软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, 2022,33(11):3930−3943 [doi: 10.13328/j.cnki.jos.006324]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

语法和语义结合的代码补全方法*

付善庆, 李 征, 赵瑞莲, 郭俊霞

(北京化工大学 信息科学与技术学院, 北京 100029)

通信作者: 郭俊霞, E-mail: gjxia@mail.buct.edu.cn

摘 要: 在软件工程领域, 代码补全是集成开发环境(integrated development environment, IDE)中最有用的技术之一, 提高了软件开发效率, 成为了加速现代软件开发的重要技术. 通过代码补全技术进行类名、方法名、关键字等预测, 在一定程度上提高了代码规范, 降低了编程人员的工作强度. 近年来, 人工智能技术的发展促进了代码补全技术的发展. 总体来说, 智能代码补全技术利用源代码训练深度学习网络, 从语料库学习代码特征, 根据待补全位置的上下文代码特征进行推荐和预测. 现有的代码特征表征方式大多基于程序语法, 没有反映出程序的语义信息. 同时, 目前使用到的网络结构在面对长代码序列时, 解决长距离依赖问题的能力依旧不足. 因此, 提出了基于程序控制依赖关系和语法信息结合共同表征代码的方法, 并将代码补全问题作为一个基于时间卷积网络(time convolution network, TCN)的抽象语法树(abstract grammar tree, AST)节点预测问题, 使得网络模型可以更好地学习程序的语法和语义信息, 并且可以捕获更长范围的依赖关系. 实验结果表明, 该方法比现有方法的准确率提高了约 2.8%.

关键词: 代码补全; 程序语法特征; 程序语义特征; 特征结合; 长距离依赖; 深度学习

中图法分类号: TP311

中文引用格式: 付善庆, 李征, 赵瑞莲, 郭俊霞. 语法和语义结合的代码补全方法. 软件学报, 2022, 33(11): 3930–3943. http://www.jos.org.cn/1000-9825/6324.htm

英文引用格式: Fu SQ, Li Z, Zhao RL, Guo JX. Code Completion Approach Based on Combination of Syntax and Semantics. Ruan Jian Xue Bao/Journal of Software, 2022, 33(11): 3930−3943 (in Chinese). http://www.jos.org.cn/1000-9825/6324.htm

Code Completion Approach Based on Combination of Syntax and Semantics

FU Shan-Qing, LI Zheng, ZHAO Rui-Lian, GUO Jun-Xia

(School of Information Science and Technology, Beijing University of Chemical Technology, Beijing 100029, China)

Abstract: In the field of software engineering, code completion is one of the most useful technologies in the integrated development environment (IDE). It improves the efficiency of software development and becomes an important technology to accelerate the development of modern software. Prediction of class names, method names, keywords, and so on, through code completion technology, to a certain extent, improves code specifications and reduces the work intensity of programmers. In recent years, the development of artificial intelligence promotes the development of code completion. In general, smart code completion uses the source code training network to learn code characteristics from the corpus, and makes recommendations and predictions based on the context code characteristics of the locations to be completed. Most of the existing code feature representations are based on program grammar and do not reflect the semantic information of the program. The network structure currently used is still not capable of solving long-distance dependency problems when facing long code sequences. Therefore, this study proposes a method to characterize codes based on program control dependency and grammar information, and considers code completion as an abstract grammar tree (AST) node prediction problem based on time convolution network (TCN). This network models can learn the grammar and semantic information of the program better, and can capture longer-range of dependencies. This method has been proven to be about 2.8% more accurate than existing methods.

Key words: code completion; program grammar feature; program semantic feature; feature combination; long distance dependency; deep learning

* 基金项目: 国家自然科学基金(61702029, 61672085, 61872026)

收稿时间: 2020-11-08; 修改时间: 2020-12-15; 采用时间: 2021-01-29

代码补全(code completion)是程序综合(program synthesis)领域的重要分支之一. 在已有代码的基础上, 代码补全技术通过对程序进行静态分析, 实现自动为软件开发人员推荐合适的代码, 从而提高开发人员的工作效率. 代码补全作为程序自动生成方法之一, 受到了学术界和工业界的广泛关注. 与基于自然语言进行代码生成不同, 代码补全方式与目前常用的 IDE (integrated development environment)类似, 通过挖掘先前代码之间的深层含义和特征, 对程序的下一个标识符(token)进行预测.

传统的代码补全技术[1], 一类是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, 另外一类则是通过代码样例或者前文语义信息进行匹配补全. 例如, 目前的 IDE 主要利用各种启发式规则和静态类型信息向用户进行推荐, 但该方法通常不考虑代码上下文之间的结构和语义信息. Bruch 等人[2]通过使用方法调用的频率和在类似情况下的代码历史样例, 查找最有可能的方法调用进行推荐. 该方法在其补全的相关性方面优于其他代码补全系统, 但该方法具有较大的局限性, 因为其中大量工作是通过手工完成, 并且完全依赖开发人员具备的先验知识, 导致当程序量增大、先验知识不足时, 方法就会失效, 通用性较差.

深度学习方法由数据驱动, 通过构建深度神经网络, 学习和挖掘数据中隐含的特征. 近年来, 深度学习网络在众多领域得到了广泛关注. 在基于深度学习的代码补全领域, 有研究人员通过使用深度学习技术, 根据已有的大量源程序数据, 深层次学习程序中蕴含的特征[3], 替代传统方法中通过人工进行特征提取的方式, 使得深度学习模型获得更能表现程序深层次的特征, 推进了代码补全技术的发展. 其中, 数据集是利用深度学习进行代码补全的关键. 目前, 伴随着开源网站和开源社区的发展, 越来越多的高质量代码提供给了开发人员, 也为深度学习奠定了数据基础. 考虑到程序语言与自然语言具有一定相似性, 目前, 代码补全常用的深度学习网络有 RNN, LSTM 等语言模型. 但程序语言具有其独特的性质, 如程序语言特有的结构性、上下文之间的长依赖性和自定义标识符预测等. 模型能否合理地解决这些问题, 将决定其代码补全效果的好坏.

目前, 多数研究人员基于程序 AST (abstract syntax tree)进行建模, 但语言模型大多是序列化输入, 因此在数据预处理阶段需要对 AST 进行遍历, 从而得到满足语言模型输入的 AST 节点序列. 这样可能出现对两个不同的程序, 虽然程序的 AST 不同, 但经过遍历序列化之后, AST 中包含的结构信息丢失而导致输入序列相同的情况. 程序的结构信息对于程序语义的理解十分关键, 因为在程序语言中, 这样的结构无处不在, 并且相似代码由于拥有不同的结构而拥有不同的语义. 如图 1 所示, 两例 python 程序图 1(a)、图 1(b)有相同的代码语句, 但两例程序由于代码结构不相同而具有不同的语义; 但是将两段程序解析为 AST 之后, 通过遍历得到的对应 AST 序列相同. 通过图 1 可以看出, 在 AST 序列中, 每一个 AST 节点包含两个属性, 分别为节点的 *type* 和节点的 *value*. 其中, *type* 值代表当前节点所属类型; *value* 代表当前节点具体值, 如变量名、类名、数字文本、运算符等. 两个属性之间以“:”分隔. AST 非叶节点只包含 *type* 属性, 叶节点包含 *type* 和 *value* 属性. 由于节点 *value* 值只存在于 AST 的叶节点中, 因此对于 AST 的非叶节点的 *value*, 其值设置为 *Null*. 图 1 中两例不同的程序被解析为相同的 AST 序列, 因此没有体现出程序不同的结构所带来的程序语义变化.

另一方面, 标准的深度语言模型(RNN, LSTM 等)在解决代码补全问题时, 仍然面临着长距离依赖问题. 当面对长序列数据时, RNN 或 LSTM 网络会将当前序列信息压缩为一个固定大小的向量, 因而丢失大量有用信息. 有研究人员尝试将注意力机制结合标准 LSTM 应用在代码补全问题中, 并取得了比标准 LSTM 网络更好的效果. 但存储过去的信息需要很大的内存容量, 导致长序列的泛化能力很差, 因此处理长距离依赖的能力依旧不足.

针对上述问题, 基于代码补全技术现有的工作, 本文提出一种将程序语法和语义结合的深度代码补全方法, 利用程序的语法信息和程序语句间控制依赖关系, 使深度神经网络能够学习到程序的语法和语义特征; 同时, 为了解决长距离依赖问题, 本文将 TCN (temporal convolutional network)应用于代码补全问题中, 并结合注意力机制, 使本文模型达到较好的预测效果.

本文工作的主要贡献如下:

1) 提出利用程序控制依赖关系表示程序语义特征, 将其与语法特征结合表征程序特征, 构建更完善的代码补全模型, 并在目前主流使用的 LSTM 网络基础上进行实验, 表明加入语义特征可以有效地帮

助代码补全模型对程序的理解，提高预测准确率.

2) 为了充分利用程序上下文信息，在语义与语法结合的基础上，采用 TCN 网络用于捕获程序间长距离依赖关系；同时引入注意力机制，提出了 TCN-LSTM 代码补全模型.

3) 在 Python 真实数据集上进行了模型评估. 实验结果表明：与现有模型相比，TCN-LSTM 在 *type* 值预测和 *value* 值预测方面分别提高了 2.7%和 2.8%.

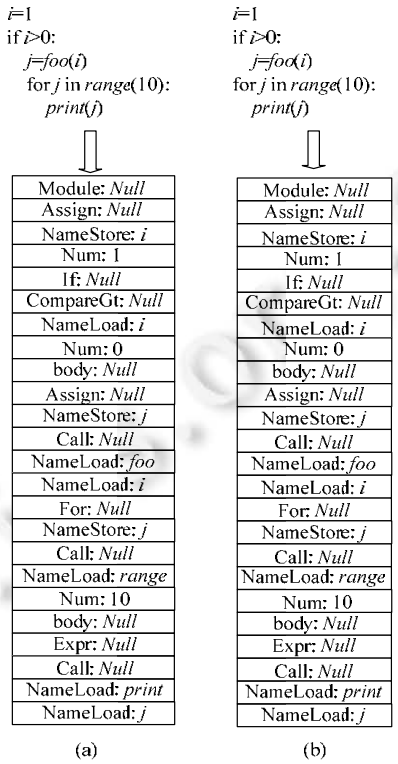

图 1 Python 程序及其对应 AST 序列

本文第 1 节对代码补全相关研究进行介绍. 第 2 节对本文问题进行定义. 第 3 节对本文方法进行介绍. 第 4 节介绍实验以及相关分析. 第 5 节进行总结和展望.

1 相关研究

代码生成和代码补全作为程序自动生成技术的两种实现方式，被认为是提高软件开发自动化程度和最终质量的重要方法[1]. 代码生成技术要求通过用户给定的需求来自动构建程序片段，其需求通常是代码输入输出样例、功能描述等，在一定程度上可以让机器代替开发人员编写代码的工作. 代码补全技术则基于开发人员已有的代码，即时预测待写代码中的类名、方法名和代码片段等.

目前，基于深度学习的代码生成技术分为基于输入输出的代码生成和基于功能描述的代码生成. 基于输入输出的代码生成一般通过学习给定的输入-输出样例来产生程序，如 Reed 等人[4]提出 NPI (neural programmer-interpreters)框架，将程序视为一组词向量，利用神经网络学习输入-输出对和程序执行轨迹来生成程序执行所需要的语句及参数. 基于功能描述的代码生成则是完成从自然语言描述到代码的自动转换，如 Dong 等人[5]将输入的自然语言编码成向量表示，利用基于注意力机制的编码-解码器模型，可以从词汇表中选择单词或者是直接从输入中复制单词，从而实现代码生成. Sun 等人[6]提出一种基于树的神经架构 TreeGen 来产生代码，TreeGen 利用 Transformers 的注意机制来缓解长依赖问题，并引入了一种新的 AST 读取器(编码器)，

将语法规则和 AST 结构整合到网络中.

代码补全技术目前在实际工程开发中得到了广泛应用, IDE 是软件开发人员必不可少的工具, 当程序员对代码库还不熟悉时, IDE 为开发人员提供了提高软件开发效率的代码补全功能. 开发人员在使用静态程序语言(Java, C 等)时, 使用变量之前必须声明其数据类型, 因此, 代码补全技术通过编译时的类型信息来预测程序下一个 Token[7,8]实现静态语言的补全. 但对于动态语言(Python 等), 由于代码中缺少类型的声明, 代码补全仍然存在挑战.

近年来, 为了在动态语言上能够实现有效的代码补全, 越来越多的研究人员将注意力集中在通过使用自然语言处理方法来解决代码语言处理问题[9,10]. Hindle 等人[11]提出程序语言是由工作中的人类创造的, 并伴随着所附带的约束和限制, 因此就像自然语言一样, 程序语言也是重复和可预测的. *N*-gram 模型在一定程度上可以捕捉程序语言的局部信息, 因此, Hindle 等人将其应用于代码语言的建模. 尽管 *N*-gram 捕获了当前词与相邻近词的关系, 有效地学习代码片段上下文信息, 但是 *N*-gram 模型无法处理长距离的上下文信息, 这对于程序语言的理解有了很大的限制, 并且 *N*-gram 模型中每一个输入片段的长度是固定的, 大大限制了模型的通用性. 为解决该问题, 研究人员开始探索其他深度网络模型.

研究证明: RNN 同样作为语言模型, 可以比 N-gram 模型更好地捕获程序之间的规律性. Pavol 等人[12]通过对程序分析得到程序中的 API 调用序列, 利用 RNN 模型对序列建模, 实现了对 API 方法的推荐. Mario 等人[13]首次将 RNN 模型应用于代码补全问题中, 并在 Java 数据集进行实验, 验证了方法的有效性. Dam 等人[14]对标准的 RNN 与 LSTM 进行了比较, 发现 LSTM 在代码补全上的效果更好, 因为 LSTM 能够比 RNN 更好地学习源代码中的长距离依赖关系. 以上研究在词的级别上进行建模, 建立的模型较为有效地捕获了程序上下文信息. 但是由于程序中存在大量的自定义变量, 在程序预处理阶段形成了巨大的词表, 导致模型学习难度增大, 因此, Bieli 等人[15]提出了一种适用于字符级语言建模的统计模型, 并取得了比 *N*-gram 更好的效果. 但是程序语言有其特有的结构性, 以上通过语言模型学习程序的序列化表示方法不能充分理解到程序的结构性. Liu 等人[16]提出将程序解析为 AST, 利用 AST 中蕴含着的丰富结构信息构建基于 AST 的 RNN 模型, 根据已有 AST 节点来预测下一个 AST 节点. Shuai 等人[17]通过对 AST 采用不同的遍历方法得到不同的 AST 序列, 并以此进行建模. 实验证明, 该方法对不同编程语言的代码具有较好的泛化能力. 尽管如此, 程序语言与自然语言有不一样的语言特性, 如 AST 节点的值中同样包含了大量程序员自定义的标识符, 如函数名、类名、变量名等, 并且不同用户的程序自定义标识符命名习惯又不相同, 因此, 构建的词汇表不能包含程序中所有出现的词, 总会有许多词出现在词表外(OoV). 为解决该问题, Bhoopchand 等人[18]和 Li 等人[19]在预测节点值时引入了指针网络[20], 通过指针网络进行上下文节点值的复制, 并且结合注意力机制[21,22]实现了对 OoV 的预测. 有研究希望通过树形神经网络来处理 AST, 如 Zhang 等人[23]提出了 ASTNN 网络, 基于语句粒度将 AST 划分为若干子树, 并且 ASTNN 将每个子树遍历为一维序列, 由 GRU 网络对子树进行编码、组合来捕获语句级语法知识, 若基于 token 级别使用 ASTNN 进行 AST 节点的编码, ASTNN 将成为基于 token 的 RNN 模型, 获取到的语句语法知识将更少, 无法体现 AST 中各个 token 的语义特征. Wei 等人[24]使用 Tree-LSTM 来处理 AST 来获取程序的词汇和句法信息, 由于不同 AST 的不同节点的子节点的数量不同, 会导致模型中参数共享的问题, 因此, 该工作将 AST 转换为二叉树, 采用 Tree-LSTM 网络自底向上对 AST 节点进行编码. 转换后的 AST 破坏了程序原本的语法结构, 削弱了神经模型捕捉更真实的语义的能力[23]. 同时, 基于树的深度学习模型计算成本较高, 与基于序列的深度学习模型相比, 树形神经网络需要更复杂的数据结构, 且在对 AST 进行建模时, 依旧面临长距离依赖问题.

近年来, 卷积网络在音频合成[25]、语言建模[26]、机器翻译[27]等方面都取得了较好的成果, 在序列化数据的建模任务中, 卷积网络也开始逐渐被采用. 其中, TCN 因具有较好的捕捉长序列历史信息的能力, 可以更好地解决长距离依赖问题. Bai 等人[28]在多个序列建模任务中验证了 TCN 明显优于 LSTM 等典型的循环网络架构, 更适合需要长历史信息的问题领域.

为了让深度神经网络更好地理解程序的语义信息, 本文提出将程序的控制依赖关系加入神经网络模型当

中，作为语义特征与基于当前上下文的语法特征进行结合. 同时，为了捕获序列数据之间的长距离依赖关系，本文采用 TCN 网络，并搭建 TCN-LSTM 混合网络模型. 通过实验比较证明，本文提出的方法取得了更好的预测效果.

2 研究问题定义

任何编程语言都有明确的上下文无关语法，并且程序语句和 AST 之间可实现互相转换，因此，AST 是大多数代码补全问题中常用的输入之一. 本文输入包含两部分，分别为程序的语法序列输入和语义序列输入，且均以 AST 序列的形式表示. AST 是一个有根的树，在 AST 中，每个非叶节点对应于上下文无关文法中指定结构信息的非终结符，每个叶节点对应上下文无关文法信息中的一个终结符，图 2 展示了图 1 程序(b)对应的 AST.

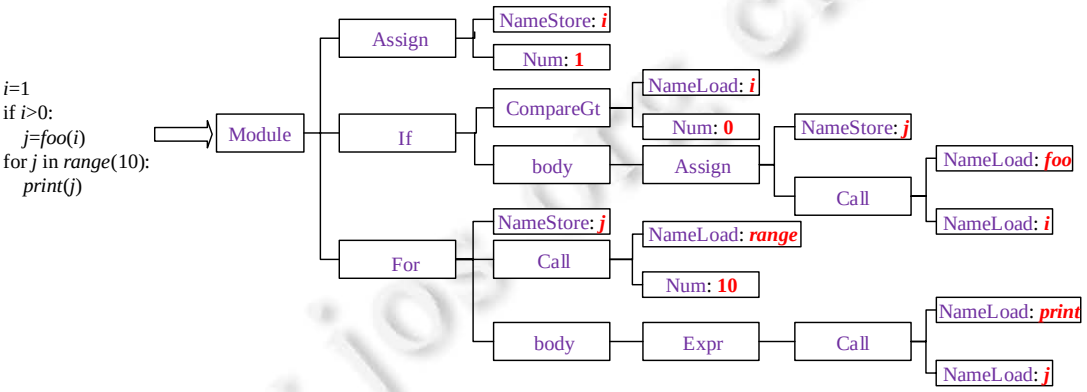

图 2 Python 程序及其对应语法 AST

程序依赖表示程序中各语句间的依赖关系，其中，控制依赖描述了程序的控制结构. 为了便于描述，控制依赖通常以控制流图的形式表示[29]，图 3 展示了图 1 中示例程序的控制流图，其中, ENTER 为程序入口，控制流图中每个节点代表程序中的一条语句，边表示程序中所有可能的执行顺序. 由图 3 看出，控制流图在一定程度上反映了各程序语句之间的依赖关系和结构关系. 如在图 3 程序(a)中，语句“*print*(*j*)”的控制依赖语句为(“if *i*>0”, “for *j* in *range*(10)”).

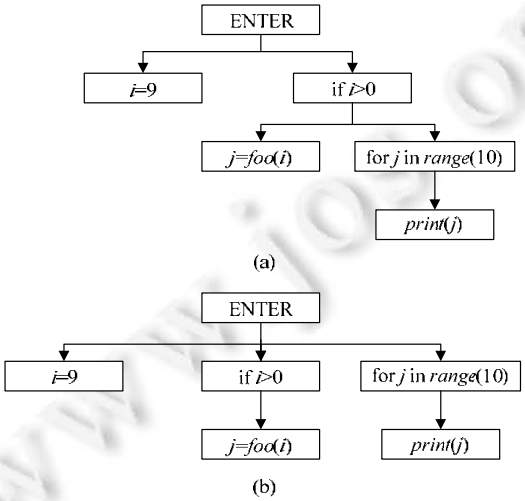

图 3 程序控制流图

定义 1(语法输入). 对源程序 P 所有语句进行语法解析, 生成的 AST 序列称为程序 P 的语法输入, 记为 $L(P)=(x_1,x_2,x_3,\ldots,x_n)$. 其中, x_i 表示一个 AST 节点, 即 $x_i=(T_i:V_i)$, T_i 表示节点 x_i 的 *type* 值向量, V_i 表示节点 x_i 的 *value* 值向量.

定义 2(控制依赖表示). 对源程序 P 所有语句进行静态分析, 生成的语句控制依赖序列称为程序 P 的控制依赖表示, 记为 $C(P)=(s_1,s_2,\ldots,s_n)$. 其中, s_i 表示节点 x_i 所在语句的控制依赖语句.

定义 3(语义输入). 对于源程序 P 对应的语法输入 $L(P)$和控制依赖表示 $C(P)$, 将对语法输入的控制依赖表示解析得到的 AST 序列称为程序 P 的语义输入, 记为 $D(P)=(d_1,d_2,d_3,\ldots,d_n)$. 其中, d_i 表示控制依赖语句 s_i 对应的 AST 序列.

定义 4(输出). 对于给定语法输入 $L(P)=(x_1,x_2,x_3,\ldots,x_n)$, 输出定义为 $O(P)=x_{n+1}$. 其中, x_{n+1} 同样表示一个 AST 节点, 即 $x_{n+1}=(T_{n+1}:V_{n+1})$.

以图 1(b)程序为例, 图 2 为该程序对应的 AST, 对 AST 遍历得到该程序语法输入序列 L 为((Module:*Null*),(Assign:*Null*),(Nmaestore:*i*),…,(NameLoad:*print*),(NameLoad:*j*)). 对于其中一个输入节点, 例如 x_n=(NameLoad:*print*), 该节点所在语句为"*print*(*j*)", 由图 3(b)得到该语句的控制依赖语句为 s_n=("for *j* in *range*(10)"), 由此可获得当前节点(Nameload:*print*)的语义输入序列为 d_n=((For:*Null*),(NameStore:*j*),(Call:*Null*),(NameLoad:*range*),(Num:10)), 预测输出节点为(NameLoad:*j*).

3 基于语法和语义结合的模型

本文模型设计基于以下两个方面考虑: 首先, 为了验证将程序语义信息和语法信息进行结合对代码补全模型的影响, 基于目前代码补全领域的研究, 本文在传统方法的基础上加入语义建模, 构建了 2-LSTM 模型, 以对本文加入语义建模后的有效性进行验证; 其次, 为了更好地捕获程序间长距离依赖关系, 在语义建模的基础上, 本文提出了 TCN-LSTM 模型. 本节首先概述提出的模型结构, 然后详细介绍模型的各个组成部分.

3.1 2-LSTM网络

LSTM 网络在代码补全问题中取得了较好的效果, 成为目前广泛使用的语言模型. 基于此, 为了对语义信息进行利用, 本文提出了 2-LSTM 模型. 与现有研究不同的是, 2-LSTM 模型利用两个 LSTM 网络分别对程序序列的语法信息和语义信息进行编码, 最后将语法向量和语义向量进行结合作为模型的输出.

图 4 展示了本文的 2-LSTM 模型, 该模型对于语法输入序列 L 和语义输入序列 D 分别使用 LSTM 网络作为基本编码器, 用于捕获程序之间的语法信息和语义信息; 并且本文在对语法输入序列 L 进行编码时, 结合了注意力机制, 最后将两个编码器的输出和注意力向量进行结合, 对下一个节点的 *type* 和 *value* 值进行预测.

- 语法编码器

LSTM 模型最早是由 Hochreite 等人[30]为解决 RNN 梯度消失和梯度爆炸现象而提出的. 本模型以 LSTM 网络为基础架构, 标准的 LSTM 网络层单元隐藏状态更新如公式(1)所示, LSTM 单元将 n–1 时刻 LSTM 单元隐藏状态的输出 h_{n-1} 和当前 n 时刻的输入向量 x_n 作为输入, 得到 n 时刻 LSTM 单元隐藏状态的输出 h_n:

$$h_n=LSTM(x_n,h_{n-1}) \tag{1}$$

为获取输入序列的语法信息, 本文使用 LSTM 网络对语法输入序列 L 进行编码, 对于 L 中每一个时刻的输入 x_n, 在进行编码后, 得到其对应的隐藏状态 $h_n\in t$ 作为当前输入节点 x_n 的语法向量, t 为隐藏层状态的大小.

- 语义编码器

为了对程序的语义信息进行建模, 本文同样使用 LSTM 网络对处理得到的语义序列 D 进行编码, 如公式(2)所示, 在 n 时刻, 语义编码器负责对当前输入 x_n 的语义输入序列 d_n 中的 m 个节点进行编码, 并将 LSTM 单元隐藏状态最后的输出 $h_m\in t$ 作为当前输入节点 x_n 的语义向量:

$$h_m=LSTM(d_m,h_{m-1}) \tag{2}$$

- 注意力层

对于语法输入序列 L, 本文在 LSTM 输出层加入注意力机制, 通过之前隐藏状态的输出向量, 计算词与词

之间的相关程度，提高模型的记忆能力．本文注意力层的计算如公式(3)–(6)所示，在第 n 步时，设置 $M_n \in K \times t$ 用于存储之前输出的 K 个隐藏状态 $h_i \in t$，其中，$i \in [n-K, n-1]$．通过计算 h_n 和 M_n 的关系，得到注意分数 $\alpha_n \in 1 \times K$ 和上下文向量 $c_n \in t$．其中，参数 W^M, $W^h \in t \times t$, $v \in t$ 为可训练参数:

$$M_n=[h_{n-K},h_{n-K+1},\ldots,h_{n-1}] \tag{3}$$

$$G_n = \tanh(W^M M_n + 1_K^T (W^h h_n)) \tag{4}$$

$$\alpha_n=softmax(v^T G_n) \tag{5}$$

$$c_n = M_n \alpha_n^T \tag{6}$$

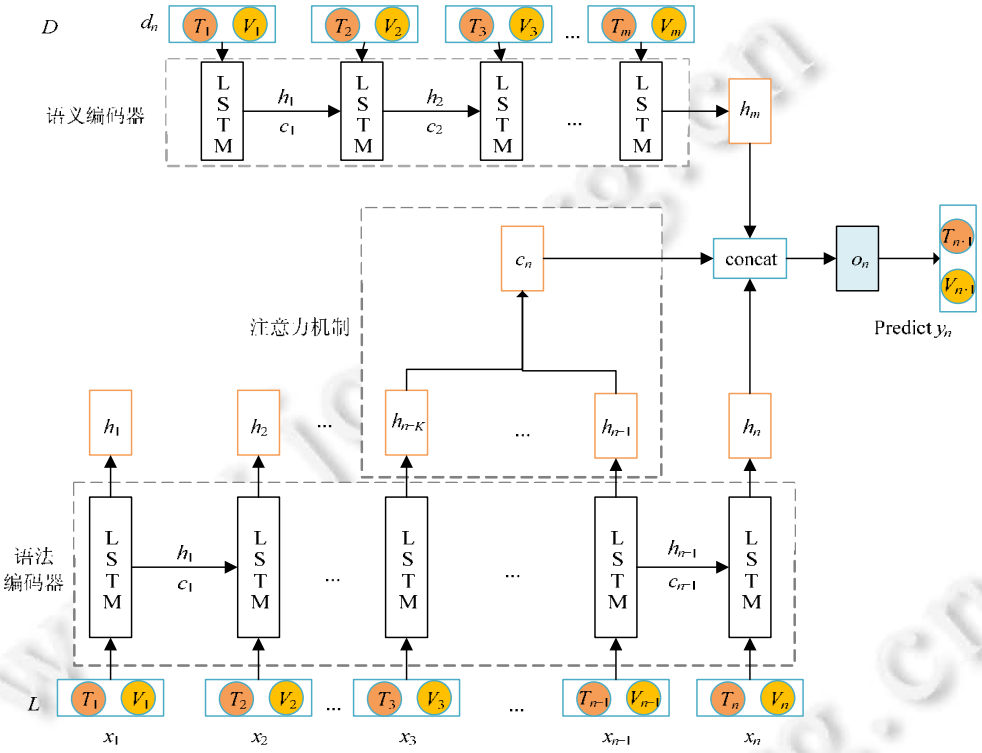

图 4　2-LSTM 模型架构

- 输出层

利用注意力层得到上下文向量 c_n 之后，为了对下一个节点进行预测，本模型将 c_n 和 LSTM 层的两个输出向量 h_m, h_n 进行结合，得到最终的输出向量 $o_n \in t$, o_n 对下一个节点值的分布进行编码并投影到词汇表中．最后，通过 softmax 函数得到最后的概率分布 $y_n \in V$, V 为词表的大小．如公式(7)、公式(8)所示，其中，$W^A \in t \times 3t$, $W^V \in V \times t$, $b^V \in V$ 均为可训练参数:

$$o_n = \tanh\left(W^A \begin{bmatrix} h_n \\ h_m \\ c_n \end{bmatrix}\right) \tag{7}$$

$$y_n=softmax(W^V o_n+b^V) \tag{8}$$

3.2 TCN-LSTM模型

由于 LSTM 网络模型的限制，为了提高模型解决长距离依赖的能力，在对语义建模的基础上，本文提出了 TCN-LSTM 模型，使得该模型实现既对程序的语法、语义信息进行建模，又能有效地解决程序的长距离依

赖问题.

图 5 展示了本文提出的 TCN-LSTM 模型, 对于语法输入序列 L, 使用 TCN 网络作为其编码器得到语法向量, 对语义输入序列 D, 使用 LSTM 网络进行编码得到语义向量. 同样地, 为了更好地结合上下文信息, 在使用 TCN 对语法输入序列 L 进行编码时, 对 TCN 的输出使用注意力机制得到上下文向量. 最后, 将语法向量、语义向量和上下文向量结合, 进行下一个节点 *type* 和 *value* 值的预测.

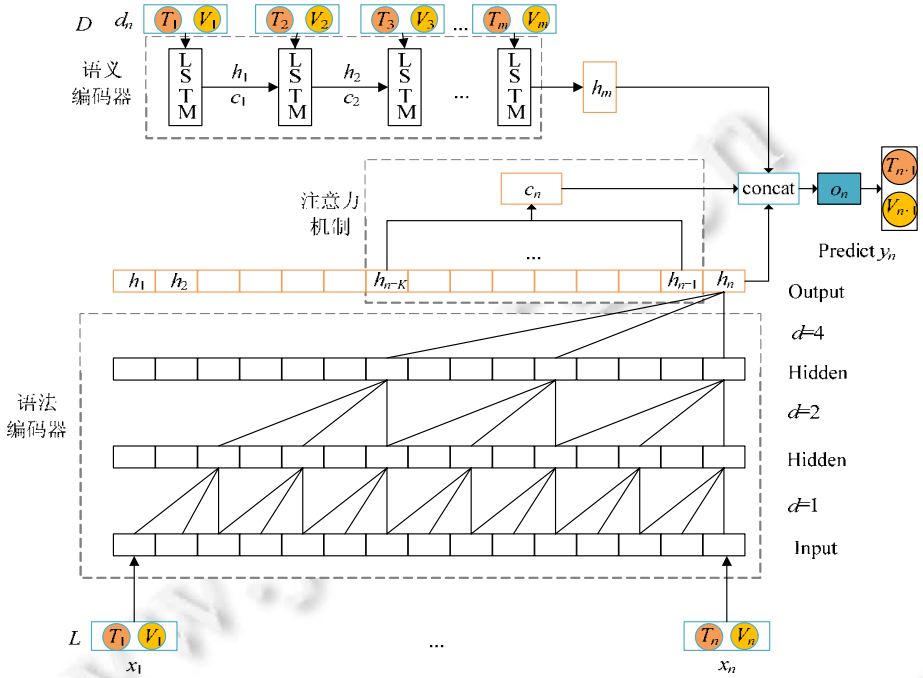

图 5 TCN-LSTM 模型架构

- 语法编码器

TCN 网络[28]作为 CNN 网络的一个分支, 采用了因果卷积保证对于时刻 n 的预测值 y_n 仅与 n 时刻之前的输入有关. 同时, 为了解决长序列数据的长距离依赖问题, 使用扩张卷积扩大模型的感受野, 利用扩张卷积中的过滤器跳过部分输入值来获取距离当前步更远的输入信息. 对于输入序列 $L=(x_1,x_2,x_3,\ldots,x_n)$, 经过卷积得到的输出序列为$(h_1,h_2,h_3,\ldots,h_n)$, TCN 层计算如公式(9)所示, 其中, f 为过滤器, d 为扩张因子, k 为过滤器大小:

$$h_n=(x\times_d f)(n)=\sum_{i=0}^{k-1} f(i)\times x_{n-d\times i} \tag{9}$$

为了使输出层接受更多的序列信息, TCN 通过增加扩张因子 d 和过滤器大小 k 来控制模型的记忆长短. 图 5 以 $d=2^i(i=0,1,2)$, $k=3$ 为例, 展示了 TCN 因果卷积和扩张卷积的过程. 同时, 为了避免在训练过程中出现梯度消失问题, 在输入层和输出层之间进行了残差连接.

- 语义编码器

在 TCN-LSTM 模型中, 本文使用与 2-LSTM 模型中相同的语义编码器结构, 得到语义向量 h_m.

- 注意力层

为获得输出向量之间的关联程度, 在使用 TCN 网络得到语法向量之后, 增加注意力层, 利用公式(3)–公式(6)计算输出向量与 n 时刻前 K 个输出向量之间的关系, 得到上下文向量 c_n.

- 输出层

为计算当前输出 y_n 的上下文向量, 在得到 h_n, h_m 和 c_n 后, 根据公式(10)得到输出向量 $o_n\in t$, 最后由公式(8)将输出向量 o_n 投影到词汇表中, 并通过 softmax 函数生成最终预测节点值的概率分布:

$$o_n = Relu\left(W^A\begin{bmatrix} h_n \\ h_m \\ c_n \end{bmatrix}\right) \tag{10}$$

- 多任务学习机制(MTL)

在代码补全问题中，节点的 *type* 和 *value* 是两个密切相关的属性，二者相互影响，相关任务之间的联系可以为每个任务的学习过程提供有效的约束，从一个任务中获得的知识可以帮助其他任务. 基于以上考虑，Liu 等人[31]提出将多任务学习用于代码补全技术中，并取得了较好的效果. 受到多任务学习思想的启发，本文同样将多任务学习技术加入本模型中，并且为解决多任务学习损失权重难以确定的问题，本文通过“不确定性”来调整损失函数中的加权超参，使得每个任务中的损失函数具有相似的尺度[32]，其损失函数如公式(11)所示，其中，$loss_T$ 和 $loss_V$ 分别为模型在 *type* 和 *value* 值预测得到的损失，w_T 和 w_V 为可训练参数:

$$loss = \frac{1}{w_T^2}\times loss_T + \frac{1}{w_V^2}\times loss_V \tag{11}$$

4 实验评估

4.1 数据集及预处理

本文使用 Python 数据集来评估比较不同的方法，该数据集从 Github 上获取，并多次用于之前的研究中[16,19,33]，可通过 https://www.sri.inf.ethz.ch/py150 获取，以源码和 AST 序列的形式存储. 为了获取数据集中程序语句的控制依赖关系并进行实验比较，本文基于 Python 的第三方库 AST 重新处理了源程序文件，以获取程序 AST 和语句的控制依赖表示，以便最终获得语法和语义输入序列. 由于源程序文件集中没有进行训练、测试数据集的划分，因此，最后构建的数据集无法与原数据集保持一致，本文所有实验均是基于重新构建的数据集进行. 本文对源程序划分和处理后得到的训练、测试数据集统计信息如表 1 所示，其中有 100 000 个程序文件用于训练，50 000 个程序文件用于测试. 处理过后，每个程序依旧存储为 AST 序列形式，包括语法和语义序列.

表 1 数据集统计

Train programs	100 000
Train programs lines	1.3×10^7
Test programs	50 000
Test programs lines	6.4×10^6
Train queries	5.8×10^7
Test queries	3.0×10^7
Type vocabulary	318
Value vocabulary	3.4×10^6

为了将本文方法与 Li 等人[19]的研究进行公平的比较，本文采用相同的数据预处理方式，在构建词汇表时增加了 3 个特殊值: *UNK* 表示词汇表外的值，*EOF* 表示每个程序的结束，*EMPTY* 表示非 AST 叶节点的 *value* 值. 通过表 1 发现，*value* 节点值的词表(3.4×10^6)非常大. 其原因在于: 程序中包含了大量用户自定义的标识符，并且用户自定义标识符只出现在 AST 序列的 *value* 节点值中. 本文无法构建包含所有词汇的词汇表，因此选取了程序中出现频率最高的 50 000 个 *value* 值来构建 *value* 节点的词汇表，将训练集和测试集中所有词汇表外的值设为 *UNK*; 并且在训练过程中，每当训练序列的目标值为 *UNK* 时，将其损失函数设置为 0. 在模型的训练和测试过程中，将所有目标值为 *UNK* 的预测都视为错误的预测. *Type* 值的词表大小在允许范围内，因此，本文将所有 *type* 值用于构建词汇表.

4.2 实验问题及实验设置

在代码补全问题中，普遍使用准确度作为度量评估的指标. 因此，为评估本文提出的方法，所有实验均使用准确度作为评估度量指标，其计算如公式(12)所示，本文记 *Accuracy*-1 为 Top-1, *Accuracy*-5 为 Top-5:

$$Accuracy\text{-}N=\frac{\text{Correct number among top } N \text{ predictions}}{\text{Total samples}} \tag{12}$$

为验证本文方法中加入语义建模的有效性和解决长距离依赖问题的能力, 本文实验讨论以下研究问题:

- 问题 1: 与目前预测效果较好的方法相比, 本文将语法和语义进行结合的有效性如何?

为了回答该问题, 本文将 2-LSTM 模型与 Li 等人[19]提出的指针混合网络进行比较. 本文复现了该模型, 并在原始数据集上得到了与原文基本相同的实验结果. 本文参照该方法中相同的实验参数设置, 其中, *batch_size* 为 128, *type* 值和 *value* 值词向量的嵌入大小分别问 300 和 1 200, 其嵌入词向量不经过预训练, 初始学习率为 0.001, 整个训练过程进行了 8 次迭代, 注意力窗口大小为 50, 训练过程中使用了交叉熵损失函数和 Adam 优化器[34]. 本次实验将每个程序的语法输入序列分成由 50 个连续的 AST 节点组成的段, 如果最后一段没有满, 则用 *EOF* 进行填充. 同时, 将语法输入序列中每个 AST 节点的语义输入序列长度设置为 20, 如果语义输入序列没有满, 同样使用 *EOF* 进行填充. 实验中, 隐藏状态 h_0、单元状态 c_0 初始化为零向量, 其余所有变量在[−0.05,0.05]区间进行随机初始化.

实验结果如表 2 所示, 本文方法在下一个节点的 *type* 值预测准确度基本与指针混合网络相同, 在 *value* 值的预测准确度优于指针混合网络. 本文在 *type* 值上的 Top-1 准确率为 79.2%, Top-5 准确率为 97.0%, 与 Li 等人[19]的指针混合网络在 Top-1 准确率上相差 0.1%, 在 Top-5 准确率上相同. 本文在 *value* 值上的 Top-1 准确率为 69.8%, Top-5 准确率为 78.0%, 比 Li[19]等人的指针混合网络分别高出了 0.4%, 0.8%. 由此得出, 2-LSTM 模型在 *type* 值上的预测达到与指针混合网络基本相同的准确率, 在 *value* 值上的预测达到了优于指针混合网络的准确率, 说明对程序的控制依赖信息进行编码在一定程度上帮助了代码补全模型对程序的理解, 并且对 *value* 值预测的提升效果较为明显.

表 2 2-LSTM 对比实验结果(%)

	Type		*Value*	
	Top-1	Top-5	Top-1	Top-5
Pointer mixture network	**79.3**	97.0	69.4	77.2
2-LSTM	79.2	97.0	**69.8**	**78.0**

考虑到程序中存在着大量的用户自定义标识符, 并且这些自定义标识符只存在于 *value* 节点中, 为进一步探究程序语义参与建模后, 模型在 *value* 节点预测准确率提升的表现, 本文在对 *value* 节点值预测的实验中, 统计了模型对自定义标识符的预测准确率, 结果见表 3.

表 3 自定义标识符对比实验结果(%)

	Value		*Identifier*	
	Top-1	Top-5	Top-1	Top-5
Pointer mixture network	69.4	77.2	70.0	80.3
2-LSTM	**69.8**	**78.0**	**72.6**	**81.1**

通过表 3 看出: 在对自定义标识符的预测中, 本文方法 Top-1 准确率为 72.6%, Top-5 准确率为 81.1%, 比 Li 等人[19]的指针混合网络分别高出了 2.6%, 0.8%. 通过实验结果得出: 在对自定义标识符的预测中, 本文将语法和语义信息进行结合后, 在预测准确率上比现有方法有较好提升, 因此验证了在加入语义特征建模后, 模型的预测效果比其他方法更加有效, 尤其是对自定义标识符的预测有较大提高.

- 问题 2: 本文提出的 TCN-LSTM 模型与其他语言模型相比, 在解决长距离依赖问题上是否更有效?

为了回答该问题, 本次实验将本文模型和目前代码补全问题中普遍使用的模型进行比较. 本次实验参数中最重要的是过滤器大小 k 和扩张因子 d, 来确保 TCN 具有足够大的接收场以覆盖任务所需的上下文数量, 但过大的 k 和 d 反而会制约模型的表现. 本文通过设置不同的 k 和 d 进行了多次实验, 并最终将 k 和 d 设置为 6, 梯度裁剪率设为 0.4. 所有变量均由高斯分布 $N(0,0.01)$初始化. 初始学习率设为 0.001, *batch_size* 为 32. *type* 值和 *value* 值词向量的嵌入大小分别为 300 和 600, 训练过程中同样使用了交叉熵损失函数和 Adam 优化器, 整个训练过程进行了 8 次迭代. 另外, 注意力窗口大小为 50.

在本次实验中, 选用多种网络并进行了实验对比, 其中包括标准 LSTM 网络、标准 LSTM 网络结合注意力机制、指针混合网络和 2-LSTM 网络. 本次实验同样选取 Top-1 和 Top-5 的预测准确率进行比较, 实验结果见表 4.

通过表 4 看出: 本文提出的 TCN-LSTM 取得了较高的准确率, 且优于现有方法. 与指针混合网络相比, *type* 和 *value* 预测在 Top-1 的准确率分别提高了 2.3%和 2.4%, 在 Top-5 的准确率分别提高了 0.4%和 0.5%, 并且优于 2-LSTM 模型, 表明 TCN 模型可以更好地捕获序列长距离依赖信息.

表 4 实验对比结果(%)

	Type		*Value*	
	Top-1	Top-5	Top-1	Top-5
LSTM	77.9	97.0	67.1	77.6
Attentional LSTM	78.9	97.2	69.2	78.1
Pointer mixture network	79.3	97.3	69.4	78.2
2-LSTM	79.2	97.3	69.8	78.4
TCN-LSTM (no MTL)	**81.6**	**97.7**	**71.8**	**78.7**

为进一步探究加入多任务学习技术对模型效果的影响, 本次实验采用上述实验相同的参数设置. 考虑到数据集不同, 本文无法与 Liu 等人[31]的实验结果直接进行比较, 只对加入多任务学习技术前后的 TCN-LSTM 模型进行比较, 实验结果见表 5.

表 5 加入多任务学习前后实验结果(%)

	Type		*Value*	
	Top-1	Top-5	Top-1	Top-5
TCN-LSTM (no MTL)	81.6	97.7	71.8	78.7
TCN-LSTM (with MTL)	**82.0**	**97.8**	**72.2**	**79.5**

通过表 5 可以看出: 在加入多任务学习后, 模型在 *type* 值和 *value* 值上的预测准确率都得到了提高, 其中, *type* 值的 Top-1 准确率提高了 0.4%, 在 *value* 值预测的准确率提高了 0.4%. 由此可得出: 在加入多任务学习之后, 本文模型在预测准确度上取得了进一步的提高.

4.3 讨论与分析

通过问题 1 实验中 2-LSTM 模型的 *value* 值预测准确率高于现有方法, 说明利用程序的控制依赖信息参与建模, 可以使得神经网络模型在一定程度上学习到程序的语义信息; 并且语义与语法信息结合共同参与下一节点值的预测, 可以取得更好的预测效果. 通过进一步的实验发现, 2-LSTM 模型在特殊的自定义标识预测准确率方面取得了很好地提升, 同样也证明了程序的语义信息对于程序中自定义标识符预测至关重要, 并且本文通过编码得到的语义特征更好地捕获了自定义变量、方法的语义信息. 但同时本文也发现: 当程序中不存在结构性语句(如 if, while 等)时, 程序语句之间没有较强的控制依赖关系, 神经网络模型对程序语义的理解效果也无法得到提升. 因此, 数据集的特点与实验结果有着直接的关系. 为了与其他方法进行直观的比较, 本文使用了公开数据集, 该数据集从 Github 上获取且具有一定的普遍性.

通过问题 2 实验中 *type* 和 *value* 值预测准确率都高于现有方法, 说明 TCN 网络可以更有效地捕获长序列程序语句的长距离依赖信息. 其原因在于: TCN 网络通过扩张卷积考虑了输入序列中更长的历史信息, 过滤器大小 k 和扩张因子 d 决定了 TCN 网络扩张卷积的接收场, 同时也决定了 TCN 网络捕捉长距离依赖信息的效果. 为了探究不同的 k 和 d 对 TCN-LSTM 的影响, 本文通过设置不同的 k 和 d 来观察预测准确率的变化, 除了 k 和 d 设置不同, 本次实验其他参数的设置均和实验 2 相同, 其实验结果如图 6 所示.

在图 6 中, 横坐标为训练迭代次数, 纵坐标为测试准确率. 通过图 6 看出, 不同的过滤器大小 k 和扩张因子 d 的值对 TCN-LSTM 模型效果具有一定的影响. 在一些参数设置下(如 k=2, d=6 或 k=6, d=2), 其收敛速度较快, 但准确率略低; 对于较大的 k 和 d, 准确率得到了提高. 当 k=6, d=6 时, 模型充分挖掘到长序列数据的长距离依赖信息, 预测效果达到最优.

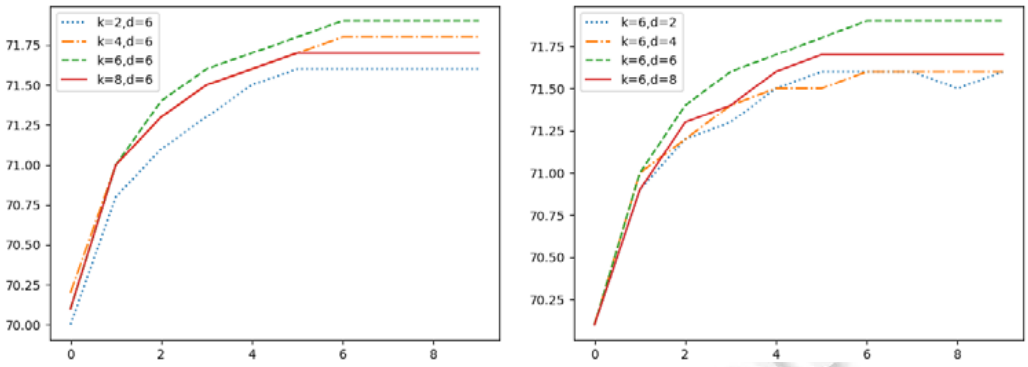

图 6　不同 *k*, *d* 下的准确率

在模型设计中，本文选择使用 TCN 作为语法编码器，LSTM 作为语义编码器，并通过实验验证了其有效性. 本文没有选择 TCN 作为语义编码器，为探究不同的语义编码器带来的模型预测准确率的变化，本文分别使用 LSTM, TCN 作为语义编码器进行实验，其结果如表 6 所示.

表 6　不同语义编码器实验结果(%)

语义编码器	*Type*	*Value*
LSTM	81.6	71.8
TCN	81.6	71.7

由表 6 可以看出，使用 LSTM 作为语义编码器获得了与 TCN 基本相同的结果. 其原因在于: 在对语义输入序列进行编码时，当前输入序列长度下 LSTM 不受长距离依赖的限制，最后获得的是语义输入的“概要”信息向量，因此，LSTM 网络可以达到与 TCN 相同的效果. 同时，在使用 TCN 网络作为语义编码器，超参数 *k* 和 *d* 的设置决定了语义编码器的效果, LSTM 中不需要超参数的设置和调整，且 LSTM 占用更少的内存. 因此，基于以上考虑，本文选择 LSTM 作为语义编码器.

5　总结与展望

本文研究了用于动态语言 Python 的代码补全神经网络模型，为了更好地结合程序语义信息，本文提出利用程序控制依赖信息进行语义编码; 同时，为了解决长距离依赖问题，提出了 TCN-LSTM 模型，用于捕获更远距离的输入序列信息. 实验结果表明: 本文模型取得了比现有模型更好的效果，并且在针对于自定义标识符的预测效果具有明显提升. 将来会考虑通过解决词汇表外单词(OoV)预测问题来进一步提高模型的有效性. 此外，将探讨模型如何根据用户自己编写的代码来进行更新学习，从而使得模型具有更强的用户可自定义性.

References:

[1] Hu X, Li G, Liu F, Jin Z. Program generation and code completion techniques based on deep learning: Literature review. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1206−1223 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

[2] Bruch M, Monperrus M, Mezini M. Learning from examples to improve code completion systems. In: Proc. of the 7th Joint Meeting of the European Software Engineering Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Engineering. ACM, 2009. 213−222. [doi: 10.1145/1595696.1595728]

[3] Vechev M, Yahav E. Code completion with statistical language models. In: Proc. of the 35th ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2014). New York: Association for Computing Machinery, 2014. 419−428. [doi: 10.1145/2594291.2594321]

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

[5] Dong L, Lapata M. Language to logical form with neural attention. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics, Vol.1. 2016. 33−43. [doi: 10.18653/v1/P16-1004]

[6] Sun Z, Qihao Z, Yingfei X, Yican S, Lili M, Zhang L. TreeGen: A tree-based transformer architecture for code generation. In: Proc. of the AAAI Conf. on Artificial Intelligence. 2020. 8984−8991. [doi: 10.1609/aaai.v34i05.6430]

[7] Maalej W, Pagano D. On the socialness of software. In: Proc. of the Int'l Conf. on Dependable, Autonomic and Secure Computing. 2011. 864−871. [doi: 10.1109/DASC.2011.146]

[8] Nguyen TT, Nguyen AT, Nguyen HA, Nguyen TN. A statistical semantic language model for source code. In: Proc. of the Joint Meeting on Foundations of Software Engineering. 2013. 532−542. [doi: 10.1145/2491411.2491458]

[9] Peng H, Mou LL, Li G, Liu YX, Zhang L, Jin Z. Building program vector representations for deep learning. In: Proc. of the 8th Int'l Conf. on Knowledge Science, Engineering and Management (KSEM 2015). Berlin, Heidelberg: Springer, 2015. 547−553. [doi: 10.1007/978-3-319-25159-2_49]

[10] Nguyen TD, Nguyen AT, Phan HD, Nguyen TN. Exploring API embedding for API usages and applications. In: Proc. of the 39th Int'l Conf. on Software Engineering (ICSE 2017). IEEE, 2017. 438−449. [doi: 10.1109/ICSE.2017.47]

[11] Hindle A, Barr ET, Su Z. On the naturalness of software. In: Proc. of the 34th Int'l Conf. on Software Engineering (ICSE). IEEE, 2012. 837−847. [doi: 10.1109/ICSE.2012.6227135]

[12] Bielik P, Raychev V, Vechev M. Phog: Probabilistic model for code. In: Proc. of the Int'l Conf. on Machine Learning. 2016. 2933−2942.

[13] Linares-Vásquez M, Poshyvanyk D. Toward deep learning software repositories. In: Proc. of the Working Conf. on Mining Software Repositories. 2015. 334−345. [doi: 10.1109/MSR.2015.38]

[14] Dam HK, Tran T, Pham TTM. A deep language model for software code. In: Proc. of the Foundations Software Engineering Int'l Symp. 2016. 1−4.

[15] Bielik P, Raychev V, Vechev M. Program synthesis for character level language modeling. In: Proc. of the ICLR. 2017.

[16] Liu C, Wang X, Shin R. Neural code completion. In: Proc. of the ICLR 2017. 2017.

[17] Wang S, Liu JY, Qiu Y, Ma ZY, Liu JF, Wu ZH. Deep learning based code completion models for programming codes. In: Proc. of the 3rd Int'l Symp. on Computer Science and Intelligent Control (ISCSIC 2019). New York: Association for Computing Machinery, 2019. Article 16. [doi: 10.1145/3386164.3389083]

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

[19] Li J, Wang Y, King I, *et al*. Code completion with neural attention and pointer networks. In: Proc. of the Int'l Joint Conf. on Artifical Intelligence (IJCAI). 2018. 4159−4165.

[20] Vinyals O, Fortunato M, Jaitly N. Pointer networks. In: Proc. of the Int'l Conf. on Neural Information Processing Systems. 2015. 2692−2700.

[21] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv:1409.00473, 2014.

[22] Cheng J, Dong L, Lapata M. Long short-term memory-networks for machine reading. arXiv:1601.06733, 2016.

[23] Zhang J, Wang X, Zhang H, *et al*. A novel neural source code representation based on abstract syntax tree. In: Proc. of the IEEE/ACM 41st Int'l Conf. on Software Engineering (ICSE). 2019. 783−794. [doi: 10.1109/ICSE.2019.00086]

[24] Wei H, Li M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In: Proc. of the 26th Int'l Joint Conf. on Artificial Intelligence. 2017. 3034−3040.

[25] Oord AVD, Dieleman S, Zen H, *et al*. WaveNet: A generative model for raw audio. arXiv:1609.03499, 2016.

[26] Dauphin YN, Fan A, Auli M, Grangier D. Language modeling with gated convolutional networks. In: Proc. of the Int'l Conf. on Machine Learning. 2017. 933−941.

[27] Gehring J, Auli M, Grangier D, *et al*. A convolutional encoder model for neural machine translation. arXiv:1611.02344, 2016.

[28] Bai S, Kolter JZ, Koltun V. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. arXiv: 1803.01271, 2018.

[29] Tian S. JavaScript static slicing tool. Computer and Modernization, 2016(8): 46−51 (in Chinese with English abstract).

[30] Hochreiter S, Schmidhuber J. Long short-term memory. Neural Computation, 1997, 9(8): 1735−1780. [doi: 10.1162/neco.1997.9.8.1735]

[31] Liu F, Li G, Wei BL, Xia X, Fu ZY, Jin Z. A self-attentional neural architecture for code completion with multi-task learning. In: Proc. of the 28th Int'l Conf. on Program Comprehension (ICPC 2020). New York: Association for Computing Machinery, 2020. 37−47. [doi: 10.1145/3387904.3389261]

[32] Cipolla R, Gal Y, Kendall A. Multi-Task learning using uncertainty to weigh losses for scene geometry and semantics. In: Proc. of the IEEE/CVF Conf. on Computer Vision and Pattern Recognition. Salt Lake City, 2018. 7482−7491. [doi: 10.1109/CVPR.2018.00781]

[33] Raychev V, Bielik P, Vechev M. Probabilistic model for code with decision trees. In: Proc. of the Int'l Conf. on Object-oriented Programming, Systems, Languages, and Applications. 2016. 731−747. [doi: 10.1145/3022671.2984041]

[34] Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv:1412.6980, 2014.

附中文参考文献:

[1] 胡星, 李戈, 刘芳, 金芝. 基于深度学习的程序生成与补全技术研究进展. 软件学报, 2019, 30(5): 1206−1223. http://www.jos.org.cn/1000-9825/5717.htm [doi: 10.13328/j.cnki.jos.005717]

[29] 田生. JavaScript 静态切片工具. 计算机与现代化, 2016(8): 46−51.

付善庆(1996－), 男, 硕士, 主要研究领域为代码推荐, 代码补全.

赵瑞莲(1964－), 女, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为软件测试, 软件可靠性分析.

李征(1974－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为智能化软件工程, 大规模程序源代码的分析与测试.

郭俊霞(1977－), 女, 博士, 副教授, CCF 高级会员, 主要研究领域为网络用户行为分析, 网络应用程序测试.