基于卷积和和循环神经网络的自动代码特征提取技术

**摘 要**

近几年来神经网络已经在众多领域取得了突破性的进展。在软件工程领域，神经网络强大的自动特征提取能力极大程度上缓解了传统的由人工制定启发式规则进行手工提取特征的压力。相比于自然语言，代码中的单词之间有更强的依赖关系和逻辑结构。因此该领域亟需解决的一个重要问题就是如何对潜藏在代码中的特征进行编码。基于编码得到的代码向量，一些上层的具体任务便可得到高效地解决。传统的信息检索方式以及近几年来的一些处理代码的深度学习模型简单将代码视为自然语言进行处理，将其转变成一维序列结构而忽略代码元素之间的相互依赖关系来提取信息。一些研究人员借助代码的抽象语法树(AST, abstract syntax tree)提取代码的结构信息，然而AST的规模往往过于庞大直接处理整棵树会带来严重的梯度消失问题。因此，在该论文中，我们提出了一个新的对AST切割后的子树序列进行处理的深度学习模型CVRNN（convolution and recurrent neural network），利用卷积神经网络对潜藏在这些子树的结构信息进行提取，应用双向循环神经网络(biRNN, bidirectional recurrent neural network)对代码块之间的序列信息进行提取，借助代码分类任务训练模型，拟合编码器内部参数。之后利用分类任务训练好的编码器对代码进行编码，基于得到的代码向量，对近似代码搜索任务展开研究。实验结果表明，近似代码搜索Top1、NDCG、MRR的值分别能达到0.56,、0.679和0.638，对比近几年顶尖的用于提取代码特征的深度学习模型以及常用的代码相似度检测工具有显著的优势。

**关键词 代码特征提取，代码分类，程序理解，近似代码搜索**

**Automated code feature extraction based on convolutional and recurrent neural network**

**Abstract** Neural network has made breakthroughs in various domains. In software engineering domain, the powerful automated feature extraction capabilities of neural network has extremely alleviated the problem of extracting features manually. Compared to natural language, because of a stronger dependency existing between words in the source code, code has a stronger logical structure. Hence, one key problem is how to extract this logical structure information and compress into a code vector in fixed length. Eventually, a number of concrete tasks could be handled efficiently based on these code vectors. Traditional information retrieval technologies and some recent deep learning models simply process the code as natural language, regarding the code as one-dimension sequence data structure and neglecting the dependencies between code elements. Recently, some researchers use abstract syntax tree(AST) to represent source code and then extract structure information. However, the size of AST is usually too large and existing models often directly process the entire AST which these models are vulnerable to the gradient vanishing problem. In this paper, we split AST into a sequence of small trees with logical information, then use convolutional neural network to extract structure information hidden in these child AST trees and bidirectional recurrent neural network to extract the sequence information between code blocks. We use the code classification to train the model and then apply the model in a common program comprehension task: similar code search, to demonstrate this way of generating code vector can better compress the information hidden in source code. The metric Top1, NDCG, MRR value could be up to 0.56 0.679 and 0.638 respectively. compared with the state-of-the-art code feature extraction deep learning model and traditional code similarity detection tools, our model has significant advantages.

**Keywords** code feature extraction, algorithm classification, program comprehension, similar code search

# 1、研究的背景和意义

在“大代码”背景的驱动下，如何高效地提取代码特征并对其进行编码以快速地处理海量数据的需求已日益迫切。如今，深度学习在自然语言处理的很多领域已经取得了突破性的进展，人工智能也逐渐从“感知智能”迈向“认知智能”，机器不仅能够感知客观世界所释放的信息，更能够对这些信息向人一样进行理解和推理。Hindle等人[1]已经论证了程序语言和自然语言类似，具备众多可供分析的统计属性，因此代码语言可以和自然语言一样能够被机器理解和分析。许多学者简单的将代码作为自然语言来处理。例如，代码就被表示为一个字符串序列应用在克隆检测[2, 3]，漏洞定位[4]以及代码作者身份分类（code authorship classification）[5]的任务当中。尽管代码和自然语言有很多共性的特征，都是由一系列单词组成且都能表示成语法树的形式，然而代码有许多自己专有的特性，代码具有更强的逻辑结构，有自定义的标志符，标志符之间存在长距离依赖。因此，简单地将代码视作自然语言进行处理必然会造成严重的信息丢失。为了使模型更适用于处理代码语言，部分学者借助软件工程的领域知识，制定了一些启发式规则来静态地提取代码特征并利用机器学习方法来解决克隆检测[6]以及漏洞检测[7]等问题，然而这种静态提取代码特征的方式有以下三个弊端：

1. 完全依赖开发者的先验知识来提取特征，所提取的特征数目有限。
2. 当系统过于庞大和复杂，特征规则的制定也会相应变的复杂，难以适用于对海量且结构复杂代码数据的处理。
3. 规则的制定往往是面向特定任务的，可迁移性差。

因此结合深度学习强大的自动特征提取能力以及代码所独具的特性来解决软件工程领域的各项问题便成了近几年软件工程领域的一个主要研究方法。许多的研究工作都利用代码的AST作为其重要的特征提取来源。例如：Hu等人[8]就通过添加括号的方式来限AST中定节点的作用域，将代码的AST转化成一个节点序列来生成相应的注释；Alon等人[9]借助函数体的AST路径来生成相应的函数名；White等人[10]则分别根据代码的标志符序列以及AST节点序列得到代码的语义向量和结构向量，根据这两个向量对代码克隆检测展开研究。然而这些方法有一个明显的弊端，就是都将树的二维结构转化成一维的序列结构进行预处理，这种降维的处理方式必然会造成节点之间某些依赖关系的损失，从而最终的代码向量也不包含这部分丢失的信息。

因此，部分学者提出了不降维而直接处理AST的树形深度学习模型。与White的方法类似，Wan等人[11]也将代码分为两部分作为输入，使用循环神经网络（RNN，recurrent neural network）对代码的标志符序列进行编码得到语义向量，不同的是，Wan等人使用Tree-LSTM模型[12]去处理AST，得到代码的结构向量而并没有简单地将AST转化成节点序列来处理。Wei等人[13]同样使用Tree-LSTM对克隆检测展开研究。Mou等人[14]提出了一种基于树的卷积神经网络模型TBCNN，直接在AST上进行卷积计算，然后使用动态池化技术将不同规格的AST压缩成代码向量，该向量能够很好地提取代码中的结构信息，在算法分类任务有很好的效果。然而这些树模型有两个局限性：第一，与自然语言的长文本类似，当AST十分庞大，深度足够深的时候，很容易在模型的训练过程中出现梯度消失的情况[15-17]，因此无论是Wan等人使用Tree-LSTM或者Mou等人使用的卷积网络直接对AST进行编码，都会丢失那些存在长期依赖关系节点之间的部分信息。第二，Wan等人使用的Tree-LSTM要先将AST先转化为二叉树再进行编码，该预处理操作破坏了AST原始的结构且转化后的树的深度将会大幅增加，加剧了梯度消失所带来的的问题。

为了解决上述基于基于树的神经网络的问题，一种解决方法就是获得代码的控制流图以及数据依赖图，静态地建立节点与节点之间的联系，将代码表示成一种图的数据结构，最终使用图嵌入技术[18]对代码进行表示。例如Allamanis等人[19]就通过相同的函数以及变量名来静态地建立这些函数以及变量之间存在的依赖关系，Tufano等人[20]则直接构建程序的控制流图来补充节点之间的控制关系。然而程序中元素的依赖关系往往要借助编译后代码的中间表示或者字节码才能得到[21]。在现实环境中，很多代码不能够被编译，因此这些方法所适用的范围将会受到很大的限制。

在这篇论文中，我们结合TBCNN以及Zhang等人[22]提出的ASTNN模型构建了一个基于卷积和循环神经网络的自动代码特征提取模型CVRNN。借鉴ASTNN切割AST的思想，模型并未直接对整棵AST进行处理，而是根据AST中的if、while、for以及function这4个程序控制块的根节点将AST切割成一系列子树，利用TBCNN模型对这些子树进行卷积运算。子树的规模远小于原先的AST，因此可以有效地解决节点长期依赖导致的梯度消失问题。每棵子树对应着一个代码块，为了保存这些代码块之间的序列信息，我们采用双向循环神经网络[23]，内部神经元采用LSTM[24]，对TBCNN生成的代码向量进行后续处理， 将每个时间步生成的代码向量存放到一个向量矩阵中，最终采用最大层池化得到代码向量。我们使用上述方法在代码分类任务上进行训练，训练数据采用Mou[14]等人提供的算法分类领域公开的数据集。我们假设：面向分类任务的深度学习模型能够将代码内部潜藏的结构信息编码进固定长度的代码向量中，且两个代码片段越相似，代码向量彼此之间的几何距离就越短。在后面的实验部分会论证这一假设。基于如上假设，我们使用训练好的分类编码器对代码进行编码，根据编码后的向量进行近似代码搜索的实验。实验数据是从leetcode编程网站上选取的100个编程题，每个编程题对应两个题解，这些题解都经过系统调试并通过。利用分类任务训练好的编码器对这些题解进行编码，在搜索过程中使用编程题的其中一个题解作为查询语句，另一个作为匹配对象放入数据库中。为了扩充搜索空间，我们向数据库中添加负样本作为干扰项，采用Top，NDCG以及MRR作为搜索的度量指标，结果显示，各项指标均高于当前顶尖的用于特征提取的深度学习以及常用的代码相似检测工具。该论文的主要贡献是：

1. 提出了一个树形结构的词向量训练模型。
2. 提出了一个基于卷积和循环神经网络的自动提取代码特征的深度学习模型，该模型能够高效地提取代码的结构信息，且所提取的特征具有很强的迁移能力，不局限于具体的训练任务。
3. 构建了一个高效的用于搜索近似代码的系统。

# 2、模型

模型部分主要分为三个模块：（1）基于树的word2vec词向量训练模型。（2）基于卷积以及循环神经网络的特征提取模型。（3）代码分类以及近似代码搜索模型。

## 2.1 基于树的word2vec词向量训练模型

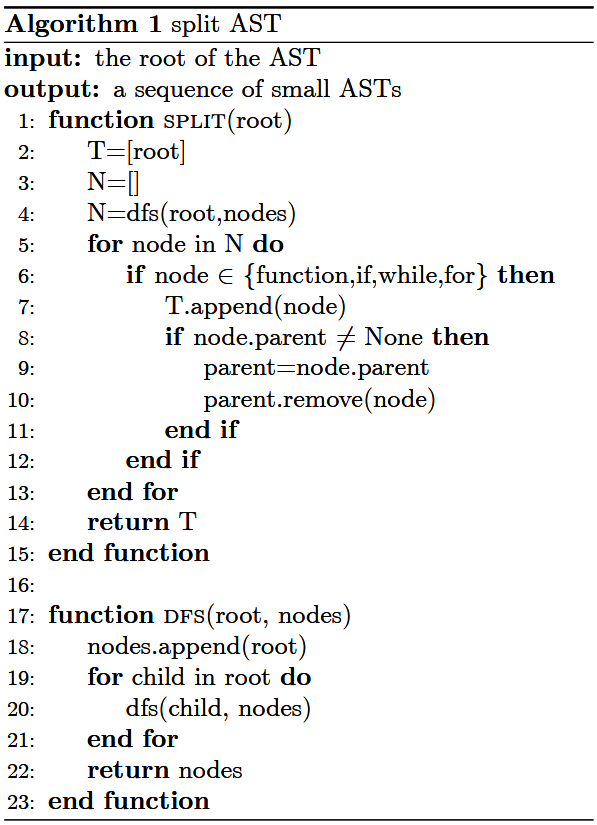
词嵌入技术是将自然语言文本中的单词嵌入到低维空间的一种向量表示方法[25]。近几年，许多学者也将该方法成功地应用在处理代码的具体任务中，如api推荐[26, 27]，漏洞检测[28]等等。然而这些词嵌入技术简单地将代码视为序列化的文本进行处理，往往采用类似Word2vec[29]滑动窗口的方式建立特定单词的上下文情景，并以此为依据来生成相应的词向量，如Zhang等人[22]就通过先序遍历的方式是获得AST的节点序列，然后根据这个序列使用Word2vec生成词向量，而代码元素之间的依赖关系其实是一种二维的树形结构，应用这种方法获取词的情景必然会导致大量代码的结构信息难以编码进生成的词向量中。Mou等人[14]在对AST中节点进行编码时只将当前节点的孩子节点作为情景，这种情景信息的提取方式仍然不完备。因此本文借鉴Perez等人[30]构造树中节点上下文情景的方法，以当前节点的祖先节点、兄弟节点以及子孙节点作为情境，在提取祖先节点以及子孙节点作为情境的过程中分别设置一个向上以及向下探测的参数来控制深度，并设计一个类似Word2vec的词向量训练模型根据提取的情景来生成AST节点的词向量。为了提高近似代码搜索的准确率以及提高模型的训练速度，将不考虑AST叶子节点的信息即代码中的标志符信息。在近似代码搜索任务中，我们使用的数据集是从leetcode[[1]](#footnote-1)编程网站上选取的100个题目，每个题目对应两个题解，用作查询对，这些题解都被编译测试通过，在训练过程中，我们使用的是Mou等人[14]提供的104算法分类数据集。在模型训练过程中不使用AST叶子节点信息原因有二：（1）由于代码中的标志符是程序员自定义的，而据我们观察104算法分类数据集的词汇表与leetcode的词汇表并不重叠（2）题解所使用的变量名仅仅起一个占位符的作用并没有语义表示的功能，更多的是采用a，b，temp等不携带语义信息的单词作为变量名。因此在训练过程中提炼训练集AST叶子节点的语义对近似代码搜索任务并没有多大的帮助。我们使用srcml来生成代码的AST，该工具不检查代码的正确性，因此即使编译不能通过的代码也能生成AST，使模型训练更多的数据。在算法分类以及近似代码搜索的任务中，这些代码语言都是C/C++，而srcml用来表示这些代码结构所使用的预定义节点只需用到60个，因此在整个模型的训练过程中，词汇表的大小将仅仅只有60，加入叶子节点后词汇表的大小将扩充至上万，这将大大地减缓模型的收敛速度。

图1（a）展示了节点情景信息的提取过程，图（b）展示了词向量的训练流程。选取节点b为当前节点，向上探测寻找祖先节点的深度设置为1，向



图1: (a)节点情景信息的提取（b）词向量训练模型

Fig1:(a)context information extraction (b) node embedding training model

下探测子孙节点的深度设置为2，因此得到祖先节点情景信息a，兄弟节点情景信息c，以及子孙节点情景信息（d,e,f,g,h）。在模型的训练过程中，首先我们使用正太分布对词汇表矩阵进行初始化，得到矩阵，其中表示词汇表的大小，表示词向量的维度。模型的输入时分为两部分，情景节点以及当前节点在词汇表中的索引，其中表示词汇表的大小，表示词向量的维度。模型的输入分为两部分，情景节点以及当前节点在词汇表中的索引，通过查找词汇表分贝可以得到情景节点以及当前节点的词向量表示，分别对应图中的context embedding以及current vector，之后context embedding经过两层全连接神经网络得到变换后的情景向量context vector，我们使用交叉熵作为损失函数来进行反向传播，为了生成的context vector能够与current vector进行交叉熵运算，输出层的输出向量的维度与词汇表中向量的维度相同。整个计算过程的数学公式如下：



其中，表示经过隐层后得到的输出向量，

是隐层的权值矩阵，用来将维度是的向量映射成维向量，是隐层网络的偏置项。是输出层的权值矩阵，是输出层的输出向量，是当前节点的词向量，表示向量在第个维度上的值，同理。

通过后面的消融实验中可以得出这样的结论：使用该方法得到的词向量不仅能加快整个代码分类模型的收敛速度，同时也会提高代码分类以及近似代码搜索的精度。

## 2.2基于卷积以及循环神经网络的代码分类模型

### 2.2.1 AST切割

图2 切割AST算法

Fig2 the algorithm of splitting AST

我们定义一个用于存储AST分裂节点的集合S={if, while, for, function, unit}。其中if用来提取条件子树，while、for用于提取循环子树，function

用来提取方法体子树，unit是未切割前AST的根节点，因此AST剩余的节点都在该子树上。图2展示了具体的切割算法。该算法以AST的根节点作为输入。首先用根节点初始化一个数组T，后面将应用该数组存储切割后的子树序列。通过深度优先遍历获得节点序列N，对N中的节点进行遍历，若节点属于集合{function, if, while, for}，该节点连同其子节点将会被切割取出存储到T中。值得注意的是，当出现嵌套的情况，例如if子树中还包含for子树，for子树将会从if子树上提前切割下来，因为深度优先遍历将先访问for再访问if节点，最终得到两个独立的子树将按序存储到T中。

### 2.2.2卷积以及循环神经网络模型



图3 TBCNN 模型

Fig3 TBCNN model

在介绍CVRNN模型之前之前，先详细地阐述一下Mou等人[14]提出的基于树的卷积神经网络TBCNN。

图3展示了该模型执行的具体流程。首先应用词嵌入技术将AST中的节点转化成向量表示，不

同的是，该方法只使用了被表示节点的孩子节点作为情境，没有考虑到兄弟节点以及祖先节点的信息，具体的词嵌入公式如下：



其中表示双亲节点，表示孩子节点，是的

权值矩阵，（孙子节点的数目除以孩子节点的数目）是系数，是偏置项。然后通设置一个固定深度的滑动窗口遍历整个AST再引入最大层池化得到一个形状与原先一样的AST，最后使用动态池化[31]以及两层全连接神经网络得到最终的代码向量。

图4展示了CVRNN模型代码向量具体的生成过程。首先根据2.1得到的词向量表将AST中的节点表示为向量，再使用TBCNN模型对AST进行编码得到编码后的向量，然后使用双向循环神经网络以LSTM作为神经元对得到的向量进行编码以提取代码的序列信息。标准的RNN是单向的，其编码受限于过去的信息，采用双向RNN则能够同时使用过去和未来两个方向的信息。在后面的实验分析中，我们将对这个两个不同的策略进行对比。

给定一棵代码的AST，假设被切割成棵子树，这些子树经过TBCNN编码后将得到一个向量序列，令表示这个序列，表示每个向量的维度。在某个时间步，LSTM的计算公式如下：





图4 CVRNN模型

Fig4 CVRNN model

其中 是sigmoid激活函数，表示新状态，是输出向量，是权值矩阵，

是偏置项。表示输入门，决定中哪一部分将被保留添加到中，表示遗忘门，决定的哪一部分将被遗忘，是输出门用来计算输出向量。整个双向RNN的计算可以形式化如下：



至此，每棵子树将会变转化成一个向量。接着我们将所有的向量放入到一个矩阵中，考虑到不同的子树的重要程度可能不同，例如if代码块中的语句要多于for代码块中的语句，直观上该if子树所携带的信息将高于for子树，因此这里我们选取最大层池化，而没有使用均值池化得到代码向量。

## 2.3代码分类以及近似代码搜索模型

### 2.3.1 代码分类

代码分类流程如图6所示，经过CVRNN的编码，任意代码段都能转换成一个固定长度的向量表示，为了使模型适配104代码分类任务，在该模型的基础之上再添加了一层全连接网络，将映射成维度是104的向量。使用one-hot方法生成每个代码的标记向量，标记向量的维度是104，若代码属于第5类，则代码向量的索引位置5上的值为1，其余位置为0。依旧选取交叉熵作为损失函数训练该分类网络。本文的主要目标并不是为了在代码分类任务上取得更大的进步，Zhang等人[22]的ASTNN模型已经做的足够好，在104代码分类任务上取得了98.2%的分类精度，值得一提的是，在后面的实验中ASTNN模型并没有取得这么高精度的一个重要原因是在我们的工作中并没有使用AST叶子节点的信息，而在原文的工作中，ASTNN对叶子节点也进行了编码。本文所着力研究的是代码理解领域的另一个问题­—近似代码搜索。近似代码搜索直接使用分类模型训练好的编码器对代码向量进行编码，因此为了让编码器更精确地学习代码特征，我们只将104数据集分为两个部分，训练集和验证集，验证集仅有1000条数据，剩余的51000条数据全都归入训练集，使编码器获得更多的训练数据从而更好地拟合内部参数提升最终近似代码搜索的效果。

### 2.3.2 近似代码搜索

我们从leetcode上选取了100个题目，每个题目有两个题解，这些题解都被测试通过。一个题解用作查询，另一个用作匹配对象，这100个匹配对象构成的数据库对应图6中的正样本，为了使近似代码搜索更贴近真是的环境，我们扩充了代码的搜索空间。使用104数据集中的52000条数据作为负样本，正样本和负样本共同构成了代码搜索的数据库。图6是代码向量的生成过程，应用代码分类模型训练好的CVRNN模型对数据库中的代码进行编码最后将代码向量存放到一个数据库中。图7是代码查询的过程，模型的输入是每个题目的一个题解，经过CVRNN编码后得到代码向量。我们借助facebook所提供的faiss[32]工具根据与查询向量之间的欧式距离对数据库中的代码向量进行排序，选择前10个搜索结果作为结果统计样本。搜索时间可以忽略不计，在普通笔记本上的搜索时间小于2ms。

# 3.实验论证

## 

图6 代码分类 图7 近似代码搜索

Fig6 code classification Fig7 similar code search

## 3.1 CVRNN模型在分类任务上的效果如何？生成的代码向量是否满足相似度越高，彼此之间的几何距离就越短的性质？在近似代码搜索任务上的实验结果如何？

表1 CVRNN 参数

Table1 CVRNN hyperparameters

|  |  |
| --- | --- |
| **hyperparameter** | **value** |
| Embedding dimension | 40 |
| Convolutional layer dimension | 150 |
| Number of convolutional layers | 5 |
| Birnn hidden dimension | 200 |
| Initial learning rate | 0.0001 |
| Learning rate decay | 0.95 |
| L2 penalty | None |

为了更好地论证模型的性能，我们选取TBCNN以及ASTNN作为深度学习的对比模型。在利用这三个深度学习模型进行训练之前，先使用

2.1节介绍的词向量训练模型生成AST节点的词向量表，3个深度学习模型将使用同一张词向量表来初始化词向量，以此保证对比的公平性。由于TBCNN模型嵌入在CVRNN模型之中，因此CVRNN该部分所使用的参数与对比的TBCNN模型完全相同。为了保证在近似代码搜索任务上，编码器最终输出的代码向量的维度相同，若TBCNN模型卷积层维度设为150，则最终的的代码向量的维度也是150，而CVRNN以及ASTNN模型因为都使用了双向循环神经网络，因此二者生成的代码向量的维度均是400，为了对比的公平性，在TBCNN卷积层之后再添加一个线性层，将150维的向量变换成400维的向量，即是最终的代码向量，当然这一层在代码分类任务中添加，参与训练，在分类过程中是模型的隐层，上面还有一层输出层能够将400维的向量编码成104维的代码向量，然而在近似代码搜索任务中该层是编码器的输出层。另外选取了3个常用的代码相似度检测工具Stanford moss[[2]](#footnote-2)、jplag[[3]](#footnote-3)以及sim[[4]](#footnote-4)作为非深度学习的对比模型。此外我们还将数据部署在一个开源的代码搜索引擎search code[[5]](#footnote-5)上进行测试，该搜索引擎也是通过代码在数据库中去搜索近似的代码，然而实验结果是0，因此没有在实验结果统计表中列出。在近似代码搜索中，应用以下三个在搜索领域广泛使用的度量指标来衡量模型的效果：

（1）top@k



Q表示查询样本，表示查询样本的数目，

若前个候选样本中有一个匹配，则认为该查询样本命中结果，表示命中结果的查询样本数目。

（2）NDCG



表示候选样本的数目，默认为10。表示第个候选样本与查询样本的相关程度，在近似代码搜索任务中，若候选样本与查询样本匹配则值为1，反之为0。以及的定义同上。IDCG表示DCG的最优计算结果，假设为5，前5个搜索结果与查询样本的相关程度分别为(0,0,1,0,0)，则使用（1,0,0,0,0）计算IDCG的值，（0,0,1,0,0）计算DCG得值，以此得到NDCG的值。

（3）MRR



表示第个样本第一个命中的位置索引，以及的定义同上。

图8展示了经过每一轮训练之后，三个深度学习模型在验证集上的分类精度。可以看出CVRNN模型相对于其它两个模型收敛的速度更快，在经过1轮训练之后，CVRNN模型在分类的精度就达到了76.5%，而ASTNN以及TBCNN则分别是57.7%和65%（对应图8纵轴的截距）。经过30轮训练之后，CVRNN模型的分类精度达到94.4%，而ASTNN以及TBCNN则分别是90.7%和80.9%。在这30轮的训练过程中，可以看出，CVRNN的每一轮的验证精度均高于另外两个深度学习模型。



图8 3个深度学习模型每训练一轮的验证精度

Fig8 the valid accuracy of three deep learning models in each epoch

经过30轮的分类训练，我们使用训练好的编码器对代码向量进行编码，3个深度学习模型最终得到的代码向量的维度均是400，图9展示了使用t-sne方法对leetcode 100个题目构成的问答对代码向量降维打印后的图片，可以看出很多同一个问答对中的代码靠的很近，因此也验证了我们如上的假设，使用CVRNN模型可以将相似的代码向量聚集在一起，相似度越高，彼此之间的几何距离就越短。

表2展示了各个模型在近似代码搜索上的实验精度，3个深度学习均采用第30轮训练结束之后的编码器。实验结果表明：深度学习模型的实验效果相对于3个传统的相似度检测工具有明显的优势，CVRNN在各项度量指标上均高于其它任何一个模型。尽管在分类精度上TBCNN要劣于ASTNN，ASTNN要高出9.8个百分点，然而除了top1二者的精度相等之外，TBCNN其他几项的精度均高于ASTNN。这说明，尽管可以借助代码分类任务训练编码器，然而模型在两个任务上所取得的性能并不是正相关的，即在代码分类任务上精度越高并不代表近似代码搜索的效果越好，然而CVRNN模型在两个任务上的精度都是最高的。

表 2 近似代码搜索准确率

Table2 similar code search accuracy

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **model** | **Top1**  **/%** | **Top3**  **/%** | **Top10**  **/%** | **NDCG**  **/%** | **MRR**  **/%** |
| jplag(max) | 0.25 | 0.27 | 0.31 | 0.278 | 0.268 |
| jplag(avg) | 0.25 | 0.29 | 0.31 | 0.282 | 0.274 |
| moss | 0.32 | 0.33 | 0.33 | 0.326 | 0.325 |
| sim | 0.44 | 0.46 | 0.46 | 0.452 | 0.450 |
| ASTNN | 0.50 | 0.58 | 0.62 | 0.560 | 0.540 |
| TBCNN | 0.50 | 0.62 | 0.75 | 0.622 | 0.581 |
| CVRNN | **0.56** | **0.69** | **0.81** | **0.679** | **0.638** |

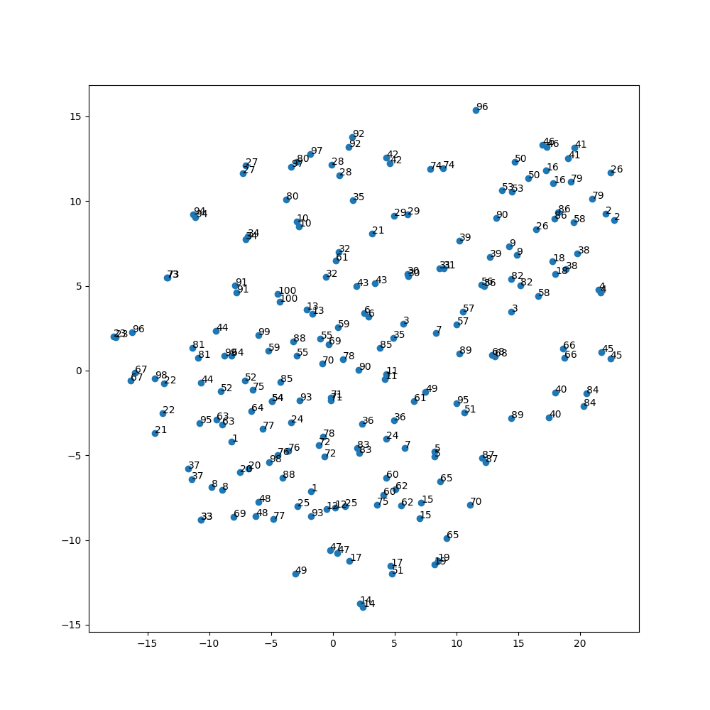


图9 leetcode代码向量

Fig9 leetcode code vector

## 3.2 CVRNN各模块对整个模型性能的影响

下面我们对CVRNN模型的双向循环神经网络模块以及词向量模块做具体分析。我们设计了如下3个模型：

模型1：将双向循环神经网络直接从模型移除。

模型2：将双向循环神经网络 替换成单向的循环圣经网络。

模型3：使用正太分布随机初始化词向量。

模型1直接将双向循环神经网络移除，表示整个CVRNN模型将不提取代码的序列信息，而模型2将双向循环神经网络替换成单向循环神经网络，因为双向循环神经网络是双层的，为了对比的公平性，采取叠加的方式构造双层循环神经网络，每一层均是单向的，每个时间步仅根据过去的信息进行编码。从表3可以看出尽管模型2试图去挖掘代码的序列信息，然而在各项度量指标上均要劣于模型1，而使用双向循环神经网络的CVRNN在各项度量指标上均要高于其他对比的模型。

图10 展示了随机初始化词向量以及使用预训练词向量在训练过程中每一轮在验证集上的分类精度。可以看出使用预训练词向量模型将收敛的更快，经过第一轮训练之后，随机初始化词向量模型的分类精度为72%，而使用预训练词向量的模型精度能达到76.5%。并且在每一个训练轮次之中，使用预训练向量的模型都有更高的精度，最终达到94.4%的分类精度，而随机初始化词向量的模型最终的分类精度是91.4%。

表3 CVRNN 消融实验

Table3 CVRNN ablation research

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **model** | **top1** | **top3** | **top10** | **ndcg** | **mrr** |
| CVRNN(rnn) | 0.46 | 0.58 | 0.68 | 0.568 | 0.533 |
| CVRNN(without bi-rnn) | 0.49 | 0.63 | 0.75 | 0.612 | 0.569 |
| CVRNN(random) | 0.55 | 0.68 | 0.79 | 0.666 | 0.627 |
| CVRNN | **0.56** | **0.69** | **0.81** | **0.679** | **0.638** |



图10 CVRNN随机初始化词向量以及使用预训练词向量每一轮的分类精度对比

Fig10 the valid accuracy of CVRNN versus between using random innitializing node embedding and pretraining embedding in each epch

1. Hindle, A., et al. *On the naturalness of software*. in *2012 34th International Conference on Software Engineering (ICSE)*. 2012. IEEE.

2. Kamiya, T., S. Kusumoto, and K.J.I.T.o.S.E. Inoue, *CCFinder: a multilinguistic token-based code clone detection system for large scale source code.* 2002. **28**(7): p. 654-670.

3. Sajnani, H., et al. *SourcererCC: Scaling code clone detection to big-code*. in *Proceedings of the 38th International Conference on Software Engineering*. 2016.

4. Zhou, J., H. Zhang, and D. Lo. *Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports*. in *2012 34th International Conference on Software Engineering (ICSE)*. 2012. IEEE.

5. Frantzeskou, G., et al., *Examining the significance of high-level programming features in source code author classification.* 2008. **81**(3): p. 447-460.

6. Chilowicz, M., E. Duris, and G. Roussel. *Syntax tree fingerprinting for source code similarity detection*. in *2009 IEEE 17th International Conference on Program Comprehension*. 2009. IEEE.

7. Steidl, D. and N. Göde. *Feature-based detection of bugs in clones*. in *2013 7th International Workshop on Software Clones (IWSC)*. 2013. IEEE.

8. Hu, X., et al. *Deep code comment generation*. in *Proceedings of the 26th Conference on Program Comprehension*. 2018.

9. Alon, U., et al., *code2seq: Generating sequences from structured representations of code.* 2018.

10. White, M., et al. *Deep learning code fragments for code clone detection*. in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2016. IEEE.

11. Wan, Y., et al. *Improving automatic source code summarization via deep reinforcement learning*. in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 2018.

12. Tai, K.S., R. Socher, and C.D.J.a.p.a. Manning, *Improved semantic representations from tree-structured long short-term memory networks.* 2015.

13. Wei, H. and M. Li. *Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code*. in *IJCAI*. 2017.

14. Mou, L., et al. *Convolutional neural networks over tree structures for programming language processing*. in *Thirtieth AAAI Conference on Artificial Intelligence*. 2016.

15. Bengio, Y., P. Simard, and P.J.I.t.o.n.n. Frasconi, *Learning long-term dependencies with gradient descent is difficult.* 1994. **5**(2): p. 157-166.

16. Hochreiter, S.J.I.J.o.U., Fuzziness and K.-B. Systems, *The vanishing gradient problem during learning recurrent neural nets and problem solutions.* 1998. **6**(02): p. 107-116.

17. Le, P. and W.J.a.p.a. Zuidema, *Quantifying the vanishing gradient and long distance dependency problem in recursive neural networks and recursive LSTMs.* 2016.

18. Ou, M., et al. *Asymmetric transitivity preserving graph embedding*. in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. 2016.

19. Allamanis, M., M. Brockschmidt, and M.J.a.p.a. Khademi, *Learning to represent programs with graphs.* 2017.

20. Tufano, M., et al. *Deep learning similarities from different representations of source code*. in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 2018. IEEE.

21. Myers, E.M. *A precise inter-procedural data flow algorithm*. in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1981.

22. Zhang, J., et al. *A novel neural source code representation based on abstract syntax tree*. in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019. IEEE.

23. Schuster, M. and K.K.J.I.t.o.S.P. Paliwal, *Bidirectional recurrent neural networks.* 1997. **45**(11): p. 2673-2681.

24. Gers, F.A., J. Schmidhuber, and F. Cummins, *Learning to forget: Continual prediction with LSTM.* 1999.

25. Henkel, J., et al. *Code vectors: Understanding programs through embedded abstracted symbolic traces*. in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018.

26. Gu, X., et al. *Deep API learning*. in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016.

27. Nguyen, T.D., et al. *Exploring API embedding for API usages and applications*. in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017. IEEE.

28. Pradel, M. and K.J.T.D. Sen, Department of Computer Science, *Deep learning to find bugs.* 2017.

29. Mikolov, T., et al., *Efficient estimation of word representations in vector space.* 2013.

30. Perez, D. and S. Chiba. *Cross-language clone detection by learning over abstract syntax trees*. in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 2019. IEEE.

31. Socher, R., et al. *Dynamic pooling and unfolding recursive autoencoders for paraphrase detection*. in *Advances in neural information processing systems*. 2011.

32. Johnson, J., M. Douze, and H.J.I.T.o.B.D. Jégou, *Billion-scale similarity search with GPUs.* 2019.

1. https://leetcode-cn.com/ [↑](#footnote-ref-1)
2. http://theory.stanford.edu/~aiken/moss/ [↑](#footnote-ref-2)
3. https://jplag.ipd.kit.edu/ [↑](#footnote-ref-3)
4. https://dickgrune.com/Programs/similarity\_tester/ [↑](#footnote-ref-4)
5. https://github.com/boyter/searchcode-server [↑](#footnote-ref-5)