TBCNN: A Tree-Based Convolutional Neural Network for Programming Language Processing

Lili Mou[1], **Ge Li**[1*], **Zhi Jin**[1], **Lu Zhang**[1], **Tao Wang**[2]
[1] Software Institute, Peking University, [*] Corresponding author
{moull12, lige, zhijin, zhanglu}@sei.pku.edu.cn
[2] Stanford University, twangcat@stanford.edu

Abstract

Deep neural networks have made significant breakthroughs in many fields of artificial intelligence. However, it has not been applied in the field of programming language processing. In this paper, we propose the tree-based convolutional neural network (TBCNN) to model programming languages, which contain rich and explicit tree structural information. In our model, program vector representations are learned by the "coding" pretraining criterion based on abstract syntax trees (ASTs); the convolutional layer explicitly captures neighboring features on the tree; with the "binary continuous tree" and "3-way pooling," our model can deal with ASTs of different shapes and sizes. We evaluate the program vector representations empirically, showing that the coding criterion successfully captures underlying features of AST nodes, and that program vector representations significantly speed up supervised learning. We also compare TBCNN to baseline methods; our model achieves better accuracy in the task of program classification. To our best knowledge, this paper is the first to analyze programs with deep neural networks; we extend the scope of deep learning to the field of programming language processing. The experimental results validate its feasibility; they also show a promising future of this new research area.

Introduction

The deep neural network, also known as *deep learning*, is a highly automated learning algorithm; it has become a rapid developing research area recent years. By exploring multiple layers of non-linear transformation, the deep architecture can extract underlying abstract features of data, which is crucial to artificial intelligence (AI) (Bengio 2009). Although deep learning has been successful in various fields—like natural language processing (NLP) (Collobert and Weston 2008), computer vision (Krizhevsky, Sutskever, and Hinton 2012), and speech recognition (Dahl, Mohamed, and Hinton 2010)—its advantages are not exploited in the field of programming language processing.

Programming languages, similar to natural languages, are the reflection of human thought. Programs are complex, flexible and powerful; they also contain rich statistical properties (Hindle et al. 2012). Analyzing programs with learning-based approaches is an important research topic in both AI and software engineering (Lu, Cukic, and Culp 2012; Canavera, Esfahani, and Malek 2012; Lee, Jung, and Pande 2014).

Despite some similarities, there are also obvious differences between programming languages and natural languages (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 programs due to the limitation of human intuition capacity. (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 human intuition due to the over-complicated structure. On the contrary, 3 nested loops are pretty common in programs. Further, the grammar rules "alias" the neighboring relationships among program components. For example, the statements inside and outside a loop do not form one semantic group, and thus they are not semantically neighboring. Therefore, due to the different characteristics of natural languages and programming languages, new neural models should be proposed for programs to capture such structural information.

In this paper, we propose the novel Tree-Based Convolutional Neural Network (TBCNN) to model programming languages based on abstract syntax trees (ASTs). Our contributions include: (1) proposing the "coding criterion" to learn vector representations for each node in ASTs, serving as the pretraining phase; (2) proposing tree-based convolution approach to extract local structural features of programs; (3) introducing the notions of "continuous binary tree" and "3-way pooling" so that our model is suitable for trees with different structures and sizes.

In our experiments, program vector representations are evaluated empirically. The results are consistent with human intuition, showing that our pretraining process successfully captures meaningful features of different program symbols (nodes in ASTs). TBCNN is evaluated in the task of program classification; it achieves higher accuracy compared with baseline methods. Our study validates the feasibility of neural programming language processing.

To our best knowledge, we are the first to analyze programs by deep neural networks. We extend the scope of deep learning to the field of programming language processing.

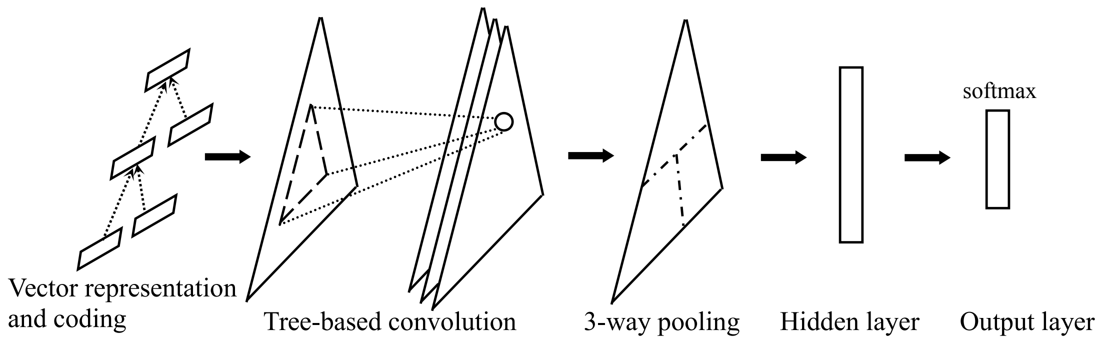

Figure 1: The architecture of the Tree-Based Convolutional Neural Network. The main components in our model include vector representation and coding, tree-based convolution and 3-way pooling. Then a fully-connected hidden layer and an output layer ($\mathrm{softmax}$) are added.

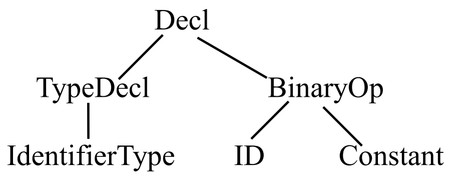

Figure 2: The AST corresponding to the C code snippet "int a=b+3;" An obvious difference of ASTs and natural language syntax trees is that the latter do not have explicit meaning for non-leaf nodes (e.g. a noun phrase); but AST nodes do (e.g. BinaryOp).

Based on current evidence, we believe it will become an outstanding technique in this new field in the near future.

Tree-based Convolutional Neural Network

The Architecture

Programming languages have the natural tree representation—the abstract syntax tree (AST). Figure 2 shows the AST corresponding to the following C code snippet (parsed by pycparser[1]):

```
int a = b + 3;
```

Each node in the AST is an abstract component in the program source code. A node p with children $c_1, \cdots, c_n$ represents the constructing process of the component $p \to c_1 \cdots c_n$.

The overall architecture of TBCNN is shown in Figure 1. In our model, each node in ASTs 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 the proposed "coding criterion."

Then we convolve a set of feature detectors on the AST and apply 3-way pooling, after which, we add a hidden layer and an output layer. For supervised classification tasks, the activation function of the output layer is $\mathrm{softmax}$.

[1]https://pypi.python.org/pypi/pycparser/

Representation Learning for AST Nodes

As all nodes in ASTs are discrete, they should be represented as real-valued, distributed vectors. A generic criterion for representation learning is that 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 children's representations to "code" its parent's via a single neural layer. Formally, let $\mathrm{vec}(\cdot) \in \mathbb{R}^{N_f}$ be the feature representation of a symbol. 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 symbol 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 number.)

Because different nodes may have different numbers of children, the number of $W_{\text{code},i}$'s is not fixed. To overcome the problem, we introduce the "continuous binary tree," where only two weight matrices W_{code}^l and W_{code}^r serve as model parameters. W_i is a linear combination of the two parameter matrices. The details will be explained in the last part of this section.

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

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

To prevent the pretraining algorithm from learning trivial representations (e.g., $\mathbf{0}$'s will give 0 distance but is 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. We would like d_c to be at least larger than that of the

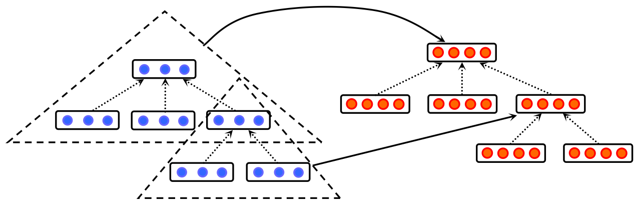

Figure 3: Tree-based convolution. Nodes on the left are the feature vectors of the symbols in AST. They are either pretrained or combined with pretrained and coded vectors. Nodes on the right are features detected by convolution. The two dashed triangles illustrate a set of convolution kernels applying over the tree. With the "continuous binary tree," we can apply the same feature detectors (with the same parameters) even if nodes have different numbers of children. Note that the dashed arrows are not part of our neural networks. They merely indicate the AST topology.

positive training sample plus a margin Δ (set to 1 in our experimental setting). Then the pretraining objective is to

$$\underset{W_{\text{code}}^l, W_{\text{code}}^r, \boldsymbol{b}_{\text{code}}}{\text{minimize}} \quad \max\{0, \Delta + d - d_c\}$$

Coding Layer

Having pretrained the feature vectors for every symbol, we feed them forwardly to the tree-based convolutional layer. For leaves, 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 Formula 1), and the coded one (right-hand side of Formula 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 Convolution 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 local feature detectors sliding over the entire tree, as is illustrated in Figure 3. This can be viewed as convolution with a set of finite support kernels. We call this "tree-based convolution."

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 set of feature detectors is

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

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

Note that, to deal with variable numbers of children, we also adopt the "continuous binary tree." In this scenario, there are three weight matrices serving as model parameters, namely W_{conv}^t, W_{conv}^l and W_{conv}^r. $W_{\text{conv},i}$ is a linear combination of these three weight matrices (explained in the last part of this section).

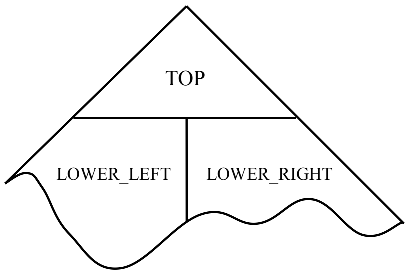

Figure 4: 3-way pooling. The features after convolution are pooled to three parts: TOP, LOWER_LEFT and LOWER_RIGHT. Each part is a fixed-size vector.

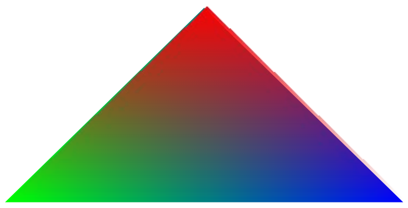

Figure 5: An analogy to the continuous binary tree model. The color of a pixel in the triangle is a combination of three-primary colors. In our model, the weight is a combination of three weight parameters, namely W_{conv}^t, W_{conv}^l, and W_{conv}^r.

3-Way Pooling Layer

After convolution, local neighboring features in the 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 also varying among different programs. Therefore, they cannot directly be fed to a fixed-size hidden layer.

We propose 3-way max pooling, which pools the features into 3 parts: TOP, LOWER_LEFT and LOWER_RIGHT according to the position in AST (Figure 4). The following strategies are used during the pooling process.

- The nodes with depth less than k * average depth of the leaves are pooled to TOP. (k is set to 0.6 in our setting.)
- Lower nodes are pooled to the LOWER_LEFT and LOWER_RIGHT according to their relative position regarding the root node.

With such 3-way pooling, local structural features along the entire AST can reach the output layer with short pathes. Hence, these structural features can be trained effectively by back propagation.

After pooling, the features are fully connected to a hidden layer and then fed to the output layer ($\mathrm{softmax}$) for supervised classification.

The "Continuous Binary Tree" Model

As stated in previous subsections, during coding or convolving, one problem is that we cannot determine the number of weight matrices because AST nodes have variable numbers of children.

One possible solution is the "continuous bag of words" model (Mikolov et al. 2013)[2], but position information will be lost completely. Similar approach is used in (Hermann and Blunsom 2014). In (Socher et al. 2013a), a different weight matrix is allocated as parameters for each position. This method fails to scale up since there will be a huge number of different positions in our scenario.

In our model, we view any subtree as a "binary" tree, regardless of its size and shape. That is, we have only 3 weight matrices for convolution and 2 for coding. We call it a "continuous binary tree."

Take convolution as an example. The three parameter matrices are W_{conv}^t, W_{conv}^l and W_{conv}^r. (Superscripts t, l, r refers to "top," "left" and "right.") For node x_i in the window, its weights for convolution $W_{\text{conv},i}$ is a linear combination of W_{conv}^t, W_{conv}^l, and W_{conv}^r with coefficients η_i^t, η_i^l, and η_i^r. The coefficients are computed according to the relative position of a node in the sliding window. Figure 5 is an analogy to the continuous binary tree model. The formulas for computing η's are listed as follows.

- $\eta_i^t = \dfrac{d_i - 1}{d - 1}$, where d_i is the depth of the node i in the sliding window and d is the depth of the window.
- $\eta_i^r = (1-\eta_i^t)\dfrac{p_i - 1}{n - 1}$, where p_i is the position of the node, and n is the total number of p's children.
- $\eta_i^l = (1-\eta_i^t)(1-\eta_i^r)$

Likewise, the continuous binary tree for coding has two weight matrices W_{code}^l and W_{code}^r. The details are not repeated here.

To sum up, the entire parameter set for TBCNN is $\Theta = \Big(W_{\text{code}}^l, W_{\text{code}}^r, W_{\text{comb1}}, W_{\text{comb2}}, W_{\text{conv}}^t, W_{\text{conv}}^l, W_{\text{conv}}^r, W_{\text{hid}}, W_{\text{out}}, \boldsymbol{b}_{\text{code}}, \boldsymbol{b}_{\text{conv}}, \boldsymbol{b}_{\text{hid}}, \boldsymbol{b}_{\text{out}}, \mathrm{vec}(\cdot)\Big)$, where W_{hid}, W_{out}, $\boldsymbol{b}_{\text{hid}}$, $\boldsymbol{b}_{\text{out}}$ are the weights and biases for the hidden and output layer. The number of hidden layer neurons is denoted as N_h. To set up supervised training, $W_{\text{code}}^l, W_{\text{code}}^r, \boldsymbol{b}_{\text{code}}, \mathrm{vec}(\cdot)$ are derived from the pretraining phase; W_{comb1} and W_{comb2} are initialized as diagonal matrices; other parameters are initialized randomly. We add ℓ_2 regularization to W's. The parameters are trained supervisedly in the program classification task using standard backpropagation algorithm by stochastic gradient descent with momentum.

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

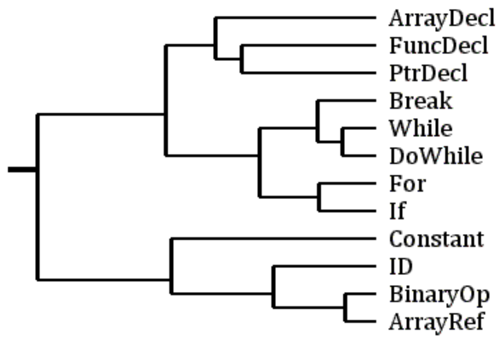

Figure 6: Hierarchical clustering results based on vector representations for AST nodes.

Experimental Results

In this section, we present two experimental results. First, we evaluate the program vector representations empirically by hierarchical clustering to show the vector representations successfully capture meaningful features of AST nodes. Second, we apply TBCNN to classify programs based on functionality to verify the feasibility and effectiveness of neural programming language processing.

The Dataset

We use the dataset of a pedagogical programming open judge (OJ) system[3]. There are a large number of programming problems on the OJ system. Students submit their source codes as the solution to certain problems; the OJ system judges the validity of submitted source codes automatically. We download the source codes and the corresponding problem IDs (labels) as our dataset. The representation learning is performed over all C codes. In the task of program classification, two groups of programming problems are selected for classification. Each group contains 4 problems, which are similar in functionality[4]. We split the dataset into 3 parts: 60% for training, 20% for cross-validating (CV) and 20% for testing.

The results, source codes, and dataset are available from our project website[5].

Evaluation of Vector Representations

We perform hierarchical clustering to evaluate the learned representations for AST nodes. The result confirms that similar symbols tend to have similar feature vectors. Figure 6 illustrates a subset of AST nodes. (The entire result and the original vector representations can be downloaded at our project website.)

As demonstrated in Figure 6, the symbols mainly fall into three categories: (1) `BinaryOp`, `ArrayRef`, `ID`, `Constant` are grouped together since they are related

[3]The URL of the OJ system is anonymized for peer reviewing. The dataset can be downloaded at our project URL.

[4]This selected dataset prevents our classification task from being trivial. For randomly chosen problems, it is likely to achieve very high accuracy with simple approaches.

[5]https://sites.google.com/site/treebasedcnn/

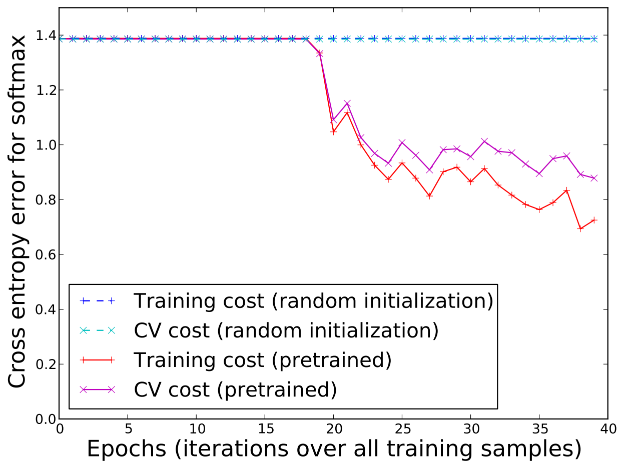

Figure 7: Learning curves with and without pretraining.

to data reference/manipulating; (2) `For`, `If`, `While`, `DoWhile` are similar since they are related to control flow; (3) `ArrayDecl`, `FuncDecl`, `PtrDecl` are similar since they are related to declarations. The result is quite meaningful because it is consistent with human understanding of programs.

Another evaluation for representation learning is to see whether it improves supervised learning of interest. We perform program classification, and plot in Figure 7 the training and cross-validating (CV) learning curves of first 40 epochs (iterations over all training examples) for both random initialized and pretrained weights. For the sake of simplicity and fairness, the hyperparameters are set as $N_f = N_c = N_h = 30$; ℓ_2 penalty = 0; momentum= 0; learning rate is set to 0.03 and fixed. These settings are selected manually in advance, different from the settings for final classification (they are chosen by CV).

As we can see from Figure 7, after a plateaux of 15 epochs, the cost functions are going down significantly in the setting with pretrained parameters. However, if we randomly initialize the weights, training becomes extremely more slow and ineffective—the decrease of cost function in first 40 epochs is less than 4 digits after the decimal point. The experiment shows that pretraining speeds up the training process to a large extents. The details of the supervised learning results will be presented in the next subsection.

Evaluation of the TBCNN model

To evaluate the feasibility and effectiveness of neural programming language processing, we apply TBCNN to classify programs. The results are presented in Table 1[6].

We compare our approach to baseline methods, namely logistic regression (LR) and SVM with linear and radial basis function (RBF) kernels. These methods adopt the bag-of-words model and use counting features, i.e., the feature vector of a program is the numbers of symbol occurrences. Linear classifiers (logistic regression and linear SVM) achieve

[6]Part of the result is first reported in (Mou et al. 2014).

Table 1: Supervised program classification error rates.

GRP.	Method	Train Err.	CV Err.	Test Err.
1	Random guess	75	55	75
	LR	24.3	26.86	26.7
	Linear SVM	24.89	27.51	28.48
	RBF SVM	4.38	12.63	11.31
	TBCNN	4.03	9.98	10.14
	TBCNN+BOW	3.86	8.37	**8.53**
2	Random guess	75	75	75
	LR	16.86	18.04	18.84
	Linear SVM	17.18	17.87	19.48
	RBF SVM	0.27	8.21	8.86
	TBCNN	0.48	5.31	4.98
	TBCNN+BOW	0.54	3.70	**3.70**

approximately the same accuracy. SVM with RBF kernel is better than linear classifiers. We also apply the existing recursive neural networks (RNN, Socher et al. 2013b) to the program classification task. But it is not trained effectively in our scenario. The cost function is roughly $-\log(0.25) \approx 1.4$ during our whole training process, where 0.25 is the probability of each (similar to the blue/cyan dashed lines in Figure 7). RNN merely learns a distribution over labels, and no useful information is effectively captured by RNN. We compare the models and analyze the results in the next section.

By exploring the program features automatically with TBCNN, we achieve better accuracy than baseline methods in both two groups. According to model design, TBCNN captures local structural information by convolution, but it loses counting information because of max pooling. On the other hand, bag-of-words (BOW) model contains counting information, but the structural information is lost. When we combine these two models, we achieve highest accuracy in both groups. This confirms that, TBCNN and BOW capture different aspects of programs—local structural features and counting features, both of which are beneficial for program classification.

Related Work in Deep Learning

Deep neural networks have made significant breakthroughs in many fields of artificial intelligence, for example, computer vision (Krizhevsky, Sutskever, and Hinton 2012), speech recognition (Dahl, Mohamed, and Hinton 2010), natural language processing (Collobert et al. 2011), etc. Stacked restricted Boltzmann machines and autoencoders are successful pretraining methods (Hinton, Osindero, and Teh 2006; Bengio et al. 2007). They explore the underlying features of data unsupervisedly, and give a more meaningful initialization of weights for later supervised learning. These approaches work well with generic data, but they are not suitable for programming language processing, because programs contain rich structural information. Further, AST structures also vary largely among different programs, and hence they cannot be fed directly to a fixed-size network.

To capture explicit structures of data, it may be important and beneficial to integrate human priors to the networks (Bengio, Courville, and Vincent 2013). One example is the convolutional neural network (CNN, LeCun et al. 1995), which specifies spatial neighboring information in data. CNN works with data in k-dimensional space; but it fails to capture tree-structural information as in programs. Another example is the recurrent neural network, which can be regarded as a time-decaying network (Mikolov et al. 2010). Hence, it is typically suitable for one-dimensional data, but structural information is also lost.

A model similar to ours is the recursive neural network (RNN) proposed in (Socher et al. 2011; 2013b) for NLP. Although structural information may be coded to some extent in RNN, the major drawback is that only root features are used for supervised learning. As we have seen, RNN is not trained effectively in our setting for the program classification task. We analyze the results and consider the following as main causes: (1) Different pretraining criteria. By the coding criterion, useful local structures are buried under uninformative high level program components (e.g. root, function call). (2) Weak information interaction. The root features are the bottleneck of RNN. Lower layer features have long-path dependency to the output layer through the bottleneck, which adds to the difficulty of training RNN in our scenario.

Different from the above models, TBCNN explores local structural features by tree-based convolution; these features are divided into three regions and pooled for supervised learning. Based on the experiments, we think RNN and TBCNN are complementary to each other, each suitable for different scenarios.

Conclusion

In this paper, we extent the scope of deep learning to programming language processing. Due to the rich and explicit tree structures of programs, we propose the novel Tree-Based Convolutional Neural Network (TBCNN). In this model, program vector representations are learned by the coding criterion; local structural features are detected by the convolution layer; the notions of continuous binary tree and 3-way pooling are introduced to model trees with varying sizes and shapes.

Empirical experiments show that meaningful features of AST nodes are successfully captured by the coding criterion. The TBCNN model is evaluated in the task of program classification; it achieves higher accuracy than baseline methods.

The experiments validate the feasibility of neural program processing; they also show a bright future of this new field. Based on current evidence, we believe deep learning will make great progress in the field of programming language processing.

References

Bengio, Y.; Lamblin, P.; Popovici, D.; and Larochelle, H. 2007. Greedy layer-wise training of deep networks. In *Advances in Neural Information Processing Systems*.

Bengio, Y.; Courville, A.; and Vincent, P. 2013. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35(8):1798–1828.

Bengio, Y. 2009. Learning deep architectures for AI. *Foundations and Trends in Machine Learning* 2(1):1–127.

Canavera, K.; Esfahani, N.; and Malek, S. 2012. Mining the execution history of a software system to infer the best time for its adaptation. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*.

Collobert, R., and Weston, J. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine learning*.

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

Dahl, G.; Mohamed, A.; and Hinton, G. E. 2010. Phone recognition with the mean-covariance restricted Boltzmann machine. In *Advances in Neural Information Processing Systems*.

Hermann, K., and Blunsom, P. 2014. Multilingual models for compositional distributed semantics. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*.

Hindle, A.; Barr, E.; Su, Z.; Gabel, M.; and Devanbu, P. 2012. On the naturalness of software. In *Proceedings of 34th International Conference on Software Engineering*.

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

Krizhevsky, A.; Sutskever, I.; and Hinton, G. 2012. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*.

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 *Proceedings of International Conference on Artificial Neural Networks*.

Lee, S.; Jung, C.; and Pande, S. 2014. Detecting memory leaks through introspective dynamic behavior modelling using machine learning. In *Proceedings of 36th International Conference on Software Engineering*.

Lu, H.; Cukic, B.; and Culp, M. 2012. Software defect prediction using semi-supervised learning with dimension reduction. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*.

Mikolov, T.; Karafiat, M.; Burget, L.; Cernocky, J.; and Khudanpur, S. 2010. Recurrent neural network based language model. In *INTERSPEECH*.

Mikolov, T.; Sutskever, I.; Chen, K.; Corrado, G.; and Dean, J. 2013. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems*.

Mou, L.; Li, G.; Liu, Y.; Peng, H.; Jin, Z.; Xu, Y.; and Zhang, L. 2014. Building program vector representations for deep learning. *arXiv preprint arXiv:1409.3358*.

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

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

Socher, R.; Pennington, J.; Huang, E.; Ng, A.; and Manning, C. 2011. Semi-supervised recursive autoencoders for predicting sentiment distributions. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Socher, R.; Le, Q.; Manning, C.; and Ng, A. 2013a. Grounded compositional semantics for finding and describing images with sentences. In *NIPS Deep Learning Workshop*.

Socher, R.; Perelygin, A.; Wu, J.; Chuang, J.; Manning, C.; Ng, A.; and Potts, C. 2013b. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of Conference on Empirical Methods in Natural Language Processing*.