计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2021,30(8):256−265 [doi: 10.15888/j.cnki.csa.008042]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于图卷积神经网络的函数自动命名①

王　堃, 李　征, 刘　勇

(北京化工大学 信息科学与技术学院, 北京 100029)
通讯作者: 李　征, E-mail: lizheng@mail.buct.edu.cn

摘　要: 函数自动命名技术旨在为输入的源代码自动生成目标函数名, 增强程序代码的可读性以及加速软件开发进程, 是软件工程领域中一项重要的研究任务. 现有基于机器学习的技术主要是通过序列模型对源代码进行编码, 进而自动生成函数名, 但存在长程依赖问题和代码结构编码问题. 为了更好的提取程序中的结构信息和语义信息, 本文提出了一个基于图卷积 (Graph Convolutional Network, GCN) 的神经网络模型—TrGCN (a Transformer and GCN based automatic method naming). TrGCN 利用了 Transformer 中的自注意力机制来缓解长程依赖问题, 同时采用 Character-word 注意力机制提取代码的语义信息. TrGCN 引入了一种基于图卷积的 AST Encoder 结构, 丰富了 AST 节点特征向量的信息, 可以很好地对源代码结构信息进行建模. 在实证研究中, 使用了 3 个不同规模的数据集来评估 TrGCN 的有效性, 实验结果表明 TrGCN 比当前广泛使用的模型 code2seq 和 Sequence-GNNs 能更好的自动生成函数名, 其中 *F*1 分数分别提高了平均 5.2%、2.1%.

关键词: 深度学习; 图卷积神经网络; 代码表示方式

引用格式: 王堃,李征,刘勇.基于图卷积神经网络的函数自动命名.计算机系统应用,2021,30(8):256–265. http://www.c-s-a.org.cn/1003-3254/8042.html

Automatic Function Naming Based on Graph Convolutional Network

WANG Kun, LI Zheng, LIU Yong

(College of Information Science and Technology, Beijing University of Chemical Technology, Beijing100029, China)

Abstract: Automatic method naming, as an important task in software engineering, aims to generate the target function name for an input source code to enhance the readability of program codes and accelerate software development. Existing automatic method naming approaches based on machine learning mainly encode the source code through sequence models to automatically generate the function name. However, these approaches are confronted with problems of long-term dependency and code structural encoding. To better extract structural and semantic information from programs, we propose a automatic function naming method called TrGCN based on Transformer and Graph Convolutional Network (GCN). In this method, the self-attention mechanism in Transformer is used to alleviate the long-term dependency and the Character-word attention mechanism to extract the semantic information of codes. The TrGCN introduces a GCN-based AST Encoder that enriches the eigenvector information at AST nodes and models the structural information of the source code well. Empirical studies are conducted on three Java datasets. The results show that TrGCN outperforms conventional approaches, namely code2seq and Sequence-GNNs, in automatic method naming as its *F*1-score is 5.2% and 2.1% higher than the values of the two approaches, respectively.

Key words: deep learning; Graph Convolutional Network (GCN); code representation

① 基金项目: 国家自然科学基金 (61902015)
Foundation item: National Natural Science Foundation of China (61902015)
收稿时间: 2020-11-23; 修改时间: 2020-12-22; 采用时间: 2021-01-07; csa 在线出版时间: 2021-07-31

1 引言

函数自动命名是根据现有源代码自动生成可读性强的自然语言来描述函数的功能. 在软件维护过程中, 开发者往往需要花费大量的时间去理解一段代码所对应的功能. 其中, 一种重要的方式便是通过阅读函数名来快速的了解代码的功能, 从而节约开发者的时间. 尽管开发者可以通过函数名理解函数的功能, 但是在实际开发过程中往往这些函数名并不是完整的或具有时效性的. 这类问题可能会误导开发者, 导致开发者花费大量的时间来理解程序, 影响开发效率. 现有工作表明: “函数名大多是常规编程语言中聚集函数行为的最小命名单位, 因此函数名是函数抽象的基石. ”[1]. 因此, 在软件开发过程中, 给定一段代码, 如何自动生成一个正确并且相关联的函数名变得尤为重要.

随着深度学习技术的快速发展, 许多研究人员尝试使用深度学习模型来解决这个问题, 例如 sequence-to-sequence 模型[2−5]. Allamanis 等[6] 将代码解析成 token 序列, 然后利用卷积神经网络 (CNN) 对 token 序列进行建模, 通过卷积操作提取代码特征, 并引入注意力机制学习代码序列与函数名之间的关键信息. 但是, 使用 token 序列仅仅是代码的展开, 模型很难去学习到代码中的结构信息, 同时 token 序列过长导致模型很难学习到其中的语义信息. 为了解决这个问题, Alon 等[7,8] 通过将代码转化为抽象语法树 (AST), 并根据每一个叶节点 (变量名、方法名等) 从 AST 中提取相关路径 (AST Path) 来作为代码的特征表示, 该特征向量包含了一定的结构信息以及语义信息. 之后 Alon 等[9] 提出了 code2seq 模型, 对先前模型进行了再一次改进, 利用 BiLSTM 网络对 AST 路径进行模型构建, 然后通过引入注意力机制来学习所生成的单词与各条 AST 路径的相关性. 该方法在函数名预测任务中达到了目前最好效果. 然而, 该方法依旧存在弱化了 AST 树形结构信息的缺陷, 从而导致其很难学习到 AST 路径之间节点的信息. Fernandes 等[10] 提出一种结合 sequence-to-sequence 模型和图神经网络 (GNN) 模型的方法来表示代码, Sequence-GNNs. 该方法使用 BiLSTM 对代码序列进行编码, 然后引入图神经网络对编码后的代码序列进行建模, 最后得到的向量便是代码的特征表示. 但是 BiLSTM 等循环神经网络模型存在长程依赖问题, 该模型很难捕捉到长距离文本之间的关系, 会导致一些关键信息丢失.

可以看出, 目前在函数命名任务中面临两大挑战: (1) 代码的表示方法. 对代码进行特征提取时, 不仅要提取其语义信息, 而且更要提取其结构信息. 在目前 code2seq 模型中, 该方法将代码表示为 AST 路径的集合, 引入注意力机制来提取代码的结构以及语义信息. 但是该方法对路径建模的同时一定程度上破坏了 AST 的结构, 而且只能提取路径内部节点之间的信息, 并不能有效提取路径间节点之间的信息, 尤其是数据流和控制流信息. (2) 长程依赖问题[11]. 当前的代码片段可能依赖于离它很远的代码段. 例如, 一个程序中第 90 行的语句“a = a + 2; ”依赖于第 10 行语句“int a = 10; ”的变量定义. 随着代码长度的增加, BiLSTM 等循环神经网络很难捕捉到长距离文本之间的信息.

本文提出一种新颖的神经网络模型 TrGCN, 采用了近期提出的 Transformer[12] 模型, 该模型可以有效缓解长程依赖问题. 然而原生的的 Transformer 模型主要用于自然语言翻译中, 并不能应用在结构性很强的代码中. 所以我们在每个 Transformer encoder block 中添加了两层图卷积层[13], 将节点以及其相关节点进行卷积操作, 从而丰富代码的结构信息. 在函数命名任务中, 我们使用了 Java-small、Java-med 和 Java-large[6,9] 数据集, 在 7 个基准模型来评估 TrGCN. 实验结果表明我们的模型明显优于其他基准模型, 相比于模型 code2seq[9] 和 Sequence-GNNs[10], TrGCN 在 $F1$ 指标上分别提高了平均 5.2%、2.1%; 在 ROUGE-2 和 ROUGE-L 指标上, 比 Sequence-GNNs 分别提高了 0.7%、3.6%.

2 相关背景

随着开源代码库的不断增加, 数据规模的不断增大, 越来越多的研究者开始使用机器学习对程序源代码进行建模, 学习其中的信息, 帮助开发者更好的理解程序. 起初研究人员把源代码作为一种符号序列, 将源代码转化为 token 序列, 并通过 RNN 或 LSTM 等序列模型 (Seq2Seq) 对其进行建模和学习序列中的信息. Allamanis 等[14] 利用 Logbilinear 模型, 通过对变量、函数名以及类名的上下文信息进行预测. 之后 Allamanis 等[6] 提出了 ConvAttention 模型, 将源代码看做一组 token 序列, 并使用神经网络对其建模, 通过 CNN 以及注意力机制来学习代码和函数名之间的关键信息. Lyer

等[15] 和 Hu 等[16] 在代码注释生成任务中利用 RNN 对源代码建模, 然后根据其建模结果使用 RNN 进行解码, 同时引入了注意力机制来学习注释文本中各个单词与函数体之间的关系. 程序语言与自然语言最大不同之处在于程序语言是一种高度结构化的语言, 而基于 token 的程序表示方法仅仅将代码线性展开, 丢失了程序语言的结构信息, 导致代码特征向量信息的不完整.

为了对程序的结构信息进行建模, 研究人员使用 AST 来对源代码进行建模. Li 等[17] 和 Liu 等[18] 在代码补全任务中, 将程序源代码解析为 AST, 然后遍历 AST 得到一组节点序列, 通过对其建模, 实现代码补全. Sun 等[19,20] 在代码生成任务中利用树卷积的方式, 将 AST 中的节点、双亲节点和祖先节点结合起来, 通过预测语法规则实现代码生成. Alon 等[8,9] 通过从 AST 中提取 AST 路径来表示程序的结构信息以及语法信息, 然后对 AST 路径建模, 并且引入注意力机制, 通过学习函数名和 AST 路径之间的关系来进行函数名预测. 虽然基于 AST 的程序表示方法有效的提取了程序的结构信息, 但是并不完整, 缺少对数据流以及控制流信息的提取.

随着图神经网络 (GNN) 的兴起, 研究人员尝试将程序源代码转化为图的形式并对其进行建模. 其转化方式为通过给 AST 增加更多的边, 使得模型可以更好的理解节点之间的关系. Allamanis 等[21] 提出了一种基于门控图神经网络 (GGNN)[22] 模型, 利用图的形式来表示程序源代码. 该方法通过将源代码的 AST 扩展为图的形式, 其中图中的节点表示 AST 序列中的 token, 图中的边表示节点与节点之间的相互关系. Fernandes 等[10] 在代码注释任务中提出一种将 GNN 与 Seq2Seq 结合的模型, 首先使用 BiLSTM 对程序进行编码, 得到其特征向量表示, 然后利用 GNN 对其进行建模, 最后得到特征向量作为程序的特征表示. 但是先前的研究所使用的模型都是 RNN、LSTM 等循环神经网络模型, 存在长程依赖问题, 导致对代码特征信息的提取不够完整. 本文利用了 Transformer 中多头自注意力机制的特性来缓解这个问题.

为了将结构信息的提取与缓解长程依赖问题相结合, 本文提出了一种基于 Transformer 模型的图卷积神经网络 (GCN) 模型 TrGCN, 使用图卷积提取代码的结构信息, 并且利用 Transformer 模型来缓解长程依赖问题.

3 模型结构

图 1 显示了 TrGCN 模型的结构图, 主要包括两个部分: AST 编码器和解码器.

3.1 Encoder-Decoder 结构

TrGCN 使用了 Transformer 模型, 是一种标准的 Encoder-Decoder 结构, 其中 Encoder 编码器将输入的 token 序列 $(x_1,\cdots,x_n)$ 转化为一组连续的特征向量 $z=(z_1,\cdots,z_n)$. 得到特征向量 z 后, Decoder 解码器根据 z 输出一组 token 序列 $(y_1,\cdots,y_m)$, 因此模型化之后的条件概率为: $p(y_1,\cdots,y_m|x_1,\cdots,x_n)$. 在解码阶段, 预测出的 token 取决于先前 token 的信息, 该概率模型为:

$$p(y_1,\cdots,y_m|x_1,\cdots,x_n)=\prod_{j=1}^{m}p(y_j|z_1,\cdots,z_n) \quad (1)$$

3.2 AST 编码器

一段代码可以解析为一颗抽象语法树, 其叶节点被称为终结符, 一般是用户所定义的变量以及方法名等; 非叶节点被称为非终结符, 代表代码中的一些结构, 例如循环、表达式以及变量声明. 图 2 表示 AST 的一部分, 其中变量名 (如 num) 代表终结符, 而条件语句 (IfStmt) 和表达式语句 (ExpressionStmt) 这些语法结构代表非终结符. 由于非终结符拥有丰富的结构信息, 终结符拥有丰富的语义信息, 所以 AST 编码器的目标就是尽可能的提取 AST 的结构信息和语义信息来帮助模型更好的理解代码. 首先将代码解析成一棵 AST, 利用前序遍历的方式将 AST 展开成一组 token 序列. 假设 AST 为 T, 则 token 序列为 $T=\{n_1,\cdots,n_L\}$, 其中 L 为序列的长度. 每个 token 可以被分割为一组字符序列 $n_i=\left\{c_1^{n_i},\cdots,c_m^{n_i}\right\}$, 其中 m 表示单词最大长度, 如果长度小于 m, 则会被填充字符填充至 m. 所有的 token 序列以及字符序列通过 table-look-up embedding 的方式来表示成具有真值的初始特征向量 $\{n_1,\cdots,n_L\}$, $\left\{c_1^{n_i},\cdots,c_m^{n_i}\right\}$.

3.3 输入序列的特征表示方法

Character Embedding. 无论是在自然语言中还是在代码中, 都会出现一些相似的单词并且拥有相近的语义, 例如 name, names. 为了利用这个特性, 本文使用 character embedding 的方式来表示每个 token, 公式如下:

$$n_i^c=W^{\text{conv}}\left[c_1^{n_i},\cdots,c_M^{n_i}\right] \quad (2)$$

其中, W^{conv} 表示一层卷积权重, 通过卷积操作来提取字符之间的特征; M 为字符序列的最大长度, 如果长度小于 M, 则会被填充字符填充至 M. 在卷积层之后会附加一层归一化层 (layer normalization[23]) 来对特征向量进行

归一化. 最终使用加和的方式来结合 word 和 character 的特征向量:

$$n_i=n_i+n_i^c \tag{3}$$

其中, character 的特征向量n_i^c会保留到下文的 character-word 注意力层, 利用注意力机制来更好的融合 character 和 word 特征向量.

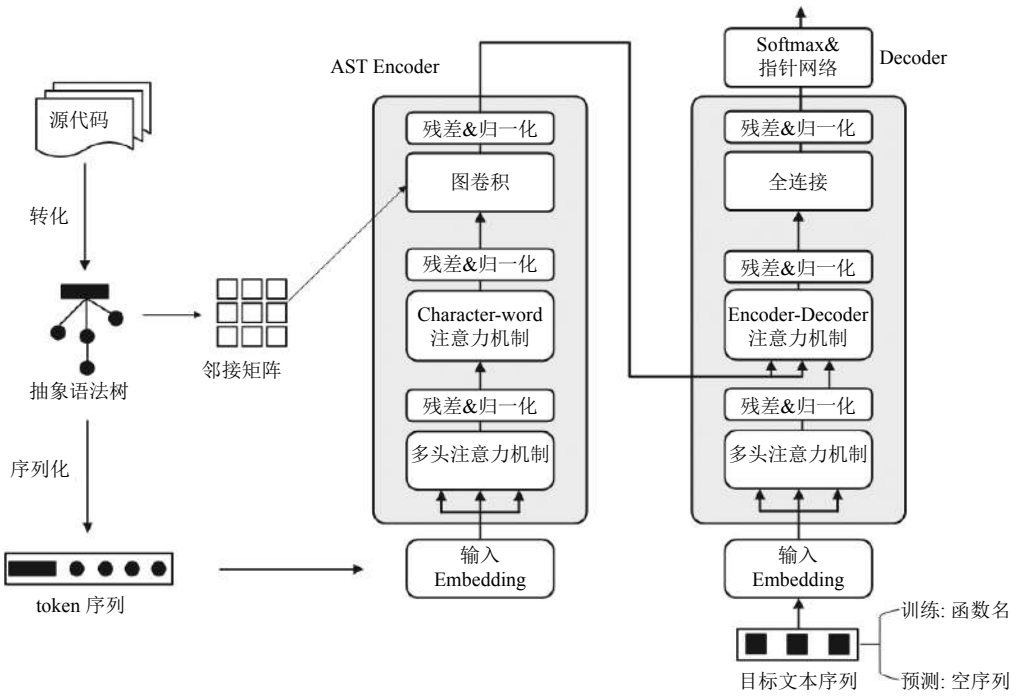

图 1 TrGCN 模型结构

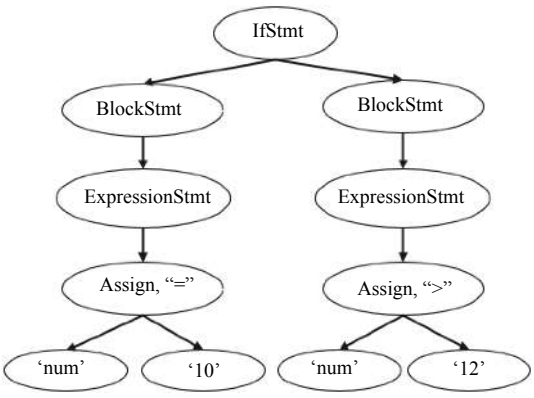

图 2 抽象语法树 (部分)

3.4 AST 编码器结构

AST 编码器是由一组 encoder 组成 (总共有 N 个 encoder). 其中每个 encoder 中都包含 3 个不同的子层, 分别为多头自注意力层, character-word 注意力层以及图卷积层, 通过这 3 个子层来提取输入代码中的特征信息. 本文将会在下面的小结中详细介绍这些子层. 其中每个子层之间添加了残差连接 (residual connection[24]) 以及归一化层, 其中残差连接可以有效缓解网络退化问题和梯度弥散问题.

多头自注意力层: AST 编码器中的 self-attention 子层与原生 Transformer 相同, 使用了多头自注意力机制来捕捉长程依赖信息.

假设有一组输入的 token 序列$n_1,\cdots,n_L$, 从上文可知, 通过 look-up table embedding 方式可以得到一组初始化的特征向量$\{n_1,\cdots,n_L\}$. 由于输入序列是有序的, 为了让模型学习到序列中的位置关系, 需要在特征向量中加入位置信息, 并使用 position embedding 来对位置信息进行编码:

$$PE(pos,2i)=\sin\left(pos/10\,000^{2i/d_{model}}\right) \tag{4}$$

$$PE(pos,2i+1)=\cos\left(pos/10\,000^{2i/d_{model}}\right) \tag{5}$$

其中, PE 是二维矩阵, 行表示节点, 列表示节点的词向量; pos 表示节点在输入序列中的位置, d_{model} 表示词向量的维度, i 表示词向量的位置.

每个 Transformer 中的 encoder 和 decoder 都会通

过多头自注意力机制来学习数据中的非线性特征, 它的主要思想是将一个序列文本的上下文词汇通过矩阵乘法的方式, 使每个词汇都拥有其上下文词汇的特征信息. 而传统的注意力机制只是关注了源文本和目标文本之间关系. 多头自注意力机制是将多个拥有不同参数的自注意力机制进行融合, 使模型增强了关注序列文本不同位置的能力; 同时丰富了序列文本词汇的特征信息. 多头自注意力机制拥有更多线性变换, 变换之后会通过激活函数进行非线性转换, 从而提高了模型非线性的学习能力. 其公式如下:

$$Y^{\text{self}} = concat(head_1, \cdots, head_H)W_h \tag{6}$$

其中, H 表示头的数量, W_h表示权重. 同时在每个头中都会添加一层 self-attention 层, 如下:

$$head_t = softmax\left(QK^{\text{T}}/\sqrt{d_k}\right)V \tag{7}$$

其中, $d_k = d/H$表示每个特征向量的长度. Q, K, V 的计算如下:

$$[Q, K, V] = [x_1, \cdots, x_L]^{\text{T}}[W_Q, W_K, W_V] \tag{8}$$

其中, W_Q, W_K, W_V是模型的参数, x_i是 AST encoder 的输入. 对于第一个 encoder 的输入来说, 它是将 word、character 以及 position 的特征向量进行加和, 例如$n_i+n_i^c+PE_i$. 而其他 encoder 的输入来自上一个 encoder 的输出.

Character-word 注意力层 [19]: 经过 self-attention 层计算得到的特征向量, 本层将会把该向量与保留下来的 character embedding 向量进行结合. 对于每一个单词, 将对y_i^{self}进行两个不同线性转化, 得到控制向量q_i和权重向量k_i^y, 对n_i^c进行一个线性转化得到权重向量$k_{k_i^c}$, 最后经过 Softmax 计算得到:

$$\left[\alpha_i^y, \alpha_i^c\right] = softmax\left\{q_i^{\text{T}}k_i^y, q_i^{\text{T}}k_i^c\right\} \tag{9}$$

得到的两个注意力分数是用来衡量 Transformer 中 self-attention 层的输出y_i^{self}和 character embedding n_i^c, 然后分别进行线性转化得到v_i^y和v_i^c:

$$h_i = \left[\alpha_i^y v_i^y + \alpha_i^c v_i^c\right] \tag{10}$$

最后 character-word 注意力层的输出为$Y^{\text{cwa}} = [h_1, \cdots, h_L]$.

图卷积层: 考虑到模型很难学习节点的双亲节点、祖先节点等相关节点之间的关系, 例如: 节点 a 和节点 b 在文本中距离很远, 但是在 AST 结构中距离却很近, 因此, 对于传统的 Transformer 模型来说很难提取这样的结构信息.

为了更好的提取程序的结构信息, 将 AST 通过增加边的方式扩展为图的形式, 其中使用了 7 种边的类型[21]:

(1) Child edge: 当前节点与孩子之间的边.

(2) Parent edge: 当前节点与双亲节点之间的边.

(3) Grandparent edge: 当前节点与祖先节点之间的边.

(4) Next edge: 当前节点与其先驱节点 (语义) 之间的边.

(5) LastUse edge: 当前节点与在词法上最近节点之间的边.

(6) Next sibling edge: 当前节点与兄弟节点之间的边.

(7) Subtoken edge: subtokens 之间的边.

TrGCN 将 AST 视为图的形式, 同时使用邻接矩阵来表示 AST 扩展之后的图. 根据上文信息, 可以得到邻接矩阵$M_1, \cdots, M_7$. 给定一个邻接矩阵M_1, 如果节点α_i是节点α_j的双亲节点, 则$M_{1,ij} = 1$. 假设节点的特征向量被表示为f_i, 则通过邻接矩阵M_1可以得到节点 i 双亲节点的特征向量, 计算如下:

$$f_i^1 = f_i M_1 \tag{11}$$

同理可以得到节点 i 的其他相关节点的特征表示, 计算如下:

$$\left[f_i^1, \cdots, f_i^7\right] = [f_i M_1, \cdots, f_i M_7] \tag{12}$$

其中, f_i^k表示节点 i 的第 k 种类型相关节点的特征向量. 对根节点来说, 其双亲节点就是其本身, 对于其他非 AST 节点 (填充节点) 来说, 其任何相关节点的特征向量都是它本身.

为了将节点与其相关节点的特征向量结合起来, 本文使用了图卷积神经网络, 公式如下:

$$y_i^{\text{gconv},l} = f\left(W^{\text{gconv},l}\left[f_i^{0,\text{gconv},l-1}, \cdots, f_i^{7,\text{gconv},l-1}\right]\right) \tag{13}$$

其中, $W^{\text{gconv},l}$是图卷积层的权重参数, 卷积核大小 k=3, l表示图卷积层的层数. 特别的是, $f_i^{0,\text{gconv},l-1}$表示当前节点的特征向量, $y_i^{\text{gconv},0}$表示 character-word 注意力机制的输出y_i^{cwa}, f是激活函数 ReLU, 被应用于这些图卷积层中.

综上所述, AST 编码器有 N 个 encoder, 其中每个 encoder 包含 3 个子层, 最后 AST 编码器的输出为$y_1^{\text{ast}}, y_2^{\text{ast}}, \cdots, y_L^{\text{ast}}$.

3.5 解码器

模型最后的模块是解码器, 其作用是通过对 AST 解码器的结果进行解码, 从而来预测函数名. 解码器与 AST 编码器的结构很类似, 由一组 N 个 decoder 组

成, 其中包含多头自注意力层, encoder-decoder 注意力层以及全连接层 3 个子层. 在每个子层都会增加残差连接和归一化层, 帮助模型训练. 其中在训练阶段, decoder 的输入为函数名, 在预测阶段, 输入为空序列 (仅由填充字符组成).

TrGCN 利用上文提到的两个注意力子层将 AST 编码器的输出与 decoder 的输入进行结合. 首先利用多头自注意力机制来提取目标语句的特征向量$y_1^{\text{d-self}},\cdots,y_p^{\text{d-self}}$, 其中 d-self 表示解码器中的多头自注意力子层, P 表示目标语句的最大长度. 然后利用 Encoder-Decoder 注意力机制来学习目标语句和源语句之间的关系. 其中 Encoder-Decoder 注意力层与多头自注意力层结构相同, 输入不同, 只要将$y_1^{\text{d-self}},\cdots,y_p^{\text{d-self}}$当做 Q 向量, 将$y_1^{\text{ast}},y_2^{\text{ast}},\cdots,y_L^{\text{ast}}$当做 K, V 向量即可.

最后使用两层全连接层, 其中第一层全连接层使用激活函数 ReLU, 第二层全连接层用来提取特征, 帮助模型预测结果.

3.6 模型训练

预测函数名的下一个单词是通过使用激活函数 Softmax 对解码器最后一层的输出进行计算, 从而得到所有候选单词的概率, 选择概率最大的单词作为结果.

为了提高语义提取效果, TrGCN 使用了指针网络[25], 该网络可以直接从函数体中复制单词来作为预测函数名时的候选单词. 如图 3 所示, 指针网络可以在预测函数名 getName 时, 从语句 return name 或者形参 String name 中复制单词 name, 进行预测.

```
public String getName(String name) {
    return name;
}
```

图 3 函数样例

指针网络通过学习一个概率来指导模型对函数体中的单词进行复制或者是生成新的单词. 公式如下:

$$P(w)=p_{\text{gen}}P_{\text{vocab}}(w)+(1-p_{\text{gen}})P_c \tag{14}$$

其中, w 表示单词, $P(w)$表示预测单词 w 的概率, $P_{\text{vocab}}(w)$表示在词表中单词 w 的概率, p_{gen}表示生成新的单词的概率, 它是根据解码器最后一层的特征向量进行线性转化, 然后通过 Sigmoid 函数计算得出; P_c表示单词 w 对源语句单词的注意力分布. 从公式中可以看出, 当词表中没有单词 w 时, 即$P_{\text{vocab}}(w)=0$, 该网络可以从函数体中进行单词的复制, 这样可以有效缓解 Out of Vocabulary[25] 问题, 避免未知单词的出现.

在指针网络进行单词复制时, 因为在 AST 中终结符包含程序的语义信息, 非终结符包含程序的结构信息, 根据非终结符不会在函数体中出现, 仅需要从 AST 中的终结符复制的特点, TrGCN 使用了 Children 注意力机制来计算P_c, 公式如下:

$$\beta_t=W_1h^{\text{dec}}W_2y^{\text{ast}} \tag{15}$$

$$\delta_t=\beta_t Mask \tag{16}$$

$$P_c=\frac{\exp\{\delta_t\}}{\sum_{j=1}^{L}\exp\{\delta_j\}} \tag{17}$$

其中, h^{dec}表示 decoder 最后一层的特征向量, y^{ast}是 AST 编码器最后一层的特征向量, Mask 是掩码矩阵, 将β_t中的非终结符置为-INF, 仅计算终结符与目标语句之间的注意力分数. 最后利用 Softmax 来计算每个目标语句中单词的概率.

4 实验评估

本文在 Java 函数命名任务中评估 TrGCN, 通过给定 Java 方法的函数体, 来预测函数名. 先前的研究表明, 这是一个很好的基准任务. 因为其数据集选自 Github 开源的 Java 项目, 其中的函数可以被认为是准确的、相关的, 函数体总体上都是完整的逻辑单元. 同时我们会对函数名进行处理, 将其转化为一组子 token 序列, 例如 getMaxNumber 可以被预测为序列 get max number. 其中目标序列的平均长度为 3.

4.1 数据集

如表 1 所示, 本文使用 3 个不同规模的 Java 数据集来评估模型, Java-small, Java-med, Java-large[6,9].

表 1 数据集统计

数据集	Java-small	Java-med	Java-large
#project-training	10	799	8998
#project-validation	1	100	250
#project-test	1	96	307
#example-training	691 911	3080 595	13 219 981
#example-validation	23 844	418 902	327 437
#example-test	57 088	419 945	424 333

Java-small 包含 12 个大型的 Java 项目, 其中 9 个项目作为训练集, 1 个项目作为验证集, 1 个项目作为测试集. 该数据集包含约 770 k 条样例.

Java-med 包含 1000 个来自 Github 的星标 Java 项目.

其中 799 个项目作为训练集, 100 个项目作为验证集, 96 个项目作为测试集. 该数据集大约有 4 M 条样例.

Java-large 包含 9555 个来自 Github 的星标 Java 项目. 其中 8998 个项目作为训练集, 250 个项目作为验证集, 307 个项目作为测试集. 该数据集大约有 14 M 条样例.

4.2 评估标准

本文采取的评估标准是 Allamanis[6] 以及 Alon[9] 等先前工作使用的评估标准, 对预测结果产生的 sub-tokens 计算 *Precision*、*Recall* 和 *F*1 值, 其中目标语句大小写不敏感. 这个指标的主要思想是一个预测出的函数名的质量依赖于其组成的子单词. 例如, 对于一个函数名 countNumbers, 如果预测出的结果为 numbersCount, 也认为它是精确的匹配; 如果预测结果为 count, 那么它的 *Precision* 为 100%, 但是 *Recall* 值却很低; 如果预测结果为 countRandomNumbers, 那么它的 *Recall* 值为 100%, 但是 *Precision* 值很低. 本文希望使 *Precision* 和 *Recall* 都尽可能的大, 但是这显然有些困难, 所以使用 *F*1 值来衡量这两个标准. 公式如下:

$$Precision = \frac{TP}{TP+FP} \tag{18}$$

$$Recall = \frac{TP}{TP+FN} \tag{19}$$

$$F1 = \frac{2*Precision*Recall}{Precision+Recall} \tag{20}$$

其中, *TP* 为预测正确单词的数量, *FP* 为预测错误单词的数量, *FN* 为没有预测出正确单词的数量.

本文还使用了 ROUGE[26] 评估标准. 该标准主要用于摘要的自动评价, 通过比较机器自动生成的摘要与人工生成的摘要中重叠单元的数量, 来评价机器自动生成摘要的效果. 本文主要使用了 ROUGE-2, ROUGE-*L*. 公式如下:

$$ROUGE-N = \frac{\sum_{S\in\{targetSeq\}}\sum_{gram_N\in S} Count_{\text{match}}(gram_N)}{\sum_{S\in\{targetSeq\}}\sum_{gram_N\in S} Count(gram_N)} \tag{21}$$

其中, 分母统计目标语句中 N-gram 的个数, 而分子统计目标语句与预测出的语句共有 N-gram 的个数, 本文将 *N* 设置为 2 或 *L*, 其中 *L* 代表预测出的语句和目标语句的最长公共子序列.

4.3 实验设置和环境配置

在 TrGCN 模型中, AST 编码器中 encoder 的数量为 N=6, 解码器中 decoder 的数量为 N=6. 词向量的维度和模型维度 $d_{\text{vector}} = d_{\text{model}} = 128$. 隐藏层的维度 $d_{\text{hid}} = 512$. 在模型输入中, 本文尝试使用参数学习的加权方式来融合 character 和 word 特征向量, 其公式为 $n_i=(1-g)n_i+gn_i^c$, $g = \left(v^{\mathrm{T}}n_i + b\right)$. 其中 v^{T} 为权值参数, b 为偏差值. 其中实验得到的 *F*1 值分别为 53.46%、53.34%, 没有显著差异 (差异可能由于实验的随机性导致). 推测原因为 TrGCN 所使用的 character-word 注意力机制的本质也是将 word 和 character 特征向量通过加权的方式融合, 而且在 AST 编码器中 N (N=6) 个 encoder 都会使用 character-word 注意力机制, 所以多一次加权融合对模型学习能力没有明显的影响. 为了减少参数量, 加快实验运行速度, 所以本文在模型输入阶段选择了直接加和的方式对 character 和 word 特征向量进行融合. 在多头自注意力机制中, 原文将头的数量设置为 $H = 8$, 为了探究头的数量对模型的影响, 分别设置 $H = 2,4,8$ 进行实验. 当 $H = 2$ 时, *F*1 值为 51.25%; 当 $H = 4$ 时, *F*1 值为 53.46%; 当 $H = 8$ 时, *F*1 为 52.45%. 可以看出 *F*1 值随着 H 的增大而提高. 但是当 $H = 8$ 时, *F*1 值有一定幅度的下降, 此时模型出现了过拟合现象. 随着 H 数量的增加模型的非线性学习能力逐渐变强, 但是数量增加到一定值后非线性学习能力太强导致出现过拟合的现象, 所以本文将头的数量设置为 4.

TrGCN 在每一层后使用了 dropout, 来防止过拟合现象 (其中包括注意力层, 卷积层以及全连接层), 其中 $dropout = 0.3$. 实验中每次选取的样本数量为 $batch\_size = 256$, 迭代次数 $epoch = 100$, 为了防止过拟合, 使用了早停法, 只要连续 10 次迭代后 *F*1 值没有比当前最大 *F*1 值高, 就停止训练, $early\_stopping = 10$. 为了防止模型出现梯度爆炸, TrGCN 把最终所有候选词的概率截断在 [1e–10,1.0] 之间. 对于模型优化, 使用了 Adam 优化器[27], 参数为默认值.

所有实验都运行在 Linux (系统版本 3.10.0-957.c17.x86-64, CPU Inter(R) Gold 6240@260 GHz 18 cores, 两块显存为 12 GB 的 (NVIDIA)TITAN Xp 显卡) 系统下.

4.4 实验结果和分析

在实验中本文使用了 6 个基准模型:

(1) Allamanis 等提出 ConvAttention 模型[6], 该

模型使用了带有注意力机制的卷积神经网络来预测函数名; (2) Alon 等提出 Path+CRFs[7] 模型, 将句法路径与条件随机场相结合; (3) Alon 等提出的 code2vec[8] 模型; (4) Alon 等提出的 code2seq[9] 模型; (5) Tai 等提出的 TreeLSTM 模型; (6) Vaswani 提出的 Transformer[12] 模型.

表 2 显示了函数命名任务在 Java 数据集中的结果. 从表中可以看出, TrGCN 在评估标准 *Precision*、*Recall* 和 *F*1 以及 3 个数据集中, 几乎都优于所有基准模型. 与 ConvAttention 对比, TrGCN 在 *F*1 值提升约 20.41% 至 21.8%, 其原因可能为 TrGCN 有效的提取了程序的结构信息. 同样在和 TreeLSTM 对比中发现, TrGCN 在 *F*1 提升约 6.12% 至 23.49%. 虽然 TreeLSTM[28] 也提取了程序的结构信息, 但是我们认为基于图的神经网络模型可以丰富结构信息, 具有更强的建模能力. 在与当前使用广泛模型 code2seq[9] 比较中发现, 我们的模型在 *F*1 值有着明显提高, 在数据集 Java-small 提升了 10.44%, 在数据集 Java-med 提升了 4.7%.

表 2 在 *Precision*, *Recall*, *F*1 以及 Java-small, Java-med, Java-large 三个数据集与基准模型比较结果

模型	Java-small			Java-med			Java-large		
	Prec	*Rec*	*F*1	*Prec*	*Rec*	*F*1	*Prec*	*Rec*	*F*1
ConvAttention[6]	50.25	24.62	33.05	60.82	26.75	37.16	60.71	27.60	37.95
Paths+CRFs[7]	8.39	5.63	6.74	32.56	20.37	25.06	32.56	20.37	25.06
code2vec[8]	18.51	18.74	18.62	38.12	28.31	32.49	48.15	38.40	42.73
TreeLSTM[28]	40.02	31.84	35.97	53.07	41.69	46.69	60.34	48.27	53.63
Transformer[12]	38.13	26.70	31.41	50.11	35.01	41.22	59.13	40.58	48.13
code2seq[9]	50.64	37.40	43.02	61.24	47.07	53.23	64.03	55.02	59.19
TrGCN	55.53	51.53	53.46	61.38	54.85	57.93	62.46	56.23	59.75
结果对比	**+4.89**	**+14.13**	**+10.44**	**+0.14**	**+7.78**	**+4.7**	**–1.57**	**+1.21**	**+0.56**

其原因可能不仅是图卷积神经网络提取到更加丰富的结构信息, 而且 character-word 注意力机制以及指针网络可以有效的提升语义提取的效果. 但是我们发现随着数据规模的增大, 我们模型提升效果逐渐减小; 尤其在数据集 Java-large 中, *Precision* 略低于 code2seq 模型, *F*1 仅提高了 0.56%. 可能的原因有: (1) 随着数据规模的增大, 但是模型复杂度不变, 模型很难进一步从庞大的数据中学习; (2) 基于图的神经网络模型虽然有很强的建模能力, 但是也具有模型复杂、较难训练的特点, 其中在 3 个数据集训练过程中, 迭代一次分别需要 40 分钟、2 小时和 4 小时; (3) 实验资源短缺, 实验室仅有两块 12 G 显存的 (NVIDIA)TITAN Xp 显卡, 随着数据量的增大, 内存以及显存无法满足训练要求, 所以只能通过减小 batch_size 来克服资源问题, 但是 batch_size 的大小一定程度会影响实验结果.

同时, 我们选择了一个基于图神经网络的基准模型, Fernandes 等[10] 提出的 Sequence-GNNs 模型, 该模型使用图神经网络来对程序进行编码. 表 3 说明我们的模型在评估标准 *F*1, ROUGE-2 以及 ROUGE-L 都优于模型 Sequence-GNNs. 在 code2seq 文章中提到, Sequence-GNNs 在数据集 Java-large 上的 *F*1 值低于 code2seq, 所以我们只使用了数据集 Java-small 进行比较.

表 3 与 Sequence-GNNs 在数据集 Java-small 的比较

模型	*F*1	ROUGE-2	ROUGE-L
Sequence-GNNs	51.4	25.0	50.0
TrGCN	53.5	25.7	53.6
结果对比	**+2.1**	**+0.7**	**+3.6**

从表 3 中可以看出 TrGCN 在 *F*1 值比 Sequence-GNNs 提升了 2.1%, 说明 TrGCN 在不考虑预测结果单词顺序的情况下有较强的预测能力; 在 ROUGE-2 和 ROUGE-L 分别提高了 0.7%、3.6%, 可以看出 TrGCN 在考虑预测结果单词顺序的情况下同样优于 Sequence-GNNs. 所以 TrGCN 模型在融合 Transformer、图卷积以及 character-word 注意力机制后的预测能力的提高是有效的.

5 结论与展望

本文提出了一个 TrGCN 模型来解决函数自动化命名任务. TrGCN 利用 Transformer 中的注意力机制来缓解模型训练时遇到的长程依赖问题; 利用图卷积神经网络提取更为丰富的结构信息, 使得程序的特征向量信息更加完整; 利用 character-word 注意力机制以及

指针网络来丰富程序的语义信息. 实验结果表明, TrGCN 的函数命名效果优于其他基准模型. 在后续的研究中, 作者将使用不同的编程语言数据集, 以及不同的代码摘要任务来验证方法的有效性.

参考文献

1 Høst EW, Østvold BM. Debugging method names. Proceedings of the 23rd European Conference on Object-Oriented Programming. Genoa, Italy. 2009. 294–317.

2 Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks. Proceedings of the 27th International Conference on Neural Information Processing Systems. Montreal, QC, Canada. 2014. 3104–3112.

3 Cho K, Van Merriënboer B, Gulcehre C, *et al*. Learning phrase representations using RNN encoder-decoder for statistical machine translation. arXiv: 1406.1078, 2014.

4 Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate. arXiv: 1409.0473, 2014.

5 Luong MT, Pham H, Manning CD. Effective approaches to attention-based neural machine translation. arXiv: 1508.04025, 2015.

6 Allamanis M, Peng H, Sutton C. A convolutional attention network for extreme summarization of source code. Proceedings of the 33rd International Conference on Machine Learning. New York City, NY, USA. 2016. 2091–2100.

7 Alon U, Zilberstein M, Levy O, *et al*. A general path-based representation for predicting program properties. ACM SIGPLAN Notices, 2018, 53(4): 404–419. [doi: 10.1145/3296979.3192412]

8 Alon U, Zilberstein M, Levy O, *et al*. Code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 2019, 3(POPL): 40.

9 Alon U, Brody S, Levy O, *et al*. Code2seq: Generating sequences from structured representations of code. arXiv: 1808.01400, 2018.

10 Fernandes P, Allamanis M, Brockschmidt M. Structured neural summarization. arXiv: 1811.01824, 2018.

11 Bengio Y, Simard P, Frasconi P. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 1994, 5(2): 157–166. [doi: 10.1109/72.279181]

12 Vaswani A, Shazeer N, Parmar N, *et al*. Attention is all you need. Proceedings of the 31st International Conference on Neural Information Processing Systems. Long Beach, CA, USA. 2017. 5998–6008.

13 Niepert M, Ahmed M, Kutzkov K. Learning convolutional neural networks for graphs. Proceedings of the 33rd International Conference on International Conference on Machine Learning. New York City, NY, USA. 2016. 2014–2023.

14 Allamanis M, Barr ET, Bird C, *et al*. Suggesting accurate method and class names. Proceedings of the 10th Joint Meeting on Foundations of Software Engineering. Bergamo, Italy. 2015. 38–49.

15 Iyer S, Konstas I, Cheung A, *et al*. Summarizing source code using a neural attention model. Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Berlin, Germany. 2016. 2073–2083.

16 Hu X, Li G, Xia X, *et al*. Summarizing source code with transferred api knowledge. Proceedings of the 27th International Joint Conference on Artificial Intelligence. Stockholm, Sweden. 2018. 2269–2275.

17 Li J, Wang Y, Lyu MR, *et al*. Code completion with neural attention and pointer networks. arXiv: 1711.09573, 2017.

18 Liu C, Wang X, Shin R, *et al*. Neural code completion. Proceedings of International Conference on Learning Representations 2017 Conference Submission. Toulon, France. 2017. 1–14.

19 Sun ZY, Zhu QH, Xiong YF, *et al*. TreeGen: A tree-based transformer architecture for code generation. The 34th AAAI Conference on Artificial Intelligence (AAAI 2020), the 32nd Innovative Applications of Artificial Intelligence Conference (IAAI 2020), the 10th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI 2020). New York, NY, USA. 2020. 8984–8991.

20 Sun ZY, Zhu QH, Mou LL, *et al*. A grammar-based structural CNN decoder for code generation. Proceedings of the 33rd AAAI Conference on Artificial Intelligence (AAAI 2019), the Thirty-First Innovative Applications of Artificial Intelligence Conference (IAAI 2019), the Ninth AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI 2019). Honolulu, HI, USA. 2019. 7055–7062.

21 Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. arXiv:1711.00740, 2017.

22 Beck D, Haffari G, Cohn T. Graph-to-sequence learning using gated graph neural networks. Proceedings of the 56th Annual Meeting of the Association for Computational

Linguistics. 2018. 273–283.

23 Ba JL, Kiros JR, Hinton GE. Layer normalization. arXiv: 1607.06450, 2016.

24 He KM, Zhang XY, Ren SQ, *et al*. Deep residual learning for image recognition. Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition. Las Vegas, NV, USA. 2016. 770–778.

25 See A, Liu PJ, Manning CD. Get to the point: Summarization with pointer-generator networks. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics. Vanconver, BC, Canada. 2017. 1073–1083.

26 Lin CY. Rouge: A package for automatic evaluation of summaries. Proceedings of 2004 Workshop on Text Summarization Branches Out. Barcelona, Spain. 2004. 74–81.

27 Kingma DP, Ba J. Adam: A method for stochastic optimization. Proceedings of the 3rd International Conference on Learning Representations. San Diego, CA, USA. 2015.

28 Tai KS, Socher R, Manning CD. Improved semantic representations from tree-structured long short-term memory networks. Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing. Beijing, China. 2015. 1556–1566.