CSE 493G1 Project Milestone - Code Classification Architectures

Rich Chen
University of Washington
rc2002@cs.washington.edu

Andy Stanciu
University of Washington
andys22@cs.washington.edu

1. Research Foundation

Our deep learning research project aims to classify code solutions to programming problems from LeetCode using various neural network architectures.

The central premise of this project is to explore different representations for code in neural networks, particularly compared against standard transformer-based models for sequences.

1.1. Problem Description and Motivation

The use of deep learning approaches for code opens up entirely new avenues of systems for the automated understanding and generation of programming solutions, particularly at scale. This has applications in areas like code completion, bug detection, and programming education. Traditional rule-based approaches in the past have struggled with the complexity and variability of code, but deep learning models, especially in recent years, have excelled at capturing patterns and semantic meaning in structured data. Thus, it becomes an interesting problem to apply deep learning methods to code in the hopes that programmatic analysis tools can be built as a result, with potentially limitless improvements to programmer productivity.

There is a great deal of literature on deep learning for sequence data such as text, particularly in applications to natural language processing. There are also well-established architectures for sequence data. In particular, the historical progression from RNNs to LSTM networks to transformer architectures has made great strides in a wide variety of sequence processing tasks for text [4]. To some extent, programs and code can be viewed as sequence data - a list of tokens that just happens to also be a list of instructions for a computer to execute.

However, there are also certain structures present in code not present in standard language sequence data. In particular, code tends to follow more repetitive and predictable structures arising from its logical nature compared to natural language, which can often be open-ended in essence [3]. This different structure suggests that an alternative representation of code may be more effective for neural networks to better learn the structure of the data. The question then becomes, what should this representation be?

Code can fundamentally be represented as an Abstract Syntax Tree (AST), a tree data structure used to encode the various structures such as operators and naming schemas present in code. They have been used extensively in compilers, especially as an effective intermediate representation of the code for syntactic analysis. Thus, we consider the problem of encoding code as its innate graph representation and training a neural network to learn code based on this representation as opposed to sequence-based representations such as those used in RNNs, LSTMs, and transformers.

1.2. Research Question

In particular, we focus on the problem of code classification - given various blocks of code intended to solve some problem, classify code snippets with the target problem as a category. We focus on this problem as an extremely simple case to explore if AST-based approaches are viable.

Thus intend on giving an answer to the question: "How effectively can various neural network architectures, especially graph neural networks, classify code solutions to programming problems on LeetCode?"

Closely related are two more open-ended questions, namely, "Why do certain kinds of neural network architectures work better or worse on the task of code classification?" and "How should code be represented in a neural network?" for which possible insights can be drawn by running experiments to answer our main question.

The significance of this particular research topic is that its results will be useful in the further development of deep learning models with applications to software engineering, a topic that will undoubtedly become increasingly important as such approaches become ever more pervasive in software development [6].

1.3. Related Work in the Field

The bulk of the work in interactions between deep learning models and computer programming code has been in the area of code generation, for the most part. There are three main paradigms: description-to-code, code-to-description,

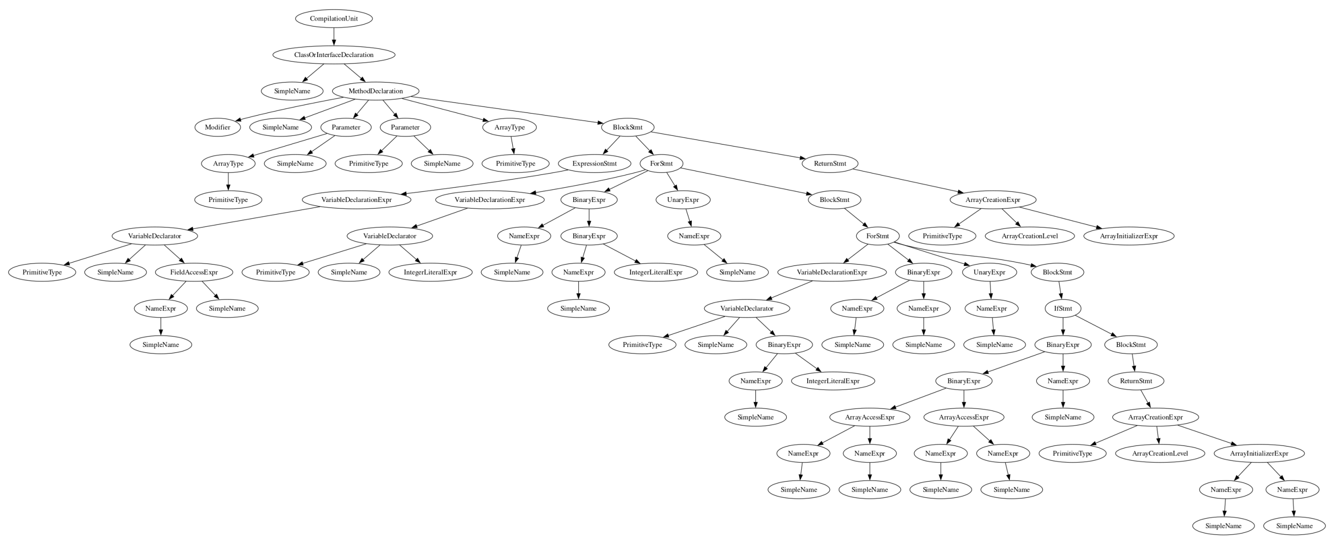

Figure 1. An example abstract syntax tree for a code snippet solution to the two-sum problem involving finding two numbers in an array that add up to a given target.

and code-to-code [5]. Popular applications within these include code generation from natural language, documentation generation, and automated program repair. Common ML models used here include recurrent neural networks, transformers, and convolutional neural networks, among others. [5]

Other than this, there has been work done in the classification of code snippets, primarily to identify the programming language used [2]. Work in this area has primarily relied on natural language processing techniques.

In both cases, datasets are extracted primarily by sources such as programming reference websites like StackOverflow, or publicly available repositories like GitHub.

Finally, there has also been research on using graph neural networks to process code, especially for software analysis tools to predict naming of variables as well as misuse of variable names as a programming aid [1].

From this, we present our novel problem, which is to classify code snippets by the problem they intend to solve, rather than the programming language used. In particular, we would like to train a model to, given a snippet of code (which may be logically correct or incorrect, but always syntactically correct), successfully assign it to a problem that the code is intending to solve, in some sense outputting the purpose of the code.

The use of a graph neural network for code, though not entirely novel, is not an area with extensive research, as most prior applications of graph neural networks have been for raw text data, image data, networked data, or scientific applications. Thus, we hope to conduct this experiment to also better the general understanding of of graph network architectures for code specifically.

2. Technical Approach

We compare a baseline transformer-based approach with our proposed method, which utilizes Graph Neural Networks (GNNs) applied to Abstract Syntax Trees (ASTs) of the code.

2.1. Baseline Method

For the baseline, we employ a transformer-based model, a proven architecture effective for sequence-based tasks [7]. This follows a fairly standard transformer architecture for classification task. First, the source code is tokenized into a sequence of tokens using a pre-defined tokenizer. These tokens are fed into the transformer, which uses self-attention mechanisms to capture both local and global dependencies within the sequence. The final representation, obtained from the transformer's last layer, is passed to a classifier to predict the appropriate label.

2.2. Proposed Method

Our proposed approach uses Graph Neural Networks (GNNs) to classify code based on its Abstract Syntax Tree (AST) representation. The AST will be able to capture the code's hierarchical syntax and structure in the process.

The first step is AST extraction. To that end, we will use a language-specific parser to extract an AST for each of our code examples.

Next, we will vectorize the AST using pre-existing methods for deriving a representation vector from an AST. For this, we will go through the entire dataset and aggregate a co-occurrence matrix, from which we will have a set of vector embeddings for each node. Our graph representations within the network then consist of the graphical relationship between all these nodes, and the individual embeddings for each node (as its node data).

Finally, the graphs will be passed into our graph neural network for classification training. Specifically, we opt for a graph convolutional network (GCN), which is a somewhat modified variant of a convolutional neural network (CNN) with graph-based convolutions rather than the typical grid-based image convolution operations 2. We pass the graph through the GCN's graph convolutional layers, which will aggregate information from neighbors to compute a representation for each node and the entire graph. Each layer is also accompanied by an activation function afterwards; in our case we use ReLU as a fairly standard activation function. At the end of the convolutional layers, a pooling layer then aggregates node embeddings into a graph-level representation via global pooling and passes the result through a classifier layer for prediction.

2.3. Dataset Details

Our data is structured to contain 100 categories, each one corresponding to a unique task or problem the code for that category is intended to solve. We then aggregate 500 code snippets for each category, mapping each code snippet to the problem it is intended to solve. All code snippets will be in the Java programming language for consistency.

2.3.1 Data Sources

The data for this project is sourced by scraping solutions from LeetCode, a popular online platform for coding challenges. Each category's specific problem corresponds to one found on LeetCode, and the aim is to collect a diverse set of 500 code snippets per problem. The scraping process involves extracting publicly available solutions from LeetCode's provided discussion forums, where users can post about their particular solutions to various problems.

2.3.2 Data Pre-Processing and Labeling

The pre-processing step involves standardizing all solutions to meet certain criteria, namely that the solution should be just the code block. To clean up the data, we use several regular expressions to eliminate unwanted or undesirable blocks of text. Furthermore, we ensure that all scraped code snippets compile and are valid Java programs - we use a Java parser to filter out any code snippets that do not. All filtered out code snippets are then deleted from the dataset.

We then apply pre-processing to construct three variations of each code snippet. These are 1) the raw code snippet (hereforth referred to as *raw code*), 2) the code snippet with category classification redacted (hereforth referred to as *redacted code*), and 3) the code snippet with both category classification redacted and all comments removed (hereforth referred to as *redacted-and-stripped code*).

In this following example, we provide an instance of raw code, a Java solution to the common two-sum problem involving finding two numbers in an array that add up to a given target:

```java
class Solution {
  public int[] twoSum(int[] nums, int t) {
    int n = nums.length;
    // Loop through  pairs of numbers
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        // If pair is found, return it
        if (nums[i] + nums[j] == t) {
          return new int[] { i, j };
        }
      }
    }
    // No solution found
    return new int[] {};
  }
}
```

Here is an instance of the same code, processed to remove identifying features of the problem. In particular, observe that the method title has been obfuscated, and thus no longer directly identifies the category:

```java
class Solution {
  public int[] method1(int[] nums, int t) {
    int n = nums.length;
    // Loop through pairs of numbers
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        // If pair is found, return it
        if (nums[i] + nums[j] == t) {
          return new int[] { i, j };
        }
      }
    }
    // No solution found
    return new int[] {};
  }
}
```

Finally, we provide an example of the same code block with obfuscation, but with comments removed as well:

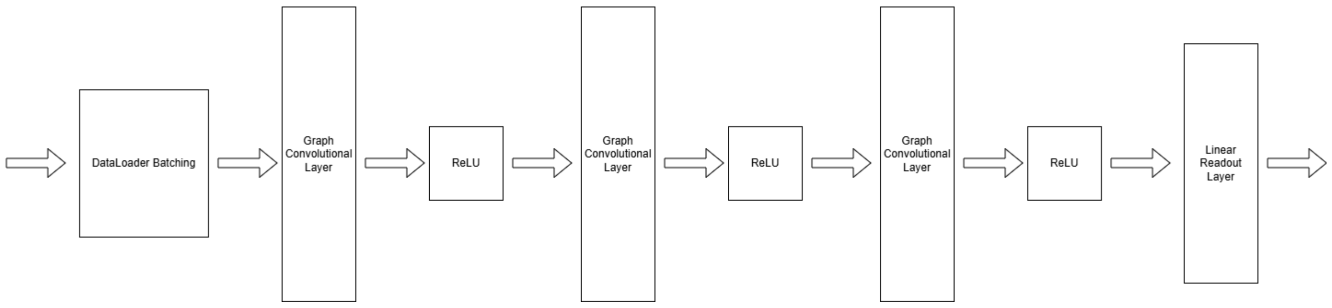
DataLoader Batching
Graph Convolutional Layer
ReLU
Graph Convolutional Layer
ReLU
Graph Convolutional Layer
ReLU
Linear Readout Layer

Figure 2. Diagram of the GNN architecture.

```
class Solution {
  public int[] method1(int[] nums, int t) {
    int n = nums.length;
    for (int i = 0; i < n - 1; i++) {
      for (int j = i + 1; j < n; j++) {
        if (nums[i] + nums[j] == t) {
          return new int[] { i, j };
        }
      }
    }
    return new int[] {};
  }
}
```

These three categories will serve to underscore important properties about the nature of the neural networks we plan on running experiments on. In particular, the intuition here is that a transformer may also be able to learn the purpose of code through contextual clues such as its method signature and commenting, while the underlying AST information captures on the logical structure of the code and thus should perform equally well regardless of the presence of contextual assisting information.

3. Implementation Plan

We scraped problems from LeetCode using GraphQL queries against LeetCode's existing API.

We then modified an external utility for Java parsing (found at https://github.com/javaparser/javaparser) to parse the scraped code snippets into our target format.

We also write our own modifications to the Java parsing tool, as well as our own custom Python scripts to handle co-occurrence matrices and the vectorization of the graphs for the GNN. In particular, we use a co-occurrence algorithm (followed by normalization) to construct feature embeddings for each type of node, then apply standard graph to vector encodings [3] using the graph structure and feature vectors for the graph neural network.

Finally, for the implementation of our models themselves, we use PyTorch's extensive library of various neural network architectures to implement our models ourselves.

Method	Raw Acc	Redact Acc	R+S Acc
Transformer			
GNN			

Table 1. Results. Ours is better.

4. Experimental Design

Both methods (our baseline and proposed methods) are trained and evaluated on the same dataset of programming problem solutions, with a portion of the dataset withheld for testing purposes. The training dataset will exclusively be selected from the set of raw examples without redacting or stripping comments.

Evaluation will then be done in three separate batches, on a test set held over from the raw code snippets, as well as the corresponding redacted and redacted-stripped variations of those same code snippets to test the generalization of the two models to situations where the model no longer has the use of any comments or method signature information to assist it.

Performance will be assessed using standard metrics, especially accuracy. Hyperparameters for both models, such as the number of layers, learning rates, and optimizer configurations, are optimized through grid search. The results from both methods will be compared to determine if leveraging structural information via GNNs provides a significant performance improvement over the baseline transformer approach.

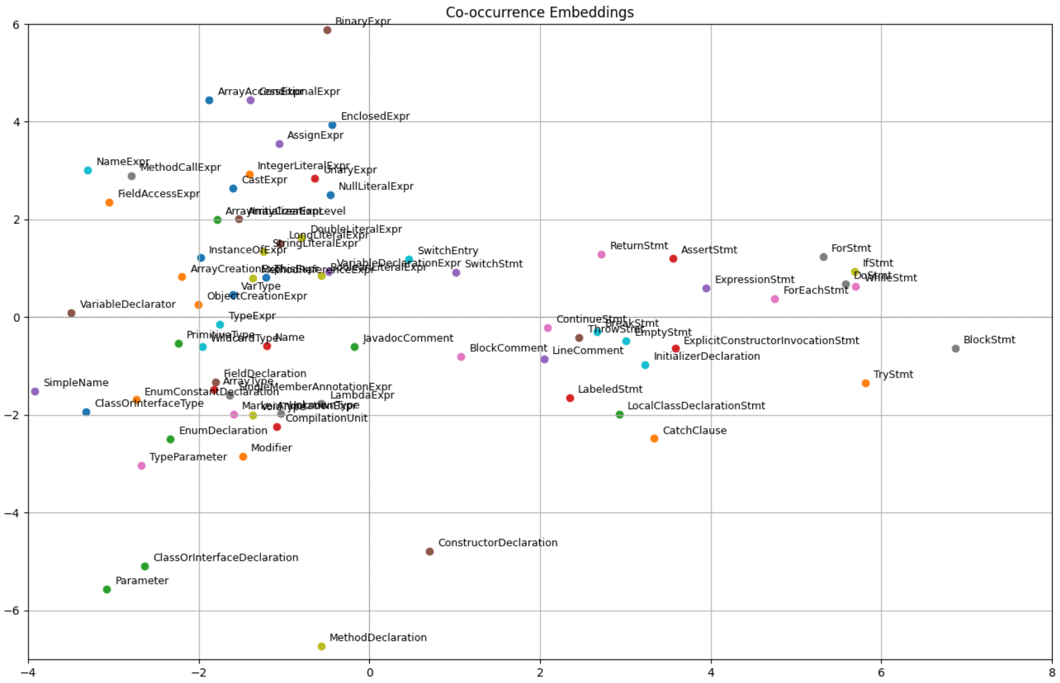

Figure 3. Our generated vector embeddings of each AST node type, visualized using PCA's top two principal component projections along the two axes. Observe how similar nodes are grouped together, such as various *Expr* nodes in a cluster towards the left, and control flow nodes like *ForStmt*, *IfStmt*, and *WhileStmt* in a cluster towards the right. The two leading principal components also correspond to, loosely, how involved a node is in the control flow of a program on the principal component plotted on the x-axis (positive is more involved) and, how involved a node is in the object-oriented features of Java (negative is more involved).

References

[1] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs, 2018. 2

[2] Kamel Alreshedy, Dhanush Dharmaretnam, Daniel M. German, Venkatesh Srinivasan, and T. Aaron Gulliver. Scc: Automatic classification of code snippets, 2018. 2

[3] Casey Casalnuovo, Kenji Sagae, and Prem Devanbu. Studying the difference between natural and programming language corpora, 2018. 1

[4] Yingxuan Chai, Liangning Jin, Shujie Feng, and Zhuo Xin. Evolution and advancements in deep learning models for natural language processing. *Applied and Computational Engineering*, 77:144–149, 07 2024. 1

[5] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. Code generation using machine learning: A systematic review. *IEEE Access*, 10:82434–82455, 2022. 2

[6] Valerio Terragni, Partha Roop, and Kelly Blincoe. The future of software engineering in an ai-driven world, 2024. 1

[7] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. 2