FCCA: Hybrid Code Representation for Functional Clone Detection Using Attention Networks

Wei Hua, Yulei Sui, Yao Wan, Guangzhong Liu, and Guandong Xu

***Abstract*—Code cloning, which reuses a fragment of source code via copy-and-paste with or without modifications, is a common way for code reuse and software prototyping. However, the duplicated code fragments often affect software quality, resulting in high maintenance cost. The existing clone detectors using shallow textual or syntactical features to identify code similarity are still ineffective in accurately finding sophisticated functional code clones in real-world code bases.**

This paper proposes FCCA, a deep-learning-based code clone detection approach on top of a hybrid code representation by preserving multiple code features, including unstructured (code in the form of sequential tokens) and structured (code in the form of abstract syntax trees and control-flow graphs) information. Multiple code features are fused into a hybrid representation which is equipped with an attention mechanism that pays attention to important code parts and features that contribute to the final detection accuracy. We have implemented and evaluated FCCA using 275,777 real-world code clone pairs written in Java. The experimental results show that FCCA outperforms several state-of-the-art approaches for detecting functional code clones in terms of accuracy, recall and F1 score.

***Index Terms*—Code clone detection, code representation, deep neural network, attention mechanism.**

I. Introduction

In software development, not "reinventing the wheel" is a double-edged sword. Reusing code from other software projects, a.k.a code cloning saves time and manpower, but it can also add significant high maintenance costs for the following key reasons [1], [2]: (a) *Redundancy*. Excessive use of code clones during software development breaks the principles of encapsulation. Unnecessary clones introduce unmanaged copy-and-pastes, which makes advancing the project difficult for subsequent developers [3], [4]. (b) *Licensing and plagiarism*. Many developers reuse code from other projects through open-source platforms (e.g., GitHub) without knowing the licensing terms and conditions of that use. The obvious result is serious copyright issues [5], and (c) *Reliability and security*. Cloning code from untrusted third-party libraries or buggy programs also poses big challenges to software analysis and testing [6], because it requires additional efforts to debug, refactor and integrate this type of cloned code [7], [8], [9].

***Existing Efforts and Limitations*.** Code clone detection is a fundamental way of improving software quality during software development and evolution. Standard practice divides code clones into four general categories based on complexity: textual clones (Type-I), lexical clones (Type-II), syntactic clones (Type-III), and functional clones (Type-IV) [2], [10].

Most researches to date have centered on detecting clone Types I-III clones [10], [11], [12], [13], [14]. The detection strategies usually start with converting source code into a bag-of-tokens or abstract syntax trees (ASTs). Clone detection is then performed by statistically counting tokens to identify the similarity between two code fragments by applying a distance metric. Although many of these strategies have been successful at detecting their target clone Types, few have achieved high accuracy in detecting Type-IV clones, which is the most difficult type for clone detection. Type-IV clones are functional clones, i.e., two fragments of code that do the same thing but not necessarily in the same way. They are the most textually unlike and the most difficult to detect.

***Insights*.** Hybrid (or multimodal) representation learning (HRL) [15], [16], [17] has recently emerged as a promising branch of deep representation learning, demonstrating its power in embedding heterogeneous information into an unified low dimensional space. The resulting representation is particularly useful for supporting better modeling of complex data, such as source code. Further, the concept of attention mechanism, has emerged out of representation learning and is now developing into an important mechanism of precision enhancement. Beyond boosting performance in a wide range of applications, attention mechanism is also playing a key role in improving the explainability of deep learning models, benefiting machine translation [18], machine reading comprehension [19], image captioning [20] and document classification [21].

This paper aims to develop the first integration of attention mechanism into a code clone detection task, and moreover, to use this attention to locate the elusive Type-IV functional code clones. Our approach relies on hybrid code representations that capture both structured and unstructured information while the attention mechanism interprets and weights the contributions of the code's features to offer improved accuracy when detecting complicated functional code clones.

***A Motivating Example*.** Figure 1 (middle) shows two simple code snippets. These snippets are Type-IV functional clones. Corresponding ASTs for these snippets are shown on the left and the control-flow graphs (CFGs) appear on the right. At

W. Hua is with the Collage of Information Engineering, Shanghai Maritime University, Shanghai 201306, China (E-mail: huawei03@stu.shmtu.edu.cn)

Y. Wan is with Zhejiang University, Hangzhou, Zhejiang, P. R. China. (E-mail: wanyao@zju.edu.cn)

G. Liu is with the Collage of Information Engineering, Shanghai Maritime University, Shanghai 201306, China (E-mail: gzhliu@shmtu.edu.cn)

Y. Sui, and G. Xu are with the School of Computer Science, University of Technology, Sydney, NSW 2008 Austrialia. (E-mail: Yulei.Sui@uts.edu.au; Guandong.Xu@uts.edu.au)

* Corresponding authors: Yulei Sui, Guangzhong Liu and Guandong Xu

The work of this paper was done during the first author's visiting at University of Technology Sydney (UTS), Australia.

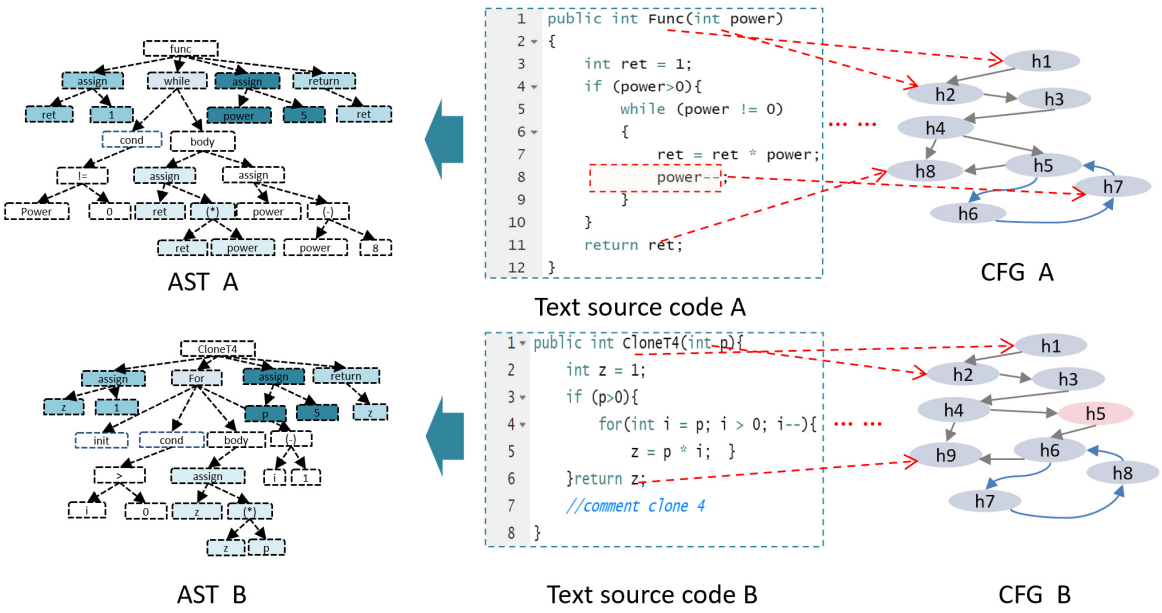


Fig. 1: A motivating example.

the different granularities, different code representations reflect different code features. In this paper, we are concerned with three granularities:

(a) Lexical features (text). The simplistic approach to handling source code is to treat the source code as a series of tokens. For an example, a post-decrement operator "power--;" and a pre-decrement operator "--power;" are functionally different (highlighted in the red frame in text source code A). Intuitively, it is easy to capture the difference between two such operations in a text representation, whereas syntactical (i.e., AST) and structured (i.e., CFG) representations cannot fully differentiate between a post-decrement and a pre-decrement. This is because, unlike text representations, neither includes sequential information. Therefore, the lexical feature (code tokens, i.e., text) have become a basic and essential feature for code clone detection.

(b) Syntactical features (AST). ASTs are a common way to represent syntactical statement-related features in source code [22], [14], [23]. In Figure 1, the method statements "ret = ret*power;" at line 7 in "Func()" and "z = p * i;" at line 5 in "CloneT4()" can be represented using the same operation, i.e., "Binary Expr : times" using an AST with two child nodes (* denotes "times" in the two statements). However, source code in the form of plain texts or CFGs cannot fully capture this syntactical similarity.

(c) Structured features (CFG). Compared to the syntactical level, structured features represent source code at a higher level, reflecting logic flows and program execution orders. Figure 1 demonstrates that the two code snippets are recognizably alike when formulated as control-flow graph, with the pink node as the only difference. Both snippets contain an "if" statement and a loop with similar logic and program control-flows. Each node in the CFG $(h1, h2, h3, \ldots h_t)$ denotes a statement in the code snippet. In this case, using ASTs or plain texts cannot capture this type of structure information because neither is capable of accurately reflecting the program control-flows.

In summary, a precise clone detector requires a hybrid code representation that can comprehensively preserve heterogeneous code features at all these three levels: lexical, syntactic, and structured, simply relying on an individual feature may result in inaccurate prediction results. Furthermore, the detection mechanism operating on top of the hybrid representation should be able to pay attention to important code parts in the presence of different feature scenarios to improve overall detection accuracy.

Our Solution and Contributions. Our solution is FCCA– a learning approach based on hybrid representations for functional code clone detection. FCCA preserves both unstructured and structured information in code representations at three granularities: texts (in the form of sequential code tokens), ASTs and CFGs. The three representations are then fused into a common latent space to represent code features in a comprehensive way. By training a deep-learning model with an attention mechanism, the model considers the importance of these three features of a program when measuring the similarity between pairs of code fragments. A further benefit of incorporating this attention mechanism is that it enhances the explainability and flexibility of the deep-learning model. We summarize the main contributions of this paper as follows.

- We propose FCCA, a new framework for functional code clone detection. FCCA is based on hybrid representations that preserve and fuse heterogeneous structured and unstructured code information from text, AST, and CFG representations.
- To the best of our knowledge, this work is the first to integrate an attention mechanism into a code clone detection scheme and demonstrate its advantages. The insights provided by our efforts to train a deep-learning network with an attention mechanism should enrich our

TABLE I: Code clone samples of Type I, Type II, Type III, Type IV.

```
// original code
public int main(int m, int n)
{
 if (m>0)
  n=m+1;
  return n;
}
```

```
// Type I
public int main(int m, int n)
{
 if (m>0)
  n=m+1;
  return n;
  //some comment
}
```

```
// Type II
public int funcT2(int u, int v)
{
 if (u>0)
  v=u+1;
  return v;
}
```

```
//Type III
public int funcT3(int u , int v)
{
    int p=0;
    p=u;
     if (p>0)
       v=p+1;
    return v;
}
```

```
//Type IV
public int funcT4(int v, int u)
{
  return v > 0 ?( v + 1 ) : u;
}
```

general understanding of how this powerful technology can be used to improve the accuracy of clone detection.
- We have conducted extensive experiments using the dataset from `BigCloneBench` [24], which is a benchmark dataset consisting of well-labeled clone pairs. The results show that FCCA outperforms the state-of-the-art clone detectors. Our tool and the dataset is publicly available at *https://github.com/preesee/CodeCloneDetection*.

Organization. The reminder of this paper is organized as follows. Section II introduces necessary background knowledge of code cloning, code representations, deep recurrent neural network and attention mechanisms. Section III presents the framework of FCCA and its main modules in details. Section IV provides details of the evaluation, including the dataset, experiment setup, experiment results and our findings from the experiments. Section V presents the discussion on the strength, threats to validity and limitations of our approach. Section VI discusses the related works. Finally, we conclude this paper in Section VII.

II. BACKGROUND

This section presents the preliminary knowledge of the techniques used in our approach. We begin with definitions of the different types of code clones, following previous practices [10], [25]. We then review three basic code representation methods with simplified notations. Likewise, we introduce the deep recurrent neural network and attention mechanism we use in this paper.

A. Clone Type Definitions

Clone types defines the level of similarity for a pair of code fragments. If a pair of code fragments are cloned, their similarity level is one of the following.

Clone Type-I (Textual Clone). The code fragments are identical except for differences in white-space, layout and comments.

Clone Type-II (Lexical Clone). The code fragments are identical except for differences in identifier names and literal values, and the exceptions listed for Type-I.

Clone Type-III (Syntactic Clone). The code fragments are syntactically similar but differ at the statement level; statements may have been added, modified, and/or removed. The differences listed for Type-I and Type-II clones may also apply.

Clone Type-IV (Functional Clone). These code fragments are semantically similar in terms of what they do but different in how they do it.

Table I provides Java code samples for each of the four clone types. Note that there is no specification for a minimum syntactical similarity in a Type-III clone pair, and the lack of consensus about this similarity in previous literature often makes it difficult to distinguish between Type-III and Type-IV clones. Hence, the clone detection benchmark BIGCLONEBENCH [10], [24] sets out some range bands of syntactical similarity to create finer-grained subcategories of Type-III and Type-IV clones. The range bands and corresponding subcategories are:

- Very Strongly Type-III (**VST3**) clones have a syntactical similarity between 90% (inclusive) and 100% (exclusive).
- Strongly Type-III (**ST3**) in 70-90%.
- Moderately Type-III (**MT3**) in 50-70%.
- Weakly Type-III/Type-IV (**WT3/4**) in 0-50%.

The focus in this paper is on Weakly Type-III/Type-IV functional clones as defined above.

B. Code Representations

There are three main approaches to generating source code representations in the existing literature.

Token-based Code Representation. The code fragment is broken down into a sequence of tokens and the frequency of each token appears in a document is counted with the bag-of-words representation [10]. The similarity of each pair is either calculated with a matching algorithm (e.g., Jaccard similarity) or the distance between the two code fragments is measured after embedding the bags-of-words in latent spaces. However, abstracting code in the form of token sequences produce a high number of false positives since these techniques ignore the structural information in source code. As a result, their precision is usually insufficient for analyzing complicated clones, such as Type-III and Type-IV clones.

AST-based Code Representation. Code representations take the form of abstract syntax trees (ASTs) [13], [23]. Code clones are detected by measuring the similarity between corresponding ASTs or their sub-trees. For a clear illustration, let T be an AST of a code fragment c. T only has one root node with no parent, and consists of n code tokens. Each token is a node on T, which has only one parent. Once a tree representation of a fragment c has been generated, it can be converted into a vector comprising a sequence of tokens $[tok_1, tok_2, tok_3, \ldots tok_n]$ by traversing T. Comparing the ASTs of two code fragments provides a distance value that quantifies the similarity between the two code fragments [22].

Graph-based Code Representation. Graph representations of programs [26], such as program dependency graphs (PDG) [27], control-flow graphs (CFG) and call graphs, can be used to model structural information of code. We use CFGs in our hybrid learning model to preserve the high-level control-flow execution information. A control-flow graph of a code fragment (or a function) is denoted as $G = (V, E)$, consisting of a set of n vertices $V = \{v_1, v_2, \ldots, v_n\}$ and a set of m directed edges $E = \{e_1, e_2, \ldots, e_m\}$ [28]. Each node v_i represents a code statement and each edge e_i connects two nodes, signifying the control-flow or execution order between two statements. Thus, the graph depicts the logic structure and the execution sequence of a code fragment.

Code representations can come in different forms both unstructured and structured, all of which can contribute to detecting functional features and, hence, clones. Further, different types of code representations reflect different aspects of the source code, which means a wide variety of code features can be detected. However, for code clone detection, it is important to design a hybrid yet accurate representation that can coverage important code features to better model the code semantics.

C. Deep Recurrent Neural Network

A simple neural network usually consists of three layers (i.e., an input layer, a hidden layer and an output layer), with each layer comprising a set of nodes. Computation happens in the nodes (similar to neuron-like switches): a node accepts input from raw data and/or its predecessors and combines it with a set of coefficients or weights to either amplify or diminish the input. The result is then passed to its successor nodes. The outputs from the last (output) layer are combined and fed into an activation function to determine whether the result should be progressed further and therefore, affect the final outcome. Further progression could be assigning, for example, a classification task. If the signal passes through, the network is activated.

The difference between a deep neural network (DNN) and a simple neural network is that DNNs have more than one hidden layer between the input and output layers, i.e., the data must pass through more layers of nodes, and this multi-step process typically results in more powerful pattern recognition. DNNs are artificial neural networks inspired by the neural architecture of a human brain. They can find the correct mathematical manipulation to capture appropriate correlations between input and outputs, such as linear/non-linear relationships.

Recurrent Neural Networks (RNNs) leverage the advantages of DNNs but are specifically designed to handle sequential data. Given a sequence $\mathbf{x} = [x_1, \ldots, x_T]$ as the input, at each time step $t \in [1, T]$, each input x_i may impact the entire representation of the sequence $\mathbf{x}$, then, a hidden-state vector $\mathbf{h}_t$ is generated as a proxy representation for the current representation at step t. To handle the sequence data, $\mathbf{h}_t$ is updated through the following equation:

$$\mathbf{h}_t = \sigma(\mathbf{W}x_t + \mathbf{U}\mathbf{h}_{t-1}), \tag{1}$$

where x_t is the input at time step t, $\mathbf{W}$ is a weight matrix connecting the inputs to the hidden-state vector, and $\mathbf{U}$ is a weight matrix on the hidden-state vector from the previous time step $\mathbf{h}_{t-1}$. The activation function σ filters less useful information for a client application and transfers useful information to the next input x_t for consecutive processing of the sequence $\mathbf{x}$. There are many options for an activation function. A typical example is a sigmoid function.

One major advantage of RNNs is that RNNs are able to connect previous data to the present information for an accurate prediction. For example, in computer vision, RNNs are used to improve pattern recognition by leveraging previous video frames to better understand the present frame.

Our goal is to apply the benefits of RNNs to functional code clone detection. The essence of the idea is to model the composition of features by capturing the complex non-linear relations between these features to detect the functional code clones. The distance between two embedding vectors of two code tokens in the latent space can be very different with respect to different code representations (e.g., the vectors of two consecutive tokens in plain texts may not be close to each other if these tokens are embedded based on the tree structure (e.g., AST)). Thus, the relations of these tokens on different code representations should be considered in holistic manner when designing the recurrent neural network to improve the overall accuracy in clone detection.

D. Attention Mechanism

RNNs are often used to model sequential information by considering time steps. However, this means that an RNN would treat all tokens in a sequence as being equally important, likely causing imprecise results for complicated client applications, such as code clone detection. However, integrating an attention mechanism into the training process of an RNN could be an effective method of enhancing existing deep-learning architectures to suit more complex tasks. Attention mechanisms help networks pay attention to the important parts of the input sequences so as to focus on learning and representing the correlations between the inputs and outputs in a more precise way.

An intuitive way to explain how attention mechanisms work is to revisit the motivating example of our code snippets, as illustrated in Figure 1. It is natural for developers to pay attention to keywords when reviewing the functionality of code snippets because they often reflect important control-flow/loop information in the source code. An attention mechanism places weights on the important items in an input, such as keywords "`for`" and "`while`", operators on ASTs, branch or joint nodes on the CFGs.

An attention mechanism is to compute an dependency score between elements from two sources [29]. Given the token embedding of a code sequence $x = [x_1, x_2, x_3, \ldots, x_t]$ and a vector representation of a query u, an attention function $f(x_i, u)$ is to measure the dependency between x_i and u, or the attention of u to x_i. All the attention scores of the t tokens are collected using Equation 2.

$$a = [f(x_i, u)]_{i=1}^{t}, \tag{2}$$

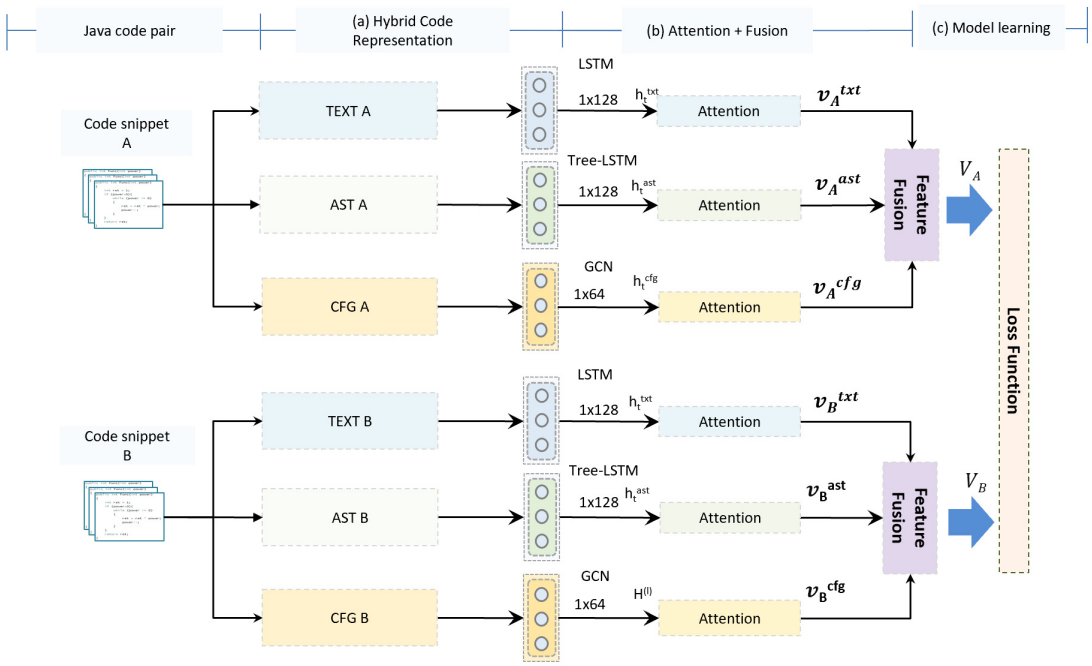


Fig. 2: The overview of FCCA architecture.

The scores are then normalized over the t tokens of $\mathbf{x}$ and transformed into a probability distribution using a `softmax` function as shown in Equation 4.

$$p(z=i|\mathbf{x},u) = \frac{\exp(f(x_i,u))}{\sum_{i=1}^{t}\exp(f(x_i,u))}, \tag{3}$$

where $p(z = i|\mathbf{x},u)$ indicate the weight (how important) that x_i contributes its information to u. Function $f(\cdot)$, which is used to measure the dependency or similarity between two vectors of two code tokens, will be detailed in Section III-B using specific attention functions for different code representations.

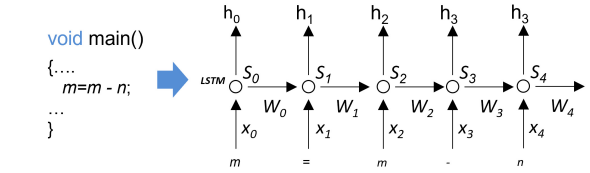
The final attention embedding vector $\mathbf{v}_x$ of $\mathbf{x}$ is a weighted sum of the vectors of all the tokens in $\mathbf{x}$ computed using Equation 4:

$$\mathbf{v}_\mathbf{x} = \sum_{i=1}^{t} p(z=i|\mathbf{x},u). \tag{4}$$

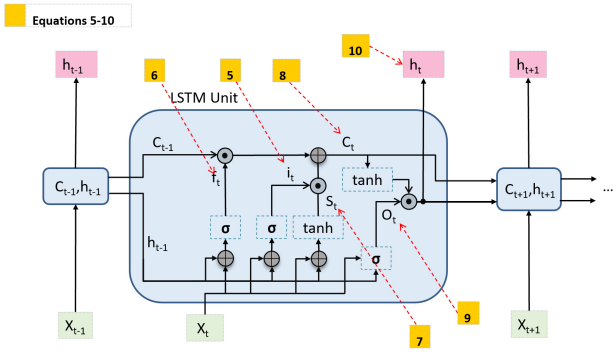
III. Proposed Approach

Figure 2 gives the overview of our clone detection approach using hybrid code representation with attention networks. It consists of three major components (a) the representations of source code at the lexical, syntactical and structural levels; (b) attention and feature fusion layers for each type of representation; and (c) the model training process, which contains a loss function.

In our model, the RNNs, the attention mechanisms and the feature fusion layers work together as additional layers that give higher weight to the salient tokens for each code representation. This method provides an effective way of understanding the importance of tokens among different features. These attention-empowered RNNs also represent a step forward towards an explainable deep-learning architecture. The aim of the loss function is to measure the similarity between fused hybrid representations and back propagate the error value to train the model until the training process converges.



(a) A sample of tokenization for source code.



(b) Convert source code to vectors.

Fig. 3: Lexical level code representation and token-based LSTM model.

A. Hybrid Code Representation

An ideal representation should be able to comprehensively preserves code features. We detail our hybrid code representation and its fusion as follows:

(a) Lexical Level. Raw source code in the form of plain texts may contain noises that do not relate to the code's func-

tionality, such as, redundant code comments, unused reference libraries or variables, unused code snippets for testing etc. These noises which may affect the clone detection are removed before our code embedding. The pre-processed texts of source code are segmented into a sequence of tokens. For example, with the Java method: "main() {int m=m - n; }", the text are tokenized into a textual sequence '{', 'main()', 'int', 'm', 'n', '-', '}' (Figure 3(a) demonstrates the tokenization process).

Next, the RNNs are used to handle the sequential tokens. There are several variants of RNN, we apply Long Short-Term Memory (LSTM) [30], [31] in our model to process sequential information. The LSTMs can capture sequential features given a token sequence $\mathbf{x} = [x_1, x_2, \ldots, x_N]$ as the inputs converted from the source code, where N is the length of the sequence. Figure 3(b) shows the inner structure of an LSTM where the component details are represented using Equations 5-10 highlighted in yellow. The current memory cell C_t (the important component of LSTM that saves the long-distance information for the input sequence) and hidden state h_t are updated for each token x_t at time step t:

$$i_t = \sigma(\mathbf{W}_i x_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i), \tag{5}$$

$$f_t = \sigma(\mathbf{W}_f x_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f), \tag{6}$$

$$S_t = \tanh(\mathbf{W}_c x_t + \mathbf{U}_c \mathbf{h}_{t-1} + \mathbf{b}_c), \tag{7}$$

$$C_t = (i_{(t)} \odot S_t + f_t \odot C_{t-1}), \tag{8}$$

$$O_t = \sigma(\mathbf{W}_{(o)} x_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o), \tag{9}$$

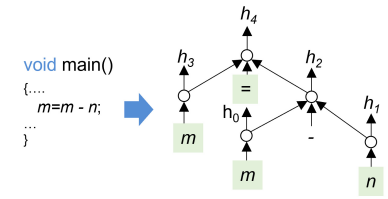
$$\mathbf{h}_t^{txt} = O_t \odot \tanh(C_t), \tag{10}$$

where i_t, f_t, S, and O_t denote an input gate, a forget gate, a state for updating the memory cell and an output gate, respectively. $\mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_c, \mathbf{W}_o$ and $\mathbf{U}_i, \mathbf{U}_f, \mathbf{U}_c, \mathbf{U}_o$ are weight metrics. $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_c, \mathbf{b}_o$ are bias-vectors, $\sigma(\cdot)$ is the activation function, and the operator $\odot$ is element-wise multiplication between vectors. After all processes, the final output of the method containing the sequential features of a code snippet is represented by a vector $\mathbf{h}_t^{txt}$.

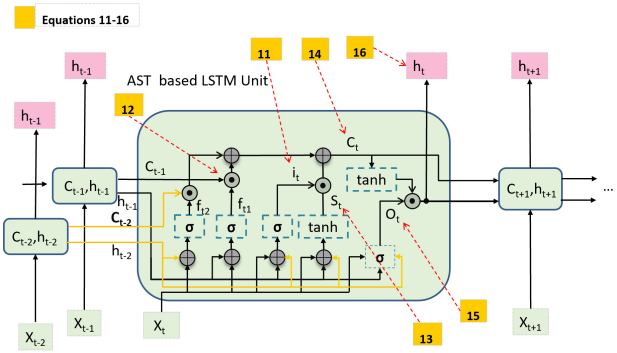
(b) Syntactic Level. The same Java method can also be parsed into a tree at the statement level by an AST-based LSTM [32]. Figure 4(a) demonstrates the conversion from a simple Java statement "m = m - n" to an AST. Although the lexical-level for the two statements "m = m - n" and "a = a - b" are different, at the lexical level, they are regarded as clones since they both perform the same operation, i.e., "-", reflected in their corresponding ASTs.

To capture this syntactical information, we selected a tree-based LSTM to parse ASTs. The difference between a tree-based LSTM and the sequential LSTM used for the lexical representations is that a tree-based LSTM accepts two inputs from two child nodes of one parent at one time step. Figure 4(b) shows this data processing method with an AST-based LSTM. Equations 11-16 are highlighted in yellow.

$$i_t = \sigma(\mathbf{W}_i x_t + \sum_{n=1}^{N} \mathbf{U}_i^n \mathbf{h}_{tn} + \mathbf{b}_i), \tag{11}$$



(a) A sample of converting a code statement "m = m - n" to an AST.



(b) AST based LSTM data process unit.

Fig. 4: Syntactic level code representation and AST-based LSTM model.

$$f_t = \sigma(\mathbf{W}_f x_t + \sum_{n=1}^{N} \mathbf{U}_f^n \mathbf{h}_{tn} + \mathbf{b}_f), \tag{12}$$

$$S_t = \tanh(\mathbf{W}_c x_t + \sum_{n=1}^{N} \mathbf{W}_c^n \mathbf{h}_{tn} + \mathbf{b}_c), \tag{13}$$

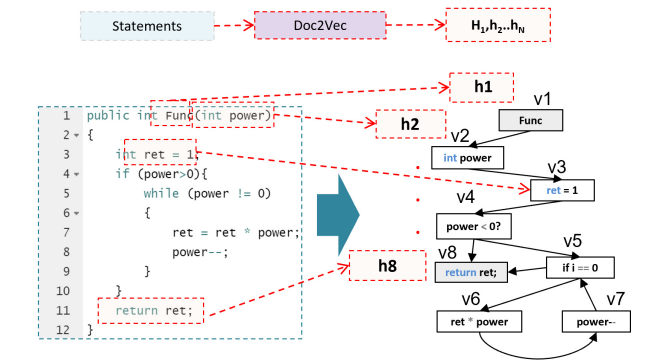
$$C_t = i_{(t)} \odot S_t + \sum_{n=1}^{N} f_t^n \odot C_{tn}, \tag{14}$$

$$O_t = \sigma(\mathbf{W}_{(o)} x_t + \sum_{n=1}^{N} \mathbf{U}_o^n \mathbf{h}_{tn} + \mathbf{b}_o), \tag{15}$$

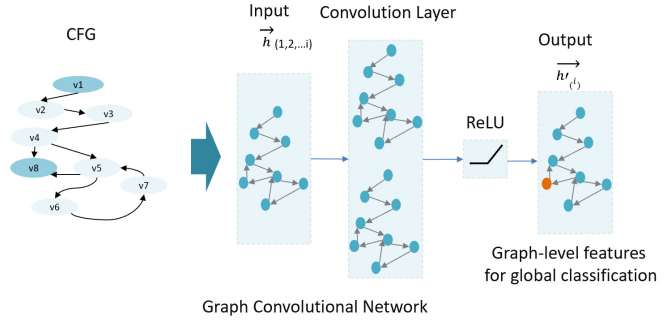
$$\mathbf{h}_t^{ast} = O_t \odot \tanh(C_t). \tag{16}$$

Note that N denotes the number of children of an AST, where the value of N varies for different AST nodes, which may cause problems in sharing parameters [13]. To simplify the process, we convert the generated ASTs into binary trees in two steps – a method also used by Hui et al. and Wan et al. [13], [32]: (1) Split the nodes with more than two children, then generate a new right child node together with the previous left child as its children. All children except the left-most child become children of the new node and this operation is performed recursively in a top-down way until only nodes with 0, 1 and 2 children are left; (2) Combine the left nodes with its children.

(c) Structural Level. We adopt the Graph Convolutional Networks (GCNs) in [33] as the base graph neural networks to model CFGs. GCN is known to generalize well-established neural models to work on arbitrarily structured graphs. The main idea of the graph convolution is to use a multi-layer neural network that operates directly on a graph and induces



(a) CFG of Func.



(b) Embed CFG in the vector representations using graph convolutional networks.

Fig. 5: Structured level representation of code through convolutional network.

the embedding vectors of nodes based on the features of their neighborhood nodes. In this section, we briefly introduce the process of the graph modeling for CFGs. A CFG $G = (V, E)$ of a Java method consists of a set of nodes $V = \{v_1, v_2, \ldots, v_N\}$, where we use program statements and a set of edges E to represent control-flows between nodes. Intuitively, we use the vectorized code statements to denote the features of the CFG nodes. In Figure 5(a), the box "statements" above "Func" represents the collection of the code statements of "Func". Doc2Vec [34] converts the statements into their corresponding vectors. Let $\mathbf{H}^0 = \{\mathbf{h}_1, \mathbf{h}_2, \cdots, \mathbf{h}_N\}$ denote the original features, i.e., a set of vectors by generated by Doc2Vec. We then use the GCNs to compute the representation of the nodes on a CFG. The convolutional operation follows Equation 17:

$$\mathbf{H}^{(l+1)} = f(\mathbf{H}^{(l)}, \mathbf{A}) = \sigma(D^{-1}\tilde{\mathbf{A}}\mathbf{H}^{(l)}\mathbf{W}^{(l)}), \qquad (17)$$

where $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ represents the adjacency matrix of the graph G with added self-connections. $\mathbf{I}_N$ is the identity matrix, $\tilde{D}_{ii} = \sum_j \tilde{\mathbf{A}}_{ij}$ represents the diagonal matrix, and $\mathbf{W}^{(l)}$ is a trainable weight matrix. We use the $\sigma(\cdot)$ to denote the activation function, $\mathbf{H}^{(l)} = \{\mathbf{h}'_1, \mathbf{h}'_2, \cdots, \mathbf{h}'_N\}$ to denote the matrix of activation in the l^{th} layer [33], [35]. We propose to train the GCN to learn the distributed representation of the graph G for its structured information.

As illustrated in Figure 5(b), in the GCN, at least one learnable linear transformation layer (i.e., the convolutional layer) is required to transform the input features together with the graph structure into the aggregated graph representations. The goal is to use the GCNs to learn the structure information from the CFGs following: $\mathbf{H}^l = ConvLayers(\mathbf{H}^0, \mathbf{A})$ (Equation 17), which accepts $\mathbf{H}^0 \in \mathbb{R}^{N\times F}$ where the t-th row, $\mathbf{h}_i$ is the vector to represent node v_i by Doc2Vec [36], [34]. F is the feature dimension of each node in the latent embedding vector. $\mathbf{A}$ is the adjacency matrix representing the control-flows to model its graph structure. The function $ConvLayers$ produces a node-level output matrix $\mathbf{H}^l \in \mathbb{R}^{N\times F'}$, which is an output feature matrix for all the nodes on G using the $(l+1)$-layer convolutional neural network (l starts from 0). In our experiment, we used a single-layer GCN. Graph-level outputs can be obtained by using pooling operation [37].

B. Attentions and Feature Fusion

Since code in the form of tokens, ASTs and CFGs may affect the final accuracy of clone detection at lexical, syntactical and structure levels differently, we apply the attention mechanisms individually to the above representations. This multimodal approach can capture important information in different code representations. We first introduce the attention mechanism to model source code at the token (or text) and AST levels. Then, we describe the graph attention model to embed CFGs. First, we leverage the attention mechanism of processing documents [21], [31] to produce the embedding vector v^{txt} for modeling code tokens. An input code fragment consists of the tokens $\mathbf{x} = [x_1, x_2, \ldots, x_N]$, where the length of the code fragment is N. After the input sequence has been processed through the LSTMs, $\mathbf{h}_t$ computed by Equation 10 represents the hidden vector for x_t. The attention mechanism uses the following three equations:

$$\mathbf{u}_t^{txt} = \tanh(\mathbf{W}_c^{txt}\mathbf{h}_t^{txt} + \mathbf{b}_c^{txt}), \qquad (18)$$

$$\alpha_t^{txt} = p(x_t|\mathbf{x}, x_c) = \frac{\exp(\mathbf{u}_t^{txt\top} \cdot \mathbf{u}_c^{txt})}{\sum_{i=1}^{N} \exp(\mathbf{u}_i^{txt\top} \cdot \mathbf{u}_c^{txt})}, \qquad (19)$$

$$\mathbf{v}^{txt} = \sum_{t=1}^{N} \alpha_t^{txt} \cdot \mathbf{h}_t^{txt}. \qquad (20)$$

In Equation 18, an one-layer MLP (MultiLayer Perceptron) (i.e., $tanh$) is used to retrieve $\mathbf{u}_t^{txt}$ as a hidden representation of $\mathbf{h}_t^{txt}$. $\mathbf{W}_c^{txt}$ is a shared weight matrix, $\mathbf{b}_c^{txt}$ is a bias vector. Both are randomly initialized and are trained as parameters. Equation 19 calculates a normalized importance weight value through a softmax expression for t-th token, denoted as α_t^{txt}. $\mathbf{u}_c^{txt}$ is a context vector. Note that $\mathbf{u}_c^{txt}$ is randomly initialized and jointly learned during the model training process. There are a few options to implement the similarity measure function $f(\cdot)$ in Equation 2 [38], [18]. To mitigate the computational load of the weights on the networks, we choose the dot-product to calculate the similarity scores, i,e., $\mathbf{u}_t^{txt\top} \cdot \mathbf{u}_c^{txt}$. Equation 20 computes the v^{txt} for the code sequence, where is the weighted representation of the target code fragment (i.e., a weighted sum of all the vectors based on their weights).

Similarly, the representation at the syntactical level is processed with the following equations:

$$\mathbf{u}_t^{ast} = \tanh(\mathbf{W}_c^{ast}\mathbf{h}_t^{ast} + \mathbf{b}_c^{ast}), \qquad (21)$$

$$\alpha_t^{ast} = \frac{\exp(\mathbf{u}_t^{ast\top} \cdot \mathbf{u}_c^{ast})}{\sum_{i=1}^{N'} \exp(\mathbf{u}_i^{ast\top} \cdot \mathbf{u}_c^{ast})}, \tag{22}$$

$$\mathbf{v}^{ast} = \sum_{t=1}^{N'} \alpha_t^{ast} \cdot \mathbf{h}_t^{ast}, \tag{23}$$

where N' is number of nodes on an AST of the target code, which can be different from the number of tokens of the code. $\mathbf{h}_t^{ast}$ in Equation 21 is computed from the output of hidden representation in Equation 16.

For the attention mechanism when embedding CFGs, we apply the graph attention networks [39] to produce the latent vector for each node by considering the importance of its neighborhood nodes. We use e_{ti} to indicate the importance of node i's features to node t. We then normalize them across all choices of i using a softmax function.

$$\begin{aligned} e_{ti} &= a(\mathbf{W}\mathbf{h}_t, \mathbf{W}\mathbf{h}_i), \\ \alpha_{ti} &= \frac{\exp(e_{ti})}{\sum_{k \in N_t} \exp(e_{tk})}, \end{aligned} \tag{24}$$

where the attention function a is a single-layer feedforward neural network [39]. $\mathbf{W}$ is a layer-specific weight matrix and applied to every node on the CFG. Note that we only compute e_{ti} for nodes $i \in \mathcal{N}_t$, where $\mathcal{N}_t$ is the neighborhood of node t (including t) on the CFG.

We can then obtain $\mathbf{h}'_t$, which denotes the structured representation of a CFG processed by the convolutional layers and an activation function σ, which is $ReLU$, using the following equation:

$$\mathbf{h}'_t = \sigma(\sum_{i \in N_{(t)}} \alpha_{ti} \mathbf{W}\mathbf{h}_i). \tag{25}$$

All the embedding vectors of the nodes on a CFG are iteratively computed by GCN. We assume every function has one exit node $\mathbf{h}_{exit}$. We choose $\mathbf{h}'_{exit}$ as the final embedding vector of the CFG, i.e., $\mathbf{v}^{cfg} = \mathbf{h}'_{exit}$.

Once the vectors for the three different code representations are generated, they are concatenated to produce a hybrid representation. The hybrid representation is then fed into an one-layer linear network, which can be denoted by the following equation:

$$\mathbf{v} = \mathbf{W}_{dt}[\mathbf{v}^{txt}; \mathbf{v}^{ast}; \mathbf{v}^{cfg}] + \mathbf{b}_{dt}, \tag{26}$$

where $\mathbf{v}$ denotes the final hybrid code representation. $\mathbf{W}_{d_t}$ balances the composition of the textual, syntactic and the structural features in the resulting hybrid representation, and $\mathbf{b}_{d_t}$ turns the model a bias towards final convergence when training the model.

C. Model Learning

Given a set of code pairs and their corresponding clone types $\mathcal{D} = \{((a_1, b_1), y_1), \ldots ((a_i, b_i), y_i) \ldots ((a_N, b_N), y_N)\}$,, where a_i and b_i denote a pair of code fragments, y_i denotes the labeled clone type of a code pair, i.e., $y_i \in \mathcal{C} = \{C_0, C_1, C_2, C_3, C_4\}$, where C_0 denotes non-clone and C_1, C_2, C_3, C_4 denote the four clone types labeled in BigCloneBench. The similarity measurement function sim calculates Manhattan distance [40] between two hybrid code representations by using the following equation:

$$x_{a,b} = sim(a, b) = \exp(-||\mathbf{v}^a - \mathbf{v}^b||_1) \ \in (0, 1), \tag{27}$$

where $\mathbf{v}^a$ and $\mathbf{v}^b$ are calculated by Equation 26 for a and b respectively. Since the set of distances $||\mathbf{v}^a - \mathbf{v}^b||_1$ are in the range $(0, \infty)$, an exponential function $\exp(\cdot)$ is used to normalize the distance values into an interval from 0 to 1 exclusively, where $x_{a,b} = 0$ means the two code fragments a and b are exactly the same.

We then conduct the final training to build a model to classify code pairs into five types (a non-clone and four clone types/classes). We utilize a *Softmax Regression* [41] (i.e., the $softmax$), an extension of logistic regression [42] for training a multi-classifier. We have a training set $\{(x_{a_1,b_1}, y_1), (x_{a_2,b_2}, y_2) \ldots (x_{a_i,b_i}, y_i) \ldots (x_{a_N,b_N}, y_N)\}$, where x_{a_i,b_i} denotes the manhattan distance between code fragments a_i and b_i computed by Equation 27.

Given a manhattan distance x as an input, our multi-class classification hypothesis $h_\theta(x)$, which estimates the probability $P(y = C_i|x)$ of each clone type (i.e., $C_0 \ldots C_4$), and then produces a 5-dimensional vector (whose elements sum to 1) for input x, indicates its probabilities of the five classes:

$$h_\theta(x) = \begin{bmatrix} P(y = C_0|x;\theta) \\ P(y = C_1|x;\theta) \\ P(y = C_2|x;\theta) \\ P(y = C_3|x;\theta) \\ P(y = C_4|x;\theta) \end{bmatrix} = \frac{1}{\sum_{k=0}^{4} e^{\theta^{(k)T}x}} \begin{bmatrix} e^{\theta^{(C_0)\top}x} \\ e^{\theta^{(C_1)\top}x} \\ e^{\theta^{(C_2)\top}x} \\ e^{\theta^{(C_3)\top}x} \\ e^{\theta^{(C_4)\top}x} \end{bmatrix}, \tag{28}$$

where $\theta = [\theta^{(C_0)}; \theta^{(C_1)}; \theta^{(C_2)}, \theta^{(C_3)}; \theta^{(C_4)}]$ are our model's parameters, which are learned from model training by minimizing the following cost function:

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \sum_{k=0}^{4} \mathbb{1}\{y_i = C_k\} \log P(y = C_k|x;\theta), \tag{29}$$

where $\mathbb{1}\{\cdot\}$ denotes the indicator function, i.e., $\mathbb{1}\{\text{a true statement}\} = 1$ and $\mathbb{1}\{\text{a false statement}\} = 0$. $P(y = C|x;\theta)$ is the calculation for the probability of each clone type under the input x. The hypothesis estimation for a $softmax$ is defined as follows:

$$P(y_i = C_k|x;\theta) = \frac{\exp(\theta^{(C_k)\top}x)}{\sum_{k'=1}^{4} \exp(\theta^{(C'_k)\top}x)}. \tag{30}$$

IV. Experiments

In this section, we first describe the dataset used in our experiments and then we introduce the implementation details and evaluation methods. Furthermore, we discuss and analyze the experiment results, aiming at answering the following three research questions:

RQ1. How does FCCA perform in comparison to the state-of-the-art approaches?

RQ2. How do different representation settings impact the effectiveness of FCCA when detecting functional code clones?

RQ3. How do the attention networks used in FCCA (plus the different code features) contribute to the accuracy of the final results?

A. Feature Engineering: Pre-process & Feature Extraction

Our model focuses on the task of clone detection at the method level. All the available Java files in IJaDataset2.0 for successful compilation are listed in our dataset, which provides a database consisting of well-labeled clone pairs. To construct ASTs of the Java files, we used `Antlr` (version 4), which is a tool for code analysis suitable for multiple programming languages [43]. In addition, `Antlr` also provides APIs to parse source code into tokens. We used the built-in visitor APIs from `Antlr` to traverse the AST for each Java method. We also employed Soot [44], a widely used static code analysis framework to extract the CFG of each Java method for compilable Java files. After CFGs created, we built the feature vector of each CFG node. The feature vectors of CFG nodes were generated by using Doc2Vec [36], a commonly used text embedding technique. Source files that are not compilable were discarded. Methods containing equal or fewer than five lines of code were also ignored, because they were considered trivial methods. The resulting ASTs and CFGs were stored in Dot files for post-processing and data analytics.

B. Data Collection

We conducted our experiments using the clone pairs from `IJaDataset2.0`[1] [45], which has been widely used for evaluating code clone detectors. It contains manually-labeled method-level clone pairs. In our experiments, we mainly focus on functional clones (i.e., Weakly Type-III and Type IV as described in Section II-A). Building CFGs requires programs to be compiled into Java `.class` files. However, some programs in the IJaDataset2.0 are incomplete. Thus, their corresponding CFGs cannot be obtained. We compiled as many source files as possible to generate their corresponding CFGs by using Soot [44]. Finally, we collected 6,351 compilable Java files, 275,777 method-level clone pairs and 269,032 non-clone pairs. It is worth mentioning that the non-clone pairs and the clone pairs in IJaDataset2.0 are all labeled by domain experts [13], [10]. In addition, we also manually validated 100 samples in negative samples to ensure that the non-clone pairs were obtained in a fair manner. Table II shows an overview of the dataset. Note that the percentage value (%) is the proportion of each clone type among all clone types. In our experiments, we randomly split the dataset into 6:2:2 (60% for model training, 20% for validation and the remaining for testing). 4,000 code pairs were randomly selected for validation to avoid overfitting during training. To handle the imbalanced data, we used the standard SMOTE method to balance the samples for Type-I,Type-II, and Type-III clone pairs[2].

C. Implementation Details

All experiments were conducted on a server with 2.2 GHz Intel Core i7 CPU, 64GB memory, and a NVIDIA P6000 GPU, running Red Hat 7.0.

TABLE II: Overall information for the dataset (clone pairs)

Clone Type	wise-pairs	%
Type-I	7,757	2.81
Type-II	3,301	1.12
MType-III	7,390	2.67
SType-III	4,279	1.56
Type-IV	253,050	91.75

The neural network model has a 128-dimensional hidden state $\mathbf{h}_t$ of each LSTM layer to handle code features (i.e., the code tokens). Each layer of the neural network contains memory cells C_t where each cell C is qualified with a time stamp t to indicate its position in this cell sequence. The cells in LSTM contain the sequential information of each feature equipped with the attention mechanism. The weights in the network were initialized with the small random Gaussian distribution which is widely used in many deep-learning models [48], [40]. We employed `Word2Vec`[3] to tokenize the code fragments.We adopted Antlr4[4] to parse the source code to ASTs first, and traversed the ASTs via the APIs provided by Antlr4. We eventually collected a dictionary at word (token) level by 58,426 tokens. We set up the exception process of the data scope that they were marked as "unknown" and initialized with randomly generated vectors of the tokens were out of vocabulary. For a quick experiment setup, we refer to hyper-parameters used in Siamese Network model as the initial settings [40].

We applied a 5-fold cross-validation to train and evaluate our model. The collected dataset was partitioned into 5 subsets, where we picked 1 subset as the testing set each time and the remaining 4 subsets were used as the training set. Then, we repeated the experiments for 5 times and each time we used a different subset. Note that the experiments were all conducted for the clones at the method-level to ensure that the comparison between our tool and the baselines is fair.

D. Comparison Methods

We selected the following six state-of-the-art baselines to validate the effectiveness of our proposed FCCA:

- DECKARD [22] is a classical AST-based detector that generates characteristic vectors for each AST of a program using predefined rules. We used the experiment results reported in [13], [47] for reference.
- DLC [14] is a recursive-neural-network-based detector that measures code similarity using Euclidean distance. We used the experiment results from the papers [14], [47] for reference.
- SOURCERERCC [10] is a bag-of-words-based clone detection tool. The results were generated using their open-source tool.
- CDLH [13] is a deep-learning-based clone detection approach that uses ASTs as representations of code features.

[1] https://github.com/clonebench/BigCloneBench

[2] https://imbalanced-learn.readthedocs.io/en/stable/over_sampling.html

[3] http://radimrehurek.com/gensim/models/`Word2Vec`.html

[4] https://github.com/antlr/antlr4/blob/master/doc/index.md

TABLE III: Parameter settings for the different tools.

Tool	Parameters
DECKARD [22]	Min tokens: 100, Stride: 2, Similarity,threshold: 0.9
DLC [14]	Hidden layer size: 400, epoch: 25, Initial learning rate: 0.003, clipping gradient range: (-5.0, 5.0), λ for L2 regularization: 0.005
SOURCERERCC [10]	Min length 6 lines, similarity 70%, function granularity.
CDLH [13]	Code length 32 for learned binary hash codes, word embeddings of length 100
TBCNN [46]	Convolutional layer dim size: 300, Dropout rate: 0.5, batch size of 10.
DEEPSIM [47]	Layers size: 88-6, (128x6-256-64)-128-32, epoch: 4, Initial learning rate: 0.001,λ for L2 regularization: 0.00003, Dropout: 0.75
Plain Text	Size of hidden states: 128, Embedding size: 300 epoch: 50, Initial learning rate: 0.001, batchsize: 32
AST	Size of hidden states: 128, Embedding size: 300, epoch: 50, Initial learning rate: 0.001, batchsize: 32
CFG	Embedding size: 64 epoch: 50, Initial learning rate: 0.001, batchsize: 16
Plain Text + ATTN	Size of hidden states: 128, Embedding size: 300 epoch: 50, Clipping gradient range: (-1.2, 1.2), initial learning rate: 0.001, batchsize: 32
AST + ATTN	Size of hidden state: 128, Embedding size: 300 epoch: 50, Clipping gradient range: (-1.2, 1.2) initial learning rate: 0.001, batchsize: 32
CFG + ATTN	Epoch: 50, Initial learning rate: 0.001, batchsize: 32
FCCA	Size of hidden states: 128 (Text), 128 (AST), Embedding size: 300 (Text), 300 (AST), 64 (CFG) Clipping gradient range: (-1.2, 1.2) epoch: 50, Initial learning rate: 0.0005, Dropout:0.6, batchsize: 32

Since the authors do not provide detailed experimental settings and their open-source implementation, we used the experiment results reported in their paper. In fact, this approach is a subset/instance of FCCA. We implemented CDLH by using FCCA when only ASTs features were considered.

- TBCNN [46] is a Tree-based CNN model which adopts convolutional networks to process ASTs. We have also compared FCCA with TBCNN (implemented by ourselves) to evaluate the performance between the LSTM-based and CNN-based methods.
- DEEPSIM [47] is a recent deep-learning-based approach of clone detection, which uses a semantic matrix that encodes control- and data-flow information of source code. We cannot reproduce the results when analyzing BigCloneBench using their online tool due to configuration problems. Thus, we refer to the results from their paper.

Precision (P), recall (R), and F1 scores were used as the evaluation metrics. The configurations of all the methods are provided in Table III. To evaluate the performance of different clone detectors, we adopted three metrics, i.e., Precision (P), recall (R) and F1 scores, which has been widely adopted in the evaluation of classification tasks.

E. Execution Time and Scalability

TABLE IV: Time performance.

Method	Prediction time	Training time
SOURCERERCC	42s	-
CDLH	90s	45,317s
TBCNN	86s	41,168s
FCCA	91s	46,769s

We evaluated the execution time of the baseline approaches with the same dataset as that used in FCCA. We compared our approach with SOURCERERCC, CDLH, and TBCNN for execution time by running the corresponding implementations. For DECKARD, DLC and DEEPSIM, we directly used their data reported in their papers. As all the tools need to extract the tokens and ASTs before the prediction processes, we excluded the pre-process time. Table IV reports the time performance of the tools. SOURCERERCC which uses the non-learning-based approach to detect the clone pairs, required less time to find clones [10]. Further, the tool which is not a learning-based approach does not need a training process. As mentioned before, we resorted to a variant of FCCA which only considers the AST feature to approximate the time cost for CDLH. TBCNN was efficient compared to the other two baselines using only 41,168s for training. FCCA had more computational cost since it required to fuse the three types of code representations. It took 936s in the training for each epoch and approximately 13 hours (46,769s) to complete the model training with 275K code clone pairs and 269K non-clone pairs.

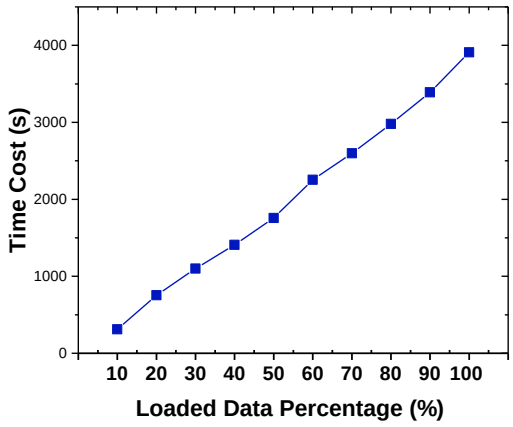


Fig. 6: Scalability analysis by loaded data percentage.

Scalability. To evaluate the scalability of our proposed approach, we randomly selected 50K code pairs from BigCloneBench and calculated its running time. Figure 6 shows the result when evaluating FCCA's scalability. We can observe that FCCA successfully scaled to our large dataset, with a proportionally increased training time regarding the percentage of the loaded data size.

F. Results and Analysis

TABLE V: Precision, recall and F1 results when comparing FCCA with the existing clone detectors. The best scores are highlighted in boldface.

Method	Precision	Recall	F1
DECKARD [22]	0.93	0.02	0.03
DLC [14]	0.95	0.01	0.01
SOURCERERCC [10]	0.88	0.02	0.03
CDLH (AST) [13]	0.92	0.74	0.82
TBCNN [46]	0.90	0.81	0.85
DEEPSIM [47]	0.97	0.98	**0.98**
FCCA	**0.98**	0.97	**0.98**

TABLE VI: F1 values with respect to the different clone types. The best scores are highlighted in boldface.

Method	T1	T2	ST3	MT3	WT3/T4
DECKARD [22]	0.73	0.71	0.54	0.21	0.02
DLC [14]	1.0	0.97	0.60	0.03	0.00
SOURCERERCC [10]	0.94	0.93	0.77	0.1	0.00
CDLH [13]	1.0	1.0	0.94	0.88	0.82
TBCNN [46]	1.0	1.0	0.93	0.80	0.86
DEEPSIM [47]	0.99	0.99	**0.99**	**0.99**	0.97
FCCA	**1.0**	**1.0**	0.95	0.97	**0.98**

***RQ1: How does FCCA perform in comparison to the six selected state-of-the-art approaches*?** Table V shows the comparison in terms of precision, recall and F1 scores. Table VI shows a detailed comparison of the F1 scores for each clone type (T1, T2, ST3, MT3, WT/T4) between FCCA and the six other tools. The results of DECKARD, DLC and DEEPSIM correspond to those reported in their papers [22], [14], [13]. The experimental results indicate that FCCA outperforms all the other approaches in terms of precision. As shown in Table VI, FCCA also performed better than all the other approaches for the most complex clone types WT3 and T4. We discuss the results of the six tools below:

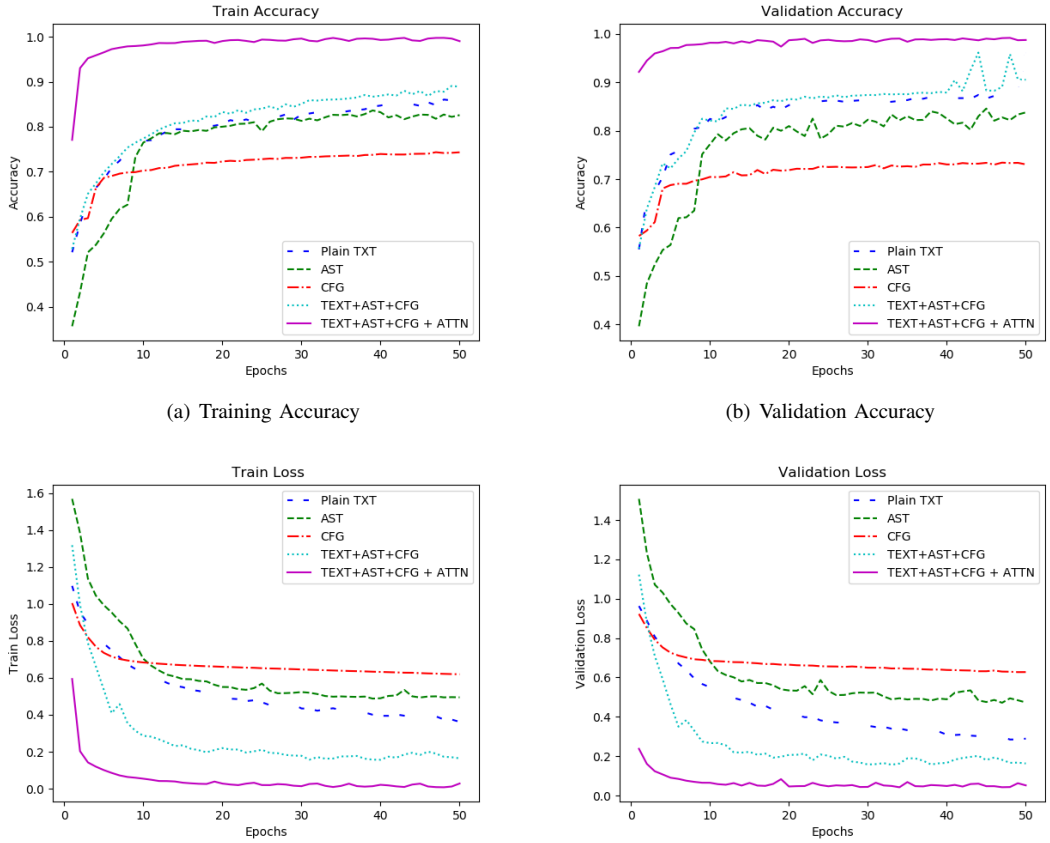
- DECKARD had relatively low F1 scores on the Type-MT3 and Type WT3/T4 clones with 0.21 and 0.02, respectively (Table VI). The low score is because functional code clones account for a large portion of the experimental dataset.
- DLC had a higher precision (0.95) than DECKARD, However, it also had a lower recall (0.01) and an F1 score (0.01), meaning that using latent features extracted with deep learning techniques without considering structured feature failed to detect functional code clones.
- SOURCERERCC performed slightly better than DECKARD and DLC. However, this single-feature token-based clone detector was imprecise when detecting WT3/T4 clones (Table VI).
- CDLH [13] has an F1 score of 0.82 (Table V). All scores were better than the three above competitors given the AST-based code representation at the syntactic level. However, the syntactic-level single-feature representation resulted in a low recall rate (0.74), which indicated some important structural information may have been missed.
- TBCNN achieved the F1 score of 0.85. We can observe that, as a tree-based CNN, TBCNN performed better than CDLH, whereas did not outperform FCCA. Based on our investigation, it is mainly because the convolutional layers ignore the long distance contextual information between two elements (tokens or nodes on AST) of the source code at the syntactical level, whereas the contextual information is the influential feature to the prediction results. On the contrary, benefiting from the attention mechanism, LSTM-based approach outperforms the TBCNN-based approach by 0.98 (to TBCNN's 0.86) in terms of the F1 score.
- DEEPSIM is a recent clone detector that leverages structured information, including control- and data-flows of a program. As shown in Table V, FCCA achieved overall better precision than that by DEEPSIM and the same F1 score. In terms of the clone types, Table VI shows FCCA with a better F1 score than DEEPSIM on the most complicated clone types WT3/T4, because of FCCA's comprehensive hybrid code representation with attentions.

TABLE VII: The effectiveness of individual features and their combinations. The best scores are highlighted in boldface.

Feature Name	Precision	Recall	F1
PLAIN TEXT	0.83	0.81	0.82
AST	0.84	0.80	0.81
CFG	0.66	0.71	0.68
TEXT+AST+CFG	0.86	0.85	0.86
TEXT+AST+CFG + ATTN (FCCA)	**0.98**	**0.97**	**0.98**

RQ2: How do different representation settings impact the effectiveness of* FCCA *when detecting functional code clones?

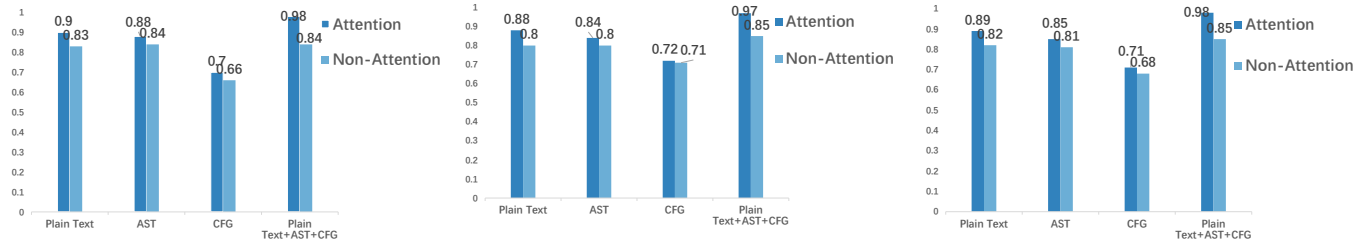
Table VII shows the results when using a single feature and combinations of features. A cursory observation reveals that the model based on individual features had relatively lower precision, recall and F1 scores. The plain text-based model performed almost as well as the AST-based model. This is mostly because both models used the same tokens in the embedding layer. The model based on AST alone was able to identify similar code fragments at the syntactic level, even



(a) Training Accuracy (b) Validation Accuracy

(c) Training Loss (d) Validation Loss

Fig. 7: Comparisons of accuracy and loss during training and validation for each feature.



(a) Comparison of Precision (b) Comparison of Recalls (c) Comparison of F1 score

Fig. 8: Comparisons of different feature with the attention mechanism for Precision, Recall and F1 score.

with very different tokens that shared little lexical information. However, AST representations contained more syntactical information, which was likely responsible for the slightly better performance. Compared to plain text and AST, CFG alone achieved relatively low scores. When fused with the text and AST features, it boosted the FCCA 's overall efficacy. In terms of the F1 scores, the relative improvements were 4% (text), 5% (AST) and 18% (CFG). These improvements demonstrated how low-level texts, ASTs and high-level control/flow information complement each other to reveal relatively complicated functional clones. However, the improvement with these two types of features capped at 5%. The attention mechanism is responsible for the remainder of the improvement. Figure 7 illustrates FCCA's training and validation phase. Figure 7(a) shows that the initial accuracy of FCCA with a single feature was higher than all the other models at the beginning of the training phase, which demonstrates that fusing the features helps the model capture the comprehensive unstructured and structured code information. Figure 7(b) shows the consistency of the results in the validation stage. Figure 7(c) depicts the training loss during the training epochs for each model, and Figure 7(d) shows the scenarios for validation loss. Overall, FCCA achieved the best results and the charts demonstrate that the attention mechanism indeed pay attention to important aspects of the code features to improve the performance of code clone detection.

RQ3: The effectiveness of the attention networks and the contributions of different code features to the final accuracy. The results of the experiments so far show that incorporating attention mechanism into the learning framework resulted in an overall improvement in FCCA 's performance. However, to analyze this impact in more detail, we conducted tests with each type of representation both with and without the attention mechanism placing weights on each token and without. The results in Figure 8 show the across-the-board improvements with the attention mechanism with all types of representations. Without even fully tuning the hyper-parameters, there was up to a 13% improvement in F1 score. It was mainly because all the three features improve the performance respectively then the improvements were summed up together to make the overall performance of FCCA significantly increase up to 0.98 on the F1 score.

According to these evaluations, FCCA's ability to detect complicated code clones, particularly Type-III and Type-IV clones, showed superior to the six state-of-the-art tools in terms of both accuracy and recall.

V. Discussion

A. Strength of FCCA

We have identified three advantages of FCCA that may explain its effectiveness in code clone detection: (a) FCCA represents the source code snippets from its multiple modalities (i.e., tokens, AST and CFG), which contain complementary information for the final code representation. (b) Equipped with an attention mechanism, FCCA can capture the important parts from different code representations for each modality, as to boost the performance of code clone detection when comparing with the other baselines in this paper. (c) Our proposed approach is an unified framework to learn the hybrid representation of source code for detecting functional clones.

B. Threats to Validity and Limitations

There are four main threats to the validity. First, we used previously published results in several baselines (e.g., DLC, CDLH and DEEPSIM). This is because the experiment settings of DLC and CDLH are not clearly introduced. It is also unclear to us about the configurations, particularly the data format, to run the experiment for DEEPSIM. However, directly using the reported data from the existing papers does not allow for a more fine-grained qualitative comparison, for example, counting and analyzing each clone fragment between FCCA and others.

Second, it is true that the experiment results are sensitive to the configurations of the tools correspondingly. This paper, however, does not focus on hyper-parameter tuning, which is an orthogonal source of precision. Nevertheless, we have carefully listed the parameters used in our approach and reported the configurations of the other tools in Table III to allow subsequent researchers to reproduce the experiments in the future.

The third threat is that the granularity of the clone detection can be different (e.g., DECKARD [22] finds clone fragments in code block level, as opposed to finding clones at the method level). Because our model partially works on top of the compiled Java code to obtain the complete control-flow graph of a program, currently it is not able to analyze incomplete source code which is not compilable.

The last threat lies in the training time. As we conducted the experiments on the large-scale corpus, which contains over 250 million lines of code (MLOC) [10], the training time is indeed very long to produce a precise hybrid representation. It is still a big challenge to optimize the model to shorten the training time.

VI. Related Work

In this section, we briefly review the related studies from three perspectives, namely deep code representation, attention mechanism and code clone detection.

A. Deep Code Representation

With the successful development of deep learning, many researchers have developed diverse code representations for source code. In [46], Mou et al. use a tree-structured convolutional neural network (Tree-CNN) to learn the distributed vector representations from code snippets for program classification. Similarly, Wan et al. [32], [49], [50] apply the tree-structured recurrent neural network (Tree-LSTM) to extract the important information from ASTs for the task of code summarization. Inspired by this approach, we consider using the graph representation together with tokens and ASTs to build a comprehensive hybrid code representation. Xiao et al. [51] apply graph embedding to perform static detection of a wide variety of vulnerabilities. This approach provides a solution to detection of control-flow-related software vulnerabilities using CFGs as the major code representation. Zhang et al. [52] present a hybrid representation learning for familial clustering of weakly-labeled Android malware by preserving heterogeneous information from multiple sources, including the results of static code analysis, the meta-information of an Android app, and the raw-labels of antivirus engines.

B. Attention Mechanism

Attention mechanism has shown remarkable success in many artificial intelligence domains such as neural machine translation [18], image captioning [53], image classification [54] and visual question answering [55]. The attention mechanisms can enhance the original models to capture the necessary parts of visual or textual inputs. Visual attention models selectively pay attention to small regions in an image to extract important features and reduce the size of information to process. Some methods have recently adopted the visual attention to improve models for image classification [56], [57], image generation [58], image captioning [59], visual question answering [60], [61], [62], etc. Attention mechanisms are also helpful for NLP tasks to find semantic or syntactic input-output alignments under the encoder-decoder framework, which can effectively process the long-term dependency in texts. These solutions have been successfully applied to many NLP tasks

including machine translation [18], text generation [63], sentence summarization [32], [38] and question answering [64]. Nam et al. propose a multi-stage co-attention learning framework to refine the attentions based on memory of previous attentions. In [65], Paulus et al. combine the inter- and intra-attention mechanism in a deep reinforcement learning model to improve the performance for the task of abstractive text summarization. In [66], Zhang et al. incorporate a self-attention mechanism into convolutional generative adversarial networks to improve the performance.

C. Code Clone Detection

Code clone detection is a classical task in software engineering. Many tools have been developed in the past few years. NICAD [12] is a lightweight, language-specific parser-based clone detector for detecting clones Type I-III, which only considers the textual features of source code. Rather than string matching, the proposed detector transforms the problem of code clone detection into a statistical clustering problem. However, the tool is unable to effectively detect functional clones. DECKARD [22] captures the code's structural information via the ASTs of source code by using a statistical model to cluster the codes. The approach is not limited at the method level, but can detect clones for code fragments.

SOURCERERCC [10] is a tool that detects clones using a fast bag-of-tokens model. It has shown good detection results for Type-III clones, which is also a baseline in this paper. However, since the bag-of-tokens model discards sequential and structural information, it suffers from low precision and accuracy when detecting Type-IV clones. In this paper, we have conducted the comparison with SOURCERERCC at the method level by running its tool. Our model produces better result than SOURCERERCC for functional clones.

DLC [14] introduces an approach based on a language model, in which, clone detection is a recursive learning procedure designed to adequately represent fragments that serve as constituents of higher-order components. DLC leverages deep learning during the source code pre-processing step.

CDLH [13] is an end-to-end learning model for code clone detection. The model learns hash functions, structure information, and representations of code fragments using an AST-based LSTM by taking both lexical and syntactical code features into account. CDLH is basically a subset of our approach and was evaluated as also outlined in Section IV.

DEEPSIM [47] is a recent tool that encodes both control- and data-flows into a compact semantic feature matrix to identify functional code clones. The tool uses a feed-forward neural network to learn the similarity from the semantic feature matrices between two code snippets. Compared to DEEPSIM, FCCA offers a more straightforward methodology for feature fusion. When combined with an attention mechanism, FCCA yields better precision than DEEPSIM's results for functional clone detection.

OREO [67] stacks several LSTM layers with good performance results reported from a 4-layer deep-learning model, containing 200 units in each layer. With this method, the approach was able to identify a special type of clones (so-called Twilight Zone) between clone Types-III and IV. However, as also mentioned in their paper, OREO failed to achieve good performance when detecting clones of Type-IV.

VII. CONCLUSION

This paper proposes FCCA, a new code clone detector based on a hybrid code representation that preserves heterogeneous code features in a compact low-dimensional latent space. Equipped with an attention mechanism, FCCA analyzes each token under different code features and then pays attention to the tokens that make the important contributions to final detection accuracy. The result is an overall performance improvement in deep-learning-based clone detection. Extensive experiments were conducted with 275,777 real-world clone pairs. The resulting precision, recall, and F1 scores show that FCCA outperforms several state-of-the-art approaches.

VIII. ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their helpful comments. This research is supported by National Natural Science Foundation of China (Grant No. 61672338 and 61373028) and partially supported by the Australian Research Council (Grant No. DP200101374, LP170100891 and DP200101328)

REFERENCES

[1] Q. Tu *et al.*, "Evolution in open source software: A case study," in *Proceedings 2000 International Conference on Software Maintenance*. IEEE, 2000, pp. 131–142.

[2] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 368–377.

[4] M. Suzuki, A. C. de Paula, E. Guerra, C. V. Lopes, and O. A. L. Lemos, "An exploratory study of functional redundancy in code repositories," in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 31–40.

[5] B. Muddu, A. Asadullah, and V. Bhat, "Cpdp: A robust technique for plagiarism detection in source code," in *2013 7th International Workshop on Software Clones (IWSC)*. IEEE, 2013, pp. 39–45.

[6] G. Xiao, Z. Zheng, and H. Wang, "Evolution of linux operating system network," *Physica A: Statistical Mechanics and its Applications*, vol. 466, pp. 249–258, 2017.

[7] M. Fowler and M. Foemmel, "Continuous integration," *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf*, vol. 122, p. 14, 2006.

[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*. IEEE, 2009, pp. 485–495.

[9] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[10] 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.

[11] 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.

[12] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*. IEEE, 2008, pp. 172–181.

[13] H. Wei and M. Li, "Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code." in *IJCAI*, 2017, pp. 3034–3040.

[14] 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.

[15] S. Pan, J. Wu, X. Zhu, C. Zhang, and Y. Wang, "Tri-party deep network representation," in *AAAI*, 2016, pp. 1895–1901.

[16] C. Shi, B. Hu, W. X. Zhao, and S. Y. Philip, "Heterogeneous information network embedding for recommendation," *TKDE*, vol. 31, no. 2, pp. 357–370, 2019.

[17] Y.-H. H. Tsai, P. P. Liang, A. Zadeh, L.-P. Morency, and R. Salakhutdinov, "Learning factorized multimodal representations," in *ICLR*, 2019.

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

[19] O. Levy, M. Seo, E. Choi, and L. Zettlemoyer, "Zero-shot relation extraction via reading comprehension," *arXiv preprint arXiv:1706.04115*, 2017.

[20] A. Show, "Tell: Neural image caption generation with visual attention," *Kelvin Xu et. al.. arXiv Pre-Print*, vol. 23, 2015.

[21] Z. Yang, D. Yang, C. Dyer, X. He, A. Smola, and E. Hovy, "Hierarchical attention networks for document classification," in *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2016, pp. 1480–1489.

[22] 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*. IEEE Computer Society, 2007, pp. 96–105.

[23] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, p. 40, 2019.

[24] J. Svajlenko and C. K. Roy, "Bigcloneeval: A clone detection tool evaluation framework with bigclonebench," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 596–600.

[25] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, 2007.

[26] J. Yousefi, Y. Sedaghat, and M. Rezaee, "Masking wrong-successor control flow errors employing data redundancy," in *2015 5th International Conference on Computer and Knowledge Engineering (ICCKE)*. IEEE, 2015, pp. 201–205.

[27] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 321–330.

[28] F. E. Allen, "Control flow analysis," in *ACM Sigplan Notices*, vol. 5, no. 7. ACM, 1970, pp. 1–19.

[29] T. Shen, T. Zhou, G. Long, J. Jiang, S. Pan, and C. Zhang, "Disan: Directional self-attention network for rnn/cnn-free language understanding," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[30] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[31] Y. Wang, M. Huang, L. Zhao *et al.*, "Attention-based lstm for aspect-level sentiment classification," in *Proceedings of the 2016 conference on empirical methods in natural language processing*, 2016, pp. 606–615.

[32] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 397–407.

[33] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[34] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[35] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.

[36] A. Narayanan, M. Chandramohan, L. Chen, Y. Liu, and S. Saminathan, "subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs," *arXiv preprint arXiv:1606.08928*, 2016.

[37] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams, "Convolutional networks on graphs for learning molecular fingerprints," in *Advances in neural information processing systems*, 2015, pp. 2224–2232.

[38] A. M. Rush, S. Chopra, and J. Weston, "A neural attention model for abstractive sentence summarization," *arXiv preprint arXiv:1509.00685*, 2015.

[39] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

[40] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity." in *AAAI*, vol. 16, 2016, pp. 2786–2792.

[41] http://deeplearning.stanford.edu/tutorial/supervised/SoftmaxRegression/.

[42] D. G. Kleinbaum, K. Dietz, M. Gail, M. Klein, and M. Klein, *Logistic regression*. Springer, 2002.

[43] T. Parr, *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[44] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*. IBM Corp., 2010, pp. 214–224.

[45] J. Svajlenko and C. K. Roy, "Evaluating clone detection tools with bigclonebench," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*. IEEE, 2015, pp. 131–140.

[46] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[47] 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*. ACM, 2018, pp. 141–151.

[48] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[49] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 13–25.

[50] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Transactions on software Engineering*, 2020.

[51] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2019, pp. 41–50.

[52] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security*, 2019.

[53] Q. You, H. Jin, Z. Wang, C. Fang, and J. Luo, "Image captioning with semantic attention," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4651–4659.

[54] T. Xiao, Y. Xu, K. Yang, J. Zhang, Y. Peng, and Z. Zhang, "The application of two-level attention models in deep convolutional neural network for fine-grained image classification," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 842–850.

[55] J. Lu, J. Yang, D. Batra, and D. Parikh, "Hierarchical question-image co-attention for visual question answering," in *Advances In Neural Information Processing Systems*, 2016, pp. 289–297.

[56] V. Mnih, N. Heess, A. Graves *et al.*, "Recurrent models of visual attention," in *Advances in neural information processing systems*, 2014, pp. 2204–2212.

[57] M. F. Stollenga, J. Masci, F. Gomez, and J. Schmidhuber, "Deep networks with internal selective attention through feedback connections," in *Advances in neural information processing systems*, 2014, pp. 3545–3553.

[58] K. Gregor, I. Danihelka, A. Graves, D. J. Rezende, and D. Wierstra, "Draw: A recurrent neural network for image generation," *arXiv preprint arXiv:1502.04623*, 2015.

[59] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, "Show, attend and tell: Neural image caption generation with visual attention," in *International conference on machine learning*, 2015, pp. 2048–2057.

[60] Z. Yang, X. He, J. Gao, L. Deng, and A. Smola, "Stacked attention networks for image question answering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 21–29.

[61] C. Xiong, S. Merity, and R. Socher, "Dynamic memory networks for visual and textual question answering," in *International conference on machine learning*, 2016, pp. 2397–2406.

[62] K. J. Shih, S. Singh, and D. Hoiem, "Where to look: Focus regions for visual question answering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4613–4621.
[63] J. Li, M.-T. Luong, and D. Jurafsky, "A hierarchical neural autoencoder for paragraphs and documents," *arXiv preprint arXiv:1506.01057*, 2015.
[64] A. Kumar, O. Irsoy, P. Ondruska, M. Iyyer, J. Bradbury, I. Gulrajani, V. Zhong, R. Paulus, and R. Socher, "Ask me anything: Dynamic memory networks for natural language processing," in *International conference on machine learning*, 2016, pp. 1378–1387.
[65] R. Paulus, C. Xiong, and R. Socher, "A deep reinforced model for abstractive summarization," *arXiv preprint arXiv:1705.04304*, 2017.
[66] H. Zhang, I. Goodfellow, D. Metaxas, and A. Odena, "Self-attention generative adversarial networks," *arXiv preprint arXiv:1805.08318*, 2018.
[67] V. Saini, F. Farmahinifarahani, Y. Lu, P. Baldi, and C. V. Lopes, "Oreo: Detection of clones in the twilight zone," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 354–365.