Structurally-Informed Code Representation Improves Natural Language Code Search

Faizan Khan
McGill University
faizan.khan3@mail.mcgill.ca

Andrea Jang
McGill University
eunbee.jang@mail.mcgill.ca

Abstract

A programming language is a form of written communication as natural language is, and a growing number of studies show the successful application of natural language processing (NLP) techniques to tackle software engineering challenges. One key problem to this domain is the general-purpose source code representation that can be used for different downstream tasks. While most work around source code representation is limited to abstract syntax tree (AST) traversal techniques, little is known of the impact of the AST inputs to source code encoder. In this work, we compare the original code input to the AST-based sequence on a source code encoder to tackle the code retrieval task called the CodeSearchNet challenge. We later introduce structurally-informed attention that allows the learning of the code structure internal to model architecture. Our result shows that naively using the AST-based code sequence as input complicates the model training. However, configuring the internal model structure to attend to the AST sequence as input improves our baseline by (put number for MRR) in the CodeSearchNet Challenge.

1 Introduction

Numerous studies have attempted to explore natural language processing (NLP) techniques to programming language in an effort to improve source code representation due to the relatedness of programming and natural language. Both can be expressed in strings of letters and hold the property of recursiveness using a set of defined rules and instructions. Unlike human language, code is less ambiguous, always well-defined, and absent from redundancy. The grammar of a program is determined based on the use of symbols and punctuation, and it is analyzed and compiled into a set of executable instructions in the background to run a program.

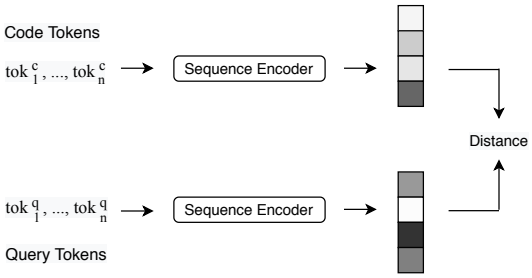

Figure 1: Model Architecture overview

In recent years, there have been several attempts to encode source code using an abstract syntax tree (AST) in different software engineering downstream tasks. AST is a tree representation that represents the structure of source code through a syntactic analysis during program compilation. It exposes deep down structural details, in which a written program does not surface. Current approaches are limited to learning different traversal paths rather than directly encoding the structural components present in the AST input.

Thus, we aim to study the impact of the AST representation by comparing its tokens with original code tokens as input. We use a code retrieval task called CodeSearchNet challenge[1], a task introduced to standardize code search effort. The objective of the challenge is to retrieve the most semantically related code block using the natural language query (comments). The key idea is to design a system that can jointly capture the representation of code and the corresponding natural language query (i.e. Figure 1).

The study consists of two sub-experiments: (1) we first compare the raw code sequence input to the AST-based sequence to observe the model performance. (2) We design and a structurally-informed attention layer, a work introduced by Strubell et al.

[1] https://github.com/github/CodeSearchNet

(2018) to learn the parent-to-child relationship of AST-based input tokens. This is to observe whether code syntax can be directly learned during model training by attending the attention layer to the structural information present in AST.

To best of our knowledge, our work is the first in using the AST sequence in the CodeSearchNet challenge. Our experiment shows that naively using the AST-based sequence does not improve the code search task when compared with a original code sequence. However, the performance improves significantly, even with one informed attention layered in the encoder.

2 Background

Code Search in Software Engineering There have been numerous literature on automatic source code engines in the software engineering community. However, most are limited to simple statistical matching techniques without leveraging machine learning. FaCoY (Kim et al., 2018) builds a task-specific knowledge base using the functional behavior of the input code and natural language query. Their search is computed using TF-IDF and cosine similarity. CodeHow (Lv et al., 2015) focuses on the relationship of natural language query to API functions. Their novelty is the use of API documents as a query to find the relevant code snippet using boolean logic and classical set theory. Portfolio (McMillan et al., 2011), on the other hand, utilizes a visualization approach using a function graph builder, where a chain of function calls are highlighted to perform code search. They rank the graph network using Spreading Activation Network and PageRank algorithm.

Contrary to the mentioned models, BVAE (Chen and Zhou, 2018) introduces neural framework to code retrieval task. They use MLP-based variational autoencoders (VAE) to model source code and natural language query. These VAEs are trained to learn a joint representation of the code and search query to calculate the contextual distance between them. Their work is similar to our model architecture, but the authors use the bag-of-words assumption for both source code and query, which discards both sequential and structural information of source code.

Source Code Representation Several machine learning and NLP-based approaches to code representation are introduced in recent years on different source code representation. Successful literature includes machine learning models learning different code AST traversals. Code2vec (Alon et al., 2018) was the first successful work that utilized code AST on tasks involving variable name, method name, and full Java type prediction. They represent code using multiple AST paths as input to the encoder. The code is treated as a bag of path-contexts. Code2Seq (Alon et al., 2019a) is an alternative approach to the encoding programming language that leverages the sequential information of the AST paths vectors on an encoder-decoder based model. More recently, SLM (Alon et al., 2019b) leverages the sequence of AST tree traversal based on the product of the conditional probabilities of decomposed nodes. The goal of the work is to predict the missing code lines from its body.

CodeBERT is a bimodal pre-trained language representation for both programming and natural language. It supports downstream tasks such as code search and code document generation that incorporates the bimodal data. Unlike previous approaches, CodeBERT uses raw code sequence as input. Their model is a transformer with a pre-training task of masked language modelling (MLM) and replaced token detection (RTD) with the parameters of RoBERTa initialized. Their model is the current state-of-the-art in the CodeSearchNet challenge, with the mean reciprocal rank score of 0.8685 in *Python* language.

Informed Attention Recent work in natural language processing has shown that linguistically informed language models improve the state-of-the-art performance on one of the challenging tasks, namely, semantic role labeling. This work is explored by Strubell et al. (2018), where the authors incorporate a syntactically informed self-attention layer. The novelty of this experiment is an end-to-end neural network model that performs multi-task learning by stacking multi-head self-attention layers. Instead of having high-dimensional input features that are heavily preprocessed, the authors propose layering self-attentions, where each attention head learns different linguistic tasks such as dependency parsing, part-of-speech tagging, predicate detection, and SRL. With strong evidence from previous work, we aim to explore code structure in

AST using informed attention.

2.1 Abstract Syntax Tree

AST is a tree representation of source code that contains meta-information about the node type of code tokens and their relationship to each other. It is constructed as a result of the syntactic analysis phase of a compiler. The nodes are connected based on their dependency relationship and the resulting tree also provides a hierarchical relationship between code tokens. AST has a particular property where all identifiers and constants are leaf nodes, and the operators are intermediate nodes. Figure 3 show an example of an AST of the code in Figure 2.

3 Method

We aim to study the impact of providing syntax knowledge about the source code for the encoder. AST exposes code syntax by grouping the tokens by their functions and dependency information and assigning functional node type to the clustered nodes. While AST-based input introduces additional tokens that are not present in the original source code, we want to observe their impact on model training.

More specifically, we compare the raw code sequence input to the AST-based sequence to observe the performance of the code retrieval task. Following that, we introduce a structurally-informed attention layer that learns the parental relationship of AST nodes to show that code structure can be learned model internally.

3.1 Model Architecture

Model Architecture Our model architecture follows the structure presented in the CodeSearchNet challenge. The model employs two separate encoders, one for natural language query and another one for code. Both of these encoders are trained together to map into a single, joint vector space, as shown in Figure 1. The training objective is to map code and the corresponding language onto vectors that are near to each other in a joint embedding space.

Preprocessing The input sequences for both natural language and code sequences are tokenized and converted into byte-pair encodings. For code tokens, they are split into subtokens by the indicators such as camel case (i.e. `camelCase` into `camel` and `Case`) and special symbols (i.e. `read_input` as `read` and `input`) before being converted into the byte-pair representation.

Raw Sequence Input (Baseline) Our baseline is the transformer model present in the CodeSearchNet challenge that utilizes raw code sequence as input. Our intuition is that the purpose of a code block is harder to learn since the functional type information behind each token and the dependency relationship between tokens are absent in the sequence. An example of a raw sequence for the code in Figure 2 is shown below.

```
def, sum, two, numbers, (, a, b, ), :,
return, a, +, b
```

Table 1: Example of raw sequence input for the code snippet in figure 2

Sequence Encoders The query and code tokens are then passed through their respective sequence encoders to obtain their contextualized embedding vectors. We use the encoder part of BERT by Devlin et al. (2015) to represent source code and corresponding search query.

Our baseline transformer model for both the encoders have an embedding dimension of 128. The number of attention heads employed for the study are 8 and and the number of layers in both the encoders are set to 3.

Training Objective During training, we are given a set of N code instances for every natural language input. With these, we instantiate a code encoder E_c and a query encoder E_q. The training is then done by minimizing the log loss:

$$-\frac{1}{N}\sum_i log(\frac{exp(E_c(c_i)^T E_q(q_i))}{\sum_j exp(E_c(c_j)^T E_q(q_i))}) \quad (1)$$

The above equation maximizes the inner product of the code and query encodings of the pair, while minimizing the inner product between each c_i and the distractor snippets c_j $(i \neq j)$.

3.1.1 AST-based Sequence (DFS Sequence)

To feed the code AST into a transformer model, we linearize the tree by traversing the nodes. While

```
def sum_two_numbers(a, b):
    return a + b
```

Figure 2: Example of Python Method

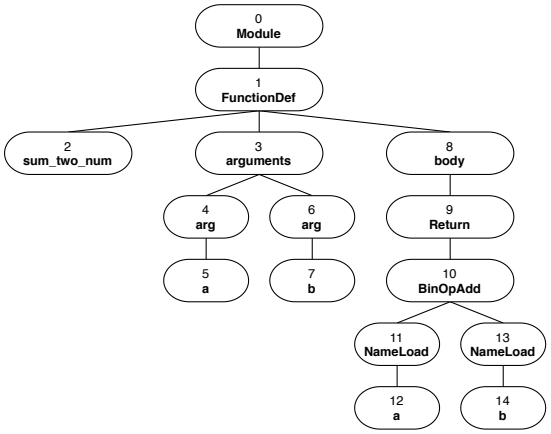

Figure 3: AST for the code snippet in Fig. 2

there are several techniques for traversing tree structures, the default to the AST modules are *depth-first* (DFS) and *breath-first* (BFS) traversals in a pre-ordered manner. For our work, we work with DFS traversal. For the code in Figure 2, a DFS traversal would result in the following sequence:

Module, FunctionDef, sum_two_numbers, arugments, arg, a, arg, b, body, Return, BinOpAdd, NameLoad, a, NameLoad, b

Table 2: DFS sequence for the code snippet in Fig. 2

Since the goal of the study is to observe the impact of structural information which the original code misses, we chose the pre-order DFS sequence for the experiment. DFS sequence preserves the order of the original code sequence. Tokens describing variable names and identifiers remain the same as original code before the tokenization, and the functional keywords are replaced by the nodes that describe them in AST. For example, the function keyword *def* in Figure 2 is replaced with *FunctionDef* in the AST (Figure 3). Moreover, the nodes' position of DFS reflects the grammar of symbols and punctuation in the source code. The tokens *"("* and *")"* are translated as the node *arguments*, and their arguments *a* and *b* are placed as children to the node along with their types (i.e. *arg*). When linearized with DFS, the arguments *a* and *b* along with their variable types follow *arguments* token. We parse the code using a built-in python AST parser.

3.2 Structurally-Informed Code Encoder

The second sub-study consists of structurally-informed attention attending to the parental node for every node in AST. The goal is to observe whether we can control the attention-heads to learn the specific functional components of source code directly. Thus, we modify the code encoder as in Figure 4, where the first P layers have the structurally-informed attention heads and the next layers up to J are the normal multi-Head self-attention (CITE). In our case, we have one informed self-attention and two multi-head self-attention layers. The input to this model is the pre-order DFS sequence.

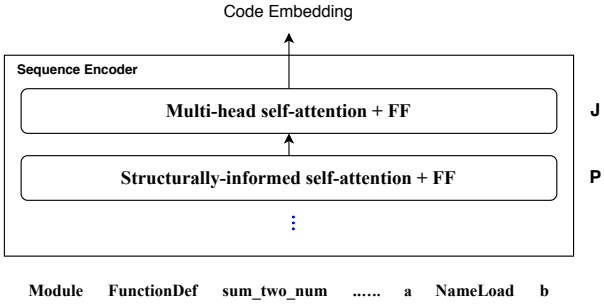

Figure 4: Sequence encoder with structurally-informed attention for source code

Structurally-informed attention head Figure 5 shows one informed attention-head that is forced to predict its parent tokens for every token in the input sequence. Just like all of the other attention heads we project the input-sequence into *Query*, *Key* and *Value* representations. The *Query* and *Key* vectors represent the child and parent nodes respectively. We force this attention head to retain the tree structure by having it project the DFS sequence into a sequence of its parent tokens.

More concretely, for the informed attention head, $Query$ is the projected vector of the DFS sequence and Key is the project vector of DFS-Parent sequence. The job of the attention head is to project the $Query$ and the Key to the same embedding space, learning latent relationships between tokens in the DFS sequence and its corresponding parent tokens in the DFS-parent sequence. These latent relationships are stored in the attention weight matrix A_{parse}.

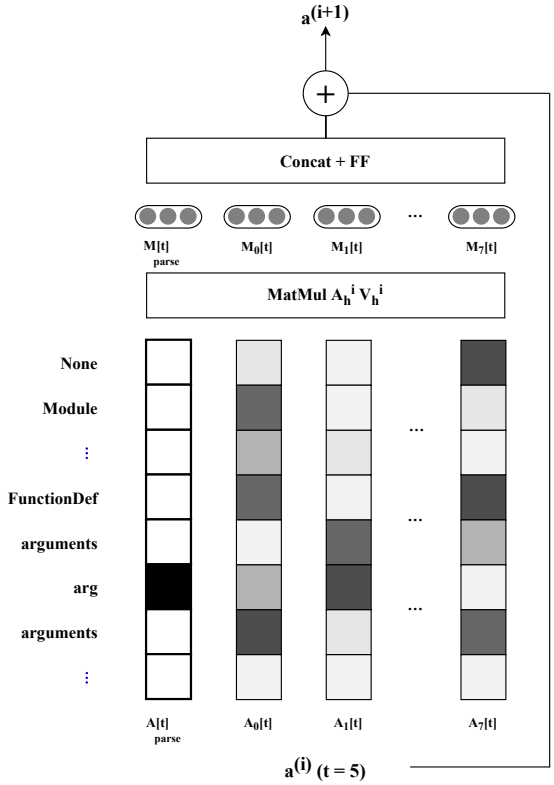

Figure 5: Structurally-informed self-attention for the node `a` at index 5 in the input sequence. Attention weights A_{parse} heavily weight the token's syntactic governor, `arg`, in a weighted average over the token values V_{parse}. The other attention heads act as usual, and the attended representations from all heads are concatenated and projected through a feed-forward layer to produce the structurally informed representation for the first argument `a`

DFS-Parent Sequence To get a sequence of parent tokens, we store the parent node of each token observed during the creation of DFS sequence with pre-order traversal. Table 3 shows the DFS-Parent sequence observed when traversing Figure 3 in a DFS manner.

None, Module, FunctionDef, FunctionDef, arguments, arg, arguments, arg, FunctionDef, body, Return, BinOpAdd, NameLoad, BinOpAdd, NameLoad

Table 3: Parent nodes corresponding to DFS sequence in Table 2

Informed Attention Weights If A_{parse} represents the informed attention weights then $A_{parse}[q, k]$ represents the attention weight from token q to its parent k. Thus $A_{parse}[q]$ represents the distribution of attention weights for token q over all possible ks.

We define a root node as having a special `None` token as its parent. Thus, this attention heads emits a directed graph where the parent for each token q is the one given by the highest weight by $A_{parse}[q]$.

Strictly-Informed Attention Instead of having this attention head predict the parent nodes, we can also inject the gold structure of the tree at test time by setting A_{parse} to represent the correct parent-child node relationship in the DFS sequence.

We report the results for both the normal informed self-attention and the strictly informed self-attention in Table 6.

4 Experiment

4.1 Dataset

In order to introduce standardized metrics to measure the evaluation of the code retrieval task, the CodeSearchNet challenge (Husain et al., 2019) was introduced by the GitHub and Microsoft research team. The objective of the challenge is to find the most semantically related code snippet using the natural language query. The setup of the task is to jointly capture the representation of code and the corresponding natural language query.

Query

```
Add the given messages to lease management.
```

Code

```
def lease(self, msg):
    self._manager.leaser.add(msg)
    self._manager.pause_consumer()
```

Figure 6: Example of source code with its corresponding query, "output to html file"

The CodeSearchNet challenge presents a corpus, a collection of source code along with corresponding search queries. In their data, programmers manually validated the relevance of the code and the search queries. The example data is presented in Figure 6. It contains a total of 2 million natural language and code pairs in 6 different languages: *Go, Java, JavaScript, PHP, Python, and Ruby*. Our experiment focuses on data samples in *Python* only, which contains a total of $457,461$ query-code pairs. Our choice of language is based on the relevance score for human annotation - the relevance annotation for *Python* is most evenly distributed compared

to other languages. *Python* is also the language that achieved the highest mean reciprocal rank reported in the original CodeSerchNet challenge. The distribution of the dataset into train, test and validation split is shown in Table 4. Table 5 shows the percentile token distributions for the length of code and query tokens in *Python* data.

	Original Dataset	Python-Only
# Train	1,880,853	412,178
# Valid	89,154	23,107
# Test	100,529	22,176
# Total	2,070,536	457,461

Table 4: CodeSearchNet Dataset Statistics

Percentiles	Code Length	Query Length
0.50	72	10
0.70	114	15
0.80	155	20
0.90	237	33
0.95	341	48

Table 5: Percentile Token Distributions for Python Dataset

4.2 Evaluation Metrics

All of the models are trained with the objective function shown in section 3.1. While it does not directly correspond to the real target task of code search, it has been widely used as a proxy for training similar models (Cambronero et al., 2019; Yao et al., 2019). For testing purposes, a set of 999 distractor snippets c_j are fixed for each test pair (c_i , q_i) and all models evaluated against it.

Mean Reciprocal Rank (MRR) In addition to the above objective, we calculate the MRR results for all of our models. The MRR is a statistic measure for evaluating any process that produces a list of possible responses to a sample of queries, ordered by probability of correctness. The rank of a query response is the position of the first correct answer. The mean of the reciprocal ranks then is given by the following equation.

$$MRR = \frac{1}{|Q|}\sum_{i}^{|Q|}\frac{1}{rank_i} \tag{2}$$

In practice, the $rank$ is a vector calculated by, first, multiplying a query tensor Q with the code tensor C. The diagonal of the product contains the ground truth scores. Then for each query q_i the rank is calculating using Equation 3, where I represents an inclusion function: it returns 1 if the conditions are met and 0 otherwise.

$$rank_{q_i} = \sum_{j}^{|C|} I(C_j >= C_i) \tag{3}$$

5 Results

Table 6 shows the MRR observed in all 4 approaches after 8 epochs. The training is done only one time each given the limited computational resources available. To our surprise, the DFS sequence did not improve performance when compared to the original raw code sequence. We interpret that the extra tokens introduced in DFS sequence might not necessarily be meaningful to the model. Not to mention, the training time increased more than a double compared to the baseline input.

The model with DFS sequence input and a structurally-informed attention layer performed the best. This model has a dedicated attention head that attends to the structural relationship between tokens. The experiment serves as a proof-of-concept that code structure can be learned even with a simple modification to an attention layer in a transformer model.

The performance of the *Strictly Informed Attention Model* is on par but slightly lower than the *Informed Attention Model*. This is not surprising because our implementation of the informed attention head is very naive, therefore less gains are expected from the gold trees structures.

Model	MRR
Baseline (CodeSearchNet Challenge)	0.6922
Baseline (Our run)	0.6792
DFS Sequence	0.5766
DFS Sequence + Informed	**0.7209**
DFS Sequence + Strict Informed	0.7191

Table 6: MRR Result on Code Retreival Task: *Baseline* uses the raw token sequence

6 Conclusions and Future Directions

In this paper, we tackle the CodeSearchNet challenge by configuring the source code encoder to learn the code structure reflected in code AST. Our experiment show that naively using an AST-based sequence as input to a transformer gives us a very poor performance as opposed to the raw sequence inputs. On the other hand, when the model is configured to learn the structural information of source code reflected on AST with DFS sequence as input, the encoder can better represent the input source code. Our experiment serves as a proof-of-concept that code syntax can be learned directly at the model level rather than learning different tree traversal techniques, the approach taken by previous literature.

In the future, we plan to extend our model to include more informed-layers that can perform multi-task learning of different structural features reflected in the AST. In addition, analyzing the relationship of code structure to its semantics would be an interesting direction to explore. Likewise, we propose to extend the work to other programming languages to observe the generalizability of the structurally-informed attention on source code representations.

References

Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019a. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*.

Uri Alon, Roy Sadaka, Omer Levy, and Eran Yahav. 2019b. Structural language models for any-code generation. *ArXiv*, abs/1910.00577.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29.

Jose Cambronero, Hongyu Li, Seohyun Kim, Koushik Sen, and Satish Chandra. 2019. When deep learning met code search.

Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 826–831.

Jacob Devlin, Hao Cheng, Hao Fang, Saurabh Gupta, Li Deng, Xiaodong He, Geoffrey Zweig, and Margaret Mitchell. 2015. Language models for image captioning: The quirks and what works. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 100–105, Beijing, China. Association for Computational Linguistics.

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436*.

Kisub Kim, Dongsun Kim, Tegawendé F Bissyandé, Eunjong Choi, Li Li, Jacques Klein, and Yves Le Traon. 2018. Facoy: a code-to-code search engine. In *Proceedings of the 40th International Conference on Software Engineering*, pages 946–957.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. 2015. Codehow: Effective code search based on api understanding and extended boolean model (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 260–270. IEEE.

Collin McMillan, Mark Grechanik, Denys Poshyvanyk, Qing Xie, and Chen Fu. 2011. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 111–120.

Emma Strubell, Patrick Verga, Daniel Andor, David Weiss, and Andrew McCallum. 2018. Linguistically-informed self-attention for semantic role labeling. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5027–5038, Brussels, Belgium. Association for Computational Linguistics.

Ziyu Yao, Jayavardhan Reddy Peddamail, and Huan Sun. 2019. Coacor: Code annotation for code retrieval with reinforcement learning. *The World Wide Web Conference on - WWW '19*.