Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master Thesis

Training a cross-language code-clone detection model with graph-to-sequence task

Graph-to-sequenceタスクで言語間コードクローン検出モデルを訓練する

Xuyang Yao
シュウヤン イアォ

Supervisor: Professor Shigeru Chiba

January 2018

Abstract

Cross-language code-clones refer to code fragments written in different programming languages but shares some level of similarity in their structure or functionality. In this work, we propose to build a cross-language code-clone detection model based on graph neural network (GNN). Detecting cross-language code-clones in different programming languages is vital for tasks like version control and bug fixing for cross-platform applications, but it remains a problem due to the difference in syntax between different programming languages. Consequently, many recent works have turned to compare the abstract syntax tree (AST) for structural similarity between code fragments. In our work, we introduced the popular graph neural network (GNN) model which is capable of extracting structural information from graphs, in order to efficiently extract information from the tree structure of an AST. However, existing cross-language code-clone datasets are not suitable for training our model, thus we proposed to train it with an especially designed sequence prediction task with modified dataset. Though we did not reach the state of the art performance, we proved the potential of GNN model for the cross-language code-clone detection task.

概要

言語間コードクローンとは、構造的・機能的に類似する、異なるプログラミング言語で書かれたコード片の組を指す。本研究ではグラフニューラルネットワーク（graph neural network, GNN）を基に構築した言語間コードクローン検出モデルを提案する。言語間コードクローンの検出は、さまざまなプラットフォームで動作するアプリケーションのバージョン管理やバグ修正において必要不可欠な技術であるが、言語間での構文の違いの大きさなどのため、未だ解決していない課題である。近年提案されている手法ではコード片を抽象構文木（abstract syntax tree, AST）へと変換し、ASTを用いて構造的な比較を行っている。本研究で提案する手法では、AST という木構造から効率的に情報抽出を行うため、グラフから構造的な情報を抽出する能力のある GNN を基にしたモデルを構築した。しかしながら言語間コードクローン検出のための既存データセットは提案モデルを訓練することに適していなかったため、特殊な列推定のタスクとして訓練を行った。提案モデルは最先端の研究結果ほどの性能を出すことはできなかったものの、本研究の成果はGNNを基にしたモデルが言語間コードクローンの検出に有用であることを示すことができた。

Contents

Chapter 1

Introduction

Code-clones refer to code fragments that share some level of similarity in their functionalities. With the explosive growth of code lines, such repeativeness of code becomes further unavoidable in software developing. code-clones can be caused by many programming behaviours. Intuitively, copying and pasting from other projects produce code-clones that are totally identical. Besides, re-implementing an existing functionality unconsciously or on purpose will lead to functionally similar code-clones. To better study the influence of code-clones with different level of similarity, previous study [1] briefly classified all code-clones into 4 types. While Type I to Type III clones include code pairs with a decreasing syntactical similarity, Type IV clones mostly refers to functionally similar code pairs that are quite different on code text.

Detecting code-clones is a necessary task for software development and maintenance. Copying and pasting contributes the most to Type I and Type II code-clones. It's a great boost to efficiency to discover those clones when we want to apply the same updating or bug fixing to all the copies. On the other hand, unconscious re-implementation potentially leads to Type III and IV clones. And according to [2], such clones working in the same project are highly possible of causing runtime errors. code-clone detection is also essential in many other tasks, such as selecting library candidate with similar utility or interface [3], aiding program comprehension by clustering code fragments according to functionality [4, 5], and detecting malicious software by their code behaviour [6], etc.

Important as it is, code-clone detection has been a long studied topic. Previous works have reached promising performance on detecting Type I and II clones. Due to the similarity in syntax between Type I and Type II clones, they can be efficiently detected by directly comparing the text of source code. On the other hand, Type III and Type IV clones share much less syntactical similarity, thus they are difficult to be detected by comparing source code. To this end, many related works have turned to compare the structural similarity to detect the similarity in code function of Type III and Type IV clones.

AST based code-clone detection Among many code structure representations, abstract syntax tree (AST) has been preferred in many code-clone detection studies. AST is a tree representation of the abstract syntactic structure of source code written in programming language. By "abstracting" the "syntax", ASTs ignore most of the text level details while preserving the

structure of the source code. Thus it is a common case where different code fragment may share roughly similar ASTs. By parsing code fragments into corresponding ASTs, code-clone detection is transferred into a tree comparison task. To compare the similarity of ASTs, some previous work applied algorithmic approach [7], while some others introduced deep learning methods to extract code information into vector representations [8].

1.1 Cross-language code-clone detection

In this work, we focus on a more specific group of Type IV code-clones, cross-language code-clones. While it is natural to find similar code lines in one programming language, it is also common case that the same functionality is implemented in different programming languages. For example, PyTorch is a well known and widely used open source machine learning library written in Python, but it also provides C++ and Java APIs. The implementation of the same interface provided for different programming language may look quite different on the code, but can be considered as code-clones due to their identical functionality. Besides, building the same deep learning model with PyTorch in different programming language would also result in cross-language code-clones.

The basic difference on syntax between programming languages makes cross-language code-clones even more difficult to be detected than monolingual Type IV clones. Again, researchers turn to abstracting code fragments into ASTs in order to avoid the syntactical difference between programming languages. However, on the one hand, recent studies haven't reached an agreement on how to efficiently compare ASTs. Algorithm ways require much manual work and professional knowledge on the programming language to make comparison rules. Deep learning based method face difficulty in the tree structure of ASTs, since conventional neural network models can only take linear inputs. On the other hand, the basic structure of ASTs from different programming languages are also different. In order to detect cross-language code-clone detection, we have to either remove such differences by manual modification, or to extract functionality similarity while resisting the interfere of the structural difference. In a word, there is still much space of improvement for cross-language code-clone detection.

1.2 Graph neural network based code-clone detection

Based on our study, we believe that a recent popular topic, the graph neural network (GNN) model, is a potential solution to the problems we addressed. GNN is a class of efficient graph embedding models. A typical GNN collects neighborhood information by message passing between nodes, and can further generate an embedding to represent the entire graph by aggregating from all nodes. By processing ASTs with a GNN model, we expect information about local code functionality to be collected in node embeddings, and summarize local functionalities into a general code embedding.

In this work, we intend to build a GNN based code embedding generator, which maps code fragments into a latent space. To achieve the target of code-clone detection, we expect the model to generate close vector representations for similar code fragments. Then for an arbitrary pair of code fragments, we can feed the code into our model, and compare the distance between generated code embeddings as a similarity score of the corresponding code fragments.

Though it is a very straightforward model design, it needs careful considering on how to train the model for generating such meaningful embedding. To the best of our knowledge, all existing deep learning based code-clone detection models are trained with typical code-clone datasets. In such a dataset, each pair of code fragments are simply labeled as clone or not, without any further information. However, the real relationship of similarity between code fragments is very complicated in high dimension. To this end, we propose to train this code embedding model with labels that contain more information.

We propose to train this model with a code to natural language prediction task, resulting in a Graph to Sequence model. We intend to dig code information from such natural language labels and train the GNN encoder to extract such information into code embeddings. From another perspective, our model takes the similarity between natural language labels as similarity scores of corresponding code fragments, and train the code-clone detection model with more detailed code-clone labels. Another reason for taking natural language sequence as training target is that most code fragments are naturally paired with some sort of document, thus code to natural language dataset is easy to acquire.

To better exploit the information buried in the natural language labels, we utilize the strong power of recurrent neural network (RNN). In our model, the RNN decoder takes the code embedding generated by GNN encoder as input to predict the target sequence. The code embedding can be considered to contain sufficient information of code label if the model manages to predict the target sequence.

1.3 Contributions

We conclude the contributions of our work as follows:

Introducing GNN into cross-language code-clone detection GNN has been widely used in many deep learning tasks that deal with graph structured inputs. Recently, we are also seeing a few attempts on analyzing code structures (e.g. AST, control flow graph (CFG)) with GNN models. One of the best attempts has successfully applied GNN to monolingual code-clone detection task and reached fairly high performance. However, to the best of our knowledge, none have successfully implemented a model for extraction of comparable information from cross-language code-clone code fragments. Thus in this work, we put our efforts on building a GNN based model that is capable of analyzing code fragments from different programming languages. Specifically, we focus on the task of cross-language code-clone detection. Though there remains much limitation on the utility and performance of our work, we argue that our work is an acceptable first step, and GNN has great potential on dealing with cross-language code tasks.

Slightly improving cross-language code-clone detection performance Our final target of building a GNN based code encoder is to detect cross-language code-clones. Though we did not manage to solve the problem perfectly with our proposed model, we observed a slight improvement on the performance compared to a recent work. The precision is improved from 19% to 30.2% using our graph-to-sequence-based code embedding model. On the other hand, unlike other works that embed ASTs from different languages into different latent spaces, and

requires another neural network, we managed to map cross-language ASTs into one single space, approximately according to their code structures.

Dataset generation As previously mentioned, we believe it's not the best choice to train a code-clone detection model with a typical code-clone dataset. In search of a better solution, we focused on a variety of code-to-sequence datasets. However, a natural code-to-sequence dataset contains sequence labels of different styles, and the focus of the labels may be on different perspective of the corresponding code. Thus it is too noisy for a code-clone detection model to capture the relationship between code fragments and their corresponding labels. To this end, we put our efforts on manually modifying the training dataset. We believe our model works better trained with the modified dataset, and this dataset would potentially help train other similar models.

1.4 Organization

Here we briefly introduce the contents of the rest of this thesis. In chapter 2, we will discuss about cross-language code-clone detection and existing works in detail, which is the background and motivation of this work. Then we present the overview of our proposed model, and go through every notable detail of our model structure in chapter 3. We will also explain how we build a code-to-sequence model for cross-language code-clone detection. Chapter 4 will cover the evaluation detail of our model. We compare the performance of our model with different model settings and other works from many perspectives. Finally in chapter 5, we summarize our work, discuss about our contributions and remaining problems, then finish with a future plan.

Chapter 2

Graph neural network based code embedding

The major target of this work is to build a graph neural network (GNN) based code encoder which is capable of generating meaningful embedding for an input code fragment. Such code embedding is expected to contain sufficient information about the corresponding code fragment, and can be utilized by many related downstream tasks. While in this work, we focus on utilizing the code embedding for cross-language code-clone detection task.

2.1 Background

Code-clones Code-clone detection is a long discussed topic. The general purpose of code-clone detection is to build a model for recognizing similar code fragment pairs from given dataset. To further evaluate the different levels of similarity between code fragments, previous work [1] categorized code-clones into the following four types:

Type I: Identical code fragments except for variations in white space (may be also variations in layout) and comments.

Type II: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.

Type III: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

Type IV: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

Code-clone detection has been proved necessary by previous study [2]. The example code fragments shown in Figure 2.1 and Figure 2.2 can be classified as Type III clones. Though they

```
def twoSum(self, nums, target):
    map = {}
    for i in range(len(nums)):
        x = nums[i]
        if target-x in map:
            return [map[target-x], i]
        map[x] = i
```

Figure 2.1. Two sum in Python

```
def countOccurrence(self, nums):
    map = {}
    for i in range(len(nums)):
        x = nums[i]
        if x not in map:
            map[x] = 1
        map[x] += 1
```

Figure 2.2. Count Occurrence in Python

are implemented for totally different functionalities, they share much similarity in source code text. Detecting such clones has been thoroughly studied with promising results.

2.1.1 Monolingual code-clone detection

Syntactical code-clone detection The basic problem of code-clone detection is detecting code-clones written in same programming language. For Type I and Type II clones, we can simply tokenize the text of source code and compare the resulted sequence, since they are at most different in a few variable and type tokens. It gets a little tricky to detect Type III clones. Since adding/removing of statements is included, a more complicated rule for token sequence comparison is needed in order to detect Type III clones in a syntactical way. As an example, CCFinder [9] follows a tokenize-transform-compare working flow for detecting syntactically different code-clones. The transform step modifies the token sequence according to pre-defined rules so that minor difference would be neglected.

```
x = 0
print(x)
```

Figure 2.3. Code example A

```
x = 0
pass
print(x)
```

Figure 2.4. Code example B

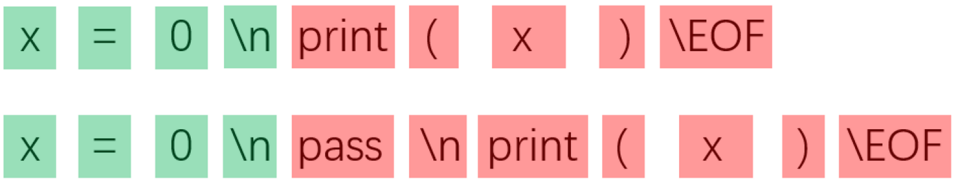


Figure 2.5. Tokenized result

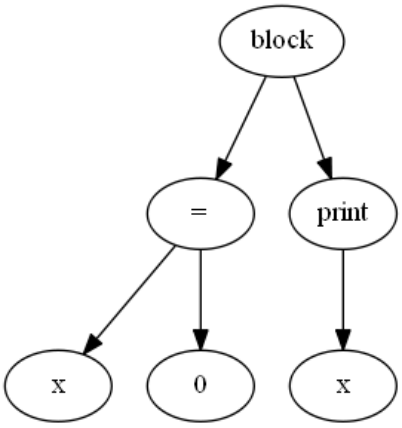


Figure 2.6. Tree structure of code fragment A

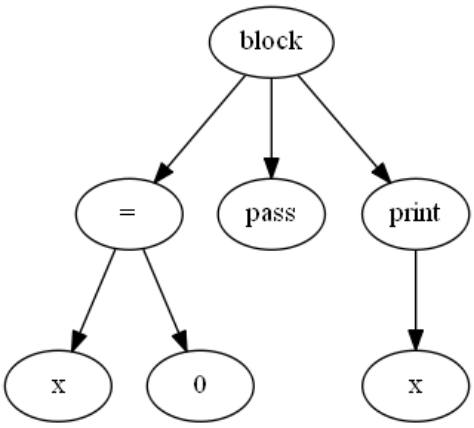


Figure 2.7. Tree structure of code fragment B

Building a transformation rule requires professional knowledge to the target programming language as well as heavy manual work. Figure 2.3-2.5 presents a typical example of such difficulty. Thought the code fragments shown in Figure 2.3 and Figure 2.4 are apparently code-clones, their tokenized sequences become totally unpaired on the second half, because of the insertion of a single 'pass' statement. However, such difference is much less influential on the structural representation of a code fragment. As shown in Figure 2.6 and Figure 2.7, the inserted 'pass' statement only resulted in a new child of the 'block' node, without any influence on the rest of the code structure representation.

Tree-based code-clone detection Many recent works have turned to compare the code structure for detection of Type III and Type IV code-clones. Among them, abstract syntax tree (AST) is a widely used code structure representation. AST is a tree representation of the abstract syntactic structure of source code written in programming language. By 'abstracting' the 'syntax', ASTs ignore most of the text level details while preserving the structure of the source code. Consequently, the code-clone detection task is converted into a tree comparison problem. A typical example is provided by Lawton et al. [7]. In their work, they built a tree comparison model, which linearize the tree in a depth-first-search order, and apply SmithWaterman algorithm on

the resulted node sequences for a similarity score.

Another work from Jiang et al. [10] applied a more special approach for tree comparison. Instead of directly designing tree comparison rules, the authors built a series of models that maps sub-tree in the AST into a multidimensional real number vector, which is also referred to as a feature vector. Then they further generate a feature vector for the corresponding code fragment according to sub-tree feature vectors. As a result of such processes, their model compresses the information of an entire AST into a single feature vector. Finally, code-clones are detected by clustering feature vectors. This is probably one of the earliest attempts on code embedding generation.

The idea of embedding became popular in deep learning area due to the invention of Word2vec algorithm. Generating an embedding for an object means building a model that extracts sufficient information from the object and represent the information with a multidimensional real number vector. Such method has been applied to the task of code-clone detection by Wang et al. [11]. In this work, instead of manually designing mapping functions, the authors applied a special deep learning model which is known as Graph Neural Network (GNN) to learn the embedding for code fragments automatically. We will talk about GNN with more detail in Section 2.2.2.

2.1.2 Cross-language code-clone detection

With the fast development of a variety of programming languages, the significance of efficiently detecting cross-language code-clones has also been stressed. As shown in Figure 2.8, this Java code fragment is an exact re-implementation of Python code in Figure 2.1. Though being exactly same in functionality and code structure, they still look quite different by text, due to the natural divergence in the syntax of different programming languages. Therefore, cross-language code-clones are mostly classified as Type IV clones.

Basically, each of the monolingual code-clone detection methods mentioned above could be generalized to the cross-language task with some modification. But because of the major different

```
1  public class Two_Sum {
2      public int[] twoSum(int[] nums, int target) {
3          Map<Integer, Integer> map = new HashMap<>();
4          for (int i = 0; i < nums.length; i++) {
5              int x = nums[i];
6              if (map.containsKey(target - x)) {
7                  return new int[]{map.get(target - x), i};
8              }
9              map.put(x, i);
10         }
11     }
12 }
```

Figure 2.8. Two Sum in Java

of syntax between different programming languages, it becomes difficult to detect cross-language code-clones with syntactical approaches. Thus, many recent works have turn to comparing code structure representation, such as AST, for cross-language code-clone detection. It is worth mentioning that many of them applied deep learning method for automatic modeling in order to reduce manual work.

Daniel et al. [8] applied a typical deep-learning based code embedding approach for cross-language code-clone detection. They first collected a rather large amount of code fragments from public projects. Then they parsed all the code fragments into ASTs, and collected the neighboring node pairs in all the ASTs as context, to form a node 'corpus'. On such 'corpus', a skip-gram algorithm [12] is applied to generate an embedding for each of the nodes. Given a code-clone dataset, an arbitrary code fragment is again parsed into an AST, then linearized to a node sequence in depth-first-search order. The node sequence is then fed into a bi-directional LSTM followed by a fully-connected neural network, to generate an embedding corresponding to the input code fragment. Finally, the code similarity is computed by the code embeddings.

2.1.3 AST based code embedding

Actually, code embedding is an independent topic which is also being widely studied. A well trained code embedding model would also perform well in code-clone detection tasks, since the embeddings should represent sufficient information from the corresponding code fragment, including structure and functionality. Our insight is to generalize code embedding models for cross-language task.

[13] is a recent study on AST based code embedding. In this work, Uri et al. propose a AST-path based AST context representation. To be specific, they extracted AST paths from an AST as local structure representations, then represent the structural information of the entire code fragment with a weighted combination of AST paths.

AST path An AST path is the route of AST nodes between a pair of terminal nodes, with the terminal nodes as the head and tail of the route respectively. A terminal node in AST refers to a leaf node of the tree structure, oppositely a non-terminal node refers to the root nodes of sub-trees with more than one nodes. Specifically, an AST path is composed of a pair of terminal nodes and a sequence of non-terminal nodes between them. From an AST with n terminal nodes, $O(n^2)$ different AST paths can be extracted. The authors further inserted $\uparrow$ and $\downarrow$ symbols between non-terminal nodes to denoted the parent-child relationships. An typical example is shown in Figure 2.9.

Next, an attention model is trained for learning the importance of each different AST path, The attention model is a universal model which computes an importance score of an AST path according to its representation. Then the computed importance scores of all AST paths would be normalized as follows:

$$\alpha_i = \frac{\exp I_i}{\sum_j \exp I_j}$$

where I_i refers to the importance of an AST path i, and the exponents in the equation are used to make the attention weights positive. Such normalization works as a standard softmax

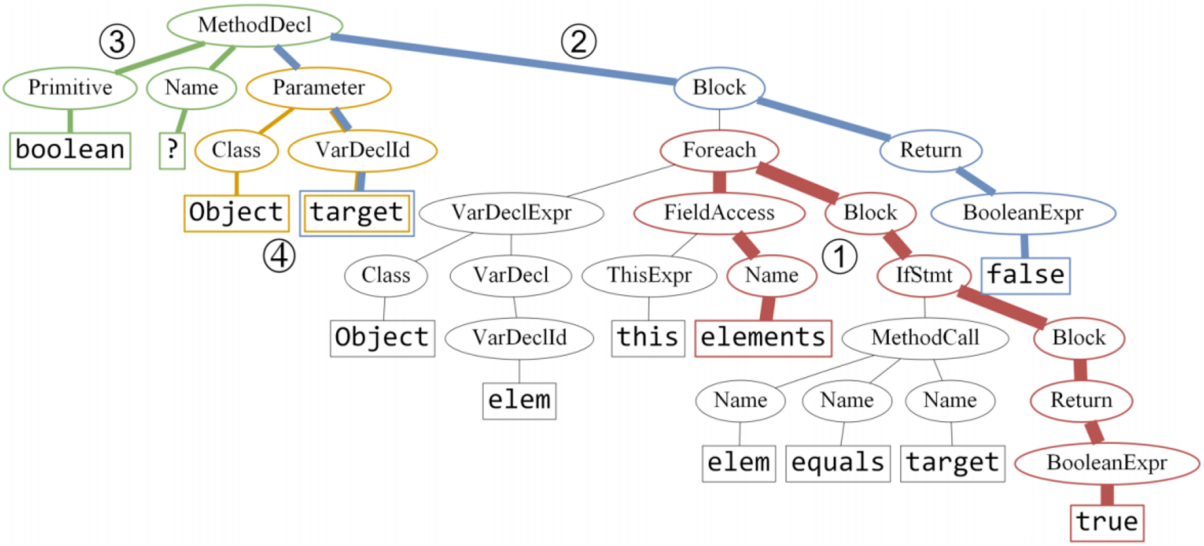


Figure 2.9. Examples of AST paths [13]. Specifically, the AST ① colored in red is extracted as $< elements, (Name \uparrow FieldAccess \uparrow Foreach \downarrow Block \downarrow IfStmt \downarrow Block \downarrow Return \downarrow BooleanExpr), true >$

function. Finally a weighted average is computed over the AST paths to generate the code embedding.

2.2 Motivation

In this work, we also intend to follow the deep learning based code embedding generation approach, for the target of cross-language code-clone detection task. That is, we aim to build a code embedding model that is capable of extracting functionality information from code fragments, and the code embeddings for similar code fragments should be close even for cross-language code pairs. Here we stress the major problem that limits the performance of previous models on the task of cross-language code-clone detection.

2.2.1 AST processing problem

Intuitively, most deep learning based tasks rely on the strong modeling power of Deep Neural Network (DNN). However, conventional DNNs can only take a vector representation as input. An advanced variant of DNN, the Recurrent Neural Network (RNN), can further take a sequence of vector representation as input. Neither of them is capable of directly processing the tree structure of an AST. Thus in previous tree based code embedding works, the nodes in ASTs are either directly merged into one vector, or linearized into a sequence, in order to be processed by a DNN or RNN. Such methods break the potential hierarchical information hidden in the tree structure.

Take the code embedding model from [13] as an example. As introduced, [13] extracts AST paths from an AST as a component of the final code representation. While generating the final code embedding, only information of AST path itself is considered, but the relationships between different AST paths and the hierarchical positions of the AST paths in the AST are ignored. For a concrete example, two identical AST paths appearing in different hierarchical level of the code share identical representation, and thus are assigned with totally same importance.

2.2.2 Graph-Neural-Network-based code encoder

In this work, we intend to address the problem of processing an AST by introducing a recent popular class of model, the Graph Neural Network.

Graph Neural Network Basically, GNN works as a node embedding generation model, specially designed to deal with graph structures. It works with a message passing mechanism. Given an arbitrary graph, first each of the nodes is assigned with an initial embedding according to their feature or randomly. Then in the message passing stage, the model computes a message representation for each node and each of its neighbors. The model updates the embedding of each node with the computed message representation. Such message passing process is repeated for several epochs until the embeddings reach a stable equilibrium.

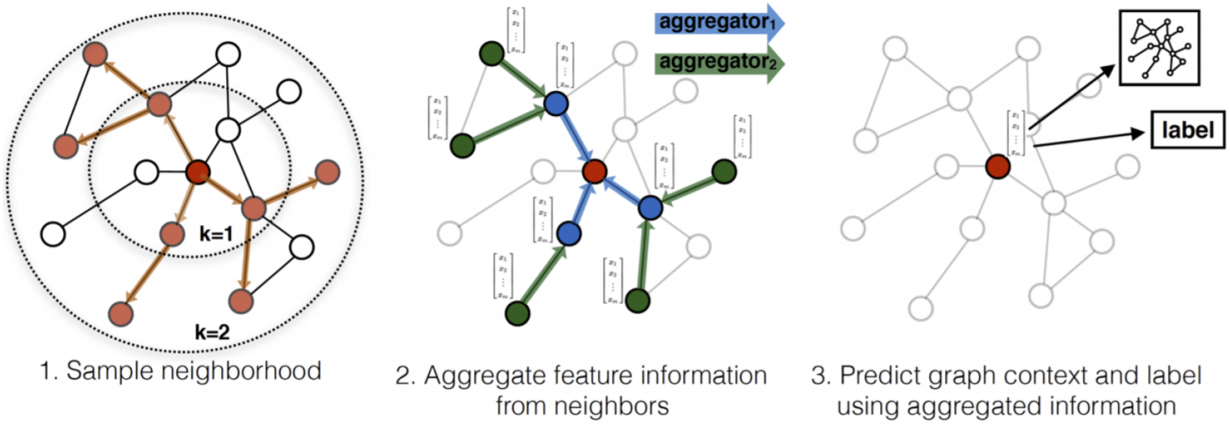


Figure 2.10. An example of a typical message passing process of GNN [14]

Further, the final node embeddings can be aggregated by a read-out function to generate the vector representation of the entire graph. With such read-out function, a GNN is extended to a graph embedding generator. Typically, the read-out function proposed by [15] consist of a fully-connected neural network that extracts information from each of the node embeddings, and an attention mechanism which learns to decide how much each of the node embeddings should contribute to the final graph embedding.

GNN for monolingual code-clone detection Wang et al. [11] is the first to introduce GNN models to the task of monolingual code-clone detection. In order to achieve the target of

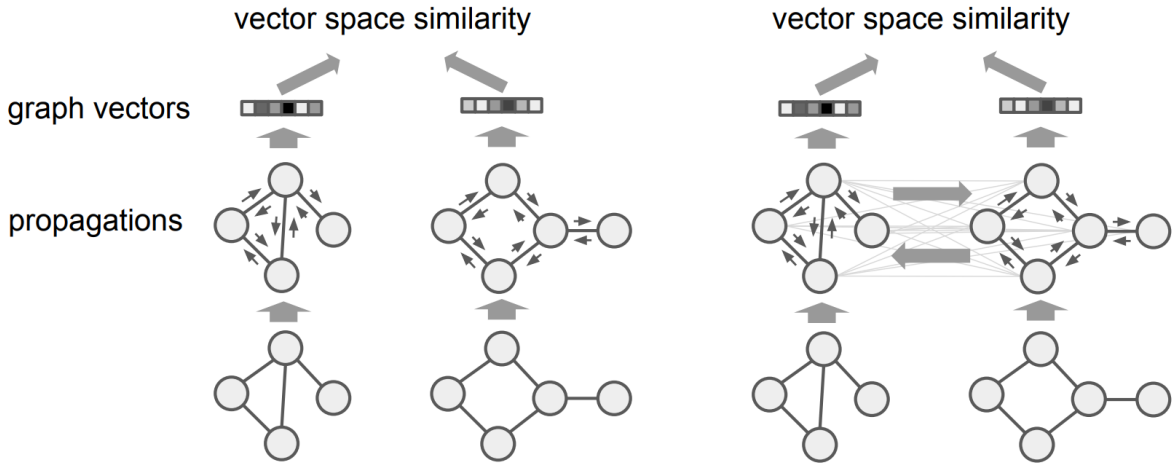


Figure 2.11. Model comparison of a typical GNN (left) and GMN (left) [16]

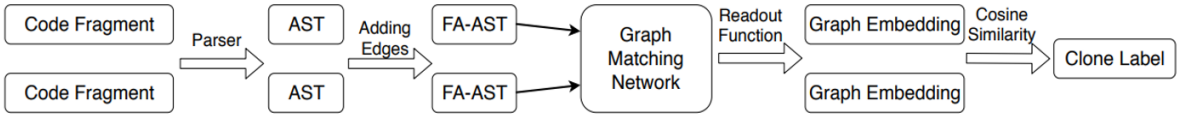


Figure 2.12. Overview of GMN based code-clone detection model [11]

code-clone detection, the authors implemented one of the variants of GNN, the Graph Matching Network (GMN) [16]. Unlike a typical GNN model that processes one single graph at a time, the GMN model possesses a Siamese structure that processes a pair of graph at the same time. Further more, a GMN model is implemented with a pair-wise attention. During the message passing stage, for each node in the pair of graphs, the pair-wise attention computes a similarity of the node to each of the nodes in the other graph, and aggregates the node embeddings into from the other graph into a context vector according to their similarity score. Such context vector is computed for each of the nodes in the pair of graphs, and is used for updating the node embeddings. By such design, the model is sensitive to detailed differences while capturing major similarities between graphs. Therefore, the GMN model is very suitable for comparing ASTs.

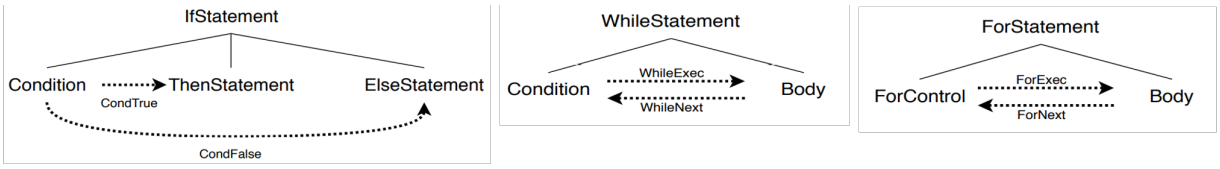


Figure 2.13. Example of modified ASTs [11]

Figure 2.12 shows the overview of the GMN based code-clone detection model. Since it's a straightforward idea to apply the GMN model to the monolingual code-clone detection task, the authors focused their effort on modifying the ASTs by adding control-flow edges and data-flow edges to improve the performance of their model, as shown in Figure 2.13.

Representing ASTs from different languages As proven by Wang et al., GNN models have great potential in extracting information from ASTs for code-clone detection tasks. However, a GNN model generates graph embedding by aggregating from node embeddings, which means the distribution of graph embedding greatly depends on the distribution of node embeddings. In the case of code embedding, it is to say that code embedding generated by a GNN model depends on node type embeddings of the ASTs. Different from the monolingual case, the ASTs comes from different programming languages in the cross-language code-clone detection task, and naturally do not share the same set of node types. Consequently, the distribution of node embeddings for different programming languages is different. Further, the code fragments are embedded separately, thus can't be directly used for code-clone detection. This is the major obstacle for generalizing a GNN based code embedding model to cross-language tasks.

2.3 Training problem

To the best of our knowledge, all code-clone detection models are trained with typical code-clone datasets. We refer to a typical code-clone dataset as a collection of code fragments (monolingual or cross-language), where each pair of code fragments is labeled as 'clone' or 'not clone' according to their similarity. However, code similarity relationships should be far more complicated than a binary label. For a simple example, as shown in Figure 2.14, if Listing funcA and Listing funcB are labeled as clone, Listing funcB and Listing funcC are labeled as clone, should Listing func and Listing funcC also be labeled as clone? In such case, a binary 'clone' or 'not clone' label is not sufficient for expressing the relationship between funcA and funcB. Such case is universal in all existing code-clone datasets, making them confusing for a deep learning model to capture the hidden similarity relationship. To this end, we intend to train the code embedding model with a

Listing 2.1.

```
def funcA():
    if condition_1:
        action_1
    if condition_2:
        action_2

    return
```

Listing 2.2.

```
def funcB():
    if condition_1:
        action_1
    if condition_2:
        action_2
    if condition_3:
        action_3
    return
```

Listing 2.3.

```
def funcC():


    if condition_2:
        action_2
    if condition_3:
        action_3
    return
```

Figure 2.14. An example for problematic clone labeling

more expressive label.

Chapter 3

Graph-to-sequence model for cross-language code-clone detection

In this section, we present our code embedding based cross-language code-clone detection model. Generally, we implement a GNN based code embedding model to generate code embeddings. We introduced embedding pre-training to deal with the cross-language issue, and optimized the training problem by training with natural language sequence labels. Consequently, we result in a graph-to-sequence model for generating code embeddings, and we aim to improve the performance of cross-language code-clone detection with the generated code embeddings.

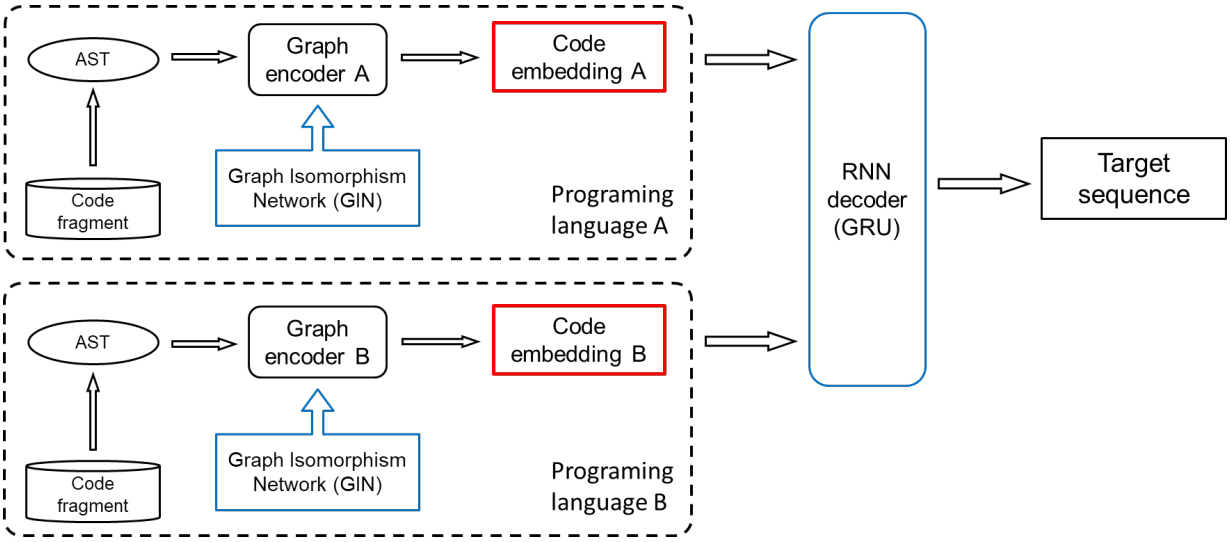


Figure 3.1. Overview of graph-to-sequence model

Model overview Here we present the overview of our implemented model, as shown in Figure 3.1. Intuitively, we follow an encoder-decoder structure to achieve the code to sequence prediction target. To be specific, we first parse the input code fragment into an AST which the model

encodes into an embedding, then the decoder takes the embedding and tries to generate the target sequence. For the encoder part, the GNN model first computes an embedding for each node in the AST by message passing, thus the embedding contains the information of the node itself and its neighbors within a certain distance. After that a readout function is applied to aggregate the node embedding into a code embedding. In training phase, this graph embedding is expected to contain sufficient information of the source code, so that the RNN decoder with Gate Recurrent Unit (GRU) can predict the target natural language sequence from it. We further implemented attention mechanism, which allows the GRU to take information from the entire input AST, to improve the sequence prediction performance. After training , in the inferring stage, we the model stops at the code embedding, and we can compute a similarity score of a pair of code fragments by computing the distance between their code embeddings.

3.1 Embedding pre-training

ASTs are heterogeneous graphs, which means each node in the graph has individual features, unlike homogeneous graphs where all nodes can be regarded as same. In order to be processed by a GNN model, the node feature of each node type has to be represented as a node embedding. In common deep learning tasks utilizing GNN, the initial node embeddings are usually assigned randomly, obeying certain distribution (e.g. normal distribution). However, as we mentioned, the generated graph embedding greatly depends on the distribution of node embeddings. In cross-language code-clone detection task, the model has to deal with two groups ASTs who don't share any node types. Our target for the code embedding model is to capture the structure similarity in ASTs from different programming language, and such difference in node types greatly influences the generated representation from the GNN model. In other words, the model represents the same structure information differently since they are captured from different programming languages.

The most straightforward idea for dealing with such problem is manually clustering similar node types from different programming languages, as Lawton et al. [7] did in their work. The problem is, on the one hand, manually clustering node types requires high level of professional knowledge on both programming language, and is very time consuming. We would like to reduce

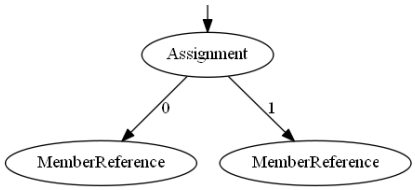


Figure 3.2. AST of Java statement 'a=b;'

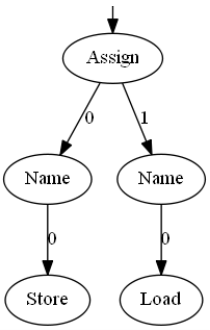


Figure 3.3. AST of Python statement 'a=b'

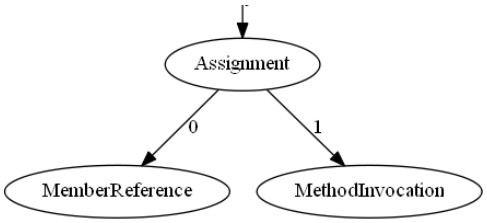


Figure 3.4. AST of Java statement 'a=b();'

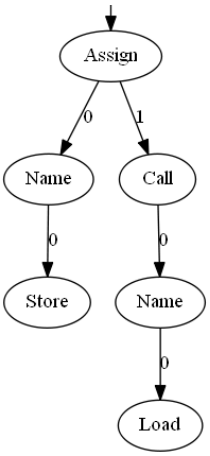


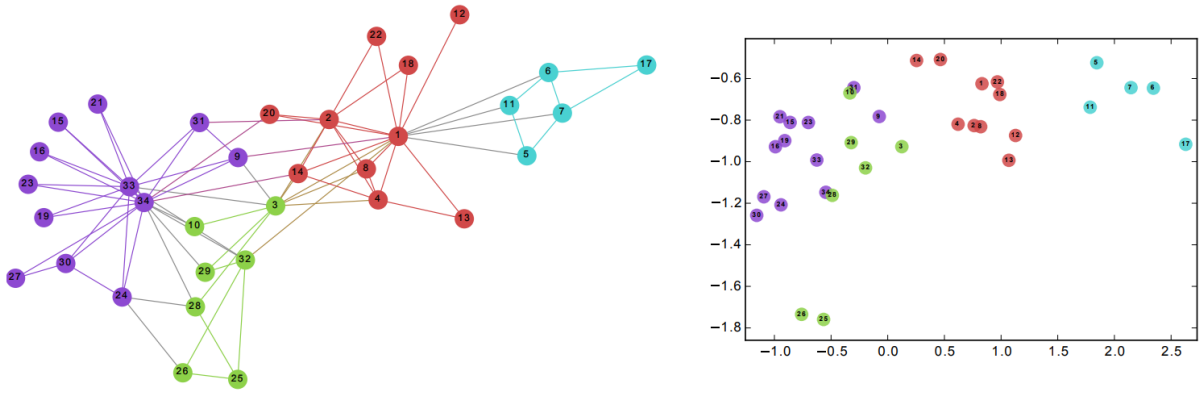
Figure 3.5. AST of Python statement 'a=b()'

such manual work with deep learning based methods. On the other hand, due to the different modeling of ASTs from different programming languages, some times a node type can not be easily clustered.

As the examples shown in Figure 3.2 and Figure 3.3, these are the AST of the same statement 'a=b' written in Java and Python. While Python distinguishes the variable on different side of the assignment operator by the 'Store' and 'Load' nodes, Java AST ignores such difference and represents both variables with 'MemberReference' node. And in Figure 3.4 and Figure 3.5, when we replaced the variable 'b' with a method invocation 'b()', Java AST replaced 'MemberReference' node with a 'MethodInvocation' node, while Python simple inserted a new 'Call' node to represent such difference. In a word, it requires heavy human work to model all those differences between cross-language ASTs, yet still resulting in sub-optimal results.

DeepWalk To address the cross-language node embedding problem, we introduce another model in GNN family, the DeepWalk model [17]. The DeepWalk model is a novel unsupervised approach for learning latent representations of vertices in a network. The latent representations generated by DeepWalk model encode social relations in a continuous vector space, which is easily exploited by statistical models.

For a given graph $G = (V, E)$, the DeepWalk model first assigns an initial embedding for each of the nodes as a typical GNN model does. Then, the model samples a random walk R from the given graph. A random walk R of length l is sampled as follows: starting from an arbitrary node $v_i \in V$, the model selects the next node v_j randomly from its neighbors N_i, and selects a further next node v_k from the neighbors of v_j, until the selected route reaches the target length l. After that, the model processes the sampled random walk R with a skip-gram model [12], regarding the random walk as a special 'sentence'. By such design, a graph structure is treated as a huge corpus of sentences composed of nodes from the graphs, and a semantic information is extracted for each node according to its neighboring relationships with other nodes.



(a) Input: Karate Graph (b) Output: Representation

Figure 3.6. A graphic example of DeepWalk model capturing social relationship from a given graph

Then a key problem arises. DeepWalk is originally designed for generating node embeddings for a single homogeneous graph, while our dataset contains multiple heterogeneous graphs (ASTs generated from different code fragments). And our idea is simple, we merge all the ASTs in our dataset into one single graph.

Applying DeepWalk on ASTs Intuitively, a homogeneous graph can also be viewed as a heterogeneous graph where all nodes in the graph are different and independent. Thus for a large dataset of ASTs, we can generate a single homogeneous graph by merging all nodes of the same type into one single node, while preserving all edges between nodes. A brief example is shown below. Figure 3.7 and Figure 3.8 represent two independent ASTs, and they are merged into the graph in Figure 3.9 according to the rule mentioned above. Similarly, all ASTs from the dataset are merged into one general graph, which contains all existing node types uniquely, and the neighboring relationships between them. Actually, when the AST dataset is large enough, such general graph contains all possible parent-child relationships allowed by the grammar of the AST, thus we name it as the **grammar graph**. On the other hand, such grammar graph can also be acquired by re-constructing AST grammar into a graph, other than extracted from a huge AST dataset.

Picking a random walk from the resulted grammar graph is approximately equivalent to extracting random walk from the collection of independent ASTs. Thus we believe applying DeepWalk model on the grammar graph would generate meaningful embeddings for nodes that represent a universal semantics of the node for an AST. For example, a ‘For’ node and a ‘While’ node would be assigned with similar embedding because they share the same set of neighbors.

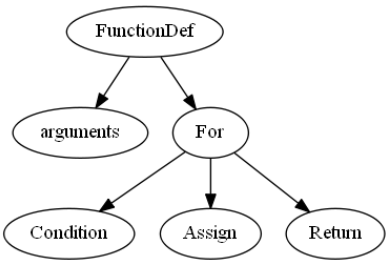


Figure 3.7. Example AST-1

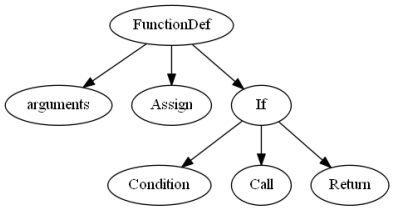


Figure 3.8. Example AST-2

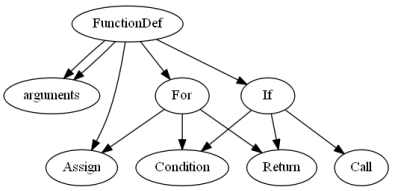


Figure 3.9. Example merged graph

Crossing language boundary Above we explained how we applied the DeepWalk model to generate node embeddings for nodes in ASTs from the same language. The idea for generalizing the embedding model to cross-language tasks, is to utilize the social relation encoding ability of DeepWalk. Intuitively, given an arbitrary graph, the nodes that share similar neighbors will be assigned with similar embeddings. Therefore, we take some special common nodes from different languages as anchors, for example the binary operator nodes like '+' and '-' that have no ambiguity. We merge the grammar graph of two different languages by merging those anchor nodes. Then during processing of the DeepWalk model, the direct neighbor of those anchor nodes, e.g. 'BinOp' node in Python AST and 'BinaryOperation' node in Java, would have similar embeddings because they share the anchor nodes as common neighbors. And such similarity will spread across the merged grammar graph.

By such design, we managed to generate similar node embeddings for AST in different languages, while greatly reducing manual work. But there is a trade off in efficiency and accuracy whenF setting the anchor nodes. The more anchor nodes we set, the more ambiguity we face when merging the nodes, thus more time and professional knowledge is required, but more similar embeddings will be generated for close node types. On the other hand, if we only set very few anchor nodes, the similarity information may fail to spread across the graph. In this work,

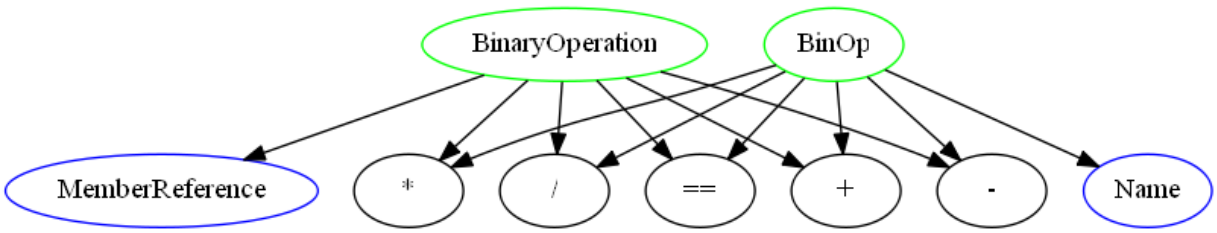


Figure 3.10. In the merged grammar graph, 'BinaryOperation' node from Java and 'BinOp' node from Python would have similar embedding, because they share many common neighbors; further, 'MemberReference' node from Java would also share similarity with 'Name' node from Python, because they share similar neighbors.

we simply set those nodes without ambiguity as anchor nodes without deep consideration, since we do not observe significant influence of anchor node numbers to the cross-language code-clone detection performance.

3.2 Graph Neural Network encoder

The key component for generating a code embedding is the GNN based AST encoder, which extracts information from the structure of an AST into a vector representation.

A typical GNN model works as follows. A given graph input is represented as $G = (V, E)$, where $V = \{v_i | i \in \{1, 2, \cdots, k\}\}$ and $E = \{(v_i, v_j) | i, j \in \{1, 2, \cdots, k\} \wedge v_i, v_j \; is \; linked\}$ denotes the set of nodes and edges in the graph respectively, $k = |V|$ is the total number of nodes in G. Before the processing starts, the GNN model first assigns each node $v_i \in V$ with an initial embedding h_i^0, so that they can be fed into a neural network. The initial embedding is assigned randomly or according to a embedding dictionary D. Next the model moves to the message passing stage. In this stage, for each node $v_i \in V$, the model computes a message representation with v_i and its neighbor $v_j \in N_i$ as $message_{i,j} = \delta(v_i.v_j)$. $N_i = \{v_j | (v_i, v_j) \in E\}$ denotes the set v_i's neighbors, and δ denotes the message function. With the messages from v_i's neighbors and the embedding of v_i itself, the model then updates the vector representation of v_i as $h_i^{t+1} = \phi(h_i^t, aggr(\{message_{i,j}^t | v_j \in N_i\}))$. Here, ϕ refers to an updating function, $aggr$ is a aggregation method that collects message from v_i's neighbors, and t denotes the number of current message passing stage. This message passing stage is repeated several times until each node representation collects enough information from a certain range of its neighbors. Finally in the output stage, a graph embedding is computed as $g = \theta(h_i^{-1} | i \in \{1, 2, \cdots, k\})$, where θ is a pre-defined read-out function that aggregates information from the entire graph, and h_i^{-1} denotes the final vector representation of node v_i.

Along the entire working flow of a GNN model, there are several key parts requiring further discussion: the message function δ, the updating function ϕ, the neighbor aggregation method $aggr$, and the read-out function θ. Basically, we follow the model structure of Graph Isomorphism Network, which claims to be the most expressive GNN model.

3.2.1 Graph Isomorphism Network

In [18], the authors proved that the upper bound of the expression ability for a GNN model is equal to an algorithm called Weisfeiler-Lehman test (WL test) [19]. In other words, a GNN model is at most as powerful as the Weisfeiler-Lehman test in distinguishing graph structures.

Weisfeiler-Lehman test The graph isomorphism problem asks whether two graphs are topologically identical. This is a challenging problem: no polynomial-time algorithm is known for it yet. The Weisfeiler-Lehman test of graph isomorphism is an effective and computationally efficient test that distinguishes a broad class of graphs. Its 1-dimensional form, "naïve vertex refinement", is analogous to neighbor aggregation in GNNs. The Weisfeiler-Lehman test iteratively (1) aggregates the labels of nodes and their neighborhoods, and (2) hashes the aggregated labels into unique new labels. The algorithm decides that two graphs are non-isomorphic if at some iteration the labels of the nodes between the two graphs differ.

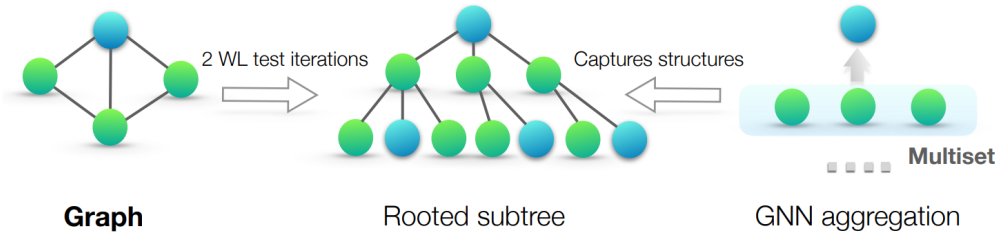


Figure 3.11. An overview of GIN's theoretical framework. Middle panel: rooted subtree structures (at the blue node) that the WL test uses to distinguish different graphs. Right panel: if a GNN's aggregation function captures the full multiset of node neighbors, the GNN can capture the rooted subtrees in a recursive manner and be as powerful as the WL test. [18]

For a GNN model, being as powerful as the Weisfeiler-Lehman test means being able to embed any different graphs into different embeddings, so that different graphs can be distinguished by their embeddings, i.e. the GNN model is injective. Based on their mathematical analysis, the following conclusion is reached:

A GNN model maps any graphs G_1 and G_2 2 that the Weisfeiler-Lehman test of isomorphism decides as non-isomorphic, to different embeddings if the following conditions hold:

1. *The updating function ϕ and the neighbor aggregation method $aggr$ are injective.*
2. *The graph level read-out function θ which operates on the node features is injective.*

To be more specific, GIN models the updating function and the neighbor aggregation method $\phi \circ aggr$ together with one Multi-Layer Perceptron (MLP), because MLPs can represent the composition of functions. Thus each iteration of the message passing stage works as

$$h_i^t = MLP^t(h_i^{t-1} + \sum_{v_j \in N_i} h_j^{t-1})$$

note that the MLP for each message passing iteration is modeled independently.

For the graph level read-out function, GIN implemented as

$$g = CONCAT(SUM(\{h_i^t | i \in \{1, 2, \cdots, k\}\}) | t \in \{1, 2, \cdots, T\})$$

which concatenated the summation of all node embeddings of each message passing iteration. While in our work, we replaced the read-out function with the one proposed in [15]

$$g = tanh(\sum_{i=1}^{k} sigmoid(MLP_w(h_i)) \odot tanh(MLP(h_i)))$$

This read-out function introduced an attention mechanism by replacing the sum of node embeddings with a weighted sum, where the weight is learned by the MLP_w linear neural network.

Besides, this read-out function only focuses on the final state of node embeddings in order to reduce computation cost. And on the other hand, paying attention to each of the iterations does not significantly improve the performance.

3.3 Recurrent Neural Network decoder

Now we have acquired a powerful GNN based model for code embedding generation. The following question is how to train the embedding model so that it generates meaningful embeddings, especially for the task of code-clone detection. As we discussed in Section 2.3, binary clone labels are not suitable for training a code-embedding based code-clone detector, for the similarity relationships between code fragments are far more complicated.

In this work, we intend to exploit code similarity potentially buried in other labels, specifically, natural language sequence corresponding to the code fragment. On the one hand, dataset that assigns code fragments with meaningful and closely related natural language sequence is universal. On the other hand, based on promising natural language processing techniques, it's possible to dig discrete similarity labels from the natural language labels. To be more specific, we train a model to predict the corresponding sequence from a code fragment, thus teaching the model to dig information from code. Therefore, we propose train our model with a graph (AST) to sequence task. To be specific, we feed the code embedding generated by the GNN encoder into a Recurrent Neural Network (RNN) decoder to generate a meaningful natural language sequence.

3.3.1 Recurrent Neural Network

Recurrent Neural Network (RNN) is a class of artificial neural networks that specializes in dealing with sequences, A typical Deep Neural Network model consists of multiple hidden layers, where the output of a previous layer is fed to the next layer as input. However, each RNN model possesses only one hidden layer. During the forward propagation stage of an RNN model, the output of its hidden layer, which is also referred to as RNN's hidden state, is fed back again into the hidden layer, together with the next input from the sequence.

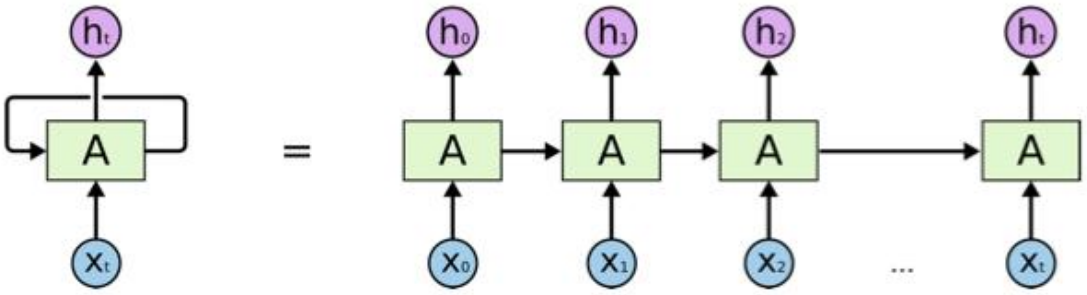


Figure 3.12. Typical RNN model

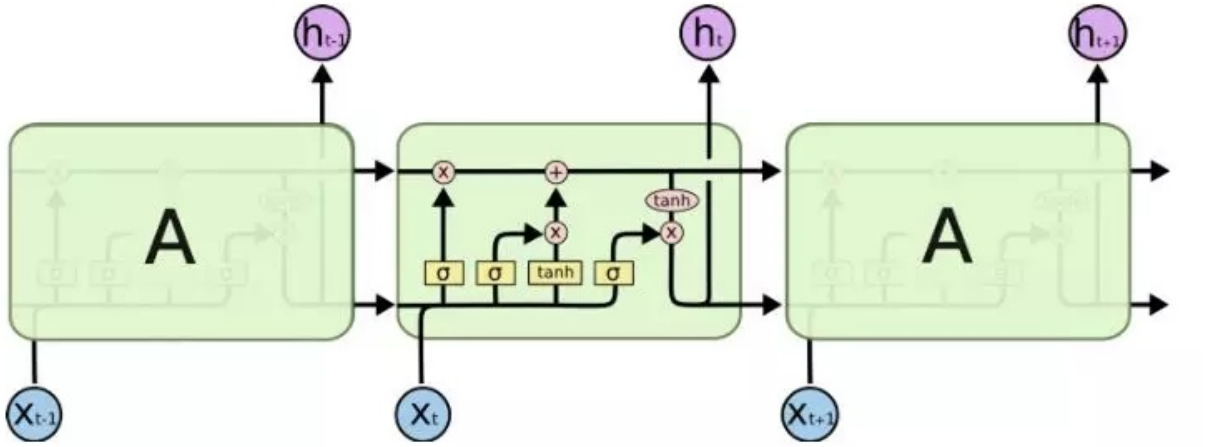


Figure 3.13. Structure overview of a LSTM model [20]

$$h^{t+1}, hidden\_state^{t+1} = hidden\_layer(input^t, hidden\_state^t)$$

In this way, the only single layer of an RNN model is re-used 'recurrently'. Unlike typical DNNs that can only aggregate all information from a sequence with out any order, the RNN model takes in the input sequence tokens one at a time, thus the output of each time step contains the ordered information from all previous inputs. Figure 3.12 presents the structure of a typical RNN model. A denotes the hidden layer of RNN, $x_0, x_1, \cdots, x_t$ are the tokens of the input sequence, i.e. the input of RNN in each time step. The real output h of each time step and the *hidden_state* used for the next step can be same or different, according to the detailed structure of the hidden layer.

Though the general structure of an RNN model is simple, the detailed structure of the only hidden layer has been carefully studied with many different designs. A well known example is the LSTM, as shown in figure 3.13. A the hidden layer of an LSTM model contains many gate units that control the data flow of the input data and the previous hidden state. Such gate units are designed for better dealing with memorizing long sequences and solving the problem of gradient exploding and vanishing gradient during training.

In our work, we implement a light-weight version of LSTM, the GRU model, as the hidden layer of our RNN model. Given the previous hidden state h^{t-1} and the new input x^t, the GRU first passes them through a reset gate r to compute a reset parameter

$$r = sigmoid(MLP(h^{t-1}, x^t))$$

Then the previous hidden state H^{t-1} is weighted by the reset parameter r, and used to compute a temporary new hidden state h' with the new input x^t

$$h' = tanh(MLP(r \odot h^{t-1}, x^t))$$

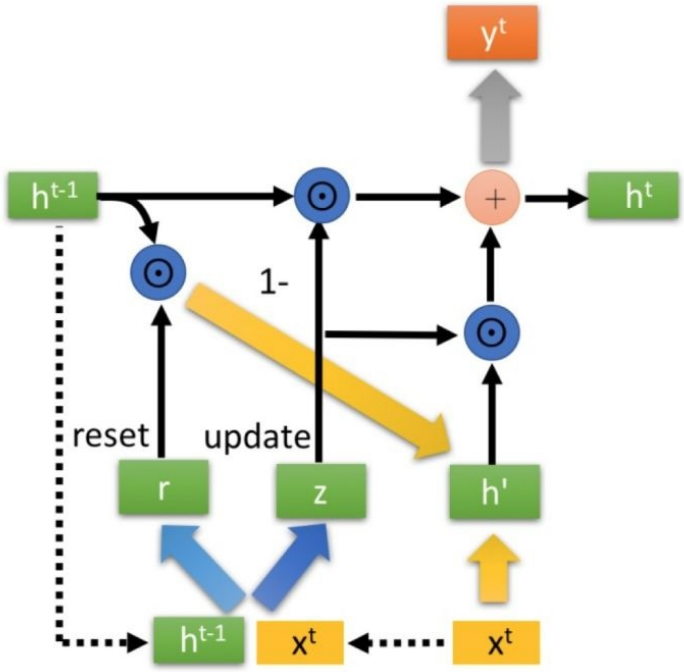


Figure 3.14. Structure overview of GRU hidden layer [21]

Meanwhile, h^{t-1} and x^{t-1} is passed through a forget gate to compute a forget parameter z

$$z = sigmoid(MLP(h^{t-1}, x^t))$$

The forget parameter z is used to leverage how much the previous hidden state h^{t-1} and the temporary new hidden state h^{prime} should contribute to the final new hidden state h^t

$$h^t = (1 - z) \odot h^{t-1} + z \odot h'$$

Finally, the new hidden state h^t is used as both the output of the current time step and the input hidden state of the next time step.

The gate units helps the GRU model to deal with previous information differently according to the new input, thus it's more powerful in dealing with long sequences than RNN that simply uses a fully-connected neural network as hidden layer. Compared to the LSTM design, GRU uses one less gate unit than LSTM, thus consuming less computation power, while reaches approximately same level of performance as LSTM.

3.3.2 RNN decoder

In this work, we build an RNN decoder using GRU as hidden unit, and we refer to the RNN decoder as 'RNN decoder' in the following of this thesis. When working as an encoder, an RNN model takes a blank vector (usually zero vector) as input of the first time step, i.e. a blank initial

hidden state. Then while taking in the tokens of the input sequence one by one, the hidden state gradually stores information of previous tokens, and finally becomes a vector representation of the entire sequence. Oppositely, an RNN decoder takes in a initial hidden state that represents the entire target sequence, and tries to predict the tokens of the target sequence during each time step. Thus in our work, we use the code embedding generated from the GNN encoder as the initial hidden state of the RNN decoder, and train the entire graph to sequence model by asking the RNN decoder to predict the target sequence. We train the model with a cross entropy loss by computing the difference of the predicted token and the target token.

3.3.3 Attention

However, power of RNN learning semantic knowledge from sequences is too strong, while the information provided by the code embedding is limited. Consequently, the RNN decoder gets over-fitted within very few training epochs, and predicts target sequence mainly based on the semantic regularities it captured from the sequence dataset. In such case, the graph to sequence prediction model actually degenerates into a graph to word prediction model, where only the first word is predicted according to the input graph, while the rest of the sequence is irrelevant to the graph embedding.

Attention mechanism The attention mechanism [22] is a widely used technique in natural language processing, which enables the RNN decoder to utilize the information from input in every time step of target sequence prediction. To be specific, the vector representations of input elements are weighted-summed into a context vector, and fed into the hidden layer to help predict the next output, where the weights of input elements is computed by the attention model according to the previous hidden state.

Graph-wise attention Though attention mechanism is originally used in sequence to sequence tasks, its idea can be easily generalized to our graph to sequence model. Here we introduce the

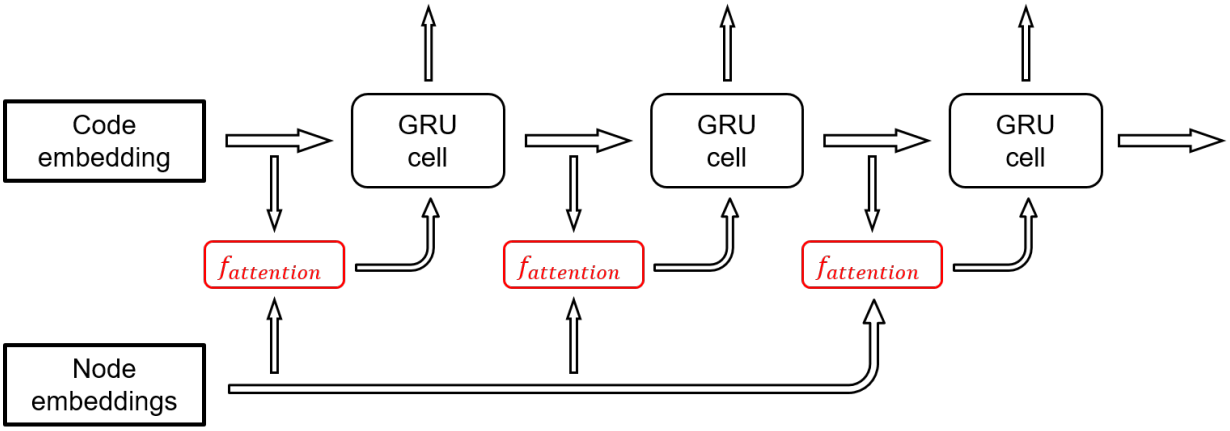


Figure 3.15. Graph-wise attention

graph-wise attention mechanism, which enables the RNN decoder to utilize information from the input AST during prediction of each target word. A brief overview is shown in Figure 3.15. During the prediction time step t, given the previous hidden state h^{t-1} and node embeddings $v_i \in V$ from the input graph, an importance score of each node is computed by the attention model as

$$w_i^{t-1} = MLP(h^{t-1})\dot{v}_i$$

which works as a correlation function. The importance scores are normalized by a standard softmax function into the weight of each node embedding

$$\alpha = \frac{\exp(w_i)}{\sum_j \exp(w_j)}$$

Then the node embeddings are weighted summed into a context vector

$$ctx^{t-1} = \sum_{v_i \in V} \alpha_i v_i$$

Finally, the context vector is used to aid the prediction

$$h^t, hidden\_state^t = GRU(hidden\_state^{t-1}, ctx^{t-1}, input^t)$$

We implement the attention mechanism for two major reason:

- By forcing the model to utilize more information from node embeddings, the contribution of RNN hidden layer to the prediction task is reduced, thus more information is learned and stored by the code embedding.
- Though RNN possesses strong power on dealing with sequence data, it becomes difficult to predict a long sentence only with the hint of an initial hidden state. By utilizing the information from AST in every prediction step, the RNN can generate sequence that is more related to the input code fragment.

Chapter 4

Evaluation

In this section, We present all the results from our experiments to qualitatively support our contribution. First of all, we discuss about the modifications we made on the dataset, followed by some detailed model settings. Next we display the performance of our model on our final target, the cross-language code-clone detection task, and compare it to a recent work. Then, we show some observed results as evidence of the effectiveness of our model.

Data preprocessing The original dataset contains 342 Python files and 104 Java files, each named with its corresponding programming contest question. Since the target of our model is to detect cross-language code-clones, we first removed the unpaired code files from both language, so that for any programming contest question, both answers written in Python and Java exist in the dataset. Next, for the rest of the data, we found many of them contains multiple implementation with different algorithms hidden in code comments. We extract those extra implementations and named them with the corresponding question. Besides, we noticed some implementation for the same question does not utilize same algorithm, we modified the python code to make sure they utilize similar algorithm or remove the question-answer pair if modification is not possible. Finally, we modify the question names so that the name better represents the corresponding code fragment (e.g. add code detail to the name of code fragments that implement the same question with different algorithms).

Listing 4.1. Original python code fragment

```
while l1 or l2:
    val = carry
    if l1:
        val += l1.val
        l1 = l1.next
    if l2:
        val += l2.val
        l2 = l2.next
    curr.next = ListNode(val % 10)
    curr = curr.next
    carry = val / 10
```

Listing 4.2. Original Java code fragment

```
while (p != null || q!= null) {
     int x = (p != null) ? p.val : 0;
     int y = (q != null) ? q.val : 0;
     int digit = carry + x + y;
     carry = digit / 10;
     curr.next = new ListNode(digit % 10);
     curr = curr.next;
     if (p != null) p = p.next;
     if (q != null) q = q.next;
 }
```

Listing 4.3. Modified Python code fragment

```
while l1!=None or l2!=None:
    x = l1.val if l1!=None else 0
    y = l2.val if l2!=None else 0
    digit = carry+x+y
    curr.next=ListNode(digit%10)
    curr=curr.next
    if l1!=None:
        l1=l1.next
    if l2!=None:
        l2=l2.next
```

Listing 4.1 - 4.3 is an example of our modification on the dataset. Python code fragment Listing 4.1 and Java code fragment Listing 4.2 are the origin code fragments in the dataset labeled as cross-language code-clone. They are implemented for the same LeetCode question and they achieve same functionality regardless of the difference in their code structure. In a monolingual code-clone detection task, the similarity in such different code pairs should be learnt

by the detector. But in our case, we are facing a cross-language task, our model is not strong enough to capture such divergence. Besides, the hint of such difference is not included in their labels, i.e. the LeetCode question. Thus we modified the Python code to share same structure with the Java code, to reduce the learning difficulty for our model.

The resulted dataset contains 114 code pairs from Python and Java. Each code fragment in the dataset is labeled with its corresponding programming contest question, which is also used as its file name. Each pair of code fragments from different programming languages is labeled as cross-language clones, while any other pairs are labeled as not clones. We utilize both labels during the evaluation of our model.

Model settings Apart from node embedding for input ASTs and word embedding used in sequence prediction, all parameters in our model are initialized with the default setting of PyTorch. Node embeddings are generated with DeepWalk model, we download the code from their github repository and used the default setting. The dimension of node embedding is set to 400. For the RNN decoder, we set the dimension of hidden state to 500, which means the code embedding generated from GNN encoder should also have 500 dimensions. We use the word embedding from Python package pytorch-pretrained-bert. The times of message passing in the GNN encoder is set to 10. We optimize the model with Adam optimizer, with $weightdecay = 5e-4$. The learning rate is set to $1e-4$, decayed by a factor of 0.9 every 50 epochs.

4.1 Cross-language code-clone detection

Graph-to-sequence model performance Though it is not our final target to correctly predict the question name for a LeetCode answer, we still observe and evaluated the performance of our graph-to-sequence model. As shown in Figure 4.1, the loss of the model continuously decreases during training, approaching 0. The prediction precision of target sequence reaches 100% when loss reaches about 0.01. However, no correct prediction on the testing dataset is observed during the training stage, which indicates severe over-fitting. This is mostly because our dataset is too small, while the regularity hidden in the target sequences is too complicated, the model gets over-fitted before it captures the semantic pattern from the target sequence set. Thus we conclude this model is unsuccessful on the graph-to-sequence prediction task.

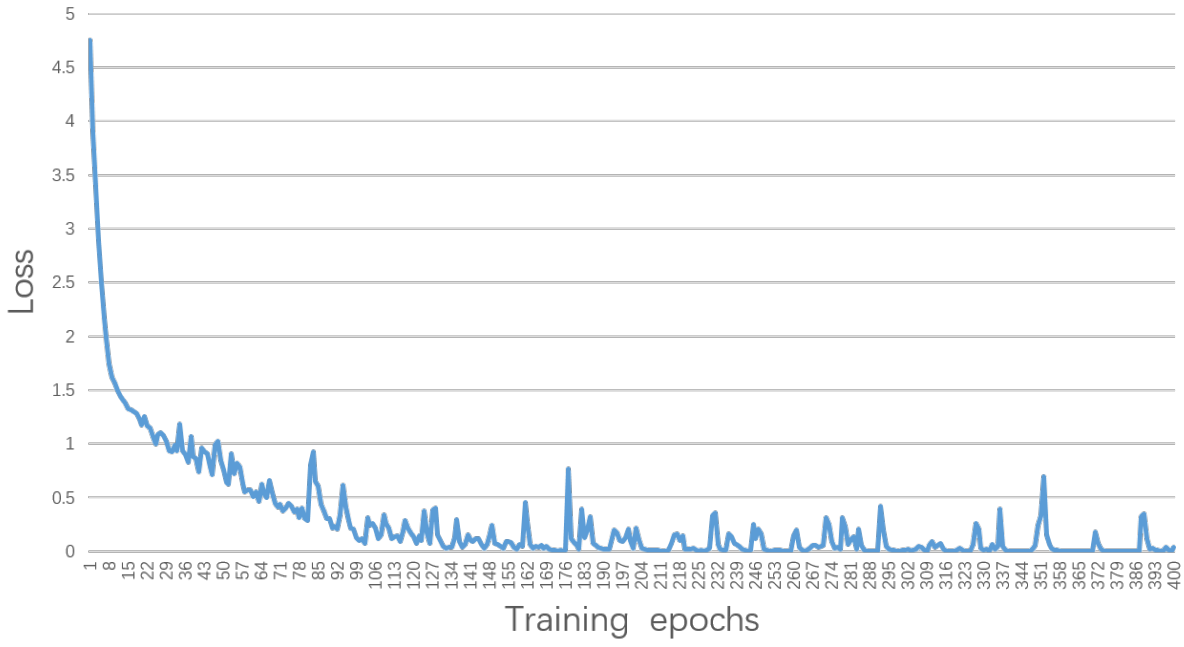


Figure 4.1. Loss curve for during training

Listing 4.4. 'Find anagram mappings'

```
def anagramMappings(self, A, B):
    val_index = {}
    ans = []
    for i, n in enumerate(B):
        val_index[n] = i
    for n in A:
        ans.append(val_index[n])
    return ans
```

Listing 4.5. 'Find the difference'

```
def findTheDifference(self, s, t):
    res = 0
    for i in range(len(s)):
        res = ord(t[i])res = ord(s[i])
    return chr(res)
```

Sequence-based code-clone detection During our observation on the result of graph-to-sequence model, we noticed rare cases where part of the target sequence in testing dataset is correctly predicted. In most of such cases, the model manages to predict the first word in the sequence, especially when the first word is a verb, and has appeared in the training dataset. The code fragments in Listing 4.4 and Listing 4.5 show a typical example of such cases. Code fragment labeled with 'Find anagram mappings' appeared in the training dataset, while code fragment labeled with 'Find the difference' appeared in testing dataset, and is predicted as 'Find anagram mappings'. Conservatively speaking, we can infer that our model indicates that the two code fragments are similar by assigning them with the same function name. Further, we notice that the two code fragments are similar in for loop followed by return statement, which is exactly the behavior of 'find'. We may even guess that our model manages to capture the semantic meaning of natural language task.

Basically, such cases only happen with our model is trained for about 100 epochs, thus we decide 100 epochs is when our model captures some regularity from the code fragments yet not over-fitted to the training dataset. We checked the sequence prediction of our model when trained for 100 epochs, and notice that it is not rare case that the model predicts the same sequence for the clone pair from Python and Java. Though the predicted sequence is different to the sequence label, predicting same sequence for similar code fragments from different programming languages already matches our target of cross-language code-clone detection. We regard the code pairs with same predicted sequences as detected clones, and the model reaches about 65.5% recall and 30.2% precision on cross-language code-clone detection.

Embedding based code-clone detection Then we evaluate our model by detecting cross-language code-clones by comparing code embeddings. We used the Euclidean distance between code embeddings to measure the similarity of a code pair. Since our model is not directly trained with a code-clone detection task, we evaluated the cross-language code-clone detection performance on the entire dataset instead of part of it. According to our observation, the distance of code embeddings between most clone pairs a less than 5, so we set the threshold to 5. However, our model does not perform well in this evaluation. Though setting the threshold to 5 could classify 25 of the 29 cross-language code-clone pairs, there are more non-clone pairs that have closer distances than the clone pairs, leading to an extremely low precision on detected cross-

Table 4.1. Cross-language code-clone by comparing predicted sequence

Results	
Clone pairs	29
Total code pairs	$841 (= 29^2)$
True positives (TP)	19
False positives (FP)	44
True negatives (TN)	768
False negatives (FN)	10
Recall$(= TP/(TP + FN))$	65.5%
Precision$(= TP/(TP + FP))$	30.2%

Table 4.2. Cross-language code-clone by comparing code embeddings

	Results
Clone pairs	29
Total code pairs	$841(=29^2)$
True positives (TP)	25
False positives (FP)	315
True negatives (TN)	497
False negatives (FN)	4
Recall$(=TP/(TP+FN))$	86.2%
Precision$(=TP/(TP+FP))$	7.35%

language code-clones. Meanwhile, there is not obvious improvement on precision when we lower the threshold to make the model more strict. The detailed result is shown in Table 4.2.

Finally we compare the performance of our model with the work from Daniel et al. [8]. Our model only showed slight improvement in precision of detected clones when judging clones by the predicted sequences.

Table 4.3. General performance and comparison

models	recall	precision
Sequence-based	65.5%	**30.2%**
Embedding-based	76.3%	7.35%
Perez et al.	**90%**	19%

4.2 Code embedding generation

After the cross-language code-clone detection task, we then qualitatively evaluate the performance of our model as a code embedding generator. There is no mathematical criteria for evaluating the quality of embeddings, so the following results are mainly evaluated by manual observation.

4.2.1 Pre-trained node embedding

First we assess the effectiveness of our pre-training method for cross-language node embeddings. All the embeddings presented below are generated by the online tool Embedding Projector [23].

Figure 4.2 is a general distribution of all nodes from the merged grammar graph of Python and Java. We pick some typical node embeddings as examples. As highlighted in Figure 4.3, we can observe that the 'While' node from Python ASTs and the 'WhileStatement' node from Java ASTs are embedded closely. In fact, such node pairs should be assigned with exactly identical embeddings if they share totally same neighbor set. But since we only merged part of the nodes as anchor nodes, the rest un-shared neighbors finally lead to the distance between their embeddings. We pick 'BinOp' node from Python ASTs and 'BinaryOperation' node from Java

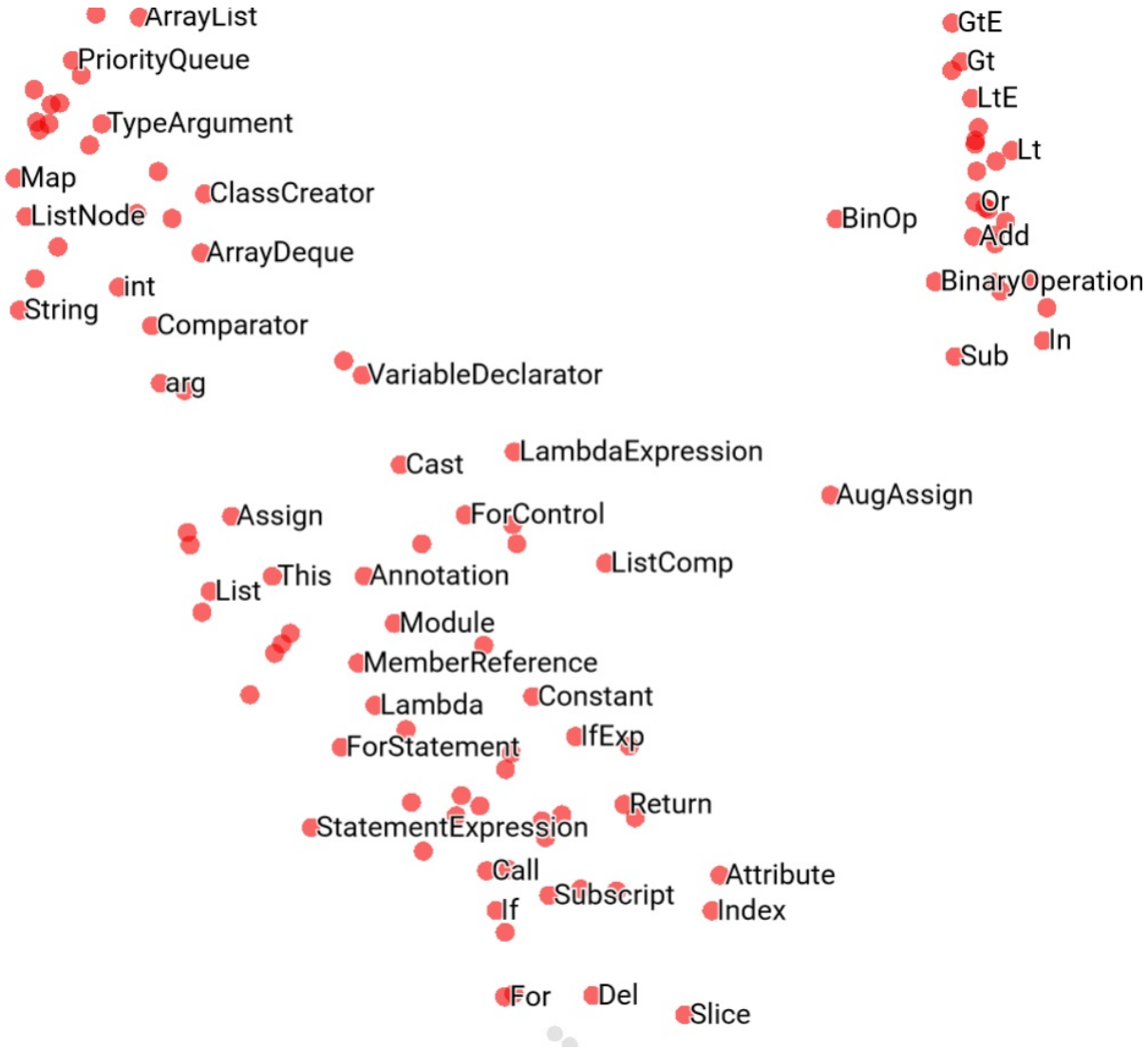


Figure 4.2. Cross-language node embedding

Figure 4.3. Embeddings for node 'While' from Python and 'WhileStatement' from Java

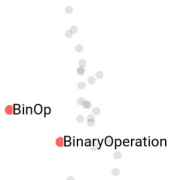


Figure 4.4. Embeddings for node 'BinOp' from Python and 'BinaryOperation' from Java

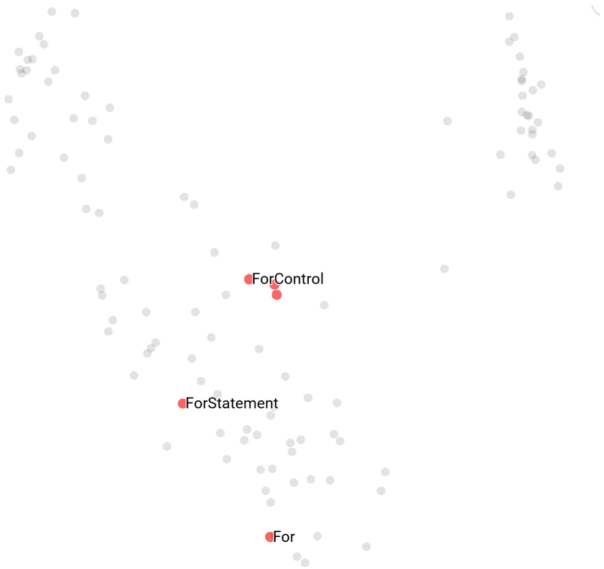


Figure 4.5. Embeddings for node 'For' from Python and 'ForStatement' from Java

ASTs as the next example. According to the general embedding distribution shown in Figure 4.2, we notice the binary operator nodes (e.g. Greater than(GT), Less than (LE), etc.) focus on the top right corner of the 2D latent space. As expected, their most common direct neighbors, 'BinOp' node and 'BinaryOperation' node, are embedded close to them. Since all these binary operator nodes are set as anchor nodes, the 'BinOp' node and 'BinaryOperation' node share more common neighbor than the 'While' node and 'WhileStatement' node, thus are embedded closer. There are also exceptions. As shown in Figure 4.5, the 'For' node from Python ASTs and the 'ForStatement' from Java ASTs are not embedded as close as the previous examples. According to our observation, these two nodes almost share no common neighbors in the merged grammar graph, indicating that it might be more appropriate to merge them as anchor nodes to guide the embedding of their neighbors.

It is worth mentioning that these node pairs are actually selected as anchor nodes in our other experiments. We only separate them to generate this demonstration.

4.2.2 Code embeddings

In this section, we present a qualitative evaluation of our code embedding generation model. Figure 4.6 shows a general distribution in a 2D latent space of the code embeddings generated with

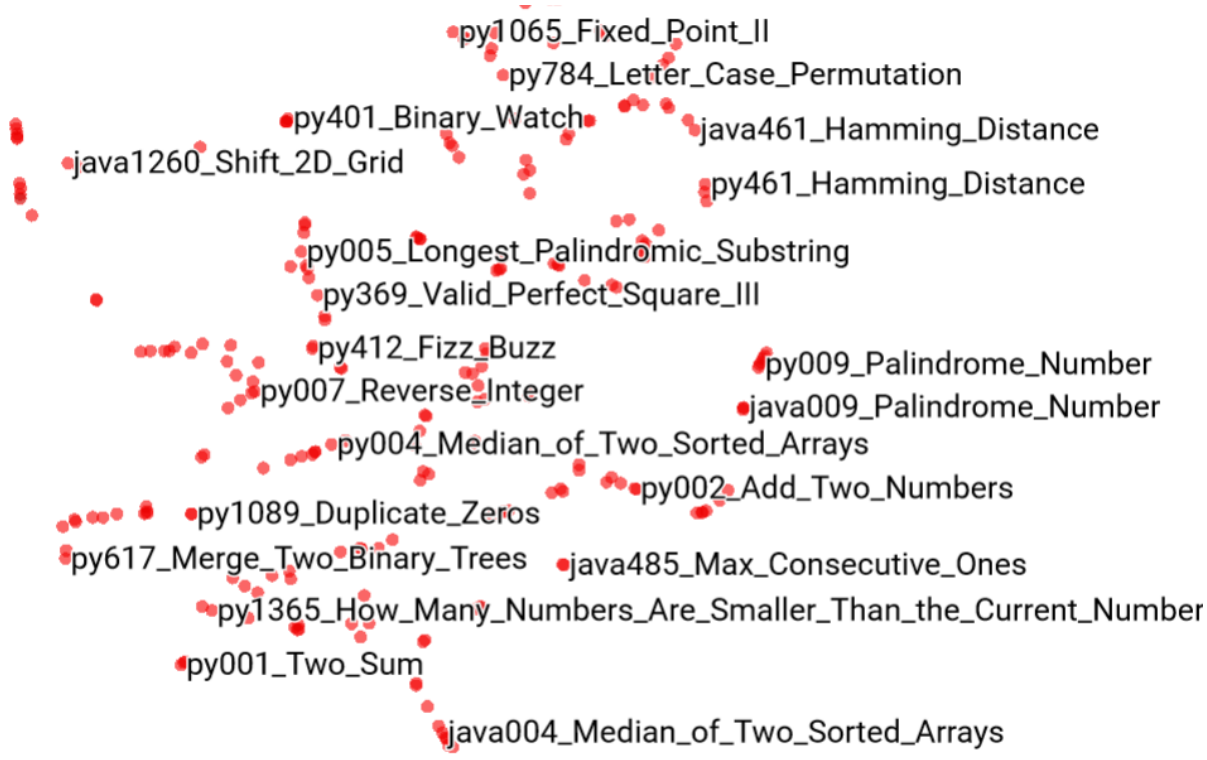


Figure 4.6. General distribution of code embeddings

Figure 4.7. Code embedding for code fragment 'Find_the_Difference'

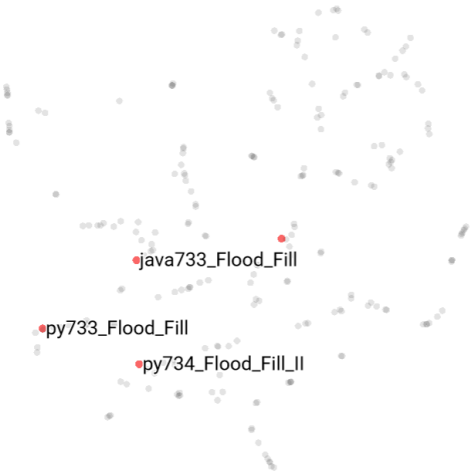


Figure 4.8. Code embedding for code fragment 'Flood_Fill'

our model. From this general distribution we can already notice some cross-language code-clone pairs that are embedded closely, e.g. 'Hamming_Distance' and 'Palindrome_Number'. Then we take some typical code embeddings to demonstrate the effectiveness of our code embedding model.

As shown in Figure 4.7, the code fragments written in Python and Java are embedded closely

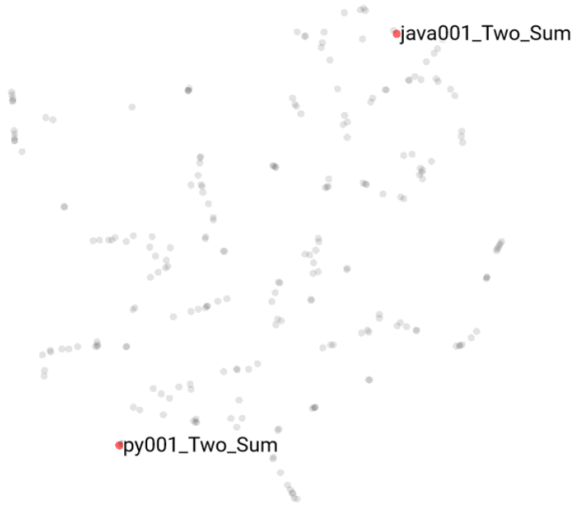


Figure 4.9. Code embedding for code fragment 'Two_Sum'

on the top right corner of the latent space. However, considering that we modified the dataset so that cross-language code-clone pairs share exactly same code structures, these code pairs are not embedded close enough. Ideally, given an arbitrary code fragment from our dataset, its clone pair should be encoded closer than any other code fragment, since there is no code fragment that is more similar than the clone pair. This explains the low precision on detected clones of our model. In the next example in Figure 4.8, we noticed that not only 'Flood_Fill' from both Python and Java are embedded close to each other, the code fragment 'Flood_Fill_II' which contains another solution for the flood fill problem, is also embedded in a close position. Such similarity in code fragments is originally contained in the question name. Since the code embedding generating process does not involve the RNN model, we can conclude that our GNN encoder managed to capture similarity information from the AST during training. Finally, Figure 4.9 shows a failure case. The clone pair are embedded to the opposite corner of the latent space.

Chapter 5

Conclusion

5.1 Summary

In this work, we proposed a Graph Neural Network (GNN) based code embedding model for cross-language code-clone detection. Specifically, we train our model with a graph (AST) to sequence task instead of typical code-clone detection task. We dig information from the sequence label with an RNN decoder, to guide the embedding process of the GNN encoder. In general, our model follows a GNN-encoder-RNN-decoder structure, implemented with attention mechanism to improve the performance.

One important feature is that it introduces the Graph Neural Network (GNN) model to the cross-language code-clone detection task. On the one hand, many previous works on the AST-based cross-language code-clone tasks lack an appropriate approach to deal with the tree structure of an AST. Some of them manually defined AST comparison rules which is time consuming and requires much professional knowledge, while others linearized the ASTs so that they can be processed by conventional deep learning models, potentially breaking the hierarchical information. In our work, we introduced the GNN models which is capable of directly processing ASTs, and have a strong power of extracting structural information from graph (tree) structures.

On the other hand, though there has been attempts on applying GNN model to monolingual code-clone detection task [11], their model faces two major problems. First, the Graph-Matching-Network they introduced is implemented with a pair-wise attention mechanism, which boosts the model's capability of capturing minor differences. However, such design fails to generalize to cross-language task, because the basic structures of the input ASTs are different. Second, their model works in a pair-wised way of inputting a pair of code fragments and outputting a predicted clone label, thus the code embedding generated by the GMN depends on the other input. Such embedding cannot be used independently. While in our work, we first propose to generate grammar graph from ASTs, then applies the DeepWalk model to the merged grammar graph to generate node embeddings that cross the language boundary. Besides, we train our model with natural language labels which potentially contains more complicated information than binary clone labels.

In the evaluation stage, our results show that our model doesn't work well on the task of cross-language code-clone detection, yet it still achieved slight improvement on precision of detected

clones with the help of a RNN decoder, compared to a previous work. As a code embedding model, our qualitative evaluations show that both our node embedding model and code embedding model achieved some result. They both managed to approximately embed the nodes and code fragments into a latent space according to their social relationship or semantics. Most importantly, they both managed to break the language boundary and embed node and code fragments from different programming languages into the same latent space.

5.2 Threats of validity

There are many details in the structure of our model, that is not carefully mentioned in the thesis but may have influence on our presented result, we list them as follows.

dataset Though we are training our model with a sequence prediction task, the cross-language dataset is still a typical code-clone dataset by nature. The model is capable of capturing same regularities for cross-language code fragments, because the code-to-sequence dataset is symmetric for the two target languages. In other words. the model probably would not work if the code-to-sequence datasets for the two target languages are irrelevant. Though we may not be able to find cross-language code-to-sequence dataset that possesses same regularities, but we can always label a cross-language code-clone dataset with sequence labels, so that our model can still work. Anyway, we believe a complicated label with more information is always better than the binary clone label.

Model details For many detailed settings of our model, for example the selection of aggregation function or read-out function, we basically followed the description from other related works. Same for the choice of hyper parameters, we chose them based on observation and experience. Due to the limit of time, we did not conduct sufficient experiments to choose the best combination of the model settings.

Performance comparison The results from Daniel et al. was picked directly from their published paper. We did not re-implement their model, nor did we run their code in our environment, due to limit of time.

5.3 Future work

Currently our code embedding model does not perform well on the task of cross-language code-clone detection. But we still believe in the potential of GNN models extracting information from ASTs while crossing the language boundary. There are still plenty of space of improvement for our model.

First we still want to stress the importance of a proper dataset. Though our dataset has already been modified to fit our model, it is still too small and limited. Besides, the information contained in the natural language labels is too abstract, increasing the learning difficulty of our model. Thus on the one hand, we would further look for a better dataset that suits our

expectation. On the other hand, we would test our model on more general datasets to test its ability of generalization.

Secondly, as we mentioned in Section 4.1, our model has the potential of capturing the relationship between code behavior and natural language expressions. Such feature has not been yet rigorously proved. Further exploring the model's potential in such feature may bring huge leaps on the embedding performance of our model.

Finally, as we observed in the evaluation stage, the code-clone pairs from different programming languages are not correctly embedded as we expect. This indicates that our model does not totally over come the language boundary. Apart from taking more consideration on pre-training node embeddings, we should also improve our model structure for the cross language task. For example, a straightforward idea would be implementing independent GNN models to deal with ASTs from different languages, while setting appropriate constraints.

Publications and Research Activities

(1) Antoine Tu, Shigeru Chiba. Outlook on Composite Type Labels in User-Defined Type Systems. 日本ソフトウェア科学会第 34 回大会, 2017.9.19-21.

References

[1] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

[2] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *IEEE International Conference on Software Engineering*, 2009.

[3] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The development of a software clone detector. 1995.

[4] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Proceedings. Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 36–43, 2002.

[5] Matthias Rieger. *Effective clone detection without language barriers*. PhD thesis, Verlag nicht ermittelbar, 2005.

[6] Andrew Walenstein and Arun Lakhotia. The software similarity problem in malware analysis. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[7] Lawton Nichols, Mehmet Emre, and Ben Hardekopf. Structural and nominal cross-language clone detection. In Reiner Hähnle and Wil van der Aalst, editors, *Fundamental Approaches to Software Engineering*, pages 247–263, Cham, 2019. Springer International Publishing.

[8] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528. IEEE, 2019.

[9] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, 2002.

[10] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE'07)*, pages 96–105. IEEE, 2007.

[11] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271, 2020.

[12] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26:3111–3119, 2013.

[13] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. 2018.

[14] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[15] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015.

[16] Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. *arXiv preprint arXiv:1904.12787*, 2019.

[17] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710, 2014.

[18] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[19] B Weisfeiler and A Leman. The reduction of a graph to canonical form and the algebgra which appears therein. *NTI, Series*, 2, 1968.

[20] Ralf C Staudemeyer and Eric Rothstein Morris. Understanding lstm–a tutorial into long short-term memory recurrent neural networks. *arXiv preprint arXiv:1909.09586*, 2019.

[21] Cheng Chen. Easily understanding gru. `https://zhuanlan.zhihu.com/p/32085405` Last edited January 2, 2018.

[22] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[23] Google. Embedding projector. `http://projector.tensorflow.org/`.

Acknowledgements

I wish to express my sincere gratitude to my supervisor, Professor Shigeru Chiba, whose guidance, encouragement, patience and insight were indispensable to the completion of my master's research.

I also want to extend my gratitude to my Japanese and Chinese friends, who accompanied me during my two year in Japan, and lent me hands whenever I had difficulties.

Last but not least, I want to thank my mother and father, without whose support I would not have been able to embark on what turned out to be the greatest adventure of my life.