thesis investigates a new method that proposes using recurrent neural network autoencoders to learn a vector representation of graphs, to be used for comparison and classification, which we evaluate on bioinformatics benchmark datasets. We write our own implementation of the proposed method to compare our results with the original authors' findings, as well as compare it to other related state-of-the-art methods. Then, we conduct a few experiments to further understand the method and try to improve its results.

We found the method yielded similar results to what the authors claimed, but also that the method gives largely comparable performance to the existing sequence-based methods that learn node embeddings. Through our experiments we learned that the specific classifier used to classify the learned vector representation is interchangeable, which enabled us to reduce the computational complexity of the method to be linear, rather than quadratic, in the number of graphs. Additionally, we found that performance can be improved by updating the graph embedding function used to summarize all the vectors representing a single graph. Finally, we tested simplifying the approach by performing the classification directly in the neural network. While this yielded lackluster results, it provided useful insight into what network architectures would be better suited for such a problem, providing direction for future work in the area.

Contents

Notation xiii

1 Introduction 1

2 Background & Related Work 3
2.1 Background . . . 3
2.1.1 Graphs . . . 3
2.1.2 Weisfeiler-Lehman Algorithm . . . 3
2.1.3 Recurrent Neural Networks . . . 5
2.1.4 Autoencoders . . . 9
2.1.5 Support Vector Machines . . . 9
2.2 Related Work . . . 10
2.2.1 Feature Extraction . . . 10
2.2.2 Kernel Methods . . . 10
2.2.3 Machine Learning Methods . . . 11
2.3 Summary . . . 13

3 Datasets 15
3.1 Datasets Description . . . 15

4 Methods 17
4.1 Preparing the Input for the Network . . . 17
4.1.1 Sequence Generation . . . 17
4.1.2 Vertex Embedding . . . 18
4.2 Building the Network Architecture . . . 19
4.2.1 Sequence-to-Sequence Autoencoder . . . 19
4.2.2 Models . . . 20
4.2.3 Loss Functions . . . 22
4.3 Defining the Graph Embedding Function & Performing Classification . . . 22
4.3.1 Graph Embedding Function . . . 22
4.3.2 Classification . . . 23
4.4 Summary . . . 23

5 Implementation Comparison 25
5.1 Implementation Notes & Parameter Choices . . . 25
5.1.1 Sequence Generation . . . 26
5.1.2 Network Architecture . . . 26
5.1.3 Classification using SVM . . . 27

5.2 Performance Comparison . . . 27
5.2.1 Model 1: S2S-AE . . . 29
5.2.2 Model 2: S2S-AE-PP . . . 30
5.2.3 Model 3: S2S-AE-PP-WL1,2 . . . 30
5.2.4 Model 4: S2S-N2N-PP . . . 31
5.3 Observations from the Implementation . . . 32
5.3.1 Lower Performance with Random Walks . . . 33
5.3.2 Performance without Training . . . 35
5.4 Comparison to Other Sequence-based Methods . . . 35
5.4.1 DeepWalk . . . 36
5.4.2 node2vec . . . 37
5.4.3 Comparison to the Main Method . . . 39
5.5 Summary . . . 40

6 Extensions & Discussion 43
6.1 Experiment 1: More Informative Vector Representation . . . 43
6.1.1 Results . . . 45
6.2 Experiment 2: Using a Neural Network in Place of an SVM . . . 46
6.2.1 Results . . . 48
6.3 Experiment 3: Using a Neural Network to Predict the Class Directly . . . 48
6.3.1 Results . . . 49
6.4 Summary . . . 49

7 Conclusion 51
7.1 Future Work . . . 52

Bilbiography 53

A Source Code Description 57
A.1 graph-rnn-ae File Descriptions . . . 57
A.2 Instructions to Run . . . 58

List of Figures

2.1 LSTM Memory Cell 7

4.1 LSTM Autoencoder 20

5.1 Training and Validation Losses Over Epochs 28
5.2 Model 1 Performance by Weisfeiler-Lehman Iteration 29
5.3 Model 2 Performance by Weisfeiler-Lehman Iteration 31
5.4 Model 3 Performance by Weisfeiler-Lehman Iteration 32
5.5 Effect of Average Node Size on the Number of Unique Nodes in a Random Walk 33
5.6 Classification Accuracy by Random Walk Length 34
5.7 DeepWalk Performance by Weisfeiler-Lehman Iteration 37
5.8 node2vec Classification Performance by Weisfeiler-Lehman Iteration 38
5.9 Performance Comparison of Sequence-based Classification Methods 40

6.1 Experiment 1: Classification Accuracy of the SVM vs. Random Forest 44
6.2 Experiment 1: Classification Accuracy of More Informative Vector Representations 45
6.3 Experiment 2 Classification Accuracy Performance over Epochs 47

List of Tables

3.1 Summary Statistics of the Datasets 15

4.1 Schema Overview of the Model Notation 19

5.1 Parameter Summary 25
5.2 Sequence Frequency and Repetition in the Training Data 27
5.3 Search Space of the Hyperparameters Used in the SVM 28
5.4 Model 1: S2S-AE Classification Accuracy 29
5.5 Model 2: S2S-AE-PP Classification Accuracy 30
5.6 Model 3: S2S-AE-PP-WL1,2: Classification Accuracy 31
5.7 Model 4: S2S-N2N-PP Classification Accuracy 32
5.8 Classification Accuracy with No Network Training 35
5.9 Classification Accuracy Using DeepWalk 36
5.10 Classification Accuracy Using node2vec 38

6.1 Experiment 1 Classification Accuracy 46
6.2 Experiment 2 Classification Accuracy 47
6.3 Experiment 3 Classification Accuracy 48

Notation

AE	Autoencoder
BFS	Breadth-First Search
BPTT	Backpropagation Through Time
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
RW	Random Walk
SP	Shortest Path
SVM	Support Vector Machine
WL0	Weisfeiler-Lehman Node Labeling at Iteration 0
WL1	Weisfeiler-Lehman Node Labeling at Iteration 1
WL2	Weisfeiler-Lehman Node Labeling at Iteration 2

I Introduction

In recent years, there has been an increased focus and interest around graph-structured data. Many fields naturally lend themselves to representing their data as a graph, such as in bioinformatics, social networks, telecommunication networks, among others. The format provides an effective way of encapsulating important aspects of the data, capturing information not only about an entity itself, but also about the relationship between those different entities. For example, when working with chemical compounds, we care not only about the atoms in the molecule, but also about the bonds between the different atoms. Both the atomic element and the bonds carry important information. Similarly, proteins are also well suited to be stored as a graph based on the nature of their spatial structure, and the importance of connections between different components. Storing this data as a graph enables the data to encode information about these connections, in a way that traditional vector data does not.

As with traditional ways of structuring data, e.g. vector data, one can be interested to answer various questions, such as, given two objects, how similar are they? Are they identical? Given a data point, can we classify it as one thing or another? Just as we want to answer these questions with vector data, so do we with graphs. For example, given a new chemical compound, we may wish to classify whether the chemical compound is carcinogenic or not. Or given a protein, one may wish to determine what type of protein it is. This can be interpreted as either a graph similarity problem, or a graph classification problem.

From a graph-theoretic perspective, these are hard questions to answer. Although a graph and its corresponding adjacency matrix provide a rich way to encode information, it leaves the user with a problem of how to work with that data. Its inherent high dimensionality make methods often computationally infeasible, and its structure limits the use of established vector methods. For example, simply determining whether two graphs are identical, which is straightforward to solve with vectors, is a very challenging question with graphs. This problem, known as the graph isomorphism problem, has no known solution in polynomial time.

While the graph isomorphism problem answers the question of whether two graphs are identical, it is often not as useful of a question as answering how similar two graphs are. After all, two graphs can be identical except for a single edge, and according to the graph isomorphism test they are not isomorphic. It is often more helpful to quantify the degree of similarity between the objects, instead of using a binary test for identicality.

However, even when we are interested in assessing similarity, we still face the challenge of how to represent the graph to make the comparison, and how to actually measure this similarity in a way that is computationally reasonable. There is a field of research looking into how one can represent a graph such that these problems and questions are tractable. While there are various approaches to address this problem, many are concerned with finding a vector representation of a graph, since we have a plethora of methods to compute distances (or similarities) of vectors, classifying vector data is a well established area, and much of the machine learning work with

vector data. Thus, finding a vector representation of the graph would make a myriad of methods immediately available to solve these questions.

This thesis sits directly within the field of learning representations of graphs, and more generally, of graph classification. It investigates a recently proposed approach that leverages advances in machine learning to solve this problem. It is grounded upon the work of Taheri et al., who suggested decomposing a graph into a set of sequences, and then using a recurrent neural network autoencoder to learn a vector representation of a graph [29]. The autoencoder reduces the dimensionality of the input in a way that best captures as much information as possible, and provides a fixed size vector representing our input. Once we have this representation, we can perform any kind of similarity calculation or classification task we are interested in.

We will begin in the Background & Related Works chapter with an explanation of relevant concepts and methods necessary to follow the contents of this thesis. We will provide a survey of the different high-level approaches that other researchers have used to answer this question, highlighting a few relevant examples to give a fuller picture of the field.

Next, we will transition to describe the mechanics of how the approach investigated in this thesis works in the Methods chapter. We will provide a detailed explanation of what the original authors proposed, and the specifics behind their approach. In the following chapter on Implementation Comparison, we will discuss the results of our own implementation of the method, and compare it to the results stated by the authors. We will close that chapter with a comparison of this method's performance to the performance of related approaches that can be considered a standard comparison for sequence-based graph classification.

After establishing the performance of the original method, and comparing its performance to a few other methods, we will then discuss our efforts to extend the original method in the Experiments & Discussion chapter. We consider three hypotheses we want to test, and run an experiment on each to determine the answer. Finally, we close with a summary of the main findings of this thesis, and discuss potential directions for future work in this area in the Conclusion.

2 Background & Related Work

This chapter will provide an overview of the important concepts and relevant related methods pertinent to this thesis. We will begin by reviewing the fundamentals of the major topics discussed in this thesis, and then provide an overview of other approaches that seek to compare and classify graphs.

2.1 Background

The main method investigated in this thesis touches upon many different areas. We will begin by giving a short primer on graph theory, including the well-known Weisfeiler-Lehman algorithm, which although initially intended as a graph isomorphism test, has been often repurposed as a node relabeling mechanism. Then we will cover the main concepts of recurrent neural networks (RNNs) and the popular variant Long-Short Term Memory Networks (LSTMs). We will introduce the specific neural network structure of autoencoders, and finally, provide an overview of classification using Support Vector Machines.

2.1.1 Graphs

A *graph* G is a pair of vertices V and edges E, such that G = (V, E), $|V| = n$, and $|E| = m$. Two vertices i and j are *adjacent* if there is an edge between them, that is, $e_{ij} \in E$. These relationships are represented in an $n \times n$ adjacency matrix A, where:

$$A_{ij} = \begin{cases} 1, & \text{if } e_{ij} \in E \\ 0, & \text{otherwise.} \end{cases}$$

The neighborhood of a node v is the set of nodes with an edge directly to v: $N(v) = \{u \in V \mid e_{uv} \in E\}$. We say a graph is *undirected* if $A_{ij} = A_{ji}$, meaning the adjacency matrix is symmetric, which implies that an edge between two nodes does not convey any directional information. A *directed* graph, on the other hand, does encode directional information, and thus A_{ij} does not give any information about A_{ji}. In this thesis we will work with with *labeled* graphs, where we refer specifically to labels on the set of nodes, though the methods can easily be extended to *unlabeled* graphs. A labeled graph in this context implies that each node has a label, whereas an unlabeled graph each node label is the same. We will work with graphs where there is no node correspondence, meaning the node position in the adjacency matrix is arbitrary.

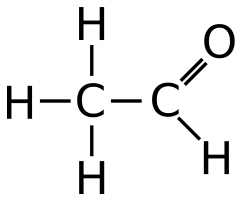
2.1.2 Weisfeiler-Lehman Algorithm

When working with graphs, it is often useful to perform a vertex labeling (in the case of an unlabeled graph) or a vertex relabeling (in the case of a labeled graph) in order to encode more informa-

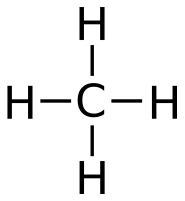
tion in the node label itself. A common choice for this is using the Weisfeiler-Lehman algorithm, which is compatible with both labeled and unlabeled graphs. Although this was initially intended as a graph isomorphism test, it later become a popular way to define similarity between graphs. This is in fact more useful; as mentioned before, the graph isomorphism test is very rigid, and a single additional edge makes two otherwise identical graphs non-isomorphic. A more useful measure would be to quantify how similar two graphs are, which this algorithm has later been used to do. It can also be used as a mechanism to relabel nodes based on the multiset of its own node label and the sorted list of the labels of its neighbors. This process can be repeated for a given number of iterations h, using the labels assigned in the previous iteration to determine the multiset for the current iteration (up until a certain point, as the algorithm will eventually stabilize).

For illustration, we consider a dataset with just two graphs, acetaldehyde and methane. Methane is not suspected to be carcinogenic, whereas acetaldehyde is considered to be a probable carcinogen [24]. These two chemical compounds have the following structures:

Acetaldehyde

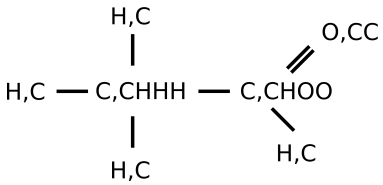


Methane

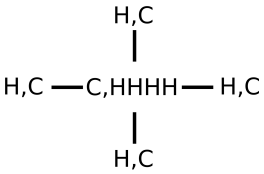


In the first step of the Weisfeiler-Lehman algorithm, we replace the node labels with the multiset of its label plus the sorted list of its neighbors' labels.

Acetaldehyde



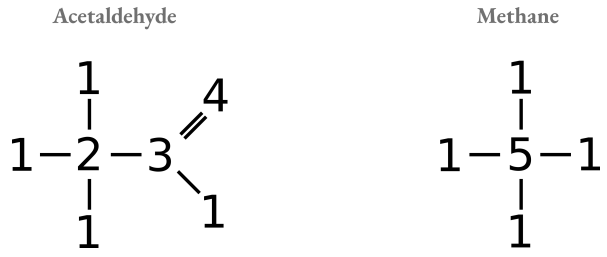
Methane



Then all unique values are hashed to a new label:

Multiset	Hash Label
H,C	1
C,CHHH	2
C,CHOO	3
O,CC	4
C,HHHH	5

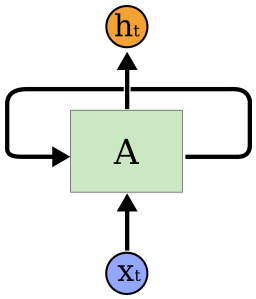
and the node labels in the graph would be updated to their respective hashing value.



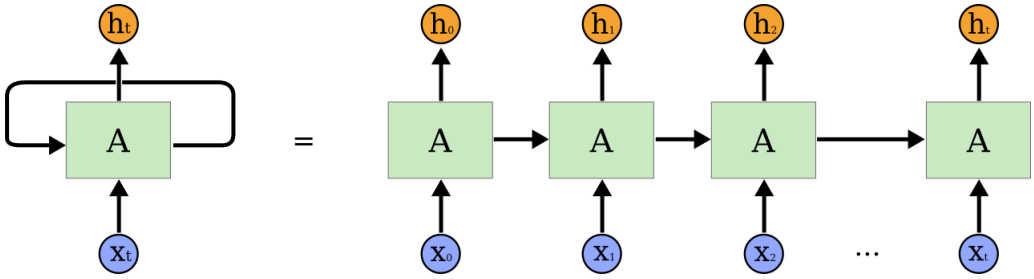
This is the result of the first iteration of the Weisfeiler-Lehman algorithm (WL1), and it is possible to see that the node labels now represent information about the node itself as well as its neighbors. This procedure can be repeated for higher iterations. For a given iteration h, the algorithm captures information about the nodes h hops away from the given node. As mentioned, this process is also suitable to be used for graphs without node labels. In this case, at the starting iteration every node is given the same label, and the process is repeated just like above. It is worth noting that the first iteration of the algorithm on unlabeled graphs (where we assume the 0^{th} iteration is original node labels) amounts to the degree function of the graph.

2.1.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are an extension of the neural network framework to handle sequential data. Whereas traditional feed-forward networks require the input to be fixed size, independent observations, RNNs are designed to incorporate variable-length sequences where the time dependency between the input is taken into account. It is through this perspective that RNNs are often depicted as a neural network with a cycle that feeds back into itself, which we reproduce here [22].



Although the network looks cyclical, it can also be understood more simply unrolled as a very deep feed-forward network. Each observation x that is given as a single data point to the network is a sequence of observations $x_1, ..., x_t$ where x_i represents the observation from time step i. As each x_i is given to the network, the network receives as input both x_i as well as what was learned up until that point, passed from the previous time step, or layer, h_{i-1}. In this way, it is able to receive both the new information x_i but also factor in all the previous information gained from $x_1, ..., x_{i-1}$.



Although an RNN can be understood quite similarly to a traditional feed-forward network, a major difference is that the weights, for example between the input-to-hidden layer, hidden-to-hidden layers, and hidden-to-output layer, are shared between time steps [10, p. 20]. This aspect is particularly important, as it allows the network to handle arbitrarily sized sequences, rather than requiring all input sequences to be equal length.

There are several different high-level architectures for RNNs, which each serve different purposes. RNNs can be *many-to-one*, taking a sequence as an input and predicting a single thing as the output. For example, given a song, one may want to classify it into a particular genre. On the other hand, there is also *one-to-many*, where there is a single point as input and the output is a sequence. One could take the genre as the input, and then generate a song sequence as an output. Finally, there is the *many-to-many* structure, which takes a sequence as input and also predicts a sequence as output. A classic use case for many-to-many is in machine translation, where the input could be a sentence in English, and the output is the translation of that sentence into German. We will work specifically with this last option, *many-to-many*, where we will give one sequence from the graph as the input to the network, and try later to reconstruct that sequence. The specifics of this modified setup will be detailed further in Section 2.1.4. In discussing RNNs going forward, we will use the terminology from Graves [10], and make minor updates to Graves' notation in order to align with the notation observed in Taheri et al. paper.

LSTMs

RNNs have historically suffered when trying to learn long-term dependencies in sequences. This is commonly referred to as the vanishing or exploding gradient problem, which is due to the fact that the weights between time steps are shared [10, p. 20]. During backpropagation, one ends up multiplying the weight matrices by itself many times. Depending on the values of the eigenvalues, this leads to either an exploding gradient (if the eigenvalues of the weight matrix are greater than 1) or a vanishing gradient (if the eigenvalues of the weight matrix are less than 1). A major breakthrough in RNN research came with the introduction of Long-Short Term Memory Networks (LSTMs) [15], which directly addresses the problem of the vanishing gradient (an exploding gradient can be easily resolved by clipping the gradient if it exceeds a threshold value). LSTMs introduce a memory cell into the network, with corresponding gates to determine whether information should be added to the memory cell, as well as whether stored information should be forgotten. The memory cell is then able to better learn long-term dependencies in the data. We work with a further modified version of the LSTM called an LSTM with Peephole Connections,

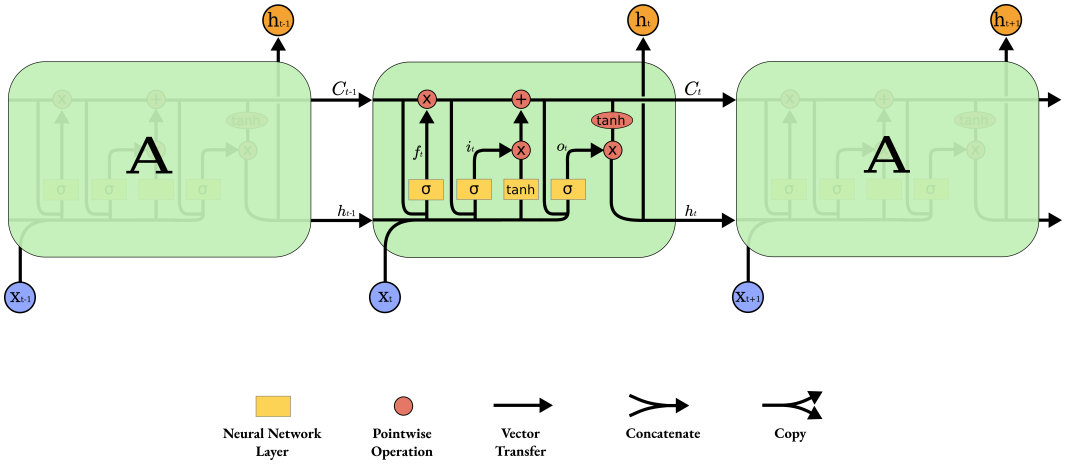


Figure 2.1: A reproduction of a visualization from Olah [22], depicting the various operations performed inside an LSTM Memory Cell with Peephole Connections.

introduced by Gers et al. [9], which allows the various gates to "peep" into the state of the cell, also known as the cell state, by using the cell state as an additional input into the equation. At each time step, a series of computations are done within the LSTM memory cell in order to determine which information should be added to the memory cell state and which should be forgotten. The resulting cell state C_t and cell output h_t are then passed to the next time step.

The input, forget, and output gates (i_t, f_t, and o_t respectively), as well as the cell state C_t and the cell output h_t are computed accordingly, where $\odot$ signifies element-wise multiplication, and the weight matrices W, U, and K connect the input, cell output, and cell state respectively to the gates [29]:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + K_i C_{t-1} + b_i) \tag{2.1}$$
$$f_t = \sigma(W_f x_t + U_f h_{t-1} + K_f C_{t-1} + b_f) \tag{2.2}$$
$$o_t = \sigma(W_o x_t + U_o h_{t-1} + K_o C_{t-1} + b_o) \tag{2.3}$$
$$C_t = f_t \odot C_{t-1} + i_t \odot \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.4}$$
$$h_t = o_t \odot \tanh(C_t). \tag{2.5}$$

Backpropagation Through Time (BPTT)

As mentioned above, it is helpful to unroll the RNN when visualizing the network. This becomes particularly useful once one begins to consider how backpropagation would work in an RNN. The network parameters are trained via gradient descent, using a variation of traditional backpropagation called Backpropagation Through Time (BPTT).

With the network unrolled along the time steps $x_1, \ldots, x_t$, it is easier to understand this simply as backpropagation on a computational graph. As on any computational graph, one needs to compute the gradient for a given parameter by using the chain rule for partial derivatives along

any path that connects the parameter of interest to the loss function, and then sum over all paths. BPTT is just an application of this, with the nuance that the loss function depends on a given unit in the network not only through its direct path the output layer, but also through the hidden unit's influence on the hidden layer in the next time step [10, p. 20].

We now introduce some adapted notation and equations from Graves to give an example [10, p. 36-38]. For a given component a at time point t in the network, and loss $\mathcal{L}$, we will use the following shorthand:

$$\delta_{a_t} = \frac{\partial \mathcal{L}}{\partial a_t}. \tag{2.6}$$

This will become useful to simplify some of the intermediate steps of the derivative. For illustration, suppose we are interested in computing the gradient of a given weight in the input gate weight matrix W_i, which we will simply name w_{ij} to improve readability. Before looking at the specific weights we will want to update, we consider the more general question of finding the derivative of a given part of the network. We will first consider the linear function in the input gate before its sigmoid activation, which we will denote in_{i_t}. As a reminder, we have the following equations:

$$in_{i_t} = W_i x_t + U_i h_{t-1} + K_i C_{t-1} + b_i \tag{2.7}$$
$$i_t = \sigma(W_i x_t + U_i h_{t-1} + K_i C_{t-1} + b_i) = \sigma(in_{i_t}). \tag{2.8}$$

If we are interested to find $\frac{\partial \mathcal{L}}{\partial in_{i_t}}$, we need to consider all paths linking in_{i_t} to our output and our loss:

$$\delta_{in_{i_t}} = \frac{\partial \mathcal{L}}{\partial in_{i_t}} = \frac{\partial \mathcal{L}}{\partial C_t} \frac{\partial C_t}{\partial i_t} \frac{\partial i_t}{\partial in_{i_t}}. \tag{2.9}$$

From our previous equations for the forward pass, we can solve each of these components, incorporating both their effect at time steps t and $t+1$:

$$\frac{\partial \mathcal{L}}{\partial C_t} = \frac{\partial \mathcal{L}}{\partial h_t} \frac{\partial h_t}{\partial C_t} + \frac{\partial \mathcal{L}}{\partial C_{t+1}} \frac{\partial C_{t+1}}{\partial C_t} + \frac{\partial \mathcal{L}}{\partial i_{t+1}} \frac{\partial i_{t+1}}{\partial C_t} + \frac{\partial \mathcal{L}}{\partial f_{t+1}} \frac{\partial f_{t+1}}{\partial C_t} + \frac{\partial \mathcal{L}}{\partial o_{t+1}} \frac{\partial o_{t+1}}{\partial C_t} \tag{2.10}$$
$$\frac{\partial C_t}{\partial i_t} = \tanh(W_c x_t + U_c h_{t-1} + b_c) \tag{2.11}$$
$$\frac{\partial i_t}{\partial in_{i_t}} = \sigma'(in_{i_t}). \tag{2.12}$$

To find the gradient of our weight of interest w_{ij} at a given time point t, we can use the result (2.9) and multiply it by $\frac{\partial in_{i_t}}{\partial w_{ij}}$. In order to compute the full gradient, we will sum over the computed derivative for each time step, beginning with $t = T$ and going backwards until $t = 1$:

$$\sum_{t=1}^{t=T} \delta_{in_{i_t}} \frac{\partial in_{i_t}}{\partial w_{ij_t}}. \tag{2.13}$$

2.1.4 Autoencoders

We also consider the special neural network setup of autoencoders. Autoencoders are a way of performing dimensionality reduction with neural networks, a way of finding a lower dimensional representation of the input data that captures as much information as possible about the data it is representing. It can be understood as a non-linear version of Principal Component Analysis. An autoencoder takes a given input, and in the first portion of the network encodes this input to an intermediate representation z. The second portion of the network then takes the intermediate representation and tries to reconstruct the input as its output. There are two things to note here: first, the input and the output should be the same. Second, in contrast to more traditional neural networks, the output of the network is not the goal itself. The loss function of an autoencoder is based on the reconstruction error, and through the training of the network, the intermediate representation z is optimized to best reconstruct the original input. At the end of training, we extract z and use it for clustering or classification, and network itself is then no longer of use.

2.1.5 Support Vector Machines

Finally, we will briefly discuss the popular classification method Support Vector Machines (SVM). Given the long history and ubiquity of this method we proceed without a detailed investigation here.

The Support Vector Machine builds upon the Perceptron by finding a classifier with the maximal possible margin between itself and the data points. Although the classic SVM does not permit any missclassifications (and thus will not converge if the data is not linearly separable, or can overfit the data if there are outliers), it is possible to use a variation that tolerates some degree of missclassification, which is specified by the user. We thus are trying to solve the following optimization problem [13, p.420, §12.8]:

$$\min_{\beta,\beta_0} \frac{1}{2}||\beta||^2 + C\sum_{i=1}^{N} \xi_i \tag{2.14}$$

$$s.t.\ \xi_i \geq 0,\ y_i(x_i^T\beta + \beta_0) \geq 1 - \xi_i\ \forall i, \tag{2.15}$$

where ξ_i denotes missclassifications and C is a regularization parameter, which impacts how many missclassifications are tolerated. Once this is converted to the Lagrangian dual objective function, we are optimizing [13, p.420, §12.13]:

$$L_D = \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{i'=1}^{N} \alpha_i\alpha_{i'}y_iy_{i'}x_i^Tx_{i'}. \tag{2.16}$$

In cases where the data is not linearly separable, it is possible to use the kernel trick and transform the data using a valid kernel, in the hopes the data may be linearly separable in the transformed space. We observe that the optimization function relies on the data x only through an inner product, thus we are able to replace the inner product with any positive semi-definite kernel. A common kernel choice is the *radial basis function* [13, p. 424, §12.22], defined as:

$$K(x, x') = \exp\left(-\gamma||x - x'||_2^2\right). \tag{2.17}$$

2.2 Related Work

Now that we have a understanding of the concepts underpinning the approach we are using, we will introduce other approaches to solving the general problem of comparing and classifying graphs. In particular, we will discuss three high-level methods that have been popular in recent years: feature extraction, graph kernels, and machine learning methods.

2.2.1 Feature Extraction

One popular method for comparing and classifying graphs is *feature extraction*, which represents the graph as a vector made up of different graph-theoretical features of the graph. For example, Cai et al. discovered that describing a graph using a histogram or empirical distribution function of five features for each node v: the degree of v, the minimum, mean, maximum, and standard deviation of the degree from its set of neighbors, already performs very well on the benchmark datasets [4]. It achieves competitive results on the baseline datasets for unattributed/unlabeled graphs, and though not competitive on attributed/labeled graphs, still beats several other methods.

Feature extraction has two main advantages. First, the result is a fixed size vector, which can then be easily used to compute distances/similarities or perform classification. Second, because it compresses the graph into a vector, it is relatively computationally efficient: the method described above can be done in linear time. However, this approach tends to lose a lot of information about the graph, and it relies on manual curation of relevant features, which may limit what can be learned.

2.2.2 Kernel Methods

Another class of methods which has been quite successful are *kernel methods*. Kernels can be defined on structured objects, like graphs, by defining a kernel over some sub-structure and then aggregating this over the whole graph. This is often achieved using the $\mathcal{R}$-Convolution Kernel [14]. Given a structured object X, which can be decomposed into substructure parts $x_1, ..., x_d$, we can define a relation R using the original notation of Haussler where $R(x_1, ..., x_d, x) = 1$ (or more succinctly, $R(\vec{x}, x) = 1$, when we define $\vec{x} = x_1, ..., x_d$) if and only if $x_1, ..., x_d$ are parts of X. The inverse relation $R^{-1}(x)$ is then comprised of all objects that are in relation to x: $R^{-1}(x) = \{\vec{x} : R(\vec{x}, x)\}$. The $\mathcal{R}$-Convolution Kernel is defined as:

$$K_{\mathcal{R}}(x,y) = \sum_{\vec{x}\in R^{-1}(x),\vec{y}\in R^{-1}(y)} \prod_{d=1}^{D} K_d(x_d, y_d). \tag{2.18}$$

More intuitively, for our application it can be understood as follows: given two graphs G_1 and G_2, a valid base kernel $k_{base}(x,y)$, and some substructure s, it aggregates over all substructure comparisons in G_1 and G_2:

$$K_{\mathcal{R}}(G_1,G_2) = \sum_{s\in S, s'\in S'} k_{base}(s,s'). \tag{2.19}$$

Among popular kernel methods for graphs are the Shortest Path Kernel, Random Walk Kernel, Graphlet Kernel, and the Weisfeiler-Lehman Subtree Kernel. The Shortest Path Kernel [1] transforms G_1 and G_2 into graphs that represent the shortest paths between all pairs of nodes in a given graph, then counts the matching number of shortest paths in G_1 and G_2. Similarly, the Random Walk Kernel [12] computes the matching number of random walks in two graphs. The Graphlet Kernel [27] counts all matching subgraphs between G_1 and G_2 for a specified subgraph of size k. Lastly, the Weisfeiler-Lehman Subtree Kernel [26] considers the number of matching labels assigned by the Weisfeiler-Lehman algorithm up to a given iteration h.

Kernel methods have had strong performance on graph classification tasks. However, they suffer from computational complexity, as it is quadratic in the number of graphs ($\mathcal{O}(N^2)$), and often polynomial in the number of nodes for a given pair of graphs. Additionally, classification using graphs kernels are restricted to methods that can work directly with kernelized data, like SVM, since we generally do not get a vector representation for each graph. The Weisfeiler-Lehman Subtree Kernel is a notable exception to this, where we can in fact return a vector representation of each graph.

2.2.3 Machine Learning Methods

Finally, there has been a recent effort to leverage the powerful gains made in machine learning in recent years and apply them to graph-structured data. One of the main challenges here is that most machine learning methods work with vector data, or otherwise require node correspondence to be able to effectively leverage the adjacency matrices. While some machine learning approaches, like the main approach considered in this thesis, try to first identify a vector representation of a graph that can be used later for clustering or classification, others use machine learning to make the classification directly. For example, Jin et al. use an RNN to encode each graph as a fixed size vector, then make the classification in the same network [16]. They begin by decomposing a graph into a set of sequences generated by a modified version of random walks, then feed in the sequences to the RNN to compress each walk into a fixed sized vector, and then aggregate the vectors for a given graph within the network itself, in order to then classify at the graph level.

There are several other papers that approach the problem entirely differently. GRAPH2VEC [20] draws inspiration from natural language processing and makes the connection between graph em-

beddings and document embeddings. Rather than finding an embedding for individual features (e.g. nodes) or substructures of the graph, and then aggregating them, GRAPH2VEC aims to use unsupervised learning in order to find a cohesive single vector representation of the graph as a whole, independent from context or domain, or even class labels. It does so by adapting the popular framework DOC2VEC to the graph domain. Whereas DOC2VEC uses the words that appear in a document to comprise the "context" of that document, GRAPH2VEC uses rooted subgraphs around each node in the graph. Like DOC2VEC, GRAPH2VEC then uses the skipgram architecture to learn a vector representation of each training point, which in this case is an entire graph [19].

PATCHY-SAN [21] is a generalization of convolutional neural networks (CNN) for arbitrary sized graphs, and works with directed and undirected graphs, and (continuous or discrete) attributed or unattributed nodes and edges. It solves the node correspondence problem by assigning its own ranking scheme to nodes in a graph according to some measure (e.g. betweenness centrality, normalized node degree, etc.), assuring that the representation of each graph, which is the input to the network, has a comparable ordering across different graphs. Principally, the idea is to select a fixed number of nodes w based on some ranking measure to represent each graph. For each selected node, one chooses the top k vertices in the neighborhood of that node (beginning with 1-hop neighbors, and continuing to 2-hop, 3-hop, and so on until k vertices are selected). Once k nodes have been added, they are then ordered first according to edge distance from the source node, and then according to the specified ranking function. The goal is for similar substructures within graphs to surface in this process. The input dimensions for the neural network then becomes (w, k, a_v) for a graph which has node attributes of dimension a_v (or reshaped more compactly (wk, a_v), and (w, k, k, a_e) for graphs with node and edge attributes (or (wk^2, a_e)). They applied a 1-dimensional convolutional layer with a stride size equal to k in the former, and k^2 in the latter. PATCHY-SAN then operates similarly to a CNN, where it moves a filter over different local areas of the input to try and extract relevant features from the data.

Another approach is by creating a neural graph fingerprint for each graph [7], as a differentiable extension of the fixed fingerprint method such as the circular fingerprint computed using the Extended Connectivity Fingerprint (ECPF). ECPF compresses a graph into a fixed size vector representation, however, this vector can be extremely large and hard to interpret. The neural graph fingerprints method is similar in that it considers a radius of size r, then represents the individual atoms in a molecule as the concatenation of the atom label and its neighbors, for increasing radii. The set is then hashed to an index and represented in a feature vector, which is aggregated over all atoms to represent a single molecule (or graph), reminiscent of the relabeling procedure in Weisfeiler-Lehman. The method proposed by Duvenaud et al. adapts the algorithm to use a layer of a neural network to compute the vector representation of the graph [7]. A smaller vector representation can be achieved for a given graph, and it adds the advantage that it can capture similarities between substructures. This improves upon the older circular fingerprints, which considers any deviation in a substructure to be unique and results in a different hashed label. Interesting, relabeling using Weisfeiler-Lehman suffers from the same problem, whereby an individual deviance results in an entirely different hashed value.

2.3 Summary

In this chapter we discussed the basics of graph theory, providing the necessary foundation to interpret the rest of this thesis. We introduced the popular Weisfeiler-Lehman algorithm, which has been widely used as a node relabeling procedure on graphs in order to incorporate more information into the node labels themselves.

We then delved into the world of recurrent neural networks, exploring the various network architectures and equations integral to the process we are investigating. We discussed the updates to the traditional neural network structure to account for temporal dependencies, and how those temporal dependencies impact the training of the network through Backpropagation Through Time. We then provided an overview of the classification method used by the original authors, that is, a Support Vector Machine.

Subsequently, we moved into a discussion of the related methods and approaches other researchers have pursued in order to solve the question of graph classification. While each high-level approach has its own merits and drawbacks, they all generally struggle to balance the same two concerns: how to encode as much information as possible in order to effectively classify a graph, without being computationally infeasible. Feature extraction offers a relatively computationally efficient method for graph classification, but can suffer from information loss, given the features are often hand-crafted. Kernel methods have long established strong performance, but can suffer from computational complexity since it is quadratic in the number of graphs. Finally, there has been a growing mass of machine learning approaches as another way to classify graphs, where it is often possible to learn a representations of a graph without manually defining features.

Now that we have situated ourselves in the field of graph classification, we will introduce the specific datasets we are concerned with, and then proceed with a more detailed inspection of the main method considered in the thesis.

3 Datasets

In this thesis, we work with labeled graph datasets, where both the graph has a class label and the nodes have node labels. We work with two general types of data, where the dataset is either comprised of chemical compounds, or represents proteins. Together these datasets are among some of the classic benchmark datasets in bioinformatics.

3.1 Datasets Description

The MUTAG dataset [5] consists of 188 chemical compounds, where the node labels correspond to the atomic element and the edges correspond to the bonds between atoms. Specifically, these molecules are various aromatic and heteroaromatic compounds that were tested for mutagenic effects on the bacterium Salmonella typhimurium TA98.

The PTC_MR dataset [30] contains 344 chemical molecules whose carcinogenicity was evaluated on male and female rats and mice. For this thesis we work specifically with the results from the male rat dataset, PTC_MR. The nodes represent the atomic element and the edges represent the bonds between atoms.

The ENZYMES dataset [25] contains 600 examples of enzymes that belong to one of six different functional groups from the Enzyme Commission top level enzyme classes.

The PROTEINS dataset [2] comprises 1113 different proteins which are labeled as either enzyme or non-enzyme. In this dataset, the nodes represent the secondary structure component of the protein, and the edges represent its neighbors in the amino acid chain or the larger 3D space.

NCI1 [31] is another dataset of 4110 chemical compounds. It is a subset of the larger NCI dataset, and in this case was tested for activity against non-small cell lung cancer.

NCI109 [31] is a different subset of the NCI dataset containing 4127 chemical compounds, tested for activity against ovarian cancer cell lines.

Table 3.1: Summary statistics of the datasets.

Dataset	**# Graphs**	**Avg |V|**	**Avg |E|**	**Avg d(v)**	**# Labels**	**# Classes**	**Class Ratio**
MUTAG	188	17.93	19.79	2.20	7	2	125/63
PTC_MR	344	14.29	14.69	2.06	18	2	152/192
ENZYMES	600	32.46	62.14	3.83	3	6	100 each
PROTEINS	1113	39.05	72.82	3.73	3	2	663/450
NCI1	4110	29.87	32.30	2.16	37	2	2053/2057
NCI109	4127	29.68	32.13	2.17	38	2	2048/2079

4 Methods

In this chapter we will discuss the new approach proposed by Taheri et al. to find a vector representation of a graph using recurrent neural network autoencoders [29]. This provides the necessary detail to understand the approach we are investigating and trying to improve upon.

The overarching goal is to take a dataset of graph-structured data and find a suitable vector representation for each graph, which can later be used for either clustering or classification. The method proposed to achieve this is to feed the graphs into a recurrent neural network autoencoder, extract a fixed size vector representation for each graph, and use that learned representation to classify the graph. This approach can broadly be segmented into three steps:

1. Preparing the input for the network
2. Building the network architecture
3. Defining the graph embedding function & performing classification.

We will discuss each of the three steps in greater detail, using the notation from Taheri et al.

4.1 Preparing the Input for the Network

The first step takes the raw data, i.e. the dataset comprised of adjacency matrices, node labels, and graph labels, and transforms it into a format that can be used as an input to an RNN, namely sequences. This section will discuss the two main aspects of preparing the input for the neural network: generating sequences from a graph, and embedding the vertices.

4.1.1 Sequence Generation

A key aspect of this approach is the decision to represent the graph as a set of sequences. This allows us to work with RNNs, which handle sequential data, and is a logical choice given the historical success of graph kernel methods that are based off of sequences, such as the Shortest Path Kernel [1] and the Random Walk Kernel [12]. We now consider three methods to generate graph sequences.

Random Walk

For each node in a graph, one or multiple random walks of a given length are generated originating at that node. The original paper considered random walks of length $\{3, 5, 10, 15, 20\}$. At each step in the walk, the next node v_{t+1} is chosen at random with probability $\frac{1}{d(v_t)}$ where v_t is the current node and $d(v_t)$ is the degree of v_t, i.e, its number of neighbors.

Shortest Path

In this approach, the shortest path between all pairs of nodes in a graph is found using the Floyd-Warshall algorithm, which is computable in $\mathcal{O}(n^3)$, where n is the number of nodes in the graph [8]. The input for the network becomes the sequences of shortest paths between the nodes. Note that in this approach, in contrast to random walks, we will have sequences of varying length. This is permissible due to the nature of RNNs, which can accommodate variable length input. There is one sequence per pair of nodes, which in a given graph with n vertices results in $\frac{n(n-1)}{2}$ sequences for that individual graph. We reproduce the algorithm here for completeness.

Algorithm 1: Floyd-Warshall Shortest Path

Data: Adjacency matrix A for graph $G = (V, E)$, $n = |V|$
Result: S, a matrix of shortest path lengths between all pairs of nodes

$$S_{ij} = \begin{cases} 1, & \text{if } A_{ij} = 1 \\ 0, & \text{if i=j} \\ \infty, & \text{otherwise} \end{cases}$$

for $k \in 1, ..., n$ **do**
  for $i \in 1, ..., n$ **do**
    for $j \in 1, ..., n$ **do**
      if $S_{ij} > S_{ik} + S_{kj}$ **then**
        $S_{ij} = S_{ik} + S_{kj}$
      end
    end
  end
end

Breadth-First Search

Breath-first search takes a node and generates a sequence by searching out laterally to all nodes within a given distance before extending the search to include nodes of a greater distance. The authors considered BFS at most 1 edge away from a node, making this effectively a list of a node's neighbors. If a node has more than 10 neighbors, the sequence is split into multiple sequences such that each sequence begins with the source node, and is no longer than 10 nodes long. In this approach, one sequence is generated per node in each graph, although the number of sequences given as input could be larger due to the splitting of long sequences.

4.1.2 Vertex Embedding

Once the graph has been represented as a list of sequences, it is in a format that can be used as input to a recurrent neural network. The final step before passing it to the network is to embed the vertices, meaning convert the node label to a numeric representation that can be given to the neural network. Given a set of all unique node labels, we can assign a d-dimensional vector with

values sampled uniformly at random from $[-1, 1]$. Any two nodes that share the same label will have the same vertex embedding. The vertex embedding step can be performed for node labels corresponding to any iteration of the Weisfeiler-Lehman algorithm detailed in the Background section. The set of unique node labels will correspondingly be different for different iterations of the algorithm, but the mechanism by which vertices are embedded is the same. The authors consider two separate approaches to vertex embeddings: one where the embeddings are considered fixed, and another where the embeddings are considered trainable parameters of the model, and are updated during the training of the neural network. When the vertex embeddings are trainable, the hope is that the network will learn the embedding in such a way that similar nodes are close to one another in the embedding space.

4.2 Building the Network Architecture

Once the data has been prepared, we concern ourselves with the architecture of the network itself. First we will discuss the general architecture approach that is pursued, and then discuss the nuances of the individual models considered.

4.2.1 Sequence-to-Sequence Autoencoder

Taheri et al. chose a sequence-to-sequence recurrent neural network autoencoder, inspired by the sequence-to-sequence models in natural language processing, but adapted for graph-structured data [28, 17]. The *sequence-to-sequence* aspect of this network means that the input is a sequence, and the output of the network is also a sequence. To achieve this, we need to use two RNNs (in our case LSTMs): one to convert the sequence to a fixed size representation, and a second to convert the fixed size representation to a sequential output. The *autoencoder* aspect simply means that the output of the network is the same as the input, i.e, the network is trying to predict its own input. In this process, the first part of the network encodes the input to an intermediate representation, and the second decodes this representation back to a sequential output. The network optimizes the intermediate representation such that it is best able to reconstruct the input sequence. After training is complete, we can extract the intermediate representation – which is a fixed size vector representation of the initial input sequence – to cluster or classify the graphs.

Table 4.1: A short reference table for the notation used in this section.

Symbol	Meaning
$\overline{v_t}$	Predicted vertex at time t
$Emb(v_t)$	Embedding of true vertex v_t
$\overline{Emb(v_t)}$	Predicted embedding of true vertex v_t
$Emb(\overline{v_t})$	Embedding of predicted vertex v_t
h_t^{enc}	The cell output at time point t in the encoder portion of the network
h_t^{dec}	The cell output at time point t in the decoder portion of the network
$LSTM_{enc}(s)$	The learned vector representation for sequence s

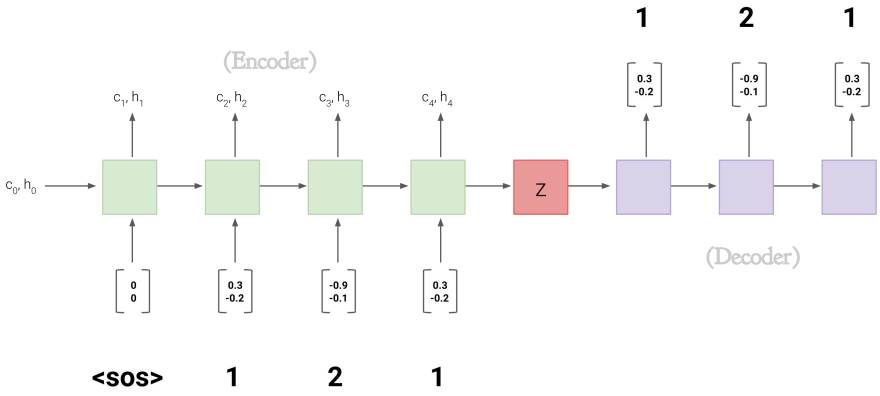


Figure 4.1: In a sequence-to-sequence LSTM, the network takes a sequence as input, with one node in the sequence per time point. In this example, our sequence is "1-2-1", to which we've appended a start of sequence (<sos>) token. The node labels are first converted to a vertex embedding (in this case, a two-dimensional embedding), which are then given as input to the network. The network receives this information, as well as the information from the previous time step, h_{t-1} and c_{t-1}. Once all the time steps have been read into the network, we have successfully encoded the sequence down to a fixed size vector representation z. The decoder portion of the network then uses this vector z, along with the input v_t (not visualized in the decoder portion) to try and reconstruct the original sequence.

4.2.2 Models

We will now detail the specific model variations considered in the Taheri et al. paper, using their original notation. In each model the following parameters were specified by the authors: the AdaGrad optimizer [6] with learning rate = 0.01 and mini-batch size = 100.

S2S-AE

The first model, S2S-AE (for Sequence-to-Sequence Autoencoder), is the base model considered. All combinations of fixed or trainable vertex embeddings, shortest path (SP), random walks (RW), or breadth-first search (BFS) for sequence generation, and node labels from iterations 0, 1 and 2 of Weisfeiler-Lehman (WL0, WL1 or WL2) are tested. The first LSTM encodes the input sequence as follows:

$$h_t^{enc} = LSTM_{enc}(Emb(v_{t-1}), h_{t-1}^{enc}). \tag{4.1}$$

The second LSTM decodes it, using the true vertex labels as input in the decoder LSTM:

$$h_t^{dec} = LSTM_{dec}(Emb(v_{t-1}), h_{t-1}^{dec}). \tag{4.2}$$

When the embeddings are fixed, h_t^{dec} then passes through a ReLU activation to return the predicted embeddings for each vertex v_t:

$$\overline{Emb(v_t)} = ReLU(h_t^{dec} W_y + b_y). \tag{4.3}$$

If the vertex embeddings are trainable, the predicted vertex $\overline{v_t}$ is computed as follows:

$$\overline{v_t} = \underset{l \in L}{\arg\max}(h_t^{dec} \cdot Emb(l)) \tag{4.4}$$

where L denotes the set of all vertex labels. In both situations $Emb(v_0)$ is a vector of zeros.

S2S-AE-PP

This next model is similar to S2S-AE, and similarly supports all combinations of fixed/trainable vertex embeddings, SP, RW, or BFS sequences, and WL0, WL1 or WL2 vertex labels. However, instead of giving the true node x_t from the input sequence at each time step as input into the decoder LSTM, it uses the predicted vertex label (or predicted vertex embedding, if the vertex embedding is fixed) from the previous time step in the sequence. The decoder portion of the network is thus updated to the following:

$$h_t^{dec} = LSTM_{dec}(Emb(\overline{v_{t-1}}), h_{t-1}^{dec}). \tag{4.5}$$

S2S-AE-PP-WL1,2

The third model considered extends the second model, S2S-AE-PP, to include the vertex labels from both the 1st and 2nd iteration of the WL algorithm. The network will correspondingly return a prediction for both the WL1 and WL2 labels for a given node. Our gate calculations in the LSTM (2.1) are thus modified to include both inputs, and correcting for a minor notation error in the original paper are thus:

$$i_t = \sigma(W_{i_1} x_{1_t} + W_{i_2} x_{2_t} + U_i h_{t-1} + K_i c_{t-1} + b_i) \tag{4.6}$$
$$f_t = \sigma(W_{f_1} x_{1_t} + W_{f_2} x_{2_t} + U_f h_{t-1} + K_f c_{t-1} + b_f) \tag{4.7}$$
$$o_t = \sigma(W_{o_1} x_{1_t} + W_{o_2} x_{2_t} + U_o h_{t-1} + K_o c_{t-1} + b_o). \tag{4.8}$$

In this setup, we still consider all sequence generation possibilities (SP, RW, BFS), but only treat the vertex embeddings as trainable, and test only the combination of node labels WL1+WL2.

S2S-PP-N2N

The final model is a modification again of S2S-AE-PP. As input to the network, it takes the average of the vertex embeddings of the neighbors for each node in the sequence, rather than the embedding of the node itself. The original node is therefore not part of the input. Thus the encoding layer thus becomes:

$$h_t^{enc} = LSTM_{enc}(\underset{v_i \in N(v_t)}{Avg}(Emb(v_i)), h_{t-1enc}). \tag{4.9}$$

The decoding layer is the same as in (4.5), and tries to predict the label of the original node. Note that this model is not an autoencoder, since the input and the output are not the same. This setup considers specifically WL1 for node labels, RW sequences, and trainable vertex embeddings.

4.2.3 Loss Functions

During the training of the neural network, one of two loss functions was used, depending on whether the vertex embeddings were fixed or trainable.

Squared Loss

When the vertex embeddings were fixed, the Squared Loss was computed:

$$loss_{SE}(s) = \frac{1}{|s|} \sum_{t=1}^{|s|} ||\overline{Emb(v_t)} - Emb(v_t)||_2^2. \tag{4.10}$$

Categorical Cross Entropy

If the vertex embeddings were trainable, the Categorical Cross Entropy was used:

$$loss_{CE}(s) = -\sum_{t=1}^{|s|} \log p(\overline{v_t} = l) \tag{4.11}$$

$$p(\overline{v_t} = l) = \frac{e^{h_t^{dec} \cdot Emb(l)}}{\sum_{l' \in L} e^{h_t^{dec} \cdot Emb(l')}}. \tag{4.12}$$

Here, l is the true node label of v_t and $\overline{v_t}$ is computed as in (4.4).

4.3 Defining the Graph Embedding Function & Performing Classification

Once the training of the neural network is complete, each training sequence corresponds to a fixed dimensional vector representation. The final two steps in the method proposed by Taheri et al. is to aggregate over all the sequences in each individual graph, and perform some clustering or classification on the aggregated learned graph representations.

4.3.1 Graph Embedding Function

The authors chose to average over the learned vector representations for all sequences in a given graph to obtain a single vector representation per graph. Correspondingly, the graph embedding function for a given graph G is computed as follows:

$$\Phi(G) = \frac{1}{|Seq(G)|} \sum_{s \in Seq(G)} LSTM_{enc}(s). \tag{4.13}$$

4.3.2 Classification

Finally, equipped with a fixed size vector representation of each graph in the data set, we can perform classification (we note it is also possible to perform clustering, but for our purposes we will focus more on classification). The authors chose to classify the graphs using C-SVM with a radial basis kernel function via repeated 10-fold cross validation. The hyperparameters of the C-SVM were tuned using nested cross validation.

4.4 Summary

Taheri et al. proposed a novel method to classify graphs by learning a vector representation via recurrent neural network autoencoders. The principal aspects of the method are to (i) represent a graph as a set of sequences, (ii) learn a vector representation of the graph by training a RNN autoencoder, and (iii) classify the learned representations using a C-SVM classifier. The method showed promising results, achieving strong graph classification accuracy on most benchmark datasets. We will now discuss the specifics of our implementation of this method and compare our results to theirs.

5 Implementation Comparison

Having explained the details of the method we are investigating, we now turn to our own implementation. In this chapter, we will begin by detailing the specifics of our implementation, including the parameters we used and any high-level decisions that were made. Then we will proceed with a comparison of the performance that we observed across the different models, and where possible, with the performance achieved in the paper. We will discuss the observations we made during the process, and finally, will end with a comparison of the performance to other state-of-the-art sequence-based methods.

5.1 Implementation Notes & Parameter Choices

All implementations were using PyTorch 1.0.1, and are based off of the description of the implementation in the original paper. While our implementation is in large part aligned with that of the original paper, there are a couple of differences observed. This is suspected to be due to differences in parameter choices, as not all parameters were specified in the paper. We reached out to the authors to compare our implementation with theirs to reconcile the differences, however, they are not yet releasing their code. We followed up with specific questions on certain parameter choices, twice, but have not heard back.

We will therefore specify the parameters and decisions made in each step of the re-implementation to facilitate reproducibility, organized by the different steps of the method. We used the same parameters as the authors whenever they were provided, and among those that were not specified, if we found a parameter provided by one of the papers the original method was based upon, we

Table 5.1: Summary of parameter values used in the implementation.

Parameter	**Value**	**Specified by**
Length of each RW	5	Taheri et. al [29]
# Sequences per Node (for RW)	5	Ourselves
Vertex Embedding Dimension	100	Taheri et. al [29]
Learning Rate	0.01	Taheri et al. [29]
Learning Rate Decay	0	Ourselves
Weight Decay	0	Ourselves
Minibatch Size	100	Taheri et al. [29]
Weight Initialization	$\mathcal{U}(-0.08, 0.08)$	Sutskever et al. [28]
# Epochs	1	Ourselves

used those. For parameters not specified at all, we tried using a random search to select these parameters. However, the computational cost of the necessary double cross validation proved to be a considerable barrier (>10 days on some datasets), and thus we instead use a fixed value chosen a priori for these parameters to prevent overfitting to our data. While this means some of the parameters are sub-optimal, and likely lowers the classification accuracy a bit, it at least ensures we did not choose an over-optimistic estimate of the classification accuracy, and it provides us with a baseline performance that we can make comparisons across models with. We will now explain specific parameters and decisions for each of the three steps in this process, with an overview of the parameters and their source provided in Table 5.1. In all steps the random seed was 47.

5.1.1 Sequence Generation

When generating shortest path sequences between all pairs of nodes, it is possible to have multiple shortest paths of equal length between a given node a and b. In this situation, we chose the shortest path randomly. When generating sequences using breadth-first search, we sorted the list of neighbors in the sequence to facilitate learning. For random walk generation, as cited in the original paper, we used a walk length of 5 as that is what they found to have the best performance. They specified that they generated "multiple" random walks per vertex; we interpreted that as 5 sequences per node.

5.1.2 Network Architecture

The neural network is a single-layer LSTM autoencoder. Weights in the network were initialized uniformly at random from $[-0.08, 0.08]$. This was not specified in the paper, but was the choice of one of the papers this method was based upon [28]. All bias terms were initialized to 1, as is standard practice. As specified in the paper, we used a minibatch size of 100, a learning rate of 0.01, and the AdaGrad optimizer. The number of epochs was not specified, so we report the results after one epoch of training. As mentioned in the paper, sometimes the vertex embeddings were treated as fixed, other times as trainable. They found the trainable vertex embeddings outperformed the fixed variant and only report on those; we do the same. Learning rate decay and weight decay were not specified, so we used a fixed value of 0.

Consideration of Whether to use a Validation Set

When working with neural networks, a natural question to arise is whether the neural network is overfitting to the data, which typically becomes apparent when one calculates the generalization error. In our setup with a neural network autoencoder, the question is whether it is overfitting when reconstructing the input sequences. However, we have to introduce an extra step to check this, since our generalization error of classification accuracy comes from the SVM step, which is an isolated step from the autoencoder. We therefore considered introducing a validation set in the neural network to check if the network was overfitting, and compare the reconstruction error of a training versus validation set. We could withhold training samples at the sequence level, or at the graph level. To be safe, we decided to withhold samples at the graph level, in case there were within graph sequence similarities that could bias our results since the sequences from the same graph could exist in both sets in this setup. We introduced a validation set with 10% of

graphs withheld. However, it quickly became apparent that this was not useful in this setup. Even though we withheld entire graphs, the sequences were repeated so often that the validation loss mirrored the training loss, as visible in Figure 5.1, since the sequences in the validation set inevitably existed in the training set as well. Thus the validation loss tracked the training loss very closely, and never results in the the classic U shape that indicates overfitting, no matter how long the network network was trained for.

Upon further investigation it became clear just how severe this sequence overlap was. Table 5.2 shows how small a percentage of sequences in the dataset are unique, regardless of sequence type. It shows that this is particularly pronounced for the datasets with lower average degree (MUTAG, PTC_MR, NCI1 and NCI109). It also became clear that the sequences are also highly concentrated, with the top 5% of unique sequences accounting for as much as 77% of the total sequences in a given dataset.

5.1.3 Classification using SVM

We fit an SVM using `sklearn's SVC` classifier with balanced class weights to handle class imbalance. As in the paper, we used the radial basis function and used an inner cross validation (with 5 folds) to select the C and γ parameters. The inner cross validation performed a grid search over the C and γ parameters, with the grid made over the values specified in Table 5.3. The radial basis function is sensitive to differences in scale, so we normalized the learned graph vector representations using the max norm.

5.2 Performance Comparison

We now present our results, and where possible provide a comparison to the results published by the paper. There were at least 40 different configurations tested in the original paper, varying between one of four models, one of three different versions of sequence generation (SP, RW, BFS), treating the vertex embeddings as fixed or trainable, and one of the first three iterations of the WL algorithm, inclusive of the original labels at iteration 0 (WL0, WL1, WL2). As mentioned

Table 5.2: Percentage of unique sequences and concentration of the top 5% of unique sequences in the training data, across sequence generation methods and datasets, using node labels from WL1.

Method	**MUTAG**	**PTC_MR**	**ENZYMES**	**PROTEINS**	**NCI1**	**NCI109**
RW % Unique	0.10	0.25	0.61	0.58	0.08	0.08
SP % Unique	0.10	0.25	0.51	0.55	0.24	0.24
BFS % Unique	0.05	0.17	0.53	0.48	0.03	0.03
RW Top 5% Unique, as % of Total	0.73	0.48	0.28	0.33	0.71	0.71
SP Top 5% Unique, as % of Total	0.66	0.39	0.32	0.33	0.53	0.54
BFS Top 5% Unique, as % of Total	0.68	0.51	0.30	0.37	0.77	0.77
RW # Total Sequences	16855	24575	97370	217330	613735	612470
SP # Total Sequences	30505	46645	372600	1992069	2149633	2136355
BFS # Total Sequences	3371	4915	19474	43492	122747	122494

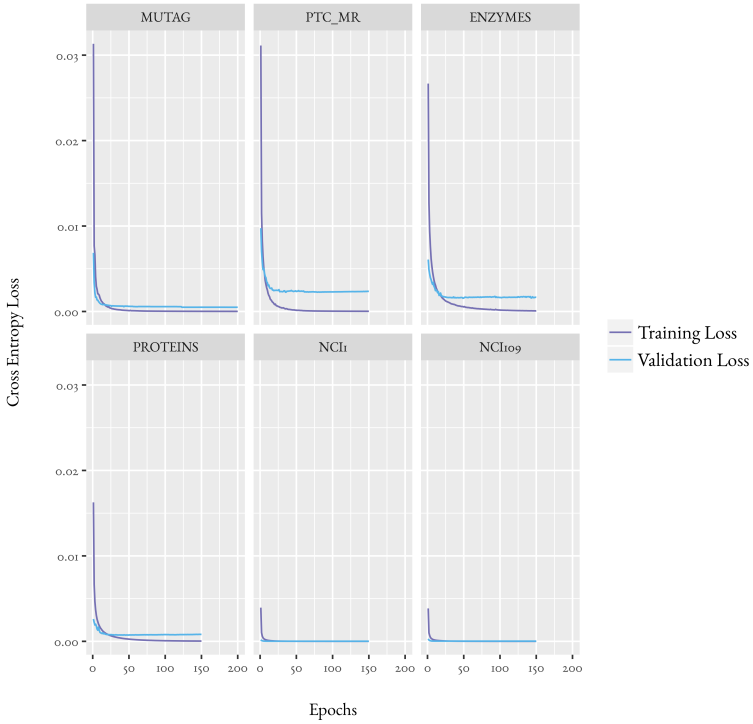


Figure 5.1: The training and validation loss over epochs across all datasets, when 10% of graphs are withheld in the validation set. All datasets used BFS for sequence generation and WL1 for node labels. Due to the high repetition of sequences, the validation loss tracks the training loss very closely, and never diverges to the class U shape that indicates overfitting.

above, since the trainable vertex embeddings had superior performance to the fixed embeddings, the authors only present the trainable findings, which we do as well. However, despite all these configurations, only the results from their fourth and best performing model were provided. Results for Models 1-3 are partially provided in graph form, rather than table form, but only for MUTAG, PTC_MR & ENZYMES. We nevertheless will present our results from all our models across all datasets and where feasible, compare our performance with theirs.

Table 5.3: Search space of the parameters used in the SVM.

Parameter	**Values**
C	{0.001, 0.01, 0.1, 1, 10, 100}
γ	{0.000000001, 0.00000001, 0.0000001, 0.000001, 0.00001, 0.0001, 0.001, 0.01, 0.1, 1, 10, 100, 1000}

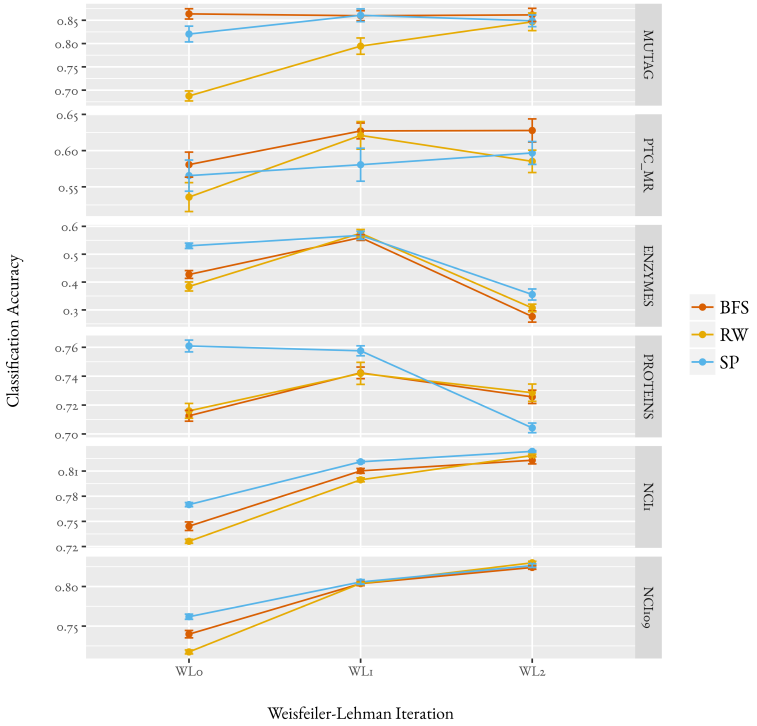


Figure 5.2: Performance of Model 1 by WL iteration and by sequence type.

5.2.1 Model 1: S2S-AE

Model 1 is the standard LSTM Autoencoder, but uses the true vertex embeddings as input into the decoder portion of the network. The authors found strongest performance in WL1, which we found to have relatively good and consistent performance across datasets and sequence types. They did not provide performance for NCI1 or NCI109 on this model, but we find better per-

Table 5.4: Classification accuracies across datasets, iterations of Weisfeiler-Lehman node relabeling, and sequence generation types using Model 1 (S2S-AE).

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
WL0 BFS	**86.37 (± 1.1)**	58.06 (± 1.7)	42.72 (± 1.4)	71.26 (± 0.4)	74.42 (± 0.5)	73.98 (± 0.5)
WL0 RW	68.75 (± 1.1)	53.58 (± 2.0)	38.42 (± 1.6)	71.60 (± 0.5)	72.62 (± 0.2)	71.74 (± 0.2)
WL0 SP	82.04 (± 1.7)	56.55 (± 2.1)	53.02 (± 1.0)	**76.09 (± 0.4)**	77.00 (± 0.2)	76.19 (± 0.3)
WL1 BFS	85.97 (± 1.1)	62.72 (± 1.1)	55.98 (± 1.0)	74.23 (± 0.4)	81.02 (± 0.3)	80.36 (± 0.3)
WL1 RW	79.45 (± 1.8)	62.10 (± 1.9)	**57.53 (± 1.3)**	74.19 (± 0.8)	79.96 (± 0.2)	80.39 (± 0.2)
WL1 SP	86.07 (± 1.4)	58.05 (± 2.3)	56.75 (± 1.4)	75.76 (± 0.3)	82.43 (± 0.1)	80.57 (± 0.3)
WL2 BFS	86.15 (± 1.4)	**62.78 (± 1.6)**	27.63 (± 2.0)	72.57 (± 0.5)	82.29 (± 0.4)	82.41 (± 0.2)
WL2 RW	84.67 (± 1.9)	58.52 (± 1.5)	30.70 (± 1.4)	72.86 (± 0.6)	82.85 (± 0.2)	**83.00 (± 0.2)**
WL2 SP	84.88 (± 1.3)	59.67 (± 1.6)	35.53 (± 2.0)	70.42 (± 0.3)	**83.34 (± 0.1)**	82.64 (± 0.2)

formance using WL2 on these two datasets, across all sequence generation methods. Whereas the authors found generally comparable performance across sequence generation methods for MUTAG and PTC_MR, with roughly 85% and 60% classification accuracy respectively, we only observe that among BFS and SP sequences. RW often has lower performance, particularly in WL0, as well as in MUTAG with WL1. The authors found ENZYMES performance to be less consistent across methods and WL iterations, which we also observed. We observe a generally improved performance when moving from WL0 to WL1, but an inconsistent improvement across datasets comparing WL1 to WL2. Figure 5.2 shows the performance of Model 1 across the different iterations of Weisfeiler-Lehman and among different sequence types.

5.2.2 Model 2: S2S-AE-PP

Model 2 shares the same general architecture as Model 1, with the modification that the *predicted* vertex from the previous time step is used as the input in the decoder, rather than the true vertex. This is supposed to help the network "learn" more and rely less on the training data. Despite this difference in setup the performance is largely similar to Model 1, as is also observed by the original authors, and is shown in Figure 5.3. As in Model 1, WL1 performance is generally superior to the other iterations, with the notable exception of PROTEINS when the sequence is generated by shortest path, which we observe to decrease as the iteration of Weisfeiler-Lehman increases. Despite this, the trends remain similar. Again we see relatively similar performance for MUTAG and PTC_MR across WL iterations and sequence generation methods (with the exception of RW), and much more variable results for ENZYMES.

5.2.3 Model 3: S2S-AE-PP-WL1,2

Model 3 is an extension of Model 2 that takes the vertex labels from both WL1 and WL2 as input. The original authors found similar, but perhaps slightly lower performance for this model on MUTAG and PTC_MR, and notably worse performance on ENZYMES. We observed similar

Table 5.5: Classification accuracies across datasets, iterations of Weisfeiler-Lehman node relabeling, and sequence generation type using Model 2 (S2S-AE-PP).

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
WL0 BFS	85.46 (± 1.4)	57.00 (± 1.7)	43.37 (± 1.6)	70.78 (± 0.8)	74.48 (± 0.2)	73.52 (± 0.3)
WL0 RW	69.15 (± 2.0)	54.00 (± 1.5)	37.67 (± 1.4)	71.97 (± 0.5)	72.05 (± 0.3)	71.89 (± 0.4)
WL0 SP	83.19 (± 1.4)	56.97 (± 1.6)	49.42 (± 1.6)	**76.22 (± 0.3)**	77.24 (± 0.2)	76.16 (± 0.4)
WL1 BFS	84.75 (± 1.0)	63.46 (± 1.9)	56.05 (± 1.0)	73.05 (± 0.6)	81.29 (± 0.1)	80.76 (± 0.3)
WL1 RW	79.61 (± 1.7)	62.24 (± 1.7)	56.52 (± 1.2)	74.64 (± 0.4)	79.70 (± 0.5)	80.49 (± 0.2)
WL1 SP	85.41 (± 0.8)	58.49 (± 2.1)	**59.92 (± 0.9)**	73.87 (± 0.5)	82.36 (± 0.2)	81.40 (± 0.2)
WL2 BFS	**86.41 (± 1.5)**	62.87 (± 2.6)	26.58 (± 1.2)	71.99 (± 0.4)	81.55 (± 0.3)	82.43 (± 0.2)
WL2 RW	86.17 (± 1.1)	**63.63 (± 1.5)**	28.57 (± 1.2)	74.46 (± 0.2)	**82.66 (± 0.2)**	**82.84 (± 0.2)**
WL2 SP	85.20 (± 2.3)	62.92 (± 1.5)	38.80 (± 1.7)	69.69 (± 0.6)	82.03 (± 0.2)	80.79 (± 0.2)

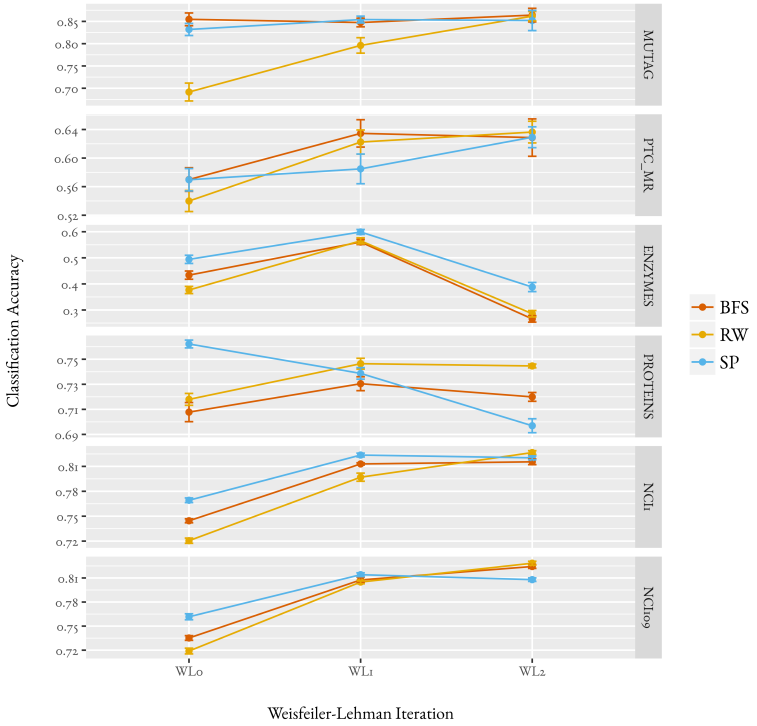


Figure 5.3: Performance of Model 2 by WL iteration and by sequence type.

trends, noting as well that the performance of ENZYMES is about 10 percentage points lower than its performance in Model 2.

5.2.4 Model 4: S2S-N2N-PP

In Model 4, the authors updated the architecture of Model 2 to take the average of a node's neighbors embeddings as input, and then predict the original node's embedding as output. In this model, they only tested random walks as sequence generation, and Weisfeiler-Lehman labeling from iteration 1. This was their best performing model, with actual results published. Our results here are a couple percentage points below the authors results on most datasets, and very far below on MUTAG (7 percentage points). We attribute this to two things. First, since we did not have all the parameters used by the authors, the roughly 2 percentage points difference observed

Table 5.6: Model 3 (S2S-AE-PP-WL1,2) classification accurarcies across datasets and sequence types.

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
BFS	**85.21 (± 1.4)**	61.75 (± 1.4)	45.73 (± 1.0)	75.78 (± 0.3)	81.61 (± 0.2)	83.20 (± 0.4)
RW	84.79 (± 1.2)	**62.78 (± 2.0)**	45.78 (± 1.9)	**76.12 (± 0.4)**	81.50 (± 0.2)	**83.45 (± 0.2)**
SP	82.85 (± 1.6)	60.57 (± 2.2)	**49.47 (± 0.9)**	75.88 (± 0.5)	**83.68 (± 0.2)**	82.74 (± 0.2)

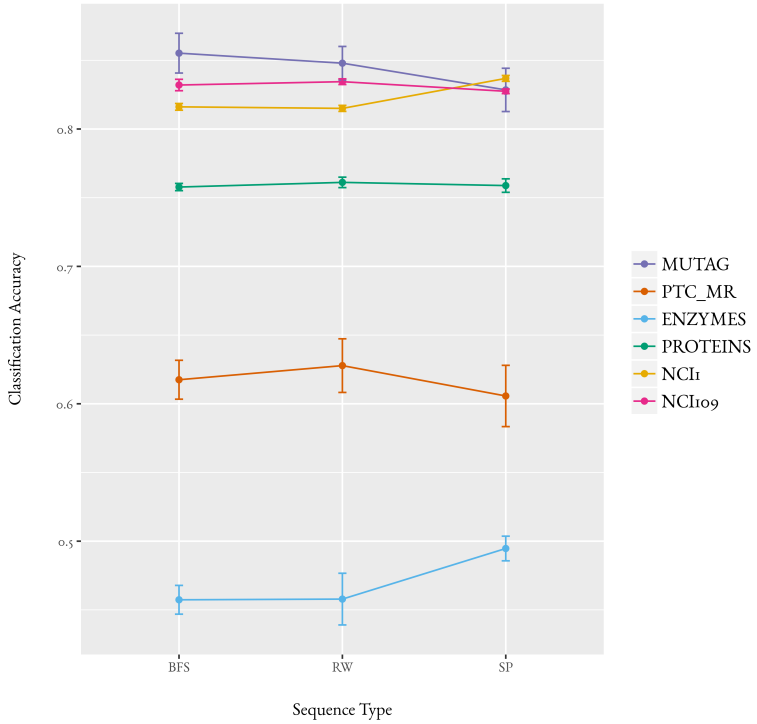


Figure 5.4: Model 3 performance by sequence type, using a combined input of WL1 and WL2 node labels.

in most datasets can likely be explained by this. As for MUTAG, where the difference is more extreme, we attribute this to the poor performance of random walks, which we've observed across all models for this dataset. It is also possible the authors generated random walks in a different manner, and could account for this difference. However, considering that Model 4 improves upon the Model 2 equivalent (sequences generated by random walks and using WL1 node labels) in all datasets, except PROTEINS, where it is almost identical, we conclude this model has potential to be the best performing model, as the authors found.

5.3 Observations from the Implementation

While we found generally consistent results with the main authors' implementation, with some consistent slight decreases that we attribute to differences in parameter values, there were a few

Table 5.7: Classification accuracies across datasets using Model 4 (S2S-N2N-PP) vs. the paper results.

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
WL1 RW	82.71 (± 1.6)	62.21 (± 1.9)	57.52 (± 0.8)	74.29 (± 0.5)	80.89 (± 0.4)	81.54 (± 0.3)
Taheri et al.	89.86 (± 1.1)	64.54 (± 1.1)	63.96 (± 0.6)	76.72 (± 0.5)	83.72 (± 0.4)	83.64 (± 0.3)

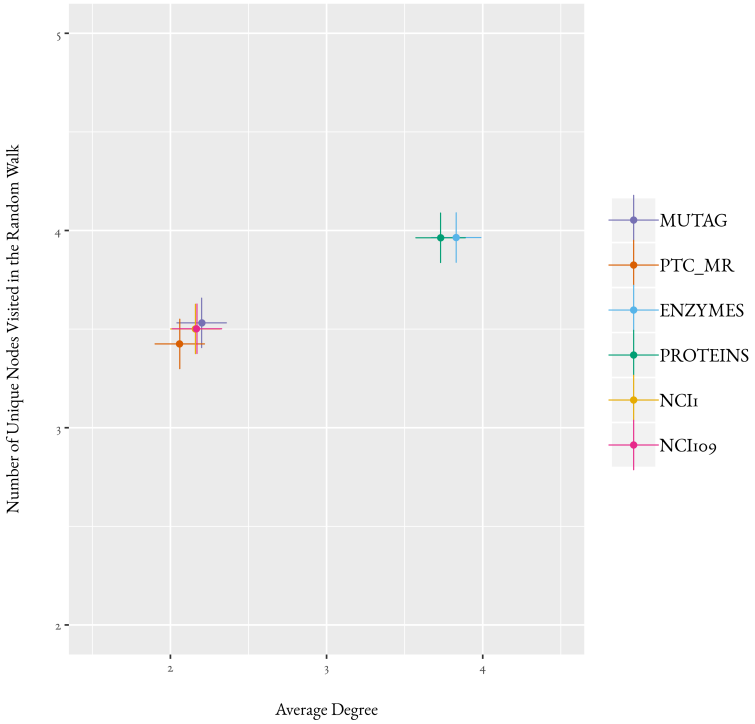


Figure 5.5: When generating random walks of length 5, datasets with a smaller average degree visited fewer unique nodes on its walk on average compared to datasets with larger average degree.

findings that seemed worthy to note: lower performance on random walks, and what happens when the network does not train at all. In this section we will explore these two findings in greater detail.

5.3.1 Lower Performance with Random Walks

As mentioned above, we have found our results to be generally consistent with the results found by the paper, and suspect that the discrepancies are due to differences in parameter choices. The one major exception to this is that our experience with sequences generated by random walks did not perform consistently well, and in particular performed poorly for WL0, and on MUTAG for WL1. It is possible this could be due to tottering, the phenomenon when a walk gets stuck going back and forth between the same two nodes instead of traversing elsewhere in the graph [18]. We observed that the datasets with a lower average degree tended to visit fewer nodes unique nodes in the walk, meaning they were tottering a bit more back and forth, compared to the datasets with higher average degree. For these datasets with smaller average degree, namely MUTAG, PTC_MR, NCI1 and NCI109, more than 50% of the random walks only visited 2 or 3 unique nodes out of the 5 on the random walk. In comparison, the datasets with larger average degree, ENZYMES and PROTEINS, only had around 30% of walks visiting so few nodes. Figure 5.5 shows the average number of nodes visited on the random walk of length 5 by their average

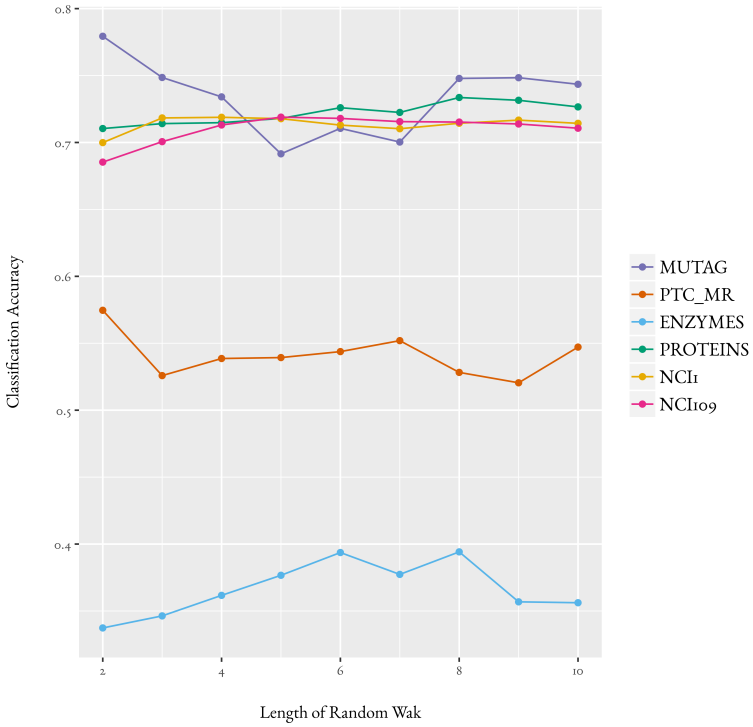


Figure 5.6: Comparison of RW performance for walks of varying fixed length. The baseline model used for this comparison is Model 2 (S2S-AE-PP) with node labels from WL0.

degree. We can observe two distinct groups, clustered by their average degree and resulting in similar unique number of nodes on the walk, which could partially explain some of the performance drop that we observed.

We decided to look at performance over different lengths of random walks, to understand if that could be impacting performance. Since we had worse performance with WL0, we used that iteration of Weisfeiler-Lehman in Model 2 to compare performance. Figure 5.6 shows performance of random walks of fixed length up to length 10. We once again observed a difference between the datasets with smaller average degree and those with a larger average degree. ENZYMES and PROTEINS, with larger average degree, enjoy more or less increasing classification accuracy as the length of the walk increases, up until length 8. MUTAG and PTC_MR on the other hand have a more erratic performance, with performance initially decreasing as the length of the walk increases. Although PTC_MR does decrease, after length 3 it begins to increase again. Furthermore, its decrease is not as dramatic as that of MUTAG, which drops nearly 10 percentage points between walks of length 2 and length 5. Based on this observation it is not surprising that MUTAG has lower performance on random walks.

5.3.2 Performance without Training

The other interesting observation we made during the baseline implementation was regarding how much the neural network truly learns in this process. Just initializing the weights at random as detailed above in Section 5.1.2, and pushing the training data through a single forward pass of the network (with no backpropagation/weight updates), already separated the data well enough to classify the datasets with decent baseline performance. Table 5.8 shows the performance when there is no training (epochs=0) versus the results after training (epochs=1) for Models 2 and 4. While the performance without training the network is not state-of-the-art, it does give a competitive performance to other graph classification methods. Furthermore, while we do see that training the network improves performance, it does so only by a small amount. On some datasets, performance even decreases, which we attribute to the large learning rate (for a smaller learning rate, this decrease is not observed, but the general trend of marginal increases compared to no training remains). On the one hand, these percentage point gains can make the difference between a decent method and a state-of-the-art method. On the other hand, it questions how much the method is actually "learning" about the graph. It seems that merely pushing the data through a series of linear and non-linear transformations, and averaging over these for a given graph is already a decent separator. It is also worth noting that it is also possible that this method is better suited for larger datasets, as neural networks tend to thrive when given a lot of data. Given that the datasets we are working with are quite small, it is a possible consequence of that.

5.4 Comparison to Other Sequence-based Methods

Now that we have results from our own implementation, we were interested to understand how much this method improves over the other existing sequence-based methods to classify graphs. Specifically, we wondered whether we could achieve similar results with the methods DeepWalk and node2vec, which use sequences to learn node embeddings on graphs. We are particularly interested to understand whether the learned graph representation based on sequences offers an advantage over a learned graph representation based on nodes, or whether they are comparable to one another. Since the node embedding methods learn a fixed sized vector for each node in a graph, we will then average over all nodes in a graph in order to have a single vector representation of a graph, similar to how we took the average over all sequences in the main method. We will then use this aggregated graph representation based on nodes as input to the same SVM we

Table 5.8: Classification accuracies comparing no training of the network (Epochs=0) with training (Epochs=1) across Model 2 (WL1, BFS) and Model 4 (WL1, RW).

Method	**MUTAG**	**PTC_MR**	**ENZYMES**	**PROTEINS**	**NCI1**	**NCI109**
M2, Epochs=0	85.63 (± 1.5)	61.57 (± 1.8)	45.92 (± 1.4)	74.69 (± 0.4)	79.98 (±0.4)	79.81 (± 0.3)
M2, Epochs=1	84.75 (± 1.0)	63.46 (± 1.9)	56.05 (± 1.0)	73.05 (± 0.6)	81.29 (± 0.1)	80.76 (± 0.3)
M4, Epochs=0	82.08 (± 1.5)	58.76 (± 1.7)	49.32 (±1.4)	73.32 (± 0.8)	78.08 (± 0.5)	78.18 (± 0.3)
M4, Epochs=1	82.71 (± 1.6)	62.21 (± 1.9)	57.52 (± 0.8)	74.29 (± 0.5)	80.89 (± 0.4)	81.54 (± 0.3)

used in the original method, using a radial basis function and a double cross validation in order to tune the hyperparameters. We trained on the same splits of the data as in the original method to ensure comparability. Since the original paper used node labels according to one the first three iterations of the Weisfeiler-Lehman algorithm, we similarly repeat our evaluation across those three iterations in order to complete a fair comparison. We will first introduce each method and their individual results, then proceed with a comparison of the best performing models from each to our best model from Taheri et al.'s method.

5.4.1 DeepWalk

DeepWalk generalizes the skipgram variant of the Word2vec algorithm for graph structured data [23]. Skipgram, and Word2vec more generally, was designed to find a vector representation of words to help solve problems within natural language processing by training a simple feed-forward network with sentences. The input in skipgram is a word, and the output is the surrounding words, also called the context words, of a specified size. Given an input word, the network tries to predict the surrounding context words, and through training, learns vector embeddings for each word, returning a vector representation for each word at the end. Translating this to the graph landscape, DeepWalk generates sequences in a graph via random walks, and considers these random walks to be the equivalent of sentences, and nodes the equivalent of words.

We used the implementation from the DeepWalk authors to generate the node representations for the datasets of interest, using the same parameters as before in order to ensure comparability. We set the same random seed, the same number of random walk sequences per node (5), same length of random walk (5), and the same vector dimension size of 100. We then averaged the node vector representation of all nodes in the graph. We repeated this for the different iterations of Weisfeiler-Lehman considered in the original paper (WL0, WL1, and WL2).

Results

We report the full results of DeepWalk in Table 5.9, with an accompanying visualization in Figure 5.7. We see performance steadily increase across all datasets except ENZYMES for higher iterations of Weisfeiler-Lehman, meaning the strongest performance was observed at WL2. We see impressive performance in particular on MUTAG, NCI1 and NCI109. ENZYMES sees a large performance gain from WL0 to WL1, then experiences a decrease from WL1 to WL2. PTC_MR and PROTEINS both gain performance as the iteration of Weisfeiler-Lehman increases, but with less dramatic increases relative to the other datasets.

Table 5.9: Classification accuracies across datasets using Deepwalk, across WL iterations.

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
WL0	70.90 (± 2.2)	56.30 (± 1.9)	30.23 (± 0.9)	69.95 (± 0.3)	65.53 (± 0.3)	64.89 (± 0.2)
WL1	85.87 (± 1.6)	59.58 (± 1.8)	**55.35 (± 1.3)**	72.46 (± 0.7)	78.44 (± 0.4)	77.45 (± 0.3)
WL2	**87.98 (± 1.1)**	**60.76 (± 1.7)**	48.80 (± 2.4)	**74.25 (± 0.4)**	**82.87 (± 0.3)**	**82.64 (± 0.3)**

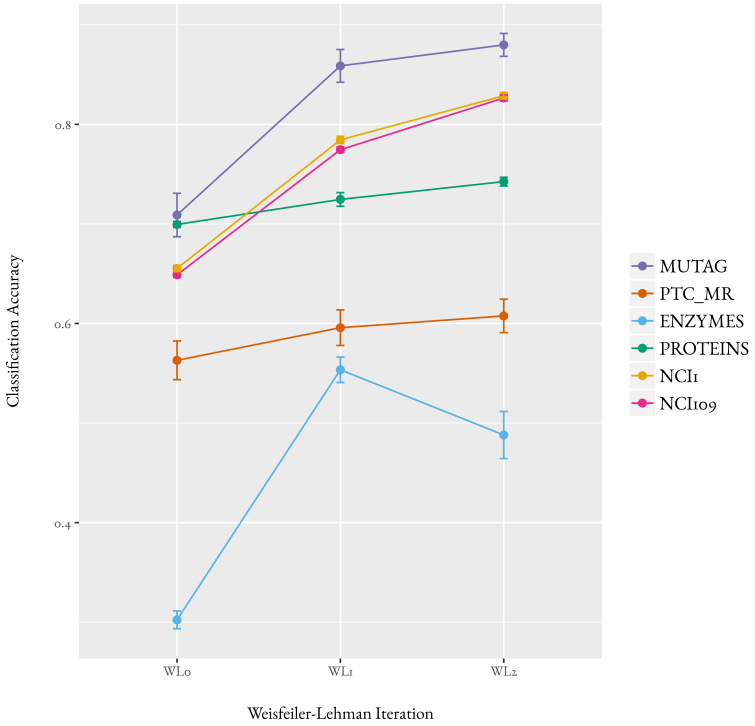


Figure 5.7: DeepWalk classification performance by WL iteration.

5.4.2 node2vec

Node2vec is another generalization of the Word2vec algorithm for graph structured data [11]. Like DeepWalk, node2vec gives as output a vector representation for each node label in the graph. It extends the idea of DeepWalk by generating biased random walks in the graph, with two parameters p and q which determine whether the walk favors staying within the neighborhood of the source node, or whether it explores new areas in the graph. p is called the return parameter, and after walking from node x_1 to x_2, the next vertex in the random walk will be x_1 again with (unnormalized) probability $\frac{1}{p}$. Nodes that are a neighbor of both x_1 and x_2 will be visited with (unnormalized) probability 1, and nodes that are only a neighbor of x_2 but not x_1 will be visited according to its exploration parameter q, with (unnormalized) probability $\frac{1}{q}$. Through these two parameters one is able to control whether the random walk looks more similar to breadth-first search (i.e. when the walk stays close to the neighborhood of the source node), or whether it starts to emulate a depth-first search (when the node starts to explore parts of the graph farther away from the source node). It is worth noting that setting p and q both equal to 1 makes node2vec equivalent to DeepWalk.

We used the implementation provided by the node2vec authors in order to compare its performance with the method we considered. As in DeepWalk, we fixed the length of the random walk and the number of random walk sequences per node each to be 5, in order to align with our results in the method. We set the context window to be 5 so that each node would predict the

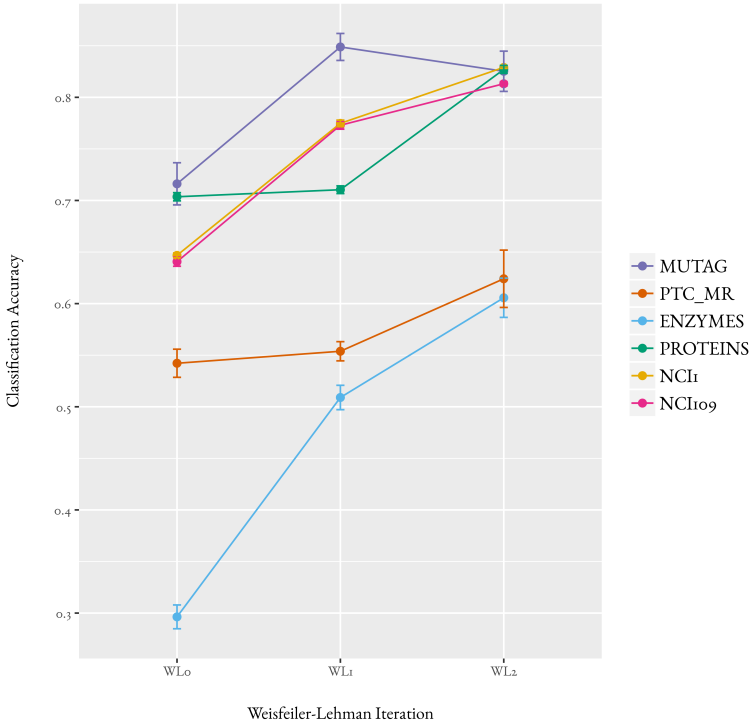


Figure 5.8: Classification accuracy of NODE2VEC by WL Iteration, with $p = 2$ and $q = 0.5$.

context of its entire surrounding sequence. The dimension of the learned representation was 100, in order to keep parity with the method we are comparing it to. To find a vector representation of each graph, we simply averaged the learned node embeddings over all nodes present in the graph, and fit an SVM in the exact same way as done in the main method for best comparability. We did this for all comparable iterations of Weisfeiler-Lehman.

Results

We report the results of NODE2VEC in Table 5.10 with parameters p and q chosen from a search space of $p, q \in \{0.5, 2\}$, visualized also in Figure 5.8. Although there were minor differences in the model performance based on the choices of p and q, the biggest changes in performance came

Table 5.10: Classification accuracies by datasets & WL iteration using NODE2VEC with $p = 2$ and $q = 0.5$.

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
WL0	71.61 (± 0.2)	54.21 (± 1.4)	29.63 (± 1.2)	70.36 (± 0.4)	64.68 (± 0.2)	64.07 (± 0.4)
WL1	**84.88 (± 1.3)**	55.38 (± 0.9)	50.90 (± 1.2)	71.04 (± 0.4)	77.48 (± 0.3)	77.29 (± 0.4)
WL2	82.52 (± 2.0)	**62.41 (± 2.8)**	**60.57 (± 1.9)**	**82.70 (± 0.4)**	**82.90 (± 0.1)**	**81.32 (± 0.2)**

from the chosen iteration of the Weisfeiler-Lehman algorithm, and thus present a representative result using $p = 2$ and $q = 0.5$. For WL0, we observe lower values compared to our main method. WL1 makes notable improvements over WL0, as we also observed in the Taheri et al. method, but yields results that are often lower than the WL1 counterpart in the Taheri et al. method. MUTAG, NCI1 and NCI109 yield results close to the original method, but PTC_MR, ENZYMES and PROTEINS are many percentage points worse. As in DeepWalk, we observe strongest performance in WL2 (except MUTAG, where it decreases slightly), and this performance performs well in comparison to Taheri et al. We note in particular excellent results for PROTEINS, achieving 82.70% classification accuracy for WL2.

5.4.3 Comparison to the Main Method

When we compare the best candidate model from each method, we observe largely similar results in terms of classification accuracy. However, whereas Taheri et al.'s method performs best with the first iteration of WL, we found that the node embedding methods performed best for the second iteration of WL. We therefore take the best of each method in order to compare. For the main method, our best results across datasets came from Model 2 (S2S-AE-PP), using shortest path for sequence generation, and WL1 for node labels. For DeepWalk, the best performance came from WL2, and for node2vec, we found best performance for WL2 using the parameters $p = 2$ and $q = 0.5$. When we compare the best from each method, we observe largely comparable performance between the graph embedding based on a sequence embedding, and the graph embeddings based off of a node embedding. Indeed, we even observe stronger performance classifying the datasets using node2vec on all datasets except MUTAG, and very similar performance on NCI109. In particular, node2vec shows much stronger performance on PROTEINS, outperforming not only this comparison, but superceding all results by any model in the Taheri et al. implementation by 6 percentage points.

Taheri et al. did include a comparison to node2vec in their paper, but reported much lower results than what we obtained. We both used the original implementation of the authors of node2vec, and we believe they made the same decision to represent the graph as the average of the nodes, as that is the most intuitive way to represent the graph from its nodes, since taking the mean over the sequences was how the original authors chose to represent a graph. However, we suspect they only ran the model on the original node labels (WL0), as their results correspond very closely with the results we obtained when we ran it with WL0. However, this does not seem to be a valid comparison of their model to node2vec, since in their own model they tested results across different results of WL and ultimately reported on their strongest performing iteration (WL1). We assume this is an oversight of the authors, and show what we consider to be a fair comparison in Figure 5.9.

While the method proposed by Taheri et al. offers a new way to represent graph representations using sequences, we can also observe that we are able to find largely similar graph classification performance simply by learning the node embeddings of a graph, and averaging over an entire graph. We therefore conclude that this is a new way of solving the graph classification problem, but that it achieves comparable results to the methods based on node embeddings.

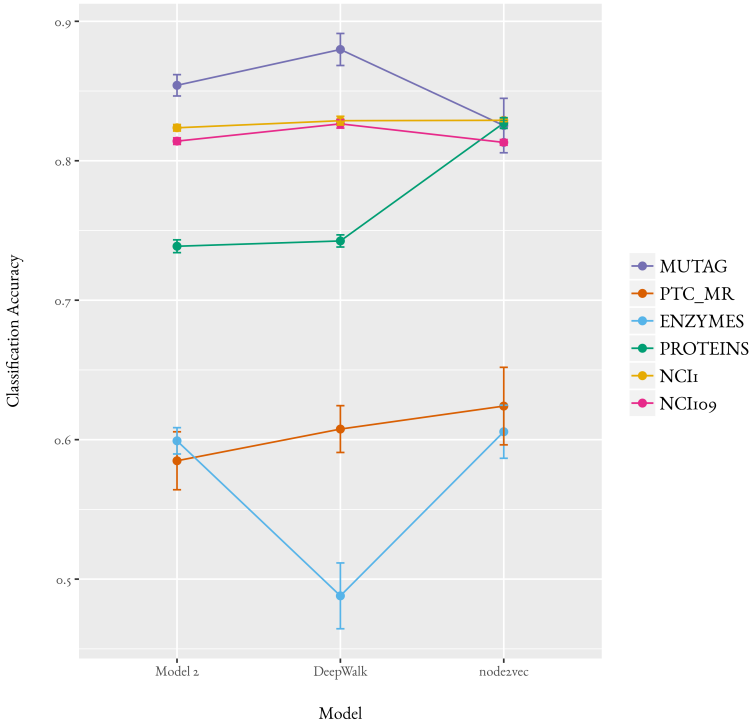


Figure 5.9: Comparison of classification accuracies using our best model (Model 2, WL1, SP), and the best models from DeepWalk (WL2), and node2vec (WL2, $p = 2$, $q = 0.5$.)

5.5 Summary

In this chapter we provided the specifics of our implementation of the Taheri et al. method, including any decisions made in the process to facilitate reproducibility. We showcased the results that we found from the various model configurations, and compared the performance to that of the method's authors. While performance is in large part comparable, with slightly lower results likely due to differences in parameter choices, we observed two peculiarities not mentioned by the authors. First, we observed that random walks did not consistently work well across all iterations of WL and datasets, which could be attributable to a bit of tottering we observed, or perhaps a slight difference in how random walks were generated. Second, the method already performs quite well without training the network at all, which suggests that much of the performance comes from pushing the data through the various transformations in the network, rather than through rigorous learning of the network itself.

This led us to wonder whether this approach improves upon the other established methods for learning representations based on sequences. As such, we compared our results to the results of the established node embedding methods that are based on sequences, DeepWalk and node2vec, and despite some individual differences, found largely similar performance, and almost uniformly better performance when using node2vec. We therefore conclude that while this method pro-

poses a new way of classifying a graph, it does not achieve considerable gains over the previously established methods based on learning node embeddings of graphs.

Nevertheless, Taheri et al. provides another angle from which to solve the graph classification problem. Since we've established a baseline performance of the method, we will now consider a few alternations to learn more about the method's behavior, and if possible, try to improve upon its results.

6 Extensions & Discussion

With a working implementation of the original approach, we will investigate a more thorough understanding of the method by testing alternatives to try and extend it. We will consider three main experiment ideas; first, to represent the graph as more than the mere mean of each dimension in its learned representation from the autoencoder. Then, we will try a second experiment, where we swap the SVM for a neural network, to keep the approach entirely within the realm of neural networks. Finally, we test whether we can improve the approach by predicting the class of each sequence directly in the original neural network, and then applying a majority vote over all sequences. For each, we will begin with a brief description of the experiment setup, present the results, and discuss our conclusions.

6.1 Experiment 1: More Informative Vector Representation

In the Taheri et al. implementation, the mean of the vector representations over all sequences in a graph is taken in order to have a single vector representing each graph. The authors briefly commented that the mean was chosen as it outperformed max pooling. We seek to investigate whether performance can be improved by using a different, more informative metric than just the mean. Our hypothesis was that the mean, while suitable, constitutes only a single measure, and that we could achieve better results by incorporating more information about our learned vector representation into our graph embedding function.

While we initially intended to simply replace the mean and keep the rest of the method identical to the original approach, it quickly became clear that we would be in a high dimensional setting with more features than training data points in some of the datasets. For example, MUTAG has only 188 graphs in its dataset, so its 100-dimensional vector representation from Taheri et al. becomes 400-dimensional if we include the mean, median, sum and standard deviation. This lowered performance on the smaller datasets, since we were overfitting to our training data. We thus swapped the SVM for a Random Forest [3] using `sklearn's RandomForestClassifier`. While this would handle the issue of high dimensionality for the smaller datasets, it also adds the benefit of reducing the computational complexity of the approach from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$, where N is the number of graphs in the dataset, since we no longer need the kernel computation in the SVM. The original authors tout one of the benefits of their approach is that it is linear in the number of graphs, but failed to account for the quadratic computation required by the kernel in the SVM. Replacing the SVM with a Random Forest would reduce this complexity, and greatly speed up training. Thus we expanded our investigation to two questions. First, can a Random Forest successfully replace the SVM in the original method and achieve comparable results? And second, can we improve performance by incorporating more distributional information in our vector representation of a graph?

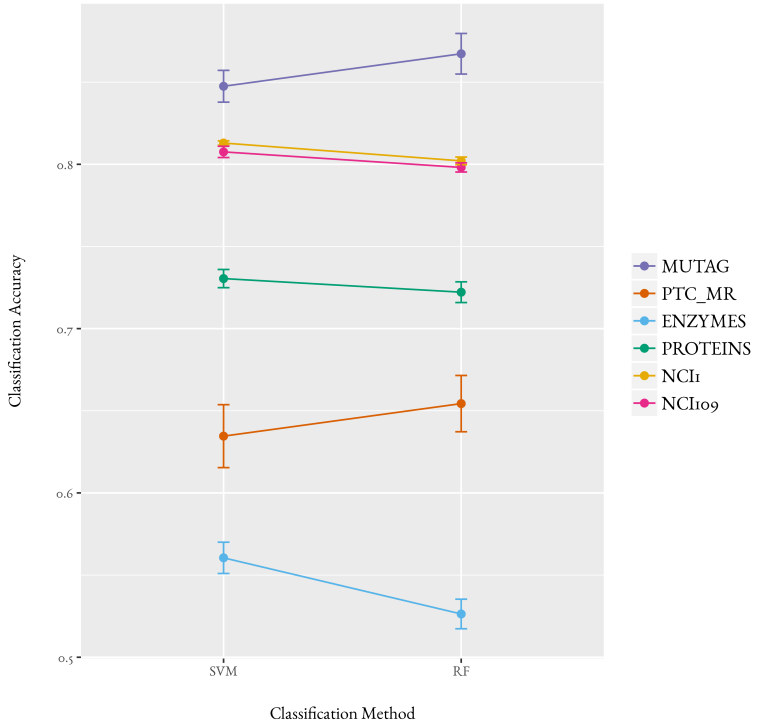


Figure 6.1: Classification Accuracy from Experiment 1 using the mean of the learned vector representations and either the SVM (the baseline performance) or a Random Forest.

We used Model 2 (S2S-AE-PP) as our baseline method to learn vector representations for each graph, since it had decent performance across all datasets. We set up exactly as was done in the original implementation with the same parameters and learning over a single epoch. We chose the specific configuration of BFS sequences using node labels generated by iteration 1 of the Weisfeiler-Lehman algorithm (WL1).

We fit a Random Forest with 500 trees, used the Gini criterion for our loss function, and placed no restrictions on the depth of the tree that can be grown. We used `class_weights` to balance the loss in our unbalanced datasets. We then trained the random forest on repeated 10-fold cross validation (repeated 10 times) to estimate the classification accuracy. We first compared the Random Forest performance using just the mean, to answer our first question. Then, we considered the following combinations of summary statistics to replace the original graph embedding function (the mean), each of these taken over all sequences in a graph, for each individual dimension in the learned vector representation:

1. Mean and Standard Deviation
2. Mean and Sum
3. Mean, Sum, Median, and Standard Deviation
4. Minimum, Mean, Median and Maximum
5. Deciles + Mean.

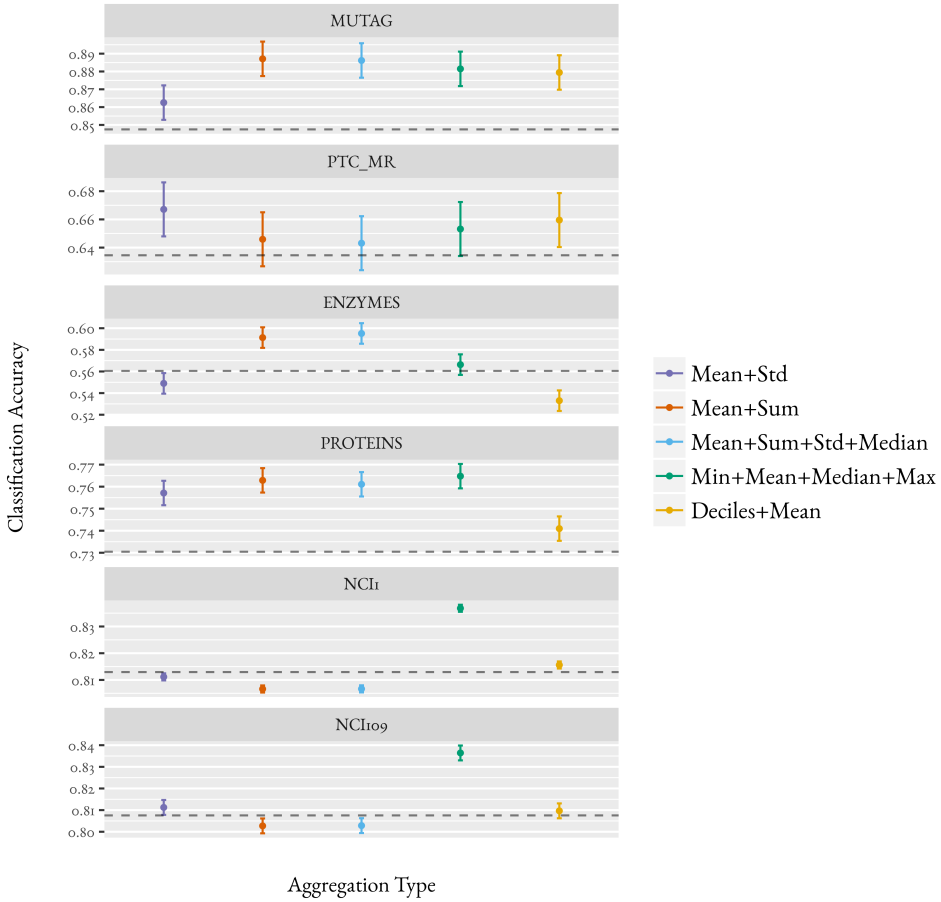


Figure 6.2: Classification Accuracy from Experiment 1 using additional summary statistics in the learned vector representation and a RF, compared to the baseline performance achieved by using the mean and an SVM (from Model 2, using BFS and WL1) of each dataset.

6.1.1 Results

At first glance it becomes clear that merely replacing the SVM with a Random Forest does give roughly comparable results. MUTAG and PTC_MR experience notable gains in performance from this single switch alone, and NCI1, NCI109, and PROTEINS have slightly lower results. ENZYMES is the only dataset that experiences a several percentage point decrease. Given the largely comparable performance, and the computational speed up, it appears this is a reasonable alternative to the SVM.

The results become more notable once we start considering the alternative graph embedding functions with the Random Forest. While all approaches improved performance across most of the datasets, two methods in particular had especially strong performance: mean & sum, and minimum, mean, median & maximum. Including the sum of the learned vector representation with the mean already seems to bring improved performance across all datasets, with the exception of

NCI1 and NCI109, where performance decreases slightly but is largely comparable. Combining the minimum, mean, median and maximum also improved performance across all datasets, and in particular was the only version that had notable results for NCI1 and NCI109. Since the minimum and maximum are not robust measures, we tried replacing these with the 10% and 90% respectively, and tried an additional time including the interquartile range as well. While these achieved decent results, its performance was not consistently better across all datasets, as it is with the minimum, mean, median, and maximum, so we do not include the results here.

6.2 Experiment 2: Using a Neural Network in Place of an SVM

After the findings from Experiment 1, we were interested to further assess how important the SVM is in this process, and more importantly, whether it really is fully replaceable in this method. This is again of interest due to the computational complexity of the SVM, as well as to understand whether it is an important driver in the performance. In this experiment, we take the learned graph representations that would normally be fed into an SVM and instead insert them into another neural network. Our goal is to understand whether we can replace the SVM with a neural network, to achieve the same or better results. If the results are not better, but comparable, this is still advantageous as the classification can then be done linearly in the number of graphs (versus quadratically due to the kernel computation in the SVM). The experiment setup is similar as before; we use the vector graph embedding learned from Model 2 (S2S-AE-PP) over a single epoch using the same model parameters as before. We feed those learned vector representations into a neural network to see if we can achieve similar classification performance.

We used a fully connected feed-forward network with four layers. Although there is plenty of network architecture search that can be done to further refine it, we choose a simple, straightforward architecture in order to assess the general viability of such an approach. As such, we built a fully connected network with three intermediate layers, with 400, 400, and 50 units respectively, and a final output layer equal to the number of classes in the dataset. All of the hidden layers

Table 6.1: Classification accuracies comparing the baseline model that uses the mean and an SVM (Model 2, WL1 BFS), to using the mean and a Random Forest, and to using additional summary statistics and a Random Forest. MSSM is mean, sum, standard deviation, and median, and MMMM is minimum, mean, median and maximum.

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
Model 2, WL1 BFS	84.75 (± 1.4)	63.46 (± 1.9)	56.05 (± 1.0)	73.05 (± 0.6)	81.29 (± 0.1)	80.76 (± 0.3)
Mean (with RF)	86.73 (± 1.2)	65.44 (± 1.7)	52.63 (± 0.9)	72.22 (± 0.6)	80.21 (± 0.2)	79.81 (± 0.3)
Mean+Std	86.25 (± 0.6)	**66.71 (± 1.8)**	54.90 (± 1.1)	75.71 (± 0.5)	81.11 (± 0.1)	81.12 (± 0.2)
Mean+Sum	**88.71 (± 0.9)**	64.59 (± 0.9)	59.13 (± 1.0)	76.29 (± 0.4)	80.66 (± 0.2)	80.27 (± 0.2)
MSSM	88.62 (± 1.0)	64.31 (± 1.6)	**59.52 (± 0.8)**	76.11 (± 0.4)	80.67 (± 0.2)	80.28 (± 0.2)
MMMM	88.15 (± 0.9)	65.32 (± 1.3)	56.63 (± 1.0)	**76.48 (± 0.5)**	**83.68 (± 0.2)**	**83.64 (± 0.2)**
Deciles+Mean	87.94 (± 0.8)	65.95 (± 1.1)	53.30 (± 0.6)	74.10 (± 0.5)	81.56 (± 0.2)	80.97 (± 0.2)

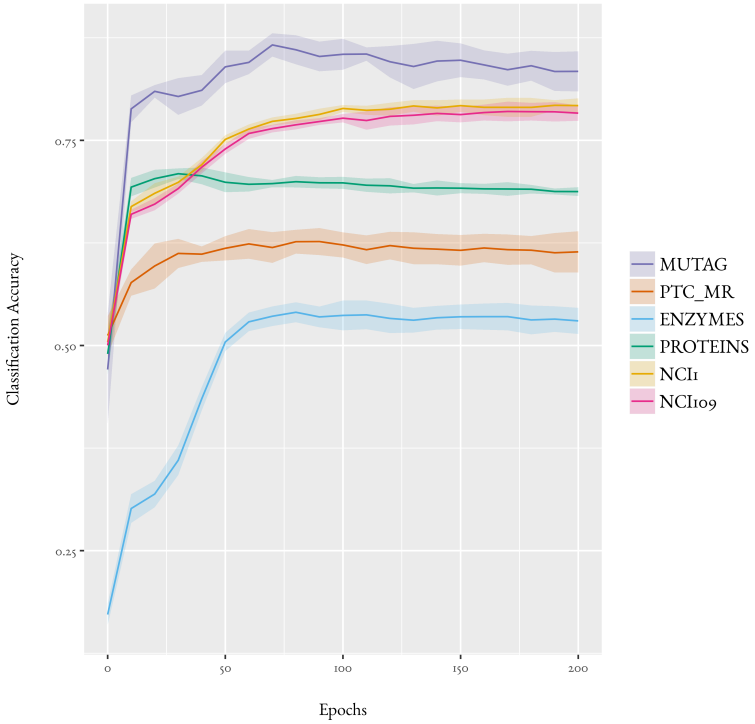


Figure 6.3: Classification Accuracy from Experiment 2 over an increasing number of epochs.

used batch normalization, and had a ReLU activation. The final output layer used the softmax activation function in order to return the probabilities of class membership. The network was trained using the AdaGrad optimizer and using the cross entropy loss function. The other parameters (initialization of weights, learning rate, etc.) have been held constant from the original method in order to keep a more streamlined implementation of the neural network. As in Experiment 1, we used the vector representations learned from BFS sequences of the first iteration of the Weisfeiler-Lehman algorithm.

Table 6.2: Classification accuracy using a simple feed-forward neural network in place of the SVM, compared to the baseline performance of Model 2 (BFS, WL1).

Method	**MUTAG**	**PTC_MR**	**ENZYMES**	**PROTEINS**	**NCI1**	**NCI109**
Model 2, WL1 BFS	84.75 (± 1.4)	**63.46 (± 1.9)**	**56.05 (± 1.0)**	**73.05 (± 0.6)**	**81.29 (± 0.1)**	**80.76 (± 0.3)**
Experiment 2	**85.49 (± 1.9)**	62.25 (± 1.5)	53.67 (± 1.8)	69.82 (± 0.7)	78.89 (± 0.4)	77.70 (± 0.5)

6.2.1 Results

We trained the network for 100 epochs, as increasing the training over epochs brought an increase in performance, as Figure 6.3 shows. (As a side node, we train over multiple epochs here since this is analogous to the training that the SVM does. We trained over a single epoch in the LSTM which generated the vector representation in both scenarios. For good measure, we checked to see if training over more epochs would improve the baseline performance, but given the parameters provided by the paper, it does not improve. Decreasing the learning rate does lead to improvements, but we do not publish those results here).

This method also achieves slightly lower performance than the original method. Given this result was achieved with minimal architecture search, it seems feasible to further improve the performance to be comparable with a more extensive parameter tuning. In either case, achieving roughly comparable performance is still interesting as it lends additional evidence to the idea that the SVM is a replaceable component of this approach, and more importantly, allows the method to scale linearly in the number of graphs.

6.3 Experiment 3: Using a Neural Network to Predict the Class Directly

In our final experiment, we aimed to understand whether we could stay entirely in the world of neural networks, but simplify the approach to have a single neural network instead of two, as in Experiment 2. We kept the same basic step of decomposing the graphs into a set of sequences, and then used a LSTM to encode the sequences into a fixed dimensional vector. Then, we connected this fixed size vector to a few layers of a fully connected network, and predicted the class of each sequence. A majority voting scheme was applied over all sequences in a graph in order to perform the classification at the graph level.

As above, we generated sequences via BFS and the first iteration of Weisfeiler-Lehman, and used the encoder from Model 2 (S2S-AE-PP) for the first portion of the network, which would encode the sequence into a fixed dimensional vector. This vector was then passed to the second portion of the network, which was had 3 intermediate fully-connected layers with 400, 400, and 50 hidden units, as above, and a softmax computation on the final output layer (sized equivalent to the number of classes) to predict class membership. We again used batch normalization and ReLU activations on the hidden layers. We then classified a graph using the majority vote over all sequences in the graph. We also considered max probability in place of majority voting, but the

Table 6.3: Classification accuracy using a single LSTM combined with a feed-forward neural network to predict the class of each sequence, which is then translated to the graph level by majority vote. Comparison to the baseline performance of Model 2 (BFS, WL1).

Method	MUTAG	PTC_MR	ENZYMES	PROTEINS	NCI1	NCI109
Model 2, WL1 BFS	**84.75 (± 1.4)**	**63.46 (± 1.9)**	**56.05 (± 1.0)**	**73.05 (± 0.6)**	**81.29 (± 0.1)**	**80.76 (± 0.3)**
Experiment 3	80.27 (± 2.7)	53.39 (± 2.9)	47.43 (± 2.0)	62.76 (± 1.4)	67.19 (± 1.5)	65.81 (± 1.9)

results were largely similar. We trained the network using repeated stratified 10-fold cross validation (repeated 10 times), making the stratification at the graph level (as opposed to the sequence level).

6.3.1 Results

The results from this experiment yielded results much lower than the other experiments, as well as lower results than the baseline method considered, across all datasets. However, upon investigation it became understandable why the results were lower, and surfaced an informative structural limitation of this approach. There are two main factors underpinning the explanation, which build off of one another: a misaligned loss function, and a highly repeated, and heavily skewed distribution of sequences in the training data.

To explain the former, we look back into the design of the neural network. Our training input was sequences, and we then predicted the class of the sequence, which was later aggregated via majority voting to classify a graph. However, the loss function that the neural network was optimized on was based on whether the class of the sequence was predicted correctly, rather than the class of the graph. We therefore realized that this approach has limited potential due to the fact that the loss function the network is trained on is not the true loss function.

Our initial hypothesis was however that the sequences would provide a good proxy of class membership. However, upon revisiting the sequence frequencies in Table 5.2, we observed a few peculiarities that fundamentally pose problems for this approach. First, there is a lot of repetition in the sequences, with only 3% of sequences in NCI1 and NCI109 being unique (for our experiment setup, i.e. BFS sequences at WL1). MUTAG and PTC_MR are similarly low with only 5% and 17% of their sequences being unique. ENZYMES and PROTEINS are less dramatic, but still observe that only around half of their sequences are unique. Furthermore, the distribution of the unique sequences is very skewed. In ENZYMES and PROTEINS, the top 5% of unique sequences account for 30% and 37% of the training data respectively. It's more drastic in MUTAG and PTC_MR, with those 5% accounting for 68% and 51%, and extreme in NCI1 and NCI109, where they account for 77% of all sequences. This wouldn't necessarily be problematic if these sequences only occurred in one class, but we found that these frequently repeated sequences existed in both classes. Thus, at the end of training, the network will always predict a single class for those sequences, since their input is the same, and can never "learn" when one sequence occurs in one class and when it occurs in another.

This, combined with the loss function optimizing for correctly predicting the classes of the sequences, rather than the classes of the graph, presents a design flaw that limits how well the approach can perform. While the experiment yielded low results, it provided useful insight into how to design a network so that it can learn something, and perform better. More details will be discussed in Section 7.1.

6.4 Summary

We ran several experiments to deeper our understanding of the original method and to try and improve upon it. Experiment 1 tested whether we could replace the SVM with a Random Forest, and whether replacing the mean of the learned vector representation with more distributional

information could enhance performance. From this we learned two key things. First, we can successfully replace the SVM with an alternate method such as a Random Forest, and thus decrease the computational complexity from quadratic to linear in the number of graphs, and achieve comparable performance. This not only improves the run time of the approach, but also indicates that the predictive power of this method is not wholly contingent upon the SVM. Secondly, we found that we can improve performance a few percentage points by incorporating a more informative vector representation of the graph, and then classifying it with a Random Forest. Specifically, the combination of the minimum, mean, median & maximum of the vector exhibited strong improvements across all datasets.

In Experiment 2, we tested whether we could classify the graph using only neural networks, swapping the SVM for a second neural network. We found that performance is similar but slightly worse, which could likely be improved with a larger architecture search. This provides additional evidence to the idea that the SVM is a replaceable component of this approach's architecture. Furthermore, replacing the SVM with a feed-forward network once again reduces the complexity to be linear in the number of graphs, which offers a considerable speed up advantage.

Finally, Experiment 3 sought to simplify the approach to a single neural network that could take a sequence as input and output a class prediction, which could be aggregated for a graph using majority voting. This approach yielded lackluster results, due to the fact that the network is training on a loss function that does not correspond to our true loss function. We then realized that this fact, combined with the overlap and repetition of sequences in both classes, fundamentally limits how successful this specific approach can be.

These experiments started to learn more about this method and spawned new ideas on how to further improve it. We will now review the main conclusions of thesis and discuss other directions that future work can take from here.

7 Conclusion

As the amount of graph-structured data increases, the demand for methods that can compare and classify these graphs abounds. While there are several high-level approaches one can take to solve such an issue, all seek to find ways that balance achieving strong performance while being computationally efficient.

This thesis set out to understand and re-implement the method proposed by Taheri et al., which uses a recurrent neural network autoencoder to learn a vector representation of graphs for use in classification. Our goal was to re-implement it, understand the complexities and nuances of it, and then perform some experiments trying to extend it. From this, we learned several useful insights. First, in broad strokes we found it fair to say that the method works as the authors claimed. However, it is worth noting that part of the strength of the method itself is the baseline power of the LSTMs to separate the data. With random initialization and without training the network, the SVM was able to classify the graphs with strong baseline (if not state-of-the-art) performance. We also observed that random walks suffered a bit from tottering, and did not always perform consistently well, in comparison to the other sequence generation methods. This was most notable when using the original labels of Weisfeiler-Lehman, but improved when using labels from higher iterations of the algorithm.

Having established a baseline performance, we sought to understand the method's performance relative to the existing sequence-based methods. We conducted a comparison to the established methods DeepWalk and node2vec, which learn a node embedding of graphs based on sequences, and aggregated over all nodes in the graph in order to perform classification with a SVM in the same manner as outlined by Taheri et al. We found that the best performing models of DeepWalk and node2vec perform comparably to the best performing model of Taheri et al., and in some cases better.

As we moved into running experiments, we tested three hypotheses. The first questioned if a more informative vector representation could enhance the graph classification accuracy. After suffering from a high dimensionality problem, we switched the classifier from SVM to a Random Forest and discovered we could achieve very similar results through this switch alone. This is significant because replacing the SVM removes the quadratic complexity step, and replaces it with a linear alternative. It also suggests that the power of the method is not dependent entirely on the SVM. We then tested whether we could improve performance by using additional summary statistics in our vector summary. Although many combinations provided better results on most datasets, combining the minimum, mean, median and maximum produced gains across all, notably so on NCI1 and NCI109.

Next, we tested whether a neural network could replace the SVM, to reaffirm the findings from our first experiment that the SVM was a replaceable component of the method, and that it could be run in linear time. Additionally, this would keep our efforts within the domain of neural networks. Initial results showed decent, but slightly lower performance to the SVM. We suspect that

performance could reach full parity with a larger architecture search and parameter tuning, but we did not investigate it more deeply here.

Finally, we wondered if we could streamline the approach to involve a single neural network that could create a vector representation and perform classification. We began with an approach that would take the sequences in as in the original method and compress them to a fixed dimensional vector via the encoding, then predict the class label directly on the sequence via a few fully connected layers. We applied a majority voting scheme over the sequences to classify the graph. After lower performance results, we realized that the design of the network had some limitations for the task at hand. First, the loss function did not represent our true loss function. Second, the structure of the sequences (i.e. high repetition of sequences, skewed distribution of sequences, and sequences occurring in both classes), made learning very difficult for the network. Thus, we were not surprised this method did not yield better results. We concluded that in order for the neural network to be successful in classifying the graph, it would be more effective to design a network that can classify the graph directly, so that the loss function is aligned with the true loss.

7.1 Future Work

Through our implementation of the main method, and the results of our experiments, we discovered many additional experiments and directions to continue in from here. The first, and most immediate recommendation builds upon the findings from Experiment 3, which although it did not provide strong results, provided insight into what architectures would be better suited in this setup. The ideal architecture in this case would take an entire graph as a single training point, and thereby perform the classification on the graph directly. This would provide a loss function directly tied to our objective. The challenge here is to incorporate two sources of variable length input, as different graphs in the dataset can have a different number of sequences representing it, and within a single graph, the sequences can be of different length. We envision a kind of nested LSTM structure to accomplish this, with an outer LSTM to read in a varying number of sequences per graph and convert it to a vector, and an inner LSTM that reads the individual sequences of varying length and compresses it to a vector, that is then fed to the outer LSTM. This problem can also be formulated in the context of *Set Function Learning*.

Another area to consider, which can also nest into the previous suggestion, would be to pretrain the node embeddings. Given the strong performance we observed with the methods DeepWalk and node2vec, one could use those methods to first learn a node embedding of the vertices, and then use that pretrained embedding. Currently the approach tries to learn the embedding of the vertex simultaneously with learning the vector representation of the sequence, but as we observed in Section 5.3.2, the learning of the network does not add substantial improvements over random initialization. It is possible that the performance could improve if the vertices had a meaningful embedding derived through a process explicitly designed for that purpose.

Finally, there is still plenty of additional experimentation and ablation studies that could be conducted on the method itself, to more fully understand which components drive the performance. In this vein, one could run further ablation studies to identify the critical components of the neural network, or consider alternative vertex labeling schemes to investigate the importance of the Weisfeiler-Lehman relabeling versus other relabeling methods.

Bibliography

[1] Borgwardt, K. M. and H.-P. Kriegel (2005). Shortest-path kernels on graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, Washington, DC, USA, pp. 74–81. IEEE Computer Society.

[2] Borgwardt, K. M., C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel (2005). Protein function prediction via graph kernels. *Bioinformatics 21*(suppl 1), i47–i56.

[3] Breiman, L. (2001). Random forests. *Machine Learning 45*(1), 5–32.

[4] Cai, C. and Y. Wang (2018). A simple yet effective baseline for non-attribute graph classification. *CoRR abs/1811.03508*.

[5] Debnath, A., R. L. Lopez de Compadre, G. Debnath, A. Shusterman, and C. Hansch (1991). Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry 34*, pp. 786–797.

[6] Duchi, J., E. Hazan, and Y. Singer (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research 12*, 2121–2159.

[7] Duvenaud, D., D. Maclaurin, J. Aguilera-Iparraguirre, R. Gómez-Bombarelli, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams (2015). Convolutional networks on graphs for learning molecular fingerprints. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'15, pp. 2224–2232. MIT Press.

[8] Floyd, R. W. (1962). Algorithm 97: Shortest path. *Communications of the ACM 5*(6), 345.

[9] Gers, F. A. and J. Schmidhuber (2000). Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, Volume 3, pp. 189–194.

[10] Graves, A. (2012). *Supervised Sequence Labelling with Recurrent Neural Networks*, Volume 385 of *Studies in Computational Intelligence*. Springer.

[11] Grover, A. and J. Leskovec (2016). node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[12] Gärtner, T., P. Flach, and S. Wrobel (2003). On graph kernels: Hardness results and efficient alternatives. In *Proceedings of the 16th Annual Conference on Computational Learning Theory and the 7th Kernel Workshop*, pp. 129–143.

[13] Hastie, T., R. Tibshirani, and J. H. Friedman (2009). *The Elements of Statistical learning: Data Mining, Inference, and Prediction, 2nd Edition*. Springer series in statistics. Springer.

[14] Haussler, D. (1999). Convolution kernels on discrete structures. Technical report, Department of Computer Science, University of California.

[15] Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural Computation 9*, 1735–80.

[16] Jin, Y. and J. F. JáJá (2018). Learning graph-level representations with gated recurrent neural networks. *CoRR abs/1805.07683*.

[17] Li, J., T. Luong, and D. Jurafsky (2015). A hierarchical neural autoencoder for paragraphs and documents. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Beijing, China, pp. 1106–1115. Association for Computational Linguistics.

[18] Mahé, P., N. Ueda, T. Akutsu, J.-L. Perret, and J.-P. Vert (2004). Extensions of marginalized graph kernels. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, New York, NY, USA, pp. 70–. ACM.

[19] Mikolov, T., I. Sutskever, K. Chen, G. Corrado, and J. Dean (2013). Distributed representations of words and phrases and their compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'13, USA, pp. 3111–3119. Curran Associates Inc.

[20] Narayanan, A., M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal (2017). graph2vec: Learning distributed representations of graphs. In *MLG*.

[21] Niepert, M., M. Ahmed, and K. Kutzkov (2016). Learning convolutional neural networks for graphs. In *International Conference on Machine Learning (ICML)*.

[22] Olah, C. (2015). Understanding LSTM networks.

[23] Perozzi, B., R. Al-Rfou, and S. Skiena (2014). Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '14, New York, NY, USA, pp. 701–710. ACM.

[24] Salaspuro, M. (2009). Acetaldehyde as a common denominator and cumulative carcinogen in digestive tract cancers. *Scandinavian journal of gastroenterology 44*, 912–25.

[25] Schomburg, I., A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, and D. Schomburg (2004). BRENDA, the enzyme database: Updates and major new developments. *Nucleic acids research 32*, D431–3.

[26] Shervashidze, N., P. Schweitzer, E. J. van Leeuwen, K. Mehlhorn, and K. M. Borgwardt (2011). Weisfeiler-Lehman graph kernels. *Journal of Machine Learning Research 12*, 2539–2561.

[27] Shervashidze, N., S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt (2009). Efficient graphlet kernels for large graph comparison. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, Volume 5 of *Proceedings of Machine Learning Research*, pp. 488–495. PMLR.

[28] Sutskever, I., O. Vinyals, and Q. V. Le (2014). Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, Cambridge, MA, USA, pp. 3104–3112. MIT Press.

[29] Taheri, A., K. Gimpel, and T. Berger-Wolf (2018). Learning graph representations with recurrent neural network autoencoders. In *KDD Deep Learning Day*.

[30] Toivonen, H., A. Srinivasan, R. D. King, and S. Kramer (2003). Statistical evaluation of the predictive toxicology challenge 2000-2001.

[31] Wale, N. and G. Karypis (2006). Comparison of descriptor spaces for chemical compound retrieval and classification. In *Sixth International Conference on Data Mining (ICDM'06)*, pp. 678–689.

A Source Code Description

We provide an overview here of the directories and files accompanying this thesis. In the provided code, we have the implementation for the Taheri et al. method and our accompanying experiments in the `graph-rnn-ae` folder. This contains all datasets, source code, and the results published in this thesis. Additionally, we are including the implementations of DeepWalk[1] and node2vec[2] respectively in the `deepwalk` and `node2vec` folders. These include slight adaptations done to the implementation in order to aggregate the node embeddings over a graph and perform classification with an SVM on the datasets we are interested in. However, since most of the code is provided by the papers' authors, we refer the reader to their implementation documentation for further details.

A.1 graph-rnn-ae File Descriptions

config.py	Contains additional global parameters to use.
decoder.py	Contains the class for the decoder layer of the network.
embedding.py	Generates the WL labels (for a given iteration) for a dataset as well as the vertex embeddings for a set of node labels.
encoder.py	Contains the class for the encoder layer of the network, which is also used as the parent class for the decoder models.
euler.sh	Shell script to set often changed parameters and run the model.
model.py	Contains the class for the models.
prepare_datasets.py	Reads in the graph data and returns the adjacency matrices, graph labels, and node labels.
sequences.py	Generates the input sequences for the neural network.
svm.py	Classifies the learned graph representations using an SVM.
train.py	Runs the training of the 1st and 2nd model (S2S-AE, S2S-AE-PP).
train_m3.py	Runs the training of the 3rd model (S2S-AE-PP-WL1,2).
train_m4.py	Runs the training of the 4th model (S2S-N2N-PP).
train_m5.py	Runs the training of Experiment 2. This uses the data from the folder pretrained-data.
train_m6.py	Runs the training of Experiment 3.
utils.py	Contains helper functions.

[1] https://github.com/phanein/deepwalk
[2] https://github.com/aditya-grover/node2vec

A.2 Instructions to Run

To run the models implementing Taheri et al., execute the `euler.sh` script in the terminal. The default setting is to run the second model (S2S-AE-PP) using BFS for sequence generation and WL1 for node labels.

Experiment 1 can be run by setting the value of the parameter `RF=True` in `utils.py`, and then setting the `MODEL_NUMBER=2, METHOD=BFS, WL=1` in `euler.sh`. Experiment 2 can be run by setting `MODEL_NUMBER=2` in `euler.sh`, and Experiment 3 can be run by setting `MODEL_NUMBER=6` there.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

LEARNING VECTOR REPRESENTATIONS OF GRAPHS USING RECURRENT NEURAL NETWORK AUTOENCODERS

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
O'BRAY	LESLIE

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, Switzerland 06.09.2019	Leslie O'Bray

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.