Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Convolutional Neural Networks over Tree Structures for Programming Language Processing

Lili Mou,[1] **Ge Li,**[1*] **Lu Zhang,**[1] **Tao Wang,**[2] **Zhi Jin**[1*]
[1]Software Institute, Peking University [*]Corresponding authors
doublepower.mou@gmail.com, {lige,zhanglu,zhijin}@sei.pku.edu.cn
[2]Stanford University, twangcat@stanford.edu

Abstract

Programming language processing (similar to natural language processing) is a hot research topic in the field of software engineering; it has also aroused growing interest in the artificial intelligence community. However, different from a natural language sentence, a program contains rich, explicit, and complicated structural information. Hence, traditional NLP models may be inappropriate for programs. In this paper, we propose a novel tree-based convolutional neural network (TBCNN) for programming language processing, in which a convolution kernel is designed over programs' abstract syntax trees to capture structural information. TBCNN is a generic architecture for programming language processing; our experiments show its effectiveness in two different program analysis tasks: classifying programs according to functionality, and detecting code snippets of certain patterns. TBCNN outperforms baseline methods, including several neural models for NLP.

Introduction

Researchers from various communities are showing growing interest in applying artificial intelligence (AI) techniques to solve software engineering (SE) problems (Dietz et al. 2009; Bettenburg and Begel 2013; Hao et al. 2013). In the area of SE, analyzing program source code—called *programming language processing* in this paper—is of particular importance.

Even though computers can run programs, they do not truly "understand" programs. Analyzing source code provides a way of estimating programs' behavior, functionality, complexity, etc. For instance, automatically detecting source code snippets of certain patterns help programmers to discover buggy or inefficient algorithms so as to improve code quality. Another example is managing large software repositories, where automatic source code classification and tagging are crucial to software reuse. Programming language processing, in fact, serves as a foundation for many SE tasks, e.g., requirement analysis (Ghabi and Egyed 2012), software development and maintenance (Bettenburg and Begel 2013).

Hindle et al. (2012) demonstrate that programming languages, similar to natural languages, also contain abundant statistical properties, which are important for program analysis. These properties are difficult to capture by humans, but justify learning-based approaches for programming language processing. However, existing machine learning program analysis depends largely on feature engineering, which is labor-intensive and *ad hoc* to a specific task, e.g., code clone detection (Chilowicz, Duris, and Roussel 2009), and bug detection (Steidl and Gode 2013). Further, evidence in the machine learning literature suggests that human-engineered features may fail to capture the nature of data, so they may be even worse than automatically learned ones.

The deep neural network, also known as *deep learning*, is a highly automated learning machine. By exploring multiple layers of non-linear transformation, the deep architecture can automatically learn complicated underlying features, which are crucial to the task of interest. Over the past few years, deep learning has made significant breakthroughs in various fields, such as speech recognition (Dahl, Mohamed, and Hinton 2010), computer vision (Krizhevsky, Sutskever, and Hinton 2012), and natural language processing (Collobert and Weston 2008).

Despite some similarities between natural languages and programming languages, there are also obvious differences (Pane, Ratanamahatana, and Myers 2001). Based on a formal language, programs contain rich and explicit structural information. Even though structures also exist in natural languages, they are not as stringent as in programs. Pinker (1994) illustrates an interesting example, "The dog the stick the fire burned beat bit the cat." This sentence complies with all grammar rules, but too many attributive clauses are nested. Hence, it can hardly be understood by people due to the limitation of human intuition capacity. On the contrary, three nested loops are common in programs. The parse tree of a program, in fact, is typically much larger than that of a natural language sentence—there are approximately 190 nodes on average in our experiment, whereas a sentence comprises only 20 words in a sentiment analysis dataset (Socher et al. 2013). Further, the grammar rules "alias" neighboring relationships among program components. The statements inside and outside a loop, for example, do not form one semantic group, and thus are not semantically neighboring. On the above basis, we think more effective neural models are in need to capture structural information in programs.

In this paper, we propose a novel *Tree-Based Convolutional Neural Network* (TBCNN) based on programs' abstract syntax trees (ASTs). We also introduce the notion of "continuous binary trees" and apply dynamic pooling to cope with ASTs of different sizes and shapes. The TBCNN model is a generic architecture, and is applied to two SE tasks in our experiments—classifying programs by functionalities and detecting code snippets of certain patterns. It outperforms baseline methods in both tasks, including the recursive neural network (Socher et al. 2011b) proposed for NLP. To the best of our knowledge, this paper is also the first to apply deep neural networks to the field of programming language processing.[1]

Related Work

Deep neural networks have made significant breakthroughs in many fields. Stacked restricted Boltzmann machines and autoencoders are successful pretraining methods (Hinton, Osindero, and Teh 2006; Bengio et al. 2006). They explore the underlying features of data in an unsupervised manner, and give a more meaningful initialization of weights for later supervised learning with deep neural networks. These approaches work well with generic data (e.g. data located in a manifold embedded in a certain dimensional space), but they may not be suitable for programming language processing, because programs contain rich structural information. Further, AST structures also vary largely among different data samples (programs), and hence they cannot be fed directly to a fixed-size network.

To capture explicit structures in data, it may be important and beneficial to integrate human priors to the networks (Bengio, Courville, and Vincent 2013). One example is convolutional neural networks (CNNs, LeCun et al. 1995; Krizhevsky, Sutskever, and Hinton 2012), which specify spatial neighboring information in data. CNNs work with signals of a certain dimension (e.g., images); they also fail to capture tree-structural information as in programs.

Socher et al. (2013, 2011b) propose a recursive neural network (RNN) for NLP. Although structural information may be coded to some extent in RNNs, the major drawback is that only the root features are used for supervised learning, which buries illuminating information under a complicated neural architecture. RNNs also suffer from the difficulty of training due to the long dependency path during back-propagation (Bengio, Simard, and Frasconi 1994).

Subsequent work. After the preliminary version of this paper was preprinted on arXiv,[2] Zaremba and Sutskever (2014) use recurrent neural networks to estimate the output of restricted python programs. Piech et al. (2015) build recursive networks on Hoare triples. Regarding the proposed TBCNN, we extend it to process syntactic parse trees of natural languages (Mou et al. 2015); Duvenaud et al. (2015) apply a similar convolutional network over graphs to analyze molecules.

[1]We make our source code and the collected dataset available through our website (https://sites.google.com/site/treebasedcnn/).

[2]On 18 September 2014 (http://arxiv.org/abs/1409.5718v1).

Tree-Based Convolutional Neural Network

Programming languages have a natural tree representation—the abstract syntax tree (AST). Figure 1a shows the AST of the code snippet "`int a=b+3;`".[3] Each node in the AST is an abstract component in program source code. A node p with children $c_1, \cdots, c_n$ represents the constructing process of the component $p \to c_1 \cdots c_n$.

Figure 1b shows the overall architecture of TBCNN. In our model, an AST node is first represented as a distributed, real-valued vector so that the (anonymous) features of the symbols are captured. The vector representations are learned by a coding criterion in our previous work (Peng et al. 2015).

Then we design a set of subtree feature detectors, called the *tree-based convolution kernel*, sliding over the entire AST to extract structural information of a program. We thereafter apply dynamic pooling to gather information over different parts of the tree. Finally, a hidden layer and an output layer are added. For supervised classification tasks, the activation function of the output layer is $\mathrm{softmax}$.

In the rest of this section, we first explain the coding criterion for AST nodes' representation learning, serving as a pretraining phase of programming language processing. We then describe the proposed TBCNN model, including a coding layer, a convolutional layer, and a pooling layer. We also provide additional information on dealing with nodes that have varying numbers of child nodes, as in ASTs, by introducing the notion of continuous binary trees.

Representation Learning for AST Nodes

Vector representations, sometimes known as *embeddings*, can capture underlying meanings of discrete symbols, like AST nodes. We propose in our previous work (Peng et al. 2015) an unsupervised approach to learn program vector representations by a coding criterion, which serves as a way of pretraining.

A generic criterion for representation learning is "smoothness"—similar symbols have similar feature vectors (Bengio, Courville, and Vincent 2013). For example, the symbols `While` and `For` are similar because both of them are related to control flow, particularly loops. But they are different from `ID`, since `ID` probably represents some data. In our scenario, we would like the child nodes' representations to "code" their parent node's via a single neural layer, during which both vector representations and coding weights are learned. Formally, let $\mathrm{vec}(\cdot) \in \mathbb{R}^{N_f}$ be the feature representation of a symbol, where N_f is the feature dimension. For each non-leaf node p and its direct children $c_1, \cdots, c_n$, we would like

$$\mathrm{vec}(p) \approx \tanh\left(\sum\nolimits_i l_i W_{\text{code},i} \cdot \mathrm{vec}(c_i) + \boldsymbol{b}_{\text{code}}\right) \qquad (1)$$

where $W_{\text{code},i} \in \mathbb{R}^{N_f \times N_f}$ is the weight matrix corresponding to the node c_i; $\boldsymbol{b}_{\text{code}} \in \mathbb{R}^{N_f}$ is the bias. $l_i = \frac{\#\text{leaves under } c_i}{\#\text{leaves under } p}$ is the coefficient of the weight. (Weights $W_{\text{code},i}$ are weighted by leaf numbers.)

Because different nodes may have different numbers of children, the number of $W_{\text{code},i}$'s is not fixed. To overcome

[3]Parsed by pycparser (https://pypi.python.org/pypi/pycparser/).

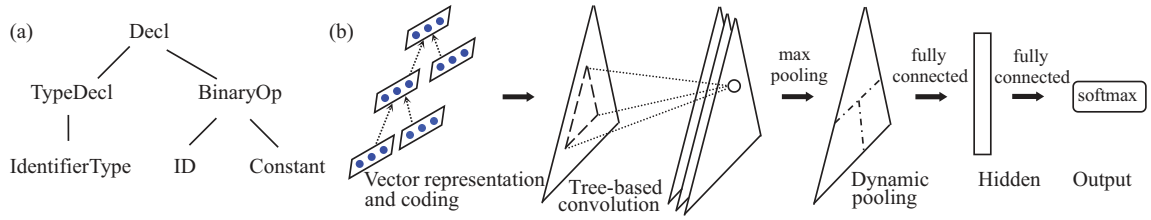

Figure 1: (a) Illustration of an AST, corresponding to the C code snippet "int a=b+3;" It should be notice that our model takes as input the entire AST of a program, which is typically much larger. (b) The architecture of the Tree-Based Convolutional Neural Network (TBCNN). The main components in our model include vector representation and coding, tree-based convolution and dynamic pooling; then a fully-connected hidden layer and an output layer (softmax) are added.

this problem, we introduce the "continuous binary tree," where only two weight matrices W^l_{code} and W^r_{code} serve as model parameters. W_i is a linear combination of the two parameter matrices according to the position of node i. Details are deferred to the last part of this section.

The closeness between $\text{vec}(p)$ and its coded vector is measured by Euclidean distance square, i.e.,

$$d = \left\| \text{vec}(p) - \tanh\left(\sum\nolimits_i l_i W_{\text{code},i} \cdot \text{vec}(c_i) + \boldsymbol{b}_{\text{code}} \right) \right\|_2^2$$

To prevent the pretraining algorithm from learning trivial representations (e.g., $\mathbf{0}$'s will give 0 distance but are meaningless), negative sampling is applied like Collobert et al. (2011). For each pretraining data sample $p, c_1, \cdots, c_n$, we substitute one symbol (either p or one of c's) with a random symbol. The distance of the negative sample is denoted as d_c, which should be at least larger than that of the positive training sample plus a margin Δ (set to 1 in our experiment). Thus, the pretraining objective is to

$$\underset{W^l_{\text{code}}, W^r_{\text{code}}, \boldsymbol{b}_{\text{code}}, \text{vec}(\cdot)}{\text{minimize}} \quad \max\{0, \Delta + d - d_c\}$$

Coding Layer

Having pretrained the feature vectors for all symbols, we would like to feed them forward to the tree-based convolutional layer for supervised learning. For leaf nodes, they are just the vector representations learned in the pretraining phase. For a non-leaf node p, it has two representations: the one learned in the pretraining phase (left-hand side of Equation 1), and the coded one (right-hand side of Equation 1). They are linearly combined before being fed to the convolutional layer. Let $c_1, \cdots, c_n$ be the children of node p and we denote the combined vector as $\boldsymbol{p}$. We have

$$\begin{aligned} \boldsymbol{p} &= W_{\text{comb1}} \cdot \text{vec}(p) \\ &+ W_{\text{comb2}} \cdot \tanh\left(\sum\nolimits_i l_i W_{\text{code},i} \cdot \text{vec}(x_i) + \boldsymbol{b}_{\text{code}} \right) \end{aligned}$$

where $W_{\text{comb1}}, W_{\text{comb2}} \in \mathbb{R}^{N_f \times N_f}$ are the parameters for combination. They are initialized as diagonal matrices and then fine-tuned during supervised training.

Tree-based Convolutional Layer

Now that each symbol in ASTs is represented as a distributed, real-valued vector $\boldsymbol{x} \in \mathbb{R}^{N_f}$, we apply a set of fixed-depth feature detectors sliding over the entire tree, depicted in Figure 2a. The subtree feature detectors can be viewed as convolution with a set of finite support kernels. We call this *tree-based convolution*.

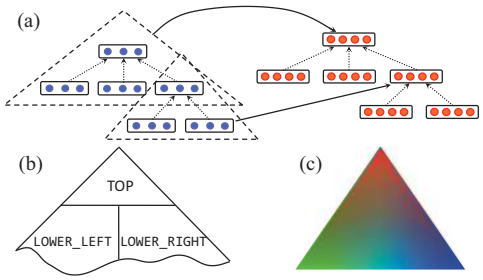

Figure 2: (a) Tree-based convolution. Nodes on the left are the feature vectors of AST nodes. They are either pretrained or combined with pretrained and coded vectors. (b) An illustration of 3-way pooling. (c) An analogy to the continuous binary tree model. In the triangle, the color of a pixel is a combination of three primary colors; in the convolution process, the weight for a node is a combination of three weight parameters, namely W^t_{conv}, W^l_{conv}, and W^r_{conv}.

Formally, in a fixed-depth window, if there are n nodes with vector representations $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$, then the output of the feature detectors is[4]

$$\boldsymbol{y} = \tanh\left(\sum\nolimits_{i=1}^{n} W_{\text{conv},i} \cdot \boldsymbol{x}_i + \boldsymbol{b}_{\text{conv}} \right)$$

where $\boldsymbol{y}, \boldsymbol{b}_{\text{conv}} \in \mathbb{R}^{N_c}$, $W_{\text{conv},i} \in \mathbb{R}^{N_c \times N_f}$. ($N_c$ is the number of feature detectors.) $\mathbf{0}$'s are padded for nodes at the bottom that do not have as many layers as the feature detectors. In our experiments, the kernel depth is set to 2.

Note that, to deal with varying numbers of children, we also adopt the notion of continuous binary tree. In this scenario, three weight matrices serve as model parameters, namely W^t_{conv}, W^l_{conv}, and W^r_{conv}. $W_{\text{conv},i}$ is a linear combination of these three matrices (explained in detail in the last part of this section).

[4]We used `tanh` as the activation function in TBCNN mainly because we hope to encode features to a same semantic space $(-1, 1)$ during coding. We are grateful to an anonymous reviewer for reminding us of using `ReLU` in convolution, and we are happy to try it in future work.

Dynamic Pooling

After convolution, structural features in an AST are extracted, and a new tree is generated. The new tree has exactly the same shape and size as the original one, which is varying among different programs. Therefore, the extracted features cannot be fed directly to a fixed-size neural layer. Dynamic pooling (Socher et al. 2011a) is applied to deal with this problem.

The simplest approach, perhaps, is to pool all features to one vector. We call this *one-way pooling*. Concretely, the maximum value in each dimension is taken from the features that are detected by tree-based convolution. We also propose an alternative, *three-way pooling*, where features are pooled to 3 parts, TOP, LOWER_LEFT, and LOWER_RIGHT, according to the their positions in the AST (Figure 2b). As we shall see from the experimental results, the simple one-way pooling just works as well as three-way pooling. Therefore we adopt one-way pooling in our experiments.

After pooling, the features are fully connected to a hidden layer and then fed to the output layer ($\mathrm{softmax}$) for supervised classification. With the dynamic pooling process, structural features along the entire AST reach the output layer with short paths. Hence, they can be trained effectively by back-propagation.

The "Continuous Binary Tree" Model

As stated, one problem of coding and convolving is that we cannot determine the number of weight matrices because AST nodes have different numbers of children.

One possible solution is the continuous bag-of-words model (CBoW, Mikolov et al., 2013),[5] but position information will be lost completely. Such approach is also used in Hermann and Blunsom (2014). Socher et al. (2014) allocate a different weight matrix as parameters for each position; but this method fails to scale up since there will be a huge number of different positions in ASTs.

In our model, we view any subtree as a "binary" tree, regardless of its size and shape. That is, we have only three weight matrices as parameters for convolution, and two for coding. We call it a *continuous binary tree*.

Take convolution as an example. The three parameter matrices are W^t_{conv}, W^l_{conv}, and W^r_{conv}. (Superscripts t, l, r refer to "top," "left," and "right.") For node x_i in a window, its weight matrix for convolution $W_{\text{conv},i}$ is a linear combination of W^t_{conv}, W^l_{conv}, and W^r_{conv}, with coefficients η^t_i, η^l_i, and η^r_i, respectively. The coefficients are computed according to the relative position of a node in the sliding window. Figure 2c is an analogy to the continuous binary tree model. The equations for computing η's are listed as follows.

- $\eta^t_i = \frac{d_i - 1}{d - 1}$ (d_i: the depth of the node i in the sliding window; d: the depth of the window.)
- $\eta^r_i = (1 - \eta^t_i)\frac{p_i - 1}{n - 1}$. ($p_i$: the position of the node; n: the total number of p's siblings.)
- $\eta^l_i = (1 - \eta^t_i)(1 - \eta^r_i)$

[5] In their original paper, they do not deal with varying-length data, but their method extends naturally to this scenario. Their method is also mathematically equivalent to average pooling.

Likewise, the continuous binary tree for coding has two weight matrices W^l_{code} and W^r_{code} as parameters. The details are not repeated here.

To sum up, the entire parameter set for TBCNN is $\Theta = \{W^l_{\text{code}}, W^r_{\text{code}}, W_{\text{comb1}}, W_{\text{comb2}}, W^t_{\text{conv}}, W^l_{\text{conv}}, W^r_{\text{conv}}, W_{\text{hid}}, W_{\text{out}}, \boldsymbol{b}_{\text{code}}, \boldsymbol{b}_{\text{conv}}, \boldsymbol{b}_{\text{hid}}, \boldsymbol{b}_{\text{out}}, \mathrm{vec}(\cdot)\}$, where W_{hid}, W_{out}, $\boldsymbol{b}_{\text{hid}}$, and $\boldsymbol{b}_{\text{out}}$ are the weights and biases for the hidden and output layers. To set up supervised training, W^l_{code}, W^r_{code}, $\boldsymbol{b}_{\text{code}}$, and $\mathrm{vec}(\cdot)$ are derived from the pretraining phase; W_{comb1} and W_{comb2} are initialized as diagonal matrices; other parameters are initialized randomly. We apply the cross-entropy loss and use stochastic gradient descent, computed by back-propagation.

Experiments

We first assess the learned vector representations both qualitatively and quantitatively. Then we evaluate TBCNN in two supervised learning tasks, and conduct model analysis.

The dataset of our experiments comes from a pedagogical programming open judge (OJ) system.[6] There are a large number of programming problems on the OJ system. Students submit their source code as the solution to a certain problem; the OJ system automatically judges the validity of submitted source code by running the program. We downloaded the source code and the corresponding programming problems (represented as IDs) as our dataset.

Unsupervised Program Vector Representations

We applied the coding criterion of pretraining to all `C` code in the OJ system, and obtained AST nodes' vector representations.

Qualitative analysis. Figure 3a illustrates the hierarchical clustering result based on a subset of AST nodes. As demonstrated, the symbols mainly fall into three categories: (1) `BinaryOp`, `ArrayRef`, `ID`, `Constant` are grouped together since they are related to data reference/manipulation; (2) `For`, `If`, `While` are similar since they are related to control flow; (3) `ArrayDecl`, `FuncDecl`, `PtrDecl` are similar since they are declarations. The result is quite sensible because it is consistent with human understanding of programs.

Quantitative analysis. We also evaluated pretraining's effect on supervised learning by feeding the learned representations to a program classification task. (See next subsection.) Figure 3b plots the learning curves of both training and validation, which are compared with random initialization. Unsupervised vector representation learning accelerates the supervised training process by nearly $1/3$, showing that pretraining does capture underlying features of AST nodes, and that they can emerge high-level features spontaneously during supervised learning. However, pretraining has a limited effect on the final accuracy. One plausible explanation is that the number of AST nodes is small: the `pycparser`, we use, distinguishes only 44 symbols. Hence, their representations can be adequately tuned in a supervised fashion.

[6] http://programming.grids.cn

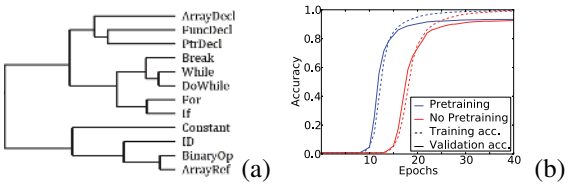

Figure 3: Analysis of vector representations. (a) Hierarchical clustering based on AST nodes' vector representations. (b) Learning curves with and without pretraining.

Statistics	Mean	Sample std.
# of code lines	36.3	19.0
# of AST nodes	189.6	106.0
Average leaf nodes' depth in an AST	7.6	1.7
Max depth of an AST	12.3	3.2

Table 1: Statistics of our dataset.

Hyperparameter	Value	How is the value chosen?
Initial learning rate	0.3	By validation
Learning rate decay	None	Empirically
Embedding dimension	30	Empirically
Convolutional layers' dim.	600	By validation
Penultimate layer's dim.	600	Same as conv layers
l_2 penalty	None	Empirically

Table 2: TBCNN's hyperparameters.

Nonetheless, we think the pretraining criterion is effective and beneficial for TBCNN, because training deep neural networks is usually time-consuming, especially when tuning hyperparameters. The pretrained vector representations are used throughout the experiments below.

Classifying Programs by Functionalities

Task description In software engineering, classifying programs by functionalities is an important problem for various software development tasks. For example, in a large software repository (e.g., SourceForge), software products are usually organized into categories, a typical criterion for which is by functionalities. With program classification, it becomes feasible to automatically tag a software component newly added into the repository, which is beneficial for software reuse during the development process.

In our experiment, we applied TBCNN to classify source code in the OJ system. The target label of a data sample is one of 104 programming problems (represented as an ID). That is, programs with a same target label have the same functionality. We randomly chose exactly 500 programs in each class, and thus 52,000 samples in total, which were further randomly split by 3:1:1 for training, validation, and testing. Relevant statistics are shown in Table 1.

Hyperparameters TBCNN's hyperparameters are shown in Table 2. Our competing methods include SVM and a deep feed-forward neural network based on hand-crafted features, namely bag-of-words (BoW, the counting of each symbol) or bag-of-tree (BoT, the counting of 2-layer subtrees). We also compare our model with the recursive neural network

Group	Method	Test Accuracy (%)
Surface features	linear SVM+BoW	52.0
	RBF SVM+BoW	83.9
	linear SVM+BoT	72.5
	RBF SVM+BoT	88.2
NN-based approaches	DNN+BoW	76.0
	DNN+BoT	89.7
	Vector avg.	53.2
	RNN	84.8
Our method	TBCNN	**94.0**

Table 3: The accuracy of 104-label program classifications.

(RNN, Socher et al. 2011b). Hyperparameters for baselines are listed as follows.

SVM. The linear SVM has one hyperparameter C; RBF SVM has two, C and γ. They are tuned by validation over the set $\{\cdots, 1, 0.3, 0.1, 0.03, \cdots\}$ with grid search.

DNN. We applied a 4-layer DNN (including input) empirically. The hidden layers' dimension is 300, chosen from $\{100, 300, 1000\}$; learning rates are 0.003 for BoW and 0.03 for BoT, chosen from $\{0.003, \cdots, 0.3\}$ with granularity 3x. ℓ_2 regularization coefficient is 10^{-6} for both BoW and BoT, chosen from $\{10^{-7}, \cdots, 10^{-4}\}$ with granularity 10x, and also no regularization.

RNN. Recursive units are 600-dimensional, as in our method. The learning rate is chosen from the set $\{\cdots 1.0, 0.3, 0.1 \cdots\}$, and 0.3 yields the highest validation performance.

Results Table 3 presents the results in the 104-label program classification experiment. Using SVM with surface features does distinguish different programs to some extent—for example, a program about string manipulation is different from, say, matrix operation; also, a difficult programming problem necessitates a more complex program, and thus more lines of code and AST nodes. However, their performance is comparatively low.

We tried deep feed-forward neural networks on these features, and achieved accuracies of 76.0–89.7%, comparable to SVMs. Vector averaging with $\mathrm{softmax}$—another neural network-based competing method applied in NLP (Socher et al. 2013; Kalchbrenner, Grefenstette, and Blunsom 2014)—yields an accuracy similar to a linear classifier built on BoW features. This is probably because the number of AST symbols is far fewer than words in natural languages, and thus the vector representations (provided non-singular) can be absorbed into the classifier's weights. Comparing these approaches with our method, we deem TBCNN's performance boost is not merely caused by using a better classifier (neural networks versus SVM, say), but also the feature/representation learning nature, which enables automatic structural feature extraction.

We also applied RNN to the program classification task[7]; the RNN's accuracy is lower than shallow methods

[7]We do not use the pretrained vector representations, which are inimical to RNN: the weight W_{code} codes children's representation to its candidate parent's; adversely, the high-level nodes in programs (e.g., a function definition) are typically non-informative.

Classifier	Features	Accuracy
Rand/majority	–	50.0
RBF SVM	Bag-of-words	62.3
RBF SVM	Bag-of-trees	77.1
TBCNN	Learned	**89.1**

Table 4: Accuracy of detecting bubble sort (in percentage).

Model Variant	Validation Acc.
Coding layer → None	92.3
1-way pooling → 3-way	94.3
Continuous binary tree → CBoW	93.1
TBCNN with the best gadgets	94.4

Table 5: Effect of coding, pooling, and the continuous binary tree.

(SVM+BoT). Taking into consideration experiments in NLP (Socher et al. 2011b; 2013), we observe a degradation of RNN's performance if the tree structure is large.

TBCNN outperforms the above methods, yielding an accuracy of 94%. By exploring tree-based convolution, our model is better at capturing programs' structural features, which is important for program analysis.

Detecting Bubble Sort

Task description To further evaluate our TBCNN model in a more realistic SE scenario, we used it to detect an unhealthy code pattern, bubble sort, which can also be regarded as a (binary) program classification task. Detecting source code of certain patterns is closely related to many SE problems. In this experiment, bubble sort is thought of as unhealthy code because it implements an inefficient algorithm. By identifying such unhealthy code, project managers can refine the implementations during the maintenance process.

Before the experiment, a volunteer[8] annotated, from the OJ system, 109 programs that contain bubble sort, and 109 programs that do not contain bubble sort. They were split 1:1 for validation and testing.

Data augmentation To train our TBCNN model, a dataset of such scale is insufficient. We propose a simple yet useful data augmentation technique for programs. Concretely, we used the source code of 4k programs in the OJ system as the non-bubble sort class. For each program, we randomly substituted a fragment of program statements with a pre-written bubble sort snippet. Thus we had 8k data samples in total.

Results We tested our model on the annotated real-world programs. Note that the test samples were written by real-world programmers, and thus the styles and forms of bubble sort snippets may differ from the training set, for example, sorting an integer array versus sorting a user-defined structure, and sorting an array versus sorting two arrays simultaneously. As we see in Table 4, bag-of-words features are not illuminating in this classification and yield a low accuracy of 62.3%. Bag-of-trees features are better, and achieve 77.06%.

[8]The volunteer has neither authorship nor a conflict of interests.

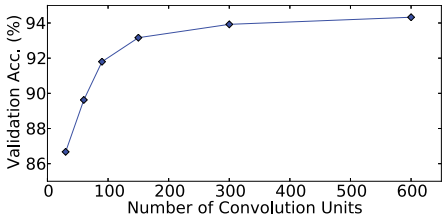

Figure 4: Validation accuracy versus the number of convolution units.

Our model outperforms these methods by more than 10%. This experiment also suggests that neural networks can learn more robust features than just counting surface statistics.

Model Analysis

We now analyze each gadget of TBCNN quantitatively, with the 104-label program classification as our testbed. We report validation accuracies throughout this part.

Effect of coding layer In the proposed TBCNN model for program analysis, we represent a non-leaf node by combining its coded representation and its pretrained one. We find that, the underneath coding layer can also integrate global information in addition to merely averaging two homogeneous sources. If we build a tree-based convolutional layer directly on the pretrained vector representations, all structural features are "local," that is, confined in the convolution window. The lack of integrating global information leads to 2% degradation in performance. (See the first and last rows in Table 5.)

Layers' dimensions In our experiments, AST nodes' vector representations are set to be 30-dimensional empirically. We chose this small value because AST nodes have only 44 different symbols. Hence, the dimension needs to be, intuitively, smaller than words' vector representations, e.g., 300 in Mou et al. (2015). The dimension of convolution, i.e., the number of feature detectors, was chosen by validation (Figure 4). We tried several configurations, among which 600-dimensional convolution results in the highest validation accuracy. This analysis also verifies that programs have rich structural information, even though the number of AST symbols is not large. As the rich semantics are emerged by different combinations of AST symbols, we are in need of more feature detectors, that is, a larger convolutional layer.

Effect of pooling layer We tried two pooling methods in our TBCNN model, and compare them in Table 5 (the second and last rows). 3-way pooling is proposed in hope of preserving features from different parts of the tree. However, as indicated by the experimental result, the simple 1-way pooling works just as fine (even 0.1% higher on the validation set). This suggests that TBCNN is not sensitive to pooling methods, which mainly serve as a necessity for packing varying sized and shaped data. Further development can be addressed in future work.

Effect of continuous binary tree The continuous binary tree is introduced to treat nodes with different numbers of

children, as well as to capture order information of child nodes. We also implemented the continuous bag-of-words (CBoW) model, where child nodes' representations are averaged before convolution. Rows 4 and 5 in Table 5 compare our proposed continuous binary tree and the above alternative. The result shows a boost of 1.3% in considering child nodes' order information.

Conclusion

In this paper, we applied deep neural networks to the field of programming language processing. Due to the rich and explicit tree structures of programs, we proposed the novel Tree-Based Convolutional Neural Network (TBCNN). In our model, program vector representations are learned by the coding criterion; structural features are detected by the convolutional layer; the continuous binary tree and dynamic pooling enable our model to cope with trees of varying sizes and shapes. Experimental results show the superiority of our model to baseline methods.

Acknowledgments

We would like to thank anonymous reviewers for insightful comments; we also thank Xiaowei Sun for annotating bubble sort programs, Yuxuan Liu for data processing, and Weiru Liu for discussion on the manuscript. This research is supported by the National Basic Research Program of China (the 973 Program) under Grant No. 2015CB352201 and the National Natural Science Foundation of China under Grant Nos. 61421091, 61232015, 61225007, 91318301, and 61502014.

References

Bengio, Y.; Lamblin, P.; Popovici, D.; and Larochelle, H. 2006. Greedy layer-wise training of deep networks. In *NIPS*.

Bengio, Y.; Courville, A.; and Vincent, P. 2013. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35(8):1798–1828.

Bengio, Y.; Simard, P.; and Frasconi, P. 1994. Learning long-term dependencies with gradient descent is difficult. *IEEE Trans. Neural Networks* 5(2):157–166.

Bettenburg, N., and Begel, A. 2013. Deciphering the story of software development through frequent pattern mining. In *ICSE*, 1197–1200.

Chilowicz, M.; Duris, E.; and Roussel, G. 2009. Syntax tree fingerprinting for source code similarity detection. In *Proc. IEEE Int. Conf. Program Comprehension*, 243–247.

Collobert, R., and Weston, J. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*.

Collobert, R.; Weston, J.; Bottou, L.; Karlen, M.; Kavukcuoglu, K.; and Kuksa, P. 2011. Natural language processing (almost) from scratch. *JRML* 12:2493–2537.

Dahl, G.; Mohamed, A.; and Hinton, G. 2010. Phone recognition with the mean-covariance restricted Boltzmann machine. In *NIPS*.

Dietz, L.; Dallmeier, V.; Zeller, A.; and Scheffer, T. 2009. Localizing bugs in program executions with graphical models. In *NIPS*.

Duvenaud, D.; Maclaurin, D.; Aguilera-Iparraguirre, J.; Gómez-Bombarelli, R.; Hirzel, T.; Aspuru-Guzik, A.; and Adams, R. 2015. Convolutional networks on graphs for learning molecular fingerprints. *arXiv preprint arXiv:1509.09292*.

Ghabi, A., and Egyed, A. 2012. Code patterns for automatically validating requirements-to-code traces. In *ASE*, 200–209.

Hao, D.; Lan, T.; Zhang, H.; Guo, C.; and Zhang, L. 2013. Is this a bug or an obsolete test? In *Proc. ECOOP*, 602–628.

Hermann, K., and Blunsom, P. 2014. Multilingual models for compositional distributed semantics. In *ACL*, 58–68.

Hindle, A.; Barr, E.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *ICSE*, 837–847.

Hinton, G.; Osindero, S.; and Teh, Y. 2006. A fast learning algorithm for deep belief nets. *Neural Computation* 18(7):1527–1554.

Kalchbrenner, N.; Grefenstette, E.; and Blunsom, P. 2014. A convolutional neural network for modelling sentences. In *ACL*, 655–665.

Krizhevsky, A.; Sutskever, I.; and Hinton, G. 2012. ImageNet classification with deep convolutional neural networks. In *NIPS*.

LeCun, Y.; Jackel, L.; Bottou, L.; Brunot, A.; Cortes, C.; Denker, J.; Drucker, H.; Guyon, I.; Muller, U.; and Sackinger, E. 1995. Comparison of learning algorithms for handwritten digit recognition. In *Proc. Int. Conf. Artificial Neural Networks*.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *NIPS*.

Mou, L.; Peng, H.; Li, G.; Xu, Y.; Zhang, L.; and Jin, Z. 2015. Discriminating neural sentence modeling by tree-based convolution. In *EMNLP*, 2315–2325.

Pane, J.; Ratanamahatana, C.; and Myers, B. 2001. Studying the language and structure in non-programmers' solutions to programming problems. *Int. J. Human-Computer Studies* 54(2):237–264.

Peng, H.; Mou, L.; Li, G.; Liu, Y.; Zhang, L.; and Jin, Z. 2015. Building program vector representations for deep learning. In *Proc. 8th Int. Conf. Knowledge Science, Engineering and Management*, 547–553.

Piech, C.; Huang, J.; Nguyen, A.; Phulsuksombati, M.; Sahami, M.; and Guibas, L. 2015. Learning program embeddings to propagate feedback on student code. In *ICML*.

Pinker, S. 1994. *The Language Instinct: The New Science of Language and Mind*. Pengiun Press.

Socher, R.; Huang, E.; Pennin, J.; Manning, C.; and Ng, A. 2011a. Dynamic pooling and unfolding recursive autoencoders for paraphrase detection. In *NIPS*.

Socher, R.; Pennington, J.; Huang, E.; Ng, A.; and Manning, C. 2011b. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *EMNLP*, 151–161.

Socher, R.; Perelygin, A.; Wu, J.; Chuang, J.; Manning, C.; Ng, A.; and Potts, C. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *EMNLP*, 1631–1642.

Socher, R.; Karpathy, A.; Le, Q.; Manning, C.; and Ng, A. Y. 2014. Grounded compositional semantics for finding and describing images with sentences. *TACL* 2:207–218.

Steidl, D., and Gode, N. 2013. Feature-based detection of bugs in clones. In *7th Int. Workshop on Software Clones*, 76–82.

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