Dynamic Syntax Tree Model for Enhanced Source Code Representation

Jaden Roy [*] , Rodolfo Patel , Pasir Nguyen , Sartran Simon

Posted Date: 14 December 2023

doi: 10.20944/preprints202312.1108.v1

Keywords: Tree modeling; Code representation learning

Article

Dynamic Syntax Tree Model for Enhanced Source Code Representation

Jaden Roy *, Rodolfo Patel, Pasir Nguyen and Sartran Simon

Briar Cliff University
* Correspondence: jroy@briarcliff.edu

Abstract: The art of representing source code is pivotal in numerous programming analysis applications. Recent strides in neural networks have marked notable successes in this realm. However, the peculiar structural characteristics inherent in programming languages have not been fully exploited in existing models. While neural models based on abstract syntax trees (ASTs) adeptly manage the tree-like nature of source codes, they fall short in discerning the diverse substructural nuances within programs. This paper introduces the Dynamic Syntax Tree Model (DSTM), an innovative approach that fuses various neural network modules into tree architectures tailored to the specific AST of the input. Distinct from preceding tree-based neural models, DSTM adeptly discerns the semantic variances across different AST substructures. We validate DSTM through rigorous testing in program classification and code clone detection, outperforming contemporary methods and demonstrating the benefits of harnessing intricate source code structures.

Keywords: tree modeling; code representation learning

1. Introduction

The field of source code representation learning, a pivotal aspect of software engineering, aims to create methodologies enabling systems to autonomously derive representations necessary for feature detection or classification from source code [1–4]. The advent and rapid advancement of deep learning technologies have brought a paradigm shift in this field. Source code modeling using neural networks, a relatively recent endeavor, has attracted increasing attention due to its promising results in various programming analysis tasks [7–10]. The Dynamic Syntax Tree Model (DSTM) represents a significant leap in this direction, offering a sophisticated approach to understanding and interpreting the complex structures inherent in programming languages.

Deep-learning-based techniques for representing source code have demonstrated remarkable success across a spectrum of applications, including program classification, defect prediction, code clone detection, code summarization, and even malware detection. These applications highlight the growing versatility and importance of neural network approaches in the realm of software engineering and programming language analysis.

As highlighted by Hindle et al. [15], programming languages exhibit statistical properties akin to natural languages. This observation has guided researchers to adopt successful natural language processing models for source code representation. Traditional methodologies in this domain have treated source code similarly to natural language text, employing tokenization processes where the code is segmented into a sequence of tokens. These methods have extensively utilized various forms of neural networks, such as Convolutional Neural Networks (CNNs) [16–18] and Recurrent Neural Networks (RNNs) [19–21]. Despite the efficacy of these models, they are constrained by the inherent simplicity of natural language structures, which starkly contrasts with the more intricate and explicitly structured nature of programming languages.

The complex and explicit structural information of programming languages is most comprehensively captured in their abstract syntax trees (ASTs). ASTs are rich, tree-like structures that represent the syntactic structure of source code, mirroring the grammatical rules that define the programming language. Each node in an AST corresponds to a construct occurring in the source code, offering a detailed breakdown of its syntax. This level of granularity in representation is essential for accurately capturing the semantics and functionalities of different code segments. The Dynamic Syntax Tree Model (DSTM) is specifically designed to leverage this detailed structural information, offering a more nuanced and accurate representation of source code than previous models.

The innovation of DSTM lies in its ability to dynamically compose a tree-structured neural network framework, which models different AST semantic units with distinct neural network modules. Each module is specifically tailored to represent a particular type of semantic unit, as defined by the grammar of the programming language. This approach contrasts sharply with existing tree-structured neural network models, such as Tree-LSTM [22] and TBCNN [23], which apply a one-size-fits-all neural unit to represent various semantic units in ASTs. While these models successfully utilize the tree structure of source code, they fail to differentiate between the unique semantic meanings of different AST components, often resulting in a generalized and less accurate representation.

DSTM addresses this gap by providing a diverse array of neural modules, each intricately designed to capture the specific semantic meaning of different AST components. This modular approach enables DSTM to construct a highly customized and accurate neural network architecture for each input AST, ensuring that the nuanced differences between various semantic units are adequately represented. The result is a more precise and semantically rich representation of the source code, which significantly enhances the model's performance in various programming analysis tasks.

To validate the effectiveness of the Dynamic Syntax Tree Model, we conduct extensive experiments in two critical areas of software engineering: program classification and code clone detection. Our evaluations reveal that DSTM not only achieves superior accuracy in program classification but also markedly improves the F1-score in code clone detection when compared to existing models. These results are a testament to the model's ability to leverage the detailed semantic information embedded in source code, leading to more accurate and efficient program analysis.

The subsequent sections of this paper will delve deeper into the background of deep learning technologies in the context of programming language analysis (Section 2), the intricate structure and workings of DSTM (Section 3), the practical applications and benefits of employing DSTM in program classification and code clone detection (Section 4), and a thorough analysis of our experimental results (Section 5). We will also discuss potential threats to the validity of our findings (Section 6) and review related works in this domain (Section 7), before concluding with a summary of our contributions and the implications of our research (Section 8).

2. Related Work

The endeavor of learning distributed representations for source code, central to machine learning in software engineering, involves obtaining a nuanced representation of code snippets. The Dynamic Syntax Tree Model (DSTM) is a novel contribution in this field, which typically employs supervised models trained on tasks with labeled data like program classification, defect prediction, clone detection, and code summarization to acquire intermediate code representations [2,4,24,25]. This section reviews the evolution of source code representation methodologies, noting the influence of natural language processing models in this progression. We categorize neural network models for code modeling into sequential, hierarchical, and tree-structured types. Additionally, we explore prior efforts in modeling different semantic compositions in non-programming language tree structures, highlighting their limitations in handling ASTs and laying the groundwork for our DSTM approach.

Source code, akin to natural language, can be conceptualized as a sequence of tokens. Lexical analyzers are often employed to break down code into these tokens. Iyer et al. [19] introduced Code-NN, a model using LSTM networks [26] with attention mechanisms for generating descriptions from C# code snippets and SQL queries. Allamanis et al. [16] implemented a convolutional neural network with attention for summarizing Java source code. Wang et al. [7] utilized deep belief networks [29] for defect prediction. Other models serialize the AST using depth-first traversal and employ RNNs for tasks like code completion [20,21]. These sequential models, while efficient, often overlook the inherent structural complexity of source code.

Beyond simple sequences, programs can also be represented as hierarchies. Each line in a program is a sequence of tokens, and the entire program is a sequence of such lines. Huo et al. explored this concept, using CNNs for within-line and LSTM for cross-line modeling in bug localization tasks [18,30]. However, these models do not fully capture the logical structures within code lines, especially in constructs like loops and conditional branches.

Tree-structured models, primarily built on program ASTs, are integral to source code analysis. These models use a neural unit to compute the representation of a semantic unit in the tree. Socher et al. [31] introduced recursive neural networks for this purpose. However, adapting these networks to non-binary ASTs of source code often complicates the training process. Liang et al. [32] and Tai et al. [22] made significant strides with models like Code-RNN and Tree-LSTM. Wei et al. [33] and Mou et al. [23] further extended these concepts to tasks like code clone detection. However, these models typically use uniform neural units for different semantic units, limiting their effectiveness. Allamanis et al. [34] took a different approach by augmenting ASTs into graphs for variable misuse and naming tasks, illustrating the potential of incorporating additional static analysis data.

The concept of varying semantic units is also relevant in natural language parse trees. Researchers in this area have explored the use of different compositional functions for diverse tree structures [35–37]. For instance, Socher et al. [35] employed distinct weight matrices for different sibling combinations, while Dong et al. [36] introduced AdaRNN for sentiment classification. Arabshahi et al. [37] and Liu et al. [38] developed models with varied parameters for different mathematical and linguistic structures. These innovations underline the importance of tailored compositional functions, though their application to the complex and diverse structures of program ASTs remains a challenge, one that the Dynamic Syntax Tree Model (DSTM) seeks to address.

3. Preliminary

This section delves into the theoretical underpinnings of our proposed Dynamic Syntax Tree Model (DSTM), focusing on tree-structured neural networks, particularly the recursive neural network and tree-LSTM, which form the basis of our model's architecture.

3.1. Recursive Neural Network (RvNN)

The Recursive Neural Network (RvNN) stands as a fundamental archetype in the realm of tree-structured neural networks. Consider a binary tree node j with two children, $c1$ and $c2$. The RvNN computes the representation of node j using its children's representations as follows:

$$h_j = tanh(W[h_{c1} : h_{c2}] + b) \tag{1}$$

Here, W represents the shared weight matrix, and b denotes the bias term. The RvNN's design necessitates a uniform branching factor across the tree, typically binary, to ensure consistency in computation. This model finds its primary application in structures akin to natural language constituency trees, where leaf nodes encapsulate the lexical tokens of sentences, thereby forming the foundational inputs for the network.

3.2. Tree-LSTM

The Tree-LSTM extends the Long Short-Term Memory (LSTM) architecture to accommodate tree-structured network topologies, as detailed in Tai et al. [22]. Traditional LSTM, a variant of recurrent neural networks, was conceived to address the challenge of long-term dependencies that plagued conventional RNNs. In an LSTM, the hidden state at any given time step t, denoted as h_t, is a function of both the current input x_t and the preceding hidden state h_{t-1}:

$$h_t = tanh(Wx_t + Uh_{t-1} + b) \tag{2}$$

In this equation, W and U are distinct weight matrices for the input vector and hidden state, respectively. The LSTM cell incorporates several gates - a forget gate f, an input gate i, an output gate o, and a memory cell - each equipped with unique parameters (W, U, b) and responsible for controlling the flow and retention of information. The Tree-LSTM adapts these mechanisms to tree structures, where instead of a single preceding hidden state, the gates consider the combined hidden states of all child nodes.

This architecture manifests in two primary variants: Child-Sum Tree-LSTM and N-ary Tree-LSTM, each with its unique approach and limitations.

3.2.1. Child-Sum Tree-LSTM

The Child-Sum Tree-LSTM aggregates the hidden states of all child nodes of a given parent node j and utilizes this aggregated state as input for the input and output gates. This variant is flexible and can be implemented on trees with arbitrary structures. A notable shortcoming, however, is its insensitivity to the order of child nodes - a limitation stemming from the summation operation, which inherently disregards sequence information.

3.2.2. N-ary Tree-LSTM

Contrasting with its Child-Sum counterpart, the N-ary Tree-LSTM is tailored for trees with a branching factor of at most N. This variant is particularly adept at leveraging the sequential order of child nodes. Unlike the Child-Sum approach, which uniformly sums the hidden states, the N-ary Tree-LSTM assigns distinct weight matrices U to each child, thereby capturing more nuanced semantic details and the relative importance of each child's position.

Applying the N-ary Tree-LSTM to structures like abstract syntax trees, which typically have indefinite branching factors, necessitates a conversion to N-ary (often binary) trees. This conversion process, while enabling compatibility with the N-ary Tree-LSTM, often disrupts the intrinsic syntactic relationships and deepens the tree structure, potentially exacerbating long-term dependency issues.

In the subsequent sections, we will explore how the Dynamic Syntax Tree Model (DSTM) builds upon these foundational concepts, addressing their limitations and innovatively applying them to the realm of source code representation.

4. Methodology

Our Dynamic Syntax Tree Model (DSTM) innovatively refines the traditional child-sum tree-LSTM framework by incorporating specialized neural modules for different AST semantic units. This strategy is adaptable to various programming languages, though it necessitates designing unique modules for each language's specific semantic units. For instance, programming languages like C and Python have distinct semantic units and structures.

In this paper, we specifically focus on DSTM for C programming, while acknowledging the model's potential for generalization to other languages with tailored neural modules. Figure 2 illustrates DSTM's framework, processing ASTs of source code into vector representations. We first outline the general structure of DSTM units, then introduce neural modules for capturing semantic features in C, and finally describe how DSTM computes vector representations for ASTs.

4.1. DSTM Structure

In DSTM, the traditional aggregation of children's hidden states in Child-Sum Tree-LSTM is replaced by one of our specially designed neural modules. For a given node j, a DSTM unit computes its hidden state as follows:

$$\widetilde{h}_j = F_{type-j}(h_{s_1}, ..., h_{s_{n_j}}) \tag{3}$$
$$i_j = \sigma(W^{(i)}_{type-j} x_j + U^{(i)}_{type-j} \widetilde{h}_j + b^{(i)}_{type-j}) \tag{4}$$
$$f_{jk} = \sigma(W^{(f)}_{type-j} x_j + U^{(f)}_{type-j} h_k + b^{(f)}_{type-j}) \tag{5}$$
$$o_j = \sigma(W^{(o)}_{type-j} x_j + U^{(o)}_{type-j} \widetilde{h}_j + b^{(o)}_{type-j}) \tag{6}$$
$$u_j = \tanh(W^{(u)}_{type-j} x_j + U^{(u)}_{type-j} \widetilde{h}_j + b^{(u)}_{type-j}) \tag{7}$$
$$c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k \tag{8}$$
$$h_j = o_j \odot \tanh(c_j) \tag{9}$$

The parameters W, U, and b vary with the node type j, and $\odot$ denotes element-wise multiplication. Equation (3) specifically employs our neural module F_{type-j}, tailored to the node type of j. We discuss F_{type-j} in section 3.2. For some node types without specific modules, we revert to the Child-Sum Tree-LSTM approach. The remaining equations resemble standard Tree-LSTM, covering gates and memory cells of the DSTM unit.

Each DSTM unit can be visualized as a specialized neural module encased in a Tree-LSTM-like framework, each with $8d^2 + 4d$ parameters. We propose two parameter-sharing strategies:

DSTM-a: Shared container parameters across all DSTM units in a program's AST.

DSTM-b: Unique container parameters for each semantic unit type with a distinct F_{type} module, and a shared set for those without specific F_{type} modules.

Figure 3 shows a comparison between DSTM-a and DSTM-b, with different colors representing various container parameters.

4.2. Neural Modules For Different Semantic Units

C programming language semantic units fall into categories such as logic structures, data declarations, and computation operations. We focus on units fundamental to program structure, designing eight distinct neural modules for different semantic units, including `FuncDef`, `While`, and others. Additionally, we have a module F_{seq} for nodes with a variable number of sequential children. Table 1 lists these modules and their parameter counts given a hidden size d. Below, we detail the implementation and rationale behind these modules.

4.2.1. FuncDef, While, DoWhile, and Switch

These nodes share structural similarities, each with two distinct child types. We employ recursive neural network modules with unique parameters for each node type, capturing their specific semantic differences.

4.2.2. For

The `For` node, comprising initialization, condition, iteration, and body, requires nuanced handling. We model the loop's control elements with an RvNN unit, combining its output with the body node's state in another RvNN unit, as shown in Figure 4.

4.2.3. If

The `If` node in C varies in children count, making its representation challenging. We use recursive networks to combine condition and branch bodies. In cases with an else statement, we process each branch separately and then merge their outputs using element-wise maximization.

4.2.4. Case

`Case` nodes, integral to switch-case structures, have two distinct parts: a constant expression and a sequence of statements. We encode statement sequences using LSTM and combine the last LSTM state with the constant expression node's state using a recursive neural layer.

4.2.5. Nodes with Sequential Children

In the AST for C programming, certain node types, like the Compound node which denotes a code block, have a sequence of statements as children of arbitrary length. We employ an LSTM approach to encode these child nodes, considering the final state of the LSTM as the node's output representation.

Table 1. Neural modules for DSTM. Each module is tailored to a specific semantic unit, with $LSTM()$ processing a sequence of inputs to produce the final hidden state.

ine Semantic Unit	Function	Parameters
ine `FuncDef`$\rightarrow$ decl, body	$h_{FuncDef} = tanh(W_{FuncDef}[h_{decl} : h_{body}] + b_{FuncDef})$	$2d^2 + d$
ine `While`$\rightarrow$ cond, stmt	$h_{While} = tanh(W_{While}[h_{cond} : h_{stmt}] + b_{While})$	$2d^2 + d$
ine `DoWhile`$\rightarrow$ cond, stmt	$h_{DoWhile} = tanh(W_{DoWhile}[h_{cond} : h_{stmt}] + b_{DoWhile})$	$2d^2 + d$
ine `For`$\rightarrow$ init, cond, next, stmt	$h_{For} = RNN_{For}(W_{For0}[tanh(W_{For1}[h_{init} : h_{cond} : h_{next}] + b_{For1}) : h_{stmt}] + b_{For0})$	$5d^2 + 2d$
ine `If`$\rightarrow$ cond, iftrue	$h_{If} = RNN_{If}(W_{If}[h_{cond} : h_{iftrue}] + b_{If})$	$2d^2 + d$
`If`$\rightarrow$ cond, iftrue, iffalse	$h_{If} = max(RNN_{If}(W_{If}[h_{cond} : h_{iftrue}] + b_{If}), RNN_{If}(W_{If}[h_{cond} : h_{iffalse}] + b_{If}))$	
ine `Switch`$\rightarrow$ cond, stmt	$h_{Switch} = RNN_{Switch}(W_{Switch}[h_{cond} : h_{stmt}] + b_{Switch})$	$2d^2 + d$
ine `Case`$\rightarrow$ expr, stmt1, ..., stmtk	$h_{Case} = RNN_{Case}(W_{Case}[h_{expr} : LSTM_{Case}(h_{stmt1}, ..., h_{stmtk})] + b_{Case})$	$10d^2 + 5d$
ine `Seq`$\rightarrow$ child1, ..., childk	$h_{seq} = LSTM_{seq}(h_{child1}, ..., h_{childk})$	$8d^2 + 4d$
ine		

For a hidden size d, the total parameter count for all module types is $33d^2 + 16d$. Thus, a DSTM-a model encompasses $8d^2 + 4d + 33d^2 + 16d = 41d^2 + 20d$ parameters, whereas a DSTM-b model comprises $9(8d^2 + 4d) + 33d^2 + 16d = 105d^2 + 52d$ parameters.

4.3. Computing Code Snippet Representations with DSTM

To obtain vector representations of source code using DSTM, code snippets are first parsed into ASTs. We construct a DSTM network mirroring the structure of each AST. In C ASTs, the semantic unit type corresponds uniquely to a parent node type, simplifying the process of determining module types to employ. Nodes in {`FuncDef`, `While`, and others} are matched with corresponding DSTM units. For nodes with sequential children, $DSTM_{seq}$ is used, while others utilize traditional Child-Sum Tree-LSTM units. The network's hidden states are computed bottom-up, with the root node's hidden state representing the entire program. Each node's initial input vector x is its word embedding.

4.4. Applications with DSTM

After outlining DSTM's methodology for generating vector representations of C ASTs, we now focus on applying DSTM to real-world tasks: program classification and code clone detection.

4.4.1. Program Classification

The goal here is to categorize C code snippets based on functionality. Assuming M possible classifications, the code's vector representation v is transformed into an M-dimensional prediction $\hat{y}$ through a fully-connected layer:

$$\hat{y} = Wv + b$$

Training involves minimizing cross entropy loss:

$$\text{loss}(\hat{y}, y) = -\log\left(\frac{\exp(\hat{y}_y)}{\Sigma_j \exp(\hat{y}_j)}\right)$$

Here, y is the true class label. For evaluation, class prediction is determined by the maximum value in $\hat{y}$: $\text{argmax}_k \hat{y}_k$.

4.4.2. Code Clone Detection

This task involves assessing whether two code fragments are duplicates. Each fragment is processed by a separate DSTM with shared parameters, yielding representations v_1 and v_2. Their similarity is gauged using cosine similarity $\hat{y} = \frac{v_1 \cdot v_2}{|v_1| \cdot |v_2|}$. True clones are assigned a ground truth value of 1, and non-clones -1. Training employs mean squared error as the loss function:

$$\frac{1}{d} \sum_{i=1}^{d} (y_i - \hat{y}_i)^2$$

During evaluation, a pair is classified as a clone if $\hat{y} > 0$, otherwise not.

5. Experiments

5.1. Datasets

5.1.1. Program Classification

We compiled a dataset from the POJ[1] student programming platform. This dataset, a more comprehensive version than those used in previous studies [33,39,40], categorizes programs by problem-solving task. It includes 58,600 C program files from 293 distinct problems, each represented by 200 files. On average, files contain 22.8 lines of code and are transformed into ASTs with an average of 116.62 nodes. We divided the dataset into training, validation, and test sets in an 8:1:1 ratio. Table 2 presents the frequency of various node types in the dataset, indicating a predominance of nodes with sequential children. `For` and `If` nodes are notably frequent, while `DoWhile` and `Switch` nodes appear less often.

[1] http://poj.org/

Table 2. Distribution of structural AST nodes in the program classification dataset.

ine Node type	Count of appearances
ine `FuncDef`	67,806
ine `While`	13,315
ine `DoWhile`	1,645
ine `For`	102,870
ine `If`	116,152
ine `Switch`	625
ine `Seq`	519,425
ine	

5.1.2. Code Clone Detection

Code cloning is categorized into four types [41], with Type 4 being the most challenging as it involves syntactically different fragments with similar functionalities. Our focus is primarily on Type 4 clones.

For this, we use the OJClone dataset [33], containing 15 diverse programming problems, each with 500 C source code files. The average length of these files is 35.25 lines of code. We label code fragments from the same problem as true clones (Type 3 or 4) and from different problems as non-clones. The dataset is split into training, validation, and testing sets in an 8:1:1 ratio, ensuring no overlap between training and other sets. We also apply up-sampling to balance the distribution of true and false clone pairs in the training set.

Table 3. Classification accuracy for the 293-classes program classification task

ine Method	Accuracy
ine CNN	68.3%
LSTM	80.8%
Bi-LSTM	80.7%
ine Code-RNN	64.8%
TBCNN	79.0%
GGNN	61.0%
Tree-LSTM	85.2%
DSTM-a	**86.5%**
DSTM-b	86.2%
ine DSTM-a w/ id	**92.6%**
DSTM-b w/ id	**92.9%**
ine	

5.2. Experiment Settings

ASTs for C programs are generated using pycparser[2]. This parser distinguishes between identifier and non-identifier nodes, with the former containing type and value information. To focus solely on algorithmic features and exclude problem-specific identifier details, we prune identifier information, retaining only node types for both program classification and clone detection tasks. In the classification task, node embeddings and LSTM hidden states in the DSTM-b model are set to 200 dimensions, while in the clone detection task, they are set to 100. The DSTM-b model for program classification thus contains approximately 4.21 million parameters. To maintain a similar parameter count, the hidden sizes for tree-LSTM and DSTM-a models are adjusted accordingly (720 for tree-LSTM and 320 for DSTM-a in program classification). Models are optimized using ADAM [42] with a learning rate of 0.001. Node embeddings are randomly initialized and learned during training. Models are implemented in PyTorch [3] and trained on an NVIDIA Tesla P100 GPU.

[2] https://github.com/eliben/pycparser
[3] https://pytorch.org/

Batch processing in tree-structured models like DSTM and tree-LSTM is challenging due to the need for building distinct trees for different inputs. We employ manual gradient accumulation for mini-batch optimization, dividing the dataset into batches and accumulating gradients across examples within each batch before optimization. The batch size is set to 32. Although dynamic batching techniques exist, they are not yet supported in PyTorch. Future implementations may leverage such techniques for enhanced efficiency.

5.2.1. Baselines

We compare DSTM against several sequential and tree-structured models in both tasks:

Sequential models: Standard LSTM, bi-directional LSTM, and one-dimensional CNN (following the approach in [43]) are used. ASTs are converted into token sequences using depth-first traversal, as in [20,21].

Tree-structured models: Child-sum Tree-LSTM [22], TBCNN [23], and Code-RNN [32] (modified for abstract syntax trees) are included as baselines.

Graph-based models: Gated graph neural networks (GGNN)[44] are also considered. Following [45], we construct program graphs from ASTs, omitting data flow edges to focus solely on structural aspects.

Additional baselines for code clone detection include Deckard [46], DLC (Deep Learning for Code Clones) [47], and SourcererCC [48], each with its own strengths in detecting various types of clones.

5.3. Experiment Results

In our experiments, we explore the following research questions (RQs) and analyze the results obtained:

5.3.1. How effective is the DSTM in the program classification task?

The performance of our DSTM in program classification is summarized in Table 3. Both DSTM-a and DSTM-b variants outperform all baseline models. This highlights the importance of differentiating between various semantic units in ASTs for neural network models. The inclusion of identifier information further boosts DSTM's accuracy, aligning with our expectations. Interestingly, GGNN shows lower performance compared to even sequential models, potentially due to its limitations in capturing AST node hierarchies.

For statistical significance, we performed the Wilcoxon signed-rank test [49] comparing tree-LSTM with both DSTM-a and DSTM-b. The results, with p-values below 0.05, indicate that DSTM's improvements are statistically significant. The slight difference in average accuracy between DSTM-a and DSTM-b is not statistically significant.

The convergence speed of models is another key aspect we investigated. Faster convergence implies reaching optimal accuracy in fewer epochs, reducing training time. Figure 8 plots the test accuracy over training epochs, showing DSTM models' significant early improvements over Tree-LSTM. DSTM achieves acceptable accuracy earlier and reaches peak performance sooner.

A critical aspect of neural network models in real-world scenarios is their performance with limited training data. We evaluated this by downsampling the training set and keeping the validation and test sets constant. Table 4 displays the impact of reduced training data, with a widening performance gap between DSTM and Tree-LSTM, indicating DSTM's superiority under data constraints.

Table 4. Impact of training data reduction on program classification.

Reduction	Tree-LSTM	DSTM-a	DSTM-b
12.5%	70.0%	72.6%	72.2%

5.3.2. What is the performance of DSTM in clone detection tasks?

In the clone detection task, DSTM models excel in precision, recall, and F1 scores, surpassing all baseline models (Table 5). This superior performance, especially in recall, indicates DSTM's effectiveness in detecting elusive type-4 clones. MTN-b slightly outperforms MTN-a. The comparison of similarity score distributions between DSTM and tree-LSTM (Figure 9) reveals DSTM's more definitive distinction between clone and non-clone pairs, facilitating the selection of a universal threshold.

Table 5. Performance in code clone detection.

ine Method	Precision	Recall	F1
ine Deckard	0.6	0.06	0.11
DLC	0.70	0.18	0.30
SourcererCC	0.97	0.1	0.18
CDLH	0.21	0.97	0.34
CNN	0.29	0.43	0.34
LSTM	0.19	0.95	0.31
Bi-LSTM	0.18	0.97	0.32
Code-RNN	0.26	0.97	0.41
GGNN	0.20	0.98	0.33
Tree-LSTM	0.27	**1.0**	0.43
ine DSTM-a	0.84	0.98	0.90
DSTM-b	**0.86**	0.98	**0.91**
DSTM-a w/ id	**0.91**	**0.98**	**0.95**
DSTM-b w/ id	**0.91**	**0.99**	**0.95**
ine			

The convergence analysis (Figure 10) demonstrates that DSTM models converge to a lower loss compared to Tree-LSTM. Despite a higher loss, tree-LSTM's convergence is confirmed, as increasing its hidden size does not further reduce loss. This trend is also observed in other baselines, suggesting it's not related to model capacity. Precision and F1 score trends further highlight DSTM's learning efficiency.

Table 6. Comparison of DSTM with Tree-LSTM in clone detection.

Model	p@r=0.8	p@r=0.9	p@r=0.95	p@r=0.99	ROC_AUC
Tree-LSTM	0.861	0.831	0.816	**0.677**	0.994
DSTM-a	0.984	0.970	0.934	0.470	0.995
DSTM-b	**0.996**	**0.983**	**0.977**	0.450	**0.997**

5.3.3. What is the contribution of individual DSTM modules?

To dissect the contribution of each DSTM module, we conducted an ablation study on the clone detection task. We selectively replaced DSTM-b units with traditional tree-LSTM units and observed the impact. Our focus was on five frequent MTN units: FuncDef, For, If, While, and Seq. The results (Table 6) reveal that removing the Seq module most significantly affects performance, suggesting its crucial role in understanding statement order. The other units' removal led to minor performance drops, likely due to their fixed subtree structures, which simpler models like tree-LSTM could adequately learn from.

Table 7. Ablation study results for DSTM on clone detection.

Model	precision	recall	f1	ROC_AUC
DSTM-b	**0.859**	0.975	**0.913**	**0.997**
-FuncDef	0.826	**0.991**	0.901	**0.997**
-For	0.775	0.985	0.868	**0.997**
-If	0.779	0.985	0.870	0.996
-While	0.769	0.972	0.859	0.996
-Seq	0.275	**0.991**	0.431	0.995

6. Conclusion and Future Work

This research introduces the Dynamic Syntax Tree Model (DSTM), an advanced modular tree-structured recurrent neural network. DSTM significantly enhances the ability to capture intricate semantic details in source code compared to prior tree-structured neural networks. We applied DSTM to two specific tasks within the realm of C programming: the classification of program functionality and the detection of code clones. Our findings reveal that DSTM demonstrates superior performance over existing sequential and tree-structured neural network models, primarily due to its refined handling of the semantic variances within AST units.

Looking ahead, we plan to adapt DSTM to various other programming languages, broadening its applicability and utility. Additionally, an intriguing direction for future research lies in leveraging DSTM for program generation. Current AST-based program generation models typically construct programs from the root to the leaves, a process not directly compatible with DSTM's architecture. However, DSTM holds potential as an encoder for partially constructed trees in program generation, offering valuable structural context to decoders. This could open new pathways in automated code generation, enhancing both the efficiency and accuracy of generated programs.

References

1. Baker, B.S. A program for identifying duplicated code. *Computing Science and Statistics* **1993**, pp. 49–49.
2. Bellon, S.; Koschke, R.; Antoniol, G.; Krinke, J.; Merlo, E. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering* **2007**, *33*, 577–591.
3. Fei, H.; Ren, Y.; Ji, D. Retrofitting Structure-aware Transformer Language Model for End Tasks. Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, 2020, pp. 2151–2161.
4. Börstler, J. Feature-oriented classification for software reuse. Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering. Knowledge Systems Institute, 1995, Vol. 95, pp. 22–24.
5. Fei, H.; Ren, Y.; Ji, D. Boundaries and edges rethinking: An end-to-end neural model for overlapping entity relation extraction. *Information Processing & Management* **2020**, *57*, 102311.
6. Li, J.; Fei, H.; Liu, J.; Wu, S.; Zhang, M.; Teng, C.; Ji, D.; Li, F. Unified Named Entity Recognition as Word-Word Relation Classification. Proceedings of the AAAI Conference on Artificial Intelligence, 2022, pp. 10965–10973.
7. Wang, S.; Liu, T.; Tan, L. Automatically learning semantic features for defect prediction. Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on. IEEE, 2016, pp. 297–308.
8. Li, J.; Xu, K.; Li, F.; Fei, H.; Ren, Y.; Ji, D. MRN: A Locally and Globally Mention-Based Reasoning Network for Document-Level Relation Extraction. Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021, 2021, pp. 1359–1370.
9. Fei, H.; Wu, S.; Ren, Y.; Zhang, M. Matching Structure for Dual Learning. Proceedings of the International Conference on Machine Learning, ICML, 2022, pp. 6373–6391.
10. Li, J.; He, P.; Zhu, J.; Lyu, M.R. Software defect prediction via convolutional neural network. Proceedings of the 2017 International Conference on Software Quality, Reliability and Security. IEEE, 2017, pp. 318–328.
11. Wu, S.; Fei, H.; Li, F.; Zhang, M.; Liu, Y.; Teng, C.; Ji, D. Mastering the Explicit Opinion-Role Interaction: Syntax-Aided Neural Transition System for Unified Opinion Role Labeling. Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence, 2022, pp. 11513–11521.

12. Shi, W.; Li, F.; Li, J.; Fei, H.; Ji, D. Effective Token Graph Modeling using a Novel Labeling Strategy for Structured Sentiment Analysis. Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2022, pp. 4232–4241.
13. Fei, H.; Zhang, Y.; Ren, Y.; Ji, D. Latent Emotion Memory for Multi-Label Emotion Classification. Proceedings of the AAAI Conference on Artificial Intelligence, 2020, pp. 7692–7699.
14. Wang, F.; Li, F.; Fei, H.; Li, J.; Wu, S.; Su, F.; Shi, W.; Ji, D.; Cai, B. Entity-centered Cross-document Relation Extraction. Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, 2022, pp. 9871–9881.
15. Hindle, A.; Barr, E.T.; Su, Z.; Gabel, M.; Devanbu, P. On the naturalness of software. Software Engineering (ICSE), 2012 34th International Conference on. IEEE, 2012, pp. 837–847.
16. Allamanis, M.; Peng, H.; Sutton, C. A convolutional attention network for extreme summarization of source code. International Conference on Machine Learning, 2016, pp. 2091–2100.
17. Fei, H.; Wu, S.; Ren, Y.; Li, F.; Ji, D. Better Combine Them Together! Integrating Syntactic Constituency and Dependency Representations for Semantic Role Labeling. Findings of the Association for Computational Linguistics: ACL/IJCNLP 2021, 2021, pp. 549–559.
18. Huo, X.; Li, M.; Zhou, Z.H. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code. IJCAI, 2016, pp. 1606–1612.
19. Iyer, S.; Konstas, I.; Cheung, A.; Zettlemoyer, L. Summarizing source code using a neural attention model. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2016, Vol. 1, pp. 2073–2083.
20. Bhoopchand, A.; Rocktäschel, T.; Barr, E.; Riedel, S. Learning Python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* **2016**.
21. Li, J.; Wang, Y.; King, I.; Lyu, M.R. Code Completion with Neural Attention and Pointer Networks. Proceedings of the 27th International Joint Conference on Artificial Intelligence. AAAI Press, 2018, pp. 4159–4165.
22. Tai, K.S.; Socher, R.; Manning, C.D. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), 2015, Vol. 1, pp. 1556–1566.
23. Mou, L.; Li, G.; Zhang, L.; Wang, T.; Jin, Z. Convolutional Neural Networks over Tree Structures for Programming Language Processing. AAAI, 2016, Vol. 2, p. 4.
24. Wu, S.; Fei, H.; Ren, Y.; Ji, D.; Li, J. Learn from Syntax: Improving Pair-wise Aspect and Opinion Terms Extraction with Rich Syntactic Knowledge. Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, 2021, pp. 3957–3963.
25. Fei, H.; Li, F.; Li, B.; Ji, D. Encoder-Decoder Based Unified Semantic Role Labeling with Label-Aware Syntax. Proceedings of the AAAI Conference on Artificial Intelligence, 2021, pp. 12794–12802.
26. Hochreiter, S.; Schmidhuber, J. Long short-term memory. *Neural computation* **1997**, *9*, 1735–1780.
27. Fei, H.; Wu, S.; Li, J.; Li, B.; Li, F.; Qin, L.; Zhang, M.; Zhang, M.; Chua, T.S. LasUIE: Unifying Information Extraction with Latent Adaptive Structure-aware Generative Language Model. Proceedings of the Advances in Neural Information Processing Systems, NeurIPS 2022, 2022, pp. 15460–15475.
28. Fei, H.; Ren, Y.; Zhang, Y.; Ji, D.; Liang, X. Enriching contextualized language model from knowledge graph for biomedical information extraction. *Briefings in Bioinformatics* **2021**, *22*.
29. Hinton, G.E.; Osindero, S.; Teh, Y.W. A fast learning algorithm for deep belief nets. *Neural computation* **2006**, *18*, 1527–1554.
30. Huo, X.; Li, M. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. Proceedings of the 26th International Joint Conference on Artificial Intelligence. AAAI Press, 2017, pp. 1909–1915.
31. Socher, R.; Lin, C.C.; Manning, C.; Ng, A.Y. Parsing natural scenes and natural language with recursive neural networks. Proceedings of the 28th international conference on machine learning (ICML-11), 2011, pp. 129–136.
32. Liang, Y.; Zhu, K. Automatic Generation of Text Descriptive Comments for Code Blocks. AAAI Conference on Artificial Intelligence, 2018.

33. Wei, H.H.; Li, M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. Proceedings of the 26th International Joint Conference on Artificial Intelligence. AAAI Press, 2017, pp. 3034–3040.
34. Allamanis, M.; Brockschmidt, M.; Khademi, M. Learning to represent programs with graphs. International Conference on Learning Representations (ICLR), 2018.
35. Socher, R.; Bauer, J.; Manning, C.D.; others. Parsing with compositional vector grammars. Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2013, Vol. 1, pp. 455–465.
36. Dong, L.; Wei, F.; Tan, C.; Tang, D.; Zhou, M.; Xu, K. Adaptive recursive neural network for target-dependent twitter sentiment classification. Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2014, Vol. 2, pp. 49–54.
37. Arabshahi, F.; Singh, S.; Anandkumar, A. Combining Symbolic Expressions and Black-box Function Evaluations in Neural Programs. International Conference on Learning Representations (ICLR), 2018.
38. Liu, P.; Qiu, X.; Huang, X. Dynamic compositional neural networks over tree structure. Proceedings of the 26th International Joint Conference on Artificial Intelligence. AAAI Press, 2017, pp. 4054–4060.
39. Wei, H.; Li, M. Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training. IJCAI, 2018, pp. 2840–2846.
40. Ben-Nun, T.; Jakobovits, A.S.; Hoefler, T. Neural Code Comprehension: A Learnable Representation of Code Semantics. *arXiv preprint arXiv:1806.07336* **2018**.
41. Roy, C.K.; Cordy, J.R. A survey on software clone detection research. *Queen's School of Computing TR* **2007**, *541*, 64–68.
42. Kingma, D.P.; Ba, J.L. Adam: A method for stochastic optimization. International Conference on Learning Representations (ICLR), 2015, Vol. 5.
43. Kim, Y. Convolutional Neural Networks for Sentence Classification. Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), 2014, pp. 1746–1751.
44. Li, Y.; Tarlow, D.; Brockschmidt, M.; Zemel, R. Gated graph sequence neural networks. International Conference on Learning Representations (ICLR), 2016.
45. Brockschmidt, M.; Allamanis, M.; Gaunt, A.L.; Polozov, O. Generative Code Modeling with Graphs. 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019, 2019.
46. Jiang, L.; Misherghi, G.; Su, Z.; Glondu, S. Deckard: Scalable and accurate tree-based detection of code clones. Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007, pp. 96–105.
47. White, M.; Tufano, M.; Vendome, C.; Poshyvanyk, D. Deep learning code fragments for code clone detection. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM, 2016, pp. 87–98.
48. Sajnani, H.; Saini, V.; Svajlenko, J.; Roy, C.K.; Lopes, C.V. SourcererCC: scaling code clone detection to big-code. 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, 2016, pp. 1157–1168.
49. Wilcoxon, F.; Wilcox, R.A. *Some rapid approximate statistical procedures*; Lederle Laboratories, 1964.