DOI:10.7544/issn1000-1239.2019.20190185
56(8): 1605-1620, 2019

基于深度学习的程序理解研究进展

刘 芳 李 戈 胡 星 金 芝
(北京大学信息科学技术学院 北京 100871)
(高可信软件技术教育部重点实验室(北京大学) 北京 100871)
(liufang816@pku.edu.cn)

Program Comprehension Based on Deep Learning

Liu Fang, Li Ge, Hu Xing, and Jin Zhi
(*School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871)
(*Key Laboratory of High Confidence Software Technologies* (*Peking University*), *Ministry of Education, Beijing* 100871)

Abstract Program comprehension is the process of obtaining relevant information in programs by analyzing, abstracting, and reasoning the programs. It plays an important role in software development, maintenance, migration, and other processes. It has received extensive attention in academia and industry. Traditional program comprehension relies heavily on the experience of developers. However, as the scale and complexity of software continue to grow, it is time-consuming and laborious to rely solely on the developer's prior knowledge to extract program features, and it is difficult to fully exploit the hidden features in the program. Deep learning is a data-driven end-to-end method. It builds deep neural networks based on existing data to mine the hidden features in data, and has been successfully applied in many fields. By applying deep learning technology to program comprehension, we can automatically learn the features implied in programs, which can fully exploit the knowledge implied in the program and improve the efficiency of program comprehension. This paper surveys the research work of program comprehension based on deep learning in recent years. Firstly, we analyze the properties of the program, and then introduce mainstream program comprehension models, including sequential models, structural models, and execution traces based models. Furthermore, the applications of deep learning-based program comprehension in program analysis are introduced, which mainly focus on code completion, code summarization and code search, etc. Finally, we summarize the challenges in program comprehension research.

Key words program comprehension; program analysis; software engineering; deep learning; data mining

摘 要 程序理解通过对程序进行分析、抽象、推理从而获取程序中相关信息,在软件开发、维护、迁移等过程中起重要作用,因而得到学术界和工业界的广泛关注.传统程序理解很大程度上依赖开发人员的经验,但随着软件规模及其复杂度不断增大,完全依赖开发人员的先验知识提取程序特征既耗时耗力,又很难充分挖掘出程序中隐含特征.深度学习是一种数据驱动的端到端的方法,它根据已有数据构建深度

收稿日期:2019-03-20;**修回日期**:2019-05-13
基金项目:国家"九七三"重点基础研究发展计划基金项目(2015CB352201);国家自然科学基金项目(61620106007,61751210)
This work was supported by the National Basic Research Program of China (973 Program) (2015CB352201) and the National Natural Science Foundation of China (61620106007, 61751210).
通信作者:金芝(zhijin@pku.edu.cn)

神经网络对数据中隐含的特征进行挖掘,已经在众多领域中获得成功应用.将深度学习技术运用于程序理解中,根据具体任务以及大量数据自动地学习程序数据中蕴含的特征,可以充分地挖掘出程序中隐含的知识,提高程序理解的效率.对基于深度学习的程序理解研究工作进行综述,首先对程序所包含的性质进行分析,然后介绍主流的程序理解模型,包括基于序列、结构以及执行过程的程序理解模型.随后展示基于深度学习的程序理解在程序分析中的应用,主要针对代码补全、代码注释生成、代码检索等任务.最后,分析并总结程序理解研究所面临的挑战.

关键词 程序理解;程序分析;软件工程;深度学习;数据挖掘

中图法分类号 TP311

程序理解(又叫软件理解)是通过对程序进行分析、抽象、推理从而获取程序中相关信息的过程,自1968年软件工程诞生起,程序理解就开始受到关注.程序理解是软件复用、维护、迁移、逆向工程等任务中的首要活动[1],其目的是通过对程序进行多方面、多层次、多角度地分析来提取程序中的相关信息.程序理解是软件工程中的一个重要部分,在整个软件维护与开发过程中起着举足轻重的作用.

早期的程序理解大多基于人为制定的启发式规则确定程序的特征,然后根据所确定的特征从程序中提取出的相关信息并进行分析,分析结果完全依赖于开发人员具备的先验知识.主要方法包括程序静态分析、动态分析等,其中,静态分析直接分析程序源代码,分析过程不需要执行程序[2];动态分析用于理解程序运行时的性质,在程序执行过程中获取其输入输出关系等[3].这些方法主要局限有3个方面:

1) 完全依赖开发人员具备的先验知识来确定需要从程序中提取什么特征;

2) 一旦软件系统过于复杂规模过大,则程序性质分析不能有效地进行,不具备可扩展性;

3) 针对不同程序分析需求,需制定特定规则进行分析,通用性较差.

深度学习是一种数据驱动的端到端的方法,它根据已有数据构建深度神经网络来挖掘数据中隐含的特征.近年来,深度学习已经在语音识别、图像处理、自然语言处理等众多领域中受到广泛关注.程序理解需要从程序中提取出与程序理解任务相关的特征信息,若能运用深度学习技术根据程序理解任务以及已有数据自动地学习程序数据中蕴含的特征,将会极大地提高程序理解的效率.开源社区(例如GitHub①)和编程问答网站(StackOverflow②)等平台上存在大量高质量的开源代码和文档,这些数据中蕴含着许多知识,为研究者们提供了利用深度学习技术挖掘代码和文档数据等中包含的知识来解决程序理解相关问题的新途径.因此,人们开始尝试如何将深度学习技术应用于程序理解任务,利用深度神经网络表征程序中包含的信息,获得程序的特征向量表示,从而替代传统方法中的人工特征提取.

程序理解中常用的深度学习技术包括循环神经网络(recurrent neural network, RNN)、卷积神经网络(convolution neural network, CNN)、序列到序列(seq2seq)模型以及注意力(attention)机制等,它们可以用于不同的程序理解应用场景:

1) RNN通过增加神经网络中隐藏层节点之间的连接引入循环机制来处理序列中前后关联问题,适合对较长的序列进行建模.基于RNN及其变体LSTM[4]和GRU[5]的模型可用于对程序语言的建模,获得程序序列的特征向量表示,因此RNN在代码补全、代码检索、代码克隆检测等任务中被广泛使用.

2) CNN利用卷积核对输入数据进行卷积运算提取数据中的局部特征,擅长于处理具有局部性的数据.在程序理解任务中,CNN可用于提取程序中的局部特征,可被应用于程序分类、程序方法名预测等任务中.

3) 序列到序列(seq2seq)模型则用于从输入序列到输出序列映射的任务中,它包含编码器(encoder)和解码器(decoder)两部分,其中编码器用于将输入序列表示为特征向量,解码器则根据该向量生成输出序列.可用于程序生成、程序注释生成等任务中.

4) 注意力(attention)机制[6]最早在seq2seq模型中被使用,通过引入该机制,神经网络在生成每一个输出时动态地关注输入中与当前输出更相关的内容,在输入较长的情境下也能减轻网络对长序列的

① https://github.com/

② https://stackoverflow.com/

记忆负担,能有效提升模型性能.注意力机制在程序生成、程序注释生成等任务中被广泛使用.此外,它还可用于基于 RNN 的语言模型,在预测下一个元素时可以重点关注上文中更相关的部分.因此,注意力机制也可用于代码补全任务.

5) 指针网络(pointer networks)[7]是利用注意力机制设计的一种神经网络结构.它直接利用注意力机制计算得到的权重作为指针,利用该指针从输入文本中选择最相关的元素作为解码器的输出.See 等人[8]利用指针网络生成文本摘要,以解决摘要生成中包含词表之外的词(out of vocabulary, OoV)的问题.受此启发,近年研究者开始将指针网络应用于代码补全任务中.

上述方法最早被应用于自然语言处理任务中,由于程序语言和自然语言在一定程度上的相似性,这些技术被迁移到了程序理解的相关任务中.但程序语言具有其自身独特的性质,例如更强的结构性、可执行性等,因此设计程序理解模型时需要考虑程序中包含的这些性质.同时,程序理解的不同任务需求对模型的设计也有相应的要求,例如在程序变量命名任务中(代码风格修正),需要考虑程序代码中的数据流和控制流信息,所以一般构建基于图的神经网络进行建模;在程序自动生成任务中,生成的程序需要严格符合语法规则,需要对程序的抽象语法树进行建模.总之,设计程序理解模型,需要重点考虑程序自身的特性以及具体的理解任务需求.

1　程序性质及程序理解框架

1.1　程序性质

程序语言与自然语言有许多相似之处,例如它们都由词素(token)组成,都可以解析为语法树的形式,并且都具有较高的自然性(重复性)[9]等.因此,一些适合于自然语言处理任务的模型可以被迁移到程序理解任务中.例如近几年很多基于统计的语言模型被用于程序语言处理中[9-14],其中包括基于深度学习的统计语言模型.但是,程序语言与自然语言之间仍然有较大的差别,程序语言具有其自身独特的性质,主要包括强结构性、自定义标识符、长依赖和可执行性,这些性质是构建程序理解模型不可忽视的.

1.1.1　强结构性

程序具有更强的结构性,程序中存在着大量的嵌套结构.例如图 1 是 Python 语言实现的冒泡排序的算法①,其中 2 个 for 循环相互嵌套,在内部的 for 循环中包含一个 if 语句,这样相互嵌套语句中蕴含着程序的逻辑结构,这些结构对于程序语义的理解十分关键.在程序语言中,这样的结构无处不在.在一些较长的程序中,嵌套层次往往会非常深.当把程序解析为语法树之后,语法树往往非常大.程序的结构对于整个程序的理解至关重要,也是程序理解中面临的难点之一.

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp=alist[i]
                alist[i]=alist[i+1]
                alist[i+1]=temp
```

Fig. 1　Bubble sort algorithm of Python code

图 1　Python 语言实现的冒泡排序算法

1.1.2　自定义标识符

与自然语言不同,程序中包含着大量用户自定义的标识符,例如函数名、变量名、类名等.在对自然语言建模时,通常需要预先构建一个能够覆盖数据集中绝大多数词语的词汇表.在自然语言相关的任务中,很少会出现创造新词的情况.而程序中却存在着大量用户自己定义的标识符,并且不同用户的命名习惯又不尽相同.因此在对程序代码的 token 建模时,预先构建好的词汇表往往不能满足建模的需求.无论定义多大的词表,总会有许多出现在词表之外(OoV)的新词,这对于程序的理解造成了较大的干扰.

1.1.3　长依赖

在程序语言中,上下文之间依赖间隔可能会很长.例如在程序开始时定义的变量可能会在程序的末尾用到.并且由于程序中存在着作用域,一些局部变量或者方法只会在其各自的作用域中用到,而不会在其他地方用到,而这些局部变量或方法的存在会增加上下文直接的依赖间隔.在程序中,大部分前文信息很可能与当前的 token 是无关的.

1.1.4　可执行性

程序必须是可以执行的,这就要求程序代码必须严格符合语法规则,这也是程序语言与自然语言的一个很大的不同之处.在自然语言中,即使语句中

① 此处不特指 Python 语言,其他语言的实现与之类似

存在一些语法错误,其表达的含义仍然是可以理解的,而程序语言中只要包含一个语法错误,便会导致其无法运行.因此,在程序理解中程序的语法至关重要.

1.2 基于深度学习的程序理解框架

在构建程序理解模型时,程序特性以及具体的任务需求不容忽视.针对不同程序理解任务需求,程序可以被表示为多种形式,以便于对程序中包含的不同特性进行建模.目前主要有 3 种表示形式:

1) 序列.基于序列的表示方法将程序表示为序列的形式,例如 token 序列、字符序列或者 API 序列.

2) 结构.基于结构的表示方法考虑程序语言的结构,将程序表示为抽象语法树(abstract syntax tree, AST)或者程序图(graph)的形式.

3) 执行过程.基于执行过程的表示方法重点关注程序的动态执行过程,将程序运行过程中产生的中间结果作为程序的表示,例如在执行过程中调用的子程序及其参数,或者在执行过程中程序中各个变量值的变化情况.

基于深度学习的程序理解框架可以用图 2 来展示.程序中包含不同性质,不同程序理解任务需求对程序性质的侧重点不同,因此在人们在完成程序理解任务时会选择合适的形式(序列、结构或执行过程)来表示程序,然后构建相对应的程序理解模型对不同形式的程序进行建模.表 1 列出了程序理解中主要任务、任务重点关注的性质以及完成该任务主要采用的程序表示形式.

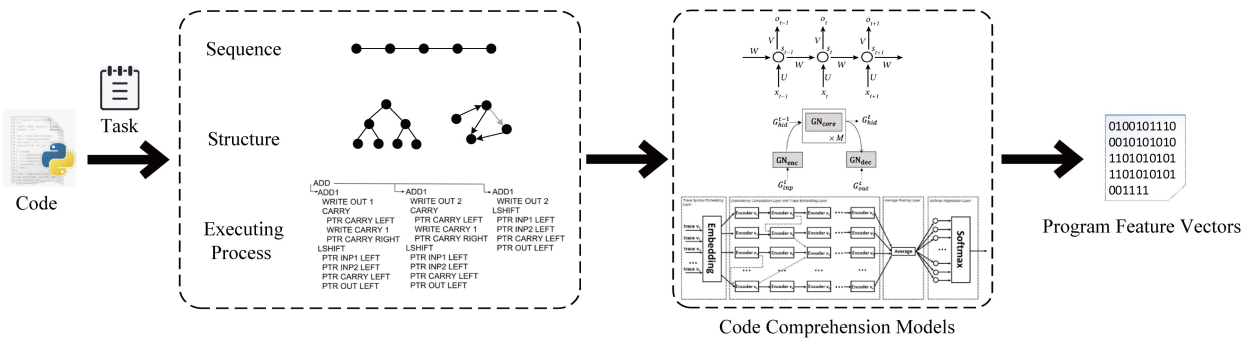

Fig. 2 Program comprehension architecture based on deep learning

图 2 基于深度学习的程序理解框架

Table 1 Program Comprehension Tasks, Program's Natures they Concern and Program Representations they Use.

表 1 程序理解任务、关注性质和采用的程序表示形式

Task	Nature	Representation
Code Completion	Naturaless, Structure	Sequence, Structure
Code Summarization	Naturaless, Structure	Sequence, Structure
Code Pattern Detection	Structure	Structure
Code Generation	Structure, Executability	Structure, Execution Trace
Code Convention Correction	Data Flow, Control Flow	Structure

2 程序理解的深度学习模型

不同程序理解任务根据其需求采用不同的程序表示方式,然后构建相应模型完成具体任务.基于此,本文将基于深度学习的程序理解模型分为 3 类:基于序列的程序理解模型、基于结构的程序理解模型和基于执行过程的程序理解模型,本节结合这些模型的应用场景对这 3 种模型进行详细介绍.

2.1 基于序列的程序理解模型

程序是由符号序列构成的,因此我们可将程序代码形式化地表示为字符(character)或词素(token)序列.除此之外,程序中的应用程序接口(application programming interface, API)序列中蕴含着程序的功能信息,因此 API 序列也成为了许多研究工作的研究对象.目前已有的基于序列的程序理解研究工作主要基于构建统计语言模型来学习程序的序列化表示.对于程序序列 $S=t_1,t_2,\cdots,t_n$

来说,该序列出现的概率为:

$$P(S)=P(t_1,t_2,\cdots,t_n)=P(t_1)\times P(t_2|t_1)\times P(t_3|t_1,t_2)\times\cdots\times P(t_n|t_1,t_2,\cdots,t_{n-1}),\quad(1)$$

其中,$P(t_1)$表示第 1 个词是 t_1 的概率,$P(t_n|t_1,t_2,\cdots,t_{n-1})$表示给定前 $n-1$ 个词的情况下,第 n 个词出现的概率.早期的研究工作大多基于 N-gram 语言模型对程序序列进行建模[9,15].随着深度学习的发展,近年来,基于深度循环神经网络的统计语言模型开始被应用于程序序列建模中.表 2 列出了基于序列的程序理解相关研究工作.

Table 2　Research on Sequence-Based Program Comprehension

表 2　基于序列的程序理解相关研究工作

Type	Task	Reference	Model
Character Sequence	Code Generation	Cummins et al.[12]	LSTM
Token Sequence	Code Completion	Raychev et al.[11]	N-gram,RNN
		White et al.[13]	RNN
		Bhoopchand et al.[14]	LSTM,Pointer Network
	Code Summarization	Allamanis et al.[16]	CNN
		Iyer et al.[17]	Seq2seq
		Hu et al.[18]	Seq2seq,Transfer Learning
	Code Search	Gu et al.[19]	RNN,MLP
	Bug Fixing	Gupta et al.[20]	LSTM,RL
API Sequence	Code Summarization	Hu et al.[18]	Seq2seq,Transfer Learning
	Code Search	Gu et al.[19]	RNN,MLP
	API Sequence Search	Gu et al.[21]	Seq2seq

2.1.1　基于字符序列的程序理解模型

基于字符序列的程序理解模型将程序表示为字符序列,利用深度神经网络对其进行建模,学习代码字符序列中包含的信息.由于程序中包含的字符数目是确定的,因此将程序表示为字符序列在构建词汇表时不会出现 OoV 的问题,这也是采用字符序列表示程序的主要原因.Cummins 等人[12]构建了字符级别的 LSTM 语言模型学习程序,该方法将程序代码视为字符构成的序列.他们基于该模型构建了程序生成的工具 CLgen,利用该工具生成了大量程序作为基准数据(benchmark data),为其他程序理解相关研究提供了测评数据.尽管基于字符序列的程序理解模型不会出现 OoV 问题,但极大地增加了程序序列长度,使得模型很难学习到整段代码的语义.不仅如此,用字符序列表示程序无法捕捉程序中的 token,API 等中包含的信息,这些信息对于程序语义的理解十分重要.

2.1.2　基于词素(token)序列的程序理解模型

由于字符的表达能力较弱,许多程序理解模型将程序词素(token)序列作为研究对象.基于 token 序列的程序理解模型将程序表示为 token 序列,利用深度神经网络对 token 序列进行建模,学习序列中包含的信息.给定部分程序序列,模型可以预测之后最可能出现的 token.因此,该模型可被应用于代码补全任务.除此之外,该模型也可以得到整个程序序列的特征向量表示,基于该向量人们可以完成程序注释生成、代码检索等任务.

在代码补全任务中,一些工作直接构建基于 RNN 的语言模型对 token 序列建模,根据已有 token 序列预测下一个 token.例如 White 等人[13]设计了一个 RNN 语言模型对程序代码的 token 序列建模,为了验证该模型的有效性,他们将其应用于 Java 语言的代码补全任务中并取得了 72.2%的准确率.Bhoopchand 等人[14]则提出了基于指针网络的程序语言模型,该工作旨在解决程序代码中自定义标识符之间的长依赖问题.该方法在构建 LSTM 语言模型的过程中,依次记录程序代码中出现过的自定义标识符,通过指针网络选择标识符作为输出的备选项,最后结合语言模型和指针网络的结果,共同预测输出词语的概率分布.在 Python 语言的代码补全任务的实验结果表明:该模型相较于 N-gram 模型和普通的 LSTM 模型能够取得更高的准确率,并且该方法能够预测出距离当前位置较远的上文中出现的自定义标识符.以上基于 token 序列的代码

补全模型相较于传统基于统计和规则的方法而言能够取得更高的准确率,且实现较为简单.但这些模型也存在着一定的缺陷,例如将程序表示为 token 序列会丢失程序中的结构信息,导致模型不能很好地对程序的结构进行建模,从而无法为代码补全任务提供充分的信息.

在代码注释生成任务中,许多工作基于 seq2seq 框架构建模型完成该任务.这些工作首先将代码表示为 token 序列,然后构建基于 RNN 或 CNN 的编码器对代码进行建模,得到其特征向量表示,最后根据该向量构建解码器生成代码的自然语言描述.例如,Iyer 等人[17]和 Hu 等人[18]使用 RNN 作为编码器对代码进行编码得到其向量表示,然后根据该向量利用解码器生成代码注释,并引入了注意力机制.而 Allamanis 等人[16]则采用基于注意力机制 CNN 对程序建模,提取程序中存在的关键语句以及层次结构性的特征,将该模型应用于代码注释生成(方法名的预测)任务中.该方法将函数体表示为 token 序列,在序列上进行卷积操作并引入注意力机制学习序列和函数名相关的关键信息.同时引入了指针网络来解决 OoV 的问题,在 Java 项目的方法名预测任务上达到了超过 40%的 F1 值.相较于 CNN,RNN 能够记忆更长的代码序列信息,而 CNN 模型更关注于程序局部特征并且模型训练效率较高.以上这些工作相较于传统基于人工定义的规则或模板抽取代码中对应的内容的方法能够生成更加丰富的注释内容,且不需要人为制定规则,模型的可扩展性更强.但是这些基于 token 序列的代码摘要模型在对代码进行建模时未能充分考虑程序中的结构性,且生成的摘要通常比较简短,很难用于生成较为复杂的代码的摘要.

在代码检索任务中,目前基于 token 序列的模型将程序表示为 token 序列,然后构建 RNN 模型得到序列的向量表示,将该向量与其对应的自然语言查询语句的向量映射到相近的向量空间中.在检索时,直接根据查询语句的向量在向量空间中搜索与其相近的代码向量作为检索结果.Gu 等人[19]提出了基于深度学习的代码检索模型,使用 RNN 得到自然语言查询语句与其对应的代码的向量表示,使得这 2 种向量映射到相近的向量空间中.对于自然语言,该方法使用 RNN 以及最大池化方法进行编码得到其向量表示;对于代码,该方法分别对代码中的 token 序列、方法名序列以及 API 序列进行编码,分别采用了 RNN,RNN 以及 MLP 模型,之后进行最大池化操作,然后将这 3 个向量进行组合作为代码的向量表示.

在得到代码和查询语句的向量表示之后,通过余弦相似度函数来衡量这 2 个向量之间的相似度,将查询语句与其对应代码的向量作为正例,然后从代码库中随机挑选一个代码作为负例,通过最大化正例之间余弦相似度和最小化负例之间的余弦相似度来训练模型,使得模型能够将自然语言查询语句与其对应的代码的向量表示映射到相近的向量空间中.实验结果表明,该方法在代码检索任务中较其他方法取得了较大的提升.传统代码搜索方法基于信息检索中的相似度匹配方法来匹配自然语言和代码,只能根据较低层次的特征进行搜索,例如关键词匹配.而运用深度神经网络得到代码序列和查询语言的向量表示然后再对向量之间的空间距离进行优化的方式能够挖掘出更高层次的匹配特征,因此这种方法可以取得更好的效果.

基于 token 序列的程序理解模型还可以应用于代码纠错任务中,Gupta 等人[20]提出了基于强化学习的程序语言纠错模型,该方法首先基于 LSTM 网络对程序中的 token 序列进行编码,得到每一个 token 的向量表示,然后基于强化学习策略实现程序代码纠错.在强化学习策略中,状态(state)是一个程序文本和游标的二元组〈string, cursor〉,其中 string 表示程序文本,cursor 表示游标,用于在学习过程中选择程序中需要修改的位置.模型学习过程中,代理(agent)每一时刻根据当前状态选择要执行的动作(actions),其中动作分为 2 类,分别为游标位置更新动作(navigation actions)和修改程序文本动作(edit actions).当代理能够纠正全部错误时奖励(reward)是最大的(maximum_reward),此外还定义了中间奖励(intermediate_reward),作为纠正不低于一个错误时的奖励.该方法在学习时采用了异步的优势行动者评论家算法(asynchronous advantage actor-critic, A3C)[22]算法.实验结果表明:该方法在 C 语言程序纠错中准确率较当时最好的方法[23]取得了较大的提升.但是该方法目前只能对程序中的简单错误进行纠正,较为复杂的错误目前还没有较好的解决方案.

2.1.3 基于应用程序接口(API)序列程序理解模型

基于应用程序接口(API)序列程序理解模型将程序中包含的 API 序列作为研究对象.由于程序所调用的 API 可以在一定程度上反映了程序的功能特性,因此一些研究工作对程序中包含的 API 序列

进行建模,具体的任务包括 API 序列检索、代码检索以及代码注释生成.

在代码检索和 API 序列检索任务中,现有工作都是通过对比自然语言和代码(API 序列)向量的相似度来完成.Gu 等人[21]提出了基于深度学习的 API 检索方法,根据自然语言查询语句来检索与其相匹配的 API 序列.该方法采用 seq2seq 框架构建模型,首先将自然语言编码为一个特征向量,然后基于该向量生成 API 序列.在生成 API 序列时,该方法还考虑了 API 的重要性,在损失函数中增加了惩罚项,对数据集中出现频率较高的 API 进行惩罚,避免模型频繁地预测高频 API.该方法利用 BLEU 来评估生成的 API 序列的质量,在 API 序列检索任务中,较其他方法取得了 40%左右的提升.最近,Gu 等人[19]又提出了基于深度学习的代码检索方法,如 2.1.2 节介绍,该方法对代码中的 token 序列、方法名序列以及 API 序列进行编码,将这 3 个向量进行组合作为代码的表示.

在代码注释生成任务中,Hu 等人[18]对基于 seq2seq 的程序注释模型进行了改进.在程序的表示中,除了代码 token 序列之外,该文还考虑了代码中调用的 API 序列中所包含的知识,将这些知识输入到网络模型中辅助生成注释,使得生成的注释能够更好地描述代码的功能.实验结果表明:该方法的 BLEU 指标相比 Iyer 等人[17]的方法在开源 Java 项目的注释生成任务上提升了近 20 个百分点,取得了当时最好的效果[18].

以上工作表明了 API 序列在程序理解任务中的重要性,对 API 序列中包含的信息进行挖掘可以帮助人们更好地理解程序.基于结构的程序理解模型则考虑了程序语言的语法结构、数据流以及控制流等信息,通过将程序表示为抽象语法树或者程序图的形式来对程序的语法结构、数据流动和控制流等进行建模,在代码补全、代码注释生成、代码模式检测以及程序自动生成等任务中被广泛使用.表 3 是基于结构的程序理解相关研究工作.

Table 3　Research on Structure-Based Program Comprehension

表 3　基于结构的程序理解相关研究工作

Type	Task	Reference	Model
AST	Code Completion	Li et al.[24]	LSTM,Pointer Network
		Liu et al.[25]	LSTM
	Code Generation	Yin et al.[26]	Seq2seq
		Rabinovich et al.[27]	Seq2seq
	Code Summarization	Hu et al.[28]	Seq2seq
		Alon et al.[29]	Bi-LSTM
		Wanet al.[30]	Tree-LSTM,LSTM,RL
	Code Pattern Detection	White et al.[31]	RNN
		Wei et al.[32]	LSTM
		Mou et al.[33]	CNN
	Code Convention Correction	Alon et al.[34]	CRF,Word2vec
Graph	Code Convention Correction	Allamanis et al.[35]	GNN,GRU
	Bug Fixing	Allamaniset al.[36]	GNN,GRU
	Code Summarization	Fernandes et al.[37]	Bi-LSTM,GNN,GRU
	Code Generation	Brockschmidtet al.[38]	GNN,GRU
	Code Pattern Detection	Ben-Nun et al.[39]	Skip-gram

2.2　基于结构的程序理解模型

2.2.1　基于抽象语法树的程序理解模型

抽象语法树(AST)是一种程序代码的抽象表示形式,树中的每一个节点(及其子节点)对应源代码中的某个代码片段.在抽象语法树中,非终结符对应于树的中间节点(如 Assign,If,For,Expr 等),与代码片段的具体类型相对应,与程序结构紧密相关;而终结符则位于树的叶子节点(如字符串、变量名、方法名等),与程序语义密切相关.抽象语法树可以有效地表示程序的语法及其结构,被广泛应用于程序

理解相关任务中[24-27],如代码补全、程序自动生成、代码模式检测等.

在代码补全任务中,基于 AST 的程序理解模型针对代码的抽象语法树进行建模.在这些工作中,程序被解析为 AST,然后对 AST 进行遍历得到节点序列,然后构建基于循环神经网络的语言模型对节点序列进行建模.Li 等人[24]和 Liu 等人[25]构建基于 LSTM 的代码补全模型在 AST 节点序列上完成代码补全任务,根据已有 AST 节点来预测之后最可能出现的 AST 节点.在预测 AST 节点时分为 2 种情况,即预测节点的类型(type)和节点的取值(value).由于节点类型数目是确定的,因此在节点类型预测时不存在前文中提到的 OoV 的问题.而节点的值中包含了大量程序员自定义的标识符,会出现 OoV 问题,因此 Li 等人[24]在预测节点值时引入了指针网络来缓解 OoV.除此之外,该方法中还引入了注意力机制.到目前为止,该方法在基于 AST 的代码补全任务上取得了当时最好的效果[24].尽管 AST 可以提供程序语法和结构信息,但以上这些工作都是针对 AST 节点序列进行建模,建模的对象仍然是序列,树中包含的结构信息仍然会被丢失.此外,以上代码补全研究直接对 AST 的节点进行预测,这与实际的代码补全场景并不完全一致.在实际的代码补全任务中,需要根据程序员已写好的代码来预测下一个 token,而 AST 节点和程序 token 并非一一对应,因此,预测 AST 节点并非严格意义上的代码补全.并且,将代码解析成 AST 之后树中节点的数目远远大于源代码中 token 的数目.目前,对较长的节点序列建模是一件较困难的事情.因此,在代码补全任务中如何恰当地利用 AST 信息还需要进一步思考.

在根据自然语言生成程序的任务中,AST 被用于约束生成的代码使其符合语法规则,先生成 AST 再将 AST 转换为源码就可以保证生成的代码符合语法规则.Yin 等人[26]等人使用 seq2seq 框架完成代码生成任务,其中编码器对自然语言进行编码,然后通过解码器生成程序代码的 AST,然后再将语法树转换为代码.以抽象语法树作为中间结果,一方面缩小了生成代码 token 时的搜索空间,另一方面也能使生成的代码符合程序语言的语法规则.该模型的解码器在生成 AST 节点的过程中,利用编码器得到的特征向量以及已经生成的节点作为上文信息,同时还考虑了当前预测节点的父节点和兄弟节点,利用这些信息来增加结果的可靠性.与该工作类似,Rabinovich 等人[27]同样也基于 AST 完成代码生成任务,不同的是,该方法中还使用了有监督的注意力机制来增强结果的可靠性.以上方法面向的都是较为简单的程序生成任务,生成的程序较短且形式较为单一.若要生成较为复杂的代码,其对应的 AST 会十分庞大.目前,生成庞大的 AST 仍然是一个巨大的挑战.以上这些方法距离生成可以直接投入工业使用的代码还很远.

AST 也被应用于在代码模式检测任务中,如代码克隆检测、代码分类等.利用深度神经网络对 AST 进行建模得到其向量表示作为代码的特征信息,根据该信息完成代码模式检测任务.在克隆检测任务中,White 等人[31]提出了基于深度学习技术的代码克隆检测方法,该方法将程序表示为 token 序列和 AST 节点序列,利用 2 个不同的 RNN 分别对这 2 个序列进行建模,以学习程序中的词法和语法特征.该文将这 2 个特征相结合作为整个程序的特征向量,根据该向量判断 2 个程序是否属于一个克隆对.与该工作类似,Wei 等人[32]提出了基于 AST 的 LSTM 模型来学习程序中的词法和语法结构,该工作的目的是利用神经网络模型检测出第 4 种类型的克隆,即功能相似的代码构成的克隆对.在程序分类任务中,Mou 等人[33]提出了基于树结构的卷积神经网络,通过该网络对程序 AST 进行建模.该方法通过网络中的卷积层对抽象语法树进行卷积计算,学习程序代码的结构信息,并通过动态池化层来处理语法树大小不同的问题,最终到一个能够表示程序代码结构特征的向量表示.论文中将该向量作为提取到的程序特征应用于程序分类和代码模式检测 2 个任务中验证其模型的有效性.实验结果表明该模型在 POJ104 类程序代码分类任务上取得了 94% 的准确率,在冒泡程序检测任务中取得了 89.1% 的准确率.以上模型均使用 AST 作为代码的表示形式,旨在对程序中的语法和结构进行建模,并取得了较好的效果,这说明程序代码的语法和结构在代码模式检测任务中不容忽视.

在代码注释生成任务中,程序结构也受到了关注,近年来基于 AST 的模型也被用于该任务中.Hu 等人[28]基于 AST 实现了代码注释生成,为了保留代码中的结构和语法信息,该工作将程序解析为抽象语法树,然后提出了一种能够保留其结构的遍历 AST 的方法,将 AST 转化为节点的线性序列,然后基于 seq2seq 框架完成注释生成任务.相比于基于代码 token 序列的注释生成方法,该方法取得了近 10 个

百分点的提升.Alon 等人[34]利用抽象语法树路径(AST path)来表示程序,通过 AST 路径来表示程序的结构和语法.该方法首先将程序解析为抽象语法树,对于程序中每个元素(变量名、方法名、表达式类型),提取与其相关的路径,然后采用条件随机场和 word2vec 这 2 种方法获得该变量的表示.该方法在变量名预测、方法名预测以及表达式类型预测任务中都取得了目前最好的结果,较其他方法有较大提升.最近,Alone 等人[29]又对该模型进行了改进,通过双向 LSTM 对 AST 路径进行建模,将该模型用于代码摘要任务中.在生成摘要时还引入了 attention 机制,在生成摘要中的词语时选择与其最相关的 AST 路径.以上模型都是直接对 AST 节点序列或者路径进行建模,构建线性模型对序列化的 AST 进行建模,Wan 等人[30]则构建了 Tree-LSTM 模型对 AST 结构进行建模得到其代码的结构化表示,除此之外该方法还采用 LSTM 对源代码 token 序列进行建模得到代码的序列化表示,然后利用 Tree-LSTM 对代码的 AST 进行建模得到代码的结构化表示,最后将二者结合起来作为代码的最终表示,根据该表示完成代码注释生成任务.在生成注释时引入了强化学习机制缓解 exposure bias(即模型的解码器部分在训练过程中输入的数据来自真实的样本(groundtruth),采用最大似然估计算法来预测下一个单词;而在测试时,每一时刻的输入来自前一时刻的输出.一旦前面的单词生成错误,错误就会传递下去,使得生成结果越来越差)的问题,该方法在训练过程中不再采用最大似然估计算法,而是直接根据生成结果的好坏计算奖励值(reward),根据奖励值来更新模型参数,这样就可以缓解 exposure bias 的问题,提升模型的性能.目前的代码注释生成研究大多是生成程序中某个方法的摘要,相对于完整的程序而言,方法的长度一般比较短,因此其 AST 也不会过于庞大,相较于代码补全任务,在注释生成任务中对 AST 进行建模相对容易一些.

2.2.2　基于图的程序理解模型

图的结构可以对程序中的数据流动关系进行建模,例如控制流图以及数据依赖图.图中节点表示程序中的元素,例如程序中的 token 或变量[35-37]、程序中指针(pointer)的内存地址[40];图中的边表示节点之间的依赖关系或数据流动情况.近年来许多程序分析的研究工作基于图展开,通过将程序表示为图的形式使得模型能够更好地理解程序中变量之间的依赖关系以及程序语义.随着深度学习的发展,基于深度神经网络的图模型被用于程序理解相关任务中,包括程序代码风格修正、代码修复、程序注释生成等.

在代码风格修正任务中,Allamanis 等人[35]基于 GGNN[40],提出了基于图的程序表示方法,旨在学习程序中的数据依赖关系.该工作基于 AST 将程序表示为图的形式,图中每个节点表示程序代码中的 token,图中的边则包含了 AST 中的节点之间的关联以及变量之间的数据依赖关联关系.根据以上表示,该方法构建图神经网络对表示为图形式的代码进行学习,得到代码中各个 token 的特征向量表示.该方法在程序变量命名任务中达到 45%左右的准确率,而在变量误用错误检测任务中能够达到 80%的准确率.通过图来表示程序中变量之间的依赖关系也被应用于代码修复任务中,Allamanis 等人[36]提出了深度神经网络的模型,对代码中的变量之间数据流进行建模,考虑了变量的上下文(context)及其用法(usage).该文提出了一个新的代码修复的场景——智能粘贴(SmartPaste),将一段代码插入到现有的一个程序中,然后修改插入代码片段中的变量使其能够符合它所插入的程序的语义,并在该任务中验证了模型的有效性.该模型首先基于 GRU 得到程序中变量的上下文表示,然后基于变量在上下文中的使用情况来计算变量的用法(usage)表示,根据这 2 个表示来完成智能粘贴任务.

最近,基于图的程序理解模型也被用于程序生成和代码模型检测中.Brockschmidt 等人[38]在程序 AST 上增加相应的边构建程序图,然后构建图神经网络[40]对程序的结构和数据流进行建模完成程序生成任务.Ben-Nun 等人[39]提出了一种基于图和代码中间表示(intermediate representation)的程序向量表示 inst2vec,该表示是语言无关和平台无关的.该文首先通过编译器 LLVM 对代码进行编译得到代码的中间表示(LLVM IR),在该表示中,不包含数据流和控制流信息.为了对代码中的数据流和控制流进行建模,作者将数据流和控制流信息引入已经得到的中间表示中,构建了上下文流图(contextual flow graph, XFG),然后基于该图构建循环神经网络得到向量表示.在训练时该模型采用与 skip-gram 模型相同的方式来预测上下文语句,得到最终的程序向量表示.该向量在程序理解任务中取得了较好的效果,其中,在程序分类实验中准确率超过了当时最好的模型 TBCNN,达到了目前最高的准确率.

在代码注释生成任务中,基于图的程序表示方法也被采用.Fernandes 等人[37]提出了将基于序列和图的模型相结合的程序表示方法.该方法首先通过双向 LSTM 对程序的 token 序列进行编码,得到每个 token 的向量表示.然后构建图神经网络对程序中的数据流和控制流进行建模,将双向 LSTM 模型中得到的每个 token 的向量作为图中节点的初始值.然后将图神经网络得到的向量作为整个程序的表示,根据该特征向量生成代码注释.

在以上模型中,图的构建大多基于 AST 来完成,具体来说,他们通过在 AST 中增加相应的边来表示节点之间的关联以及变量之间的数据依赖关联关系完成图的构建.因此,基于图的程序理解模型可以被理解为是基于 AST 的模型的扩展,由于增加了更多信息,因此基于图的模型具有更强的建模能力.但是,基于图的模型也存在网络复杂、较难训练的缺点,在实际应用中仍然面临着一些挑战.

2.3 基于执行过程的程序理解模型

程序的动态执行过程在程序理解中也是不容忽视的一部分,在程序表示和程序生成任务中占有重要的地位.程序执行轨迹(execution traces)是程序执行过程中的中间结果,例如在执行过程中调用的子程序及其参数[42],或者在执行过程中程序中各个变量值的变化情况[43].例如 Wang 等人[43]对程序执行过程中变量值的变化情况进行建模,图 3 是该方法中执行 max 函数过程中变量值的变化情况示意图.图 4 为 Reed 等人[42]的方法中生成的加法程序在执行过程中所调用的子程序及其参数.通过对程序执行轨迹进行学习与建模,可以对程序的执行过程有更好地理解.关于程序执行轨迹的研究工作近年来主要集中在程序综合任务上[42,44-46].除此之外,文献[43]基于程序执行轨迹构建了程序表示模型得到了程序的特征向量完成程序错误分类任务.表 4 是基于执行过程的程序理解相关研究工作.

```
1   static int max (int[] arr) {
2
3     int max_val = int.MinValue;
4
5     foreach (int item in arr)
6     {
7         if   (item > max_val)
8             max_val = item;
9     }
10
11    return max_val;
12  }
```

(a) Max function

Variable Trace	State Trace
{max_val: −∞}	{max_val: −∞, item: ⊥}
{item:1}	{max_val: −∞, item:1}
{max_val:1}	{max_val:1, item:1}
{item:5}	{max_val:1, item:5}
{max_val:5}	{max_val:5, item:5}
{item:3}	{max_val:5, item:3}

(b) Variable and state traces obtained by executing function max, given **arr** = [1,5,3]

Fig. 3 Variable and state traces obtained by executing function max, given ***arr*** = [1,5,3]

图 3 ***arr*** = [1,5,3]时执行 max 函数过程中变量值的变化情况[43]

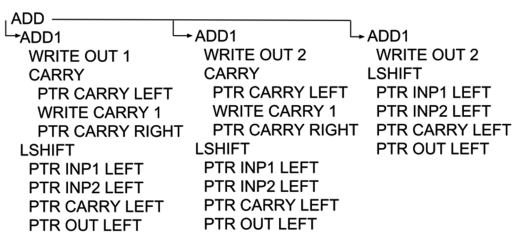

Fig. 4 Sub-programs and parameters invoked by addition program

图 4 执行加法程序过程中调用的子程序及其参数[42]

Table 4 Research Onexecution Process-Basedprogram Comprehension

表 4 基于执行过程的程序理解相关研究工作

Type	Task	Reference	Model
Executing Process	Bug Classification	Wang et al.[43]	GRU
	Program Synthesis	Reed et al.[42]	LSTM
		Cai et al.[44]	LSTM
		Xiao et al.[45]	LSTM
		Chen et al.[46]	LSTM,RL

在程序综合任务中,Reed 等人[42]提出了神经程序解释器(neural programmer-interpreters, NPI)用于表示和执行程序.图 4 所示的是该方法生成的执行加法程序过程中调用的子程序及其参数,其中,子程序包括进位(CARRY)、单位数相加(ADD1)、值的写入(WRITE)等操作.该方法由基于循环神经网络的组合神经网络构成,包含一个任务无关的循环神经网络内核、一个键值对存储单元以及一个领域相关的编码器.该文将程序执行过程中的执行轨迹作为训练数据进行有监督地训练,使得模型能够模拟程序运行过程,并利用基于栈的记忆单元对中间结果进行保存以减轻循环神经网络的记忆负担.

实验结果表明该方法能够生成加法、排序、3D 模型旋转等程序.Cai 等人[44]对 NPI 模型进行了改进,提出了 RNPI.该方法在 NPI 模型的基础上,允许程序调用其自身,使得模型具备生成包含递归的程序的能力.最近,Xiao 等人[45]对 NPI 进行了改进,提出了组合子神经程序解释器(combinatory neural programmer-interpreter, CNPI)来增强 NPI 的泛化性(universality)和易学性(learnability).该工作提出了 4 种组合子(combinators)来表示程序中最常见的 4 种模式——顺序、条件、线性递归以及树递归.其中,每一个组合子是一个"程序模板",可以作为子程序被调用.该方法中程序子的提出减少了网络运行过程中需要调用的子程序数目和复杂度,因此减轻了整个网络的负担.以上程序综合相关研究大多只能生成简单的程序,例如简单运算、排序、字符串操作等,Chen 等人[46]针对该问题,提出了能够生成复杂程序的程序综合方法,该方法能够生成将程序解析成语法树的解析器,即输入程序和其对应的语法树,该方法能够生成一个解析器程序,生成的程序能够将输入的程序解析为其对应的语法树形式,该方法基于 LL(1)文法设计了一个 LL 机(LL 机包含的指令即为程序执行轨迹),然后利用神经网络生成执行 LL 机的指令,将这些指令(执行轨迹)作为训练集.该模型的网络结构为 GRU,并且在模型中引入了强化学习策略,应用于执行轨迹缺乏的场景中.在解析器程序生成上,该方法在其测试集上达到了 100%的准确率.

除了程序综合任务,基于执行过程的程序理解模型也被用于程序错误分类任务中.Wang 等人[43]认为程序执行轨迹能够更准确地表达程序的语义,因此设计了基于程序执行轨迹的程序表示方法.该方法将程序执行过程中各个变量的取值变化情况作为程序执行轨迹.如图 3 所示,图 3(b)列出的即为图 3(a)程序执行过程中变量 max_val 值的变化情况.然后,该文构建了基于 GRU 的模型对程序执行轨迹序列进行建模,得到程序的特征向量表示.在程序错误分类任务上,该方法相较于基于 token 和 AST 的模型取得了巨大的提升.

尽管程序的执行轨迹可以更好地表达程序的动态执行过程及其语义,但程序的执行轨迹数目往往十分巨大,尤其在一些较为复杂的程序中.目前的工作所使用的程序数据都较为简单,若要对复杂程序的执行轨迹进行建模,则是一件十分困难的事情,这也是该方法的局限性.

2.4　模型对比分析

本节对 3 种不同程序理解模型进行对比分析.表 5 列出了不同模型的主要应用、技术及其面临的挑战.如表 5 所示,基于序列的程序理解模型针对序列化的程序进行建模,主要被应用于代码补全、程序注释生成以及代码检索等任务中,采用的技术主要为基于序列的深度学习模型,包括 LSTM,GRU,CNN 等.将程序表示为线性序列导致程序中的结构信息未被充分利用.此外,在序列较长时,模型无法对较长的序列信息进行充分建模.基于结构的模型主要对程序语法树或者程序数据流图、控制流图等进行建模,主要应用于程序生成、代码模式检测、注释生成等任务中,主要采用 GNN,GRU,LSTM 等网络构建模型.与基于序列的模型类似,在对 AST 进行建模时,同样面临着长依赖的问题.而在对图进行建模时存在着网络复杂、较难训练的缺点.基于执行过程的程序理解模型对程序的执行轨迹进行建模,主要被应用于程序综合中,采用 LSTM,GRU 等网络来构建模型.目前该模型处理的都是较为简单的程序,例如四则运算、排序、字符串操作等.生成可以直接投入工业界使用的程序仍是目前研究的一大难点.此外,程序轨迹的数目往往十分庞大,尤其是在一些较为复杂的程序中,直接对其进行建模也是目前面临的难点之一.

Table 5　Comparison of Different Program Comprehension Models

表 5　不同程序理解模型对比

Model	Application	Technique	Challenges
Sequence	Code Completion, Code Summarization, Code Search	LSTM,GRU,CNN	Long-term Dependency, Structure
Structure	Code Generation, Code Pattern Detection, Code Summarization	GNN,GRU,LSTM	Long-term Dependency, Hard to Ttrain
Execution Process	Program Synthesis, Bug Classification	LSTM,GRU,RL	Tasks are Simple, Large Number of Execution Traces

3 相关应用

在程序分析中,程序理解起着至关重要的作用.在本节,将介绍基于深度学习的程序理解在软件工程和程序分析中的相关应用.

3.1 代码补全

代码补全是IDE(例如Eclipse,IntelliJ,Visual Studio等)中使用频率最高的功能[47],它有效地帮助程序员预测代码的类名、方法名、关键字等,提高了程序开发的效率并减少了编码过程中的拼写错误.在该任务中,代码的理解与表示至关重要.传统代码补全方法利用静态类型信息结合各种启发式规则来决定要预测的Token,例如方法名、参数等.目前基于深度学习技术的代码补全相关研究大多通过构建语言模型对已有部分代码(context)进行建模,根据其特征向量来预测接下来最可能出现的token.学术界针对如何表示已有的部分代码提出了各种各样的模型,其中RNN,CNN等被广泛应用于代码补全任务中[13-14,24-25],包括了基于token级别和基于AST级别的模型.

相较于传统基于统计和规则的方法,基于深度学习的代码补全模型通过深度神经网络对代码token序列或者AST进行建模,能够充分挖掘代码中蕴含的特征信息.因此,在代码补全任务中基于深度学习的模型能够取得更高的准确率,且模型的实现较为简单.

3.2 代码注释生成

在软件工程中,软件维护占据了90%的软件开发资源.研究表明:给软件代码添加注释可以帮助开发人员更有效地理解相关代码背后的语义,有利于提升代码的可读性和软件的可维护性[48-50].早期关于代码注释生成的研究大多基于人工定义的规则或模板,这类方法根据规则抽取代码中对应的内容,并填入定义好的模板中,获得代码对应的摘要[51-52].基于模板的方法,通常需要对不同类型的代码进行模板定制,需要消耗大量的人力;此外,当代码中的方法名、变量名等命名不规范时,抽取式的方法很难得到有效的注释内容.近几年来随着深度学习的发展,基于seq2seq框架的模型被用于代码注释生成[17-18,29].这些模型首先将代码token序列或者AST编码为特征向量,然后根据该向量逐个生成注释中的词语.在生成注释时还利用了注意力机制选择代码中与当前生成的词语最相关的内容.

基于深度学习的程序注释生成方法运用深度神经网络对代码中包含的知识进行挖掘,得到其特征向量表示.相较于基于模板的注释生成方法,基于深度学习的方法能够生成更加丰富的注释内容,且不需要人为地针对不同类型代码制定规则,模型具有更强的可扩展性,也极大地节省了人力资源.

3.3 代码风格修正

除了程序语言自身的语法约束之外,代码风格也是约束程序的方式之一,例如代码格式、命名风格等.风格规范的代码有利于代码的理解、查询与调试[53].传统方法基于规则通过对代码进行语法分析来进行代码风格检查,例如Checkstyle① 工具.近年来随着深度学习技术的发展,利用自动化方法来修正代码风格开始被研究人员关注.目前基于深度学习的代码风格的研究主要包括变量名、函数名和类名的预测(重命名)、代码格式修正等.这些工作利用CNN,GGNN等网络得到待修正代码的特征向量表示[16,35],根据这些向量表示进行风格修正.

在大量数据的支持下,利用深度学习技术构建神经网络对待修正代码进行建模可以自动化地进行代码风格修正,省去了传统方法中人为制定语法检查规则的繁琐步骤,简单且有效.

3.4 代码搜索

代码搜索是软件开发过程中必不可少的一项操作,基于自然语言查询语句生成符合其描述的代码片段近几年来受到了学术界的关注.与程序生成任务类似,代码搜索任务同样需要同时具备自然语言和程序的理解能力.早期的代码搜索相关研究基于信息检索(information retrieval, IR)技术[54-55],通过相似度匹配等算法来进行代码搜索.随着深度学习技术的广泛应用,近几年来基于深度学习技术被应用于代码搜索的研究中[19,21].他们利用深度神经网络分别对自然语言和代码进行建模得到其向量表示,在模型训练过程中将自然语言查询语句与其对应的代码的向量表示映射到相近的向量空间中.在测试时,根据自然语言查询语句,在向量空间中搜索与其最相近的代码向量,得到对应的代码.

传统代码搜索方法基于信息检索中的相似度匹配方法来匹配自然语言和代码,只能根据关键词

① http://checkstyle.sourceforge.net/

匹配等对较低层次的特征进行搜索.而运用深度神经网络得到代码和查询语言的向量表示然后对向量之间的空间距离进行优化能够挖掘更高层次的特征,模型可以搜索出更加多样化的结果.

3.5　程序自动生成

根据需求(输入输出样例或者自然语言描述)自动生成程序近几年来引起学术界广泛关注.其中,基于输入输出样例生成程序的任务又被称为程序综合.在程序自动生成任务中,需要同时具备对需求和对程序的理解能力,因此程序自动生成也是程序理解的一个重要应用场景.传统方法基于语法规则匹配等方式完成该任务[56-57].近年来,许多研究者利用深度学习方法对程序自动生成进行了探索.

在根据自然语言描述生成对应代码的任务中,现有方法主要可分为2类:基于抽象语法树以及基于程序图的程序生成方法.其中,基于抽象语法树的程序生成方法[26-27]利用AST约束生成的代码符合语法规则;基于图的程序生成模型[38]则是在程序AST上增加相应的边构建程序图,然后构建图神经网络对程序的结构和数据流进行建模.此外,Murali等人[58]提出了基于代码骨架(sketch)的程序生成方法,可根据API生成代码.该方法根据Java语言的语法定义了程序骨架,在代码生成任务中先生成代码骨架,然后将骨架转换为代码.

在程序综合中,Reed等人[42]提出了基于深度学习的程序综合方法,构建了神经程序解释器(NPI)来表示和执行程序.此后,许多研究工作基于NPI展开,对其进行了改进[44-45].此外,Balog等人[59]提出了基于深度神经网络的程序综合方法,该方法可生成简单的程序,这些程序是由大小为34的领域特定语言(domain specific language, DSL)语句集合中的元素组合而成.给定输入输出样例,该方法通过将深度学习与传统的基于搜索的程序综合方法相结合,该方法能够生成由DSL语句组合而成的程序.较传统方法,该方法实现效率更高.

目前程序生成研究主要面向生成简单的程序,例如四则运算、排序、字符串操作等,目前已取得较高的准确率.相较于传统的程序生成方法,基于深度学习的方法基于已有数据样例构建神经网络来挖掘输入数据和生成的代码中隐含的对应关系,极大地减少了对人的依赖,在简单程序的生成任务中能够取得较高的准确率.但是,在复杂程序的生成方面目前相关研究较少,是一个重要的未来研究方向.

4　问题与挑战

近几年来,深度学习逐渐被应用于包括程序理解在内的各个领域,并且取得了一定的成就.由于程序自身复杂的特性以及程序理解任务的复杂性,基于深度学习的程序理解研究中仍然面临着以下问题与挑战.

4.1　数据集的获取

数据集是利用深度学习完成程序理解任务的关键,规范、高质量的数据集更有利于深度神经网络的学习.在图像识别和自然语言处理中,有许多公开的数据集供研究者展开研究.然而,在程序分析领域,公开的高质量数据集十分稀缺,许多研究者需要自己构造数据集来完成相应的程序理解任务,这导致了训练语料的质量参差不齐,给模型的训练与优化带来了额外的噪声.此外,相同的任务中,也存在各种各样的数据集,这对于不同方法之间的对比和评估带来很大的不便.因此,如何获取统一规范的高质量程序语料库是基于深度学习的程序理解中的一项挑战.

4.2　程序的表示

目前的程序理解研究中,程序的表示大多都只采用某一种表示方式,例如程序token序列级别的表示[13,16]、AST节点序列表示[24-25]、程序中的数据流[35-36]、程序执行轨迹[42-43,46]等.这些信息分别表示了程序的不同特性,例如程序的统计特性、结构性、语义等,人们根据任务需求选用一种表示方式对程序的某种特性进行建模.然而,这些不同的表示之间的差距较大.因此,现有的程序表示方法的泛化性较差,适用的任务范围较小,很难迁移到其他任务中.若能设计更加泛化的程序表示方式,就可以构建更加通用的程序理解模型,这也是程序理解面临的一项挑战.

4.3　自定义标识符的处理

由于程序中存在着大量用户自定义标识符,因此在基于深度学习的程序模型中,构建的词表往往非常大.即使如此,还是无法覆盖程序中全部的token,仍有许多词表之外的词(OoV)出现.这给程序的理解带来了很大的影响.为了解决该问题,Cummins等人[12]构建了基于字符级别的语言模型对程序进行建模,但直接对字符序列进行建模会丢失程序的token以及API所表达的语义;Bhoopchand等人[14]和Li等人[24]构建了基于指针网络的模型以

一定概率选择上文中的 token 来表示后文可能出现的 OoV 词语.但是该方法并不能处理未出现过的 OoV 词.以上方法并没有从本质上解决自定义标识符的问题.因此,如何解决基于深度学习的程序建模中 OoV 的问题也是未来的一个研究重点.

4.4 模型的有效性分析

目前的程序理解任务在数据来源、实验方式以及评价方法等方面存在很大的差异,难以对不同方法的有效性进行比较.一方面,缺乏通用的测试基准(Benchmark),无法对各种程序理解模型进行直接对比;另一方面,当前的评估指标大多参照自然语言处理任务中使用的评价指标(例如 BLEU),而这些评价指标是否能直接用于评价程序理解模型还有待于进一步验证.因此,如何针对程序理解建立具有普适性的测试基准和评价度量体系也是程序理解研究中面临的一项挑战.

5 总 结

程序理解是软件开发与维护中的重要活动,在软件工程中占据着重要的地位.近几年来,深度学习技术的发展和互联网中大量高质量代码的涌现为研究者们提供了利用大规模代码中包含的知识来解决程序理解相关问题的新途径.本文对近年来基于深度学习的程序理解研究工作进行了综述,对这些工作的原理和技术进行梳理与总结,最后对目前程序理解研究中存在的问题与挑战进行讨论.基于深度学习的程序理解是一个新兴的研究方向,具有重要的研究意义和广阔的发展前景.

参 考 文 献

[1] Storey M A. Theories, methods and tools in program comprehension: Past, present and future [C] //Proc of the 13th Int Workshop on Program Comprehension (IWPC'05). Piscataway, NJ: IEEE, 2005: 181-191

[2] Nielson F, Nielson H R, Hankin C. Principles of Program Analysis [M]. Berlin: Springer, 2015

[3] Ball T. The concept of dynamic analysis [C] //Proc of the ACM SIGSOFT Software Engineering Notes. Berlin: Springer, 1999: 216-234

[4] Hochreiter S, Schmidhuber J. Long short-term memory [J]. Neural Computation, 1997, 9(8): 1735-1780

[5] Cho K, Van Merriënboer B, Bahdanau D, et al. On the properties of neural machine translation: Encoder-decoder approaches [J]. arXiv preprint arXiv:1409.1259, 2014

[6] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate [J]. arXiv preprint arXiv:1409.0473, 2014

[7] Vinyals O, Fortunato M, Jaitly N. Pointer networks [EB/OL]. [2016-04-08]. http://papers. nips. cc/paper/5866-pointer-networks

[8] See A, Liu P J, Manning C D. Get to the point: Summarization with pointer-generator networks [J]. arXiv preprint arXiv:1704.04368, 2017

[9] Hindle A, Barr E T, Su Z, et al. On the naturalness of software [C] //Proc of the 2012 34th Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2012: 837-847

[10] Nguyen T T, Nguyen A T, Nguyen H A, et al. A statistical semantic language model for source code [C] //Proc of the 2013 9th Joint Meeting on Foundations of Software Engineering. New York: ACM, 2013: 532-542

[11] Raychev V, Vechev M, Yahav E. Code completion with statistical language models [C] //Proc of the ACM Sigplan Notices. New York: ACM, 2014, 49(6): 419-428

[12] Cummins C, Petoumenos P, Wang Z, et al. Synthesizing benchmarks for predictive modeling [C] //Proc of the 2017 IEEE/ACM Int Symp on Code Generation and Optimization (CGO). Piscataway, NJ: IEEE, 2017: 86-99

[13] White M, Vendome C, Linares-Vásquez M, et al. Toward deep learning software repositories [C] //Proc of the 12th Working Conf on Mining Software Repositories. Piscataway, NJ: IEEE, 2015: 334-345

[14] Bhoopchand A, Rocktäschel T, Barr E, et al. Learning python code suggestion with a sparse pointer network [J]. arXiv preprint arXiv:1611.08307, 2016

[15] Tu Zhaopeng, Su Zhendong, Devanbu P. On the localness of software [C] //Proc of the 22nd ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2014: 269-280

[16] Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code [EB/OL]. [2019-04-03]. http://jmlr.org/proceedings/papers/v48/allamanis16.html

[17] Iyer S, Konstas I, Cheung A, et al. Summarizing source code using a neural attention model [C] //Proc of the 54th Annual Meeting of the Association for Computational Linguistics Stroudsburg, PA: ACL, 2016, 1: 2073-2083

[18] Hu Xing, Li Ge, Xia Xin, et al. Summarizing Source Code with Transferred API Knowledge [EB/OL]. [2018-10-31]. https://doi.org/10.24963/ijcai.2018/314

[19] Gu Xiaodong, Zhang Hongyu, Kim S. Deep code search [C] //Proc of 2018 IEEE/ACM 40th Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2018: 933-944

[20] Gupta R, Kanade A, Shevade S. Deep reinforcement learning for programming language correction [J]. arXiv preprint arXiv:1801.10467, 2018

[21] Gu Xiaodong, Zhang Hongyu, Zhang Dongmei, et al. Deep API learning [C] //Proc of the 2016 24th ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2016: 631-642

[22] Mnih V, Badia A P, Mirza M, et al. Asynchronous methods for deep reinforcement learning [EB/OL]. [2019-04-03]. http://jmlr.org/proceedings/papers/v48/mniha16.html

[23] Gupta R, Pal S, Kanade A, et al. Deepfix: Fixing common c language errors by deep learning [C] //Proc of the 31st AAAI Conf on Artificial Intelligence. Menlo Park, CA: AAAI, 2017: 1345-1351

[24] Li Jian, Wang Yue, Lyu M R, et al. Code completion with neural attention and pointer networks [J]. arXiv preprint arXiv:1711.09573, 2017

[25] Liu Chang, Wang Xin, Shin R, et al. Neural Code Completion [EB/OL]. [2019-06-10]. https://openreview.net/pdf?id=rJbPBt9lg

[26] Yin Pengcheng, Neubig G. A syntactic neural model for general-purpose code generation [J]. arXiv preprint arXiv: 1704.01696, 2017

[27] Rabinovich M, Stern M, Klein D. Abstract syntax networks for code generation and semantic parsing [J]. arXiv preprint arXiv:1704.07535, 2017

[28] Hu Xing, Li Ge, Xia Xin, et al. Deep code comment generation [C] //Proc of the 26th Conf on Program Comprehension. New York: ACM, 2018: 200-210

[29] Alon U, Levy O, Yahav E. code2seq: Generating sequences from structured representations of code [J]. arXiv preprint arXiv:1808.01400, 2018

[30] Wan Yao, Zhao Zhou, Yang Min, et al. Improving automatic source code summarization via deep reinforcement learning [C] //Proc of the 33rd ACM/IEEE Int Conf on Automated Software Engineering. New York: ACM, 2018: 397-407

[31] White M, Tufano M, Vendome C, et al. Deep learning code fragments for code clone detection [C] //Proc of the 31st IEEE/ACM Int Conf on Automated Software Engineering. New York: ACM, 2016: 87-98

[32] Wei Huihui, Li Ming. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code [EB/OL]. [2018-06-27]. https://doi.org/10.24963/ijcai.2017/423

[33] Mou Lili, Li Ge, Zhang Lu, et al. Convolutional neural networks over tree structures for programming language processing [C] //Proc of the 30th AAAI Conf on Artificial Intelligence. Menlo Park, CA: AAAI, 2016: 1287-1293

[34] Alon U, Zilberstein M, Levy O, et al. A general path-based representation for predicting program properties [C] //Proc of the ACM SIGPLAN Notices. New York: ACM, 2018: 404-419

[35] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs [J]. arXiv preprint arXiv: 1711.00740, 2017

[36] Allamanis M, Brockschmidt M. SmartPaste: Learning to adapt source code [J]. arXiv preprint arXiv: 1705.07867, 2017

[37] Fernandes P, Allamanis M, Brockschmidt M. Structured neural summarization [J]. CoRR abs/1811.01824, 2018

[38] Brockschmidt M, Allamanis M, Gaunt A L, et al. Generative code modeling with graphs [J]. arXiv preprint arXiv:1805.08490, 2018

[39] Ben-Nun T, Jakobovits A S, Hoefler T. Neural code comprehension: A learnable representation of code semantics [EB/OL]. [2018-12-16]. http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics

[40] Li Yujia, Tarlow D, Brockschmidt M, et al. Gated graph sequence neural networks [J]. arXiv preprint arXiv: 1511.05493, 2015

[41] Scarselli F, Gori M, Tsoi A C, et al. The graph neural network model [J]. IEEE Transactions on Neural Networks, 2009, 20(1): 61-80

[42] Reed S, De Freitas N. Neural programmer-interpreters [J]. arXiv preprint arXiv:1511.06279, 2015

[43] Wang Ke, Singh R, Su Zhendong. Dynamic neural program embedding for program repair [J]. arXiv preprint arXiv: 1711.07163, 2017

[44] Cai J, Shin R, Song D. Making neural programming architectures generalize via recursion [J]. arXiv preprint arXiv:1704.06611, 2017

[45] Xiao Da, Liao Joyu, Yuan Xingyuan. Improving the universality and learnability of neural programmer-interpreters with combinator abstraction [J]. arXiv preprint arXiv:1802.02696, 2018

[46] Chen Xinyun, Liu Chang, Song D. Towards synthesizing complex programs from input-output examples [J]. arXiv preprint arXiv:1706.01284, 2017

[47] Amann S, Proksch S, Nadi S, et al. A study of visual studio usage in practice [C] //Proc of the 2016 IEEE 23rd Int Conf on Software Analysis, Evolution, and Reengineering (SANER). Piscataway, NJ: IEEE, 2016: 124-134

[48] Takang A A, Grubb P A, Macredie R D. The effects of comments and identifier names on program comprehensibility: An experimental investigation [J]. Journal of Programming Languages, 1996, 4(3): 143-167

[49] Tenny T. Program readability: Procedures versus comments [J]. IEEE Transactions on Software Engineering, 1988, 14(9): 1271-1279

[50] Woodfield S N, Dunsmore H E, Shen V Y. The effect of modularization and comments on program comprehension [C] //Proc of the 5th Int Conf on Software engineering. Piscataway, NJ: IEEE, 1981: 215-223

[51] Moreno L, Aponte J, Sridhara G, et al. Automatic generation of natural language summaries for Java classes [C] //Proc of the 21st Int Conf on Program Comprehension (ICPC). Piscataway, NJ: IEEE, 2013: 23-32

[52] McBurney P W, McMillan C. Automatic source code summarization of context for Java methods [J]. IEEE Transactions on Software Engineering, 2016, 42(2): 103-119

[53] Allamanis M, Barr E T, Bird C, et al. Learning natural coding conventions [C] //Proc of the 22nd ACM SIGSOFT Int Symp on Foundations of Software Engineering. New York: ACM, 2014: 281-293

[54] Linstead E, Bajracharya S, Ngo T, et al. Sourcerer: Mining and searching Internet-scale software repositories [J]. Data Mining and Knowledge Discovery, 2009, 18(2): 300-336

[55] Lv Fei, Zhang Hongyu, Lou Jianguang, et al. Codehow: Effective code search based on api understanding and extended boolean model (e) [C] //Proc of the 30th IEEE/ACM Int Conf on Automated Software Engineering (ASE). Piscataway, NJ: IEEE, 2015: 260-270

[56] Raghothaman M, Wei Y, Hamadi Y. Swim: Synthesizing what i mean-code search and idiomatic snippet synthesis [C] //Proc of the 2016 IEEE/ACM 38th Int Conf on Software Engineering (ICSE). Piscataway, NJ: IEEE, 2016: 357-367

[57] Lei T, Long F, Barzilay R, et al. From natural language specifications to program input parsers [EB/OL]. [2013-11-26]. http://aclweb.org/anthology/P/P13/P13-1127.pdf

[58] Murali V, Qi L, Chaudhuri S, et al. Neural sketch learning for conditional program generation [J]. arXiv preprint arXiv: 1703.05698, 2017

[59] Balog M, Gaunt A L, Brockschmidt M, et al. Deepcoder: Learning to write programs [J]. arXiv preprint arXiv: 1611.01989, 2016

Liu Fang, born in 1994. PhD candidate. Her main research interests include deep learning, program analysis.

Li Ge, born in 1977. PhD, associate professor. His main research interests include deep learning, program analysis, and natural language processing.

Hu Xing, born in 1993. PhD candidate. Her main research interests include deep learning, program analysis, and code summarization.

Jin Zhi, born in 1962. PhD, professor, PhD supervisor. Her main research interests include requirement engineering, knowledge engineering, and knowledge-based software engineering.