DeepSim: Deep Learning Code Functional Similarity

Gang Zhao
Texas A&M University
College Station, Texas, USA
zhaogang92@tamu.edu

Jeff Huang
Texas A&M University
College Station, Texas, USA
jeff@cse.tamu.edu

ABSTRACT

Measuring code similarity is fundamental for many software engineering tasks, e.g., code search, refactoring and reuse. However, most existing techniques focus on code syntactical similarity only, while measuring code functional similarity remains a challenging problem. In this paper, we propose a novel approach that encodes code control flow and data flow into a semantic matrix in which each element is a high dimensional sparse binary feature vector, and we design a new deep learning model that measures code functional similarity based on this representation. By concatenating hidden representations learned from a code pair, this new model transforms the problem of detecting functionally similar code to binary classification, which can effectively learn patterns between functionally similar code with very different syntactics.

We have implemented our approach, DeepSim, for Java programs and evaluated its recall, precision and time performance on two large datasets of functionally similar code. The experimental results show that DeepSim significantly outperforms existing state-of-the-art techniques, such as DECKARD, RtvNN, CDLH, and two baseline deep neural networks models.

CCS CONCEPTS

• **Software and its engineering → Software libraries and repositories**; **Maintaining software**; *Software evolution*;

KEYWORDS

Code functional similarity, Control/Data flow, Deep Learning, Classification

ACM Reference Format:
Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3236024.3236068

1 INTRODUCTION

Measuring similarity between code fragments is fundamental for many software engineering tasks, e.g., code clone detection [44], code search and reuse [23, 30], refactoring [20, 61], bug detection [27, 36]. While code syntactical similarity has been intensively studied in the software engineering community, few successes have been achieved on measuring code functional similarity. Most existing techniques [6, 7, 26, 28, 36, 47, 58, 60] follow the same pipeline: first extracting syntactical features from the code (in the form of raw texts, tokens, or AST), then applying certain distance metrics (e.g. Euclidean) to detect similar code. However, the syntactical representations used by these techniques significantly limit their capability in modeling code functional similarity. In addition, such simple distance metrics can be ineffective for certain feature spaces (e.g., categorical). Moreover, separately treating each code fragment makes it difficult to learn similarity between functionally similar code with different syntactics.

There exist a few techniques [15, 17, 32, 37] that construct a program dependence graph (PDG) for each code fragment and measure code similarity through finding isomorphic sub-graphs. Compared to syntactical feature-based approaches, these graph-based techniques perform better in measuring code functional similarity. However, they either do not scale due to the complexity of graph isomorphism [13], or are imprecise due to the approximations made for improving scalability (e.g., mapping the subgraphs in PDG back to AST forest and comparing syntactical feature vectors extracted from AST [17]).

In this paper, we present a new approach, DeepSim, for measuring code functional similarity that significantly advances the state-of-the-art. DeepSim is based on two key insights. First, a feature representation is more powerful for measuring code semantics if it has a higher abstraction, because a higher abstraction requires capturing more semantic information of the code. Control flow and data flow represent a much higher abstraction than code syntactical features such as tokens [28, 47] and AST nodes [26]. Hence, DeepSim uses control flow and data flow as the basis of the similarity metrics. More importantly, we develop a novel encoding method that encodes both the code control flow and data flow into a compact semantic feature matrix, in which each element is a high dimensional sparse binary feature vector. With this encoding, we reduce the code similarity measuring problem to identifying similar patterns in matrices, which is more scalable than finding isomorphic sub-graphs.

Second, instead of manually extracting feature vectors from the semantic matrices and calculating the distance between different code, we leverage the power of deep neural networks (DNN) to learn a similarity metric based on the encoded semantic matrices. In the past decade DNN has led to breakthroughs in various areas such as computer vision, speech recognition and natural language processing [19, 49, 59]. Recently, a few efforts [8, 48, 58] have also been made to tackle program analysis problems with DNN. We design a new DNN model that further learns high-level features

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236068

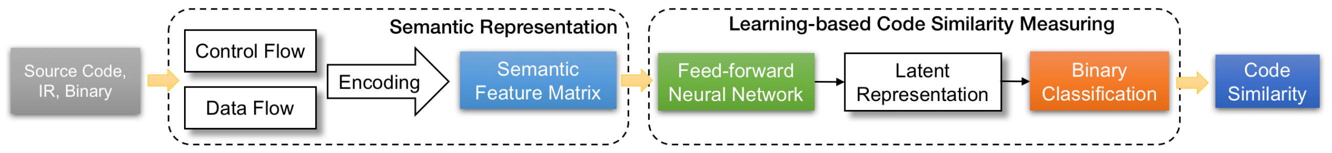


Figure 1: DeepSim System Architecture.

from the semantic matrices and transforms the problem of measuring code functional similarity to binary classification. Through concatenating feature vectors from pairs of code fragments, this model can effectively learn patterns between functionally similar code with very different syntactics.

As depicted in Figure 1, DeepSim consists of two main components: (1) Code semantic representation through encoding control flow and data flow into a feature matrix; (2) Code similarity measurement through deep learning. The first component can take code fragments in any language as input in the form of source code, bytecode or binary code, as long as a data-flow graph and a control-flow graph can be constructed. The second component is a novel deep learning model containing two tightly integrated modules: a neural network module that extracts high-level features from the matrices of a code pair, and a binary classification module that determines if the code pair is functionally similar or not. These two modules work together to fully utilize the power of learning: the neural network module extracts latent representation for the classifier, and the signal propagated backward from the classifier, in turn, helps the neural network learn better representation from the input. Finally, the trained model can be used to measure code functional similarity.

We have implemented DeepSim for Java at the method level based on the WALA framework [1] and TensorFlow [3]. We evaluated DeepSim on two datasets: a dataset of 1,669 Google Code Jam projects and the popular BigCloneBench [53], which contains over 6,000,000 tagged clone pairs and 260,000 false clone pairs. Our results show that DeepSim significantly outperforms the state-of-the-art techniques (with F1-score[1] 0.76 for Google Code Jam and 0.98 for BigCloneBench, compared to 0.50 and 0.82 the highest for the other techniques), such as DECKARD [26], RtvNN [58], CDLH [57], as well as two stacked denoising autoencoder (SdA) baseline deep learning models. Meanwhile, DeepSim achieves very good efficiency (even faster than DECKARD).

The contributions of this paper are as follows:

- We present a novel approach for measuring code functional similarity, which encodes code control and data flow into a single semantic feature matrix.
- We design a new deep learning model that learns high-level representation from the semantic matrices and learns a functional similarity metric between code with different syntactics.
- We implement a prototype tool, DeepSim[2] and evaluate it extensively with two large datasets, showing significant performance improvements over the state-of-the-art techniques.

[1] F1-score = $\frac{2*\text{recall}*\text{precision}}{\text{recall} + \text{precision}}$
[2] It is publicly available at: https://github.com/parasol-aser/deepsim.

2 BACKGROUND

Our deep learning model is based on feed-forward neural networks. In this section we first review the basic concepts.

2.1 Feed-Forward Neural Network

Feed-forward neural network, also named multi-layer perceptron (MLP), is an universal yet powerful approximator [24], and has been widely used. It is essentially a mapping from inputs to outputs: $F : \mathbb{R}^{m_0} \longrightarrow \mathbb{R}^{m_k}$, where m_0 is the dimensionality of inputs, m_k is the dimensionality of outputs generated by the last layer L_k. Each layer L_i is a mapping function: $F_{L_i} : \mathbb{R}^{m_{i-1}} \longrightarrow \mathbb{R}^{m_i}, 1 \le i \le k$. F is hence the composed function of all its layers: $F = F_{L_k}(F_{L_{k-1}}(\cdots(F_{L_1}(x)))$.

The mapping function of each layer is composed by its units (called neurons). A neuron a_{ij} (the j-th neuron in layer i) takes the outputs generated by previous layer (or the initial inputs) as inputs, and calculates a scalar output value through an activation function g:

$$a_{ij} = g(W_{ij}^T a_{i-1} + b_{ij}) \qquad a_i = \{a_{i1}, \ldots, a_{im_i}\}$$

where $W_{ij} \in \mathbb{R}^{m_{i-1}}, 1 \le j \le m_i$.

There are various activation functions according to different network designs. In this work we use ELU: $g(c) = c$ if $c \ge 0$, $e^c - 1$ if $c < 0$, where c is a scalar value [14].

The structure that several neurons form a layer and several layers form the integrated neural network has been proved effective in learning complicated non-linear mapping from inputs to outputs [21, 55]. Moreover, with the increasing computational power of modern GPUs, the depth of neural networks (the number of layers) can be made very large with a large training dataset – the so called "deep learning".

A representative feed-forward DNN model is deep autoencoder, which aims to learn a compact hidden representation from the inputs. Unlike standard encoding approaches (e.g., image/audio compression), it uses neural networks to learn the target encoding function instead of explicitly defining it. The goal is to minimize the error (e.g., squared error) of reconstructing the inputs from its hidden representation:

$$\begin{aligned} h_1 &= g(W_1 x + b_1) \\ &\ldots \\ h_k &= g(W_k h_{k-1} + b_k) \\ h'_{k-1} &= g(W' h_k + b'_{k-1}) \\ &\ldots \\ x' &= g(W'_1 h'_1 + b'_1) \end{aligned}$$

$$J(x, x') = \frac{1}{N} \sum_{i=1}^{N} ||x' - x||_2^2 \qquad (1)$$

where $W_i \in \mathbb{R}^{m_i \times m_{i-1}}$ and $b_i \in \mathbb{R}^{m_i}$ are the weights and biases of layer L_i ($W_i' = W_i^T$ if we use tied weights), h_k the encoded representation finally obtained, x the input of the model, x' the reconstruction of x, and N the number of all input samples.

Minimizing the reconstruction error forces this model to preserve as much information of the raw input as possible. However, some identity properties existed in the raw input may make the model simply learn a lookup function. Therefore, salt noise or corruption process is usually added to the input of the autoencoder, which is called Stacked Denoising Autoencoder (SdA) [55]. The corruption process works as randomly setting some components of the input to 0, which actually increases the sparsity of the autoencoder. The reconstruction process then tries to use this corrupted input to reconstruct the raw input:

$$h' = g(W \cdot \varepsilon(x) + b)$$
$$x'' = g(W'h' + b')$$
$$J(x'', x) = \frac{1}{N}\sum_{i=1}^{N} ||x'' - x||_2^2 \qquad (2)$$

where $\varepsilon(x)$ is the corruption process. In this way, it learns the hidden representation through capturing the statistical dependencies between components of the input.

2.2 Model Training

The training of neural networks is achieved by optimizing a defined loss function through gradient decent. For autoencoder, the reconstruction error discussed before is used as the loss function. For binary classification, a cross-entropy loss function is usually used:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} S^{(i)} log\tilde{S}^{(i)} + (1 - S^{(i)}) log(1 - \tilde{S}^{(i)}) \qquad (3)$$

where $\theta = (W, b)$, $S^{(i)}$ is the ground truth label for sample i, and $\tilde{S}^{(i)}$ is the predicted label.

To optimize $J(\theta)$, we can apply gradient descent to update W and b:

$$\theta := \theta - \alpha \cdot \frac{1}{n}\sum^{n} \frac{\partial J(\theta)}{\partial \theta}$$

Here n is the size of batching samples for each update. The learning rate α can be used to control the pace of parameters update towards the direction of gradient descent. Decaying α with time is generally a good strategy for finding a good local minimum [31].

Given the large depth of DNN, it is ineffective to compute the derivative of the loss function separately for each parameter. Hence, *back propagation* [45] is often applied. It consists of three stages: 1) a feed-forward stage to generate outputs using the current weights; 2) a back-forward stage to compute the responsive error of each neuron using the ground truth outputs and feed-forward outputs; and 3) a weights updating stage to compute the gradients using responsive errors and update the weights with a learning rate.

3 ENCODING SEMANTIC FEATURES

We encode the code semantics represented by control flow and data flow into a single semantic matrix. In this section, we present our encoding process.

3.1 Control Flow and Data Flow

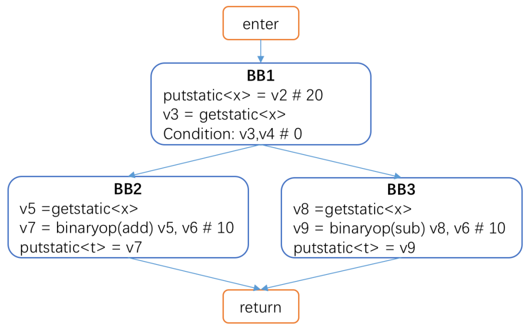
Control flow analysis and data flow analysis are two fundamental static program analysis techniques [38]. Control flow captures the dependence between code basic blocks and procedures, and data flow captures the flow of data values along program paths and operations. The information derived from these analyses is fundamental for representing the code behavior. Our basic idea is to encode code control flow and data flow as relationships between variables and between basic blocks (e.g., operations, control jumps).

To obtain the control flow and data flow, we may perform the analysis on either source code, binary code, or any intermediate representation (e.g., Java bytecode, LLVM bitcode). In this paper we focus on Java bytecode using the WALA framework [1].

```
public static int x = 0;
public static int t = 0;

public static void m1()
{
    x = 20;
    if (x!=0){
        t = x + 10;
    }else
        t = x - 10;
    }
}
```

(a) Example code



(b) Control flow graph of m1()

Figure 2: Control flow and data flow example.

Figure 2 illustrates a code example and its control flow graph (CFG). In the CFG, each rectangle represents a basic block, and in each basic block there may exist several intermediate instructions. For each basic block, a data flow graph (DFG) can be generated. Note that each variable and each basic block contain its *type* information. For example, x is a *static integer* variable, and $BB1$ is a basic block that contains a *conditional branch*. We describe how we encode them in more details in the next subsection.

3.2 Encoding Semantic Matrices

We consider three kinds of features for encoding the control flow and data flow information: variable features, basic block features, and relationship features between variables and basic blocks.

Variable features. For each variable, its type features contain its data type $V(t)$ (*bool, byte, char, double, float, int, long, short, void, other primitive types, JDK classes, user-defined types*), its modifiers

$V(m)$ (*final, static, none*) and other information $V(o)$ (*basic, array, pointer, reference*). We encode them into a 19d binary feature vector $V = \{V(m), V(o), V(t)\}$. Consider the variable x in Figure 2a. It is a *static int* variable, so we have $V = \{0, 0, \mathbf{1}, \mathbf{1}, 0, 0, 0, 0, 0, 0, 0, 0, \mathbf{1}, 0, 0, 0, 0, 0, 0\}$.

Basic block features. For each basic block, we consider the following seven types: *normal, loop, loop body, if, if body, switch, switch body*. Other types of basic blocks (e.g., *try catch* basic block) are regarded as *normal* basic blocks. Similarly, we encode this type information into a 7d one-hot feature vector B. For example, in Figure 2b, BB1 is a *if* basic block and its representation $B = \{0, 0, 0, \mathbf{1}, 0, 0, 0\}$.

Relationship features between variables and basic blocks. We encode data flow and control flow as operations between variables and control jumps between basic blocks. In total, we identify 43 different operation types and use a 43d one-hot feature vector I to represent it[3]. To preserve the information of each operation between variables in the DFG, we derive a 81d feature vectors $\mathcal{T} = \{V_{op1}, V_{op2}, I\}$, where V_{op1} and V_{op2} denote the variable feature vector for operand $op1$ and $op2$ respectively. To preserve the control flow, we extend $\mathcal{T}$ to be $\{V_{op1}, V_{op2}, I, B\}$ of size $E = 88$.

Now we formally define three rules to encode information for each relationship between variables and basic blocks:

- $\mathcal{T}(op1, op2) = \{V_{op1}, V_{op2}, I, B_{bb_{op1}}\}$ encodes the operations between two variables $op1$ and $op2$, as well as the type information of the two variables and the corresponding basic block. Here bb_{op1} denotes the basic block that $op1$ belongs to.
- $\mathcal{T}(op, bb) = \{V_{op}, V_0, I_0, B_{bb}\}$ encodes the relationship between a variable and its corresponding block. Here V_0, I_0 denote zero feature vectors.
- $\mathcal{T}(bb1, bb2) = \{V_0, V_0, I_0, B_{bb2}\}$ encodes the relationship between two neighbor basic blocks. Here $bb2$ is a successor of $bb1$ in the CFG.

Definition 3.1 (Semantic Feature Matrix $\mathbb{A}$). Given a code fragment, we can generate a matrix using the encoding rules above that captures its control flow and data flow information. Each row and column in $\mathbb{A}$ is a variable or a basic block. Their order corresponds to the code instruction and basic block order. $\mathbb{A}(i, j) = \mathcal{T}(i, j)$ is a binary feature vector of size $E = 88$ that represents the relationship between row i and column j, here $i, j = 1, 2, ..., n$, where $n = n_v + n_b$ is the summation of variables count and basic blocks count.

Example. Consider the example in Figure 2 again. Its CFG has 11 variables (9 local variables and 2 static fields) and 6 basic blocks (including *enter, return, exit* block). From the encoding rules above, we can derive a sparse matrix as visualized in Figure 3.

Compared to the raw CFG and DFG, our semantic feature matrix representation has two advantages. First, it reduces the problem of finding isomorphic subgraphs to detecting similar patterns in matrices. Second, this structure (i.e., a single matrix) makes it easy to use for the later processes, such as clustering or neural networks in our approach.

[3]In our current implementation, we rely on WALA's instruction types to distinguish different operations and control jumps. In particular, for some instruction types (e.g., SSABinaryOpInstruction) that have more than one optional opcodes (e.g., *add, div, etc.*), we treat each opcode as a different operation.

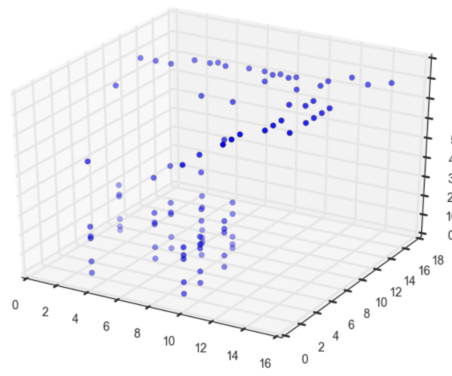


Figure 3: Semantic features matrix (17×17) generated from method *m1* in Figure 2. Along *z* axis are the 88d binary feature vectors. The value 1 is represented by a blue dot, and 0 represented by empty.

It is also worth noting that for most syntactically similar code that only differ in their identifier names, literal values or code layout, the generated semantic matrices will be identical, because these differences are normalized by CFG and DFG. This property ensures that our approach can also handle syntactical similarity.

We acknowledge that our matrix representation does not encode all information in a PDG. However, all dependence are implicitly encoded in our representation. For example, the code $y = x; z = x$ contains an input dependence; it is encoded as $\mathcal{T}(y, x)$ and $\mathcal{T}(z, x)$. Other types of dependence are encoded in similar way.

4 LEARNING-BASED CODE SIMILARITY MEASURING

From our semantic matrix encoding described in the previous section, we can see that the semantic matrices generated from two *functionally* similar methods with syntactical differences may not be identical. We cannot directly use simple distance metric (e.g., Euclidean distance) for measuring their similarity, since we have no knowledge about whether some elements (i.e., feature vector $\mathcal{T}$) in a semantic matrix is more important than another. Moreover, different elements may be functionally similar (e.g., *binary add* operations between two *int* and *long* variables respectively), but it is difficult to detect this similarity directly on the semantic feature matrix.

To address this issue, we develop a deep learning model that can effectively identify similarity patterns between semantic matrices. In particular, we train a specially designed feed-forward neural network that generates high-level features from semantic matrices for each code pair, and classify them as functionally similar or not using the classification layer at the end of the neural network. This design eliminates the need to define a separate distance metric on the learned features. More importantly, this model can be finely trained in a supervised manner to learn better representation from the input through backward propagating the label signal of the classification layer (i.e.,{0, 1}, see Eq. 3). This helps our model effectively learn patterns from similar code with very different syntactics.

4.1 Features Learning

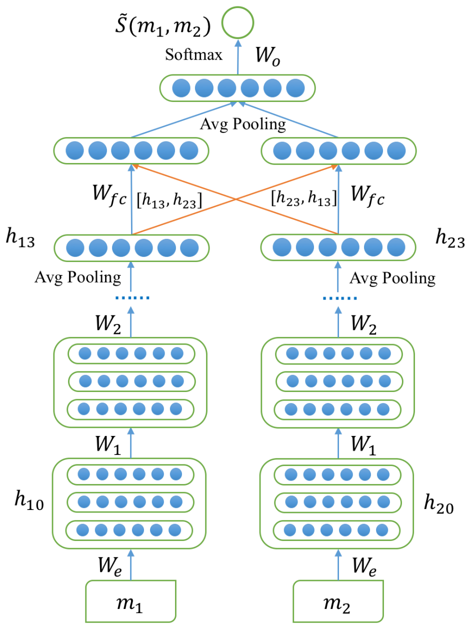


Figure 4: A schematic of our feed-forward model for measuring code functional similarity.

The architecture of our model is shown in Figure 4. As a static neural graph, this model requires a fixed size of input semantic feature matrices. Similar to recurrent neural network [22], we process all input matrices to a fixed size $k \times k$ ($k = 128$ in this work). For smaller methods, we pad their feature matrices with zero value. For larger methods, we truncate the matrices through keeping all CFG information and discarding part of DFG information that exceeds the matrix size (the original feature matrix size is $n = n_v + n_b$, after truncation, $k = n_{v'} + n_{b'} = 128$, where $n_{b'} = n_b$, $n_{v'} = 128 - n_b$).

Recall that we encode data flow and control flow into 88d sparse binary feature vectors $\mathcal{T}$. Similar to word embedding [11], we first map $\mathcal{T}$ into hidden state h_0 of size $c = 6$. This layer makes it possible to learn similarity between two relationship feature vectors. For example, *binary add* operations between two *int* and *float* variables respectively may share some similarities. We will discuss this further in Section 5.5.

Handling code statement re-ordering. We then flatten each row of $\mathbb{A}$ to a vector with length of $c \cdot k$. We add two fully connected layers taking the flattened row feature vectors as inputs, in order to extract all information related to the variable or basic block represented by each row. At last, a pooling layer is added to summarize all the row feature vectors. To mitigate the effect of code statement re-ordering, we use average pooling instead of flattening the entire matrix.

```
1       int c = 2;          1       int c = 2;
2       int x = 2*c;        2       int y = 3*c;
3       int y = 3*c;        3       int x = 2*c;
4       x = 2*y;            4       y = x*y;
5       y = x*y;            5       x = 2*y;
```

Consider the two small programs above, which have identical code statements but different statement orders at lines 2 and 3 and between lines 4 and 5. The different statement orders at lines 4 and 5 lead to different x and y values at the end of the two programs. In our matrix representation, we encode all the data flow between the three variables, and the statement order decides which variable or basic block that each row of the matrix represents. Thus, we generate two different $\mathbb{A}$s. However, if we only consider the statements at lines 1-3, the two programs should be equivalent, because there are no dependence between the two statements at lines 2 and 3.

Therefore, to make our model more robust in handling the statement re-ordering, we ignore the order of independent variables or basic blocks and keep only the order of variables or basic blocks with dependence between them. This can be achieved by performing average pooling on all those neighbor rows of $\mathbb{A}$ that represent independent variables or basic blocks, and flattening the rest rows (the order is preserved in the flattened vector). However, this would lead to various flattened vector lengths for different methods, which is difficult for a static neural network model to handle. Instead, we make approximation by directly performing averaging pooling on all rows, and leaving the rest task to our deep learning model.

Interestingly, another advantage of this design is that it reduces the model complexity, because the pooling layer reduces the input size from $c \times k \times k$ to $c \times k$ without any extra parameters (similar to pooling layers in convolutional neural network [34]). Thus, the training and predicting efficiency of our approach is increased.

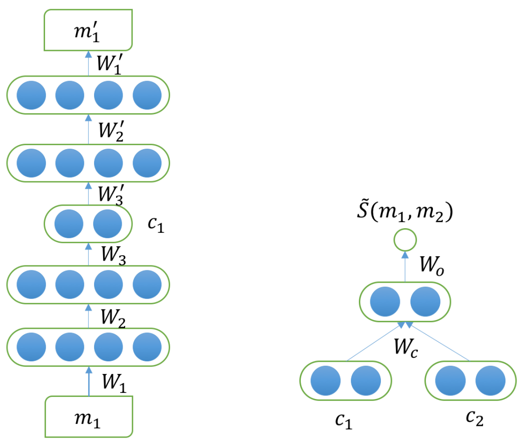
Similar to the vanilla multi-layer autoencoder (recall Section 2.1), this model can be trained in an unsupervised manner through minimizing the reconstruction error (Eq. 1 and Eq. 2). The training consists of two phases. Each layer is first pre-trained to minimize the reconstruction error, and then the entire model is trained through minimizing its overall reconstruction error. In this way, the model can learn a non-linear mapping from the input $\mathbb{A}$ to a compressed latent representation.

However, the model trained by this approach may discriminate two similar but not identical inputs, e.g., two array search methods that respectively handle *int* and *float* arrays. To enforce the model to learn similar patterns across different samples, we utilize supervised training, which is achieved by the binary classification layer that takes as input the concatenation of the final latent representations of two methods. We describe it in more details in the next subsection.

4.2 Discovering Functional Similarity

Given two methods with learned latent representations, a straightforward way to measure their functional similarity is calculating their distance in the hidden state space. This metric is applied by many existing techniques [26, 58].

However, such a simple distance metric is not satisfiable under two considerations. First, users have to perform heuristic analysis to find a suitable distance threshold for *every* dataset. Euclidean distance for measuring code similarity may not be applicable in the hidden space. For example, in the XOR problem, if we use Euclidean distance, the input (0, 0) would be closer to (0,1) and (1,0) than (1,1), which is a wrong classification. Second, a deterministic distance metric cannot leverage existing training dataset to finely train the model. Though it is feasible to obtain an optimal distance threshold using a training dataset, the parameters of the neural network model itself are not updated in this process.



(a) SdA for learning latent feature representation

(b) Measuring code similarity using latent representation

Figure 5: SdA baseline models. m_1 is a semantic matrix $\mathbb{A}$ generated from a method, c_1 and c_2 are two feature vectors generated by SdA from a method pair. For SdA-base, we put the two parts together when training, while for SdA-unsup, we do not back-propagate the error signal of (b) into (a).

We propose an alternative approach that formulates the problem of identifying functionally similar code as binary classification. More specifically, as shown in Figure 4, we concatenate the latent representation vectors h_{13} and h_{23} from methods m_1 and m_2 in different orders: $[h_{13}, h_{23}]$, $[h_{23}, h_{13}]$. Then we apply a fully connected layer on the two concatenated vectors with sharing weights W_{fc}. Another average pooling is performed on the output states, and finally the classification layers are added. We can now use cross-entropy as the cost function (Eq. 3), and minimize it to optimize the entire model. In this way, our model is able to learn similarity between each code pair with different syntactics.

Optimization. The two concatenations with different orders and the average pooling operation are important for optimizing the efficiency of our model. Note that as a similarity measuring task, we have a symmetric property: $S(m_i, m_j) = S(m_j, m_i)$. If we use only one concatenation $[h_{i3}, h_{j3}]$ or $[h_{j3}, h_{i3}]$, we have to add both (m_i, m_j) and (m_j, m_i) into our training dataset to satisfy the symmetric property, which will lead to nearly 2X training and predicting time.

When applying this model to discover functionally similar methods on a code repository of size N, we would need to run it $\frac{N(N-1)}{2}$ times. Considering that the classification layers only contain very few computations of the entire model (less than 1% matrix multiplications for our trained model), the efficiency can be significantly improved (almost 2X speedup) by separating the model into two parts during prediction. More specifically, for the N methods, we run the feed-forward phase of the model to generate their latent representation h_3. Then for each method pair, we only run the classification phase.

Baseline Model. As a comparison, we also train an SdA as the baseline model, as depicted in Figure 5. In the baseline, we try to flatten each matrix to a long input vector of size $k \times k \times E$ (i.e., 1,441,792 if k = 128 and E =88). However, even we set a small size of the first hidden layer (e.g., 100), there are over 1.44 $\times 10^8$ parameters, which makes the training difficult (unlike SdA, our model applies learning on each row followed by a row average pooling, hence its layer size is limited). Therefore, we slightly change the feature vector $\mathcal{T}$. As $\mathcal{T} = \{V_{op1}, V_{op2}, I, B\}$, $V_{op} = \{V(m), V(o), V(t)\}$, we use a 8-bit integer to represent each component of it, thus we drive a $\mathcal{T}'$ of size 8 (this can be understood as a handcrafted feature vector, as the characteristic vector in [26]). The other parts follow the standard SdA model.

We develop two settings for the SdA baseline model: *SdA-base* and *SdA-unsup*. SdA-base also adds classification layers on top of it (combine the two models in Figure 5), taking the concatenation of two latent representation vectors as inputs (with only one concatenating order). Thus, the fine training step will also optimize the feed-forward encoding part of the model. SdA-unsup uses a separate classification model with one hidden layer, as shown in Figure 5b. It works by estimating an optimal similarity metric in the hidden space. In both cases we apply the same loss function in Eq. 3. The goal of these two settings is to present a comparison between our model and the vanilla SdA, and to understand the effectiveness of our binary classification formulation.

5 EXPERIMENTAL EVALUATION

5.1 Datasets

We evaluated DeepSim on two datasets: Google Code Jam (GCJ) [2] and BigCloneBench [53]. In the first experiment, we follow recent literature [51, 52, 56] and use the programs from the Google Code Jam competition [2] as our dataset. We collected 1,669 projects from 12 competition problems, as shown in Table 1. Each project for the same problem is implemented by different programmers, and its correctness has been verified by Google. We have manually verified that the 12 problems are totally different. Therefore, programs for the same problem should be functionally similar (i.e., label 1), and those for different problems should be dissimilar (i.e., label 0). To better understand this benchmark, we manually inspected 15 projects for each problem and found that very few projects (6 pairs in total, 2 each in *Mushroom Monster* and *Rank and File*, and 1 each in *Brattleship* and *Senate Evacuation*) can be classified as syntactically similar[4]. Note that we did not remove any of these syntactically similar code, because our approach is also capable of handling them (as explained in Section 3). In addition, we found that more than 20% of the projects contain at least one obvious statement re-ordering. Most of them are caused by different orders of variable definitions used in loops or parameters for functions such as *min*, *max*, etc.

In the second experiment, we run DeepSim on the popular BigCloneBench dataset [53]. BigCloneBench is a large code clone benchmark that contains over 6,000,000 tagged clone pairs and 260,000 false clone pairs collected from 25,000 systems. Each clone

[4]Our criterion for syntactically similar code is very relax: the number of different lines of code can be up to 15 and no structural difference.

Table 1: The Google Code Jam dataset.

Dataset statistics	Value
Projects	1,669
Total lines of code	98,117
Tokens	445,371
Vocabulary size	4,803
Similar method pairs	275,570
Dissimilar method pairs	1,116,376

pair in BigCloneBench is also a pair of methods, and is manually assigned a clone type. The total number of clone pairs obtained from the downloaded database is slightly different from that reported in [53]. We discard code fragments without any tagged true or false clone pairs, and discard methods with LOC less than 5. In the end we obtained 50K methods, including 5.5M tagged clone pairs and 200K tagged false clone pairs. Most true/false clone pairs belong to clone type WT3/T4.

5.2 Implementation and Comparisons

For DeepSim and the SdA baseline models, we use WALA [1] to generate CFG/DFG from bytecode of each method, and construct the semantic feature matrix as described in Section 3. For efficiency, we store all the matrices in a sparse format. We use TensorFlow [3] to implement the neural networks models. For the GCJ dataset, we developed a plug-in in the Eclipse IDE to automatically inline source code file of each project to a single method. For BigCloneBench, because it does not provide the dependency libraries for the source code files, we modified WALA and the Polyglot-compiler framework to generate CFG/DFG for these source code files directly. For any external class type, we replace it with a unique type *Java.lang.Object*. This results in failures to get all fields in some cases. However, our tool is able to process over 95% of the clone pairs in BigCloneBench.

We also compared DeepSim with the three other state-of-the-art approaches: DECKARD [26], RtvNN [58] and CDLH [57]. DECKARD is a classical AST-based technique that generates characteristic vectors for each subtree using predefined rules and clusters them to detect code clones. In our experiment we used the stable version available in Github [4]. RtvNN uses recursive neural networks [49] to measure code similarity using Euclidean distance. Different from our approach, their model operates on source code tokens and AST and learns hidden representation for each code separately (i.e., unsupervised learning) rather than combining each code pair to learn similar patterns between them. Since the RtvNN implementation is not available, we implemented the approach following the paper [58]. CDLH is a recent machine learning-based functional clone detector that applies LSTM on ASTs. The CDLH paper [57] does not provide details about their experimental settings. To make a fair comparison, we compare with their reported numbers.

Training. We apply 10-fold cross-validation to train and evaluate DeepSim and the two baseline models on the two datasets. Namely, we partition the dataset into 10 subsets, each time we pick 1 subset as test set, the rest 9 subsets as training set. We repeat this 10 times and each time the picked test subset is different. The reported result is averaged over the 10 times.

Table 2: Parameter settings for different tools.

Tool	Parameters
DECKARD	Min tokens: 100, Stride: 2, Similarity threshold: 0.9
SdA-base	Layers size: (1024-512-256)-512-128, epoch: 6 Initial learning rate: 0.001, λ for L2 regularization: 0.000003 Corruption level: 0.25, Dropout: 0.75
SdA-separate	Feed-forward phase: Layers size: 1024-512-256, epoch: 1, λ_1 for L2 regularization: 0.0003, Corruption level: 0.25, initial learning rate: 0.001, Classification phase: Layers size: 512-128, epoch: 3, Dropout: 0.75, Initial learning rate: 0.001, λ_2 for L2 regularization: 0.001
RtvNN	RtNN phase: hidden layer size: 400, epoch: 25, λ_1 for L2 regularization: 0.005, Initial learning rate: 0.003, Clipping gradient range: (-5.0, 5.0), RvNN phase: hidden layer size: (400,400)-400, epoch: 5, Initial learning rate: 0.005 λ_1 for L2 regularization: 0.005 Distance threshold: 2.56
DeepSim	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

For RtvNN, because it needs to extract all the tokens and build vocabulary before training, we have to train and evaluate it using the same full dataset, following the same experiment setting as [58]. We try a range of different super-parameters (e.g., learning rate, layer sizes, regularization rate, dropout rate, various activation functions and weights initializers, etc.) for each model and record their testing errors and F1 scores. For DECKARD, it has no training procedure but a few parameters. We also tune its parameters and choose the set of parameters that achieved the highest F1 score on the full dataset. The optimal super-parameters that achieved the highest F1 score for these models are listed in Table 2.

5.3 Results on GCJ

Table 3: Results on the GCJ dataset.

Tool	Recall	Precision	F1 score
DECKARD	0.44	0.45	0.44
SdA-base	0.51	0.50	0.50
SdA-unsup	0.56	0.26	0.35
RtvNN	**0.90**	0.20	0.33
DeepSim	0.82	**0.71**	**0.76**

5.3.1 Recall and Precision. Table 3 reports the recall and precision of DeepSim on GCJ compared to the other approaches. Recall means the fraction of similar method pairs retrieved by an approach. Precision means the fraction of retrieved truly similar method pairs in the results reported by an approach. Overall, DeepSim achieved 81.6% recall, while DECKARD 44.0%, RtvNN 90.2%, SdA-base 51.3% and SdA-unsup 55.6%, and DeepSim achieved much higher precision (71.3%) than DECKARD (44.8%) and RtvNN (19.8%), as well as SdA-base (49.6%) and SdA-unsup (25.6%).

DECKARD achieved low recall and precision. The reason is that to recognize a similar method pair, DECKARD requires the characteristic vectors of parser tree roots for the two methods to be very close, which is essentially the syntactics of the entire code. To better understand this result, we picked all solutions from one competition problem, and found that more than half of the functionally similar code pairs have diverse parser tree structures, making them being predicted into different clusters by DECKARD.

SdA-base and SdA-unsup achieved comparable recall and are better than DECKARD, and SdA-base's precision is much higher than SdA-unsup. DeepSim achieved much higher recall than DECKARD and the two baseline models. This indicates that DeepSim effectively learns higher level features from the semantic matrices than the other approaches, and the encoded semantic matrix is also a better representation than the syntactical representation used by DECKARD.

RtvNN achieved the highest recall, but very low precision. We found that RtvNN almost reports all method pairs as similar. The reason is that RtvNN relies on the tokens and AST to generate the hidden representation for each method and applies a simple distance metric to discriminate them. However, two functionally similar methods may have significant syntactical differences, and two functionally dissimilar methods may share syntactically similar components (e.g., IO operations). After further analyzing the experiment results, we found that the distances between most methods are in the range of [2.0, 2.8]. Through reducing the distance threshold, the precision of RtvNN could be significantly improved (up to 90%), however, its recall also drops quickly (down to less than 10%). As a result, it only achieved F1 score 0.325 at the highest.

5.3.2 *False positives/false negatives.* The precision of DeepSim is 71.3%, which still has a large improvement space. After checking those false positives and false negatives of DeepSim, we found that this is mainly due to the tool's limitation in handling *method invocations*. For example, for the Problem *Senate Evacuation*, some solution projects use standard loops and assignment statements, while other solution projects employ utility methods such as *Map* and *Replace*. Because we do not explicitly encode the information inside the callee method into the semantic feature matrices, DeepSim cannot distinguish these utility methods and their corresponding statements. Nevertheless, this is a common limitation for most existing static code similarity measuring techniques [26, 47, 58]. Considering that DeepSim could generate a final hidden representation for each method after training, incorporating this information to encode method invocation instructions is feasible (re-training may be necessary). In addition, utilizing constraints-solving based approaches [29, 33] as a post-filtering to the reported results may also further improve the precision.

5.3.3 *Time Performance.* We also evaluated the time performance of these approaches on the full dataset (in total 1.4M method pairs). We run each tool with the optimal parameters (Table 2) on a desktop PC with an Intel i7 4.0GHz 4 cores CPU and GTX 1080 GPU. For DeepSim and the two SdA baseline models, they need to generate semantic feature matrices from bytecode files, so we also include the time of this procedure into the training time. For DECKARD, we use the default maximal number of processes (i.e., 8). We run each tool three times and report the average.

Table 4: Time performance on GCJ.

Tool	Prediction time	Training time
DECKARD	72s	-
SdA-base	1230s	8779s
SdA-unsup	37s	1482s
RtvNN	15s	8200s
DeepSim	34s	13545s

Table 4 reports the results. DECKARD does not need training so it has zero training time. SdA-unsup separates latent representation learning and classification phases, thus it takes fewer training epochs to get stable and has the lowest training time among the three models. While DeepSim has fewer neurons than SdA-base (SdA-base has a huge amount of neurons in the first layer), it takes more computation since its first three layers are applied on each element or row of the semantic feature matrices. Thus, DeepSim takes the longest training time. Although the training phase of DeepSim is slower than the other approaches, it is a one-time offline process. Once the model is trained, it can be reused to measure code similarity.

For prediction, RtvNN takes the least time, as it only needs to calculate the distance between each code pair and compare it with the threshold. DeepSim takes the 2nd least time, even faster than DECKARD, because it only needs to generate the semantic feature matrices and the prediction phase is performed by GPU, which is fast. SdA-unsup takes approximately the same time as DeepSim, while SdA-base is 30X slower, because it has to run the whole model for each code pair.

5.4 Results on BigCloneBench

Table 5: Results on BigCloneBench.

Tools	Recall	Precision	F1 Score
DECKARD	0.02	0.93	0.03
RtvNN	0.01	0.95	0.01
CDLH	0.74	0.92	0.82
DeepSim	**0.98**	**0.97**	**0.98**

Table 6: F1 score for each clone type.

Clone Type	DECKARD	RtvNN	CDLH	DeepSim
T1	0.73	1.00	**1.00**	0.99
T2	0.71	0.97	**1.00**	0.99
ST3	0.54	0.60	0.94	**0.99**
MT3	0.21	0.03	0.88	**0.99**
WT3/T4	0.02	0.00	0.81	**0.97**

Tables 5-6 report the results on BigCloneBench. The recall and precision are calculated according to [53]. The results of DECKARD, RtvNN and CDLH correspond to that reported in [57].

For this dataset, DeepSim significantly outperforms all the other approaches for both recall and precision. The F1 score of DeepSim is 0.98, compared to 0.82 by CDLH. DeepSim does not achieve 1.0

F1 score on T1-ST3 clones (which should be guaranteed by our encoding approaches as discussed in Section 3), because the tool misses some data flow when generating DFG from source code, due to the approximation we introduce to handle external dependencies in WALA.

Since the true/false clone pairs from BigCloneBench are from different functionalities, we run another experiment that use all the true/false clone pairs with functionality id 4 as training dataset (since it contains approximately 80% true/false clone pairs of the whole dataset), and the rest data as testing dataset. The result is shown in Table 7, which is consistent with the results reported in Tables 5-6.

Table 7: Results of DeepSim trained using data from single functionality.

Clone Type	Recall	Precision	F1 Score
T1	1.00	1.00	1.00
T2	1.00	1.00	1.00
ST3	1.00	1.00	1.00
MT3	0.99	0.99	0.99
WT3/T4	0.99	0.96	0.97

We also note that for WT3/T4 clone type, the F1 score of DeepSim on BigCloneBench is higher than that on the GCJ dataset (this is consistent with the comparison result between BigCloneBench and another OJ dataset, OJClone, as reported in [57]). We inspected several WT3/T4 clone pairs and found that although they are less than 50% similar at the statement level (which is the criterion for WT3/T4 clone type), many of them follow similar code structure and differ only on the sequence of the invoked APIs. In contrast, the GCJ projects are all built from scratch by different programmers with different code structures. It is hence more difficult to detect functional clones in the GCJ dataset.

5.5 Discussion

Why DeepSim outperforms the other DNN approaches. The reasons are three-fold. First, DeepSim is based on the semantic feature matrix that encodes DFG/CFG, which is a higher level abstraction than the lexical/syntactical representation (e.g., source code tokens or AST) used by the other DNN approaches such as RtvNN and CDLH. Compared to the two SdA baselines, DeepSim takes the semantic feature matrix as input, while for the SdA baselines each 88d feature vector $\mathcal{T}$ is manually re-encoded to 8d, and the entire matrix is flattened to a vector. The manual re-encoding may lose information since each element in the 8d vector is actually *categorical data* and a standard neural network model is limited in this scenario. In addition, flattening the entire matrix aggravates the impact of statement re-ordering because the flattened vector is strictly ordered, which causes reporting more false positives.

Second, DeepSim and the two SdA baselines all contain classification layers to support the classification of a method pair, while RtvNN only calculates the distance between the hidden representations of two methods, which may not be applicable as we discussed before. Note that the F1 score of DeepSim, SdA-base and SdA-unsup are all higher than RtvNN, this should be partly attributed to the effectiveness of our binary classification formulation and the fine training stage.

Third, the DeepSim model uses two average pooling layers, which SdA-base and SdA-unsup do not have. The first one computes the average states along all rows of the matrix, which reduces the side effect caused by the statement re-ordering. The second one computes the average states of two different concatenations of final latent representations from a method pair, which guarantees the symmetric property of code similarity, i.e. $S(m_i, m_j) = S(m_j, m_i)$.

Effectiveness of the encoding approach. In the first layer of DeepSim, we map each feature vector $\mathcal{T}$ of size 88 to a 6d vector in the hidden space. To analyze our trained model, we construct eight different feature vectors, each is encoding of a specific operation between variables. We map them into the embedding space and visual them with PCA dimensionality reduction, as shown in Figure 6.

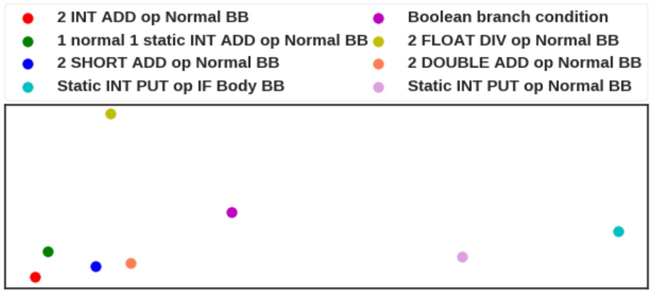


Figure 6: First layer hidden representation of 8 different input feature vectors generated by DeepSim.

We can observe that there are four very close points, corresponding to four different instructions: 2 int variables *add* operation in normal basic block, 2 short variables *add* operation in normal basic block, 2 double variables *add* operation in normal basic block, 1 int variable and 1 static int variable *add* operation in normal basic block. According to their meanings, they are quite similar and should be close in the embedding space. This manifests that our encoding does effectively preserve the data flow information, and our model successfully learned the similarity between these operations.

We also note that the two identical operations from different basic blocks (denoted by the plum and cyan points in the figure) are not close in the embedding space. This indicates that the code structure (i.e., control flow) information is also well encoded into the feature vector and successfully learned by our model for measuring functional code similarity.

The visualization of the hidden states for the eight instructions shows the effectiveness of our semantic feature matrices that encode data flow and control flow. More importantly, our specially designed DNN model can learn high-level features from the input semantic feature matrix, and patterns between functionally similar code with different syntactics. This significantly distinguishes our approach from the other approaches, which rely on syntactical representation.

Semantic feature matrix size. In this work, we fix the size of the semantic feature matrix to 128. It works well for our datasets, but such fixed length reduces computation efficiency for methods with smaller length and may lose some information for methods with larger length. To support variant matrix length, we can integrate our model with recurrent neural network (RNN). However, a challenge is that RNN requires a fixed input sequence order, which is difficult to handle code statement re-ordering. A stacked LSTM with attention mechanism [5] may be a potential solution.

Larger dataset. As a deep learning approach, DeepSim's effectiveness is limited by the size and quality of the training dataset. Building such a large and representative dataset is challenging. Even BigCloneBench only covers 10 functionalities and consists of less than 60K functions, which is a tiny proportion of the full IJaDataset [53]. In future work, this can be addressed by incrementally building a very large dataset through crowd-sourcing platform such as Amazon Mechanical Turk [39]. We also plan to build a web platform so that users can upload samples to try our tool and verify the result. The collected data will help improve the accuracy of our model.

6 OTHER RELATED WORK

Code clone detection. According to Carter et al. [12], Roy and Cordy [42], code clones can be roughly classified into four types. *Type I-III* code clones only differ at tokens and statement level, while *Type IV* code clones are functionally similar code that usually have different implementations. Existing code clone detectors mainly target type I-III code clones through measuring code syntactical similarity based on representations such as text [6, 7, 43], tokens [28, 36, 47], abstract syntax tree [10] or parse-tree [26].

Su et al. [52] proposed to measure functional code similarity based on dynamic analysis. This approach captures runtime information inside callee functions through code instrumentation, but it may miss similar code due to the limited coverage of test inputs. Several other approaches have focused on scaling code clone techniques to large repositories [25, 46, 47, 54].

In addition, Gabel and Su [18] studied code uniqueness on a large code repository of 420M LOC, and observed significant code redundancy at granularity of 6-40 tokens. Barr et al. [9] studied patching programs by grafting existing code snippets in the same program and how difficult this grafting process is. Their result indicates a promising application of code similarity measuring techniques.

Machine learning for measuring code similarity. Carter et al. [12] proposed to measure code similarity through extracting handcrafted features from source code and applying self-organizing maps on them. Yang et al. [60] derived more robust features from source code based on inverse frequency-reverse document frequency, then applied K-Means to cluster code and predict the class for a new code fragment using the cosine distance metric. This approach cannot handle code fragments that do not belong to any existing clusters. Li et al. [35] also applied deep learning on code clone detection. They first identified tokens that are likely to be shared between programs, then computed the frequency of each token in each program. Given a code pair, they computed a similarity score for each token based on their frequencies. The similarity score vector is then fed to the deep learning model to classify the code pair. Since token frequency is still a syntactical representation, the ability of this approach for measuring code functional similarity may be limited.

Semantic code search and equivalence checking. Reiss [41] used both static and dynamic specifications (e.g., keywords, class of method signatures, test cases, contracts) to search target code fragments. Stolee et al. [50] and Ke et al. [29] used symbolic execution to build constraints for each code fragment in the codebase and searched target code with input-output specifications. The returned code fragment should satisfy both the type signature and the conjoined constraints. This approach may not scale well as limited by symbolic execution and SMT solver. Moreover, Deissenboeck et al. [16] reported that 60%-70% of program chunks across five open-source Java projects refer to project specific data-types, which makes it difficult to directly compare inputs/outputs for equivalence checking across different projects. Partush and Yahav [40] presented a speculative code search algorithm that finds an interleaving of two programs with minimal abstract semantic differences, which abstracts the relationships between all variables in the two programs. This may not be applicable for non-homologous programs since there is not a clear mapping between most statements and variables.

7 CONCLUSION

We have presented a new approach, DeepSim, for measuring code functional similarity, including a novel semantic representation that encodes the code control flow and data flow information as a compact matrix, and a new DNN model that learns latent features from the semantic matrices and performs binary classification. We have evaluated DeepSim with two large datasets of functionally similar code and compared it with several state-of-the-art approaches. The results show that DeepSim significantly outperforms existing approaches in terms of recall and precision, and meanwhile it achieves very good efficiency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their suggestions on earlier versions of this paper, and Tengxiao Wang, Shiyou Huang, and Bozhen Liu for valuable discussions on DeepSim.

REFERENCES

[1] 2006. T.J. Watson Libraries for Analysis (WALA). http://wala.sourceforge.net/wiki/index.php/Main_Page/. Accessed: 2016-09-02.
[2] 2016. Google Code Jam. https://code.google.com/codejam/contests.html. Accessed: 2016-10-08.
[3] 2016. TensorFlow: An open-source software library for Machine Intelligence. https://www.tensorflow.org/. Accessed: 2017-08-02.
[4] 2017. Deckard Github repo. https://github.com/skyhover/Deckard. Accessed: May/2017.
[5] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).
[6] Brenda S Baker. 1993. A program for identifying duplicated code. *Computing Science and Statistics* (1993), 49–49.
[7] Brenda S Baker. 1996. Parameterized pattern matching: Algorithms and Applications. *J. Comput. System Sci.* 52, 1 (1996), 28–42.
[8] Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. 2016. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989* (2016).
[9] Earl T Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 306–317.
[10] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 368–377.

[11] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. A neural probabilistic language model. *Journal of Machine Learning Research* 3, Feb (2003), 1137–1155.

[12] S Carter, RJ Frank, and DSW Tansley. 1993. Clone detection in telecommunications software systems: A neural net approach. In *Proc. Int. Workshop on Application of Neural Networks to Telecommunications*. 273–287.

[13] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 175–186.

[14] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. 2015. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289* (2015).

[15] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the clones: Detecting cloned applications on android markets. In *European Symposium on Research in Computer Security*. Springer, 37–54.

[16] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Stefan Wagner. 2012. Challenges of the dynamic detection of functionally similar code fragments. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE, 299–308.

[17] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 321–330.

[18] Mark Gabel and Zhendong Su. 2010. A study of the uniqueness of source code. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, 147–156.

[19] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, et al. 2014. Deep speech: Scaling up end-to-end speech recognition. *arXiv preprint arXiv:1412.5567* (2014).

[20] Yoshiki Higo, Toshihiro Kamiya, Shinji Kusumoto, Katsuro Inoue, and K Words. 2004. ARIES: Refactoring support environment based on code clone analysis.. In *IASTED Conf. on Software Engineering and Applications*. 222–229.

[21] Geoffrey E Hinton and Ruslan R Salakhutdinov. 2006. Reducing the dimensionality of data with neural networks. *Science* 313, 5786 (2006), 504–507.

[22] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural Computation* 9, 8 (1997), 1735–1780.

[23] Reid Holmes and Gail C Murphy. 2005. Using structural context to recommend source code examples. In *Proceedings of the 27th International Conference on Software Engineering*. ACM, 117–125.

[24] K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.* 2, 5 (July 1989), 359–366.

[25] Benjamin Hummel, Elmar Juergens, Lars Heinemann, and Michael Conradt. 2010. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*. IEEE, 1–9.

[26] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 96–105.

[27] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, 55–64.

[28] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.

[29] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 295–306.

[30] Iman Keivanloo, Juergen Rilling, and Ying Zou. 2014. Spotting working code examples. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 664–675.

[31] Diederik Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[32] Jens Krinke. 2001. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 301–309.

[33] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*. Springer, 712–717.

[34] Yann LeCun, Yoshua Bengio, et al. 1995. Convolutional networks for images, speech, and time series. *The handbook of brain theory and neural networks* 3361, 10 (1995), 1995.

[35] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 249–260.

[36] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *OSDI*, Vol. 4. 289–302.

[37] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 872–881.

[38] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of program analysis*. Springer.

[39] Gabriele Paolacci, Jesse Chandler, and Panagiotis G Ipeirotis. 2010. Running experiments on amazon mechanical turk. (2010).

[40] Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. *ACM SIGPLAN Notices* 49, 10 (2014), 811–828.

[41] Steven P Reiss. 2009. Semantics-based code search. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 243–253.

[42] Chanchal Kumar Roy and James R Cordy. 2007. A survey on software clone detection research. *QueenâĂŹs School of Computing TR* 541, 115 (2007), 64–68.

[43] Chanchal K Roy and James R Cordy. 2008. 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, 172–181.

[44] Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495.

[45] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. 1988. Learning representations by back-propagating errors. *Cognitive modeling* 5, 3 (1988), 1.

[46] Hitesh Sajnani, Vaibhav Saini, and Cristina Lopes. 2015. A parallel and efficient approach to large scale clone detection. *Journal of Software: Evolution and Process* 27, 6 (2015), 402–429.

[47] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. 2016. SourcererCC: scaling code clone detection to big-code. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 1157–1168.

[48] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Network. In *USENIX Security Symposium*. 611–626.

[49] Richard Socher, Cliff C Lin, Chris Manning, and Andrew Y Ng. 2011. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*. 129–136.

[50] Kathryn T Stolee, Sebastian Elbaum, and Daniel Dobos. 2014. Solving the search for source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 23, 3 (2014), 26.

[51] Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. 2016. Identifying functionally similar code in complex codebases. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[52] Fang-hsiang Su, Kenneth Harvey, Simha Sethumadhavan, Gail E Kaiser, and Tony Jebara. 2015. Code Relatives: Detecting Similar Software Behavior. (2015).

[53] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal Kumar Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones.. In *ICSME*. 476–480.

[54] Jeffrey Svajlenko, Iman Keivanloo, and Chanchal K Roy. 2013. Scaling classical clone detection tools for ultra-large datasets: An exploratory study. In *Proceedings of the 7th International Workshop on Software Clones*. IEEE Press, 16–22.

[55] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. 2008. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*. ACM, 1096–1103.

[56] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. 2016. How are functionally similar code clones syntactically different? An empirical study and a benchmark. *PeerJ Computer Science* 2 (2016), e49.

[57] Hui-Hui Wei and Ming Li. 2017. 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, 3034–3040.

[58] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 87–98.

[59] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144* (2016).

[60] Jiachen Yang, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. 2015. Classification model for code clones based on machine learning. *Empirical Software Engineering* 20, 4 (2015), 1095–1125.

[61] Minhaz F Zibran and Chanchal K Roy. 2011. Towards flexible code clone detection, management, and refactoring in IDE. In *Proceedings of the 5th International Workshop on Software Clones*. ACM, 75–76.