Code Analysis Using Deep Learning Techniques

***Abstract*—**

I. Introduction

The static analysis of source code is performed for many reasons, such as bug detection, security analysis, quality assurance, and more. Many static analysis tools and algorithms exist to assist in detection tasks on source code. However, the manual production of such tools and algorithms can be quite costly in time and money, and are prone to software bugs and human error.

To solve these problems, we propose to use deep learning techniques to perform static analysis on source code.

To our knowledge we are the first to perform deep learning techniques on source code.

II. Background

This section contains background information necessary on which our approach is based.

A. Word Embeddings

In natural language processing, word embeddings are used to represent the semantic meaning of word. A word embedding is an n-dimensional vector that gives a distributed representation of a word with each vector dimension representing some semantic feature of the word. Words with similar semantic meaning will be clustered together in the vector space since similar words will be used in similar context [1].

GloVe (Gloval Vectors for word representation) [2] and Skip-gram [3] are the two most popular word embedding learning models. Both operate on the same fundamental idea that the basic semantic meaning of a word can be represented by utilizing surrounding words as semantic context. GloVe uses a co-occurance matrix of all the words in a corpus. When the matrix has been filled with all word co-occurances, matrix factorization is performed to produce the vector values of each word's embedding. Skip-gram defines a window size that indicates the number of surrounding words to use as context. A given word's vector values are updated by using its context as input to a neural network.

After a corpus has been input to either of these algorithms and all words have been processed, the resulting word embeddings should have grouped similar words together in the vector space. That is, the closer two words are to each other, the more similar their semantic meaning is. We propose to use a modified version of the skip-gram model to peform obtain word embeddings for words in source code.

B. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are a neural network that processes sequences of information in steps. For each step in an RNN, new input is processed with the input that came before it. That is, for step t the new input x_t is passed to the hidden state, h_t, of step t. h_t then uses x_t and the output of the previous step o_{t-1} to calculate the current output using the function $Vx_t + Wo_{t-1}$. The result is then usually transformed using a nonlinear function f, such as $tanh$. So, the final calculation for the hidden layer (and the output of that layer) is $o_t = h_t = f(Vx_t + Wo_{t-1})$. This output is then used in the next step's calculation with the next input, x_{t+1}. The steps continue until all input has been consumed and the final output is usually transformed by a softmax classifier to determine the classification (or final analysis) of the input.

III. Approach to Creating Word Embeddings

In this section we discribe a modified skip-gram model that would better facilitate the embedding of words from source code than the normal skip-gram algorithm. We also discuss some preprocessing steps of the source code before using it as input. For the sake of simplicity and readability we will use Java as the example programming language during discussion for the rest of the paper. However, it should be noted that these ideas can be applied to any programming language with some modification.

A. Variable and Function Names

The source code will need to be preprocessed before being used in the skip-gram model. The main preprocessing step is the identification of the scope of variables and functions. For example, given a particular class in Java, a class variable can have the same name as a local variable. Similarly, two different classes may have the same function name. This is not a problem for natural language as every word in a natural language is unique. While having variables and functions with the same name is legal in Java (within certain rules), it can heavily skew the values of the word embeddings as all variables and functions with the same name will be mapped to the same word embedding. To account for this problem, we propose to extend all variable and function names with the class and function they belong to based on their scope. For example, given the Java class *String*, its function *charAt* will be extended to *String.charAt*, and the local variable *index* in the function *charAt* will be extended to *String.charAt.index*. This will ensure that all variable and function names are mapped to their own word embeddings.

However, this causes a new problem. If variable and function names are extended to be specific to the class they are

defined in, then when analyzing new software there will be no word embeddings for any newly defined variables and functions. To remedy this, new software must first be input through the skip-gram model to produce the new necessary word embeddings before being input to a deep learning model.

Other preprocessing includes the removal of all unnecessary white space, some punctuation, and comments. We aknowledge that there are other possible preprocessing steps that can be taken, but we do not consider these at this time. For example, in Java it is legal to overload functions (that is, have different definitions and parameter lists for the same function name). We consider overloading the same as a word in natural language having multiple definitions. While the function is somewhat disambiguated, we believe it will not disambiguate enough to drastically change the results of a deep learning analysis.

B. Modified Skip-Gram

As described earlier, skip-gram uses a window size to determine the context of a word. This works fairly well for natural language processing because each word has a general predefined meaning and the syntax of a sentence does not allow for complex syntactic structures such as decision or loop statements. So a window size that only scans for context a bit before and after a word is able to determine a relative semantic meaning to other words. However, in source code variables and functions are defined within their class (and not always before they are used in the case of functions) and there are more complex syntactic structures. This seems to imply that a simple window size that scans before and after a word will not be sufficient in determining proper word embeddings for source code.

To modify skip-gram, we will need to separate "key words" (which includes key words, punctuation, and operands) from "constructed words" (which includes class, function, and variable names) defined in libraries or by programers. Key words are usually handled by the compiler and are not explicitly defined in a library, unlike the constructed words discussed later. We further divide key words into two categories, those that have a syntactic structure associated with them and those that do not. Key words that have syntactic structures are *if-else*, *while*, *for*, etc. These key words derive most of their meaning from their syntactic structure. Therefore, instead of using a window size, the skip-gram algorithm will consider everything that is a part of the structure as context when updating the word embedding values of that key word. For example, in the statements *for(i; i < 10; i++){ System.out.println(i); }* everything between the parenthesis after the *for* and before the { will be considered as context for the key word *for*. It is important to note that we do not consider the statement within the loop structure (*System.out.println(i);*) as part of the context for the key word *for*, as the statements within key word syntactic structures do not usually have a direct impact on the meaning of the structure (and the key word associated with that structure). For the key words that do not have a syntactic structure (*break*, *+*, *&&*, etc.), only context before and after the word is available to derive meaning. Therefore, the skip-gram algorithm will be used normally for these key words.

A constructed word's embedding will be modified in two separate ways. The first, and most important, is that word's definition. When a constructed word is defined, the entire definition is used as context to the skip-gram model. For example, for a Java class everything in the entire class is used as context; for a variable, any assignment performed on it will be used as context. The second will come from the constructed word's usage in code. In this case, the normal skip-gram algorithm will be used.

C. Deep Learning

Once preprocessing and word embeddings have been completed, the source code is ready to be analyzed by a neural network. The network we will use is an RNN. This is appropriate since the current state of code at any time is dependent on what statements have been executed before it. Starting in the main function, for each word in the code, the embedding associated with that word will be used as input to the next step in the RNN. In the next section we will discuss how we plan to evaluate the effectiveness of deep learning on source code using the described approach.

IV. Evaluation

We plan to evaluate the effectiveness of using deep learning techniques on source code by performing an analysis on data sets.

V. Conclusion

References

[1] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in neural information processing systems*, 2013, pp. 3111–3119.

[2] J. Pennington, R. Socher, and C. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[3] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.