计算机研究与发展
Journal of Computer Research and Development
DOI:10.7544/issn1000-1239.2021.20200298
58(3):638-650,2021

一种融合程序员和神经网络的自动化程序生成方法

周 鹏[1,2] 武延军[1,3] 赵 琛[1,3]
[1](中国科学院软件研究所 北京 100190)
[2](中国科学院大学 北京 100049)
[3](计算机科学国家重点实验室(中国科学院软件研究所) 北京 100190)
(zhoupengwork01@163.com)

A Programming Paradigm Combining Programmer and Neural Network to Promote Automated Program Generation

Zhou Peng[1,2], Wu Yanjun[1,3], and Zhao Chen[1,3]
[1](*Institute of Software*, *Chinese Academy of Sciences*, *Beijing* 100190)
[2](*University of Chinese Academy of Sciences*, *Beijing* 100049)
[3](*State Key Laboratory of Computer Science* (*Institute of Software*, *Chinese Academy of Sciences*), *Beijing* 100190)

Abstract Program generation is one of the core research challenges in AI. At present, the neural network methods driven by input-output data are very popular for program generation modeling. Because of incomplete information input to the model, and complete dependency and limited memory capacity of the neural network, the learning performances of these models suffer from the challenges of poor generalization, generated program accuracy assurance, and being not competent for dealing with common program structures. The memory capacity of neural network can not meet the variable storage requirements of conventional programs. Thus, we propose a programming paradigm merging the strengths of human's experience and perception with those of neural network's learning from data samples. Human programmers provide the overall program structure roughly, and leverage the neural network to generate the local trivial detail automatically. The programs run on an abstract computer like digital computer, but are end-to-end differentiable, which consists of differentiable controller, extended external memory relative to internal memory cells of neural network, and differentiable instruction set represented by differentiable state transfer function library. So, it can not only receive input-output samples but also program and execute instructions like traditional digital computers, and its extended memory visible from outside provides more capacity for program variable representation. The advantage is to promote program generation's practical applicability. The experimental results indicate that the method is effective and gets much better learning performance than other typical methods in program generation.

Key words intelligent software; program generation; software engineering; differentiable programming language; artificial intelligence

摘 要 程序生成是人工智能的核心研究问题之一,当前输入-输出样例驱动的神经网络模型是非常流行的研究方法.面临的主要挑战是泛化能力差、生成程序准确率保证、难以处理复杂程序结构(如分支、循环、递归等),主要原因是模型的输入信息单一(输入-输出对)和完全依赖神经网络.显然单一地通过

收稿日期:2020-04-28;**修回日期**:2020-09-18
基金项目:中国科学院战略性先导科技专项(A类)(Y8XD373105);中国科学院前沿科学重点研究计划项目(ZDBS-LY-JSC038)
This work was supported by the Strategic Priority Research Program of Chinese Academy of Sciences(A) (Y8XD373105) and the Key Research Program of Frontier Sciences, CAS (ZDBS-LY-JSC038).

输入输出样例倒推程序行为存在歧义性,而神经网络的记忆容量很难满足常规程序的变量存储需求.提出一种人工与神经网络生成相协作的编程模型,融合神经网络和程序员各自的优势,其中程序员用高级编程语法编写程序框架,神经网络自动学习生成程序局部的琐碎细节,从而促进自动化程序生成方法更好地应对实际应用挑战.实验表明,研究方法是有效的,跟同类代表性研究方法相比表现出更好的学习性能.

关键词　智能软件;程序生成;软件工程;可微分编程语言;人工智能

中图法分类号　TP311

教会机器理解代码、编写程序或构造可自学习的智能软件一直是人工智能的重要挑战问题之一[1].当前基于神经网络的深度学习在图像处理、语音识别、自然语言处理等领域从研究到应用取得了很大的成功,但是在程序生成上距离研究到实用仍有明显差距.许多学者在程序处理领域也使用神经网络深度学习方法进行了积极探索,提出了不少研究方法,取得了很好的效果,但是仍然未能有效解决泛化能力差、程序正确性、难以处理复杂程序结构的问题.当前基于神经网络的程序生成研究主要包括基于程序输入-输出数据样例的方法[2-6]、基于标注程序执行轨迹样例的方法[7-8]和基于标注源码的方法[9-13].

1) 基于程序输入-输出样例的方法.该方法以神经网络为待学习控制器,从输入-输出样例训练控制器学习预测对神经网络扩展外部存储的操作序列.该方法完全依赖输入输出数据对,存在从数据倒推操作序列的歧义性、泛化能力差、限定一个小的DSL(domain specific language)语言[14]空间、只能处理非常简单的程序结构等问题.比如著名的微软DeepCoder[2]只能处理自定义的一个小型DSL语言,主要预测5行以内、每行只有一个主标识符、顺序结构的平铺程序,不能处理分支、递归、跳转等复杂控制结构,因此习得的程序非常简单.

2) 基于标注程序执行轨迹样例的方法.该方法收集覆盖算法所有执行路径的逐步执行轨迹作为有监督训练数据,其标注方法是以下一步作为上一步的标注.DeepMind NPI[7]是最有代表性的相关研究之一,其优点是明显提高了泛化能力.缺点是数据标注成本高,同时考虑到每个执行轨迹是算法的特定路径,因此需要预先已知算法逻辑,所以本质上未能从输入-输出样例对中倒推学习到新代码逻辑,而是从标注执行路径中学习算法的神经网络表示;另外NPI是对标测试的主要模型之一,其泛化能力虽然有很大提升但仍然是有限的,对此实验分析部分有具体对比.

3) 基于标注源码的方法.对源代码做有监督标注.学习源码跟标注间的统计映射关系,然后通过输入标签反过来推荐源程序.该方法会面临由编程语言过多的规则、实现细节、源码本身丢失语义信息所带来的学习复杂度;同时未能输入源程序的背景知识,比如程序员预先学习的编程语言规范、计算机原理教程等,导致模型处理信息不完备.额外的复杂度加上不完备的信息给模型训练学习带来了巨大挑战.BAYOU[12]是最具代表性的相关研究之一,其优点是可以从注释标签生成大体正确的Java源代码推荐,不足是不能保证和自我检验生成的代码语法是否正确,因此生成的程序代码具有参考价值而不能直接解释执行.

因此,完全依赖神经网络,单一地以源代码标注、输入-输出对、程序执行轨迹做为训练数据输入构建程序生成模型存在局限性.比如程序生成模型在实际应用中一般需要关心模型的泛化能力、生成程序正确性、处理复杂程序结构3个指标,可以分别定性地解释为模型生成的程序表示需要具备一定程度的变通能力(举一反三是理想情况)、执行的准确率、可以处理有一定结构的程序从而能描述更大范围或有一定复杂度的任务,但是这些模型在应对3个指标时面临挑战.我们分析认为这主要是因为编程任务往往非常复杂、程序相对于自然语言有其特殊性,完全依靠神经网络构建的程序生成模型未能结合程序员的价值、未能获得丰富的输入信息(如背景知识).有意思的是观察人类学习编程的过程,程序员会预先学习编程语言的语法规则、计算机原理等背景知识;对比神经网络和人类,神经网络善于从大量数据进行自动化统计学习,人类善于问题分解、直觉感知、经验复用.受此启发,应对程序生成的复杂性和挑战性,我们探讨如何发挥神经网络和人类程序员的整合优势,实现优势互补.

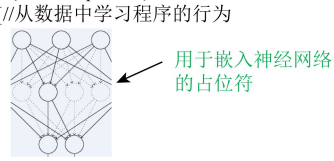
因此本文提出一种融合程序员和神经网络各自

优势的协作式编程范式 HNCP.HNCP 是一种高级可微分("可微分"是使用反向传播算法求解神经网络权重参数的基础)编程语言,执行在一种提供端到端可微分运行环境的编程语言级抽象虚拟机(differentiable virtual machine, dVM)上.在 HNCP 编程范式下,程序员用类 C 高级编程语言提供程序框架,由神经网络组件根据训练数据学习(可微分优化)生成程序的局部细节,因此最终完整确定的程序是由人类程序员和神经网络共同创作完成.如图 1 所示,图 1(a)是一个程序框架,其功能是不完整的,其中空缺部分如图 1(b)所示,无缝嵌入一个神经网络组件,程序的完整行为将由程序员提供的程序框架和通过训练数据(输入-输出对)参数化的神经网络组件共同确定.

```
1  #define size 10
2  int a[size]={2,4,9,1,7,8,6,9,3,5};
3  void bubble (int pass, int c){
4     if (c >=size-pass-1) return;
5     {//待补全的程序细节
6     _____;
7     _____;
8     _____;
9     _____;
10    _____;
11    }
12    bubble(pass, c+1);
13 }
14 bubbleSort(){
15    int i=0; while(i<size-1) { bubble(i, 0); i++; }
16 }
17 bubbleSort();
```

(a) 待生成程序的框架

```
1  #define size 10
2  int a[size]={2,4,9,1,7,8,6,9,3,5};
3  void bubble(int pass, int c){
4     if (c >=size-pass-1) return;
5     {//从数据中学习程序的行为
6
7
8
9
10
11    }
12    bubble(pass, c+1);
13 }
14 bubbleSort(){
15    int i=0; while(i<size-1) { bubble(i, 0); i++; }
16 }
17 bubbleSort();
```



(b) 神经网络组件作为类C语言内置构件

Fig. 1 Illustration of HNCP for automatic program generation

图 1 HNCP 程序自动生成示例

本文主要贡献有 3 个方面:

1) 提出了神经网络与人工程序员协作的高级编程范式 HNCP,它可以结合神经网络和程序员各自的优点,使程序生成适用于复杂的程序结构(如循环、递归、分支、函数调用等);

2) 给出了一种高级过程化编程语言可微分运行环境(dVM)的实现方法.dVM 是实现传统过程化程序结构跟神经网络组件无缝融合的基础,也是将离散的程序处理问题转换为能用可微分优化直接处理的问题的关键;

3) 实验表明,跟代表性同类程序生成模型相比,HNCP 在程序生成任务上表现出更好的学习性能(泛化能力、样本复杂度、准确率、复杂程序结构).这表明整合人工程序员和神经网络的协作式编程是一种有潜力的研究方法,可以促进程序自动生成的实际应用能力.

1 相关工作

文献[12]提出了一种基于 Java 源程序语法相似性的组合搜索生成源代码示例的方法.我们都用高级编程语言处理复杂的程序,但是文献[15]中的方法是以语法而不是语义为条件的,既不考虑生成程序的准确性,也不考虑生成程序的完整性,生成的代码示例具有参考价值而不能直接解释执行.因此,我们在研究目标和方法上都与之不同.文献[16]提出了基于 C 语言的程序骨架编程,它是基于 SAT (Boolean satisfiability problem)的归纳综合过程,即使用搜索和符号推理来寻找满足人工定义规则的程序.但在本文中,我们采用的是神经网络可微分优化法,也不需要额外的人工定义规则,因此是明显不同的;文献[15-18]扩展了神经网络使之具有较大外部记忆存储,这为可微分存储器缓存的设计提供了有益的启发.但它们都缺乏高级编程接口,而在本文中,我们提供了一个具有外部存储器和高级 C 语言接口的可微分编程系统,这对于整合程序员的编程框架或者经验、无缝融合神经网络组件到过程化程序结构中是非常重要的.

2 方法设计与实现

HNCP 是基于类 C 语法的扩展,增加了神经网络组件嵌入的语言构造.HNCP 源程序首先需要编译成中间代码表示(intermediate representation, IR),从而便于以可微分的矩阵计算方式进行程序解释执行和求解.IR 的指令集设计参考了 Forth[19] 和

SECD[20].表 1 对 IR 指令集的主要指令进行了说明.该指令集是一个栈式求值语言,我们基于该栈式求值模式构造了一个可微分的运行时环境,命名为 dVM,它是一个语言级虚拟机,满足直接对 IR 程序的可微分解释执行,从而满足对 HNCP 的类 C 源程序的解释执行.程序在 dVM 上以端到端可微分的方式执行生成程序的计算图表示,包括神经网络组件构成计算图的组成部分.进一步,通过输入-输出样例训练计算图来确定嵌入的神经网络组件参数,从而学习生成一个完整、确定的程序表示.

Table 1　Instruction Set

表 1　指令集

指令	描述	微指令
CALL L	保存返回地址(P_C+1)到栈 R,然后跳转到 L.	$P_R \leftarrow P_R+1; R[P_R] \leftarrow P_C+1; P_C \leftarrow L$
RET	跟 CALL 指令配对.从栈 R 弹出返回地址,然后跳转到该地址.	$P_C \leftarrow R[P_R]; P_R \leftarrow P_R-1$
GOTO L	跳转到 L	$P_C \leftarrow L$
GOTOF L	实现 if-then-else 结构.如果条件为假跳转到 L,否则执行下条指令(PC++);条件取自栈 S 的栈顶值.	if $S[P_S]==$False then $P_C \leftarrow L$ else $P_C \leftarrow P_C+1; P_S \leftarrow P_S-1$
DO	跟 LOOP 和 ENDDO 配对.在循环的开始从栈 S 拷贝参数到栈 R.S 栈顶是循环计数器,次顶是循环次数上限.	$P_R \leftarrow P_R+2$; $[P_R] \leftarrow S[P_S]; P_S \leftarrow P_S-1; R[P_R-1] \leftarrow S[P_S]; P_S \leftarrow P_S-1$;NEXT
LOOP L	R 栈顶值增 1.如 R 栈顶值小于次顶值跳转到 L,否则退出该循环.	$R[P_R] \leftarrow R[P_R]+1$; if $R[P_R]<R[P_R-1]$ then $P_C \leftarrow L$ else $P_C \leftarrow P_C+1$
ENDLOOP	'do … loop endloop'循环结构,清空栈 R 中的参数.	$P_R \leftarrow P_R-2$;NEXT
NEXT	显示向前推进一步.	$P_C \leftarrow P_C+1$
INC	S 栈顶值增 1.	$S[P_S] \leftarrow S[P_S]+1$;NEXT
DEC	S 栈顶值减 1.	$S[P_S] \leftarrow S[P_S]-1$;NEXT
SWAP	交换栈 S 的栈顶元素和次顶元素.	$t \leftarrow S[P_S]; S[P_S] \leftarrow S[P_S-1]; S[P_S-1] \leftarrow t$
+	以 S 栈顶元素和次顶元素为参数做加法,结果保存在 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$ADD($t_1,t_2$);
−	S 栈顶元素减去次顶元素,结果保存在 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$SUB($t_1,t_2$);
*	S 栈顶元素乘以次顶元素,结果保存在 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$MUL($t_1,t_2$);
/	S 栈顶元素除以次顶元素,结果保存在 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$DIV($t_1,t_2$);
==	检测 S 栈顶元素是否等于次顶元素,结果作为条件存 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$EQ($t_1,t_2$);
>	检测 S 栈顶元素是否大于次顶元素,结果作为条件存 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$GT($t_1,t_2$);
<	检测 S 栈顶元素是否小于次顶元素,结果作为条件存 S 栈顶.	$t_1 \leftarrow S[P_S]; P_S \leftarrow P_S-1; t_2 \leftarrow S[P_S]; S[P_S] \leftarrow$LT($t_1,t_2$);
SAVE	将 S 中临时变量值存到 H 中,S 栈顶元素是变量在 H 地址、次顶元素是变量值.	$a \leftarrow S[P_S]; P_S \leftarrow P_S-1; v \leftarrow S[P_S]; P_S \leftarrow P_S-1; H[a] \leftarrow v$; NEXT
LOAD	从 H 中加载变量值到 S 栈,S 栈顶是变量在 H 中地址.	$a \leftarrow S[P_S]; S[P_S] \leftarrow H[a]$;NEXT
HALT	停机,机器状态不再改变.	$P_C \leftarrow P_C$

注:该表给出了 dVM 的基本指令,描述了指令的过程化微操作逻辑,所以该表中的操作是标量.

2.1　HNCP 方法概述

图 2 所示是用 HNCP 协作式编程创作的源程序框架自动生成完整、确定的程序的整体处理流程.

首先通过编译技术将 C 语法的 HNCP 源程序翻译为更底层的(直接面向 dVM)指令集表示的中间语言表示;然后根据分支、控制结构对程序进行分块,

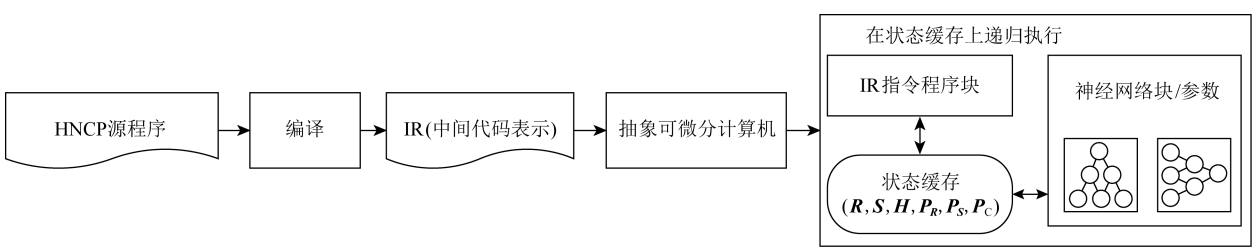


Fig. 2　Overall processing flow of HNCP program

图 2　HNCP 程序处理的整体流程

包括普通的中间语言代码块和神经网络块(统称代码块).

每个神经网络块都携带着可以通过训练数据来确定其行为的可学习参数(所以完整的程序行为是由程序框架和数据训练学习共同确定).每个代码块是用代码片段构成的函数来实现,而代码片段描述为矩阵代数运算(即指令集的可微分实现),dVM 的状态缓存部件以及对状态的访问、处理也用矩阵和矩阵代数表示,因此程序执行的每个基础步骤是寻址下一个代码块,在 dVM 的状态部件上做矩阵代数计算,程序的整体执行是由这些基础步骤一步一步推进构成的递归过程.dVM 的状态由 $\boldsymbol{R}$(返回栈)、$\boldsymbol{S}$(求值数据栈)、$\boldsymbol{H}$(随机变量堆)这些状态缓存和 $\boldsymbol{P}_{\boldsymbol{R}}$(返回栈栈顶指针)、$\boldsymbol{P}_{\boldsymbol{S}}$(数据栈栈顶指针)、$\boldsymbol{P}_{\mathrm{C}}$(程序计数器指针)组成.

2.2 dVM 可微分虚拟机

从公理语义[21]的角度将指令的执行和求值表示为虚拟机的状态转换.我们给出了一个全可微分计算系统(dVM)的实现方法,dVM 支持高级类 C 编程语言,具有 IR 指令集和扩展内存.

2.2.1 可微分虚拟机整体结构

图 3 所示,可微分虚拟机 dVM 包括 3 个状态区(即相对于神经网络内部记忆空间的扩展内存)、1 个控制器(对应于处理器)和状态转移指令集.扩展内存区包括 2 个栈结构($\boldsymbol{S}$,$\boldsymbol{R}$)和一个随机内存结构($\boldsymbol{H}$);控制器包括 $\boldsymbol{P}_{\boldsymbol{S}}$,$\boldsymbol{P}_{\boldsymbol{R}}$,$\boldsymbol{P}_{\mathrm{C}}$ 这 3 个寄存器,$\boldsymbol{P}_{\boldsymbol{S}}$,$\boldsymbol{P}_{\boldsymbol{R}}$ 分别是 $\boldsymbol{S}$,$\boldsymbol{R}$ 栈顶指针,$\boldsymbol{P}_{\mathrm{C}}$ 是程序计数器指针.支持编程控制,即支持输入用指令集写好的程序(程序框架)在虚拟机上执行.

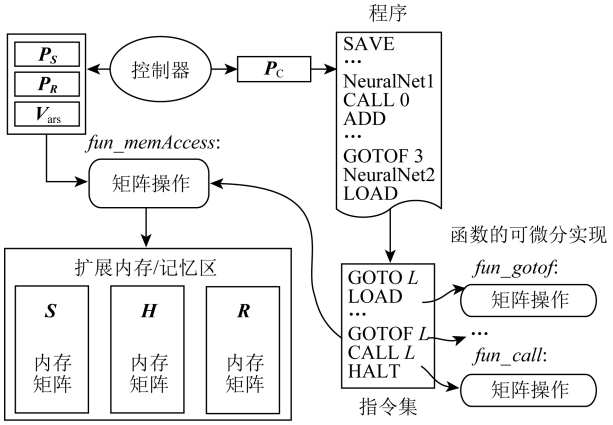


Fig. 3 The dVM architecture

图 3 可微分虚拟机整体结构

为了保证整个计算环境是连续可微分地执行程序,指令集(包括扩展内存访问指令)用算术矩阵操作实现为可微分函数(fun_foo),扩展内存区用矩阵表示,寄存器用 one-hot 向量表示.控制器负责取指令、执行指令、更新抽象机状态,控制器还包括一个指示指令寻址的向量 $\boldsymbol{P}_{\mathrm{C}}$,用于实现执行控制转移.这些基本行为都以连续的可微分函数实现.

2.2.2 指令集与指令集的可微分实现

表 1 给出了 dVM 的主要指令集的说明.为了不影响阅读连贯性,我们在附录中给出了指令集可微分实现的详细数学表示,参见附录 A.

内存读写的实现原理为,将栈指针看做分布在内存空间上的访问权重,基于此将内存访问(如 $READ$,$WRITE$)表示为位置权重跟内存单元的矩阵计算,然后基于 $READ$,$WRITE$ 的线性组合可以实现 $PUSH$,POP 等复杂的栈操作;指令集的可微分实现原理为,对 dVM 状态(扩展内存)的访问与改变实现为 $READ$,$WRITE$,$PUSH$,POP 的线性组合;对于算术逻辑操作、控制操作通过矩阵代数运算和运算的组合来实现,矩阵算术操作理论指导可参看双线性映射[22].因此所有的指令集都是用矩阵代数运算表示的连续可微分实现,可以跟神经网络(神经网络也是矩阵代数运算)无缝整合,因此可以用反向传播算法的可微分优化技术对程序生成、程序处理进行直接建模.

2.2.3 程序的可微分执行模型

算法 1 描述了过程化程序结构在 dVM 上可微分解释执行流程.从状态转换的角度,其执行过程被描述为从初始状态 $\boldsymbol{S}_0$ 出发的 $steps$ 步状态迁移.算法 1 中 $diffInstsLib$ 是指令集的可微分实现函数库,通过字典结构索引.用到的 $FetchInst$,$step$ 操作.dVM 的初始状态设置,训练开始时先将 dVM 扩展内存(即 $\boldsymbol{S}$,$\boldsymbol{R}$,$\boldsymbol{H}$ 构成的状态区)清 0,然后用标注数据"输入-输出"样例对的"输入"初始化填充 dVM 扩展内存,并将对应的栈指针初始化指向下一个空白位置,经过初始设置内存区构成初始状态 $\boldsymbol{S}_0$.附录 B 是一段程序在 dVM 上运行过程例子.

算法 1. 可微分虚拟机程序执行流程.

输入:$\boldsymbol{S}_0$ 是 dVM 初始状态、$\boldsymbol{Code}$ 是待执行程序框架、$steps$ 是执行总步数;

输出:状态转换列表 $\boldsymbol{LTRACE}=[\boldsymbol{S}_0,\boldsymbol{S}_1,\cdots,\boldsymbol{S}_{steps}]$.

① function:run($\boldsymbol{S}_0$,$\boldsymbol{Code}$,$steps$)

② $\boldsymbol{S}\leftarrow\boldsymbol{S}_0$,$\boldsymbol{P}_{\mathrm{C}}\leftarrow\boldsymbol{S}.pc$,$LTRACE[]$;

③ $LTRACE.append(\boldsymbol{S})$;

④ for ($i=1$;$i\leqslant steps$;$i++$)

⑤　$\boldsymbol{Weight}_{\text{state}} \leftarrow FetchInst(\boldsymbol{P}_{\text{C}}, \boldsymbol{Code})$;

⑥　for ($j=1; j \leqslant \boldsymbol{Code}.size; j++$)

⑦　　$\boldsymbol{S}_{\text{tmp}} \leftarrow diffInstsLib[\boldsymbol{Code}[j].key](\boldsymbol{S})$;

⑧　　$\boldsymbol{StatesVector}.append(\boldsymbol{S}_{\text{tmp}})$;

⑨　end for

⑩　$\boldsymbol{S}_{\text{next}} \leftarrow step(\boldsymbol{Weight}_{\text{state}}, \boldsymbol{StatesVector})$, $\boldsymbol{S} \leftarrow \boldsymbol{S}_{\text{next}}$;

⑪　$LTRACE.append(\boldsymbol{S})$;

⑫ end for

⑬ return $\boldsymbol{LTRACE}$.

算法 1 的执行过程是从初始状态开始的 *steps* 步连续状态迁移,每一步是通过寻址从代码空间选择指令在当前状态上执行并推进 dVM 迁移到下一个状态.因此整个执行过程构成一个 *steps* 步的 RNN 模型,程序的一遍执行可以描述为

$$\boldsymbol{exeTrace}=(\boldsymbol{S}_1, \boldsymbol{S}_2, \cdots, \boldsymbol{S}_i, \cdots, \boldsymbol{S}_{steps}), \tag{1}$$

其中 $\boldsymbol{S}_i = step(\boldsymbol{Weight}_{\text{code}}, \boldsymbol{S}_{i-1})$,$\boldsymbol{S}_1$ 是初始状态.

代码空间寻址是根据 $\boldsymbol{P}_{\text{C}}$ 和代码矩阵 $\boldsymbol{Code}$ 计算在代码行的权重分布:

$$FetchInst(\boldsymbol{P}_{\text{C}}, \boldsymbol{Code}) \leftarrow (\boldsymbol{P}_{\text{C}} \odot \boldsymbol{Code})\boldsymbol{1}^{\text{T}}. \tag{2}$$

计算取指令,其中 $\boldsymbol{Code}$ 是需要执行程序的矩阵结构化表示,$\boldsymbol{P}_{\text{C}}$ 是程序计数器的向量表示,$\boldsymbol{1}$ 是行向量(1,1,…,1),$\odot$是哈达玛积.

$$step:(\boldsymbol{Weight}_{\text{code}}, \boldsymbol{State}_{\text{current}}) \rightarrow \boldsymbol{State}_{\text{next}} =$$

$$(\boldsymbol{Weight}_{\text{code}} \odot \boldsymbol{StatesVector})\boldsymbol{1}^{\text{T}} = \sum_{i=0}^{cs} \boldsymbol{Weight}_{\text{code}}^{i} \boldsymbol{s}_i =$$

$$\sum_{i=0}^{cs} \boldsymbol{Weight}_{\text{code}}^{i} functionCB_i(\boldsymbol{state}). \tag{3}$$

单步执行,其中 $\boldsymbol{State}_{\text{current}}$ 是当前状态向量,$\boldsymbol{State}_{\text{next}}$ 是执行后迁移的下一个状态,$\boldsymbol{Weight}_{\text{code}}$ 是取指令计算得到的指令执行权重分布(用 softmax 计算),$functionCB_i$ 是汇编后的代码块(即用函数代码片段给出的指令集可微分实现),$\boldsymbol{StatesVector}$ 是状态行向量.该求值逻辑如图 4 所示,$\boldsymbol{S}_{\text{P}}$ 是执行前当前状态,$\boldsymbol{S}_{\text{N}}$ 是单步执行后迁入的状态,$\boldsymbol{S}_{\text{n\_}i}$ 是 dVM 在 $\boldsymbol{S}_{\text{P}}$ 状态下执行地址空间某个指令块得到的状态.

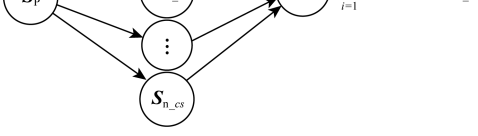


Fig. 4　Step forward to get the next state

图 4　单步执行计算下一个状态

2.2.4　训练目标

对所有训练样本求解使模型推理状态 $\boldsymbol{y}$ 跟样本标注状态 $\hat{\boldsymbol{y}}_{\text{e}}$ 之间的差距最小化的模型参数 $\boldsymbol{\theta}^*$,这可用交叉熵表示为对于每个训练样本 i,求解使得 $\hat{\boldsymbol{y}}_{\text{e}}$ 与 $\boldsymbol{y}$ 之间的交叉熵最小化的模型参数 $\boldsymbol{\theta}^*$:

$$\boldsymbol{\theta}^* = ARGMIN_{\boldsymbol{\theta}} H(\hat{\boldsymbol{y}}_{\text{e}}, \boldsymbol{y}) =$$

$$ARGMIN_{\boldsymbol{\theta}} - \sum_i \hat{\boldsymbol{y}}_{\text{e\_}i} \ln(\boldsymbol{y}_i). \tag{4}$$

在求值时对状态值 $\hat{\boldsymbol{y}}_{\text{e}}$ 和 $\boldsymbol{y}$ 按照图 5 所示的扩展内存区分量进行分解计算:

$$\boldsymbol{\theta}^* = ARGMIN_{\boldsymbol{\theta}}[H(\hat{\boldsymbol{S}}_{\text{e}}, \boldsymbol{S}) +$$

$$H(\hat{\boldsymbol{R}}_{\text{e}}, \boldsymbol{R}) + H(\hat{\boldsymbol{H}}_{\text{e}}, \boldsymbol{H})]. \tag{5}$$

我们在求值中发现通过 $\boldsymbol{P}_{\text{C}}, \boldsymbol{P}_{\boldsymbol{S}}, \boldsymbol{P}_{\boldsymbol{R}}$ 指针计算状态分量的掩码(即只计算执行结束时扩展内存的有效区域),可以免去 $\boldsymbol{P}_{\text{C}}, \boldsymbol{P}_{\boldsymbol{S}}, \boldsymbol{P}_{\boldsymbol{R}}$ 分量的距离计算.

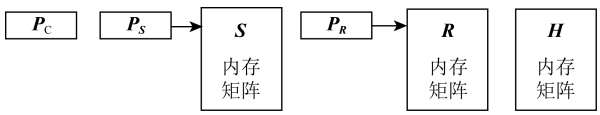


Fig. 5　dVM state buffer components

图 5　dVM 状态分量

2.3　从 C 语法到中间代码表示的编译生成

HNCP 原型实现是在 C 语法基础上扩展支持嵌入神经网络作为基本语言构造,称作 NCP(neural network component placeholder).图 6 所示为 HNCP 前端语法产生式(grammar productions)的 BNF 描述;其中 NCP 的语法构造由〈ncp〉产生式衍生.作为原型验证〈ncp〉只给出了 encoder-transform-decoder 神经网络模型结构,其中 *encoder* 对程序上下文输入参数条件编码,*transform* 对条件进行特征转换,*decoder* 解码条件并基于此预测生成执行动作.得益于 dVM 可微分运行时环境,采用类似思路可增加其他可微分模型,如 CNN(convolutional neural networks),seq2seq(sequence to sequence)等作为 HNCP 语法构造的编程原语.

HNCP 使用栈式求值语言作为中间表示(IR),通过 BNF 产生式语义规则可以定义将 HNCP 源程序编译为 IR 表示的语义动作;其基本思路是将 C 语法构造翻译为 IR 指令、语言构造表示,如 while 用 do-loop-endloop,if-else 用 GOTOF,函数调用用 CALL,函数返回用 RET,变量定义用 $\boldsymbol{H}$ 堆内存分配,不同作用域的同名变量用可区分的命名前缀表示等.附录 C 给出了 BNF 产生式语义规则的实现样例.

```
⟨prog⟩::=⟨decl_var⟩⟨def_func⟩[⟨main⟩]
⟨def_func⟩::={id'('⟨parm⟩')''{'⟨decl_var⟩⟨stmt⟩'}'}
⟨parm⟩::=int id
⟨main⟩::='main''('')''{'⟨decl_var⟩⟨stmt⟩'}'
⟨decl_var⟩::={int id';'|int id'['NUM']';'|int id'['NUM
  ']';'['NUM']';'}
⟨stmt⟩::={⟨expr_stmt⟩';'|⟨ctl_stmt⟩}/* statements of
  expression and control */
⟨expr_stmt⟩::=id'='⟨expr⟩|id'['⟨expr⟩']''='⟨expr⟩
  |id'++'|id'--'|id'+'⟨expr⟩|id'-'⟨expr⟩|id'*'⟨expr⟩
  |id'/'⟨expr⟩|return⟨expr⟩|call|⟨ncp⟩
⟨ctl_stmt⟩::='while'⟨expr⟩'{'⟨stmt⟩'}'|'do''{'⟨stmt⟩'}'
            'while'⟨expr⟩';'
            |'if'⟨expr⟩'{'⟨stmt⟩'}'⟨else_block⟩
            |'for''('⟨expr_stmt⟩';'⟨expr⟩';'⟨expr_stmt⟩')'
             '{'⟨stmt⟩'}'
⟨else_block⟩::='else''{'⟨stmt⟩'}'|ε
⟨expr⟩::=⟨expr⟩'+'⟨expr⟩|⟨expr⟩'-'⟨expr⟩|⟨expr⟩'*'
          ⟨expr⟩
         |⟨expr⟩'/'⟨expr⟩|⟨expr⟩'|'⟨expr⟩|⟨expr⟩'%'
          ⟨expr⟩
         |⟨expr⟩'>'⟨expr⟩|⟨expr⟩'<'⟨expr⟩|⟨expr⟩'=='
          ⟨expr⟩
         |⟨expr⟩'!='⟨expr⟩|'!'⟨expr⟩
⟨call⟩::=id'('⟨arg⟩')'
⟨arg⟩::=⟨expr⟩⟨arg_r⟩|ε
⟨arg_r⟩::=','⟨expr⟩⟨arg_r⟩|ε
⟨ncp⟩::='⟨%'id=⟨encoder⟩:⟨transform⟩:⟨decoder⟩'%⟩'
⟨encoder⟩::='observe''('⟨parm_ncp⟩')'
⟨parm_ncp⟩::=id['['NUM']'{','id['['NUM']']}|{NUM
  ','}|{⟨action_ncp⟩','}
⟨action_ncp⟩::=swap|nop|+|-|*|/|INC|DEC|mod|max|min
  |dup
⟨transform⟩::='transform''('⟨tf_fun⟩'('{⟨parm_ncp⟩}')'
  {','⟨tf_fun⟩'('{⟨parm_ncp⟩}')'}')'
⟨tf_fun⟩::=sigmoid|tanh|relu|linear
⟨decoder⟩::='decoder''('⟨dec_fun⟩'('{⟨parm_ncp⟩}')'')'
⟨dec_fun⟩::=choose|permute|manipulate|id
```

Fig. 6 HNCP BNF grammar productions

图 6 HNCP 的 BNF 语法产生式

HNCP 源程序的⟨ncp⟩原语翻译为 IR 表示后保持结构不变,但原语中引用的 C 语法变量会被替换为在 $\boldsymbol{S}$ 和 $\boldsymbol{R}$ 扩展内存区中分配的底层变量表示.下面对 encoder-transform-decoder 编程原语进行说明:

1) 在该神经原语中,*encoder* 以上下文参数作为条件进行特征编码,*transform* 做进一步特征转换,同时也是对⟨ncp⟩网络参数和层数做适当的调节,*decoder* 解码并做出执行动作决策.

2) *linear*(***out***)增加输出宽度为 ***out*** 的线性变换层.

3) *observe* 将 dVM 状态缓存的元素进行联合.元素取自求值数据栈($\boldsymbol{S}$)、控制返回栈($\boldsymbol{R}$)或随机访问内存堆($\boldsymbol{H}$)中的单元项.

4) $choose(action_i,\cdots,action_j)$,执行每个动作 *action*,取每个执行结果的权重($\boldsymbol{W}_h$)和作为最终结果.权重 $\boldsymbol{W}_h$ 的计算以 *transform* 的输出($\boldsymbol{h}$)为条件,即 $\boldsymbol{W}_h^i=P(action_i|\boldsymbol{h})$,以 $\boldsymbol{h}$ 为参数用 softmax 计算在 *action* 列表上概率权重分布.

图 7 所示为 encoder-transform-decoder 的典型神经网络拓扑,注意图中每个隐藏层可包含多层神经网络.

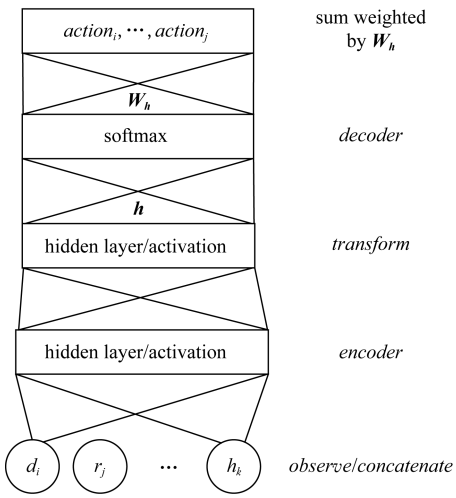


Fig. 7 A typical topology of encoder-transform-decoder

图 7 encoder-transform-decoder 网络拓扑

3 实验评估

本节对模型的程序生成学习性能、处理复杂程序结构的能力进行分析评估,包括泛化能力、样本复杂度(训练样本消耗)、准确率的评估,以及代表性程序生成模型的主要指标对比性分析.DeepMind NPI[7]是生成直接可运行程序表示一类模型中表现最好的,seq2seq[23]是程序生成研究中普遍用来做参照的基准模型,因此 HNCP 主要跟这 2 个模型做对比.要比较的学习任务采用 NPI 中使用的冒泡排序(SORT)和多位数初等加法(ADD),除了可以跟 NPI,seq2seq 做直接对比,还考虑到它们包含多层次的复杂程序结构,如函数调用、循环、分支、递归,这些结构是广泛的复杂任务编程关键,因此具有代表性.附录 D 给出了任务的程序框架详情.

3.1 生成程序泛化能力

泛化能力是评估程序生成模型实用性最重要的指标之一,定义为 $Gen=Length_{\text{test\_max}}/Length_{\text{train\_max}}$,*Gen* 值越大越好,其中 $Length_{\text{train\_max}}$ 是训练期间

见过的最大任务规模(如排序任务的序列长度),$Length_{test_max}$是测试期间可正确预测的最大任务规模.图 8 中,SORT 任务:seq2seq 模型泛化能力为 1,即泛化能力差只能对训练期间见过的序列长度正确排序,当测试样本略长于训练样本时准确率立即迅速下降);NPI 泛化能力为 3,即可正确完成 3 倍于训练样本长度的序列排序;HNCP 表现了最佳泛化能力(实验最多测试了 32 倍规模).ADD 任务:HNCP 表现最好,seq2seq 仍表现差,对于这个简单些的任务 NPI 泛化表现也很好.

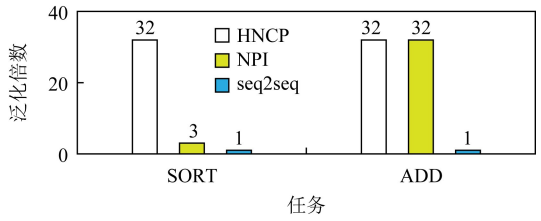


Fig. 8　Strong vs. weak generalization

图 8　泛化能力比较

实验表明,整合人工编程和神经网络自动生成构建程序生成模型有助于提升程序生成的泛化能力.

3.2　样本复杂度

模型达到最佳泛化能力所消耗的训练样本数量,该指标值越小表示模型学习性能越好.图 9 所示,对于 SORT 任务,本文 HNCP 模型消耗约 272 个样本,比 NPI 和 seq2seq 明显少;对于 ADD 任务,本文模型消耗约 320 个样本,比 NPI 和 seq2seq 模型需求明显少.该实验表明通过整合人工程序员和神经网络优势构建程序生成模型可有效提高学习效率.

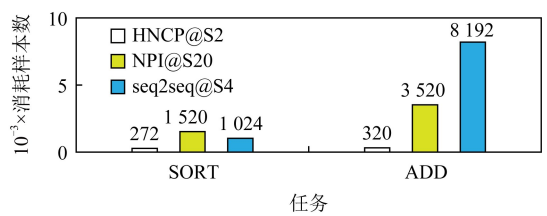


Fig. 9　Consumption of training samples

图 9　训练样本消耗量

注意,NPI 模型标注样本量是统计任务执行轨迹的步骤数,而不是直接计算输入-输出对.比如,SORT 任务样本数是$[m(m-1)/2]n$,ADD 任务样本数是$(10m)n$,其中 m 是问题规模(如排序序列长度),n 是任务个数.

3.3　生成程序准确率

生成程序准确率:模型生成的程序执行结果完全正确的任务样本占总测试样本的比例.

如图 10、图 11 所示,对于 ADD 和 SORT 任务,使用长度为 2 的样本训练的 HNCP 模型在比训练样本长很多的测试样本上可以获得 100% 的准确率,明显优于基线(NPI 和 seq2seq).测试发现 seq2seq(LSTM 实现)学习排序任务是很困难的,表现在当测试序列稍微比训练样本长时,准确率会立即快速下降.HNCP 只需用长度为 2 的序列训练程序框架,便成功地学到了排序行为,并很好地泛化到很长的任务序列.

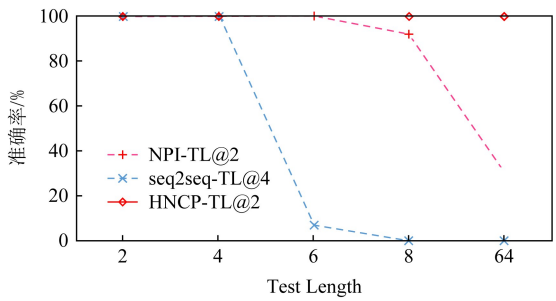


Fig. 10　Test accuracy of task ADD

图 10　加法任务生成程序测试准确率

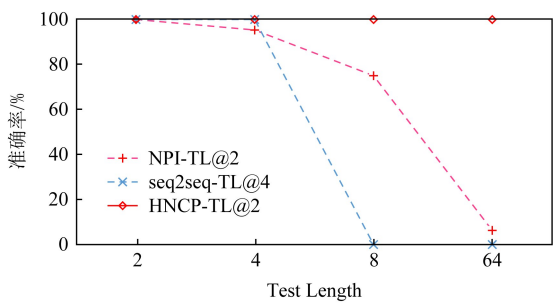


Fig. 11　Test accuracy of task BubbleSort

图 11　排序任务生成程序测试准确率

跟对标基准方法一样,这里使用测试准确率来评估生成程序的精确性,可以很好地跟同类研究做性能指标对比.当前神经网络可解释性是一个难题,因此使用神经网络构建程序生成模型的相关研究主要采用测试样例准确率来评估生成程序的精确性.所以不同于一般意义上的程序正确性证明,对于这类程序生成模型,做到类似“形式语义”等方式实现严格的程序正确性证明是困难的.

3.4　代表性程序生成模型指标对比

本节对主要的代表性程序生成模型,结合主要指标进行对比性汇总分析.如表 2 所示:

Table 2 Summaries of Indexes of Program Generation Models

表 2 程序生成模型主要指标对比汇总

模型	Gel	Ctl	Lan	Sam	Exe	Lrn	Rep
NPI[7]	Good	Yes	DSL	Medium	Yes	No	Net
seq2seq[22]	Poor	No	No	High	Yes	Yes	Net
BAYOU[12]		Yes	Java	High	No	No	Code
DeepCoder[2]		No	DSL	High	No	Yes	Code
NTM[3]	Average	No	No	High	Yes	Yes	Net
HNCP	Excellent	Yes	C	Low	Yes	Yes	Net

泛化能力(Gel):BAYOU 和 DeepCoder 预测生成的代码样例不能保证正确、可运行,也没有正确性自动检测方法,所以生成程序的泛化能力无法评估.其中 HNCP 是表现最好的,NPI 次之;复杂程序结构(Ctl):HNCP 和 BAYOU 表现最好,都支持循环、分支、函数等复杂控制结构,NPI 在自定义的 DSL 专用小语言上支持函数嵌套、分支、跳转控制结构;编程语言支持(Lan):HNCP 和 BAYOU 都支持通用编程语言,NPI 和 DeepCoder 支持自定义 DSL 专用语言,seq2seq 和 NTM 不支持编程语言;样本复杂度(Sam):该指标越低越好,HNCP 样本消耗最低,其他都消耗偏多,NPI 仅次于 HNCP,但其样本标注需要预先知道程序所有执行路径,所以标注代价偏高;可执行(Exe):BAYOU 和 DeepCoder 生成的程序不能直接执行;是否学习到新的程序行为(Lrn):根据是否需要提供完整的程序来区分,NPI 是对给出的程序路径的神经网络编码,BAYOU 需要收集大量功能相似的源码样例,剩下其他模型都可以从输入-输出样例对中学习到程序行为;学习生成的程序表示(Rep):Code 表示代码形式,Net 表示用神经网络编码形式.

综合来看 HNCP 表现出更好的程序生成学习优势,这表明结合程序框架(人工程序员创作)和输入-输出数据对学习(神经网络自动生成)探索构建程序生成模型的研究是有价值的.

3.5 讨 论

实验表明,本文方法跟同类代表性研究方法相比表现出更好的学习性能.结合模型与实验设计,我们有理由相信这种性能提升是因为融合了人工程序员和神经网络的协作,即模型将人工程序员的贡献整合进来发挥了作用;但因为当前神经网络模型的可解释性瓶颈,我们很难给出一个严格的证明.实验表明其性能提升明显、并能表达复杂程序结构,验证了这种整合模型的优势;而同时从某种意义上可能认为这种整合带来的性能优势是以牺牲全自动化程序生成为半自动化的便利性为代价(局限性),但这需要结合技术发展阶段综合考虑:首先,我们在引言中结合相关工作和观察分析探讨过,当前完全依赖神经网络的全自动或者全黑盒的程序生成模型在关键指标上表现不足(因此全自动的便利性是尚不存在的),在该背景下我们提出协作式模型并给出一种实现方法进行验证,实验表明这种融合程序员和神经网络的协助式程序生成模型表现出优势,结合分析表明这是有潜力、富有价值的研究思路;其次,考虑到程序生成总需要以一定的方式将需求提供给模型,结合引言中对程序员和神经网络各自的优势、不足的讨论,因此将程序员融合到自动化程序生成建模中是否是必要的,是一个开放性、多因素均衡问题,并且很可能会随着 AI 技术的发展阶段不同而需要做出不同选择.

关于实验评估测试任务的选择.主要选取对标基准模型(NPI,seq2seq)中用到的相同测试任务,这方便将我们的方法跟基准模型直接进行性能对比,其中 NPI 是这类模型中表现最好的,因此这种任务比较具有代表性;SORT,ADD 任务包含循环、函数调用、递归、分支、顺序执行等复杂控制结构,对于当前的程序自动生成相关研究具有足够复杂性和挑战性,能够将代表性模型(如基准模型和本文模型)的性能指标进行有效区分,另外这些程序结构足以表达其他比较复杂的任务.

4 总 结

经过学者们长期的努力,自动化程序生成研究取得了明显进步,但与图像、语音处理等相比,距实际应用仍有很大的可提升空间.本文提出了一种结合“人工编程”与“神经网络自动化程序生成”构建程序生成模型的研究方法.实验表明整合人工程序员

和神经网络各自优势的协作式程序生成模型可以明显提升程序生成的训练学习性能(泛化能力、准确率、样本复杂度),并适应复杂程序结构.因此,本文探讨的思路是可行且高效的,值得未来做更广泛的研究探索.

提出一种协作式程序生成建模方法并实验验证其技术上的可行性和有效性只是研究工作的一个方面,最终希望走向广泛的实际应用.而在工程应用中不仅关心技术可行性,还需考虑比如丰富并总结程序员跟神经网络的编程任务分工的规则、设计丰富的编程原语、正确性验证等,因此我们认为,下一步有必要对4项工作进行探讨:

1) 丰富分工协作规则并进行总结.比如可以从应用场景、任务类型出发,全面探讨和总结在程序员和神经网络间高效的编程任务分工规则.并将这些规则以程序模板、编程原语或用户编程接口等方式封装呈现.参照Java,C++等编程模型发展经验,这不是一个完全可正向评估的纯技术问题,往往依赖于Java,C++等用户生态的编程创造性的反馈和迭代,即具有开放性和创造性.

2) 如何划分协作分工是一个开放性问题.本文实验验证采用的是"程序员负责框架编制、神经网络负责局部细节生成"的分工方法,该方法在呈现协作式程序生成的概念形态、证明协作式模型的可行性、验证可取得程序生成学习性能的明显改进等方面具有优势,从而证明本文方法理念在意义上是足够的;但未必是最合理的分工方式,并且肯定不能覆盖全部的可能分工模式或规则.因此设计不同的分工协作方法,并为新分工方法构建可实现、可验证的模型具有很大的探索空间,比如在本文思路基础上,如果能够适当地设计扩展模型使得神经网络可在"程序框架的编制"任务中参与分工,将是对程序员与神经网络协作式编程模型很有意义的贡献.

3) 考虑到程序需求场景的复杂性.程序自动生成的应用不仅仅是某个或某几个技术点问题,而是涉及一系列技术集成,因此与大范围应用的图像处理、语音识别等相比,距离工程应用仍有差距.制约程序自动生成应用的因素很多,其中对需求进行建模并能恰当地跟神经网络衔接是关键因素之一;而本文探讨的程序员和神经网络协作式模型中,程序员提供的程序框架其实也属于一种需求描述规范的表达方式,如果沿着这个思路拓展并总结一套系统化需求描述规范,输入给融合的神经网络组件处理,我们相信这是值得进一步拓展探索的研究工作.

4) 生成程序的准确性证明.当前基于神经网络的程序生成相关研究主要通过测试准确率评估生成程序的精确性,对生成程序进行严格的准确性证明是一个富有挑战性的工作.

参　考　文　献

[1] Manna Z, Waldinger R J. Toward automatic program synthesis [J]. Communications of the ACM, 1971, 14(3): 151-165

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

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

[4] Frankle J, Osera P M, Walker D, et al. Example-directed synthesis: A type-theoretic interpretation [J]. ACM SIGPLAN Notices, 2016, 51(1): 802-815

[5] Feser J K, Chaudhuri S, Dillig I. Synthesizing data structure transformations from input-output examples [J]. ACM SIGPLAN Notices, 2015, 50(6): 229-239

[6] Devlin J, Uesato J, Bhupatiraju S, et al. Robustfill: Neural program learning under noisy I/O [J]. arXiv preprint, arXiv: 1703.07469, 2017

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

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

[9] Nguyen A T, Hilton M. API code recommendation using statistical learning from fine-grained changes [C] //Proc of the 24th Int Symp on Foundations of Software Engineering. New York: ACM, 2016: 511-522

[10] Polosukhin I, Skidanov A. Neural program search: Solving programming tasks from description and examples [J]. arXiv preprint, arXiv:1802.04335, 2018

[11] Dagenais B, Robillard M P. Recommending adaptive changes for framework evolution [J]. ACM Transactions on Software Engineering and Methodology, 2011, 20(4): 1-35

[12] Murali V, Qi Letao, et al. Neural sketch learning for conditional program generation [J]. arXiv preprint, arXiv: 1703.05698, 2017

[13] Lin Zeqi, Zhao Junfeng, Xie Bing. A graph database based method for parsing and searching code structure [J]. Journal of Computer Research and Development, 2016, 53(3): 531-540(in Chinese)
(林泽琦, 赵俊峰, 谢冰. 一种基于图数据库的代码结构解析与搜索方法[J]. 计算机研究与发展, 2016, 53(3): 531-540)

[14] Van Deursen A, Klint P. Domain-specific language design requires feature descriptions [J]. Journal of Computing and Information Technology, 2002, 10(1): 1-17

[15] Bošnjak M, Rocktäschel T, Naradowsky J, et al. Programming with a differentiable forth interpreter [C] // Proc of the 34th Int Conf on Machine Learning. Cambridge, MA: JMLR, 2017: 547-556

[16] Solar-Lezama A, Bodik R. Programsynthesis by sketching [D]. Berkeley: University of California, Berkeley, 2008

[17] Sukhbaatar S, Weston J, Fergus R. End-to-end memory networks [C] //Proc of Advances in Neural Information Processing Systems. Cambridge, MA: MIT Press, 2015: 2440-2448

[18] Joulin A, Mikolov T. Inferring algorithmic patterns with stack-augmented recurrent nets [C] //Advances in Neural Information Processing Systems. Cambridge, MA: MIT Press, 2015: 190-198

[19] Koopman Jr P J. A brief introduction to Forth [J]. ACM SIGPLAN Notices, 1993, 28(3): 357-358

[20] Kogge P M. The Architecture of Symbolic Computers [M]. New York: McGraw-Hill, Inc, 1990

[21] Nepomniaschy V A, Anureev I S, Promskii A V. Towards verification of C programs: Axiomatic semantics of the C-kernel language [J]. Programming and Computer Software, 2003, 29(6): 338-350

[22] Delsarte P. Bilinear forms over a finite field, with applications to coding theory [J]. Journal of Combinatorial Theory: Series A, 1978, 25(3): 226-241

[23] Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with neural networks [C] //Proc of Advances in Neural Information Processing Systems. Cambridge: MIT Press, 2014: 3104-3112

Zhou Peng, born in 1984. PhD. His main research interests include intelligent software, operating system, virtualization, system security.
周 鹏,1984 年生.博士.主要研究方向为智能软件、操作系统、虚拟化、系统安全等.

Wu Yanjun, born in 1979. PhD, professor, PhD supervisor. His main research interests include operating system and system security.
武延军,1979 年生.博士,研究员,博士生导师.主要研究方向为操作系统、系统安全.

Zhao Chen, born in 1967. PhD, professor, PhD supervisor. His main research interests include compiling, auto-testing, operating system, and networking software.
赵 琛,1967 年生.博士,研究员,博士生导师.主要研究方向为编译、自动化测试、操作系统和网络软件等.

附录 A 指令集的可微分实现

1 扩展内存访问指令

扩展内存($\boldsymbol{S},\boldsymbol{R},\boldsymbol{H}$)在逻辑上以内存矩阵表示,以索引指针提供内存矩阵的访问地址.因此内存区可以用 2 个分量的元组记为($\boldsymbol{B}_k,\boldsymbol{P}_k$),$k\in\{\boldsymbol{S},\boldsymbol{R},\boldsymbol{H}\}$,其中 $\boldsymbol{B}_k\in\mathbb{R}^{V\times L}$,$\boldsymbol{P}_k\in\mathbb{R}^{L}$,$L$ 是内存矩阵长度,V 是每个值元素向量的宽度(值向量的分量个数);为了便于数据交换 $\boldsymbol{S},\boldsymbol{R},\boldsymbol{H}$ 统一值宽度为 V,对于栈 $\boldsymbol{P}_k$ 是栈顶指针,对于堆 $\boldsymbol{P}_k$ 是变量地址.

$$READ(\boldsymbol{B}_k,\boldsymbol{P}_k)\leftarrow\boldsymbol{P}_k^{\mathrm{T}}\boldsymbol{B}_k,$$
$$\sum_{i=1}^{L}P_k^i=1,\ 0\leqslant P_k^i\leqslant 1. \tag{A1}$$

读内存操作的可微分实现,其中 P_k^i 是指针向量 $\boldsymbol{P}_k$ 的第 i 个分量,表示在内存区的第 i 个位置的访问权重,$\boldsymbol{P}_k$ 采用类似 attention 机制做内存读,可用 softmax 实现,显然 $READ$ 的实现对 $\boldsymbol{P}_k$ 和 $\boldsymbol{B}_k$ 是可微分的.

$$WRITE(\boldsymbol{x},\boldsymbol{B}_k,\boldsymbol{P}_k)\leftarrow\boldsymbol{B}_k\odot(1-\boldsymbol{P}_k\boldsymbol{1}^{\mathrm{T}})+\boldsymbol{P}_k\boldsymbol{x}. \tag{A2}$$

写内存操作的可微分实现,其中 $\boldsymbol{x}$ 是值向量,$\boldsymbol{1}$ 是元素全为 1,形状为(V,1)的列向量.

$$\vec{S}_{[i,j]}(1)=\begin{cases}1, & \text{若 } (i+1)\%L\equiv j,\\ 0, & \text{其他.}\end{cases} \tag{A3}$$

$$\overleftarrow{S}_{[i,j]}(1)=\begin{cases}1, & \text{若 } (i-1)\%L\equiv j,\\ 0, & \text{其他.}\end{cases} \tag{A4}$$

分别是向量循环右移 1 位、循环左移 1 位的变换矩阵.

$$\boldsymbol{P}_k{+}{+}\leftarrow\boldsymbol{P}_k^{\mathrm{T}}\vec{S}(1). \tag{A5}$$

$$\boldsymbol{P}_k{-}{-}\leftarrow\boldsymbol{P}_k^{\mathrm{T}}\overleftarrow{S}(1). \tag{A6}$$

分别是内存地址指针右移、左移 1 个位置.

$$PUSH_k(\boldsymbol{x})\leftarrow\{\boldsymbol{P}_k=\boldsymbol{P}_k{+}{+},$$
$$WRITE(\boldsymbol{x},\boldsymbol{B}_k,\boldsymbol{P}_k)\}. \tag{A7}$$

$$POP_k=\{\boldsymbol{r}=READ(\boldsymbol{B}_k,\boldsymbol{P}_k),\boldsymbol{P}_k=\boldsymbol{P}_k{-}{-}\}. \tag{A8}$$

2 算术操作指令

$$f_{\text{one-hot}}(i):i\rightarrow\boldsymbol{n}^i,i\in\mathbb{N}^0,$$
$$\boldsymbol{n}^i[j]=\begin{cases}1, & \text{若 } j\equiv i,\\ 0, & \text{其他.}\end{cases} \tag{A9}$$

整数的 one-hot 编码,其中 i 是非负整数,$\boldsymbol{n}^i$ 是行向量.

$$A^{o}_{[i,j,k]}=\begin{cases}1, \text{若 } (i\ o\ j)\equiv k\ \%V,\\ 0, \text{其他.}\end{cases}$$

$$o\in\{\text{add},\text{sub},\text{mul},\text{div}\}. \tag{A10}$$

通过构造 3-D 变换矩阵实现矩阵算术操作,相关理论指导可参看双线性映射数学体系.实现向量算术运算的 3-D 循环变换矩阵 $\boldsymbol{A}^{o}$ 是形状为(V,V,V)的 3-D 矩阵.

$$AOP^{o}(\boldsymbol{a},\boldsymbol{b})\leftarrow\boldsymbol{a}^{\mathrm{T}}\boldsymbol{A}^{o}\boldsymbol{b},o\in\{\text{add},\text{sub},\text{mul},\text{div}\}. \tag{A11}$$

$$f(+)\leftarrow\{PUSH_{\boldsymbol{S}}(AOP^{\text{add}}(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A12}$$

$$f(-)\leftarrow\{PUSH_{\boldsymbol{S}}(AOP^{\text{sub}}(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A13}$$

$$f(*)\leftarrow\{PUSH_{\boldsymbol{S}}(AOP^{\text{mul}}(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A14}$$

$$f(/)\leftarrow\{PUSH_{\boldsymbol{S}}(AOP^{\text{div}}(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A15}$$

3　关系操作指令

$$\varphi(x):x\rightarrow\min(\max(0,kx),1),x\in\mathbb{R},k\in\mathbb{R}_{+}. \tag{A16}$$

式(A16)是分段线性函数,将实数空间$\mathbb{R}$映射到[0,1]空间,其中 k 系数控制了函数图形的陡峭程度.

$$GT(\boldsymbol{a},\boldsymbol{b})\leftarrow\{p=\varphi((\boldsymbol{w}_{\text{one-hot}}\odot(\boldsymbol{a}-\boldsymbol{b}))\boldsymbol{1}^{\mathrm{T}}),\ p\odot\boldsymbol{one}+(1-p)\odot\boldsymbol{zero}\}. \tag{A17}$$

其中,$\boldsymbol{one}$,$\boldsymbol{zero}$ 分别是 1,0 的 one-hot 向量表示,$\boldsymbol{w}_{\text{one-hot}}$的形状为$(V,1)$的向量,$(\boldsymbol{w}_{\text{one-hot}})_i=i,i\in\mathbb{N}+$.

$$EQ(\boldsymbol{a},\boldsymbol{b})\leftarrow\{p=\varphi((\boldsymbol{a}\odot\boldsymbol{b})\boldsymbol{1}^{\mathrm{T}}),\ p\odot\boldsymbol{one}+(1-p)\odot\boldsymbol{zero}\}. \tag{A18}$$

$$LT(\boldsymbol{a},\boldsymbol{b})\leftarrow GT(\boldsymbol{b},\boldsymbol{a}). \tag{A19}$$

$$LEQ(\boldsymbol{a},\boldsymbol{b})\leftarrow LT(\boldsymbol{a},\boldsymbol{b})+EQ(\boldsymbol{a},\boldsymbol{b}). \tag{A20}$$

$$NEQ(\boldsymbol{a},\boldsymbol{b})\leftarrow LT(\boldsymbol{a},\boldsymbol{b})+GT(\boldsymbol{a},\boldsymbol{b}). \tag{A21}$$

$$GEQ(\boldsymbol{a},\boldsymbol{b})\leftarrow GT(\boldsymbol{a},\boldsymbol{b})+EQ(\boldsymbol{a},\boldsymbol{b}). \tag{A22}$$

$$f(>)\leftarrow\{PUSH_{\boldsymbol{S}}(GT(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A23}$$

$$f(<)\leftarrow\{PUSH_{\boldsymbol{S}}(LT(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A24}$$

$$f(==)\leftarrow\{PUSH_{\boldsymbol{S}}(EQ(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A25}$$

$$f(\leqslant)\leftarrow\{PUSH_{\boldsymbol{S}}(LEQ(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A26}$$

$$f(\neq)\leftarrow\{PUSH_{\boldsymbol{S}}(NEQ(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A27}$$

$$f(\geqslant)\leftarrow\{PUSH_{\boldsymbol{S}}(GEQ(POP_{\boldsymbol{S}},POP_{\boldsymbol{S}}))\}. \tag{A28}$$

4　变量存访指令

$$SAVE\leftarrow\{\boldsymbol{a}=POP_{\boldsymbol{S}},\boldsymbol{v}=POP_{\boldsymbol{S}},\ WRITE(\boldsymbol{v},\boldsymbol{B}_{\boldsymbol{H}},\boldsymbol{a})\}. \tag{A29}$$

$$LOAD\leftarrow\{\boldsymbol{a}=READ(\boldsymbol{B}_{\boldsymbol{S}},\boldsymbol{P}_{\boldsymbol{S}}),\ WRITE(READ(\boldsymbol{B}_{\boldsymbol{H}},\boldsymbol{a}),\boldsymbol{B}_{\boldsymbol{S}},\boldsymbol{P}_{\boldsymbol{S}})\}. \tag{A30}$$

5　分支控制指令

$$DECDIGIT(\boldsymbol{x}):\boldsymbol{x}\rightarrow(\boldsymbol{w}_{\text{one-hot}}\odot\boldsymbol{x})\boldsymbol{1}^{\mathrm{T}}. \tag{A31}$$

式(A31)是将 one-hot 编码整数转换到十进制编码的转换函数.

$$CALL(L)\leftarrow\{PUSH_{\boldsymbol{R}}(\boldsymbol{P}_{\mathrm{C}}++),\boldsymbol{P}_{\mathrm{C}}=f_{\text{one-hot}}(L)\}. \tag{A32}$$

$$RET\leftarrow\{\boldsymbol{P}_{\mathrm{C}}=POP_{\boldsymbol{R}}\}. \tag{A33}$$

$$NEXT\leftarrow\{\boldsymbol{P}_{\mathrm{C}}++\}. \tag{A34}$$

$$GOTO(L)\leftarrow\{\boldsymbol{P}_{\mathrm{C}}=f_{\text{one-hot}}(L)\}. \tag{A35}$$

$$GOTOF(L)\leftarrow\{p=DECDIGIT(POP_{\boldsymbol{S}}),\ \boldsymbol{P}_{\mathrm{C}}=(1-p)f_{\text{one-hot}}(L)+p(\boldsymbol{P}_{\mathrm{C}}^{\mathrm{T}}\vec{\boldsymbol{S}}(1))\}. \tag{A36}$$

有条件跳转,可用于 if..then..else 的可微分实现.

$$DO\leftarrow\{\boldsymbol{P}_{\boldsymbol{R}}=\boldsymbol{P}_{\boldsymbol{R}}{}^{\mathrm{T}}\vec{\boldsymbol{S}}(2),WRITE(POP_{\boldsymbol{S}},\boldsymbol{B}_{\boldsymbol{R}},\boldsymbol{P}_{\boldsymbol{R}}),\ WRITE(POP_{\boldsymbol{S}},\boldsymbol{B}_{\boldsymbol{R}},\boldsymbol{P}_{\boldsymbol{R}}\vec{\boldsymbol{S}}(1)),NEXT\}. \tag{A37}$$

$$LOOP(L)\leftarrow\{WRITE(READ(\boldsymbol{B}_{\boldsymbol{R}},\boldsymbol{P}_{\boldsymbol{R}})\vec{\boldsymbol{S}}(1),\boldsymbol{B}_{\boldsymbol{R}},\boldsymbol{P}_{\boldsymbol{R}})\ p=DECDIGIT(EQ(READ(\boldsymbol{B}_{\boldsymbol{R}},\boldsymbol{P}_{\boldsymbol{R}}),\ READ(\boldsymbol{B}_{\boldsymbol{R}},\boldsymbol{P}_{\boldsymbol{R}}\vec{\boldsymbol{S}}(1))))\ \boldsymbol{P}_{\mathrm{C}}=(1-p)f_{\text{one-hot}}(L)+p(\boldsymbol{P}_{\mathrm{C}}^{\mathrm{T}}\vec{\boldsymbol{S}}(1))\}. \tag{A38}$$

$$ENDLOOP\leftarrow\{\boldsymbol{P}_{\boldsymbol{R}}=\boldsymbol{P}_{\boldsymbol{R}}^{\mathrm{T}}\vec{\boldsymbol{S}}(2),NEXT\}. \tag{A39}$$

式(A37)～(A39)用于 do..loop 和 while 循环可微分实现.

$$HALT\leftarrow\{\boldsymbol{P}_{\mathrm{C}}=\boldsymbol{P}_{\mathrm{C}}\}. \tag{A40}$$

6　其他机器指令

$$\text{INC}\leftarrow\{WRITE(READ(\boldsymbol{B}_{\boldsymbol{S}},\boldsymbol{P}_{\boldsymbol{S}})\vec{\boldsymbol{S}}(1),\boldsymbol{B}_{\boldsymbol{S}},\boldsymbol{P}_{\boldsymbol{S}})\}. \tag{A41}$$

$$DEC\leftarrow\{WRITE(READ(\boldsymbol{B}_{\boldsymbol{S}},\boldsymbol{P}_{\boldsymbol{S}})\vec{\boldsymbol{S}}(1),\boldsymbol{B}_{\boldsymbol{S}},\boldsymbol{P}_{\boldsymbol{S}})\}. \tag{A42}$$

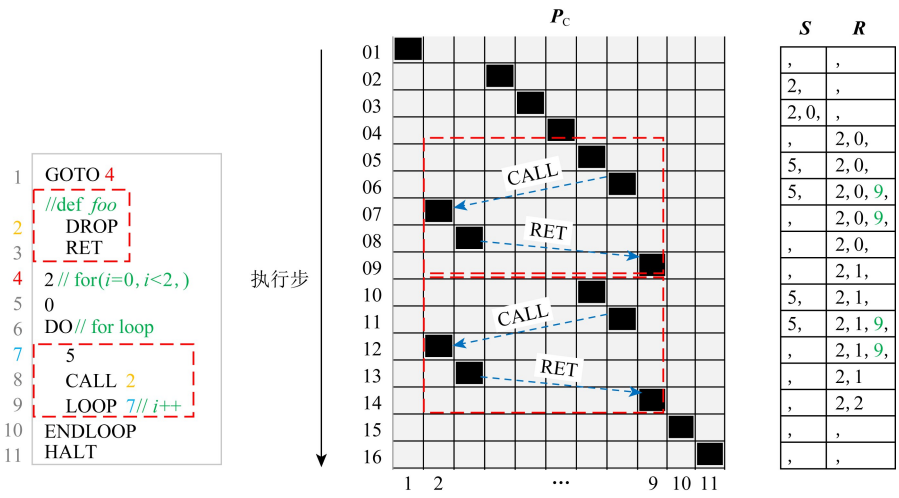
$$SWAP\leftarrow\{\boldsymbol{a}=POP_{\boldsymbol{S}},\boldsymbol{b}=POP_{\boldsymbol{S}},\ PUSH_{\boldsymbol{S}}(\boldsymbol{b}),PUSH_{\boldsymbol{S}}(\boldsymbol{a})\}. \tag{A43}$$

附录 B　dVM 运行时过程示例

图 B1 示例是 IR 程序在 dVM 上执行过程,记录程序每步指令执行后 dVM 的$(\boldsymbol{S},\boldsymbol{R})$内存区值和 $\boldsymbol{P}_{\mathrm{C}}$ 指针变化轨迹.图 B1(a)是该示例测试程序,主要由循环结构和循环内嵌函数调用构成,包含了循环、函数调用、函数返回、无条件跳转、顺序执行等复杂控制结构.这些程序结构足以表达复杂程序;$\boldsymbol{P}_{\mathrm{C}}$ 是程序指针轨迹(one-hot),$\boldsymbol{S}$ 和 $\boldsymbol{R}$ 分别是相应的求值栈和返回栈内容变化,随着程序执行其记录值数据的可视化轨迹于图 B1(b)(c)所示.该程序一共执行了 16 步,执行了 2 次循环,循环于图 B1(b)中虚框所示,对应于图 B1(a)虚框内代码,执行了 2 次函数调用-返回结构(CALL 保存在 $\boldsymbol{R}$ 栈中的返回地址 9 被相应的 RET 获得,实现返回现场),最后执行到停机指令正常终止.

附录 C　BNF 语义规则的实现示例

图 C1 所示是语义规则的一个实现示例.



(a) 控制结构例子 (b) P_C轨迹 (c) S&R栈

Fig. B1 An illustration of program execution on dVM

图 B1 dVM 程序执行过程的示例

def_func〉::=〈id '('〈parm〉')''{' 〈decl_var〉〈stmt〉'}'〉语义规则为例:

```
〈def_func〉::=id '('{curFuncName=id.name;var_count=0;} 〈parm〉')' /* 设置当前函数名和局部变量个数计数器 */
  '{'〈decl_var〉{printf("%s!\\n",〈parm〉.name);} /* 栈传值方式初始化函数参数 */
  〈stmt〉'}'{printf(";\\n",);} /* 函数定义结尾 */
  |ε{var_count=0;} /* 当前函数定义结束,将局部变量计数重置为 0 以备下一个函数定义使用 */
```

Fig. C1 An implementation example of semantic rules

图 C1 语义规则实现示例

附录 D 程序任务样例

ADD 和 SORT 任务分别如图 D1 和图 D2 所示:

```
int c=0,s,sum[10];
/* n: 被加数位数,p: 当前加法的位置 */
int lt[2]={4,1},rt[2]={5,9},n=2,p=0; /* 用训练样本初始
  化这些变量(状态) */
int multiAdd(){
  if (p==n){
    return;
  else
    /* 由训练数据决定的行为 */
〈%s=encoder(observe(lt[p],rt[p],c)):transform(linear(30)):
 decoder(choose(0,1,2,3,4,5,6,7,8,9))%〉;
〈%c=encoder(observe(lt[p],rt[p],c)):transform(linear(10)):
 decoder(choose(0,1))%〉;
    sum[p]=s;
    p++;
    multiAdd();
    }
  }
multiAdd();
print(c,sum).
```

Fig. D1 ADD for elementary multi digit addition

图 D1 多位数初等加法

```
#define size 10
int a[size]={2,4,9,1,7,8,6,9,3,5}; /* 用训练样本初始化
  这些变量(状态) */
void bubble(int pass,int c){
  if (c≥size-pass-1)
    return;
  /* 由训练数据决定的行为 */
〈%encoder(observe(a[c+1],a[c])):transform(tanh(),
 sigmoid(),linear(6)):decoder(choose(swap(a[c+1],
 a[c]),noop))%〉;
  bubble(pass,c+1);
}
bubbleSort(){
  int i=0;
  while(i<size-1){
    bubble(i,0);
    i++;
  }
}
bubbleSort().
```

Fig. D2 SORT for bubblesort

图 D2 冒泡排序