Building Program Vector Representations for Deep Learning

Hao Peng, Lili Mou, Ge Li(✉), Yuxuan Liu, Lu Zhang, and Zhi Jin

Software Institute, School of EECS, Peking University Beijing, Beijing 100871, People's Republic of China
{penghao.pku,doublepower.mou,liuyuxuan}@gmail.com,
{lige,zhanglu,zhijin}@sei.pku.edu.cn

Abstract. Deep learning has made significant breakthroughs in various fields of artificial intelligence. However, it is still virtually impossible to use deep learning to analyze programs since deep architectures cannot be trained effectively with pure back propagation. In this pioneering paper, we propose the "coding criterion" to build program vector representations, which are the premise of deep learning for program analysis. We evaluate the learned vector representations both qualitatively and quantitatively. We conclude, based on the experiments, the coding criterion is successful in building program representations. To evaluate whether deep learning is beneficial for program analysis, we feed the representations to deep neural networks, and achieve higher accuracy in the program classification task than "shallow" methods. This result confirms the feasibility of deep learning to analyze programs.

1 Introduction

Machine learning-based program analysis has been studied long in the literature [14,3]. Hindle et al. compare programming languages to natural languages and conclude that programs have rich statistical properties [10]. These properties are difficult for human to capture, but they justify using learning-based approaches to analyze programs.

The deep neural network has become one of the prevailing machine learning approaches since 2006 [11]. It has made significant breakthroughs in a variety of fields, such as natural language processing [6,19], image processing [12,4], speech recognition [7,15], etc. Such striking results raise the interest of its applications in the field of program analysis. Using deep learning to automatically capture program features is an interesting and prospective research area.

Unfortunately, it has been practically infeasible for deep learning to analyze programs up till now. Since no proper "pretraining" method is proposed for programs, deep neural networks cannot be trained effectively with pure back propagation [2,8].

H. Peng and L. Mou—Equal contribution.

© Springer International Publishing Switzerland 2015
S. Zhang et al. (Eds.): KSEM 2015, LNAI 9403, pp. 547–553, 2015.
DOI: 10.1007/978-3-319-25159-2_49

In this paper, we propose novel "coding criterion" to build program vector representations based on abstract syntax trees (ASTs). In such vector representations, each node in ASTs (e.g. `ID`, `Constant`) is mapped to a real-valued vector. They can emerge high-level abstract features, and thus benefit ultimate tasks. We analyze the learned representations both qualitatively and quantitatively. We conclude from the experiments that the coding criterion is successful in building program vector representations.

In the rest of this paper, we first we explain our approach in detail in Section 2. Then we give experimental results in Section 3. Last, we draw our conclusion in Section 4.

2 Coding Criterion for Program Representation Learning

In this section, we first discuss the granularities of program representation. We settle for the granularity of nodes in abstract syntax trees (ASTs). In Subsection 2.2, we formalize our approach and give the learning objective. In Subsection 2.3, we present the stochastic gradient descent algorithm for training.

2.1 The Granularity

Vector representations map a symbol to a real-valued vector. Possible granularities of the symbol include character-level, token-level, etc.

- **Character-level.** Treating each character as a symol. Although some research explore character-level modeling for NLP [20], it is improper for programming languages. For example, the token `double` in a C code refers to a data type. But if one writes `doubles`, it is an identifier (e.g., a function name).
- **Token-level.** Learning the representations of all tokens, including types and identifiers etc. Since programmers can declare their own identifiers in their source codes, e.g., `func1`, `func2`, many of the identifiers may appear only a few times, resulting in the undesired data sparseness. Hence, it is improper for representation learning at this level.
- **Nodes in ASTs.** Learning the representations for nodes in ASTs, e.g., `FuncDef`, `ID`, `Constant`. The AST is more compressed compared with token-level representation. Furthermore, there are only finite many types of nodes in ASTs. The tree structural nature of ASTs also provides opportunities to capture structural information of programs. This level is also used in traditional program analysis like code clone detection [1,13], vulnerability extrapolation [21], etc.
- **Statement-level, function-level or higher.** Theoretically, a statement, a function or even a program can also be mapped to a real-valued vector. However, such representations cannot be trained directly. A possible approach of modeling such complex stuff is by composition. Such researches in NLP is often referred to as *compositional semantics* [18]. It is very hard to capture the precise semantics; the "semantic barrier" is still not overcome.

2.2 Formalization

The basic criterion of representation learning is that similar symbols should have similar representations. Further, symbols that are similar in some aspects should have similar values in corresponding feature dimensions.

In our scenario, similarity is defined based on the following intuition: similar symbols have similar usages: both `ID` and `Constant` can be an operand of a binary operator; both `For` and `While` are a block of codes, etc.

We denote the vector of node x as $\text{vec}(x)$. $\text{vec}(\cdot) \in \mathbb{R}^{N_f}$, where N_f is the dimension of features. For each non-leaf node p in ASTs and its direct children $c_1, \cdots, c_n$, their representations are $\text{vec}(p), \text{vec}(c_1), \cdots, \text{vec}(c_n)$. The primary objective is that

$$\text{vec}(p) \approx \tanh\left(\sum_{i=1}^{n} l_i W_i \cdot \text{vec}(c_i) + \boldsymbol{b}\right) \tag{1}$$

where $W_i \in \mathbb{R}^{N_f \times N_f}$ is the weight matrix for node c_i; $\boldsymbol{b} \in \mathbb{R}^{N_f}$ is the bias term. The weights (W_i's) are weighted by the number of leaves under c_i and the coefficients are

$$l_i = \frac{\#\text{leaves under } c_i}{\#\text{leaves under } p} \tag{2}$$

Since different nodes in ASTs may have different numbers of children, the number of W_i's is hard to determine. To solve this problem, we propose continuous binary tree, where there are two weight matrices as parameters, namely W_l and W_r. Any weight W_i is a linear combination of the two matrices. That is, regardless the number of children, we treat it as a "binary" tree. Formally, if p has n ($n \geq 2$) children, then for child c_i,

$$W_i = \frac{n-i}{n-1} W_l + \frac{i-1}{n-1} W_r \tag{3}$$

Now that we are able to calculate the weight W_i for each node, we measure closeness by the square of Euclidean distance, as below:

$$d = \left\| \text{vec}(p) - \tanh\left(\sum_{i=1}^{n} l_i W_i \cdot \text{vec}(c_i) + \boldsymbol{b}\right) \right\|_2^2 \tag{4}$$

We applied negative sampling [5,18,17]: for each data sample x, a new negative sample x_c is generated. We randomly select a symbol in each training sample and substitute it with a different random symbol. The objective is that d_c should be at least as large as $d + \Delta$, where Δ is the margin and often set to 1. The error function of training sample $x^{(i)}$ and its negative sample $x_c^{(i)}$ is then

$$J(d^{(i)}, d_c^{(i)}) = \max\left\{0, \Delta + d^{(i)} - d_c^{(i)}\right\} \tag{5}$$

To prevent our model from over-fitting, we can add ℓ_2 regularization to weights (W_l and W_r). The overall training objective is then

$$\underset{W_l, W_r, \boldsymbol{b}}{\text{minimize}} \ \frac{1}{2N} \sum_{i=1}^{N} J(d^{(i)}, d_c^{(i)}) + \frac{\lambda}{2M} (\| W_l \|_F^2 + \| W_r \|_F^2 \ ) \tag{6}$$

where N is the number of training samples; $M = 2N_f^2$ denotes the number of weights; $\| \cdot \|_F$ refers to Frobenius norm; λ is the hyperparameter that strikes the balance between coding error and ℓ_2 penalty.

2.3 Training

The numerical optimization algorithm we use is stochastic gradient descent with momentum. The model parameters $\Theta = \Big(\text{vec}(\cdot), W_l, W_r, \boldsymbol{b} \Big)$ are first initialized randomly. Then, for each data sample $x^{(i)}$ and its negative sample $x_c^{(i)}$, we compute the cost function according to Formula 6. Back propagation algorithm is then applied to compute the partial derivatives and the parameters are updated accordingly. This process is looped until convergence. The coding criterion of vector representation learning—as a pretraining phase for neural program analysis—is "shallow," through which error can back propagate. Thus, useful features are learned for AST nodes.

To speed up training, we adopt the momentum method, where the partial derivatives of the last iteration is added to the current ones with decay ϵ.

3 Experiments

We first evaluate our learned representations by k-means clustering. We then perform supervised learning in the program classification task. The experimental results show that meaningful representations, as a means of pretraining, make the network much easier to train in deep architectures. We also achieve higher accuracy with the deep, tree-based convolutional neural network compared with baseline methods.

3.1 Qualitative Evaluation: *k*-means Clustering

As we have stated, similar nodes in ASTs (like `ID`, `Constant`) should have similar representations. To evaluate whether our coding criterion has accomplished this goal, we perform k-means clustering, where k is set to 3. The result is shown in Table 1. As we see, almost all the symbols in Cluster 1 are related to data reference/manipulating. Cluster 2 is mainly about declarations. Cluster 3 contains more symbols, the majority of which are related to control flow. This result confirms our conjecture that similar symbols can be clustered into groups with the distributed vector representations that are learned by our approach.

Table 1. The result of k-means clustering. k is set to 3.

Cluster	Sybmols
1	UnaryOp, FuncCall, Assignment, ExprList, StructRef, BinaryOp, ID, Constant, ArrayRef
2	FuncDef, TypeDecl, FuncDecl, Compound, ArrayDecl, PtrDecl, Decl, Root
3	Typedef, Struct, For, Union, CompoundLiteral, TernaryOp, Label, InitList, IdentifierType, Return, Enum, Break, DoWhile, Case, DeclList, Default, While, Continue, ParamList, Enumerator, Typename, Goto, Cast, Switch, EmptyStatement, EnumeratorList, If

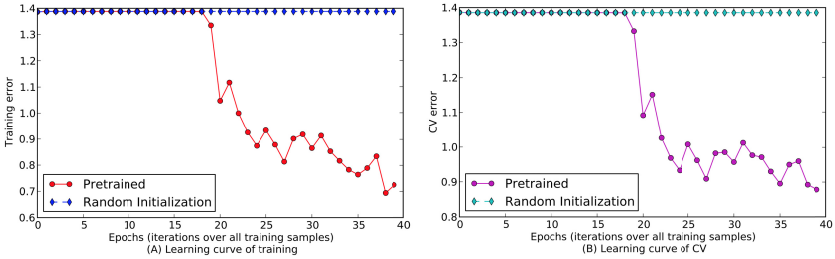

Fig. 1. Learning curves of training (A) and CV (B). The learned program vector representations improve supervised learning in terms of both generalization and optimization.

3.2 Quantitative Evaluation: Improvement for Supervised Learning

We now evaluate whether building program vector representations is beneficial for real-world tasks, i.e., whether they will improve optimization and/or generalization for supervised learning of interest.

The dataset comes from an online Open Judge system[1], which contains a large number of programming problems for students. We select four problems for our program classification task. Source codes (in C programming language) of the four problems are downloaded along with their labels (problem IDs). We split the dataset by $3:1:1$ for training, cross-validating (CV) and testing.

Since no effective program representation existed before, the TBCNN [16] model is not trained efficiently, as the blue curve demonstrates in Part A of Figure 1.

If the vector representations and the coding parameters, namely vec$(\cdot)$, W_l, W_r and $\boldsymbol{b}$, are initialized as are learned by our coding criterion, the training and CV errors decrease drastically (the red and magenta curves) after a plateaux of about 15 epochs, which leads to the high performance of TBCNN.

To evaluate whether deep learning may be helpful for program analysis, we compare TBCNN to baseline methods in the program classification task. In these baseline methods, we adopt the bag-of-words model, which is a widely-used approach in text classification [9]. As shown in Table 2, logistic regression, as a

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

Table 2. Accuracy of Program Classification.

Method	Accuracy
Random guess	25.00%
Logistic regression	81.16%
SVM with RBF kernel	91.14%
TBCNN (a deep learning approach)	**95.33**%

linear classifier, achieves 81.16% accuracy. The support vector machine (SVM) with radial basis function (RBF) kernel explores non-linearity, and improves the result by 10%. By automatically exploring the underlying features and patterns of programs, TBCNN further improves the accuracy by more than 4%. This experiment suggests the promising future of deep leaning approaches in the field of program analysis.

4 Conclusion

In this paper, we study deep learning and representation learning in the field of program analysis. We propose a novel "coding criterion" to build vector representations of nodes in ASTs. We also feed the learned representations to a deep neural network to classify programs. The experimental results show that our representations successfully capture the similarity and relationships among different nodes in ASTs. We conclude that the coding criterion is successful in building program vector representations. The experiments also confirm the feasibility of deep learning to analyze programs.

Acknowledgments. 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 No. 61232015.

References

1. Baxter, I., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance (1998)
2. Bengio, Y., Lamblin, P., Popovici, D., Larochelle, H.: Greedy layer-wise training of deep networks. In: Advances in Neural Information Processing Systems (2007)
3. Canavera, K., Esfahani, N., Malek, S.: 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 (2012)
4. Ciresan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: IEEE Conference on Computer Vision and Pattern Recognition (2012)
5. Collobert, R., Weston, J.: A unified architecture for natural language processing: deep neural networks with multitask learning. In: Proceedings of the 25th International Conference on Machine Learning (2008)

6. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. The Journal of Machine Learning Research **12**, 2493–2537 (2011)
7. Dahl, G., Mohamed, A., Hinton, G.E.: Phone recognition with the mean-covariance restricted Boltzmann machine. In: Advances in Neural Information Processing Systems (2010)
8. Erhan, D., Manzagol, P., Bengio, Y., Bengio, S., Vincent, P.: The difficulty of training deep architectures and the effect of unsupervised pre-training. In: Proceedings of International Conference on Artificial Intelligence and Statistics (2009)
9. Feldman, R., Sanger, J.: The Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data. Cambridge University Press (2007)
10. Hindle, A., Barr, E., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of 34th International Conference on Software Engineering (2012)
11. Hinton, G., Osindero, S., Teh, Y.: A fast learning algorithm for deep belief nets. Neural Computation **18**(7), 1527–1554 (2006)
12. Krizhevsky, A., Sutskever, I., Hinton, G.: ImageNet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems (2012)
13. Lazar, F., Banias, O.: Clone detection algorithm based on the abstract syntax tree approach. In: Proceedings of 9th IEEE International Symposium on Applied Computational Intelligence and Informatic (2014)
14. Lu, H., Cukic, B., Culp, M.: Software defect prediction using semi-supervised learning with dimension reduction. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (2012)
15. Mohamed, A., Dahl, G., Hinton, G.: Acoustic modeling using deep belief networks. IEEE Transactions on Audio, Speech, and Language Processing **20**(1), 14–22 (2012)
16. Mou, L., Peng, H., Li, G., Xu, Y., Zhang, L., Jin, Z.: Tree-based convolution: a new neural architecture for sentence modeling (2015). CoRR abs/1504.01106. http://arxiv.org/abs/1504.01106
17. Socher, R., Chen, D., Manning, C., Ng, A.: Reasoning with neural tensor networks for knowledge base completion. In: Advances in Neural Information Processing Systems (2013)
18. Socher, R., Le, Q., Manning, C., Ng, A.: Grounded compositional semantics for finding and describing images with sentences. In: NIPS Deep Learning Workshop (2013)
19. Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C., Ng, A., Potts, C.: Recursive deep models for semantic compositionality over a sentiment treebank. In: Proceedings of Conference on Empirical Methods in Natural Language Processing (2013)
20. Sutskever, I., Martens, J., Hinton, G.: Generating text with recurrent neural networks. In: Proceedings of the 28th International Conference on Machine Learning (2011)
21. Yamaguchi, F., Lottmann, M., Rieck, K.: Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of 28th Annual Computer Security Applications Conference (2012)