A Novel Neural Source Code Representation based on Abstract Syntax Tree

Jian Zhang[†‡], Xu Wang[†‡*], Hongyu Zhang[§], Hailong Sun[†‡], Kaixuan Wang[†‡] and Xudong Liu[†‡]
[†]SKLSDE Lab, School of Computer Science and Engineering, Beihang University, Beijing, China
[‡]Beijing Advanced Innovation Center for Big Data and Brain Computing, Beijing, China
[§]The University of Newcastle, Australia
{zhangj, wangxu}@act.buaa.edu.cn, hongyu.zhang@newcastle.edu.au, {sunhl, wangkx, liuxd}@act.buaa.edu.cn

***Abstract*—Exploiting machine learning techniques for analyzing programs has attracted much attention. One key problem is how to represent code fragments well for follow-up analysis. Traditional information retrieval based methods often treat programs as natural language texts, which could miss important semantic information of source code. Recently, state-of-the-art studies demonstrate that abstract syntax tree (AST) based neural models can better represent source code. However, the sizes of ASTs are usually large and the existing models are prone to the long-term dependency problem. In this paper, we propose a novel AST-based Neural Network (ASTNN) for source code representation. Unlike existing models that work on entire ASTs, ASTNN splits each large AST into a sequence of small statement trees, and encodes the statement trees to vectors by capturing the lexical and syntactical knowledge of statements. Based on the sequence of statement vectors, a bidirectional RNN model is used to leverage the naturalness of statements and finally produce the vector representation of a code fragment. We have applied our neural network based source code representation method to two common program comprehension tasks: source code classification and code clone detection. Experimental results on the two tasks indicate that our model is superior to state-of-the-art approaches.**

***Keywords*-Abstract Syntax Tree; source code representation; neural network; code classification; code clone detection**

I. Introduction

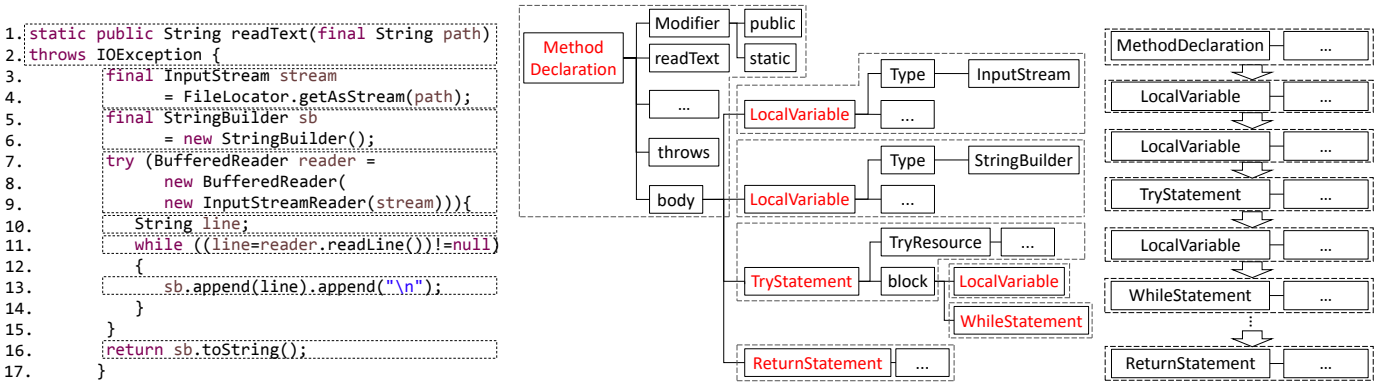
Many software engineering methods, such as source code classification [1], [2], code clone detection [3], [4], [5], [6], defect prediction [7], [8] and code summarization [9], [10] have been proposed to improve software development and maintenance. One main challenge that is common across all these methods is how to represent source code, in order to effectively capture syntactical and semantic information embedded in the source code.

Traditional approaches such as Information Retrieval (IR) usually treat code fragments as natural language texts and model them based on tokens. For example, programs are represented by token sequences or bag of tokens for code clone detection [3], [4], bug localization[11], and code authorship classification [1]. In addition, a number of researchers use Latent Semantic Indexing (LSI) [12] and Latent Dirichlet Allocation (LDA) [13] to analyze source code [14], [15], [16]. However, according to [17], the common problem of these approaches is that they assume the underlying corpus (i.e., the source code) is composed of natural language texts. Even though code fragments have something in common with plain texts, they should not be simply dealt with text-based or token-based methods due to their richer and more explicit structural information [2], [18].

Recent work [2], [5], [6] provides the strong evidence that syntactic knowledge contributes more in modeling source code and can obtain better representation than traditional token-based methods. These approaches combine Abstract Syntax Tree (AST) and Recursive Neural Network (RvNN) [5], Tree-based CNN [2] or Tree-LSTM [6] to capture both the lexical (i.e., the leaf nodes of ASTs such as identifiers) and syntactical (i.e., the non-leaf nodes of ASTs like the grammar construct *WhileStatement*) information. Such AST-based neural models are effective, yet they have two major limitations. First, similar to long texts in NLP, these tree-based neural models are also vulnerable to the gradient vanishing problem that the gradient becomes vanishingly small during training, especially when the tree is very large and deep [19], [20], [21]. For example, as our experiments show (Section V), the maximal node number/depth of ASTs of common code fragments in C and Java are 7,027/76 and 15,217/192, respectively. As a result, traversing and encoding entire ASTs in a bottom-up way [5], [6] or using the sliding window technique [2] may lose long-term context information [19], [22]; Second, these approaches either transform ASTs to or directly view ASTs as full binary trees for simplification and efficiency, which destroys the original syntactic structure of source code and even make ASTs much deeper. The transformed and deeper ASTs further weaken the capability of neural models to capture more real and complex semantics [23].

In order to overcome the limitations of the above AST-based neural networks, one solution is to introduce explicit (long-term) control flow and data dependencies graphs and employ a Graph Embedding technique [24] to represent source code. For instance, one recent study considers the long-range dependencies induced by the same variable or function in distant locations [25]. Another study directly constructs control flow graphs (CFGs) of code fragments [26]. However, as depicted in the above work, precise and inter-procedural program dependency graphs (PDGs) (i.e. control flow and data flow dependencies) [27] usually rely on compiled intermediate representations or bytecodes [28], [29], and are not applicable

*Corresponding author: Xu Wang, wangxu@act.buaa.edu.cn.



(a) Code fragment and statements (b) AST and statement trees (c) Statement naturalness

Fig. 1. An example of AST Statement nodes (marked in red)

to uncompilable and incomplete code fragments. Such a limitation hinders the applications of the code representations in many areas that involve arbitrary code fragments.

In this paper, we propose a novel approach for representing code fragments that do not have to be compilable, called AST-based Neural Network (ASTNN), which splits the large AST of one code fragment into a set of small trees at the statement level and performs tree-based neural embeddings on all statement trees. It produces statement vectors which can represent the lexical and statement-level syntactical knowledge. Here *statements* refer to the Statement AST nodes defined in program language specification [30]. We also treat MethodDeclaration as a special statement node. As an example, Figure 1 shows a code fragment from an open source project[1]. The code snippet between line 7 and line 15 contains a whole Try statement and the code snippet between line 5 and line 6 includes only the $LocalVariable$ statement initializing variable sb. For each statement like the Try statement that includes the header and other statements in the body, we split the header of the statement and all included statements. In this way, the large AST is decomposed to a short sequence of small statement trees. We use Recurrent Neural Network (RNN) [31] to encode statements and the sequential dependency between the statements into a vector. Such a vector captures the naturalness of source code [32], [33] and can serve as a neural source code representation.

More specifically, first, we build an AST from the code fragment and split the whole AST to small statement trees (one tree consisting of AST nodes of one statement and rooted at the Statement node). For example, in Figure 1, the statement trees are denoted by dashed lines and the corresponding statements (or statement headers) in the original code fragment are also marked by dashed lines. Second, we design a recursive encoder on multi-way statement trees to capture the statement-level lexical and syntactical information and then represent them in statement vectors. Third, based on the sequence of statement vectors, we use bidirectional Gated Recurrent Unit (GRU) [34], [35], one type of recurrent neural network, to leverage the sequential naturalness of statements and finally obtain the vector representation of an entire code fragment.

In summary, our proposed neural source code representation aims to learn more syntactical and semantic information about source code than the state-of-the-art AST-based neural models. It is general-purpose and can be used in many program comprehension related tasks such as source code classification and code clone detection. We have conducted experiments on the two tasks on public benchmarks and compared with state-of-the-art approaches. The experimental results show that our model is more effective. For example, for source code classification, our approach improves the accuracy from 94% to 98.2%. For clone detection, our approach improves the results of F1 values from 82% to 93.8% and 59.4% to 95.5% on two benchmark datasets, respectively.

Our main contributions are as follows:

- We propose a novel neural source code representation, which can capture the lexical, statement-level syntactical knowledge, and the naturalness of statements;
- We have applied our representation to two common program comprehension tasks (code classification and clone detection). The experimental results show that our approach can improve the state-of-the-art methods.

The remainder of this paper is structured as follows. Section II introduces the background. Section III presents our approach. Section IV describes the applications of our neural source code representation. Section V provides our experimental results. Related work and discussion about threats to validity are presented in Section VI and Section VII, respectively. Finally, Section VIII concludes our work.

II. Background

A. Abstract Syntax Tree

Abstract Syntax Tree (AST) is a kind of tree aimed at representing the abstract syntactic structure of the source code [36]. It has been widely used by programming language and software engineering tools. As illustrated in Figure 1(b), nodes

[1]https://github.com/apache/ctakes/blob/9c552c5c4f92af00d9d008b8c7f9e9d326a2450a/ctakes-core/src/main/java/org/apache/ctakes/core/resource/FileReadWriteUtil.java#L32

of an AST are corresponding to constructs or symbols of the source code. On the one hand, compared with plain source code, ASTs are abstract and do not include all details such as the punctuation and delimiters. On the other hand, ASTs can be used to describe the lexical information and the syntactic structure of source code, such as the method name *readText* and the control flow structure *WhileStatement* in Figure 1(b).

Some studies directly use ASTs in token-based methods for source code search [37], program repair [38] and source code differencing [39]. Due to the limitation of token-based approaches [17], these methods can catch little syntactical information of source code.

B. Tree-based Neural Networks

Recently Tree-based Neural Networks (TNNs) have been proposed to accept ASTs as the input. Given a tree, TNNs learn its vector representation by recursively computing node embeddings in a bottom-up way. The most representative tree-based models for ASTs are Recursive Neural Network (RvNN), Tree-based CNN (TBCNN) and Tree-based Long Short-Term Memory (Tree-LSTM).

1) Recursive Neural Network: RvNN was first proposed for the recursive structure in natural language and image parsing [40]. Specifically, given a tree structure, suppose that one parent node y_1 has two children nodes (c_1, c_2), where c_1 and c_2 are word embeddings or intermediate vector representations of nodes. The vector of node y_1 is computed by:

$$p = f(W^{(1)}[c_1; c_2] + b^{(1)})$$

where $W^{(1)}$ is a matrix of parameters, f is an element-wise activation function, and $b^{(1)}$ is a bias term. To assess the quality of this vector representation, a decoding layer is used to reconstruct the children:

$$[c_1'; c_2'] = W^{(2)}p + b^{(2)}$$

Then the training loss is measured by $E(\theta) = ||c_1 - c_1'||_2^2 + ||c_2 - c_2'||_2^2$. In this way, RvNN can recursively compute and optimize parameters across the tree, and the final vector of the root node will represent the given tree. Based on RvNN, a recursive autoencoder (RAE) is incorporated for automatically encoding ASTs to detect code clones [5], where ASTs are transformed to full binary trees due to the fixed-size inputs for simplification.

2) Tree-based Convolutional Neural Network: TBCNN performs convolution computation over tree structures for supervised learning such as source code classification [2]. Its core module is an AST-based convolutional layer, which applies a set of fixed-depth feature detectors by sliding over entire ASTs. This procedure can be formulated by:

$$y = tanh(\sum_{i=1}^{n} W_{conv,i} \cdot x_i + b_{conv})$$

where $x_1, \cdots, x_n$ are the vectors of nodes within each sliding window, $W_{conv,i}$ are the parameter matrices and b_{conv} is the bias. TBCNN adopts a bottom-up encoding layer to integrate some global information for improving its localness. Although nodes in the original AST may have more than two children, TBCNN treats ASTs as continuous full binary trees because of the fixed size of convolution.

3) Tree-based Long Short-Term Memory: Tree-LSTM is a generalization of LSTMs to model tree-structured topologies. Different from standard LSTM, Child-Sum Tree-LSTM [41] recursively combines current input with its children states for state updating across the tree structure. CDLH [6] uses Tree-LSTM to learn representations of code fragments for clone detection where code fragments are parsed to ASTs. To deal with the variable number of children nodes, ASTs are transformed to full binary trees. After a bottom-up way of computation, the root node vectors of ASTs are used to represent the code fragments.

C. The limitations of the existing work

These three tree-based methods have two major limitations. First, during gradient-based training of tree topologies, the gradients are calculated via backpropagation over structures [41], [23]. However, the structures of ASTs are usually large and deep due to the complexity of programs, especially the nested structures. Thus the bottom-up computations from the leaf nodes to the root nodes may experience the gradient vanishing problem and are difficult to capture long-range dependencies [19], [22], which will miss some of the semantics carried by distant nodes from the root nodes such as identifiers in the leaf nodes. Second, the existing tree-based methods view ASTs as binary trees by moving three or more children nodes of a parent node to new subtrees for simplification, which changes the original semantics of source code and makes the long-term dependency problem more serious. For example, CDLH [6] can only have the F1 value of 57% in one public benchmark for clone detection, and the studies in NLP [23], [41], [21] show that the tree size and depth do matter and have significant impact on the performance.

III. Our Approach

We introduce our AST-based Neural Network (ASTNN) in this section. The overall architecture of ASTNN is shown in Figure 2. First, we parse a source code fragment into an AST, and design a preorder traversal algorithm to split each AST to a sequence of statement trees (ST-trees, which are trees consisting of statement nodes as roots and corresponding AST nodes of the statements), as illustrated in Figure 1. All ST-trees are encoded by the Statement Encoder to vectors, denoted as $e_1, \cdots, e_t$. We then use Bidirectional Gated Recurrent Unit [35] (Bi-GRU), to model the naturalness of the statements. The hidden states of Bi-GRU are sampled into a single vector by pooling, which is the representation of the code fragment.

A. Splitting ASTs and Constructing ST-tree Sequences

At first, source code fragments can be transformed to large ASTs by existing syntax analysis tools. For each AST, we split it by the granularity of statement and extract the sequence of statement trees with a preorder traversal.

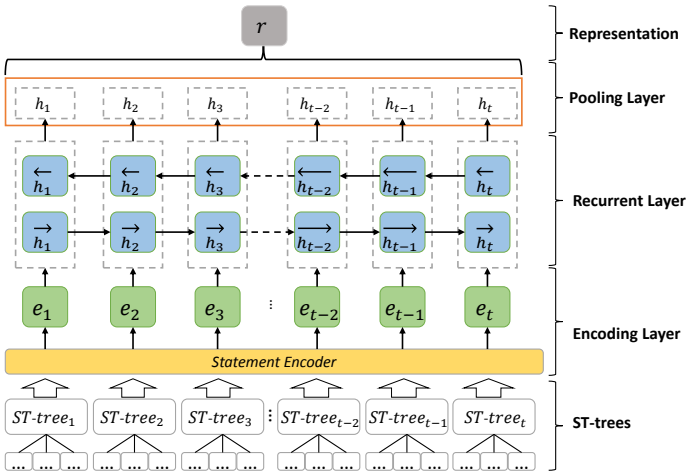


Fig. 2. The architecture of AST-based Neural Network

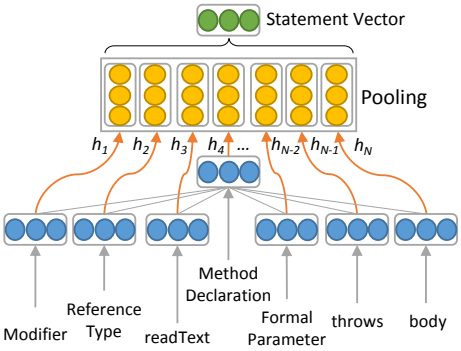


Fig. 3. The statement encoder, where blue, orange and green circles represent the initial embeddings, hidden states and statement vector, respectively.

Formally, given an AST T and a set of Statement AST nodes S, each statement node $s \in S$ in T corresponds one statement of source code. We treat *MethodDeclaration* as a special Statement node, thus $S = S \cup \{MethodDeclaration\}$. For nested statements, as shown in Figure 1, we define the set of separate nodes $P = \{block, body\}$ where $block$ is for splitting the header and body of nested statements such as Try and $While$ statements, and $body$ for the method declaration. All of the descendants of statement node $s \in S$ is denoted by $D(s)$. For any $d \in D(s)$, if there exists one path from s to d through one node $p \in P$, it means that the node d is included by one statement in the body of statement s. We call node d one substatement node of s. Then a statement tree (ST-tree) rooted by the statement node $s \in S$ is the tree consisting of node s and all of its descendants excluding its substatement nodes in T. For example, the first ST-tree rooted by $MethodDeclaration$ is surrounded by dashed lines in Figure 1(b), which includes the header part such as "static", "public" and "readText" and excludes the nodes of the two $LocalVariable$, one Try and one $Return$ statement in the body. Since nodes of one ST-tree may have three or more children nodes, we also call it multi-way ST-tree for distinguishing it from a binary tree. In this way, one large AST can be decomposed to a sequence of non-overlapping and multi-way ST-trees.

The splitting of ASTs and the construction of ST-tree sequences are straightforward by a traverser and a constructor. The traverser visits each node through the ASTs in a depth-first walk in preorder and the constructor recursively creates a ST-tree to sequentially add to the ST-tree sequences. Such a practice guarantees that one new ST-tree are appended by the order in the source code. In this way, we get the sequence of ST-trees as the raw input of ASTNN.

Note that the selection of splitting granularity of ASTs is not trivial. We choose the statement trees in this work since statements are essential units for carrying source code semantics. We also experimented with other granularities such as the node level of ASTs, the code blocks within brace pairs, and the full ASTs. We will discuss these experiments in Section V. If the size of selected granularity is too large (e.g., the full AST), similar to the related work [5], [6], we may also experience the gradient vanishing problem mentioned in Section II. But if it is too small (e.g., the node level of AST), the model will become a token-based RNN that may capture less syntactical knowledge of statements than ours. Our experimental results show that proposed statement-level granularity is better since it has a good trade-off between the size of ST-tree and the richness of syntactical information.

B. Encoding Statements on Multi-way ST-trees

1) Statement Vectors: Given the ST-trees, we design a RvNN based statement encoder, which is used for learning vector representations of statements.

Since there are a variety of special syntactic symbols in ASTs, we obtain all the symbols by preorder traversal of ASTs as the corpus for training. The word2vec [42] is used to learn unsupervised vectors of the symbols, and the trained embeddings of symbols are served as initial parameters in the statement encoder. Because all the leaf nodes of ASTs representing the lexical information such as identifiers are also incorporated in the leaf nodes of ST-trees, our symbol embeddings can capture the lexical knowledge well.

Taking the first ST-tree rooted by the node of *MethodDeclaration* in Figure 1 as an example, the encoder traverses the ST-tree and recursively takes the symbol of current node as new input to compute, together with the hidden states of its children nodes. This is illustrated in Figure 3. We only show the first two levels here. In the ST-tree, the two children nodes *readText* (i.e., the method name) and *FormalParameter* (i.e., the grammar structure defining parameters of the method) as well as other siblings enrich the meaning of *MethodDeclaration*. If we transform the ST-tree to one binary tree as described in [5], [6], for example, moving the node of *readText* to one child node or descendant of the *FormalParameter* node, the original semantics may be destroyed. Instead, we take original multi-way ST-trees as input.

Specifically, given a ST-tree t, let n denote a non-leaf node and C denote the number of its children nodes. At

the beginning, with the pre-trained embedding parameters $W_e \in \mathbb{R}^{|V|\times d}$ where V is the vocabulary size and d is the embedding dimension of symbols, the lexical vector of node n can be obtained by:

$$v_n = {W_e}^\top x_n \tag{1}$$

where x_n is the one-hot representation of symbol n and v_n is the embedding. Next, the vector representation of node n is computed by the following equation:

$$h = \sigma({W_n}^\top v_n + \sum_{i\in[1,C]} h_i + b_n) \tag{2}$$

where $W_n \in \mathbb{R}^{d\times k}$ is the weight matrix with encoding dimension k, b_n is a bias term, h_i is the hidden state for each children i, h is the updated hidden state, and σ is the activation function such as *tanh* or the identity function. We use the identity function in this paper. Similarly, we can recursively compute and optimize the vectors of all nodes in the ST-tree t. In addition, in order to determine the most important features of the node vectors, all nodes are pushed into a stack and then sampled by the max pooling. That is, we get the final representation of the ST-tree and corresponding statement by Equation 3, where N is the number of nodes in the ST-tree.

$$e_t = [max(h_{i1}), \cdots, max(h_{ik})], i = 1, \cdots, N \tag{3}$$

These statement vectors can capture both lexical and statement-level syntactical information of statements.

2) Batch Processing: For improving the training efficiency on large datasets, it is necessary to design the batch processing algorithm to encode multiple samples (i.e., code fragments) simultaneously. However, generally batch processing on multi-way ST-trees makes it difficult since the number of children nodes varies for the parent nodes in the same position of one batch [2], [6]. For example, given two parent nodes ns_1 with 3 children nodes and ns_2 with 2 children nodes in Figure 4, directly calculating Equation 2 for the two parents in one batch is impossible due to different C values. To tackle this problem, we design an algorithm that dynamically processes batch samples in Algorithm 1.

Intuitively, although parent nodes have different number of children nodes, the algorithm can dynamically detect and put all possible children nodes with the same positions to groups, and then speed up the calculations of Equations 2 of each group in a batch way by leveraging matrix operations. In algorithm 1, we batch L samples of ST-trees and then breadth-first traverse them starting from the root nodes (line 4). For the current nodes ns in the same position of the batch, we firstly calculate Equation 1 in batch (line 10), then detect and group all their children nodes by the node positions (line 12-16). As shown in Figure 4, we separate the children nodes to three groups by their positions and record the groups in the array lists C and CI. Based on these groups, we recursively perform batch processing on all children nodes (line 17-21). After getting the results of all children nodes, we compute Equation 2 in batch (line 22), and push all node vectors of batched ST-trees to the stack S (line 24). Finally we can obtain the vectors of ST-tree samples and corresponding statements by pooling described in Equation 3 (line 5).

Algorithm 1 Dynamic batching algorithm of ST-trees

Input: The array of root nodes in batched ST-trees, B;
Output: The vectors of batched ST-trees, BV;
1: $L \leftarrow len(B)$;
2: $BI \leftarrow [1, \cdots, L]$; // ST-tree indexes in the batch
3: $S \in \mathbb{R}^{N\times L\times k} \leftarrow \phi$; // record all node vectors
4: Call DynamicBatch(B,BI);
5: Perform pooling on S by Eq. 3 to get $BV \in \mathbb{R}^{L\times k}$;
6: **return** BV;
7: **function** DYNAMICBATCH(ns, ids) ▷ The batched current nodes ns and their indexes ids
8: $l \leftarrow len(ns)$;
9: $BC \in \mathbb{R}^{l\times d} \leftarrow \mathbf{0}$; // initialize the matrix
10: Calculate Eq. 1 in batch for ns and fill into BC according to ids;
11: Initialize two array list $C, CI \leftarrow \phi$ to record children nodes and their batch indexes;
12: **for** each $node \in ns$ **do**
13: **for** each children node $child$ of $node$ **do**
14: group $child$ by its position, and record $child$ to C and its batch index to CI ;
15: **end for**
16: **end for**
17: **for** $i = 0 \rightarrow len(C) - 1$ **do**
18: $\tilde{h} \in \mathbb{R}^{l\times k} \leftarrow \mathbf{0}$;
19: $\eta \leftarrow$ DynamicBatch($C[i]$, $CI[i]$);
20: $\tilde{h} \leftarrow \tilde{h} + \eta$;
21: **end for**
22: Calculate h by Eq. 2 in batch;
23: $BZ \in \mathbb{R}^{L\times k} \leftarrow \mathbf{0}$; // for calculating BV
24: Fill h into BZ according to ids and add BZ to S;
25: **return** h;
26: **end function**

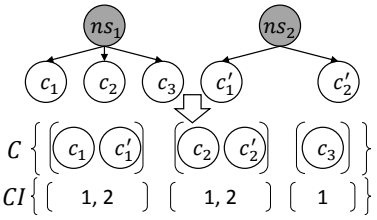


Fig. 4. An example of dynamically batching children nodes

C. Representing the Sequence of Statements

Based on the sequences of ST-tree vectors, we exploit GRU [34] to track the naturalness of statements. We also considered the choice of using LSTM and the comparison between LSTM and GRU will be discussed in our experiment.

Given one code fragment, suppose there are T ST-tree extracted from its AST and let $Q \in \mathbb{R}^{T\times k} = [e_1, \cdots, e_t, \cdots, e_T], t \in [1, T]$ denote the vectors of encoded

ST-trees in the sequence. At time t, the transition equations are as follows:

$$\begin{aligned} r_t &= \sigma(W_r e_t + U_r h_{t-1} + b_r) \\ z_t &= \sigma(W_z e_t + U_z h_{t-1} + b_z) \\ \tilde{h_t} &= tanh(W_h e_t + r_t \odot (U_h h_{t-1}) + b_h) \\ h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t} \end{aligned} \tag{4}$$

where r_t is the reset gate to control the influence of previous state, z_t is the update gate to combine past and new information, $\tilde{h_t}$ is the candidate state and used to make a linear interpolation together with previous state h_{t-1} to determine the current state h_t. $W_r, W_z, W_h, U_r, U_z, U_h \in \mathbb{R}^{k \times m}$ are weight matrices and b_r, b_z, b_h are bias terms. After iteratively computing hidden states of all time steps, the sequential naturalness of these statements can be obtained.

In order to further enhance the capability of the recurrent layer for capturing the dependency information, we adopt a bidirectional GRU [34], where the hidden states of both directions are concatenated to form the new states as follows:

$$\begin{aligned} \overrightarrow{h_t} &= \overrightarrow{GRU}(e_t), t \in [1, T] \\ \overleftarrow{h_t} &= \overleftarrow{GRU}(e_t), t \in [T, 1] \\ h_t &= [\overrightarrow{h_t}, \overleftarrow{h_t}], t \in [1, T] \end{aligned} \tag{5}$$

Similar to the statement encoder, the most important features of these states are then sampled by the max pooling or average pooling. Considering the importance of different statements are intuitively not equal, for example, API calls in the *MethodInvocation* statements may contain more functional information [43], thus we use max pooling for capturing the most important semantics by default. The model finally produces a vector $r \in \mathbb{R}^{2m}$, which is treated as the vector representation of the code fragment.

IV. Applications of the Proposed Model

The proposed ASTNN model is general-purpose. It can be trained for task-specific vector representations of source code fragments to characterize different source code semantics for many program comprehension tasks. In this work, we take two common tasks including source code classification and code clone detection as examples to show the applications of the proposed model.

Source code classification. This task aims to classify code fragments by their functionalities, which is useful for program understanding and maintenance [2], [44], [45]. Given the code fragment vector r and the number of categories M, we obtain the logits by $\hat{x} = W_o r + b_o$, where $W_o \in \mathbb{R}^{2m \times M}$ is the weight matrix and b_o is the bias term. We define the loss function as the widely used cross-entropy loss:

$$J(\Theta, \hat{x}, y) = \sum \left(-log \frac{exp(\hat{x}_y)}{\sum_j exp(\hat{x}_j)} \right) \tag{6}$$

where Θ denotes parameters of all the weight matrices in our model and y is the true label.

Code clone detection. Detecting code clones is widely studied in software engineering research [3], [4], [5], [6], [26], which is to detect whether two code fragments implement the same functionality. Suppose there are code fragment vectors r_1 and r_2, and their distance is measured by $r = |r_1 - r_2|$ for semantic relatedness [41]. Then we can treat the output $\hat{y} = sigmoid(\hat{x}) \in [0, 1]$ as their similarity where $\hat{x} = W_o r + b_o$. The loss function is defined as binary cross-entropy:

$$J(\Theta, \hat{y}, y) = \sum (-(y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y}))) \tag{7}$$

To train ASTNN models for the two tasks, the goal is to minimize the loss. We use AdaMax [46] in this paper because it is computationally efficient.

After all the parameters are optimized, the trained models are stored. For new code fragments, they should be preprocessed into sequences of ST-trees and then fed into the reloaded models for prediction. The output are probabilities p for different labels. For code classification, since there are multiple categories, the inferred value can be obtained by:

$$prediction = \arg\max_i (p_i), i = 1, \cdots, M \tag{8}$$

While for clone detection, p is a single value in the range [0,1], thus we get the prediction by:

$$prediction = \begin{cases} True, & p > \delta \\ False, & p \leq \delta \end{cases} \tag{9}$$

where δ is the threshold.

V. Experiments

In this section, we evaluate the proposed source code representation on two tasks of code classification (Task 1) and clone detection (Task 2), and compare it with several state-of-the-art approaches.

A. Dataset Description

There are two public dataset benchmarks for code classification and clone detection. One dataset consists of simple C programs collected from the Online Judge (OJ) system and made public by Mou et al. [2][2]. The programs in OJ benchmark are for 104 different programming problems. Programs have the same functionality if they aim to solve the same problem. The other dataset BigCloneBench (BCB) was provided by Svajlenko et al. [47] for evaluating code clone detection tools. BCB consists of known true and false positive clones from a big data inter-project Java repository. As benchmarks, the two datasets have been used by many researchers concerning on code similarity [48], [49] and clone detection [5], [6]. The basic statistics corresponding to our two tasks are summarized in Table I.

B. Experiment Settings

We used the pycparser[3] and javalang tools[4] to obtain ASTs for C and Java code, respectively. For both tasks, we trained embeddings of symbols using word2vec [42] with Skip-gram

[2]https://sites.google.com/site/treebasedcnn/
[3]https://pypi.python.org/pypi/pycparser
[4]https://github.com/c2nes/javalang

TABLE I
THE STATISTICS OF DATASETS USED FOR OUR TWO TASKS

Code Classification		Clone Detection		
Dataset	OJ	**Dataset**	OJ	BCB
#Programs	52,000	**#Code fragments**	7,500	59,688
#Classes	104	**%True clone pairs**	6.6%	95.7%
Max tokens	8,737	**Max tokens**	2,271	16,253
Avg. tokens	245	**Avg. tokens**	244	227
Max AST depth	76	**Max AST depth**	60	192
Avg. AST depth	13.3	**Avg. AST depth**	13.2	9.9
Max AST nodes	7,027	**Max AST nodes**	1,624	15,217
Avg. AST nodes	190	**Avg. AST nodes**	192	206

algorithm and set the embedding size to be 128. The hidden dimension of ST-tree encoder and bidirectional GRU is 100. We set the mini-batch size to 64 and a maximum of 15 and 5 epochs for the two tasks, respectively. The threshold is set to 0.5 for clone detection. For each dataset, we randomly divide it into three parts, of which the proportions are 60%, 20%, 20% for training, validation and testing. We use the optimizer AdaMax [46] with learning rate 0.002 for training. All the experiments are conducted on a server with 16 cores of 2.4GHz CPU and a Titan Xp GPU.

C. Evaluation on Two Tasks

1) Source Code Classification: We conduct extensive experiments on the OJ dataset. Apart from the state-of-the-art model **TBCNN** [2], we also take into account of traditional and other neural network based approaches including SVMs with statistical features, TextCNN [50], LSTM [51], LSCNN [52] and PDG-based Graph embedding approaches [25], [26] as follows:

- **SVMs**. We use the linear SVMs with traditional IR methods. TF-IDF, N-gram and LDA are used to extract textual features. The corpus are tokens extracted from the source code files. For N-gram, we set the number of grams to 2 and the number of max features to 20 thousand. The number of topics for LDA is 300.
- **TextCNN** and **LSTM**. These two models are widely used for sentence classification in NLP. We adapt them for this task with token sequences by treating code fragments as plain texts. For TextCNN, the kernel size is set to 3 and the number of filters is 100. For LSTM, The dimension of hidden states is set to 100.
- **LSCNN**. Originally proposed for bug location [52], LSCNN extracts program features with CNN for statement embedding and uses LSTM for statement sequences.
- **PDG based Graph Embedding**. Most recently some studies [25], [26] construct program graphs by considering control flow and data flow dependencies, and adopt graph embedding techniques such as HOPE [24] and Gated Graph Neural Network (GGNN) [53] for code representation. Although original code fragments in the OJ dataset is incomplete and uncompilable, they can be manually complemented by adding standard C header files and third-party libraries and we use an open-source tool Frama-C[5] to get their PDGs. Based on the PDGs, we represents nodes of PDGs by the numerical ID of statements in HOPE [26], and average the embeddings of all tokens in each PDG node as its initial embedding [25] in GGNN[6]. After graph embedding, we add a max pooling layer on all nodes of PDGs to obtain the final code representation.

To evaluate the effectiveness of source code classification, we use the test accuracy metric, which computes the percentage of correct classifications for the test set.

2) Code Clone Detection: There are generally four different types of code clones [54]. Type-1 is identical code fragments in addition to variations in comments and layout; Apart from Type-1, Type-2 is identical code fragments except for different identifier names and literal values; Type-3 is syntactically similar code snippets that differ at the statement level; Type-4 is syntactically dissimilar code snippets that implement the same functionality. For BCB, the similarity of clone pairs is defined as the average result of line-based and token-based metrics [47]. The similarity of two fragments of Type-1 and Type-2 is 1. Type-3 is further divided into Strongly Type-3 and Moderately Type-3, of which the similarities are in range $[0.7, 1)$ and $[0.5, 0.7)$, respectively. The similarity of Type-4 is in range $[0, 0.5)$ and its clone pairs take up more than 98% over all clone types. While in OJ, two programs for the same problem form a clone pair of unknown type.

As Table I shows, similar to the previous work [6], we choose 500 programs from each of the first 15 programming problems in OJ, namely OJClone. It will produce more than 28 million clone pairs which is extremely time-consuming for comparison, thus we randomly select 50 thousand samples instead. Likewise, we parsed nearly 6 million true clone pairs and 260 thousand false clone pairs from BCB. We compare our approach with existing state-of-the-art neural models for clone detection including RAE [5] and CDLH [6]. For RAE, the unsupervised vectors of programs are obtained by the authors' open-source tool[7] and we use them for supervised training, namely **RAE+**. Its configurations are set according to their paper. **CDLH** is not made public by the authors, so we directly cite their results from the paper since their experiments share the same datasets with ours. Since other traditional clone detection methods like DECKARD [55] and common neural models such as doc2Vec[8] have been compared in RAE and CDLH, we omit them in our experiment. Similar to the experiments on code classification, we also compare with the two PDG-based Graph embedding approaches in OJClone. However, BCB mainly contains incomplete and uncompilable method-level code fragments, we fail to get their PDGs.

Since the code clone detection can be formulated as a binary classification problem (clone or not), we choose the commonly-used Precision (P), Recall (R) and F1-measure (F1) as evaluation metrics.

[5]https://frama-c.com/
[6]https://github.com/Microsoft/gated-graph-neural-network-samples
[7]https://github.com/micheletufano/AutoenCODE
[8]https://radimrehurek.com/gensim/models/doc2vec.html

TABLE II
COMPARED APPROACHES FOR CODE CLASSIFICATION

Groups	Methods	Test Accuracy(%)
SVMs	SVM+TF-IDF	79.4
	SVM+N-gram	84.7
	SVM+LDA	47.9
Neural models	TextCNN	88.7
	LSTM	88.0
	TBCNN	94.0
	LSCNN	90.9
	PDG+HOPE	4.2
	PDG+GGNN	79.6
Our approach	ASTNN	**98.2**

TABLE III
CODE CLONE DETECTION MODELS ON BCB

Type	RAE+			CDLH			ASTNN		
	P	R	F1	P	R	F1	P	R	F1
BCB-T1	100	100	100	-	-	100	100	100	100
BCB-T2	86.5	97.2	91.5	-	-	100	100	100	100
BCB-ST3	79.9	72.2	75.9	-	-	94	99.9	94.2	97.0
BCB-MT3	66.4	74.8	70.3	-	-	88	99.5	91.7	95.5
BCB-T4	76.3	58.7	66.3	-	-	81	99.8	88.3	93.7
BCB-ALL	76.4	59.1	66.6	92	74	82	**99.8**	**88.4**	**93.8**

TABLE IV
CODE CLONE DETECTION MODELS ON OJCLONE

Metric	RAE+	CDLH	PDG+HOPE	PDG+GGNN	ASTNN
P	52.5	47	76.2	77.3	**98.9**
R	68.3	73	7.0	43.6	**92.7**
F1	59.4	57	12.9	55.8	**95.5**

D. Research Questions and Results

Based on the evaluation on the two tasks, we aim to investigate the following research questions:

RQ1: How does our approach perform in source code classification? In the task of code classification, the samples are strictly balanced among the 104 classes and our evaluation metric is the test accuracy. Experimental results are provided in Table II. The best results are shown in bold.

We can see that traditional methods such as SVMs perform poorly in our experiment. These methods mainly rely on the semantics of tokens or shallow semantics features of programs to distinguish the code functionalities, but the tokens used in OJ dataset are mostly arbitrary. For example, the names of identifiers are usually a, i, j , etc. Thus most tokens contribute little to recognize the functionalities.

For those neural models, the results of TextCNN and LSTM are better than token-based methods above because they can capture some local functional features. For example, the semantics of a short *scanf* statement can be captured by the sliding window of TextCNN or the memory cell unit in LSTM. As a neural network based on entire ASTs, TBCNN significantly improves the accuracy since it uses the convolution sliding window over ASTs to capture tree structural features. LSCNN has a relatively competitive performance among existing neural models. This can be inferred that the sequential information of statements does contribute to recognize the functionality, but the accuracy is still lower than TBCNN because it cannot capture the rich structural semantics. Graph-based approaches including HOPE and GGNN on PDGs perform poorly among the above approaches. In particular, PDG with HOPE gets an accuracy of only 4.2%, because the nodes of PDGs are represented by their numerical ID which miss lexical knowledge and only focuses on the explicit dependency information in a high abstraction level [26]. PDG with GGNN uses tokens for node embedding and has an improvement, but still lacks the syntactical information.

Among all the approaches, our model achieves the best accuracy. Specifically, our model improves TBCNN by 4.2%, since our ASTNN model performs RvNN on much smaller ST-trees than original ASTs. Unlike existing neural models, our model does not use the sliding window and binary tree transformation. Instead, it captures knowledge about AST statements and the sequential dependencies between statements.

RQ2: How does our approach perform in code clone detection? In this research question, we want to explore whether our model is effective on the challenging problem of code clone detection. We conduct experiments on the OJClone and BCB datasets.

From OJClone, we sample 50 thousand clone pairs for experiments. While from BCB, we firstly sample 20 thousand false clone pairs as the negative samples. For Type-1 to Strongly Type-3, we fetch all the true clone pairs belonging to that type as positive samples since their numbers are less than 20 thousand. For other types, we sample 20 thousand true clone pairs. Then we turn them into five groups to detect each type.

The detailed results are shown in Table III (BCB) and IV (OJClone). In Table III, as mentioned before, we cite the results of CDLH from [6]. Since there are no P and R values reported for detailed clone types, we fill them by "-" instead. BCB-ST3 and BCB-MT3 represent Strongly Type-3 and Moderately Type-3 in BCB, respectively, and so on. The BCB-ALL is a weighted sum result according to the percentage of various clone types [6]. In Table IV, we compare with two more PDG-based Graph embedding methods PDG+HOPE and PDG+GGNN as described before.

We first discuss the performances of RAE+, CDLH, and our ASTNN model on BCB. Obviously, all the three approaches are quite effective in recognizing the similarities of two code fragments in Type-1 and Type-2, since both code fragments are almost the same excepting different identifier names, comments and so on. While for other types of BCB, RAE+ performs much worse than the other two approaches since it has no mechanism on memorizing history information such as LSTM or GRU in CDLH and ASTNN. Comparing CDLH with our approach, we can see that ASTNN outperforms CDLH in terms of F1-measure especially for Type-4. In BCB Type-4, false clone pairs share syntactical similarity as well, which

TABLE V
COMPARISON BETWEEN THE PROPOSED MODEL AND ITS DESIGN ALTERNATIVES

Description	Code Classification	Clone Detection	
	Accuracy	F1(OJClone)	F1(BCB)
AST-Full	96.2	87.7	85.7
AST-Block	97.3	92.9	90.2
AST-Node	96.7	92.1	87.4
Removing Pooling-I	98.1	95.2	93.1
Removing Pooling-II	96.0	94.0	90.0
LSTM instead of GRU	97.8	**95.6**	92.3
Long code fragments	95.7	92.9	93.5
ASTNN	**98.2**	95.5	**93.8**

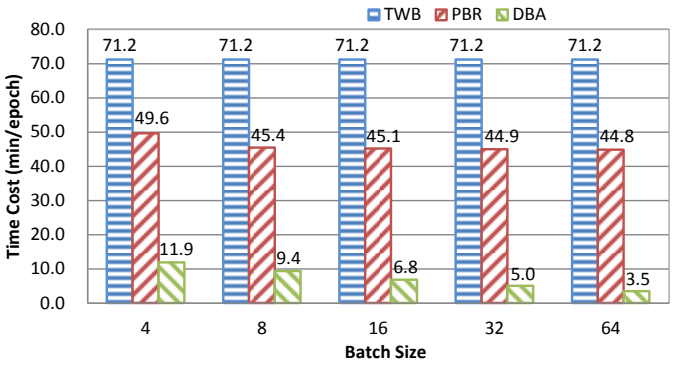


Fig. 5. The time cost of different batching methods

is validated to be coincidental [47] and is challenging to be distinguished. This indicates our ASTNN model can capture more subtle syntactical difference and complex semantics than CDLH by overcoming its limitations described in Section III and capturing sequential naturalness of statements.

In OJClone, similar results can be observed by RAE+ and our model. However, CDLH performs much worse than on BCB. Unlike BCB, the variable names of programs in OJClone are usually meaningless, thus CDLH may miss a lot of lexical information and can only capture some syntactical information. By contrast, our ASTNN model can further measure the functional similarities by learning more local syntactical knowledge and the global sequential naturalness among statements. Similar to code classification, we also compare with PDG-based Graph embedding techniques HOPE and GGNN. They achieve worse performance than our ASTNN model due to the facts mentioned in the last research question.

RQ3: What are the effects of different design choices for the proposed model? We conduct experiments to study how different design choices affect the performance of the ASTNN model on the two tasks. As shown in Table V, we consider the following design choices:

Splitting granularities of ASTs. Given a large AST, there are many ways to split it into different sequences of non-overlapping small trees. The two extreme ways are treating the original full AST as one special subtree (AST-Full), or extracting all nodes of the AST as special "trees" (AST-Node). Besides the statement level splitting, another possible way (AST-Block) is to split the AST according to blocks (compound statements that include multiple statements within the same brace pairs). After splitting, the follow-up encoding and bidirectional GRU processings are the same as those in ASTNN. We can see that AST-Block and ASTNN outperform both extreme splitting approaches of AST-Full and AST-Node. Our ASTNN model achieves the best performance, as analyzed in Section III, this is because it has a good trade-off between the size of ST-tree and the richness of syntactical information.

Pooling. In our ASTNN model, we use the max pooling on ST-trees in the statement encoder (Pooling-I) and the max pooling layer on the statement sequences after the recurrent layer (Pooling-II) as described in Section III. We study whether the two pooling components affect the performance or not by removing them and directly using the last layer hidden states. From the table, we can see that the pooling on statement sequences provides a comparatively significant performance boost, whereas pooling on ST-trees matters little. This shows that different statements of the same code fragments actually have different weights.

LSTM. In the recurrent layer of our proposed model, we use GRU by default. If replacing GRU by LSTM, the results indicate that overall LSTM has a slightly poor but comparative performance with GRU. We prefer GRU in our ASTNN model since it can achieve more efficient training.

Long code fragments. Considering only long code fragments which have more than 100 statements, the percentage of long code fragments is 1.6% in OJ, and the percentage of clone pairs which include at least one long code fragment is 4.1% in BCB. From the table, we can see that our ASTNN model can also deal with long sequences of statements well and the performance on long code fragments remains good.

RQ4: To what extent does the proposed dynamic batching algorithm contribute to the efficiency? As we described in Section III, our statement encoder can accept batch samples with arbitrary tree structures as inputs, thus can accelerate the speed of training. However, it is still unknown how efficient this algorithm is. In order to test and verify its capability, we train our model in three different ways: totally without batching (TWB), batching only on the recurrent layer (PBR), batching on the recurrent layer and the encoding layer by using our dynamic batching algorithm in Algorithm 1 (DBA). In detail, TWB means calculating one sample each time; PBR accepts batch samples, but encodes only one ST-tree at each time and performs batching on the recurrent layer by padding sequences of ST-tree vectors; DBA encodes all batch samples of ST-trees at once and then deals with ST-tree sequences as PBR does. The experiment is conducted for Task 1 and the time cost is the average running time of training and testing for each epoch.

In the experiment, the batching has no effect on the performance but changes the efficiency a lot. We find that the average time required by TWB is 71.2 minutes per epoch. From Figure 5 we can see that both PBR and DBA speed up the training and testing process when compared with TWB. DBA shows a significant improvement over the others. Furthermore, DBA

runs 12+ times faster than PBR and 20+ times faster than TWB when the batch size is 64. This confirms that the proposed batching algorithm makes our ASTNN model more efficient.

VI. Related Work

A. Source Code Representation

How to effectively represent source code is a fundamental problem for software engineering research.

Traditional IR and machine learning methods have been widely used for textual token-based code representation. Programs are transformed to regularized token sequences for code clone detection [3]. SourcererCC [4] has an improvement by exploiting token ordering along with an optimized inverted-index technique. Besides the lexical information, Deckard [55] enriches programs with some syntax-structured information for clone detection as well. Based on the statistical and machine learning methods, the n-gram model [1] and SVM [45] are used for classifying source code authorship and domains. Maletic et al. [56] adopts LSI to identify semantic similarities of code fragments, and the cohesion of classes in software is evaluated by LDA [15].

Recently deep learning based approaches have attracted much attention to learn distributed representation of source code [57]. Raychev et al. [58] adopts RNN and n-gram model for code completion. Allamanis et al. [59] uses a neural context model to suggest method and class names. For defect prediction, semantic features are extracted from source code by a deep belief network [60]. DeepBugs [61] represents code via word2vec for detecting name-based bugs. In order to capture the syntactical information of ASTs, White et al. [5] exploits a recursive auto-encoder over the ASTs with pre-trained token embeddings. TBCNN [2] uses custom convolutional neural network on ASTs to learn vector representations of code snippets. CDLH [6] incorporates Tree-LSTM to represent the functionality semantics of code fragments. Furthermore, Allamanis et al. [25] performs Gated Graph Neural Networks on program graphs which track the dependencies of the same variables and functions to predict variable names and detect variable misuses. DeepSim [62] encodes code control flow and data flow into a semantic matrix for measuring code functional similarity. Multiple different code representations such as identifiers, CFGs and bytecodes can also be integrated by the ensemble learning technique [26]. Compared with these neural networks, our model focuses on improving existing AST-based methods and can capture the lexical, statement-level syntactical knowledge and the sequential naturalness of statements.

B. Deep Learning in Software Engineering

In recent years, there are many emerging deep learning applications in software engineering. DeepAPI [43] uses a sequence-to-sequence neural network to learn representations of natural language queries and predict relevant API sequences. Lam et al. [63] combines deep neural network with IR technique to recommend potential buggy files. Xu et al. [64] adopts word embeddings and convolutional neural network to predict the related questions in StackOverflow. The neural machine translation is used to automatically generate commit messages [10]. Guo et al. [65] proposes a RNN based neural network to generate trace links. A joint embedding model is used in code search to map source code and natural language descriptions into a unified vector space for evaluating semantics similarity [66]. The above related work mainly uses neural network models to understand software-related natural language texts for various tasks while we focus on the neural representation of source code.

VII. Threats to Validity

There are three main threats to the validity. First, the OJ dataset is not collected from the real production environment. However, BigCloneBench includes code snippets of real-world Java repositories from SourceForge [47], which reduces this threat. The second threat is about the construction of OJClone. As we described, programs under the same problem belong to a clone pair. This leads to the uncertainty about whether they are true clone pairs, although similar practice has been done by previous work [6]. Nevertheless, BigCloneBench is also used for validation where the code clones are inspected manually. Therefore, we believe it is of little influence on experimental results. The last threat is that we cannot reproduce the approach of CDLH due to some details missed in that paper. Alternatively, we construct the same datasets described in their paper to reduce this threat. We will make supplement for comparison when the CDLH tool is available.

VIII. Conclusion

In this work, we have presented an efficient AST-based Neural Network (ASTNN) to learn vector representations of source code fragments, which can capture the lexical, statement-level syntactical knowledge and naturalness of statements. The model decomposes large ASTs of code fragments into sequences of small statement trees, obtains statement vectors by recursively encoding multi-way statement trees, and finally learns the vector representations of code fragments by leveraging the naturalness of statements. We have evaluated ASTNN on two common program comprehension tasks: source code classification and code clone detection. The experimental results show that our model significantly outperforms existing approaches. Our code and experimental data are publicly available at https://github.com/zhangj1994/astnn.

In the future, we will further evaluate the proposed model on larger-scale datatsets in different programming languages and for a variety of software engineering tasks. We will also explore other neural models to capture more deep semantics of source code.

Acknowledgment

This work was supported partly by National Key Research and Development Program of China (No.2016YFB1000804), partly by National Natural Science Foundation of China (No.61702024, 61828201 and 61421003).

REFERENCES

[1] G. Frantzeskou, S. MacDonell, E. Stamatatos, and S. Gritzalis, "Examining the significance of high-level programming features in source code author classification," *Journal of Systems and Software*, vol. 81, no. 3, pp. 447–460, 2008.

[2] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *AAAI*, vol. 2, no. 3, 2016, p. 4.

[3] 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*, vol. 28, no. 7, pp. 654–670, 2002.

[4] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 2016, pp. 1157–1168.

[5] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 87–98.

[6] H.-H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 2017, pp. 3034–3040.

[7] M. DAmbros, M. Lanza, and R. Robbes, "Evaluating defect prediction approaches: a benchmark and an extensive comparison," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 531–577, 2012.

[8] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "An empirical comparison of model validation techniques for defect prediction models," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 1–18, 2017.

[9] S. Haiduc, J. Aponte, and A. Marcus, "Supporting program comprehension with source code summarization," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*. ACM, 2010, pp. 223–226.

[10] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 135–146.

[11] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *2012 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 14–24.

[12] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *Journal of the American society for information science*, vol. 41, no. 6, p. 391, 1990.

[13] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *Journal of machine Learning research*, vol. 3, no. Jan, pp. 993–1022, 2003.

[14] R. Tairas and J. Gray, "An information retrieval process to aid in the analysis of code clones," *Empirical Software Engineering*, vol. 14, no. 1, pp. 33–56, 2009.

[15] Y. Liu, D. Poshyvanyk, R. Ferenc, T. Gyimóthy, and N. Chrisochoides, "Modeling class cohesion as mixtures of latent topics," in *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 2009, pp. 233–242.

[16] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella, "Using IR methods for labeling source code artifacts: Is it worthwhile?" in *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*. IEEE, 2012, pp. 193–202.

[17] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshynanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms," in *Software Engineering (ICSE), 2013 35th International Conference on*. IEEE, 2013, pp. 522–531.

[18] J. F. Pane, B. A. Myers *et al.*, "Studying the language and structure in non-programmers' solutions to programming problems," *International Journal of Human-Computer Studies*, vol. 54, no. 2, pp. 237–264, 2001.

[19] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[20] S. Hochreiter, "The vanishing gradient problem during learning recurrent neural nets and problem solutions," *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6, no. 2, pp. 107–116, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1142/S0218488598000094

[21] P. Le and W. H. Zuidema, "Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive LSTMs," in *Proceedings of the 1st Workshop on Representation Learning for NLP*, Berlin, Germany, 2016, p. 8793.

[22] S.-Y. Cho, Z. Chi, W.-C. Siu, and A. C. Tsoi, "An improved algorithm for learning long-term dependency problems in adaptive processing of data structures," *IEEE Transactions on Neural Networks*, vol. 14, no. 4, pp. 781–793, 2003.

[23] X. Zhu, P. Sobhani, and H. Guo, "Long short-term memory over recursive structures," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 1604–1612. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045289

[24] M. Ou, P. Cui, J. Pei, Z. Zhang, and W. Zhu, "Asymmetric transitivity preserving graph embedding," in *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: ACM, 2016, pp. 1105–1114. [Online]. Available: http://doi.acm.org/10.1145/2939672.2939751

[25] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

[26] G. B. M. D. P. M. W. Michele Tufano, Cody Watson and D. Poshyvanyk, "Deep learning similarities from different representations of source code," in *15th International Conference on Mining Software Repositories*, 2018.

[27] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[28] E. M. Myers, "A precise inter-procedural data flow algorithm," in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '81. New York, NY, USA: ACM, 1981, pp. 219–230. [Online]. Available: http://doi.acm.org/10.1145/567532.567556

[29] "Llvms analysis and transform passes," https://llvm.org/docs/Passes.html, accessed August 3, 2018.

[30] J. Gosling, B. Joy, G. L. Steele, G. Bracha, and A. Buckley, *The Java Language Specification, Java SE 8 Edition*, 1st ed. Addison-Wesley Professional, 2014.

[31] T. Mikolov, M. Karafiát, L. Burget, J. Černockỳ, and S. Khudanpur, "Recurrent neural network based language model," in *Eleventh Annual Conference of the International Speech Communication Association*, 2010.

[32] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 837–847.

[33] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the "naturalness" of buggy code," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 428–439. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884848

[34] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv preprint arXiv:1409.0473*, 2014.

[35] D. Tang, B. Qin, and T. Liu, "Document modeling with gated recurrent neural network for sentiment classification," in *Proceedings of the 2015 conference on empirical methods in natural language processing*, 2015, pp. 1422–1432.

[36] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[37] S. Paul and A. Prakash, "A framework for source code search using program patterns," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 463–475, 1994.

[38] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 364–374.

[39] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 2014, pp. 313–324.

[40] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng, "Parsing natural scenes and natural language with recursive neural networks," in *Proceedings of the 28th international conference on machine learning (ICML-11)*, 2011, pp. 129–136.

[41] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, vol. 1, 2015, pp. 1556–1566.

[42] 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.

[43] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 2016, pp. 631–642.

[44] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "Mudablue: An automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.

[45] M. Linares-Vásquez, C. McMillan, D. Poshyvanyk, and M. Grechanik, "On using machine learning to automatically classify software applications into domain categories," *Empirical Software Engineering*, vol. 19, no. 3, pp. 582–618, 2014.

[46] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[47] J. Svajlenko, J. F. Islam, I. Keivanloo, C. K. Roy, and M. M. Mia, "Towards a big data curated benchmark of inter-project code clones," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 476–480.

[48] A. Charpentier, J.-R. Falleri, D. Lo, and L. Réveillère, "An empirical assessment of bellon's clone benchmark," in *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2015, p. 20.

[49] P. Accioly, P. Borba, and G. Cavalcanti, "Understanding semi-structured merge conflict characteristics in open-source java projects," *Empirical Software Engineering*, pp. 1–35, 2017.

[50] Y. Kim, "Convolutional neural networks for sentence classification," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014, pp. 1746–1751.

[51] W. Zaremba and I. Sutskever, "Learning to execute," *arXiv preprint arXiv:1410.4615*, 2014.

[52] X. Huo and M. Li, "Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code," in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI'17. AAAI Press, 2017, pp. 1909–1915. [Online]. Available: http://dl.acm.org/citation.cfm?id=3172077.3172153

[53] Y. Li, D. Tarlow, M. Brockschmidt, and R. Zemel, "Gated graph sequence neural networks," *arXiv preprint arXiv:1511.05493*, 2015.

[54] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[55] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 96–105. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2007.30

[56] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," in *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 103–112.

[57] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *arXiv preprint arXiv:1709.06182*, 2017.

[58] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Acm Sigplan Notices*, vol. 49, no. 6. ACM, 2014, pp. 419–428.

[59] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 38–49.

[60] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 297–308.

[61] M. Pradel and K. Sen, "DeepBugs: A learning approach to name-based bug detection," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276517

[62] G. Zhao and J. Huang, "DeepSim: Deep learning code functional similarity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: ACM, 2018, pp. 141–151. [Online]. Available: http://doi.acm.org/10.1145/3236024.3236068

[63] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Combining deep learning with information retrieval to localize buggy files for bug reports (n)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 476–481.

[64] B. Xu, D. Ye, Z. Xing, X. Xia, G. Chen, and S. Li, "Predicting semantically linkable knowledge in developer online forums via convolutional neural network," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2016, pp. 51–62.

[65] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 2017, pp. 3–14.

[66] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proceedings of the 2018 40th International Conference on Software Engineering (ICSE 2018)*. ACM, 2018.