

**COMPARATIVE CASE STUDY BETWEEN GATED GRAPH NEURAL
NETWORKS
VERSUS
RELATIONAL GRAPH CONVOLUTIONAL NETWORKS FOR THE VARIABLE
MISUSE TASK
USING PYTHON AND DEEP LEARNING**

By

Mukund Raghav Sharma

A Capstone Project Paper Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

in

Data Science

University of Wisconsin – La Crosse

La Crosse, Wisconsin

May 2020

DEDICATION

To my grandparents and my favorite planet, Saturn.

ABSTRACT

COMPARATIVE CASE STUDY BETWEEN GATED GRAPH NEURAL NETWORKS
VERSUS
RELATIONAL GRAPH CONVOLUTIONAL NETWORKS FOR THE VARIABLE
MISUSE TASK
USING PYTHON AND DEEP LEARNING

MUKUND RAGHAV SHARMA
MASTER OF SCIENCE IN DATA SCIENCE
UNIVERSITY OF WISCONSIN – LA CROSSE, 2020

Identifying bugs in source code has been an extremely important part of software development since the inception of the industry. The majority of static analysis, the analysis of software without actually executing programs, is rule based without much involvement of deep learning until fairly recently.

This paper engages in a comparative study of determining the more performant graph neural network model on the basis of test accuracy between Gated Graph Neural Network (GGNN) models and Relational Graph Convolutional Network (RGCN) models on the Variable Misuse Task, a prediction task involving choosing the correct variable based on all the variables of the same type in a particular scope.

The data is of source code from the files of 25 trending C# repositories that are converted into a modified Abstract Syntax Tree to represent a directed graph whose vertices that represent the tokens and relationships between the tokens are represented by edges. Each of these vertices are associated with one of the aforementioned type of networks for the training phase after a particular embedding is computed for each token.

The comparison to decide the more efficient model is based on the test accuracy of all the repositories, an esoteric repository and an extremely popular repository to cover the spectrum of different types of repositories. The results show that the RGCN based models outperformed the GGNN models for all cases, albeit, within $< 5\%$ range.

Keywords: Deep Learning, Graph Neural Networks, Tensorflow, Sequence Models, Convolutional Models, Learning from Code, Static Analysis.

TABLE OF CONTENTS

DEDICATION.....	iii
ABSTRACT.....	iv
TABLE OF CONTENTS.....	v
LIST OF FIGURES.....	vii
CHAPTER 1 – INTRODUCTION.....	1
Project Background.....	1
Objectives.....	3
Rationale and Inspiration for Undertaking the Project.....	4
CHAPTER 2 – LITERATURE OVERVIEW.....	6
Previous Work.....	6
Other Presentations.....	10
CHAPTER 3 – PROJECT OVERVIEW.....	11
General Approach / Layout.....	11
CHAPTER 4 – GRAPH NEURAL NETWORK 101.....	13
Goals.....	13
The Var Misuse Task.....	14
Graph Neural Networks.....	17
Gated Graph Neural Networks.....	27
Relational Graph Convolutional Neural Networks.....	29
CHAPTER 5: PRELIMINARY STEPS IN CASE STUDY.....	31
Problem Definition.....	31
Interview Notes.....	32

CHAPTER 6: SOURCE DATA EXPLAINED.....	34
Introduction.....	34
Tokens.....	40
Type Hierarchy.....	40
Structure of the Typical Graph File	41
Other Data Sources.....	42
 CHAPTER 7: METHODOLOGY AND RESULTS.....	 44
Introduction.....	44
Experimental Design.....	45
Details Related to the Pipeline.....	50
Conducting the Experiments.....	59
Results.....	64
Next Possible Steps.....	65
 CHAPTER 8: CONCLUSION.....	 67
 REFERENCES.....	 69

LIST OF FIGURES

Fig. 1: Timeline of Graph Neural Networks.....	6
Fig. 2: Example 1 of Application of the VarMisuse Task in C#.....	14
Fig. 3: Example 2 of Application of the VarMisuse Task in C#.....	16
Fig. 4: An Example of a Graph.....	18
Fig. 5: Simple Abstract Syntax Tree.....	20
Fig. 6: Modified Abstract Syntax Tree that Represent Programs.....	21
Fig. 7: Graph Structure and Messages For Each Node.....	24
Fig. 8: Message Passing And Aggregation by Nodes via a Recurrent Unit.....	25
Fig. 9: Unrolled Effect Of the Graph Where Nodes Become Positionally Aware	26
Fig. 10: Formula for the GGNN State Update Per Node.....	28
Fig. 11: Formula for the RGCN Node State Update Per Node.....	29
Fig. 12: Data Transformation Pipeline.....	35
Fig. 13: Example Representation of the Next Token Of the Directed Graph.....	35
Fig. 14: Representing the LastWrite in the Source Code As a Graph.....	36
Fig. 15: Listing of all the Directories From the Graph Data Set.....	38
Fig. 16: Typical Directory Structure of Data From A Repository.....	39
Fig. 17: Example of a Record of the RepositoryInfo.....	43
Fig. 18: All Repositories Considered.....	46
Fig. 19: CommonMark.NET Details.....	48

Fig. 20: Dapper Details.....	49
Fig. 21: List of Models.....	52
Fig. 21: List of Gated Graph Neural Networks.....	53
Fig. 23: List of Tasks And Hyperparameters.....	54
Fig. 24: List of Scripts.....	55
Fig. 25: List of Utilities.....	56
Fig. 26: Mockup of the Pipeline.....	58
Fig. 27: Data Needed For Each of the Experiments.....	59
Fig. 28: Training Command Line Args.....	60
Fig. 29: Command Line Output For Training.....	61
Fig. 30: Testing Command Line Arguments.....	62
Fig. 31: Testing Terminal Output.....	63
Fig. 32: Final Results.....	64

CHAPTER 1

INTRODUCTION

Project Background

Since the inception of the computing industry more than 70 years ago, the need for correct and efficient program verification tools has been extremely desirous in the software engineering world. Therefore, constant improvements are continually made to improve developer productivity in the field of static analysis tools or tools that analyze the code without executing the program by highlighting bugs at development time rather than during when the code is running in production.

Deep learning application in the field of learning from source code and by extension, static analysis, is still in its nascent phase and a large part of the current work doesn't take advantage of the representational power of both the syntactic and semantic nature of source code. For example, shallow representations of source code are prevalent in recent research such as incorporating a simple sequence of tokens as described by research from Hindle et al. (2012) or flat dependency networks of variables by Raychev et al. (2015).

Recent work by Allamanis et al. (2018) involves incorporating both the semantic and syntactic nature of source code by representing programs as directed graphs and applying deep learning to create graph neural network models. Using graph neural network models has proved to, more easily, solve state-of-the-art problems in the space of learning from code. The reason for the success of graph neural network models is attributed to the representational capacity of these models that reduce the need for a large training set as well as the fact they meticulously encapsulate the semantic relationships between variables and types in the source code.

In this paper, I further build on work by Allamanis et al. (2018) in the field of Graph Neural Networks to conduct a comparative study between two models namely, Gated Graph Neural Networks (GGNN) and Relational Graph Convolutional Networks (RGCN), on the Variable Misuse Task to discern the better performing model on the basis of test accuracy. The Variable Misuse Task is a simple, yet, important prediction based task on source code involving predicting the correct variable that accurately fits a particular spot in the code by considering all the variables of the same type in the same scope.

Microsoft Research's application of Graph Neural Networks on the Variable Misuse task has caught bugs that had been deployed in production for important repositories such as RavenDB and Roslyn as per Allamanis et al. (2018). Further application of Graph Neural

Networks on newer tasks show great promise and have already started changing the way code is tested and validated and would prove to supplement the rule based approach currently employed by most static analysis tools.

The input data used for this model is that of the top 25 trending C# repositories on Github. There are three main experiments that will be described conducted to decide the better performing model on the basis of test accuracy. The rationale here was to gain insight about how the GGNN and RGCN models would fare across different types of repositories and which model out of the two would be the better predictor.

Objectives

The main objective of this project is to compute the test accuracies for the GGNN and RGCN models for three experiments namely, training and testing on data from 25 of the trending C# repositories on Github, an esoteric repository and a popular repository, in an effort to discern the more performant model.

This process of computing the test accuracies involved enumerating through some minor, more granular, objectives and these are:

1. Obtaining, Cleaning and Understanding the Input Data.
2. Choosing, from the top 25 trending C# repositories, which one would characterize as the esoteric and popular one to conduct our experiments on.
3. Establishing a working pipeline through which the experiments will be conducted.
4. Interpreting results i.e. test accuracy of the experiments to land on a conclusion.

Rationale and Inspiration for Undertaking the Project

The constant strive to build better development tools that will improve the overall user experience is one I am starting to familiarize myself with more since I joined Microsoft's Developer Division as a software engineer where I work on the Performance and Reliability of Visual Studio. My role involves creating tooling and processes that improve the prevention of performance regressions by early detection i.e. before they reach the end user. My experience with performance engineering and testing on a large code base coupled with my strong inclination towards deep learning got me going down a path to choose a topic for my capstone that would fit the said intersection.

After discovering Microsoft Research's ground breaking work, I was convinced I needed to know the ins and outs of the details of how they improved state-of-the-art research models. And as a result, to add more specificity to my topic, I decided to undertake a

comparison study between two involved models, GGNN and RGCN on an important task to learn more about advanced sequence models as well as applying convolutional networks to graphs. I was lucky enough to have all my questions answered within a couple of hours by the good people at Microsoft Research and this was another inspiration booster i.e. to work on ground breaking work with individuals who were extremely eager to help out in improving the understanding of their work.

Exploring the source code of top 25 trending C# repositories made the data exploration phase all the more exciting as some of these repositories I have used professionally; this provided me a reason to dive deep into the code and observe the models make use of the source code.

CHAPTER 2

LITERATURE REVIEW

After defining the problem, in the field of Data Science it is important to understand what solutions already exist out there and how the field has evolved over time. In this chapter, I'll be going over important developments in the field of Graph Neural Networks and how it ties into my comparative study.

Previous Work

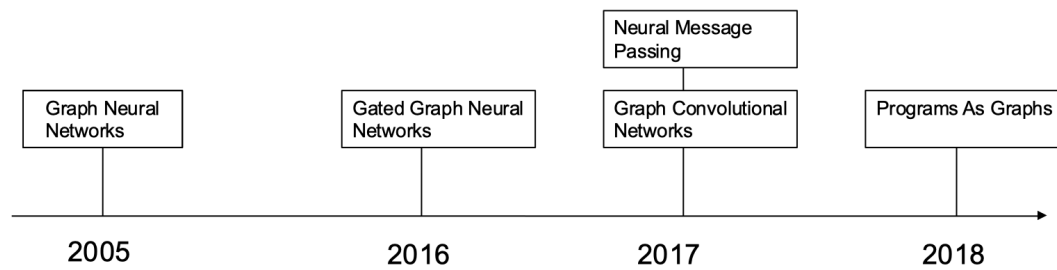


Fig. 1: Timeline of Graph Neural Networks Research

Researching through a lot of the related past research papers in the field of Graph Neural Networks, I was able to narrow down the timeline of key developments details of which are as follows:

2005: Gori et al. in their work “The Graph Neural Network Model” proposed the notion of Graph Neural Networks in the seminal research paper of this field that laid down the foundation of future work. Although, deep learning wasn’t at the forefront of statistical research when this paper was written as it is now, important fundamental concepts such as training of graph neural networks as well as the concept of message passing emerged.

2016: Li et al. in their work “Gated Graph Sequence Neural Networks” took the core concept of graph neural networks and applied sequence based concepts such as gated cells to it to improve overall capacity and prevent loss of importance of information in large, extremely spread out sequences represented as graphs. The most important aspect of this research was the propagation model that took the messaging passing logic and applied recurrent gated cells at each vertex and pushed the boundaries of research in this field after years of inactivity.

2017: Gilmer et al. in their work “Neural Message Passing for Quantum Chemistry” improved on the idea of Neural Messaging Passing where information between the neighbors of nodes is shared. The benefit of message sharing is not only does each node in a graph, through training, have information about its initial state but also information about other nodes and adding a neural network between nodes representationally captures this information. I will be expounding on this concept a lot more in a later chapter as it is central to understanding how each node learns not only about its own information but also that of other nodes in the graphs. Another paper worth mentioning in the same bucket that improved my understanding of the concept of Neural Message Passing is that by Liao, et al. (2018) entitled “Graph Partition Neural Networks for Semi-Supervised Classification”.

2017: Schlichtkrull et al. in their work “Modeling Relational Data with Graph Convolutional Network” introduced the concept of Relational Graph Convolutional Networks that took plain vanilla graph neural networks and combined used the convolution function in the state update step in graphs with different type of directed edges that represented different relationships. The premise here is that the Convolution operation that is typically used in the case of computer vision can be applied in the case of graph neural networks for each vertex to gain further insight about it’s position with respect to other vertices. This work built on top of work by one of the research papers called “Semi-supervised classification with graph convolutional networks” by Kipf et al.

(2016) that initially introduced the graph convolution network and its operations in the context of semi-supervised learning.

2018: Allamanis et al. applied concepts from all the previously mentioned research in the context of learning representation from programs following the simple yet powerful assertion that source code can be represented as graphs where the vertices represent the tokens of the source code and the edges represent the relationship between the tokens. Additionally, taking advantage of the semantic and syntactic nature of source code added more detail to the input of the graphs that resulted in quicker training. This particular paper mainly explored Gated Graph Neural Networks and builds upon similar work by Li et al. (2017), however, the code associated with the paper that can be found here: <https://github.com/microsoft/tf-gnn-samples> involves a lot more types of graph neural network models. The majority of my capstone is built on this paper and its applications and therefore, for enthusiastic readers, I would advise reading this paper and studying the source code associated with it.

A lot of the concepts at this point might be confusing as they aren't fully explained; the basics of graph neural networks is covered in Chapter 4: Graph Neural Networks 101 that'll simplify the subject matter. Needless to say, I found the extensive research on this topic of graph neural networks is truly inspiring and exciting as well as growing exponentially with developments in multiple dimensions.

Other Presentations

Other links that can be referenced are the following two Youtube Videos by Microsoft Research. In my experience, these explained the concepts of graph neural networks the best for a beginner to understand:

1. “Understanding & Generating Source Code With Graph Neural Networks” Miltos Allamanis | FLOC 2018: <https://www.youtube.com/watch?v=AVvagxSeP2Q>.
2. Graph Neural Networks: Variation and Application:
<https://www.youtube.com/watch?v=cWIeTMklzNg>.

CHAPTER 3

PROJECT OVERVIEW

General Approach / Layout

The basic steps taken to approach this project are:

1. Defining the problem and associated experiments that we'll base the comparison on.
2. Acquisition and exploration of the input data source.
3. Deep diving into the source code associated with the Learning to Represent Graphs library to gain insight about how the code is executed.
4. Once the insight is gained to run the code, customize the code to accommodate for ease of use for my experiments by constructing a working pipeline that's based on ease of use.
5. Training models with the training and validation input data on GGNN and RGCN models for the experiments.
6. Using the weights from the training process to compute the test accuracy for RGCN and GGNNs.

7. Aggregate results to discern which model was the more performant one.
8. Conclude with possible next steps.

Before I start diving deeper into the steps taken to achieve these goals, I want to shed light on what's actually happening under the hood during the training and testing of these graph neural networks models, highlight the differences between GGNN and RGCN and define the task at hand with more detail.

CHAPTER 4

GRAPH NEURAL NETWORKS 101

Goals

In this section I achieve the following goals:

1. Define and Describe the Variable Misuse Task in detail.
2. Explain how Graph Neural Networks work.
3. Deep Dive into Gated Graph Neural Networks.
4. Deep Dive into Relational Convolutional Neural Networks.

By the end of this chapter, the reader should have gained some knowledge about graph neural networks and their details with relation to the comparative study between GGNNs and RGCNs. As a note, I made it a point to not be too mathematically dense as this topic can be fairly involved. An implicit goal here is for anyone with even a novice understanding of Machine Learning and Graph Theory to pick up on at least the basics involved.

The Variable Misuse Task

Properly defining and understanding the task associated with our statistical learning algorithm is of paramount importance. The Variable Misuse Task, in a nutshell, is a task that requires an algorithm to predict the most fitting variable in a particular scope among all the variables that are of the same type. If there is a discrepancy between what the code highlights and our confidence level of the prediction from our algorithm, the intended action to warn the user that line of code is a potential bug. To get a better understanding of this task, two examples would be helpful:

```
1 private readonly object _syncRoot = new object();
2 private bool _isDisposed = false;
3
4 public override bool IsDisposed
5 {
6     get
7     {
8         lock ( #1 )
9         {
10             return #2;
11         }
12     }
13 }
```

Fig. 2: Example 1 of Application of the VarMisuse Task in C#

This snippet of code is locking on a variable to provide synchronized access and then returning another variable. As a part of this task, predicting which variable is most suited for #1 and #2 is the goal. Furthermore, if there is a discrepancy between our confidence levels from the model and what the code indicates, there is a case of variable misuse.

As an example, if the GGNN model was trained and tested on this code, the following results are obtained:

For #1: **_syncRoot** as the correct variable with a confidence of 95% and **_isDisposed** as 5%.

For #2: **_isDisposed** as the correct variable with a confidence of 99% and **_syncRoot** as the correct variable as 1%.

These results imply there was no variable misuse bug in this case.

As a counterexample highlight the discrepancy between what's predicted and the actual variable used is predicted in the code is:

```
1 int[] results = new int[] { 1, 2, 3, 4, 5 };
2 int MAGIC_NUMBER = 200;
3 for(int idx = 0; idx < results.Length; idx++ )
4 {
5     results[ MAGIC_NUMBER ] += 2;
6 }
```

Fig. 3: Example 2 of Application of the VarMisuse Task in C#

This simple snippet of code essentially adds two to all the numbers in the array while looping through all the indices associated with the array. The fact that the code is trying to add 2 to the 200th index of the results array would eventually fail during runtime as it tries to reference an index out of the scope of the results array. It is important noting that there are some rule based static analyzers that wouldn't be able to detect this potential error.

The results from a well trained hypothetical model would illuminate a low confidence of the usage of `MAGIC_NUMBER` in the source code and highlight a warning to the user that there could potentially be a bug with the code while suggesting the correct variable i.e. `idx` in this case, with the highest confidence.

I'd like to point out here that this task would be extremely difficult to achieve with high accuracy with a plain vanilla sequence model. The representative capacity of the graph neural network helps with the inference of the role and functions of the variables of the program allows the efficient learning and retaining of pertinent features. Another point worth noting is that the variable misuse task can be used as a seminal example of a task that a static analyzer can solve as it is a proxy for a what a typically important task would entail. The similarity between the Variable Misuse Task and code completion is also another aspect that can be built upon once the accuracies of this test are up to industry standards.

Graph Neural Networks

Now that the task at hand has been well defined, it is important to shift gears and talk about Graph Neural Network Models by first going over what a graph is and then how neural networks can be applied for prediction for any feasible task.

Graphs and Representation of Programs

A graph is defined by a set of nodes or vertices and a list of edges connecting the nodes. A simple example of a graph is given by Fig. 4 where 1,2,3 and 4 are the nodes and the arrows are the edges connecting these nodes.

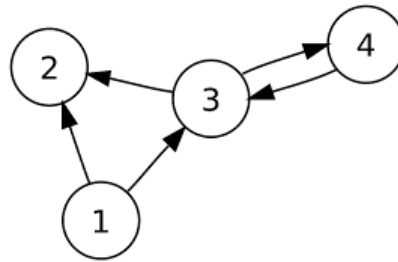


Fig. 4: An Example of a Graph

In the context of representation of programs as graphs, directed graphs or graphs with edges that can be pointing from one node to another, are specifically considered (contrasted with undirected graph where all edges are bi-directional). Further

characteristics are added to plain vanilla directed graph to better define and bolster the relationships between different components of the source code in an effort to improve representational capacity. These characteristics include associated features for each node that will represent the state vector whose value will be updated at each time step of the training of the graph neural network. Additionally, the concept of edge types is also introduced that represents different relationships between tokens such as a different edge type that connects one variable to another in case one variable reads from another.

Graph Construction

The construction of the graphs that represent programs is a modified version of the Abstract Syntax Tree constructed by the Roslyn Compiler, the .NET compiler framework. The Abstract Syntax Tree is a representation of the abstract syntactic structure of the code where the **syntax nodes** correspond to the programming language's grammar while the **syntax tokens** correspond to the leafs of the tree that represent the string from the source code.

The example below in Fig. 5 is for an Abstract Syntax Tree for the expression: $3 * 5 + 6$. For this simple expression, the syntactic tokens i.e. 3,5 and 6 are leafs of the AST and *

and + that are the corresponding arithmetical operators that are represented by non-terminal nodes.

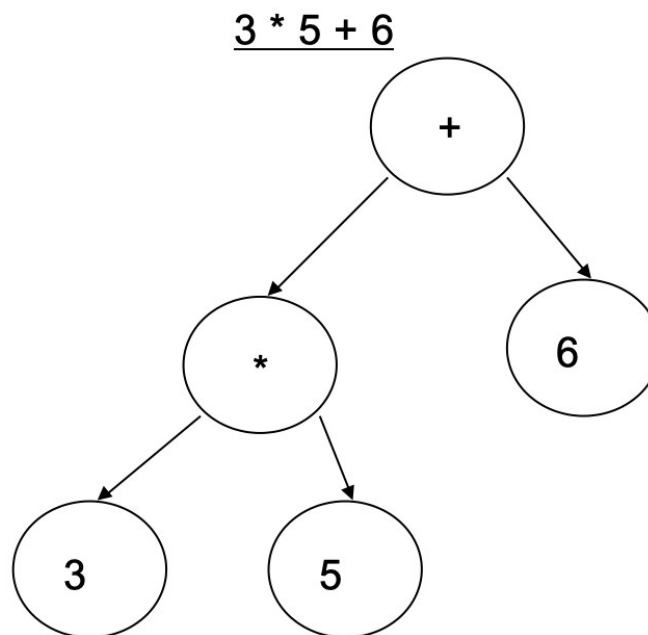


Fig. 5: Simple Abstract Syntax Tree

As an example of how these graphs can be visualized constructed from a simple statement, we borrow an example from Allamanis' presentation for the Conference on Computer-Aided Verification (2018):

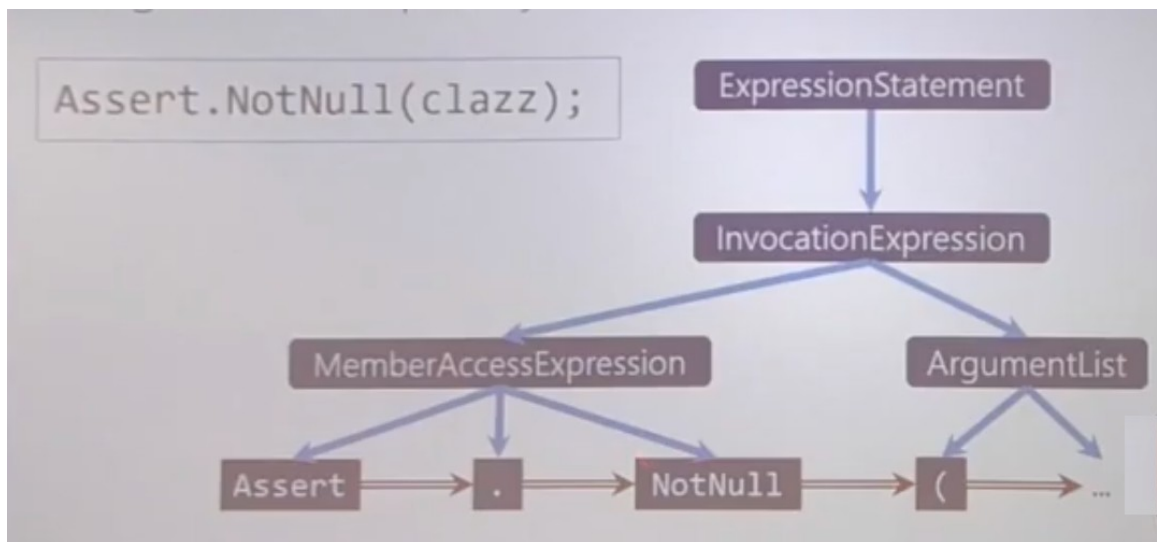


Fig. 6: Modified Abstract Syntax Tree that Represent Programs

The example highlights the Program Graph for a line of code that asserts if the “clazz” variable isn’t null. Adding these additional edges that represent different relationships between the tokens such as the ones that connect consecutive tokens e.g. Assert and “.” is the step required to transform the abstract syntax tree into the directed program graph that’ll eventually be used for the statistical learning phase.

Node Feature Embeddings

Each node is initialized by taking the embedding of the token that’s obtained as a function of the embedding of the textual representation of the token and it’s associated type. To explain further, the embeddings of subtokens of the token are averaged and then the embedding of type of the node is concatenated with the averaged value and passed through a linear layer. Embeddings for these subtokens and all tokens in this project, as confirmed by Marc Brockschmidt in the interview process, are standard token embeddings learned for the task and not via any form of transfer learning.

As an example, the token “**isSuccessfulCall**” of the bool type would be broken into the following subtokens *is*, *Successful* and *Call*. The individual embeddings for these subtokens are averaged and combined with the embedding of the type information and

passed through a linear layer to obtain the initial representation of a particular variable node.

Training

As a note, the source of the figures associated with this section is obtained from Microsoft Research's presentation by Marc Brockshmidt (2018) and I would consider this presentation to be the one that bolstered my understanding of the training of Graph Neural Networks.

The training at each time step is described as follows:

1. Each node is associated with a state i.e. its features whose value is initially set by the node feature embedding described earlier. These features of a particular node will be passed as messages to each of its neighbors based on the type of edge. There is a unique neural network for each of the type of edges associated with a particular node that demarcates the learning behavior.

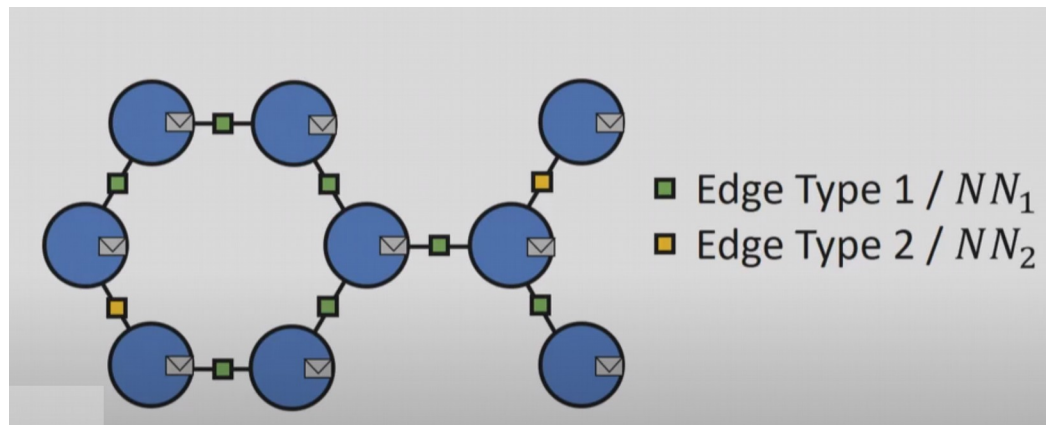


Fig. 7: Graph Structure and Messages For Each Node

2. Once the message of the previous state is sent from one node to its neighbors, it is aggregated using a summing function and is subsequently passed through an additional recurrent unit. This unit applies an additional mathematic function based on the previous state of the node and the summed up retrieved messages from its neighbors. The intuition here is that the features of a node after each time step become are some mathematical function of its original state and states of other nodes of the graph.

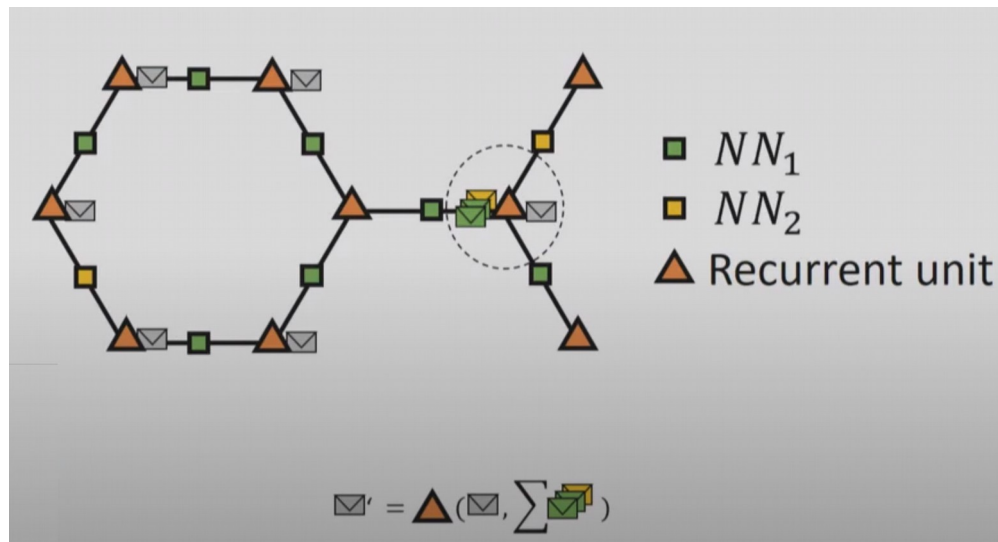


Fig. 8: Message Passing And Aggregation by Nodes via a Recurrent Unit

3. The unrolling stage is when, after some number of time steps that can be a hyperparameter, the individual nodes are “aware” of not only the features of its neighbors but also other nodes in further locations on the graph and these are indicated by the large dotted circles in the figure below. By the end of a certain number of time steps, all nodes should be acquainted with their own position as well as all other positions on the graph.

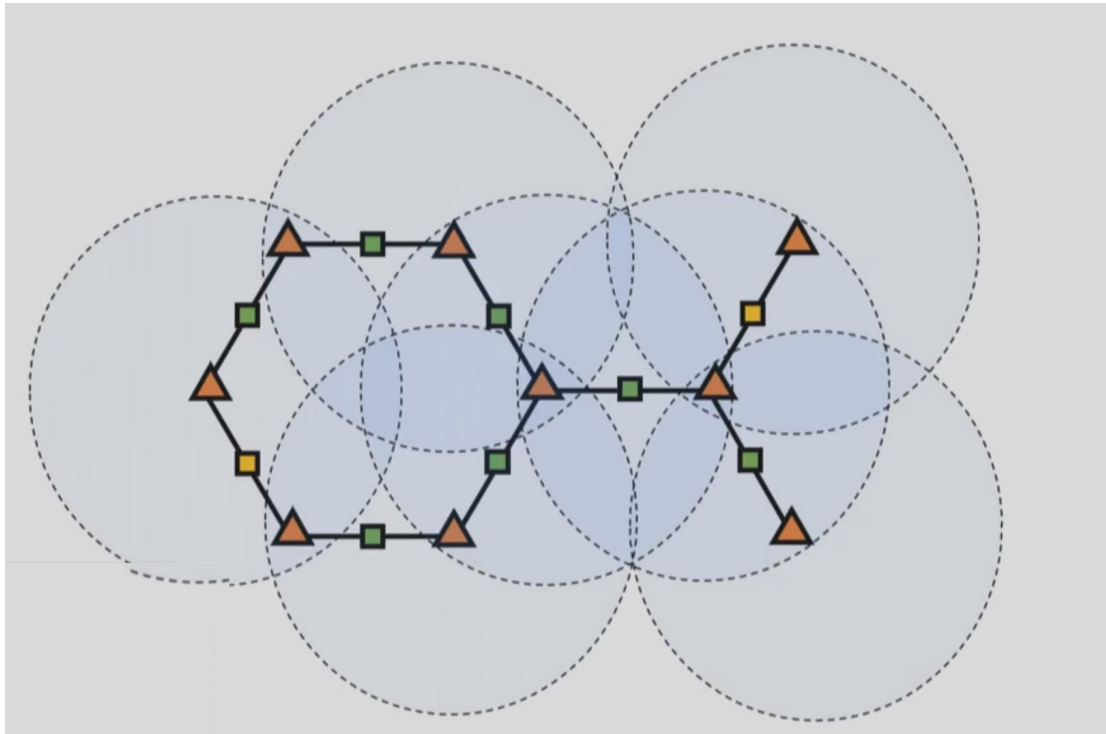


Fig. 9: Unrolled Effect Of the Graph Where Nodes Become Positionally Aware

The objective function used for the Var Misuse Task training is that of a max-margin objective on the state of a slot variable i.e. the state of an empty spot we are trying to predict and the state of all possible same typed variables in the same scope. In other words, using the state representations, we are trying to maximize the similarity of the

state of the nodes representing a variable of the same type in the scope matches the ones that would be characteristic of the slot.

This trained graph neural network can also be fed to higher layers for more complicated representations for predictions based on more involved tasks. Specific to our comparative study, we'll be making use of the trained graph neural networks with different units and aggregation functions to discern the more performant model and therefore, the two sections build on the concepts.

Gated Graph Neural Networks

As per research from Li et al. (2017), Gated Graph Neural Networks make use of a Recurrent Cell function of the Gated Recurrent Unit as the mathematical function to compute the next state; these gated recurrent units are responsible for keeping important features as the distance between nodes increase in a large graph and work on a gated mechanism that allows or disallows the unit to consider features from a node at a large distances. An example of where the Gated Recurrent Unit would come into action would be for large functions in source code with variables initialized at the very beginning of the scope that are used much later by other variables; in this case, we'd still want to know

about the characteristics of that node without losing information through the large distance by the representation of them in a graph.

The formula for the state updates for a GGNN are as follows:

$$h_v^{t+1} := \text{Cell}(h_v^t, \sum_{\ell} \sum_{(u,v) \in A_{\ell}} W_{\ell} * h_u^t)$$

Fig. 10: Formula for the GGNN State Update Per Node

To explain the formula, for each time step ‘t’, for node ‘v’, the state is computed by running the previous state of ‘v’ and the aggregated messages from all the neighbors of ‘v’ multiplied by the training weights related to all edge types (a particular edge type is represented by ‘l’) through a gated recurrent cell. We apply this same formula to compute the state of all nodes to get our gated graph neural network based model to a trained state for prediction of our slot variable or the variable what we need to predict.

Relational Graph Neural Networks

Rather than a gated recurrent unit, the relational graph neural network makes use of the convolution function using a non-linear layer on the previous state of the neighbors, weights and problem specific normalization constant that is parameterized.

$$h_v^{t+1} := \sigma\left(\sum_{\ell} \sum_{(u,v) \in A_{\ell}} 1/c_{v,\ell} * (W_{\ell} * h_u^t)\right)$$

Fig. 11: Formula for the RGCN Node State Update Per Node

Like in the case of GGNN, for time step ‘t’ and node ‘v’, the next state of the node ‘v’ is computed by passing the normalized convolved aggregated states of all neighbors of ‘v’ through a non-linear unit such as a Rectified Linear Unit (ReLU) for all graph edge types ‘l’ with trained weights ‘W’.

As per Kipf et al. (2018), the intuition here is that accumulation of the transformed feature vectors of neighboring nodes through the normalized sum provide us with apt representations for different relations given by the different edge types.

CHAPTER 5

PRELIMINARY STEPS IN CASE STUDY

Now that the basics of Graph Neural Networks have been explained, the next step is to give the problem a lot more definition and expound on the preliminary steps undertaken for this comparative study.

Problem Definition

Defining the problem as meticulously as possible is an important step in Data Science and furthermore, even more important in the case of a comparison study to avoid ambiguity in the criteria of comparison.

Our problem statement is to demonstrate which model between the Gated Graph Neural Network and Relational Graph Convolutional Network based models has a higher predictive power on the basis of the test accuracy based on data from:

1. All 25 top trending C# Repositories
2. An Esoteric Repository
3. A Popular Repository

The general goal is to identify if there is a better model out of the two i.e. which model has a higher representational capacity, one associated with a gated recurrent unit and one with the convolutional operation. Once the better model is identified, the next steps are doing further research to apply the model to more sophisticated tasks by pushing the boundaries of the intersection of deep learning and static analysis.

The impetus behind choosing the GGNN and RGCN models was to prove a definitive hypothesis about the better model between two very disparate models from gated models and convolutional models respectively. In a sense, *caeteris paribus*, identifying what type of mathematical construct generalizes and predicts better in different circumstances.

Interview Notes

As a part of the interview process for this project, I was lucky enough to exchange emails with one of the authors of the paper my comparative study is based off of, Marc Brockschmidt, who extremely graciously answered all my very specific questions about the paper and methodology. Additionally, I spoke to Anmol Joshi, a data scientist from C-1 at Boeing to get a more general perspective about deploying models in production. The main takeaways from the conversations were:

1. Keep the code simple and highly decoupled from the data.
2. Treat the code that generates the initial representations of your data with as much importance as your modeling code.
3. The code that can be used for initial representation can be made decoupled from our modeling code if the pipeline is setup correctly.
4. Experiment as much as you can. You will never get the right answer right away as there is no right answer. It's only improvement of representation.
5. Keep researching and reading white papers to learn ideas from across other subfields.
6. There is no silver bullet in data science and continual experimentation is the only way to achieve good results.

These conversations helped me gain a good foundation as to how to go through with my experimentation. The insight I gained about the methodology specific to graph neural networks was extremely helpful and a lot of my questions related to the topic were answered to alleviate all doubts about this topic.

The choice of the experiments will be highlighted in the next chapter where the source data will be described.

CHAPTER 6

SOURCE DATA EXPLAINED

Introduction

The input data for this project is very different from the typical data frame based exploratory data analysis approach as it involves making use of the unstructured source code from repositories.

The absolute first stage of the data is that of the raw source code found in Github of the 24 top trending C# repositories. The source code from these top 24 trending C# repositories is then transformed into an Abstract Syntax Tree (AST) using Roslyn, the .NET Compiler Framework. Once the AST is obtained, the next step is to convert that it into a customized directed graph with the tokens representing the nodes and relationships between the variables represented by edges.

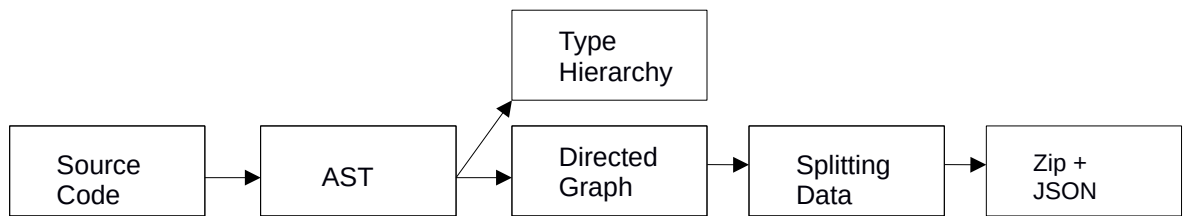


Fig. 12: Data Transformation Pipeline

Examples of the different type of edges include:

NextToken: The next token in the source code. In the example below, the x is pointing at the y as it is the next token in the source code.

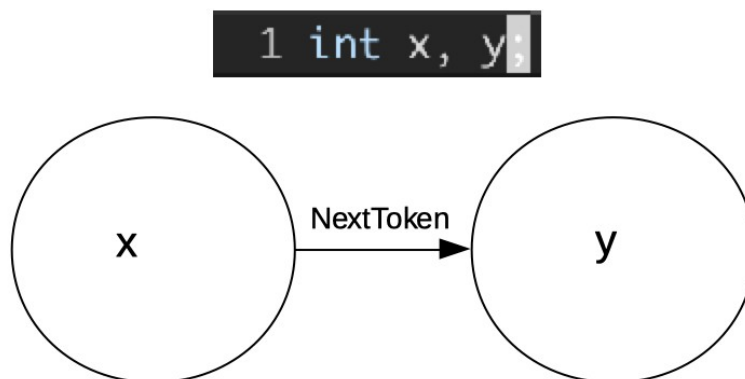


Fig. 13: Example Representation of the Next Token Of the Directed Graph

LastWrite: Represents the relationship between variables where the directed edge points to the variable that was last written to by the node that was pointing to it. For example, in the figure below, the second 'x' would point to the first one as it is the last instance in the source code that has written to the variable.

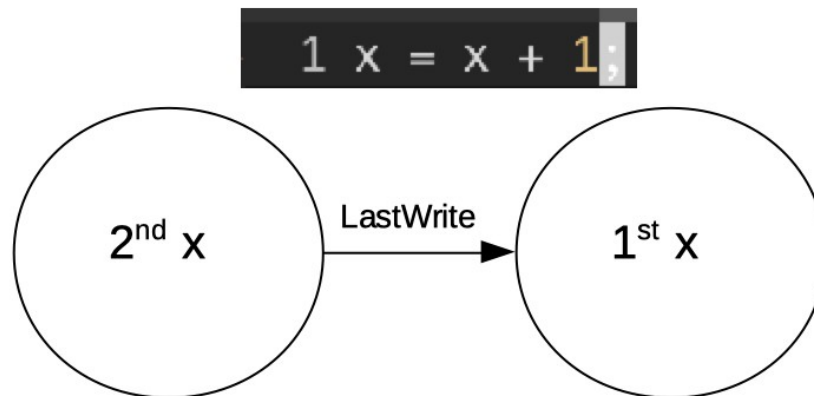


Fig. 14: Representing the LastWrite in the Source Code As a Graph

Once the graphs are successfully created, they are converted into JSON and then zipped up using gzip. The code involved with acquiring the source data from the repositories till we get all the JSON files can be found here: <https://github.com/microsoft/dpu-utils/tree/master/dotnet>.

Thankfully, the data for the top 25 trending C# repositories were easily available through Microsoft Research. The data obtained was a zip file consisting of 25 directories corresponding to each of the top trending repositories.

The structure of each of the directories representing the repositories is the same across all cases contain 2 zipped JSON files consisting of the token and type hierarchy information for the entire repository. Additionally, the data is split into testing, training and validation sets. The training and validation sets are used primarily for the training and hyperparameter validation and as suggested by the name, the testing files will be used to compute the test accuracy.

```

(base) mokosan:graph-dataset:% ls -l
total 104
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 akka.net
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 automapper
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 benchmarkdotnet
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 botbuilder
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 choco
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 commandline
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 commonmark
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 28 09:50 dapper
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 entityframework
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 humanizer
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 lean
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 nancy
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 newtonsoft
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 ninject
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 nlog
drwxr-xr-x 3 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 openlivewriter
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 opserver
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 orleans
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 polly
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 quartznet
-rw-r--r-- 1 mukundraghavsharma mukundraghavsharma 3871 Apr 13 2018 README.txt
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 restsharp
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 rxnet
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 scriptcs
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 signalr
drwxr-xr-x 6 mukundraghavsharma mukundraghavsharma 4096 Apr 13 2018 wox

```

Fig. 15: Listing of all the Directories From the Graph Data Set.

```
(base) mokosan:dapper:% tree
.
├── dapper-tokens.json.gz
├── dapper-typehierarchy.json.gz
├── graphs
│   └── dapper.1.gz
├── graphs-test
│   ├── dapper.0.gz
│   └── dapper.1.gz
├── graphs-train
│   ├── dapper.0.gz
│   └── dapper.1.gz
├── graphs-valid
│   ├── dapper.0.gz
│   └── dapper.1.gz
└── THIRD_PARTY_NOTICE.txt
```

Fig. 16: Typical Directory Structure of Data From A Repository

Next, I'll be going over the contents and structure of the zipped JSON files above. The three types of files that can be found in the data folder are the tokens file, type hierarchy and the actual graph.

Tokens

The tokens files consist of metadata associated with all the tokens as well as the data edges associated with the customized directed graph. This metadata along with the type hierarchy gives enough background information to the program to construct the graph in memory to conduct the training and testing. Examples of the type of properties in the token files include the types of the tokens, locations in terms of row and columns where the tokens were used, other variables of the same type in scope with a particular token and so on.

Type Hierarchy

The type hierarchy file consists of the hierarchy or inheritance model associated with types i.e. the literal type used in the source code as well as the base classes, if any, that they were derived from. Additionally, an array of the type hierarchy where the indices correspond to the types used in the source code were added.

Structure of Typical Graph File

The graph file consists of the context directed program graph that contains all the information needed for both the training and testing. The main part of the file consists of a representation of the graph as a dictionary of the different edges, nodes and combinations of the cases where each variable we are trying to predict is replaced by a “slot” or empty spot for which we will maximizing the match on with all the other variables of the same type and in the same scope.

Studying these files in depth gave me a considerable insight as to how meticulously the input data is structured. Verbosity in the data seems to be the prevailing trend here where there is considerable amount of emphasis on is made on over specification of the details so that there is minimal transformation during training and testing.

I would like to note that it is a bit difficult to specify all the properties in each of these classes and for that reason I stuck with a basic description. Nonetheless, I have added typical samples to the repository that can be found in the assets/samples/ directory of my fork of the main repository that can found here: <https://github.com/MokoSan/tf-gnn-samples/tree/master/assets/samples>.

Other Data Sources

The other data source considered was that of the top 25 trending C# repositories on Github that I added to a CSV file called “RepositoryInfo” and was the only structured data in this project compared to the unstructured nature of the source code. The purpose of this data was to add form to and define different aspects of my experiments; specifically answering questions such as based on the data, what should be characterized as an esoteric repository? Or what should be characterized as a popular repository? Acquiring this data involved getting the information from Github on the top 25 trending C# repository such as making use of the Github API.

The features considered in this case were the following to add form and definition to the three experiments I plan to consider:

1. **Repository:** The name of the repository considered.
2. **Link:** A link or URL of the repository on Github.
3. **Stars:** A proxy for the crowd-sourced popularity of the repository. Equivalent to the number of likes on Twitter or Facebook.
4. **Watchers:** The number of users who are actively getting updates about the repository such as issues posted by other users.

5. **Forks:** The number of copies of repositories made by other users for the purposes of independently developing the repository with their own changes.
6. **Commits:** The number of snapshots indicating some unit of work done by the contributors.
7. **Contributors:** The number of unique users who contributed to the repository since inception.

An example of a record from this data is the following:

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
akka.net	https://github.com/akkadotnet/akka.net	3.5k	304	891	5547	197

Fig. 17: Example of a Record of the RepositoryInfo

CHAPTER 7

METHODOLOGY AND RESULTS

Introduction

Now that we have gone over the preliminary case study, fundamentals of Graph Neural Networks and described the source data, the next step would be to elucidate on the methodology to conduct the experiments with the pipeline and obtaining the results. This involves a number of steps are these are:

1. Experimental Design
2. Details Related to the Pipeline
3. Conducting the Experiments
4. Results
5. Next Possible Steps

Experimental Design

Designing the experiments to properly fit the problem at hand is of paramount importance. I decided to conduct three experiments for this comparison between GGNN and RGCNs that are designed to answer the following three questions:

1. Which type of model provides a higher test accuracy across all the top trending C# repositories?
2. Which type of model provides a higher test accuracy for an esoteric repository?
3. Which type of model provides a higher test accuracy for a popular repository?

The rationale for choosing the concomitant experiments relating to the aforementioned questions was to examine the results across the entire spectrum of the type of repositories. Not all the repositories garner the same number of contributors or the number of stars, a crowd-sourced metric that's a proxy of the popularity of the repository. on Github and I wanted my comparative experiments to revolve around other dimensions of generality apart from looking at results across all repositories.

To be as specific as possible, the following three experiments conducted were in an effort to obtain the test accuracy for both the GGNN and RGCN for:

1. All repositories
2. An Esoteric Repository
3. A Popular Repository

The top 25 trending C# repositories considered were the following:

RepositoryInfo						
Repository	Link	Stars	Watchers	Forks	Commits	Contributors
akka.net	https://github.com/akkadotnet/akka.net	3.5k	304	891	5547	197
automapper	https://github.com/AutoMapper/AutoMapper	7.1k	403	1500	3512	137
benchmarkdotnet	https://github.com/dotnet/BenchmarkDotNet	5.2k	261	554	1980	129
botbuilder	https://github.com/microsoft/botbuilder-dotnet	0.496k	96	294	9010	124
choco	https://github.com/chocolatey/choco	6.4k	237	642	2965	76
commandline	https://github.com/commandlineparser/commandline	2k	64	261	1809	73
commonmark	https://github.com/Knagis/CommonMark.NET	0.929k	66	137	392	11
dapper	https://github.com/StackExchange/Dapper	12.1k	1000	3100	1387	151
entityframework	https://github.com/dotnet/ef6	1.1k	160	474	2921	79
humanizer	https://github.com/Humanizr/Humanizer	4.6k	213	672	1904	167
lean	https://github.com/QuantConnect/Lean	3.3k	347	1600	9887	111
nancy	https://github.com/NancyFx/Nancy	7.1k	460	1500	5473	263
newtonsoft	https://github.com/JamesNK/Newtonsoft.Json	7.7k	527	2600	1821	95
ninject	https://github.com/ninject/Ninject	2.3k	172	526	851	27
nlog	https://github.com/NLog/NLog	4.4k	288	1100	5735	152
openlivewriter	https://github.com/OpenLiveWriter/OpenLiveWriter	2.3k	208	484	696	46
opserver	https://github.com/opserver/Opserver	3.9k	331	785	998	55
orleans	https://github.com/dotnet/orleans	6.4k	550	1500	4824	210
polly	https://github.com/App-vNext/Polly	7.3k	374	743	961	54
quartznet	https://github.com/quartznet/quartznet	3.8k	326	1200	1295	65
restsharp	https://github.com/restsharp/RestSharp	7.1k	459	2000	1743	199
rxnet	https://github.com/dotnet/reactive	4.2k	295	548	3316	59
scriptcs	https://github.com/scriptcs/scriptcs	2.2k	152	375	1961	61
signalr	https://github.com/SignalR/SignalR	8k	712	2200	4684	81

Fig. 18: All Repositories Considered

Using the data from the “RepositoryInfo” table, I define two terms that were mentioned multiple times in our experiments:

Esoteric Repository

An esoteric repository is defined as one with a relatively low number of stars and a low number of contributors whose purpose is extremely specific; in general, the interest from the general C# populous is low for these repositories. I chose CommonMark.NET, a repository that converts Markdown to HTML, as the esoteric repository.

Quantitatively, CommonMark.NET had $< 1,000$ stars, 11 contributors and 66 watchers. These numbers are significantly lower than the other repositories considered and therefore, we consider this repository as our ideal candidate fitting the definition of an esoteric repository.

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
commonmark	https://github.com/Knagis/CommonMark.NET	0.929k	66	137	392	11

Fig. 19: CommonMark.NET Details

Qualitatively, based on the nature of the code of the repository i.e. a .NET port of an extremely popular library, CommonMark, indicates that this was probably someone's weekend project or needed for another project that gained some traction over time. Therefore, CommonMark.NET seems like the ideal candidate from even a qualitative perspective to be characterized as an esoteric repository.

Popular Repository

A popular repository is one that has a lot of stars, contributors and users watching for updates. The choice of popular repository is that of Dapper, a simple object mapper for .NET that could be used to abstract out the layer between the database and plain old class objects (POCO).

Quantitatively, Dapper has around 12.1 thousand stars, 1000 watchers, 151 contributors and 3100 forks indicating it is a well maintained and liked repository with a lot of interest in further development as it is used by many .NET developers in production.

Repository	Link	Stars	Watchers	Forks	Commits	Contributors
dapper	https://github.com/StackExchange/Dapper	12.1k	1000	3100	1387	151

Fig. 20: Dapper Details

From a qualitative perspective, since an uncontested amount of code involves accessing a database and retrieving objects, Dapper seems like the perfect candidate for a popular repository. Additionally, the main repository has branched out into many other packages under the same umbrella.

Now that the experiments and definitions associated with the experiments are well defined, the next step to detail how the code was run on the data.

Details Related to the Pipeline

This section involves the following steps in an effort to get the reader quickly up to speed with the abstractions and the pipeline to conduct experiments to generate the test accuracies:

1. Describing the Source Code To Run The Experiments
2. Preparation of the Data
3. Training
4. Testing

Describing the Source Code To Run The Experiments

The code used to run the experiments can be found here: <https://github.com/mokosan/tf-gnn-samples>. The link is to my fork of the original “tf-gnn-samples” repository or the Tensorflow based implementation of the GNN based sample library by Microsoft Research can be found here: <https://github.com/microsoft/tf-gnn-samples>. My fork contains additional modifications to the code to fit the pipeline I wanted to run as well as additional assets supplementing this paper.

The original source code that my implementation is based off of is written in Python 3.x using Tensorflow 1. There discussion that indicates that this code i.e. tf-gnn-samples is getting converted into Tensorflow 2 that, in my opinion, is considerably more user friendly than Tensorflow 1. Tensorflow 2 contains extra features such as eager execution that results in a smoother development experience as users don't have to construct a computation graph each time even if they want to execute a menial task with different tensorflow constructs.

The major abstractions include the following:

Models:

The models encapsulate different types of graph based neural networks and are essentially representations of the directed graphs containing the edges and nodes with the graph neural networks. The code in `sparse_graph_model.py` provides a general entry point to make use of other models based on the inputs from the user. The code paths made use of from these abstractions are `rgcn_model.py` and `gnn_model.py`.

```

Mukunds-MacBook-Air:tf-gnn-samples:% ll models
total 104
-rw-r--r--  1 mukundraghavsharma  staff   322B Apr 25 12:33 __init__.py
-rw-r--r--  1 mukundraghavsharma  staff   1.7K Apr 25 12:33 ggnn_model.py
-rw-r--r--  1 mukundraghavsharma  staff   1.9K Apr 25 12:33 gnn_edge_mlp_model.py
-rw-r--r--  1 mukundraghavsharma  staff   1.6K Apr 25 12:33 gnn_film_model.py
-rw-r--r--  1 mukundraghavsharma  staff   1.5K Apr 25 12:33 rgat_model.py
-rw-r--r--  1 mukundraghavsharma  staff   1.6K Apr 25 12:33 rgcn_model.py
-rw-r--r--  1 mukundraghavsharma  staff   2.0K Apr 25 12:33 rgdcn_model.py
-rw-r--r--  1 mukundraghavsharma  staff   1.9K Apr 25 12:33 rgin_model.py
-rw-r--r--  1 mukundraghavsharma  staff   19K Apr 25 12:33 sparse_graph_model.py

```

Fig. 21: List of Models

Graph Neural Network Types:

The Graph Neural Network types describe individual networks associated with the nodes and edges that contain the mathematical functions required for state updates. The files used by this project are ggnn.py and rgcn.py.

```

Mukunds-MacBook-Air:tf-gnn-samples:% ll gnn
total 128
-rw-r--r--  1 mukundraghavsharma  staff   278B Apr 25 12:33 __init__.py
-rw-r--r--  1 mukundraghavsharma  staff   4.6K Apr 25 12:33 ggnn.py
-rw-r--r--  1 mukundraghavsharma  staff   6.0K Apr 25 12:33 gnn_edge_mlp.py
-rw-r--r--  1 mukundraghavsharma  staff   6.5K Apr 25 12:33 gnn_film.py
-rw-r--r--  1 mukundraghavsharma  staff   7.7K Apr 25 12:33 rgat.py
-rw-r--r--  1 mukundraghavsharma  staff   6.0K Apr 25 12:33 rgcn.py
-rw-r--r--  1 mukundraghavsharma  staff   9.3K Apr 25 12:33 rgdcn.py
-rw-r--r--  1 mukundraghavsharma  staff   7.0K Apr 25 12:33 rgin.py

```

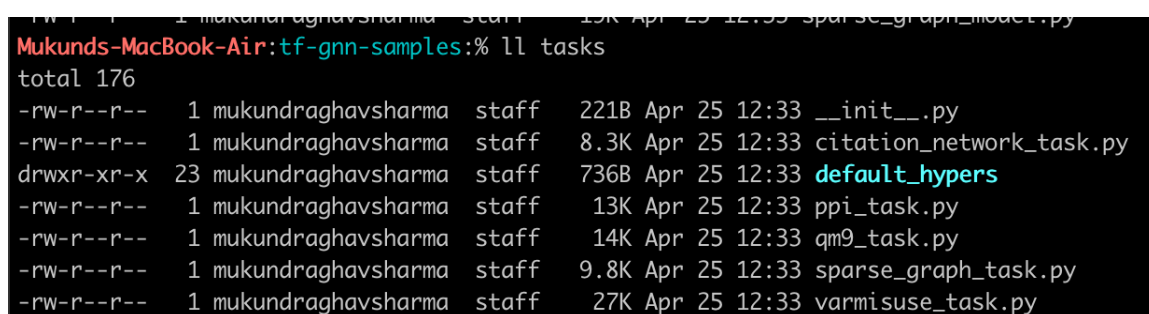
Fig. 22: List of Gated Graph Neural Networks

Tasks:

As mentioned before, this repository contains more than just the task we are working with i.e. the Variable Misuse Task. This directory contains the other tasks along with dictionaries containing the default hyperparameters associated with the tasks and the respective models.

I'll touch on this briefly, however, based on the results I found by tweaking hyperparameters such as the decay rate or learning rate during my experimentation, I

found that there wasn't much improvement overall. The hyperparameters were stored as a JSON file and then fed into the code as a dictionary in typical python fashion. The specific files that I made use of from the tasks directory were varmisuse_task.py and the default hyperparameters that were eventually used were VarMisuse_RGCN.json and VarMisuse_GGNN.json.



```

Mukunds-MacBook-Air:tf-gnn-samples:% ll tasks
total 176
-rw-r--r--  1 mukundraghavsharma  staff   221B Apr 25 12:33 __init__.py
-rw-r--r--  1 mukundraghavsharma  staff   8.3K Apr 25 12:33 citation_network_task.py
drwxr-xr-x 23 mukundraghavsharma  staff  736B Apr 25 12:33 default_hypers
-rw-r--r--  1 mukundraghavsharma  staff   13K Apr 25 12:33 ppi_task.py
-rw-r--r--  1 mukundraghavsharma  staff   14K Apr 25 12:33 qm9_task.py
-rw-r--r--  1 mukundraghavsharma  staff   9.8K Apr 25 12:33 sparse_graph_task.py
-rw-r--r--  1 mukundraghavsharma  staff   27K Apr 25 12:33 varmisuse_task.py

```

Fig. 23: List of Tasks And Hyperparameters

Running Scripts:

The three main scripts used to run the code were the `reorg_varmiseuse_data.sh` script that I handcrafted to tailor specifically to the VarMisuse Task with some adjustments that were needed to run. This script basically converts the raw JSONized directed program graphs into digestible chunks that can be pulled into memory to create the models.

The two other scripts that run as command line programs are `train.py` and `test.py`. The source code from `train.py` takes the outputted data from `reorg_varmiseuse_data.sh` and trains the models. Additionally, optimization is done on the basis of the validation data covered in the Source Data section. The weights after each epoch of training are serialized and saved as pickle files. These pickle files form the basis of which the testing is commenced by the `test.py` file that first loads the weights and runs a round of inference in the form of prediction to compute the test accuracy.

```
Mukunds-MacBook-Air:tf-gnn-samples:% ll
total 6272
-rw-r--r--@ 1 mukundraghavsharma staff 916B Apr 25 12:33 CONTRIBUTING.md
-rw-r--r--@ 1 mukundraghavsharma staff 2.1M Apr 30 12:21 FinalReport.docx
-rw-r--r-- 1 mukundraghavsharma staff 1.1K Apr 25 12:33 LICENSE
-rwxr-xr-x 1 mukundraghavsharma staff 14K Apr 25 12:33 README.md
-rw-r--r--@ 1 mukundraghavsharma staff 1.7K Apr 25 14:35 RepositoryInfo.csv
drwxr-xr-x 8 mukundraghavsharma staff 256B Apr 30 11:47 assets
drwxr-xr-x 3 mukundraghavsharma staff 96B Apr 25 12:33 data
drwxr-xr-x 13 mukundraghavsharma staff 416B Apr 25 12:33 final_results
drwxr-xr-x 10 mukundraghavsharma staff 320B Apr 25 12:33 gnns
drwxr-xr-x 11 mukundraghavsharma staff 352B Apr 25 12:33 models
drwxr-xr-x 4 mukundraghavsharma staff 128B Apr 26 14:48 previousdocs
-rwxr-xr-x 1 mukundraghavsharma staff 1.2K Apr 25 12:33 reorg_varmiseuse_data.sh
-rw-r--r-- 1 mukundraghavsharma staff 89B Apr 25 12:33 requirements.txt
-rw-r--r-- 1 mukundraghavsharma staff 2.5K Apr 25 12:33 run_ppi_benchs.py
-rw-r--r-- 1 mukundraghavsharma staff 3.2K Apr 25 12:33 run_qm9_benchs.py
-rw-r--r-- 1 mukundraghavsharma staff 4.2K Apr 25 12:33 run_varmiseuse_benchs.py
drwxr-xr-x 9 mukundraghavsharma staff 288B Apr 25 12:33 tasks
-rwxr-xr-x 1 mukundraghavsharma staff 1.8K Apr 25 12:33 test.py
-rw-r--r-- 1 mukundraghavsharma staff 4.6K Apr 25 12:33 train.py
drwxr-xr-x 7 mukundraghavsharma staff 224B Apr 25 12:33 utils
```

Fig. 24: List of ScriptsMisc Utilities:

Miscellaneous utility files include all the auxiliary functions used throughout the repository from code that is responsible for bringing the data into memory and constructing the representation of the directed program graph to dictionary look up functions used throughout the repository that map the name of the model type to its functional equivalent.

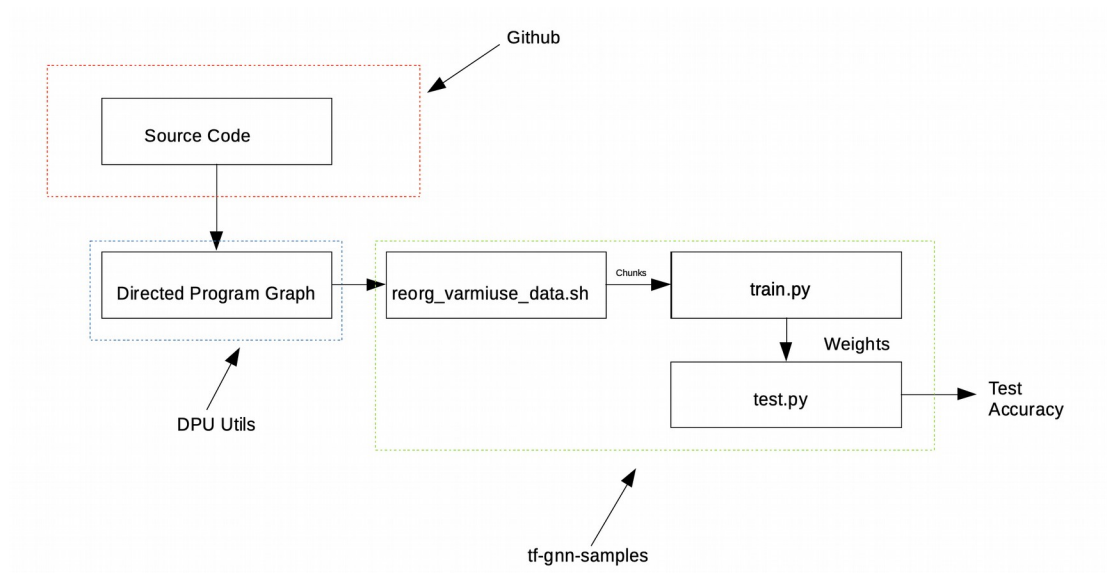
```
Mukunds-MacBook-Air:tf-gnn-samples:% ll utils
total 56
-rw-r--r--  1 mukundraghavsharma  staff   117B Apr 25 12:33 __init__.py
-rw-r--r--  1 mukundraghavsharma  staff   5.2K Apr 25 12:33 citation_network_utils.py
-rw-r--r--  1 mukundraghavsharma  staff   2.9K Apr 25 12:33 model_utils.py
-rw-r--r--  1 mukundraghavsharma  staff   5.0K Apr 25 12:33 utils.py
-rw-r--r--  1 mukundraghavsharma  staff   3.6K Apr 25 12:33 varmisuse_data_splitter.py
```

Fig. 25: List of Utilities

Pipeline

The entire pipeline is described as the following:

1. The source code is obtained through **Github**.
2. The **DPU-Utills** extract the source code and convert the code into a directed program graph.
3. The directed program graph is converted into digestible chunks by the `reorg_varmiuse_data.sh` script in `tf-gnn-samples`.
4. Using those chunks as inputs to the `train.py` method, the neural network is trained and the weights are saved as pickle files.
5. Using the weights as pickle files and picked up by `train.py` to generate the corresponding test accuracy.
6. The test accuracy is noted and the experiment is repeated for different models.
7. Once the test accuracies for different models is obtained for a particular data set, the next step is to change the experiment by swapping the input data to the `reorg_varmiuse_data.sh` script.



26: Mockup of the Pipeline

In general, I'd like to conclude by highlighting that the source code is incredibly cleanly written with emphasis on generality of the type of task, model used and the graph neural network considered to bolster separation of concerns. I also appreciated the documentation associated with all the graph neural network types that supplemented my theoretical understanding by easily being able to reason about the mathematical formulae. One other aspect of the source code that I considered to be done incredibly well done was the clearly separated utilities that are used across a lot of the files. Marc Brockshmidt, the

contributor I reached out to was also incredibly welcoming in others contributing, which made this exploration experience all the more worthwhile.

Conducting The Experiments

This section involves adding specificity to the abstractions described in the previous section to be applied to the experiments that were conducted. For each of the experiments, the data is obtained and is separated out. In general the following figure describes the data required:

Experiment	Data
General	All Repositories
Esoteric Repository	<u>CommonMark.NET</u>
Popular Repository	Dapper

Fig. 27: Data Needed For Each of the Experiments

Once the `reorg_varmisuse_data.sh` script is run on each of the data sets for each of the experiments, running `train.py` is the next step with appropriate arguments. These arguments include the location of the result of the script that creates chunks out of the data i.e. the digestible input used by the training process, the type of task and the type of model to be used.

An example of the arguments associated with the training phase are as follows:

```
(base) mokosan:~:% python train.py --data-path "./reorg" --debug RGCN Varmisuse
```

Fig. 28: Training Command Line Args

During the training phase for this specific run, what is observed is for each epoch, the training and validation accuracy is computed and if there is an improvement to the validation accuracy, it is saved in a pickle file. This loop of training continues till we

don't improve the validation accuracy based on a number of epochs (default is 5 specified by the "patience" parameter) or we reach the maximum number of epochs (default is 10,000 specified by the "max_epochs" parameter).

The command line output for a typical run is as follows where indication of each epoch of training is given:

```
== Epoch 1
Train: loss: 2.79197 || Accuracy: 0.556 || graphs/sec: 18.41 | nodes/sec: 196259 | edges/sec: 1411159
Valid: loss: 0.92193 || Accuracy: 0.598 || graphs/sec: 154.65 | nodes/sec: 235263 | edges/sec: 1586476
(Best epoch so far, target metric decreased to -0.59756 from inf. Saving to 'trained_models/VarMisuse_RGCN_2020-04-18-17-16-51_30727_best_model.pickle')
== Epoch 2
Train: loss: 0.91212 || Accuracy: 0.659 || graphs/sec: 20.46 | nodes/sec: 218029 | edges/sec: 1567688
Valid: loss: 0.89644 || Accuracy: 0.625 || graphs/sec: 409.21 | nodes/sec: 622524 | edges/sec: 4197928
(Best epoch so far, target metric decreased to -0.62500 from -0.59756. Saving to 'trained_models/VarMisuse_RGCN_2020-04-18-17-16-51_30727_best_model.pickle')
== Epoch 3
Train: loss: 0.76124 || Accuracy: 0.709 || graphs/sec: 20.97 | nodes/sec: 223542 | edges/sec: 1607327
Valid: loss: 0.91305 || Accuracy: 0.613 || graphs/sec: 411.82 | nodes/sec: 626499 | edges/sec: 4224737
```

Fig. 29: Command Line Output For Training

Along with training and validation accuracies at each step, the rate of processing of the edges is specified.

Once the training completes the weights can be used for the testing phase by the applying the parameters specifying the location of the pickle file from the training process that'll contain all the contextual information as well as parameters to compute the testing accuracy.

```
(base) mokosan:~:% python test.py trained_models/VarMisuse_RGCN_2020-04-19-02-48-26_11088_best_model.pickle
```

Fig. 30: Testing Command Line Arguments

The testing process is relatively short and simply involves simply computing the test accuracy based on the weights from the pickle file.

A typical run of testing looks like the following where the test accuracy is outputted to the terminal.

```

1 Model has 3809937 parameters.
2 Loaded model from snapshot trained_models/VarMisuse_RGCN_2020-04-18-17-16-51_30727_best_model.pickle.
3 Using the following task params: {"max_variable_candidates": 5, "graph_node_label_max_num_chars": 19, "graph_node_label_representation_size": 64, "slot_score_via_linear_layer": true,
  "loss_function": "max-likelihood", "max-margin_loss_margin": 0.2, "out_layer_dropout_rate": 0.2, "add_self_loop_edges": true}
4 Using the following model params: {"max_nodes_in_batch": 150000, "graph_num_layers": 10, "graph_num_timesteps_per_layer": 1, "graph_layer_input_dropout_keep_prob": 0.9,
  "graph_dense_between_every_num_graph_layers": 10000, "graph_model_activation_function": "tanh", "graph_residual_connection_every_num_layers": 10000, "graph_inter_layer_norm": false, "max_epochs":
  10000, "patience": 5, "optimizer": "Adam", "learning_rate": 0.00015, "learning_rate_decay": 0.98, "lr_for_num_graphs_per_batch": null, "momentum": 0.85, "clamp_gradient_norm": 1.0, "random_seed": 0,
  "hidden_size": 128, "graph_activation_function": "ReLU", "message_aggregation_function": "sum"}
5 == Running Test on data/varmisuse ==
6 Loss 0.81878 on 1105 graphs
7 Metrics: Accuracy: 0.64

```

Fig. 31: Testing Terminal Output

The output in the end is that of the test accuracy as a percentage. This is the test accuracy that will be used to discern which model outperforms the other.

As a note on computation time, from the experiments conducted the authors, it took around 2 weeks to compute all the results for all the repositories. It took me about 5 – 6 hours to run the experiment on the esoteric repository and about 2 hours longer for the popular repository. And to give a bit of a background, I built my own PC to conduct these experiments running Ubuntu 18.04 with an i9 core, 96 GB of RAM, 5 TB memory and a GTX 1080 Ti FTW3 GPU. In other words, training these extremely large graph based neural network models was extremely computationally expensive even on a state-of-the-art machine.

Results

The final aggregated results are as follows:

Criteria	RGCN	GGNN
Overall Generalization	87.2%	85.5%
Esoteric Repository - Common Mark	65%	62.5%
Popular Repository - Dapper	64.5%	61.8%

Fig. 32: Final Results

And the following are the conclusions based on these results:

- The final results indicated that **RGCN** based model was superior in terms of test accuracy in all experiments compared to the **GGNN** based model although within a margin of 5%.

- Surprisingly, the ability of both models to predict on both esoteric and popular repositories was the same i.e. the differences between the respective test accuracies of the esoteric vs. popular repositories was very similar.
- Training on more repositories provides significantly better results, as expected, than just the individual repositories.
- The gap between the RGCN and GGNN test accuracies decreased as the training was generalized on more data.

Next Possible Steps

Based on the conclusions from the results, specifically the fact that training on all repositories yields a higher test accuracy and lower difference between the RGCN and GGNN test results, the next steps in the scope of this project, would be rerun the experiments with the training based on all the repositories.

Outside the scope of this project, the next steps would be to:

Research other Graph Neural Network models such as Relational Graph Attention Networks that build or Relational Isomorphism Networks and study their behaviors on

different repositories. Additionally, further research that build on the weaknesses of these models with an effort to increase the representational capacity.

Additionally, I know I did play around with the hyperparameters and found that it was pretty much a fruitless endeavor, I don't want to absolutely negate the possibility that there is value to explore further as there are a considerable number of possibilities that can be used of further experimentation.

Future work also could include extending the problem to newer tasks and applying the entire concept of graph neural networks to other static analysis tooling. This would involve various changes to our current graph structure and could mean redesigning our models.

CHAPTER 8

CONCLUSION

To recap, in this paper I went over a comparative study between Gated Graph Neural Network models and Relational Graph Convolutional Networks on the Variable Misuse Task for the three experiments described. Additionally, I covered the basics of Graph Neural Networks, elucidated on the nuances of the input data namely, source code, went into details about the repository used to consume the source code and convert into a directed program graph and generate the results.

Using the repository, I generated the results and confirmed that the Relational Graph Convolutional Networks outperformed Gated Graph Neural Network on the basis of test accuracy, albeit, within a 5% range. And finally, I suggested next steps in this extremely deep topic that could have multiple hypotheses that can be thought of the prove or disprove.

Despite the initially extremely sharp learning curve associated with the white papers of this project, once I understood the source material and got familiar with this repository, working on this was nothing less than absolute fun! It was great to immerse myself in the

intersection between software engineering and deep learning and learn about what the future holds.

REFERENCES

- Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to Represent Programs with Graphs. <https://arxiv.org/pdf/1711.00740.pdf>.
- Conference on Computer-Aided Verification. (2018). “Understanding & Generating Source Code With Graph Neural Networks” Miltos Allamanis | FLOC 2018. [Video file]. Retrieved from <https://www.youtube.com/watch?v=AVvagxSeP2Q>.
- Hindle, A., Barr, E., Su, Z., Gabel, M., & Devanbu, P. (2012). On the naturalness of software. In International Conference on Software Engineering (ICSE). <https://people.inf.ethz.ch/suz/publications/natural.pdf>.
- Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. arXiv preprint arXiv:1704.01212. <https://arxiv.org/pdf/1704.01212.pdf>.
- Kipf, N. T., & Welling, M. (2016) Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907. <https://arxiv.org/pdf/1609.02907.pdf>.
- Li, Y., Tarlow, D., Brockschmidt, M., & Zemel, R. (2015). Gated graph sequence neural networks. In International Conference on Learning Representations (ICLR). <https://arxiv.org/pdf/1511.05493.pdf>.

Liao, R., Brockschmidt, M., Tarlow, D., Gaunt, A, L., Urtasun, R., and Zemel, R. (2018) Graph partition neural networks for semi-supervised classification. In ICLR Workshop. <https://arxiv.org/pdf/1803.06272.pdf>

Microsoft Research. (2018). Graph Neural Networks: Variation and Application. [Video file]. Retrieved from <https://www.youtube.com/watch?v=cWleTMklzNg>.

Raychev, V., Vechev, M., & Yahav, E. (2014). Code completion with statistical language models. In Programming Languages Design and Implementation (PLDI), pp. 419–428. <http://www.cs.technion.ac.il/~yahave/papers/pldi14-statistical.pdf>.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., & Monfardini, G. (2009). The graph neural network model. IEEE Transactions on Neural Networks 20(1):61–80. <http://www.cs.technion.ac.il/~yahave/papers/pldi14-statistical.pdf>.

Schlichtkrull, M., Kipf, N. T., Bloem, P., Berg, V, D, P., Titov, I., & Welling, M. (2017) Modeling relational data with graph convolutional network. arXiv preprint arXiv:1703.06103,. <https://arxiv.org/pdf/1703.06103.pdf>.

