Modular Tree Network for Source Code Representation Learning

WENHAN WANG, Peking University, China
GE LI*, Peking University, China
SIJIE SHEN, Peking University, China
XIN XIA, Monash University, Australia
ZHI JIN*, Peking University, China

Learning representation for source code is a foundation of many program analysis tasks. In recent years, neural networks have already shown success in this area, but most existing models did not make full use of the unique structural information of programs. Although abstract syntax tree-based neural models can handle the tree structure in the source code, they cannot capture the richness of different types of substructure in programs. In this paper, we propose a modular tree network (MTN) which dynamically composes different neural network units into tree structures based on the input abstract syntax tree. Different from previous tree-structural neural network models, MTN can capture the semantic differences between types of AST substructures. We evaluate our model on two tasks: program classification and code clone detection. Our model achieves the best performance compared with state-of-the-art approaches in both tasks, showing the advantage of leveraging more elaborate structure information of the source code.

CCS Concepts: • **Software and its engineering** → *Software creation and management*.

Additional Key Words and Phrases: deep learning, neural networks, program classification, code clone detection

1 INTRODUCTION

Sour code representation learning is a set of techniques that allow a system to automatically discover the representations needed for feature detection or classification from source code. Due to the significant progress in deep learning in recent years, modeling source code by deep neural networks has drawn more and more attention. Deep-learning based source code representation techniques have achieved considerable success in many tasks in program analysis, including program classification [22, 24], defect prediction [32, 37], code clone detection [34, 35], code summarization [3, 16, 22], and malware detection [38].

As Hindle et al. [11] has shown, programming languages have many similar statistical properties with natural language. So when it comes to representing programs using deep learning models,

---

*corresponding author

Authors' addresses: Wenhan Wang, Peking University, China, wwhjacob@pku.edu.cn; Ge Li, Peking University, China, lige@pku.edu.cn; Sijie Shen, Peking University, China, sjshen@pku.edu.cn; Xin Xia, Monash University, Australia, xin.xia@monash.edu; Zhi Jin, Peking University, China, zhijin@pku.edu.cn.

researchers tend to use models that have already shown success in natural language processing rather than building new models for the source code. Some basic approaches [8, 20] split the source code into a sequence of tokens like natural language sentences, where their models include convolutional neural network (CNN) [3, 15] and recurrent neural network (RNN) [8, 16, 20]. However, different from natural languages, whose structure is relatively simple, a programming language contains complex and explicit structural information [1, 24]. These pieces of structural information can be extracted from a program's abstract syntax tree (AST). An example of an AST in C programming language and its corresponding code snippet is presented in Figure 1. An AST consists of a group of semantic units, each one include a non-leaf AST node and all its children. Different types of semantic units have different semantic meaning. For example, In Fig.1, `While` nodes have two children: a condition node and a loop body node, together they build a loop sketch. The children of `BinaryOp` nodes are two operands, and together they form a calculation on identifiers. Each type of semantic unit is defined by a grammar production rule. As the grammar of a programming language contains a finite set of production rules, the total number of semantic unit types in a programming language is also finite. Abstract syntax trees are constructed from different semantic units, which form the complex structure of the source code.
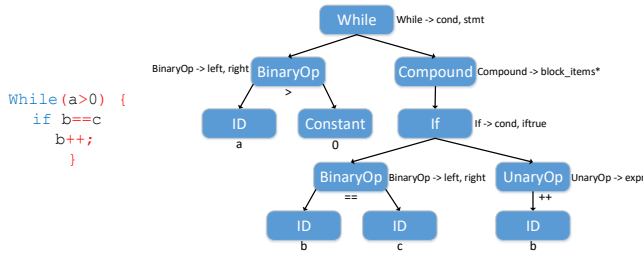

Fig. 1. A simple C code snippet and its corresponding abstract syntax tree fragment. Beside each non-leaf node shows the production rule related to the semantic unit which contains this node and its children. Some nodes, mainly identifiers, contain one or more values, while other nodes only contain a node type.

There have already been several tree-structured neural network models for representing tree-structured data, including Tree-LSTM [31] and TBCNN [24]. Although these models utilize the tree structure of input data, they share a common weakness: they use the same neural network unit to model different kinds of AST semantic units. Although this feature makes these models available for trees from arbitrary data domains, it also prevents them from handling the differences between semantic units. Using only one neural network unit to capture the meaning of all semantic units is apparently not enough, which can cause underfitting.

In this paper, we propose Modular Tree Network (MTN), a dynamically-composed tree-structured neural network framework which models different AST semantic units with different neural network modules. We select a group of semantic units related to program structure based on the grammar of programming languages and design a unique neural module for each of them. For an input AST, the network architecture is constructed with these neural modules based on the specific structure of the AST. Then the vector representation of the input AST is calculated in a bottom-up manner.

We evaluate the performance of our model on two software engineering tasks, i.e., program classification, and clone detection. In program classification, we propose a 293-classes POJ dataset and evaluate our model on it. Results show that our model achieves 1.8% higher accuracy than the state-of-the-art approaches. In clone detection, we compare our model with CDLH on OJClone dataset, and our model improves the F1-score from 0.57 to 0.90.

The main contributions of this paper are as follows:

(1) We propose a tree-structured neural network framework composed of multiple submodules for learning a representation of source code fragment based on its AST.

(2) We employ this model and the representation it learns to complete two tasks on source code analyzing: program classification and code clone detection.

(3) Our proposed model achieves better results compared to previous neural network models, showing that utilizing more detailed semantic information of source code is advantageous. We also show that MTN converges faster than baseline models and can learn better on small datasets.

Paper Organization. The remainder of the paper is organized as follows. We describe the background of deep learning in Section 2. We present the structure of our model MTN in Section 3. We describe the application of MTN on two program analysis tasks in Section 4. We demonstrate and analyze our experiments and results in Section 5. We show the threats to validity in Section 6 and list related works in Section 7. Finally, in Section 8 we make a conclusion on our work.

2 BACKGROUND

In this section, we briefly introduce the background of our model MTN: tree-structured neural networks, especially recursive neural network and tree-LSTM.

2.1 Recursive Neural Network (RvNN)

Recursive neural network (RvNN) is the most basic type of tree-structured neural networks. Suppose a node j in a binary tree has two children $c1$ and $c2$, its representation is computed from its children by:

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

W is a weight matrix, and b is a bias term. Since W is shared across all non-leaf nodes in a tree, recursive neural networks require trees with a definite branching factor, e.g., binary trees. Besides, in recursive neural networks, only leaf nodes take input vectors, so it is mostly used in structures like natural language constituency trees, where the lexical tokens of a natural language sentence are all stored in their leaf nodes.

2.2 Tree-LSTM

Tree LSTM is a generalization of Long short-term memory (LSTM) to tree-structured network topologies [31]. LSTM is a variant of recurrent neural networks (RNNs) proposed to solve the problem of long-term dependencies existed in traditional RNNs. In time step t, a traditional RNN cell calculates the hidden state of the current time step h_t by the input of current time step x_t and the hidden state of the previous time step h_{t-1}, using the below equation:

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

where W and U are different weight matrices for input vector and hidden state. An LSTM unit extends traditional RNN cell with a forget gate f, an input gate i, an output gate o, and a memory cell. The forget gate decides what information of the previous memory cell we choose to forget, the input gate controls the extent of using new information to update the memory cell, and the output gate controls the extent to which the value in the cell is used to compute the hidden state of the LSTM unit. Each gate takes x_j and h_{t-1} with a different set of parameters W, U and b. Tree-LSTM shares these gates and cell with traditional LSTM but for a node j in a tree, instead of taking the hidden state of the previous time step, a tree-LSTM unit takes the hidden states of all of its children and feed a combination of these states into its input and output gate. For the forget gates, a tree-LSTM unit contains multiple forget gates with shared parameters, and each one only receives the hidden state of its associated child.

Under these settings, there exist two versions of tree-LSTM: Child-Sum Tree-LSTM and N-ary Tree-LSTM. We will briefly introduce these two variants and analyze their drawbacks.

2.2.1 Child-Sum Tree-LSTM. In child-sum tree-LSTM, for a node j, we sum up the hidden states of all its children and use this summation as the input hidden state of the input gate and output gate. Child-sum Tree-LSTM can be built on arbitrary tree structures. However, since changing the order of children do not affect the result of the summation operation, its calculation process loses the information of the order of children.

2.2.2 N-ary Tree-LSTM. Similar to recursive neural networks, n-ary Tree-LSTM can only be applied on tree structures where the branching factor is at most N. Its main difference from child-sum tree-LSTM is that n-ary tree-LSTM aims to leverage the order of children. Instead of summing up all children hidden states like child-sum tree-LSTM, n-ary tree-LSTM associate each child with a different weight matrix U. By introducing different parameters for different children, n-ary tree-LSTM can capture more fine-grained semantic information, such as the difference in importance between children of different positions.

When applying n-ary tree-LSTM to trees with indefinite branching factors, like abstract syntax trees, one need to convert them into n-ary trees (often binary). This process not only destroys the original syntax relationships between nodes in the original AST but also make ASTs deeper, which may cause long-term dependency problems.

3 APPROACH

Our proposed model, modular tree network (MTN), refines the basic structure of child-sum tree-LSTM by replacing a part of tree-LSTM units with neural modules designed for different AST semantic units. This idea of building different neural modules for different semantic units can be applied to any programming languages. However, we need to design different neural modules for semantic units in different programming languages, because:

1. Different programming languages may contain different semantic units. E.g., C contains `Case` statements while Python does not have them, Java has `try ... catch` statements while C does not have them.

2. Similar semantic units in different programming languages may have different form. For example, in the AST of C, a `For` statement is defined by the production rule $For \mapsto (init, cond, next, stmt)$, while in Python the production rule for the `For` statement is $For \mapsto (target, iter, body*, orelse*)$ (which "orelse" seldom appears).

In this paper, we only discuss MTN for C as an example, but the proposed model can be generalized to other programming languages by designing new neural modules for different semantic units. The overall framework of MTN is shown in Figure 2. The input of MTN is the AST representation of source code, and the output of MTN is a vector representation of the AST. In the following, we show the general structure of MTN units and define a set of neural modules for semantic feature capturing in C programming language. Then, we show the whole process of applying MTN model to calculate vector representations for ASTs.

3.1 General Structure of MTN Unit

In MTN, we replace the operation of summing up all children hidden states in Child-Sum Tree-LSTM by applying one of our designed neural modules. Given a node j, an MTN unit calculates its hidden state by:
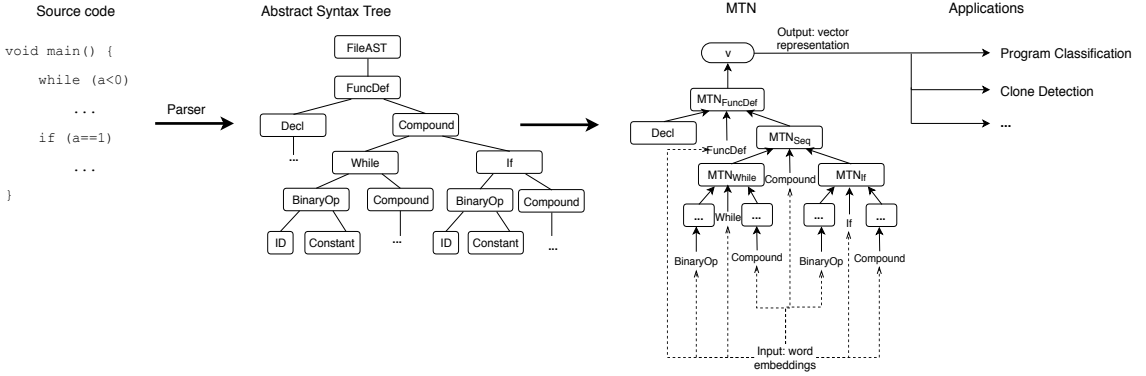
Fig. 2. The overall process of calculating vector representation for source code using MTN.

$$\widetilde{h}_j = F_{type-j}(h_{s_1}, ..., h_{s_{n_j}}) \tag{3}$$

$$i_j = \sigma(W^{(i)}_{type-j}x_j + U^{(i)}_{type-j}\widetilde{h}_j + b^{(i)}_{type-j}) \tag{4}$$

$$f_{jk} = \sigma(W^{(f)}_{type-j}x_j + U^{(f)}_{type-j}h_k + b^{(f)}_{type-j}) \tag{5}$$

$$o_j = \sigma(W^{(o)}_{type-j}x_j + U^{(o)}_{type-j}\widetilde{h}_j + b^{(o)}_{type-j}) \tag{6}$$

$$u_j = \tanh(W^{(u)}_{type-j}x_j + U^{(u)}_{type-j}\widetilde{h}_j + b^{(u)}_{type-j}) \tag{7}$$

$$c_j = i_j \odot u_j + \sum_{k \in C(j)} f_{jk} \odot c_k \tag{8}$$

$$h_j = o_j \odot \tanh(c_j) \tag{9}$$

Here, W, U, b are parameters associated with the type of j and $\odot$ is the element-wise product operator. In equation (3), MTN passes these hidden states through a neural network module designed by us. F_{type-j} is a unique neural module for the node type of j which receives the hidden states for all children of j and produce one vector as an intermediate result. We will describe the details of F_{type-j} in section 3.2. For the sake of time efficiency and implementational complexity, we only design modules for several kinds of semantic units. For the rest kinds of the node without specially designed neural modules, we add up their children's hidden state like Child-Sum Tree-LSTM instead of applying a module F. Equations (4) to (9) are the same as child-sum tree-LSTM, which act as the input gate, forget gates, output gate and the memory cell for the MTN unit.

An MTN unit can be seen as a neural module (equation (3)) wrapped by a Tree-LSTM-like "container" (equations (4)-(9)) which contains a set of gates and memory cells. With MTN hidden size d, a "container" has $8d^2 + 4d$ parameters, which is the same as child-sum tree-LSTM. For the parameters of gates and memory cells in the "containers," we propose two strategies of parameter sharing:

MTN-a: W, U, b (the container parameters) in equations (4)-(7) are shared across all MTN units in a program AST.

MTN-b: We take the idea of using different network architecture and parameters from neural modules to the whole MTN unit.For each type of semantic unit that has a unique F_{type} module, we associate it with a different set of corresponding container parameters W U and b along with different F_{type}. The set of parameters of a particular type of semantic unit are shared among all

occurrences of this semantic unit type. For those subtrees without F_{type}, they share another set of container parameters.

An intuitive demonstration between MTN-a and MTN-b is shown in Figure 3. In this figure, different colored boxes indicate different container parameters.
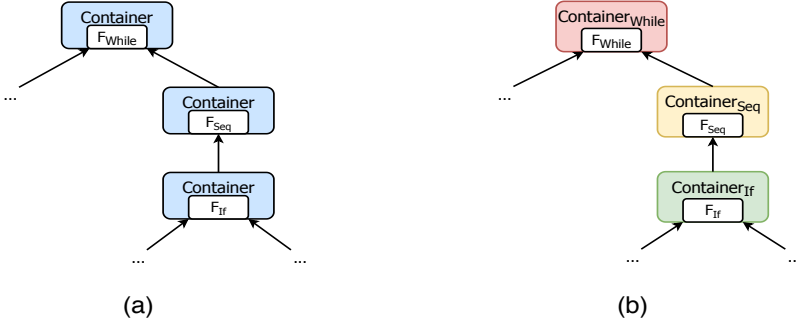

Fig. 3. The two parameter sharing strategies of MTN. (a): MTN-a. (b): MTN-b

3.2 Neural Modules For Different Semantic Units

The semantic units of C can be divided into several categories:

- Units which define logic structure and control flow information, like branches and loops.
- Units contain data information, like a variable declaration.
- Units contain calculation information, like operators and variable assignments in expressions.

In this paper, we mainly focus on semantic units which represent the basic structure of programs. We design 8 different neural network modules with different parameters for different semantic units. First there are 7 kinds of node related to program logic structure: FuncDef, While, DoWhile, For, If, Switch and Case. Apart from these kinds of nodes we mentioned before, there are also a group of nodes which the number of their children is not definite, but their children are a sequence of nodes. We also design a neural module F_{seq} for these nodes. We show the whole set of modules in Table 1 and report the number of parameters of these modules given the hidden size d. In the following part, we will give the implementation of these neural modules and shed some light on the thoughts behind the structure of them.

3.2.1 FuncDef, While, DoWhile and Switch. These four types of nodes are similar in these ways: they all have two children, and the semantics of their left children differs from their right children. So we use a recursive neural network module to model them. In order to address the semantic differences between these four types of nodes, we set different parameters W and b for their modules.
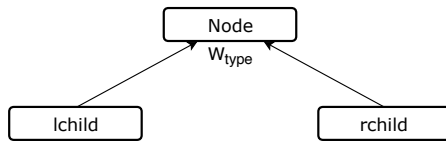

Fig. 4. Structure of FuncDef, While, DoWhile and Switch modules.

3.2.2 For. A `For` node has four children: an initialization statement, a condition statement, a next iteration statement and a loop body. The first three parts form the controller of a loop, so we use an RvNN unit to model them, and feed the intermediate output of this unit and the state of the loop body node to another RvNN unit. Figure 4 shows the structure of our For module.
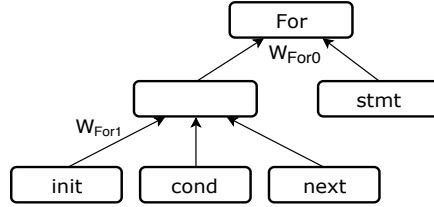

Fig. 5. Structure of For module.

3.2.3 If. Different from previously mentioned AST nodes in C, an `If` node does not always have the same number of Children. An `If` node can have 2 or 3 children, depending on whether it contains an else statement. If an `If` node does not contain an else statement, we simply use a recursive neural network to combine its condition node and branch body node. If an `If` node has an else statement, we separately combine the condition node with the iftrue statement and the iffalse statement with two recursive units with the same parameters. Then we compute the element-wise max for the output of the two recursive units to get the final output of the `If` module.
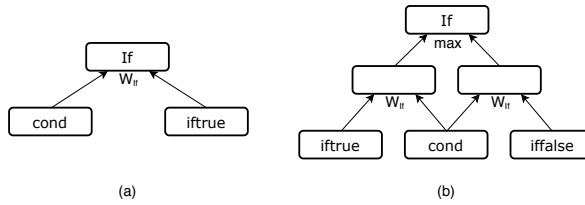
Fig. 6. Structure of If module. (a): without an else statement. (b): with an else statement.

3.2.4 Case. Since Case nodes are a component of switch-case statements, we also design a module for them. The children of a Case node can be divided into two parts: a constant expression which is the label of the current Case branch, and the body of the branch which is a sequence of statements. Since the statements of a Case branch is executed sequentially, we use an LSTM network to encode the root nodes of these statements. The last hidden state of this LSTM and the state of the constant expression node are passed into a recursive NN layer.

3.2.5 Nodes with sequential children. In C AST there are some other types of nodes whose children form a sequence of arbitrary length. A typical example is the Compound node, which denotes a code block, its children are a sequence of statements. We also use an LSTM to encode these children and take the last hidden state of the LSTM as the output.

In summary, for hidden size d, the number of parameters for all types of modules are $33d^2 + 16d$. So an MTN-a model contains $8d^2 + 4d + 33d^2 + 16d = 41d^2 + 20d$ parameters while an MTN-b model has $9(8d^2 + 4d) + 33d^2 + 16d = 105d^2 + 52d$ parameters.

Fig. 7. Structure of Case module.

Table 1. Neural modules for MTN. $LSTM()$ takes a sequence of input and output the last hidden state.

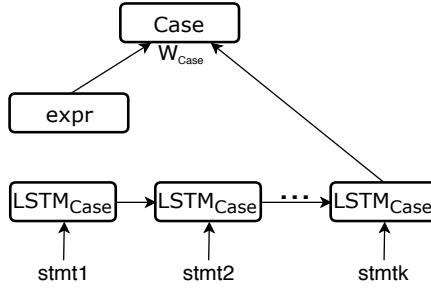| Semantic Unit	Function	Parameters
FuncDef→ decl, body	$h_{FuncDef} = tanh(W_{FuncDef}[h_{decl} : h_{body}] + b_{FuncDef})$	$2d^2 + d$
While→ cond, stmt	$h_{While} = tanh(W_{While}[h_{cond} : h_{stmt}] + b_{While})$	$2d^2 + d$
DoWhile→ cond, stmt	$h_{DoWhile} = tanh(W_{DoWhile}[h_{cond} : h_{stmt}]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

3.3 Calculate representation for code snippets by MTN

In order to calculate vector representation of source code with MTN, we need to parse our code snippets into ASTs first. After we obtain the AST of a code snippet, we built an MTN for it with the same structure of its AST. In an AST of C, each semantic unit only corresponds to a single parent node type so that we can determine the type of a semantic unit only by the parent node in it. So for nodes in {FuncDef, While, DoWhile, For, If, Switch, Case} we place corresponding MTN units in their position. For other nodes, we first check whether their children contains a sequence. If they do, we place MTN_{seq} in the position of the parent of sequences. Else we place a traditional Child-Sum Tree-LSTM unit in their position. After we finish the building of MTN, we can calculate the hidden state of each node in a bottom-up way, and the hidden state of the root node is engaged as the representation of the whole program. The input vector x at every node is their initial word embedding.

4 APPLICATIONS

In the above section, we described the general process of calculating the vector representation of a C AST by MTN. In real application scenarios, we first use training data and ground truth labels to train MTN networks, then evaluate testing data on our trained mode. In this section we show the training and evaluation process of MTN on two program analysis tasks: program classification and code clone detection.

4.1 Program Classification

For the program classification task, we meant to classify a C code snippet by its functionality. Suppose we aim to classify source code into M classes, after we get the code vector representation v, we calculate the M-dimension predicted result $\hat{y}$ by applying a fully-connected neural network

layer:

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

When training, we use the cross entropy loss function:

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

Where y is the ground truth class label, an M-dimension one-hot vector. During evaluation, we predict the class label by picking the dimension with the biggest value from $\hat{y}$, i.e. $\text{argmax}_k \hat{y}_k$.

4.2 Code Clone Detection

The core task of code clone detection is to determine whether two code fragments are duplicate. In this case, these two code fragments are passed through two separate MTNs with shared parameters. After we get their representation v_1 and v_2, we measure their relatedness by compute their cosine similarity $\hat{y} = \frac{v_1 \cdot v_2}{|v_1| \cdot |v_2|}$. Because $\hat{y} \in [-1, 1]$, we set the ground truth of true clone pairs are set to 1, and -1 for false clone pairs. When training, the loss is computed by mean squared error:

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

Which is reduced to squared error since the dimension d of $\hat{y}$ is 1. During the evaluation, we predict whether a pair of code fragments is a clone by:

$$isClone = \begin{cases} True & \hat{y} > 0 \\ False & \hat{y} \leq 0 \end{cases} \tag{10}$$

5 EXPERIMENTS

5.1 Datasets

5.1.1 Program Classification. Our dataset is collected from an online student programming website POJ[1]. A smaller version of this dataset is already utilized by multiple tasks [7, 33, 34]. We consider programs to solve the same problem belong to the same class. Our dataset contains 58,600 C program files from 293 programming problems, each of which contains 200 files. On average each file contains 22.8 lines of code and 116.62 AST nodes when transferred into AST. We split these files by 8:1:1 as training, validation and test set. Table 2 shows the quantity of several kinds of nodes in the whole set. We find that the number of nodes with sequential children far more exceeds other structural nodes. `For` and `If` nodes have a very high frequency of appearances, while the counts of `DoWhile` and `Switch` nodes are rather small.

Table 2. Statistics of our program classification dataset on the appearances of different structural AST nodes.

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

---

[1]http://poj.org/

5.1.2   Code Clone Detection. Code clones can be categorized into four types [26]: Type 1: Identical fragments except for variations in whitespace, comments, and layout. Type 2: Identical fragments except for variations in identifier names and literal values. Type 3: Syntactically similar fragments that differ at the statement level. The fragments have statements added, modified, or removed. Type 4: Syntactically dissimilar fragments that implement the same functionality. Type 4 clones are the hardest to detect, so we mainly focus on type 4 clones in our experiment.

In order to show the advantage of MTN for detecting type-4 clones, we engage the OJClone dataset [34] for the clone detection task. OJClone contains 15 different programming problems, and each problem contains 500 source code files in C. The average lines of code (LoC) in OJClone is 35.25[34]. We consider that two different code fragments from the same problem are a true type-3 or 4 clone pair, while two code fragments from different problems are a false clone pair. We split all 7,500 code fragments by 8:1:1 for training, validation, and testing to guarantee the code fragments in validation and test set do not overlap with those in the train set. As there are far more false clone pairs than true clone pairs, we apply up-sampling to the true clone pairs in the training set. For the training set, we randomly sample the same number of 20,000 positive clone pairs and negative pairs from the training code fragments. For the validation and test set, we each sample 5,000 pairs from their set of code fragments no matter they are positive or negative clone pairs.

5.2   Experiment Settings

We apply pycparser[2] to generate ASTs for C programs. In pycparser, there are two types of nodes: identifier node and non-identifier node. Non-identifier nodes only contain a type, while identifier nodes contain a type and at least one (most times only one) value. In our dataset, the names and values of identifiers used by programmers are highly related to the problem description. Therefore, these details may provide extra benefits for classifying programs. Since we want to classify programs only by their algorithms, this information of identifiers needs to be pruned. So in our default settings, we only keep the node type as the input for both the program classification and the clone detection tasks. In order to show the effect of identifiers, we also add the results for MTNs when names or values are used as the input of identifier nodes.

In the program classification task, we set the dimension of node embeddings, LSTM hidden states of MTN-b model to 200. In the clone detection task, we set them to 100. The MTN-b model for program classification contains $105 \times 200^2 + 52 \times 200 \approx 4,210K$ parameters. For tree-LSTM and MTN-a, we use larger hidden sizes so that their number of parameters are similar to MTN-b (e.g. to compensate the parameter differences, in the program classification task we set the hidden size of tree-LSTM to 720 and MTN-a to 320). We use adaptive moment estimation (ADAM) [19] optimizer with a learning rate of 0.001 to train our model. For all models, the node embeddings are initialized with random-value vectors and learned through the training process. We implement our model in PyTorch [3] and train the models on an NVIDIA Tesla P100 GPU.

For tree-structured neural network models like MTN and tree-LSTM, batch processing is difficult since we need to build different trees for different input data. However, we still want to optimize our model with mini-batches, so we perform manually gradient accumulation. In manual gradient accumulation, we first divide our dataset into batches, which is the same as conventional batch processing. For each input example in a batch, we build a network following the input AST structure, feed the input example to it and calculate the loss. Then we manually accumulate the gradient computed by each example and perform optimization on the accumulated gradient. In our experiment, we set the training batch size to 32. Currently, there exist some dynamic batching

---

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

techniques which can batch dynamic neural networks by putting instances of the same operation together [25], but so far PyTorch does not support these techniques. In the future, if we require a higher running speed of MTN, we can reimplement our models with other deep learning libraries that support dynamic batching.

5.3 Baselines

In both tasks, we compare our MTNs to several sequential and tree-structured neural network models:

Sequential models: we employ a standard LSTM network, a bi-directional LSTM, and a one-dimensional CNN as our baselines. For CNN, we follow the approach in [18], which used multiple filter window sizes (3,4,5). In all these sequential models, we apply depth-first traverse to convert an AST into a sequence of tokens as input for the model, the same as [8, 20].

Tree-structured models: we take child-sum Tree-LSTM [31] and TBCNN [24] as our baselines. We also compare our results to Code-RNN [22], a modified recursive neural network which is originally used to generate comments for source code. Slightly different from [22], we employ abstract syntax trees for input data instead of parse trees (concrete syntax trees).

Graph-based models: we also compare our models with gated graph neural networks (GGNN)[21]. GGNN has already been used in several programming language-related tasks such as variable naming and identifying variable misuse [2, 9]. In our experiment we basically follow the approach in [9] to build program graphs on ASTs. As our approaches do not consider the identifier values, we omit the data flow edges and only use Child, NextToken and NextSib edges.

For the code clone detection tasks, we also include the baselines used by [34]:

Deckard [17]: a popular AST-based clone detection tool based on a characterization of subtrees with numerical vectors.

DLC (Deep Learning for Code Clones): a deep learning-based approach proposed by White et al. [35]. It converts ASTs into binary trees and uses recursive neural networks to represent them. Unlike [35], we train the model with supervised clone labels, same as MTNs.

SourcererCC [27]: a token-based clone detector that achieved strong recall and precision on type 1-3 clones.

5.4 Research Questions and Results

We investigate these following research questions by showing and analyzing our experiment results:
RQ1: How does our model perform in the program classification task?

Our results for the program classification task are shown in Table 3. Our models, MTN-a and MTN-b, both achieve results better than all sequential and tree-structured baselines. This indicate that in addition to tree structures, modelling differences among semantic units in ASTs can provide more insightful information to neural network models. After applying identifier information, the accuracy of MTN is further improved, which is consistent with our anticipation. An unexpected finding is that GGNN performs poorly on the program classification task, its accuracy even lower than sequential models. This is likely because the GGNN model cannot well handle the hierarchical relationship between AST nodes.

In the program classification task, we run tree-LSTM and each MTN model for 10 times and report their average accuracy. To determine whether the accuracies of MTNs are significantly higher than tree-LSTM, we run the Wilcoxon signed-rank test [36] on tree-LSTM with both MTN-a and MTN-b. In both tests the p-value is smaller than 0.05, meaning that our improvement is significant. Although the average accuracy of MTN-a is slightly higher than MTN-b, our significance test shows that there is not a significant difference between their performances.

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

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

Next, we compare the converging process among models on the program classification task. A model that converges faster can reach maximum accuracy in fewer training epochs, thus indirectly reduce its training time. We display the convergence of our model by recording the test accuracy after each epoch of training. Figure 8 illustrates the changing process of test accuracy during the training of MTN and Tree-LSTM. During the first epochs, MTN models show substantial improvements over Tree-LSTM. Compared to Tree-LSTM, MTN models can achieve an acceptable result in the early phase of training and reach maximum accuracy sooner.
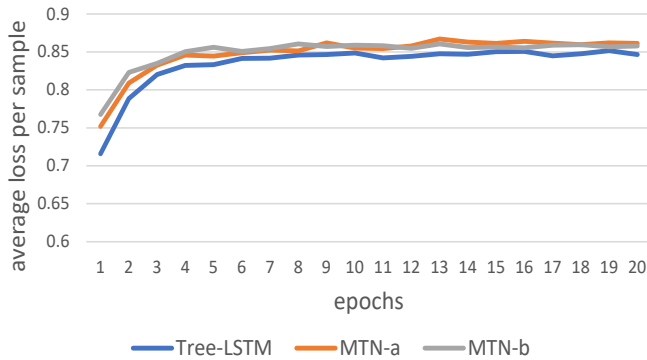
Fig. 8. Test accuracy after every epoch of training in the program classification task.

In real-world applications, it is often difficult to acquire a large number of programs with labeled categories to train neural network models, so we make further analysis on the performance of our models with fewer training data. We down-sample the training set by certain proportions and keep the validation and test set unchanged. The results of different down-sampling proportions are shown in Table 4. We can observe that when we shrink the training set, the performance gap between MTNs and Tree-LSTM becomes larger. This phenomenon indicates that MTNs are more advantageous for fewer training data.

RQ2: How does our model perform in the clone detection task?

Table 5 shows the result of the code clone detection task. For this task, we evaluate the precision, recall, and F1 score, where we set true clone pairs as positive samples. Similar to the program

Table 4. Results of the program classification task when we sub-sample the training set by different proportions.

Down-sampling proportion	Results		
	Tree-LSTM	MTN-a	MTN-b
12.5%	70.0%	72.6%	72.2%
25%	77.1%	79.3%	78.4%
50%	82.0%	83.2%	83.5%

Table 5. Precision, recall and F1 score of the code clone detection task.

Method	Precision	Recall	F1
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
MTN-a	0.84	0.98	0.90
MTN-b	**0.86**	0.98	**0.91**
MTN-a w/ id	**0.91**	**0.98**	**0.95**
MTN-b w/ id	**0.91**	**0.99**	**0.95**

classification task, both MTN-a and MTN-b also outperform all the baselines in F1 scores. MTN models significantly outperform non-neural-network baselines in terms of recall, mainly because that MTN models are effective at detecting type-4 clones where the baselines failed. When compared to neural network models, MTNs achieve remarkable promotion in precision compared to most baselines. The performance of MTN-b is slightly higher than MTN-a. It is worth noting that most neural network baselines (e.g. Tree-LSTM) have unbalanced precision and recall. To find out the reason behind this, we take a look at the learned similarity scores between code pairs in Tree-LSTM and MTN. Figure 9 shows the distribution of similarity scores predicted by MTNs and tree-LSTM on the OJClone test set. We can see that for positive clone pairs, the prediction of tree-LSTM and MTNs are similar. However, for negative clone pairs, the behavior between tree-LSTM and MTN are highly different. While the predictions of MTN-a and MTN-b are close to -1, the predictions of tree-LSTM are much higher, which most of them fall in the interval of [-0.4,0.2]. Although we can improve the precision of tree-LSTM by increasing the threshold between positive and negative clones, the threshold must be chosen delicately since the gap of scores between positive and negative pairs are very small (in this case [0.4,0.6]). The selection of a data-specific threshold can be difficult if we do not have a proper validation set. On the other hand, the predictions of MTNs are close to 1 and -1, meaning that a simple choice of threshold (here 0) can be data-independent.
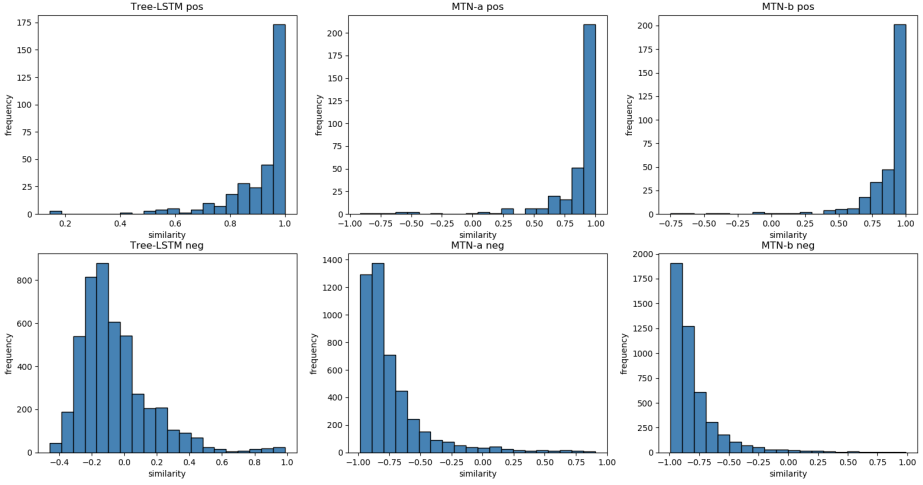
Fig. 9. The distribution of similarity scores predicted by MTNs and tree-LSTM for the positive and negative clone pairs.
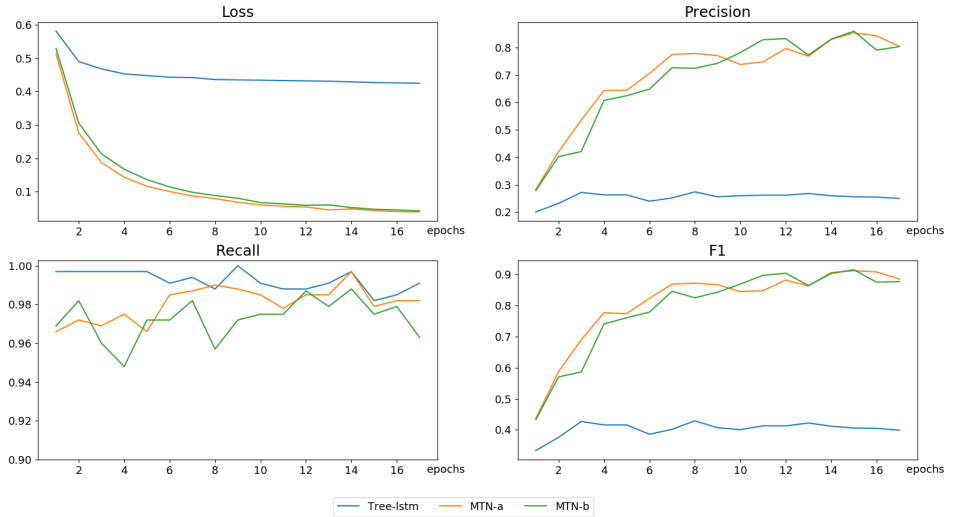
Fig. 10. The training loss and the test set precision/recall/f1 score of MTN-a, MTN-b and tree-LSTM during the training procedure of the clone detection task.

For the code clone detection task, we also analyze the convergence of MTN models. Figure 10 shows the average training loss per sample and the precision/recall/F1 score of the validation set during the training procedure. We can observe that compared to Tree-LSTM, MTNs can converge to a much lower loss. Although the loss of tree-LSTM is higher than MTNs, we make sure that tree-LSTM has come to convergence since, in our clone detection experiments, the loss of tree-LSTM

finally drops to around 0.4 and does not drop further even if we continue to increase its hidden size (e.g., we run tree-LSTM on clone detection with hidden size 720, which is the same as the program classification task and the parameter size is nearly 4 times as the tree-LSTM for clone detection reported in this paper). Moreover, this higher loss than MTNs also exists in several other baselines, including both sequential models like LSTM and more powerful models like Code-RNN and GGNN, which further prove that this phenomenon is unlikely relevant to model capacity. The differences in learning speed are also indicated by the changing process of precision and f1.

Table 6. Detailed results for MTNs compared to Tree-LSTM on clone detection task.

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
MTN-a	0.984	0.970	0.934	0.470	0.995
MTN-b	**0.996**	**0.983**	**0.977**	0.450	**0.997**

To have a more comprehensive understanding of the difference between our model and baselines on the clone detection tasks, we further compare MTNs with the best-performed baseline tree-LSTM on several other metrics, including the precision values for specific recalls, and area under ROC (Receiver-Operating Characteristic) curve. Table 6 shows the precision values for different recalls and the ROC_AUC score for three models. The MTN-b achieves the highest ROC_AUC among three models, and generally, its precision is higher than baseline models except for extreme circumstances when the recall is close to 1. We also draw the precision-recall curve for these three models in Figure 11. We can clearly see that when the precision is high (>0.8), the recall of MTN-a and MTN-b are significantly higher than tree-LSTM.
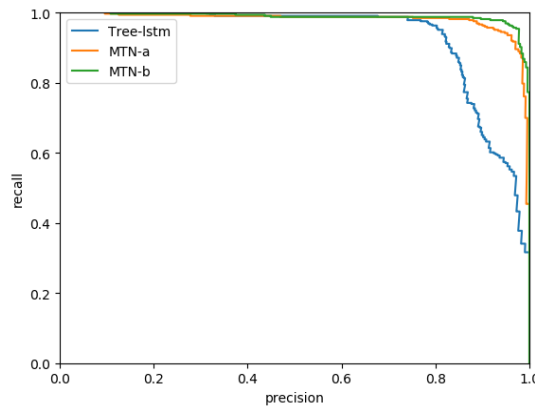
Fig. 11. The precision-recall curve of MTNs and tree-LSTM on OJClone test set.

Additionally, we make a visualization of the embeddings of code snippets learned by MTN through clone detection. Figure 12 shows the distribution of 15 classes of programs in the OJClone dataset. We can find out that for programs from the same question, their embeddings are similar, and the boundaries between most questions are clear. So we can believe that MTN can learn semantic-aware embeddings for source code snippets even without direct class labels, which shows the potential of combining MTN with unsupervised learning.

Then, we will display some example clone pairs to show the capability of MTN intuitively:
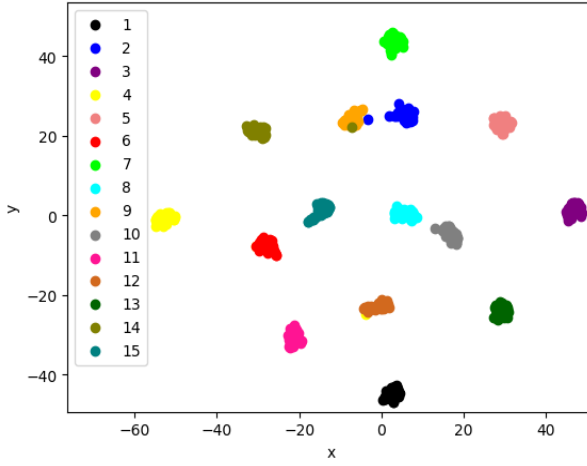
Fig. 12. A 2D visualization of code embeddings learned by MTN-b in the clone detection task using t-SNE.

Figure 13 shows 3 false clone pairs in OJClone that are wrongly predicted as positive by Tree-LSTM. We can see that when two code fragments have some similar patterns like they both include two function declarations (a), or both contains If statements inside loops (b,c). From we previously mentioned, compared to MTNs, Tree-LSTM has a slightly higher recall and a much lower accuracy. So for code pairs with similarities Tree-LSTM have a higher chance to predict them as a clone, but MTNs can more accurately capture their semantics and give the correct prediction.

Figure 14 shows a true clone pair which both Tree-LSTM and MTN-b predicted as false. Although these two programs solved the same question, their structure and algorithm are quite different. Even a human reader cannot quickly determine that they implemented the same functionality.

Figure 15 shows a false clone pair which both Tree-LSTM and MTN-b predicted as true. Although the questions they aimed to solve are different, these two programs both involve structure and linklist operations. After a quick peeks, a human reader may find that they are very similar in structure.

Table 7. Ablation results for MTN-b on the clone detection task.

Model	precision	recall	f1	ROC_AUC
MTN-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

RQ3: How does each module of MTN contribute to our results?

To analyze the contribution of each different MTN unit in our models, we perform an ablation study on the clone detection task. Each time we separately remove an MTN unit from our MTN-b model and replace it by a traditional child-sum tree-LSTM unit. We focus on five different MTN

```
int f(int a,int x)
{
  int i,j;
  if(x==2)
  return 1;
  else
  {
  int result=1;
  for(i=a;i<=x;i++)
  if(x%i==0&&x/i>=i)
  result=result+f(i,x/i);
  return result;
  }
}
main()
{
  int i,j,k;
  int n;
  scanf("%d",&n);
  for(i=0;i<n;i++)
  {
  int a;
  scanf("%d",&a);
  printf("%d\n",f(2,a));
  }
}
```

```
int main()
{
    int jvzhen();
    int n,i,a;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        a = jvzhen();
        printf("%d\n",a);
    }
    return 0;
}
int jvzhen()
{
    int m,n,i,j,sum;
    sum=0;
    scanf("%d %d",&m,&n);
    int b[100][100];
    for(i=1;i<=m;i++){
        for(j=1;j<=n;j++)
            scanf("%d",*(b+i)+j);
    }
    for(i=1;i<=n;i++)
        sum=sum+*(*(b+1)+i);
    if(m>1){for(i=1;i<=n;i++)
        sum=sum+*(*(b+m)+i);}
    if(m>2) for(j=2;j<m;j++)
    sum=sum+*(*(b+j)+1)+*(*(b+j)+n);
    }
    return(sum);
}
```

(a)

```
int main(){
    double n;
    int k,i,j=0,l;
    char s[501];
    char c[501];
    scanf("%lf",&n);
    scanf("%s",s);
    scanf("%s",c);
    k=strlen(s);
    l=strlen(c);
    for(i=0;s[i];i++){
        if(k!=l||
(s[i]!='A'&&s[i]!='G'&&s[i]!='C'&&s[i]!='T')||
(c[i]!='A'&&c[i]!='G'&&c[i]!='C'&&c[i]!='T')){
            j=-1;
            break;
        }
        if(0==s[i]-c[i]){
            j++;
        }
    }
    if(n<=1.00*j/k&&j!=0){
        printf("yes");
    }else if(j==-1){
        printf("error");
    }else{
        printf("no");
    }
    return 0;
}
```

```
main(){
int row,col,n,sum,a,b,c,d,s;
long pic[100][100];
scanf("%d\n",&n);
for(row=0;row<n;row++){
    for(col=0;col<n;col++)
    {scanf("%d",&pic[row][col]);}
    }
sum=0;
for(row=0;row<n;row++)
    for(col=0;col<n;col++)
    {if ((pic[row][col]==0)
        &&(pic[row-1][col-1]==255)
        &&(pic[row-1][col]==255)
        &&(pic[row][col-1]==255))
        {a=row;
        b=col;}
    }
for(row=n-1;row>=0;row--)
    for(col=n-1;col>=0;col--)
    {if ((pic[row][col]==0)
        &&(pic[row+1][col+1]==255)
        &&(pic[row+1][col]==255)
        &&(pic[row][col+1]==255))
        {c=row;
        d=col;}
    }
s=(c-a-1)*(d-b-1);
printf("%d",s);
return 0;
}
```

(b)

```
int main()
{
    int i,j,t=0,k,m;
    int a[100][100];
    for(j=0;j<100;j++)
        {
        scanf("%d",&a[t][j]);
        if(a[t][j]==0) {t=t+1;}
        if(a[t][j]==-1)  {break;}
    }
    for(k=0;k<t;k++)
        {
        m=0;
        for(i=0;i<100;i++)
            {
            for(j=0;j<100;j++)
                {
                if(a[k][i]==2*a[k][j]&&a[k]
[i]!=0)
                m++;
            }
        }
        printf("%d\n",m);
    }
    return 0;
}
```

```
int main()
{
    int n;
    int a[91];
    int i;
    for( i=0 ; i<91 ; i++ )
    {
        a[i]=0;
    }
    cin>>n;
    cin>>i;
    cout<<i;
    a[i-10]=1;
    while( n>0 )
    {
        n--;
        int b;
        cin>>b;
        if( a[b-10]==0 )
        {
            cout<<' '<<b;
            a[b-10]=1;
        }
    }
}
```

(c)

Fig. 13. Examples of false clone pairs which MTN-b correctly predicted but Tree-LSTM failed.

units which have higher frequencies than other units: $MTN_{FuncDef}$, MTN_{For}, MTN_{If}, MTN_{While} and MTN_{Seq}. Table 6 shows the result of our ablation study. We can observe that removing MTN_{Seq} has the biggest impact on the performance of MTN-b, and its precision and f1 only achieve slightly higher than tree-LSTM, implying that handling the order of statements and expressions are the key of MTN's improvements. For other MTN units, removing one of them only causes a slight drop in f1 or roc_auc. This is probably because that the structures of subtrees corresponding to these units are fixed (e.g., an If node always has a condition child and a loop body child), so a simpler model (tree-LSTM) can already learn enough information from these structures, and the effect of MTN units are limited.

RQ4: What are the time and memory efficiency of MTN?

We compare the time and memory efficiency of MTN models with tree-LSTM since they are all tree-structured models. In table 8, the first two rows show the running time of performing an epoch

```
int main(int argc, char* argv[])        int main()
{                                        {
  int i,j,C[1000][1000];                   int a,is1[4000],is2[4000],s,n;
  int n,s;                                 int k = 0;
  int a,b;                                 cin >> n;
  scanf("%d",&n);                          for (int i = 0; i < n * n;i++)
  for(i=0;i<n;i++)                         {
    for(j=0;j<n;j++)                           cin >> a;
      scanf("%d",&(C[i][j]));                   if (a == 0)
  for(i=0;i<n;i++){                           {
    int sum=0;                                   is1[k] = i / n;
    for(j=0;j<n;j++){                             is2[k++] = i % n;
      if(C[i][j]==0)                           }
        sum = sum +1;                        }
      if(sum>2)                            cout << (is1[k - 1] - is1[0]-1)*
        a=sum;}}                         (is2[k - 1] - is2[0]-1);
  for(j=0;j<n;j++){                         return 0;
    int um=0;                            }
    for(i=0;i<n;i++){
      if(C[i][j]==0)
        um = um +1;
      if(um>2)
        b=um;}}
  s=(a-2)*(b-2);
  printf("%d\n",s);
  return 0;
}
```

Fig. 14. An example of a true clone pair which both Tree-LSTM and MTN-b predicted as false.

of training and testing on our clone detection task, and the third row shows the GPU memory cost during training.

Table 8. The time and GPU memory cost for MTN and tree-LSTM in our clone detection task.

	Tree-LSTM	MTN-a	MTN-b
Training time (s)	28871	29993	29366
Test time (s)	1720	1810	1793
GPU memory cost (MB)	743	731	721

We can see that when equipped with equal numbers of parameters, the training and testing time of MTNs are longer than Tree-LSTM only by a tiny scope (less than 6%). Within two MTN models, the running speed of MTN-b is faster than MTN-a, which means that a larger hidden size of MTN-a brings more computational cost than MTN-b. When we look at the memory cost, the ranking for these three models is tree-LSTM > MTN-a > MTN-b. Since larger hidden size leads to larger intermediate tensors, this phenomenon fits our expectations. Generally, with the same number of parameters, MTN-b is more time and memory-efficient than MTN-a, and both models are nearly equally efficient to tree-LSTM.

6 THREATS TO VALIDITY

Threats to internal validity relates to the implementation of the CDLH baseline. In the paper of CDLH the authors did not give the specification of their model, so we reimplemented their model using PyTorch. The hyperparameters of CDLH is the same as our MTN models.

Threats to external validity relates to the dataset we used and the generalizability of our model. Because the authors of [34] did not give the specification of their OJClone dataset, we reproduced all their baselines and CDLH on our OJClone dataset. Still our approach improves them by a substantial

```
int n;
struct student {
  int a;
  char num[20];
  struct student *next;
};
struct student *creat(){
  struct student
*head,*headtwo,*p1,*p2,*p3,*p4;
  p1=(struct student *)malloc(len);
  scanf("%s%d",p1->num,&p1->a);
  head=(struct student *)malloc(len);
  head->next=p1;
  p2=head;
  p1->next=NULL;
  headtwo=(struct student *)malloc(len);
  p4=headtwo;
  p4->next=NULL;
  int i,t=0;
  for (i=0;i<n-1;i++){
    t=0;
    p3=(struct student *)malloc(len);
    scanf("%s%d",p3->num,&p3->a);
    if(p3->a>=60){
      p2=head;
      p1=head->next;
      while(p1!=NULL){
        if(p3->a>p1->a){
          p3->next=p1;
          p2->next=p3;
          t=1;
          break;}
        p2=p1;
        p1=p1->next;}
      if(t==0) {p2->next=p3;p3->next=NULL;}
    }
    if(p3->a<60){
      p4->next=p3;
      p4=p3;
      p4->next=NULL;}}
  p1=head;
  while(p1->next!=NULL)
    p1=p1->next;
  p1->next=headtwo->next;
  return head;
}
void main(){
  scanf("%d",&n);
  struct student *head,*p;
  head=creat();
  p=head->next;
  while (p!=NULL){
    printf("%s\n",p->num);
    p=p->next;}
}
```

```
struct Student
{
  int number;
  int math;
  int chinese;
  int total;
  struct Student *next;
};
void main()
{
  int i=0,n,max;
  struct Student *linkHead, *linkTail,
*student,*ppre,*pmax,*premax;
  scanf("%d",&n);
  linkHead = linkTail = NULL;
  while(i<n){
    student = (struct Student *)malloc(Len);
    scanf("%d %d %d",&student->number, &student-
>math, &student->chinese);
    student->total=student->math+student->chinese;
    student->next = NULL;
    if (linkTail == NULL)
      linkHead = linkTail = student;
    else {
      linkTail->next = student;
      linkTail = student;
    }
    i++;
  }
  i=0;
  while(i<3){
    student=linkHead;
    ppre=student;
    max=0;
    pmax=linkHead;
    premax=linkHead;
    while(student != NULL ){
      if(student->total>max){
        max=student->total;
        pmax=student;
        premax=ppre;
      }
      ppre=student;
      student=student->next;
    }
    printf("%d %d\n", pmax->number, pmax->total);
    if(pmax==linkHead)
      linkHead=student->next;
    else premax->next=pmax->next;
    i++;
  }
}
```

Fig. 15. An example of a false clone pair which both Tree-LSTM and MTN-b predicted as true.

margin. Since our approach mainly works for C-language, we evaluate it using OJClone. However, the idea of MTN is to build different neural modules for different grammar production rules, we can extend our model to OO-languages by building neural modules for OO production rules. In the future, we'd like to evaluate MTN on some OO-language benchmarks like BigCloneBench [30].

7 RELATED WORK

Source code representation learning is a task that gets a distributed representation of a source code snippet through machine learning. In the field of source code representation learning, most models obtain distributed representations in a supervised manner. Supervised source code representation models are trained by tasks with labeled data, such as program classification, defect prediction, clone detection, and code summarization, and an intermediate result is obtained as the representation.

In this section, we first take a brief view of previous approaches in learning source code representations. In most of these tasks, the proposed models come from the natural language processing community. The neural network models for source code modeling can be divided into three categories: sequential, hierarchical and tree-structured models. Then we show some previous work on modeling different semantic compositions in non-programming language tree structures, which is slightly related to our work, and show their insufficiency for working on ASTs.

7.1 Sequential Models

Like natural language, source code snippets can be treated as sequences of tokens. Most investigations employ lexical analyzers to decompose source code into lexical tokens. Iyer et al. [16] proposed Code-NN, which uses a Long Short-Term Memory (LSTM) [13] network with attention to generate natural language descriptions from C# code snippets and SQL queries. Allamanis et al. [3] applied a multi-layer convolutional neural network with attention on Java source code to generate short, name-like summaries. Wang et al. [32] engaged deep belief networks [12] for defect prediction in several open source Java projects. Apart from models based on sequences of lexical tokens, [8, 20] serialized abstract syntax tree by depth-first traversal and applied recurrent neural networks to learn representations of code snippets for code completion. Although these sequential models were easy to implement and often fast to execute, they abandoned the explicit higher-level structural information of source code.

7.2 Hierarchical Models

Slightly more complicated than sequences, programs can also be represented as two-level hierarchies. Each program statement, often taking up one line of code, is a sequence of lexical tokens, and the whole program is a sequence of lines of code. Huo et al. [15] used one-dimensional convolutional neural networks for both within-line and cross-line modeling for program bug localization, while Huo et al. [14] applied CNN for within-line and LSTM for cross-line on the same task. Although these models utilized the sequential structure between lines of code, they failed to capture some kinds of logic structure within code lines, because lines of code in some program structures like loop and branches are not executed sequentially.

7.3 Tree-structured Models

Most tree-structured neural models for source code are built on programs ASTs. The basic component of these tree-structured models is a neural network unit which calculates the representation of a semantic unit. To represent tree structures, Socher et al. [29] proposed recursive neural networks. In recursive neural networks, the vector embedding of a non-leaf node is calculated by its children with a fully-connected neural network, and the embedding of the root node is considered as the representation of the whole tree. However, traditional recursive neural networks require the trees to be binary, while the ASTs of source code are not. One way to apply recursive neural networks to ASTs is converting ASTs into binary trees [35], but the converting process make the ASTs much deeper and the model much harder to train. To adapt recursive neural networks for source code, Liang et al. [22] proposed Code-RNN, which operates on Java parse trees to learn source code representations for program classification and comment generation. Tai et al. [31] proposed Tree-LSTM, a tree-structured recursive neural network framework augmented with LSTM units. In a basic Tree-LSTM unit, hidden states of child nodes are passed into an LSTM-like NN unit and combined with the token embedding of the parent node to get the hidden states of the parent node. Tree-LSTM contains two types of model: Child-Sum Tree-LSTM and N-ary Tree-LSTM. Child-Sum Tree-LSTM can be built on trees with an indefinite branching factor, but it does not consider the order of children. N-ary Tree-LSTM is capable of capturing the order of children, but it requires that the branching factor of the tree must be known. Wei et al. [34] converted ASTs into binary trees, then applied binary Tree-LSTMs to detect code clones in Java and C. Mou et al. [24] introduced tree-based convolutional neural network (TBCNN), whose basic unit is a single dynamic convolution kernel over all semantic units of an AST. Unlike recursive networks, TBCNN did not calculate representations bottom-up, but by first calculating representations of semantic units separately, then applying pooling over all semantic units representations to get the final

representation. They evaluated their model on tasks of program classification and bubble sort detection. These tree-structured networks all use the same neural unit with the same parameters for different semantic units. Apart from tree-structured models on ASTs, Allamanis et al. [2] augmented ASTs with edges of variable calls into a graph, then built gated graph neural network [21] on which to solve variable misuse and variable naming tasks in C#. Unlike other AST-based models, this model requires additional execution trace information which cannot be obtained by a static parser. Alon et al. [4, 5] extract syntactic paths from ASTs and use an attention mechanism to aggregate the embeddings of all paths.

7.4 Modeling Different Semantic Compositions in Natural Language

The differences among semantic units also exist in tree structures other than ASTs, like natural language parse trees. In these areas, researchers often use "composition" to refer to tree semantic units, mainly because in natural language constituency trees, a non-leaf node does not contain its own semantic information, where its semantics are composed by leaf nodes. Some researchers propose to use different compositional functions for different compositions [6, 10, 28]. For example, Socher et al. [28] proposed syntactically untied recursive neural network where each different sibling combination in a probabilistic contex-free grammar is assigned by a different weight matrix. Dong et al. [10] Introduced AdaRNN, a recursive neural network with multiple compositional functions for sentiment classification. During the compositional phase, the composition result is calculated based on a probabilistic distribution over all composition functions. Arabshahi et al. [6] modeled mathematic expressions with a Tree-LSTM in which each mathematical function was associated with a different set of parameters. In [23], instead of defining separate composition functions in advance, it applied a meta-network of the same structure as the original network on the input data to produce dynamic parameters for the original network. Although these approaches pointed out that using different compositional functions for different semantic compositions is useful, the differences between compositional functions are only in the parameters. Moreover, most of these models only work for binary trees with simple structures, like natural language constituency trees, while program ASTs are often much deeper and wider. Thus these models cannot be properly applied to program ASTs.

8 CONCLUSION AND FUTURE WORK

In this paper we propose MTN, a modular tree-structured recurrent neural network which can capture more detailed semantic information than previous tree-structured neural networks. We apply MTN in two tasks on C programming language: program functionality classification and code clone detection. Our results show that MTN outperformed previous sequential and tree-structured neural network models, due to its exploitation of semantic differences between AST semantic units.

In the future, one of our priorities is to extend MTN to other programming languages. Another task worth investigating is to explore the potential of MTN for program generation. Most existing AST-based program generation tasks require the decoder to generate a program from the root node to leaves, so MTN cannot be used as the decoder for program generation. But MTN can be used to encode the partial tree during the generation process, provide the decoder with structural context information.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 81.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to represent programs with graphs. In *International Conference on Learning Representations (ICLR)*.

[3] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*. 2091–2100.

[4] Uri Alon, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 40.

[6] Forough Arabshahi, Sameer Singh, and Animashree Anandkumar. 2018. Combining Symbolic Expressions and Black-box Function Evaluations in Neural Programs. In *International Conference on Learning Representations (ICLR)*.

[7] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. 2018. Neural Code Comprehension: A Learnable Representation of Code Semantics. *arXiv preprint arXiv:1806.07336* (2018).

[8] Avishkar Bhoopchand, Tim Rocktäschel, Earl Barr, and Sebastian Riedel. 2016. Learning Python code suggestion with a sparse pointer network. *arXiv preprint arXiv:1611.08307* (2016).

[9] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. 2019. Generative Code Modeling with Graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*.

[10] Li Dong, Furu Wei, Chuanqi Tan, Duyu Tang, Ming Zhou, and Ke Xu. 2014. Adaptive recursive neural network for target-dependent twitter sentiment classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Vol. 2. 49–54.

[11] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 837–847.

[12] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. 2006. A fast learning algorithm for deep belief nets. *Neural computation* 18, 7 (2006), 1527–1554.

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

[14] Xuan Huo and Ming Li. 2017. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 1909–1915.

[15] Xuan Huo, Ming Li, and Zhi-Hua Zhou. 2016. Learning Unified Features from Natural and Programming Languages for Locating Buggy Source Code.. In *IJCAI*. 1606–1612.

[16] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 2073–2083.

[17] 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.

[18] Yoon Kim. 2014. Convolutional Neural Networks for Sentence Classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. 1746–1751.

[19] Diederik P Kingma and Jimmy Lei Ba. 2015. Adam: A method for stochastic optimization. In *International Conference on Learning Representations (ICLR)*, Vol. 5.

[20] Jian Li, Yue Wang, Irwin King, and Michael R Lyu. 2018. Code Completion with Neural Attention and Pointer Networks. In *Proceedings of the 27th International Joint Conference on Artificial Intelligence*. AAAI Press, 4159–4165.

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2016. Gated graph sequence neural networks. In *International Conference on Learning Representations (ICLR)*.

[22] Yuding Liang and Kenny Zhu. 2018. Automatic Generation of Text Descriptive Comments for Code Blocks. In *AAAI Conference on Artificial Intelligence*.

[23] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. 2017. Dynamic compositional neural networks over tree structure. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. AAAI Press, 4054–4060.

[24] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing.. In *AAAI*, Vol. 2. 4.

[25] Graham Neubig, Yoav Goldberg, and Chris Dyer. 2017. On-the-fly operation batching in dynamic computation graphs. In *Advances in Neural Information Processing Systems*. 3971–3981.

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

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

[28] Richard Socher, John Bauer, Christopher D Manning, et al. 2013. Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 455–465.

[29] 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.

[30] Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480.

[31] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, Vol. 1. 1556–1566.

[32] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 297–308.

[33] Huihui Wei and Ming Li. 2018. Positive and Unlabeled Learning for Detecting Software Functional Clones with Adversarial Training.. In *IJCAI*. 2840–2846.

[34] 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.

[35] 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.

[36] Frank Wilcoxon and Roberta A Wilcox. 1964. *Some rapid approximate statistical procedures*. Lederle Laboratories.

[37] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep Learning for Just-in-Time Defect Prediction.. In *QRS*. 17–26.

[38] Zhenlong Yuan, Yongqiang Lu, Zhaoguo Wang, and Yibo Xue. 2014. Droid-sec: deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 371–372.