ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2000/00(00)-0000-00
doi: 10.3778/j.issn.1673-9418.

E-mail: fcst@vip.163.com
http://www.ceaj.org
Tel: +86-10-89056056

一种基于层次 LSTM 的程序自动补全方法

刘芳，李戈+，金芝+
北京大学 信息科学技术学院软件研究所高可信软件教育部重点实验室，北京 100871

An Automatic Program Completion Method Based on Stacked LSTM

Liu Fang, Li Ge+, Jin Zhi+
Key Lab of High Confidence Software Technologies (Peking University), Ministry of Education
+ Corresponding author: E-mail: lige@pku.edu.cn, zhijin@pku.edu.cn

Abstract: Automatic program completion problems can be considered as a predictive problem. That is to predict the next most likely token based on the current input token. The Long Short-Term Memory neural network in deep learning is dedicated to dealing with sequence prediction problems. This paper explores the use of deep learning to learn the automatic program completion based on the recurrent neural network. Since the programming language is different from other data, it is a structured text. If we take it as a linear input, the result of the program completion will be poor due to the loss of the structure information of the code. Thus, this paper uses the abstract syntax tree as auxiliary tool to transform the program into serialized data containing the structural information. Besides, the Long Short-Term Memory neural network has a lot of limitations when dealing with structured data. It can be improved by adding memory units to the network, so that it can learn the structure information of the program. This paper proposes and implements an automatic program completion model based on stacked LSTM——StackLSTM. Experiment results show that StackLSTM model has a very strong learning ability, which can learn the semantics and structural information of the program only in a small data set. So it can accelerate training process. What's more, it can also achieve better results in predicting nonterminal and terminator than the traditional LSTM network model in large data set.
Key words: Automatic program completion; LSTM; Program structure; Deep learning

摘 要：程序自动补全问题可以视为序列预测问题，即基于已输入的代码预测下一个可能出现的代码片段。深度学习中的长短期记忆神经网络擅长于处理序列预测问题，因此基于长短期记忆神经网络对程序自动补全进行研究。不同于其他序列化数据，代码是一种包含结构信息的文本数据，若直接将代码视为序列化数据进行学习，则会因为丢失代码中包含的结构信息而使得程序补全效果较差。为此，可利用抽象语法

树作为辅助工具，将代码转化为包含完整结构信息的序列化数据。另外传统的长短期记忆神经网络在处理结构化的数据时具有一定的局限性，因此可通过在网络中增设记忆单元的方式对其进行改进，以学习输入数据中结构信息。提出并实现了一种基于层次 LSTM 的程序自动补全模型——StackLSTM 模型，实验结果表明，StackLSTM 模型具有十分强大的学习能力，在小数据集中就可以很好地学会程序的语义以及结构信息，可以有效加速训练；它在大数据集中也能取得更好的效果，预测非终结符和终结符的准确率均高于传统的长短期记忆神经网络模型。

关键词： 程序自动补全；长短期记忆神经网络；程序结构；深度学习
文献标志码：A 中图分类号：TP312

1 引言

随着互联网的发展，各个传统行业都在与互联网进行深度融合以追求新的生机，这就创造了大量的对程序的需求。近几年来，深度学习技术飞速发展。在海量数据的支持下，基于深度学习的人工智能在语音识别、图像识别、自动驾驶、自然语言处理等领域取得了巨大的进步，已经能够在一定程度上代替人类从事脑力劳动。如华尔街领军的金融集团之一摩根大通就开发了一款金融合同解析系统 COIN,它只需几秒就能完成原先需要人工花费 36 万小时才能完成的工作。未来，人工智能会逐渐与各行各业相结合，由它来代替人类完成更多的工作，即使是复杂的编程工作也有可能由人工智能代替完成[1,2,3]。

在软件工程的发展过程中，计算机科学家们为了降低软件开发的强度，缩短开发周期，一直以来都致力于程序自动补全方面的研究。通常为完成一段具备特定功能的程序，程序员会基于已有的编程知识，逐词逐句地编写代码，事实上这种编程模式是可以模仿的。

可以将程序补全问题视为文本序列预测问题，即基于已输入的代码预测下一个可能出现的代码片段。而这与自然语言处理中的文本序列预测问题是相似的，因此本文基于该思想并结合深度学习技术对程序自动补全问题进行研究。

2 相关工作

当前主流的程序补全方法大多基于程序语言概率模型实现[4,5,6,7]。构建程序语言概率模型主要有以下几种方法[8]：

（1）N-gram 模型：N-gram 模型是最流行的程序语言概率模型，因为它十分简单并且学习效率高（Hindle 等人第一次将该模型用于程序语言的学习[9]）。该模型最初用于自然语言，它基于这样一种假设：第 N 个词的出现只与前面的 N-1 个词相关，而与其它任何词都不相关。因此可以用前 N-1 个连续的单词来预测下一个单词出现的概率。它可以轻松地扩展到代码补全的问题中，只要将每个 token 看做是一个单词，然后在代码库中学习相应的概率矩阵。然而，对于在所使用的代码库中未出现的 token，该模型却无法较好地处理。并且，仅仅根据前 N-1 个 token 来预测下一个 token 的方式本身就具有局限性，实际程序中，预测下一个代码片段所依赖的上文代码长度是无法确定的。

（2）词向量（Word2vec）神经网络语言模型：Bengio 等人首次提出用于构建语言模型的单词的向量特征空间[10]，该方法显著优于传统的基于 n-gram 的方法，将每个单词表示为向量的形式。该方法基于深度学习的思想，通过训练，一步步地学习词语中的特征，将词语映射为向量，该向量是词语的特征表示。将文本内容转化为向量空间中的向量，从而进行计算。这些向量的语义相似度可以通过向量空间上的相似度来表示。T. Mikolov 等人证明了该模型可以捕捉到词对之间的关系（例如：vking-vqueen=vman-vwoman）。之后，Mikolov 等人提出了 CBOW 模型和 skip-gram 模型[11]， CBOW 模型是由上下文预测当前词语出现的概率，而 skip-gram 模型则正好相反，是由当前词语预测上下文 token 的概率。Word2vec 输出的词向量可以被用来做很多自然语言处理的相关的工作，比如聚类、找同义词、词性分析等等，同样也可用于处理程序语言。

（3）概率语法：上下文无关文法（CFG）可以较好地解析程序，但是在评估程序的正确性方面却比较差。因此，许多可以被成功解析的程序却并不正确，不符合用户需求。因此，提出基于概率的上下文无关文法(PCFG)来处理程序补全问题。概率上下文无

关文法遵循上下文无关文法中的每一条生成规则，同时还可以学习训练数据中每条规则出现的概率。然而，仅仅考虑概率因素是不够的，使用哪一条规则补全程序还与当前程序包含的信息密切相关，因此该方法效果并非十分理想。

Bielik 等人提出的代码概率模型[8]结合了 n-gram 模型与概率语法模型的优势，至今为止在 javascript 代码补全问题上取得了最好的性能。

以上基于程序语言概率模型的程序补全方法具有一定的局限性，这些方法在补全下一个词语时可利用的上下文信息均有限。与自然语言处理中的文本序列预测问题类似，程序补全基于已输入的代码预测下一个可能出现的代码片段。深度学习技术中的循环神经网络（RNN，Recurrent Neural Networks）恰好擅长处理上述序列预测问题，Mikolov 等人在 2010 年提出了基于循环神经网络的语言模型[12]。循环神经网络的最大优势在于，它可以充分地利用所有上文信息来预测下一个词，而不像之前其它工作那样，只能开一个 n 个词的窗口，只用前 n 个词来预测下一个词。但由于 RNN 存在着梯度消失的问题，而长短期记忆神经网络（LSTM, Long Short-Term Memory，一种特殊的 RNN 网络结构）[13]则可以有效地解决该问题，因此本文在 LSTM 模型的基础上做进一步研究。

3 程序自动补全模型的设计

程序自动补全模型的研究涉及两个关键问题：首先，代码数据不同于其他文本数据，它具有复杂的结构，若直接将代码视作序列化的数据，便会丢失代码中包含的结构信息。其次，传统的长短期记忆神经网络模型缺乏学习结构化信息的能力。本章将着手解决上述两个问题。

3.1 保留程序结构信息

程序具有复杂的结构性。一个类的定义、一个函数的定义或一个赋值语句等均可视作一个完整的代码段，而各个代码段之间可相互嵌套，每个代码段内部的变量又拥有各自的作用域，程序复杂的结构正源于此。若直接将代码视作序列化的文本数据，代码中包含的程序结构信息就会全部丢失，循环神经网络也难以从这种缺乏关键信息的数据中学会编程。

为了方便神经网络学习程序的语法规则及结构，需要用一种合适的形式来表示程序。近年来，抽象语法树被广泛应用于程序分析中，它既可以表示程序的语义，同时也可以表示程序的结构，因此本文采用抽象语法树对程序进行解析，最终将程序转换为便于神经网络学习的序列化形式。

3.1.1 程序转换为 AST

抽象语法树(AST, Abstract Syntax Tree)是一种程序的抽象表示形式，它可将源代码抽象成一种由语法结构构成的树形表示，该树的每一个节点(及其子节点)就对应源代码中的某个程序片段。因此，抽象语法树具有以下优越的特性：

(1) 首先它会过滤掉源代码中的注释等无用信息；

(2) 其次它的树形结构能清晰地表达源代码的层次嵌套关系。

通常来说，python 代码的结构信息体现在代码的缩进，一般的方法难以在保持代码结构的前提下将其转化为序列化格式。但是利用抽象语法树的特性，我们可以将 python 代码的结构信息存储在语法树的树形结构中，以便后续处理。

以一句简单的 python 代码为例：

```
1. while i<5:
2.     print(i)
```

上述代码可转化为图 1 所示的抽象语法树。

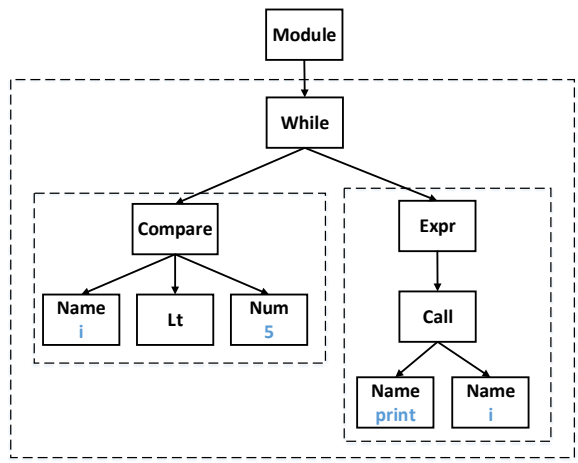

Fig.1 python syntax tree instance of while loop

图 1 while 循环 python 语法树实例

在这段源代码中，while 循环中的控制条件与“while”通过空格隔离开来，然后通过下一行的缩进来表示循环体。当该源代码转换为抽象语法树之后，while 循环的结构完整地存储在了树中，树

每一个节点及其子树都对应于代码中的一个结构。通过将源代码表示为抽象语法树，可以在过滤掉无用信息的同时完整地保留程序中的结构信息。

3.1.2 AST 序列化

上文已将 Python 源代码解析为抽象语法树这种中间表示，现需进一步将该表示转化为序列化数据，供神经网络学习。通常可按深度优先搜索的策略对语法树进行遍历，从而将语法树转换为线性序列，但是直接对抽象语法树进行遍历则会丢失语法树的结构信息，而语法树的结构信息对于我们来说是至关重要的。为了能够在遍历过程中保留语法树的结构信息，本文在遍历过程中，采用“{”和“}”将语法树的每个中间节点包围起来，并且在前面注明该节点类型，即每一个代码段开始于“{”，结束于“}”，这样就可以保留 AST 中的结构。图 1 中的 while 循环经过上述序列化处理之后就会变为：

Module { While { Compare { Name('i') Lt Num } Expr { Call { Name('print') Name('i') } } } }

从上述表示形式可看出，程序中的每一个代码段都用“{”与“}”所包围，并在大括号前面注明了代码段名称。例如 while 的循环控制条件(i<5)由 Compare 后面的大括号所包围，而循环体 print 语句则由 Expr 后的大括号包围；上述两个代码段又属于 While 这个大结构，所以被包围在 While 后面的大括号中。

上述序列与 AST 的对应关系如图 2，通过这种方式可将抽象语法树转换为可供神经网络学习的、包含程序结构信息的线性序列。

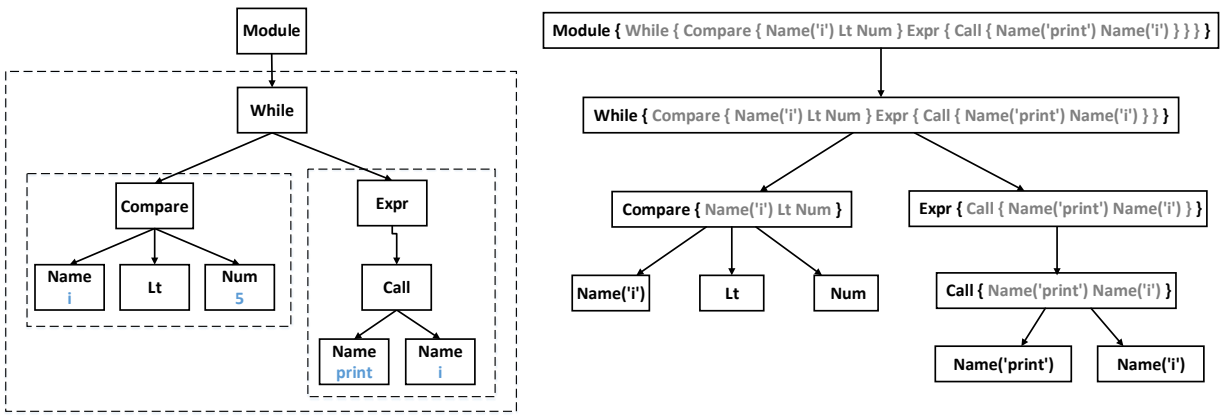

Fig.2 The relationship between serialized AST and the original AST

图 2 序列化处理后的 AST 与原 AST 对应关系

3.2 模型设计

上文已介绍如何在保留结构信息的前提下，将 python 代码转换为序列化数据供神经网络使用。

注意到 LSTM 通常处理的都是文本、语音等不包含结构信息的简单序列化数据，它缺少学习数据中的结构信息的能力。本节将在标准 LSTM 的基础上，通过在网络中增设记忆单元的方式，设计能够学习数据中结构信息的模型。标准 LSTM 模型和改进模型均采用 3.1 节中方法对输入代码进行了处理，将保留了程序结构的序列化数据作为训练集。

3.2.1 标准 LSTM 模型

标准的 LSTM 模型结构如图 3 所示：

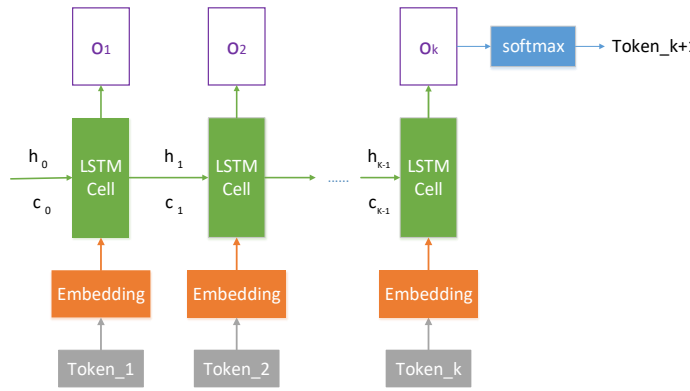

Fig.3 Standard LSTM model structure diagram

图 3 标准 LSTM 模型结构图

该模型的输入为 k 个 token(*Token_1, To-*

ken_2,…,Token_k)，然后通过 embedding 层将 token 转换为特征向量，输入隐藏层的 LSTM Cell，该层输入除了当前程序 token 的 embedding 之外还包括上一时刻隐藏层状态 h_{t-1} 以及细胞状态 c_{t-1}；该层输出为当前时刻隐藏层状态 h_t 以及细胞状态 c_t。将隐藏层输出 o 输入模型的输出层 softmax 层，最后得到预测结果的概率分布向量，向量中第 i 个元素的值代表结果为词汇表中第 i 个 token 的概率。下面分别对模型的每一层进行详细介绍。

(1) Embedding 层

由于神经网络处理的单位为向量，因此必须将输入的每一个 token 转换为向量形式才能输入 LSTM 中，同时，该向量表示输入 token 的特征。由于程序 token 总数较大，如果直接采用 one-hot 编码则会造成巨大的输入空间，不但会占用大量内存，也会使得计算速度十分缓慢。因此，本文使用 embedding 的主要目的有以下两个：

1) 降维

2) 提取输入特征

本文先将输入 token 转换为 id（整型），然后采用 Word2vec 的方法将 id 映射成 embedding。在 LSTM 神经网络初始化时随机生成输入的 embedding，然后在整个网络训练过程中不断更新 embedding，逐步学习 token 中的特征。embedding 的大小为：[*vocab_size*, *size*]，其中 *vocab_size* 为词汇表大小，也就是 token 的数目，*size* 为隐藏层大小。模型训练结束之后的 embedding 可以有效地表示 token 中的特征。

(2) 隐藏层

隐藏层为 LSTM 的核心部分，该层输入为：当前程序 token 的 embedding：x_t，上一时刻隐藏层状态：h_{t-1} 以及细胞状态：c_{t-1}；输出为当前时刻隐藏层状态：h_t 以及细胞状态：c_t。

H 与 c 的计算公式如下：

$$\begin{pmatrix} q \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} P_{J,2J} \begin{pmatrix} x_i \\ h_{i-1} \end{pmatrix}$$
$$c_i = f \square\ c_{i-1} + q \square\ g \qquad (1)$$
$$h_i = o \square\ \tanh(c_i)$$

(3) Softmax 层

LSTM 的输出层采用 softmax 层，该层的输入为隐藏层的输出 h，该层的输出是大小为 *vocab_size* 的向量，对应于预测结果 token 的概率分布。向量中第 i 个元素的值代表结果为词汇表中第 i 个 token 的概率，每个元素大小都处于 0-1 之间。

计算公式如下：

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for j=1, ... , K} \qquad (2)$$

3.2.2 StackLSTM 模型

尽管 LSTM 在计算机视觉、语音识别等领域取得了巨大成功，但是仍有许多问题是 LSTM 所无法解决或者很难解决的，就以本文中的程序语言为例，它包含复杂的结构，LSTM 并不擅长处理此类具有结构化的数据。

为了解决此类问题，研究人员提出了许多解决方案。其中，一种比较有效的解决方案为：通过增加记忆单元，减轻 LSTM 的记忆负担，并且学会数据中存在的结构，从而增强 LSTM 的学习能力[2,14,15]。Scott Reed 等人在《Neural Programmer-Interpreters》这篇文章中通过将隐藏层状态以及程序和相关参数通过栈进行保存恢复来减轻神经网络的记忆负担，增强了模型的学习与泛化能力[2]。本文受到了该思想的启发，在 LSTM 网络中增设记忆单元来学习程序语言中包含的结构信息。据调查，目前尚未有人将该方法用于学习现实中的程序语言（目前已知的最新的 NPI[2]、RNPI[3]中生成的是 DSL 语言）补全相关问题。

在本文模型中，代码段的相互嵌套导致了的输入数据中包含了大量的结构信息，使用栈作为记忆单元，则可以模拟代码段的嵌套关系。

循环神经网络通过隐藏层状态来对上文信息进行记忆，该状态所记录的信息对预测起决定性作用。在传统的 LSTM 中，每一个输出的预测都根据隐藏层中记录的全部输入信息产生。这就是它的问题所在，在预测下一个 token 时，并非依赖于上文全部程序，例如图 4 中在预测 add 函数的调用时，隐藏层中记录的 add 函数的实现细节对于预测没有任何帮助，反而会因为额外的记忆负担而影响预测准确率。

```
01. def add(a,b):
02.     # something else
03.     sum=a+b
04.     # something else
05.     return sum
06.
07. c=add(1,2)
```

Fig.4 code example

图 4 代码示例

若能让隐藏层状态只保留上文程序中的关键信息，就能有效解决上述问题。因此，在 LSTM 网络中增设栈作为记忆单元使其能够具备学习数据中结构信息的能力。网络在学习过程中依次读入输入数据中的每一个 token，读入之后对输入值进行判断：若为代码段开始标记“{”，则表示进入一个新的代码段，此时将当前隐藏层状态保存到栈中，然后对该代码段内部的相关代码进行学习；当输入为代码段结束标记“}”时，将隐藏层状态恢复至进入该代码段时的状态，此时，代码段内部的实现细节并未保存在隐藏层状态中，因此不会对后文代码预测造成影响，也极大地减轻了 LSTM 的记忆负担。

由于神经网络的学习是通过不断调整网络中的各个权重进行的，尽管清空了上文代码段内部代码的相关记忆，但是网络在进入代码段内部学习时仍在更新权重，因此代码段内部的实现逻辑全部被神经网络学习记录。并且在学习过程中权重自始至终都是共享的，这就保证了网络充分利用所有的代码进行学习，隐藏层状态却只保留对关键信息的记忆，以此为依据对下一个 token 进行预测。

综上所述，整个模型不但可以学习代码段内部的相关逻辑，也可以学习代码段之间的关系即程序的结构信息。

模型结构如图 5（图中省略了输出层部分，该部分与标准 LSTM 相同）所示：

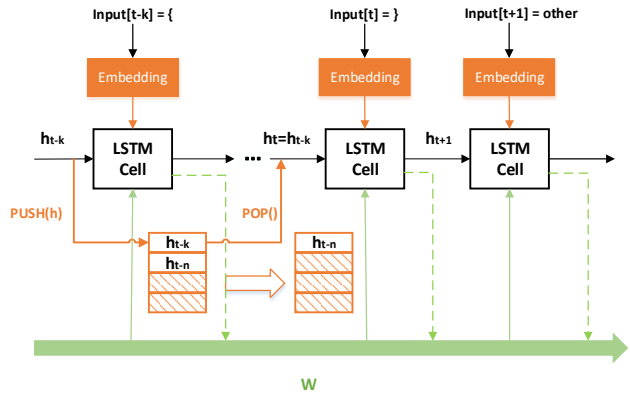

Fig.5 StackLSTM model structure diagram

图 5 StackLSTM 模型结构图

进入代码段时隐藏层状态保存在栈中之后，面临两种选择：是否将隐藏层状态置零。置零则意味着将上文的记忆全部清空，只关注当前代码段，不受任何上文的影响；而不置零则意味着上文中的关键信息还存在隐藏层状态中，这些相关信息会对下一个 token 的预测造成影响。本文对这两种方式都进行了相关实验，分别对应于下文的置零模型和不置零模型，通过实验对比两种模型的效果，得出相关结论。

(1) 置零模型

根据上文描述，算法伪代码如图 6：

算法 1 置零模型

输入: *TokenEmbeddings*

```
function STEP(input, h, numsteps)
    for i = 1 to numsteps do
        if input[i] = StartMarker then                ▷ StartMarker为代码段开始的标记
            PUSH(h)                                   ▷ 保存h到stack中
            h ← 0                                     ▷ 隐藏层状态h置零
            (output, h) ← LSTMCELL(input[i − 1], h)   ▷ Feed-forward LSTM, 获取当前代码段类型
        else if input[i] = EndMarker then             ▷ EndMarker为代码段结束的标记
            h ← POP()                                 ▷ 恢复h
        end if
        (output, h) ← LSTMCELL(input[i], h)           ▷ Feed-forward LSTM，更新h并得到输出
    end for
end function
```

Fig.6 reset model algorithm pseudocode

图 6 置零模型算法伪代码

该伪代码描述的是置零模型处理一个 batch 时的算法，第 2 行中 for 循环每次读入代码中的一个 token，一共分为以下几种情况：

(a) 当前 token 为代码段开始的标记（第 3 行），数据集中用“{”表示，则将隐藏层状态压栈（第 4 行），然后将隐藏层状态置零（第 5 行），这里需要注意，置零之后 LSTM 将会完全忘记前面的信息，由于代码段的名称是标注在“{”之前的，因此此时 LSTM 无法确定当前属于哪个代码段，无法准确预测代码段中第一个 token，所以这里进行了相关处理，在第 6 行中重新对当前 token 的上一个 token（也就是代码段名称）进行一次 LSTM 的前向转播，此时隐藏层就记录了代码段的相关信息，因此就可以依据此信息预测代码段中第一个 token。

(b) 当前 token 为代码段结束的标记（第 7 行），数据集中用“}”表示，则将隐藏层状态恢复至压栈前的状态。

(c) 除了上两种情况，其余 token 则不用对隐藏层状态作额外处理。

最后再进行 LSTM 的前向传播（第 10 行），更新状态 h 并得到对应的输出。

综上所述，置零模型中，每当进入一个新的代码段时将会丢弃上文代码的全部信息，只保留当前代码段的信息，然后对代码段中的内容进行学习。当代码段结束时，将隐藏层状态恢复至进入该代码段时的状态，将代码段内部的相关记忆清空。

(2) 不置零模型

不置零模型的算法如图 7：

算法 2 不置零模型

输入: *TokenEmbeddings*

```
1: function STEP(input, h, numsteps)
2:     for i = 1 to numsteps do
3:         if input[i] = StartMarker then          ▷ StartMarker为代码段开始的标记
4:             PUSH(h)                             ▷ 保存h到stack中
5:         else if input[i] = EndMarker then       ▷ EndMarker为代码段结束的标记
6:             h ← POP()                           ▷ 恢复h
7:         end if
8:         (output, h) ← LSTMCELL(input[i], h)     ▷ Feed-forward LSTM，更新h并得到输出
9:     end for
10: end function
```

Fig.7 unreset model algorithm pseudocode

图 7 不置零模型算法伪代码

在不置零模型中，当进入一个新代码段时，不会忘记之前的相关信息，仅仅是保存当前状态便于代码段结束时恢复，这也是与置零模型的唯一区别。因此，在预测下一个代码片段时，隐藏层状态中记录了上文代码中的关键信息。

StackLSTM 模型中除了隐藏层状态的变化，模型剩余部分均与 3.2.1 中的标准 LSTM 相同，包括 embedding、softmax 层、隐藏层个数，损失函数等等。

4 实验结果与分析

本文模型基于 Ubuntu 16.04.1 平台，使用 tensorflow 1.0 以及 python3.0 实现。

4.1 数据

本文采用 python 代码作为数据集，数据源为 github 开源社区中 star 数大于 5 的 python 项目，完成数据预处理之后保留了 3.8 万个 python 代码文件。

4.2 模型训练细节

本文中三个模型（标准 LSTM、不置零、置零）中均使用 2 层 LSTM 结构，隐藏层大小为 200。词汇表大小为 5000，因此转换为 embedding 大小为[5000, 200]。初始学习率为 1.0，从第五周期开始，以 0.5 的衰减率调整学习率，共训练 13 个周期。

理论上 LSTM 可以利用的上文长度无大小限制，但是实际应用中需要根据使用场景规定其最大依赖上文长度 *numsteps*，每一个迭代周期会依次接收 *numsteps* 个输入，每个输入都会经过 LSTM 网络得

到对应的输出。由于训练数据为独立的 python 代码文件，每个代码文件中的 token 预测只与当前代码文件中该 token 之前的代码相关，因此本文模型在每个迭代周期开始时将隐藏层状态置零，消除对其他代码文件的记忆，并且将 *numsteps* 大小设为代码的长度，即预测 token 时最大依赖上文长度为该 token 所属代码文件的长度。但是，由于训练集中每一个程序的长度各不相同，而模型在训练时 *numsteps* 的大小必须固定。为了解决该问题，本文分别考虑了不同的 *numsteps*，选定 *numsteps* 大小之后，选取数据集中长度不超过 *numsteps* 的程序作为训练数据，其中小于 *numsteps* 的程序末尾用“PAD”填充，因此训练集中每一个程序长度都为 *numsteps*，如图 8 所示：

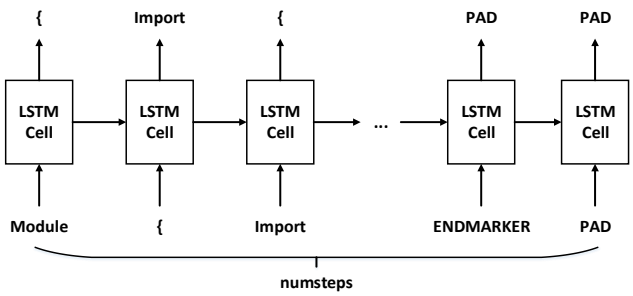

Fig.8 LSTM unrolling diagram

图 8 LSTM 展开图

为了评估不同输入序列长度程序补全的效果，本文分别对 *numsteps* 为 60、100、200、400 的代码进行训练，每种 numsteps 长度均采用 1000 个代码文件进行训练，随着 numsteps 增加则使得训练集中每一个代码文件的长度增加，因此训练时输入-输出对增加，所以训练数据随着 *numsteps* 增加而增加。训练时采用了随机梯度下降法（SGD）进行学习。

4.3 实验结果及分析

本文分别测试了 3 种模型补全终结符（T）和非终结符（NT）的准确率；在抽象语法树中，非终结符对应于中间节点，例如：Assign、If、For、Expr 等，在本文模型中对应于每一个代码段的具体类型，与程序结构紧密相关；终结符则对应于叶子节点，例如：Num、Arg、Str、AttrName，UNK 等，与程序语义密切相关。

本文分别对 *numsteps* 为 60、100、200、400 的代码在三种模型（标准 LSTM、置零、不置零）上进行了训练与测试。由于得到的输出是结果的概率分布，因此分别测试了取概率最高的一个（top1）token 作为结果的准确率以及从概率前三（top3）以及前五（top5）中选择结果的准确率。

4.3.1 非终结符预测

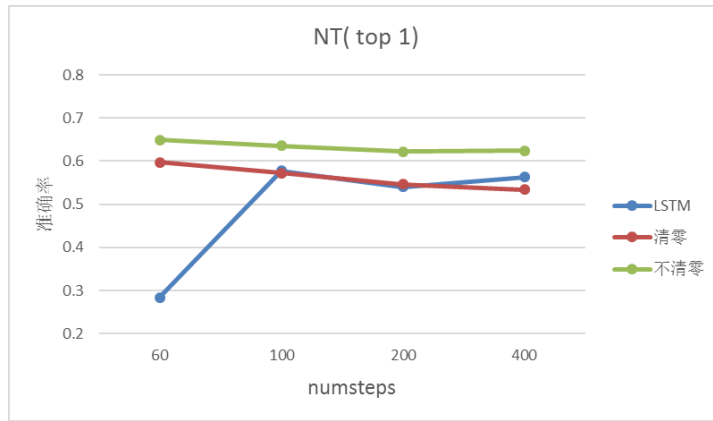

Fig.9 Non-termination(NT) top1 prediction results

图 9 非终结符（NT）top1 预测结果

图 9 是三种模型在不同的代码长度下预测非终结符（NT）取 top1 结果的准确率情况，由于 StackLSTM 模型的目的在于学习程序结构，因此预期 StackLSTM 模型应在预测程序结构时具有一定的优越性，而这里的程序结构（也就是代码段信息）对应于 AST 的中间节点也就是非终结符的预测。图 8 显示，代码长度为 60 时，标准 LSTM 的准确率仅有 28.4%，远远低于不置零和置零模型，其中不置零效果最好，不置零次之。随着 *numsteps* 增加（训练数据增加），不置零模型的准确率基本保持不变，而标准 LSTM 模型随着训练数据的增加准确率提高，最后与置零模型持平，在 *numsteps* 为 400 时略高于置零模型，但是仍低于不置零模型。

从上述结果可看出，不置零模型取得了较好的效果，仅需要很少的数据集就能学会程序中的结构信息，而标准的 LSTM 却需要大量数据才能够开始学到一定的程序结构信息，并且由于 LSTM 在处理结构化文本方面的局限性，取得的效果差于不置零模型。而置零模型在数据量较大时与 LSTM 的结果相近甚至低于 LSTM 模型的准确率，这是由于置零模型在预测时没有保留上文的程序结构信息。所以，可得出以下结论：上文程序中的结构信息对于预测当前代码段中的程序具有一定的影响，因此置零模

型具有一定的局限性，这就解释了在数据量增加时置零模型准确率低于标准的 LSTM 模型，因为 LSTM 模型可以记录上文程序信息。

图 10 和 11 分别是 top3 和 top5 的准确率：

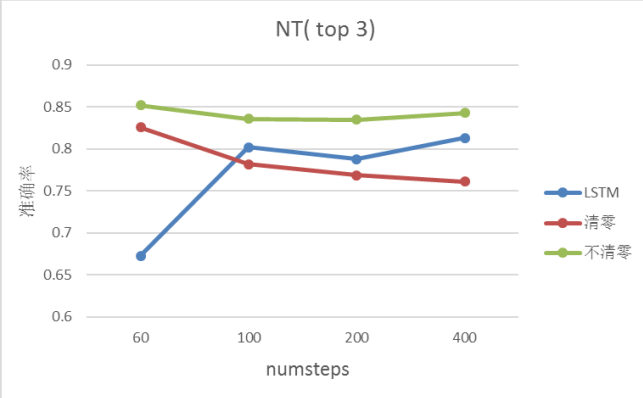

Fig.10 Non-termination(NT) top3 prediction results

图 10 非终结符（NT）top1 预测结果

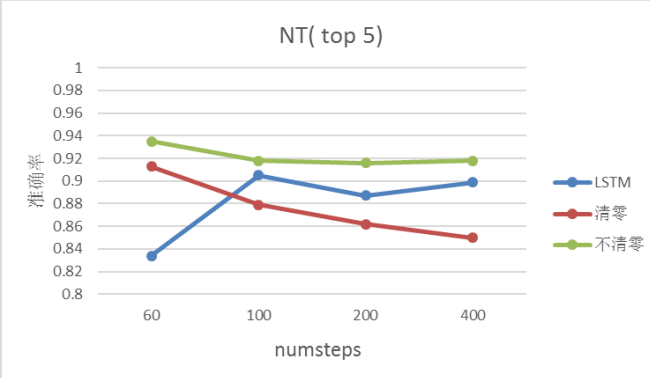

Fig.11 Non-termination(NT) top5 prediction results

图 11 非终结符（NT）top5 预测结果

可见，随着结果选择范围的增加，在预测非终结符时的准确率均得到了提高，这是由于程序结构的多样性，训练集中程序多样性较大，因此程序结构并不唯一。例如：Import 语句后面可能是另一个 Import 语句，或者也可能是 Assign 语句等等，这些都是合法的语句，模型的预测结果中不止存在一个合法的 token。因此扩大选择范围，从结果中概率较高的几个 token 中进行选择将会提高预测正确的可能性。结果显示，不置零模型仍然是效果最好的模型。

4.3.2 终结符预测

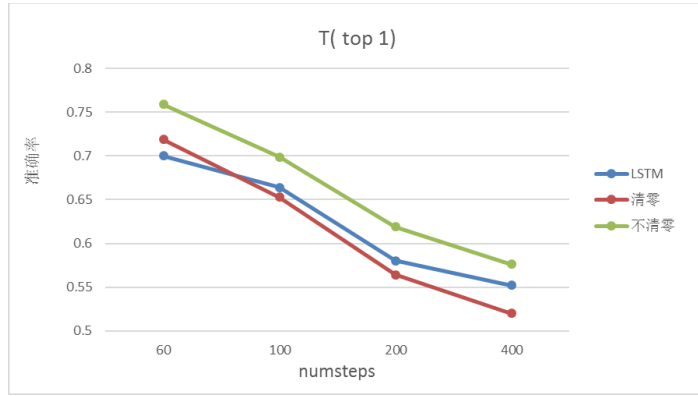

Fig.12 Termination(T) top1 prediction results

图 12 终结符（T）top1 预测结果

图 12 结果显示不置零模型在预测终结符时准确率也高于标准 LSTM 模型，这也证明了不置零模型不但能够很好地学会程序的结构，同时也能够学会程序中的语义信息。

因此，不置零模型具有十分强大的学习能力，在小数据集中就可以很好地学会程序的语义以及结构信息，强于标准的 LSTM 模型。

图 13 和 14 分别是 top3 和 top5 的准确率：

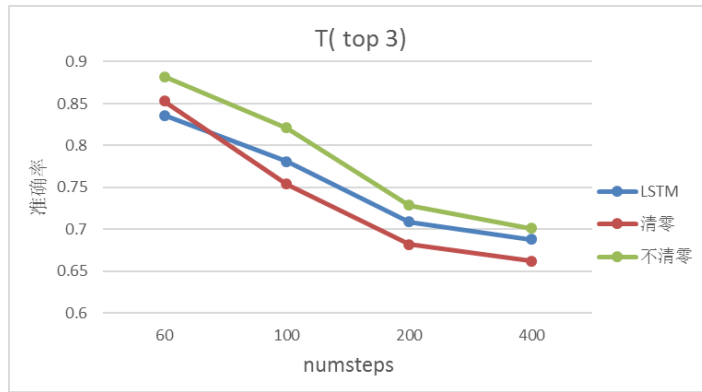

Fig.13 Termination(T) top3 prediction results

图 13 终结符（T）top3 预测结果

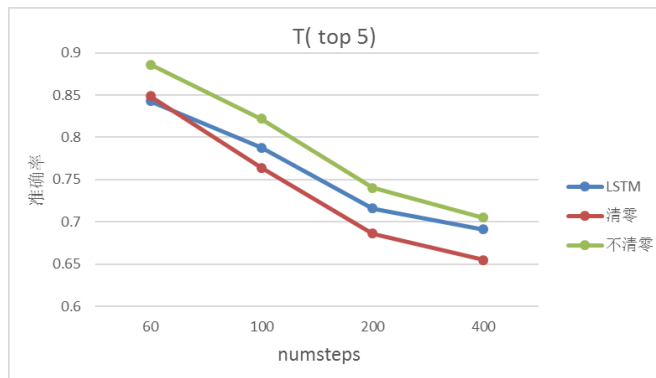

Fig.14 Termination(T) top5 rediction results

图 14 终结符（T）top5 预测结果

结果显示，虽然随着选择范围的增加，准确率的提升却没有十分明显，这是由于程序语义的多样性较低，因此扩大选择范围无太大意义，只需选择预测结果中概率最高的 token。

综上所述，StackLSTM 模型中的不置零模型具有十分强大的学习能力，在小数据集中就可以很好地学会程序的语义以及结构信息，可以有效地加速训练。在大数据集中，仍能取得比标准 LSTM 模型更好的效果，在补全非终结符和终结符时准确率均高于标准的 LSTM 模型。

5 总结与展望

近几年来，深度学习在许多领域取得了巨大的成功。本文将深度学习应用于程序自动补全的研究中，基于循环神经网络对 python 语言的程序自动补全展开研究，并对传统的 LSTM 模型进行了改进，提出了基于层次 LSTM 的程序自动补全模型——StackLSTM。通过对大量代码集进行学习，StackLSTM 模型在一定程度上学会了程序语言的语义以及结构信息。

5.1 本文贡献

（1）将 LSTM 模型用于处理程序补全问题，提出了能够保留程序结构的同时将其序列化的方法。

（2）通过增设记忆单元拓展标准的 LSTM 结构，减轻了 LSTM 的记忆负担，使其具备学习数据中的结构信息的能力。

5.2 未来展望

本文模型也存在着一些不足之处，仍然有许多需要改进的地方。

(1) 目前，本文模型在栈中仅存储了隐藏层状态，神经网络在预测下一个 token 时可以依据的信息量太少，所以模型无法处理更复杂的情况。在未来，可以尝试存储更多信息，如程序参数等，以提高程序补全的准确率，并使之可以应对更为复杂的场景。

(2) 本文提出的程序补全模型是以已有的代码为基础，通过对大量代码数据集进行学习使得模型学会程序中的语义及结构信息。但是模型并未将用户的需求作为程序补全的影响因素，在未来工作中，可以考虑利用用户的具体需求来对程序补全的过程加以约束，使得程序补全的结果更为有意义。

References:

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

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

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

[4] Collins M. Head-driven statistical models for natural language parsing[J]. Computational linguistics, 2003, 29(4): 589-637.

[5] Allamanis M, Sutton C. Mining idioms from source code[C]//Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014: 472-483.

[6] Allamanis M, Barr E T, Bird C, et al. Suggesting accurate method and class names[C]//Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, 2015: 38-49.

[7] Maddison C, Tarlow D. Structured generative models of natural source code[C]//Proceedings of the 31st International Conference on Machine Learning (ICML-14). 2014: 649-657.

[8] Bielik P, Raychev V, Vechev M. PHOG: probabilistic model for code[C]//International Conference on Machine Learning. 2016: 2933-2942.

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

[10] Bengio Y, Ducharme R, Vincent P, et al. A neural probabilistic language model[J]. Journal of machine learning research, 2003, 3(Feb): 1137-1155.

[11] Mikolov T, Sutskever I, Chen K, et al. Distributed representations of words and phrases and their compositionality[C]//Advances in neural information processing systems. 2013: 3111-3119.

[12] Hornik K, Stinchcombe M, White H. Multilayer feedforward networks are universal approximators[J]. Neural networks, 1989, 2(5): 359-366.

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

[14] Graves A, Wayne G, Danihelka I. Neural turing machines[J]. arXiv preprint arXiv:1410.5401, 2014.

[15] Joulin A, Mikolov T. Inferring algorithmic patterns with stack-augmented recurrent nets[C]// Advances in neural information processing systems. 2015: 190-198.

Liu Fang. She is a Ph.D. candidate at Peking University. Her research interest is deep learning.
刘芳，女, 北京大学博士研究生，主要研究领域为深度学习

Li Ge. He is an associate professor at Peking University. His research interests include deep learning, program analysis and natural language processing, etc.
李戈，男，北京大学信息科学技术学院副教授，主要研究领域为深度学习、程序分析、自然语言处理等

Jin Zhi. She is a professor at Peking University. Her research interests include knowledge engineering and requirements engineering, etc.
金芝，女，北京大学信息科学技术学院教授，主要研究领域为需求工程、知识工程等