# Code Embedding

Linyang Wu

2024.2.27

## 1 Brief Introduction

Code embedding is a technique used in machine learning and software engineering to convert source code into a numerical representation (also known as a "vector") that captures syntactic and semantic features of the code. This process is analogous to word embedding in natural language processing, where words are mapped to vectors of real numbers.

### 1.1 Features

Code embedding captures both syntax (the structure of code) and semantics (the meaning/functionality of code).Embeddings can be learned from raw source code or from a parsed structure such as an AST.The resulting vectors can be used to measure similarity or dissimilarity between code snippets, facilitating tasks like detecting code clones or suggesting code changes.

### 1.2 Applications

Facilitates tasks such as code search, where developers can query codebases with natural language or other code snippets.

Enables code completion tools that suggest next tokens or code blocks to developers while coding.

Enhances capabilities in code summarization, generating natural language descriptions from source code.

Assists in automated code review, vulnerability detection, and program synthesis.

### 1.3 Developing process

Traditional techniques include Bag of Words (BoW) and Term Frequency-Inverse Document Frequency (TF-IDF).

Advancements in neural networks led to the adoption of word embedding methods such as Word2Vec and GloVe for the purpose of embedding code tokens in a continuous vector space. The success of deep learning in NLP inspired similar approaches in code embeddings. Recurrent Neural Networks (RNNs)

and Long Short-Term Memory networks (LSTMs) began to be utilized for their ability to capture sequential information and long-range dependencies within code.

The introduction of Transformer models and their self-attention mechanisms significantly improved the ability to handle long-range dependencies and to capture the context better than RNNs or LSTMs.Pre-trained models like BERT (Bidirectional Encoder Representations from Transformers) were adapted for source code, leading to models capable of learning from vast amounts of code data, such as CodeBERT and GraphCodeBERT.

# 2 Building program vector representations for deep learning

## 2.1 Introduction

The popularization of AI applications in analyzing source code provides a method to estimate the behavior, functionality, complexity, and more of a program.

Programming languages, like natural languages, contain rich statistical properties that are crucial for program analysis. While there are similarities between natural language and programming languages, there are also notable differences. Programs contain rich structural information, whereas natural language has less strict structure. Nested relative clauses are rare in natural language, whereas nested loops are common in programs.However, existing machine learning models struggle to capture these properties, and human-designed features are even worse.

Deep neural networks have made breakthroughs in speech recognition, computer vision, and natural language processing.

This article proposes a tree-based convolutional neural network based on the abstract syntax tree of a program.

## 2.2 The coding criterion for AST nodes' representation learning

### 2.2.1 Motivation

Due to the fact that all program symbols, are discrete, there is no inherent ordering defined for these symbols. These discrete symbols cannot be directly inputted into a neural network. One possible solution is to map each symbol to a real-valued vector in some dimension. Each element of the vector can capture a specific feature of the symbol. Hence, this representation is also known as distributed representation.

Due to the structural differences between natural language and programming languages , existing natural language processing representation learning algorithms are not suitable for programs.Therefore, new representation learning algorithms are needed.

### 2.2.2 The Granularity

In order to map a symbol to a real-valued distributed vector, we first need to determine the granularity of the symbol.

Character-level.Although some researches explore character-level modeling for NLP,it is improper for programming languages.For example,in a C code,double refers to a data type and doubles may refers to a function.They have similar vector representations because their most characters are same,but they have completely meaning.

Token-level. Unlike natural languages, where the number of words is generally fixed, programmers can declare their own identifiers in their source codes,Because some identifiers may appear only a few times (e.g., tmp), we will suffer from the problem of undesired data sparseness.

Nodes in ASTs.The ASTs are more compressed and 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.

Statement-level, function-level or higher. It is very hard to capture the precise semantics; the "semantic barrier" is still not overcome.Such representations cannot be trained directly.

After analyzing different granularities of program representations,the representation for nodes in ASTs has theoretical foundations, and is feasible to learn and useful in applications.

### 2.2.3 Formal Description

To capture such similarity using AST structural information, The idea is that the representation of a node in ASTs should be "coded" by its children's representations via a single neural layer.

We denote the vector of node $x$ as $vec(x)$. $vec() \in \mathbb{R}^{N_f}$,where $N_f$ is the dimension of features. ($N_f$ is set to 30 empirically in our experimental setting.) For each non-leaf node $p$ in ASTs and its direct children $c_1, \cdot \cdot \cdot, c_n$, their representations are $vec(p)$, $vec(c_1), \cdot \cdot \cdot, vec(c_n)$. The primary objective is that

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

in which $W_i \in \mathbb{R}^{N_f}$is weight,and $b \in \mathbb{R}^{N_f}$ is bias.

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

Because different nodes may have different numbers of children, the number of $W_i$ is not fixed. the "continuous binary tree can solve this problem., if p has n (n 2) children, then for child $c_i$,

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

**Algorithm 1:** StochasticGradientDescentWithMomentum

**Input**: Data samples $x^{(i)}$, $i = 1..N$;
Momentum $\epsilon$;
Learning rate $\alpha$

**Output**: Model parameters $\Theta = \left(\text{vec}(\cdot), W_l, W_r, \boldsymbol{b}\right)$

Randomly initialize $\Theta$;
**while** *not converged* **do**
  **for** $i = 1..N$ **do**
    Generate a negative sample $x_c^{(i)}$ for $x^{(i)}$;
    Propagate forward and backward to compute $J^{(i)}$ and the partial derivative $\frac{\partial J^{(i)}}{\partial \Theta}$;
    $\frac{\partial J^{(i)}}{\partial \Theta} \leftarrow \epsilon \frac{\partial J^{(i-1)}}{\partial \Theta} + \frac{\partial J^{(i)}}{\partial \Theta}$; // momentum
    $\Theta \leftarrow \Theta - \alpha \frac{\partial J^{(i)}}{\partial \Theta}$; // gradient descent
  **end**
**end**

Figure 1: Gradient Descent Algorithm

We measure closeness by the square of Euclidean distance, as below:

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

To prevent the pretraining algorithm from learning trivial representations,negative sampling is applied. For each pretraining data sample p, c1, $\cdots$, cn, we substitute one symbol (either p or one of c's) with a random symbol. The distance of the negative sample is denoted as dc, which should be at least larger than that of the positive training sample plus a margin $\Delta$ (set to 1 in our experiment). Thus, The error function of training sample $x^i$ and its negative sample $x_c^i$ is then

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

Also,to prevent our model from over-fitting,add regularization.

The numerical optimization algorithm we use is stochastic gradient descent with momentum.The model parameters are first initialized randomly. Then, for each data sample ,compute the cost function according to Formula (5). Back propagation algorithm is then applied to compute the partial derivatives and the parameters are updated accordingly. This process is looped until convergence.

The details of this algorithm are as shown in Figure 1.

# 3 Tree Based Convolutional Neural Network Model

## 3.1 Coding Layer

After pretraining,we can forward the vector 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 $p$,

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

## 3.2 Tree-based Convolutional Layer

In a fixed-depth window, if there are n nodes with vector representations $x1$, $\cdots$, $xn$, then the output of the feature detectors is

$$\boldsymbol{y} = \tanh\left(\sum_{i=1}^{n} W_{\text{conv},i} \cdot \boldsymbol{x}_i + \boldsymbol{b}_{\text{conv}}\right) \tag{6}$$

## 3.3 Dynamic Pooling

After extracting structural features,a new tree is generated.Trees' shape and size vary among different programs. Therefore, the extracted features cannot be fed directly to a fixed-size neural layer.Dynamic pooling is applied to deal with this problem.

The one-way pooling,which pool all features to one vector,is simple but works well.. Concretely, the maximum value in each dimension is taken from the features that are detected by tree-based convolution.

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.

### 3.3.1 The "Continuous Binary Tree" Model

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

A modle "continuous binary tree" is proposed.For node xi in a window, its weight matrix for convolution $W_{conv}$,i is a linear combination of $W_{conv}^t$, $W_{conv}^l$,$W_{conv}^r$with coefficients $n_i^t$,$n_i^l$,$n_i^r$ respectively.

$n_i^t = \frac{d_i - 1}{d - 1}$($d_i$: the depth of the node i in the sliding window; $d$: the depth of the window.)

$n_i^r = (1 - n^t)i)\frac{p_i - 1}{n - 1}$($p_i$: the position of the node; $n$: the total number of p's siblings.)

$n_i^l = (1 - n_i^t)(1 - n_i^r)$

## 3.4 Experiment Results

### 3.4.1 Program Vector Representations

The hierarchical clustering result based on a subset of AST nodes is consistent with human understanding of programs.Compared with random initialization, pre-training makes supervised training one-third faster.This shows that

| 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** |

Figure 2: Accuracy of Program Classification

| 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** |

Figure 3: Accuracy of decting bubble sort

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.Hence, their representations can be adequately tuned in a supervised fashion.

#### 3.4.2 Classifying Programs by Functionalities

TBCNN is applied to classify sourcecode in the OJ system. The target label of a data sample is one of 104 programming problems.At the same time, SVM, DNN and RNN are also doing this work.Figure2 is the result.

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.

#### 3.4.3 Detecting Bubble Sort

TBCNN is used to detect bubble sort,which is an unhealthy code pattern.SVM is also used to compare with TBCNN.Figure3 is the result.

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

## 3.5 Conclusion

In the article,deep neural networks is aplied 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 the 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.