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，从而具有更强的逻辑结构。因此亟需解决的一个重要问题就是如何提取潜藏在代码中丰富的逻辑结构信息，如何将信息压缩进一个固定长度的代码向量中。基于生成的代码向量，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，将其视为一维序列结构而忽略代码元素之间的相互依赖关系进行处理。一些研究人员借助代码的抽象语法树提取代码的结构信息，然而抽象语法树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。在该论文中，我们将抽象语法树切割成一系列携带逻辑信息的子树，借助代码分类任务进行训练，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，利用双向循环神经网络对代码块之间的序列信息进行提取。我们将该模型应用在一个常见的代码理解任务上-近似代码搜索。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，对比顶尖的用于提取代码特征的深度学习模型以及传统的非深度学习模型有显著的优势。

**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 extreamly alleviated the problem of extracting features manully. 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 retrivel 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 recurrent neural network to extract the sequence information between code blocks. We applied 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 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 method has significant advantage.

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

# 1、研究的背景和意义

在“大代码”背景下，如何提取代码特征对其进行抽象表示以推动软件工程各子领域发展的需求已日益迫切。Hindle等人[1]已经论证了程序语言和自然语言类似具备可供分析的统计属性，利用这些属性可以推动软件工程中许多任务的发展。许多学者应用传统的信息检索方法对这些统计属性进行挖掘，将代码简单地看做自然语言进行处理，所设计的模型也仅仅将代码中的标志符按序作为输入。例如，程序就被表示为标志符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。还有一部分方法借助人工制定的启发式规则静态地提取代码特征利用机器学习方法进行克隆检测[6]以及漏洞检测[7]，然而这些方法有以下三个弊端：

1. 完全依赖开发者的先验知识来提取特征。
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。
3. 这些规则的制定往往是面向特定任务的，可迁移性差。

深度学习是一个数据驱动的端到端自动特征提取技术。在过去的几年中，深度学习已经在软件工程中的许多领域取得了突破性的进展，如代码分类[8-11]，克隆检测[11, 12]，代码总结[13, 14]等等。这些方法所共同面临的一个挑战就是如何对代码进行表示，高质量的表示结果能够极大程度上调高模型提取代码结构和语义信息的能力。深度学习强大的自动特征提取能力在自然语言处理领域取得了许多令人瞩目的成绩，机器翻译[15]，对话系统[16]，推荐系统[17]等等，软件工程领域的许多方法也借鉴自然语言处理领域所应用的深度学习模型完成相应的任务，最广泛的就是借鉴机器翻译模型展开一些生成式方向的工作，如注释生成[18-20]，方法名生成[13]等等。尽管代码和自然语言有很多共性的特征，代码有自己独具的特性，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，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。

近几年来，部分学者提出适用于树的深度学习模型来帮助提取代码的结构信息。例如White等人[21]就将程序的输入分为两个两个部分，代码的标志符序列以及AST节点序列，基于这两部分输入利用深度学习模型得到代码的语义向量和结构向量，合并这两个向量得到最终的代码向量，基于代码向量判断代码是否是克隆。在代码总结任务中，与White的工作类似，Wan等人[18]也将代码分为两部分作为输入，使用基于树的LSTM模型处理处理代码的AST使用另外一个LSTM处理代码的标志符序列，然后使用混合的attention机制拼接这两个向量得到代码向量。Mou等人[8]提出了一种基于树的卷积神经网络TBCNN，直接在AST上进行卷积计算，然后使用动态池化的方法将不同大小的AST压缩得到代码向量，该向量能够很好的提取代码中的结构信息，在算法分类任务有很好的效果。然而这些树的方法有两个局限性，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[22-24]，因此Wan等人[18]直接和对整棵AST进行编码或者Mou等人[8]使用滑动窗口技术对AST进行卷积计算必然会丢失那些存在长期依赖节点之间的关系信息。第二，Wan等人[18]提出的Tree-LSTM将AST先转化为二叉树再进行编码必然会破坏AST原始的结构。转化后的树的深度将会增加进而削弱模型捕捉更加真实且复杂的原始代码信息的能力。

为了克服上述基于基于树的神经网络的困难，一种解决方法就是引入精确的控制流图以及数据依赖图，使用图嵌入技术[25]对代码进行表示。例如Allamanis等人[26]通过相同的函数以及变量来构建程序中存在的长期依赖以此缓解梯度消失带来的问题，Tufano等人[27]则直接构建程序的控制流图。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[28]，然而在现实环境中，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模型以及Zhang等人[11]提出的ASTNN模型构建了一个基于卷积神经网络和循环神经网络的自动代码特征提取模型。参考ASTNN切割抽象语法树的思想，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节点将AST切割成一系列子树，利用TBCNN模型处理这一系列子树，子树的规模远小于原先的AST，因此可以有效的解决代码节点长期依赖导致的梯度消失的问题，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[29]，神经元采用LSTM[30]，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矩阵当中，最终采用最大层池化对每个特征维度选取最大值得到代码向量。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[8]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。基于如上假设，我们使用训练好的编码器对代码进行编码，测试数据是从LeetCode编程网站上选取的100个题解，每个题解对应两段代码，利用训练好的编码器对这些代码进行编码，在搜索过程中使用每个题解的一个代码作为查询语句，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[31]。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[32, 33]，漏洞检测[34]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[35]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，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。因此本文借鉴Perez等人[12]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的词向量训练模型基于提取的情景来生成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，我们将不考虑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所选取的100个题解，而在训练过程中，我们使用的是Mou等人[8]提供的104算法分类数据集，一方面这些题解所出现的词汇与训练的数据集词汇并不重叠，另一方面题解所使用的词汇并不规范，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，因此在训练过程中所提炼AST叶子节点的语义并不能提升后续近似代码搜索模型的性能。我们使用srcml来生成代码的抽象语法树，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用56个，因此在整个模型的训练过程中，只需要训练这56个词向量，加入叶子节点后词汇表的大小将扩充至上万，这将急剧地延长模型的收敛速度。 图1是展示了词向量的具体流程。





在图1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，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 embedding，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 embedding进行交叉熵运算，输出层神经元的数目设置为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，表示情景向量，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。

最终训练得到的词向量将满足如下性质，若节点之间的语义越相近，则相应节点向量之间的几何距离就越短，在后面的消融实验中可以得出这样的结论：使用训练好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

我们定义一个集合，用于存储分割AST的节点，，其中用来提取的条件子树，，用于提取循环子树，用来提取方法体子树，是未切割前AST的根节点，用来保存被以上四个节点切割后剩余的子树。图展示了具体的切割过程。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，来存储切割后的子树序列，这里的就是上文的，然后获得该树的深度优先序列，并对其进行遍历，若节点出现在集合

中，该子树将会别切割存储到中。值得注意的是，当出现嵌套的情况，例如子树中还包含子树，子树将会从子树上切割向量，最终得到两个独立的子树按序存储到中。



### 2.2.2卷积以及循环神经网络模型



在介绍我们的模型CVRNN之前，先详细的阐述一下Mou等人[8]提出的基于树的卷积神经网络TBCNN。

图展示了该模型的具体流程。首先该模型AST中的节应用词嵌入的技术奖每个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，（相对于，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棵AST以及加入一个最大层池化的过程得到一个形状与原先一样的AST，接这使用动态池化[36]以及两层全连接神经网络得到最终的代码向量。



图展示了我们代码向量具体生成过程。首先根据2.1得到的节点向量词汇表将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RNN并选取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：



其中是sigmoid激活函数，表示新地状态，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，例如代码块中的语句要多于代码块中的语句，直观上该子树所携带的信息将高于子树，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

代码分类流程如图所示，经过CVRNN处理，任意代码段都能转换成一个向量表示，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，的维度是104。使用one-hot方法生成每个代码的标记向量，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[11]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，因此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，验证集有1000条数据，剩余的51000条数据全都归入训练集。

### 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个匹配对象对应图中的正样本，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。图左边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。值得注意的是，所产生的代码向量其实就是图中的，而不是代码分类任务中后续经过全连接网络处理后得到的，因此代码向量的长度可以是任意长度，不受到代码类别数目的限制，当然这个长度应在代码分类任务中指定，因为代码搜索直接使用的代码分类任务训练好的CVRNN，不能够后续更改。图右边是代码查询的过程，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。我们使用欧式距离来量化两个向量之间的距离，借助facebook所提供的faiss[37]工具来协助完成搜索工作，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。最终我们根据欧式距离的大小选择前10个搜索得到的代码作为度量指标的计算样本。

## 3.实验论证

## 3.1 CVRNN模型是否能够借助代码分类任务使得编码器自动地提取代码特征，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？

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | top1 | top2 | top3 | top5 | top10 | ndcg | mrr |
| CVRNN(rnn) | 0.46 | 0.56 | 0.58 | 0.62 | 0.68 | 0.568 | 0.533 |
| CVRNN(without birnn) | 0.49 | 0.56 | 0.63 | 0.67 | 0.75 | 0.612 | 0.569 |
| CVRNN(random) | 0.55 | 0.62 | 0.68 | 0.76 | 0.79 | 0.666 | 0.627 |
| CVRNN | **0.56** | **0.64** | **0.69** | **0.74** | **0.81** | **0.679** | **0.638** |

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，深度学习模型在训练之前，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该部分的所使用的参数与对比的TBCNN模型完全相同。因ASTNN模型之中也使用了双向RNN，同样CVRNN该部分所使用的参数也完全相同。

另外选取3个常用的代码相似度检测工具Stanford moss、jplag以及sim作为非深度学习的对比模型。

实验结果如表所示，可以看出

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Top1 | Top2 | Top3 | Top5 | Top10 | ndcg | mrr |
| jplag(max) | 0.25 | 0.27 | 0.27 | 0.31 | 0.31 | 0.278 | 0.268 |
| jplag(avg) | 0.25 | 0.29 | 0.29 | 0.31 | 0.31 | 0.282 | 0.274 |
| moss | 0.32 | 0.33 | 0.33 | 0.33 | 0.33 | 0.326 | 0.325 |
| sim | 0.44 | 0.46 | 0.46 | 0.46 | 0.46 | 0.452 | 0.450 |
| ASTNN | 0.50 | 0.54 | 0.58 | 0.60 | 0.62 | 0.560 | 0.540 |
| TBCNN | 0.50 | 0.60 | 0.62 | 0.69 | 0.75 | 0.622 | 0.581 |
| CVRNN | **0.56** | **0.64** | **0.69** | **0.74** | **0.81** | **0.679** | **0.638** |

## 

|  |  |
| --- | --- |
|  | accuracy |
| TBCNN | 0.809 |
| ASTNN | 0.907 |
| CVRNN | **0.944** |

## 3.2 CVRNN各模块对整个模型性能的影响。

|  |  |
| --- | --- |
|  | accuracy |
| CVRNN(without birnn) | 0.738 |
| CVRNN(rnn) | 0.925 |
| CVRNN(random) | 0.914 |
| CVRNN | **0.944** |



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

9. Ben-Nun, T., A.S. Jakobovits, and T. Hoefler. *Neural code comprehension: A learnable representation of code semantics*. in *Advances in Neural Information Processing Systems*. 2018.

10. DQ, B.N., Y. Yu, and L. Jiang. *Bilateral dependency neural networks for cross-language algorithm classification*. in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2019. IEEE.

11. 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.

12. 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.

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

14. Jiang, S., A. Armaly, and C. McMillan. *Automatically generating commit messages from diffs using neural machine translation*. in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017. IEEE.

15. Vaswani, A., et al. *Attention is all you need*. in *Advances in neural information processing systems*. 2017.

16. Chen, H., et al., *A survey on dialogue systems: Recent advances and new frontiers.* 2017. **19**(2): p. 25-35.

17. Zhang, X., J. Zhao, and J.C. Lui. *Modeling the assimilation-contrast effects in online product rating systems: Debiasing and recommendations*. in *Proceedings of the Eleventh ACM Conference on Recommender Systems*. 2017.

18. 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.

19. Zhou, Y., et al., *Augmenting Java method comments generation with context information based on neural networks.* 2019. **156**: p. 328-340.

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

21. 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.

22. 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.

23. 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.

24. 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.

25. 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.

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

27. 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.

28. 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.

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

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

31. 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.

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

33. 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.

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

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

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

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