软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software, [doi: 10.13328/j.cnki.jos.000000]

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

dAMP:可微分抽象机混合编程系统*

周 鹏[1,2], 武延军[1], 赵 琛[1]

[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)

通讯作者: 周鹏, E-mail: zhoupeng@iscas.ac.cn

摘 要: 自动化编程是智能软件的核心挑战之一,使用程序执行轨迹或输入输出样例学习程序是自动化编程典型研究方法,这些方法无法弥合常规程序元素与神经网络组件间的隔阂,不能吸收经验信息输入、缺乏编程控制能力.本文给出了一种可无缝结合高级编程语言与神经网络组件的混合编程模型:使用高级编程语言元素和神经网络组件元素混合开发应用程序,其中编程语言描述程序的框架、提供经验信息,关键复杂部分则用未定、可学习的神经网络组件占位,应用程序在可微分抽象机上运行生成程序的连续可微分计算图表示,然后使用输入输出数据通过可微分优化方法对计算图进行训练,学习程序的未定部分,自动生成完整的确定性程序.可微分抽象机混合编程模型给出了一种能够将编程经验与神经网络自学习相结合的程序自动生成方法,弥合编程语言元素与神经网络元素间隔阂,发挥并整合高级过程化编程和神经网络可训练学习编程各自的优势,将复杂的细节交给神经网络未定部分自动生成降低编程难度或工作量,而适当的经验输入又有助于未定部分的学习,同时为复用长期积累的宝贵编程经验提供输入接口.

关键词: 智能软件;可微分编程;可微分抽象机;混合编程;人工智能

中图法分类号: TP204

中文引用格式: 周鹏,武延军,赵琛.dAMP:可微分抽象机混合编程系统.软件学报. http://www.jos.org.cn/1000-9825/ 0000.htm

英文引用格式: Zhou P, Wu YJ, Zhao C. dAMP: System of Hybrid Programming on Differentiable Abstract Machines. Ruan Jian Xue Bao/Journal of Software, 2018 (in Chinese). http://www.jos.org.cn/1000-9825/0000.htm

dAMP: System of Hybrid Programming on Differentiable Abstract Machines

ZHOU Peng[1,2], WU Yan-JUN[1], Zhao Chen[1]

[1](Institute of Software, Chinese Academy of Sciences, Beijing 100190, China)

[2](University of Chinese Academy of Sciences, Beijing 100049, China)

Abstract: Automated programming is one of the central challenges of intelligent software. Learning program by program execution traces or input output pairs are typical automatic programming research methods, but these methods can not bridge the gap between normal program elements and neural network components, can not absorb programing experience as input, and lack of programming control interface. In this paper we present a hybrid programming model that seamlessly combines advanced programming language with neural network components: The program is composed by using a mixture of elements from high-level programming language and neural network component, in which the language describes the sketch to provide experience information, with the key complex parts placed with undetermined and learnable neural network components. The program runs on differentiable abstraction machines to generate its continuous differentiable computational graph representation. Then, using program input-output pairs to train the graph by differentiable optimization method to learn to generate the complete program automaticly. This programming model provides an automatic program generation method which can combine

* 基金项目: 中国科学院战略性先导科技专项, 自完善智能操作系统(A 类)
Foundation item: Chinese Academy of Sciences Strategic Pilot Science and Technology Project, SIIOS(Class A)
收稿时间: 0000-00-00; 修改时间: 0000-00-00; 采用时间: 0000-00-00; jos 在线出版时间: 0000-00-00
CNKI 在线出版时间: 0000-00-00

programmer experience with neural network self-learning, bridges the gap between elements from programming language and neural network, which integrate the advantages of procedural and neural network programming, the complex details are automatically generated by neural networks to reduce the difficulty or workload of programming. Experience input is heuristic-helpful to the learning of undetermined parts and provides an input interface for reusing valuable programming experience accumulated over a long period of time.

Key words: intelligent software;differentiable programming;differentiable abstract machines;hybrid programming;artificial intelligence

自计算机诞生以来,实现帮助计算机自动编写程序的系统一直是人工智能研究者所追求的重要目标之一,相关研究最早可追溯到 1971 年[1].机器编程的研究总体分为类自然语言代码挖掘和代码生成两大类,代码生成的研究又可划分为程序空间搜索和连续可微分优化法.近年来随着深度学习的兴起,结合神经网络以连续可微分优化理论为基础构建编程模型的研究也成为一个研究热点[2,3,4],总体而言,从输入素材角度可将这些神经网络类编程模型分为基于程序执行轨迹样例的方法[2]和基于程序输入输出数据样例的方法[3,4].

当前基于执行轨迹的研究方法需要收集逐步的执行轨迹有监督数据,数据成本高,并且该模型学习的本质是通过对不同执行路径的监督数据学习形成算法的神经网络编码,而覆盖不同路径的收集本身意味着要求算法是明确的,因此严格意义上,该模型并没有学习到新的代码,基于输入输出数据样例的方法直接提供输入输出数据训练集,让神经网络尝试从数据中学习从输入转换为输出的规律,该方法面临的问题是从输入到输出的转换规则可能存在多种,即存在有歧义性,并且该方法会面临编程语言过多的规则、实现细节带来的学习复杂度,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、计算机原理教程)导致模型处理信息不完备,也不能为训练学习过程提供珍贵的编程经验,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,导致习得算法通常非常简单,比如很难处理分支、递归等复杂控制结构,并且泛化能力明显不足.因此,当前单纯的以代码或输入输出数据为输入构建的神经网络编程模型都存在局限性.我们认为在程序生成或程序语义处理问题上应该跳出编程者视角,尽可能提升习模型建模的抽象层次,在更具抽象层面用公里语义的方式将程序描述为抽象机的状态转换,从而避免从操作语义角度去描述程序的执行语义而陷入编程语言的复杂规范、底层实现细节等操作层面,提升程序处理的抽象层次是本文整个研究的基本动机,同时我们认为程序生成的研究不能单纯的只从代码或数据中学习,进一步的研究需要整合软件工程中长期积累的宝贵编程经验为学习过程提供指导,而本文的研究也说明提升模型的抽象层次是支持输入并整合编程经验的一个有效途径.

本文研究了一种可将高级编程语言与神经网络组件无缝结合的混合式编程模型(简称 dAMP),该模型的优点是可以同时使用过程化的高级编程语言元素和神经网络组件元素混合开发应用程序,从而可以弥合两种元素的隔阂,发挥并整合高级过程化编程和神经网络可训练学习编程各自的优势,并为基于神经网络的程序自动生成模型提供了经验知识的输入途径.dAMP 的研究的基本出发点是从抽象层次研究程序生成问题,其整体思路是:首先从纯机器演算角度构造一个通用的抽象机(AM),用公里语义的方式将程序执行描述为抽象机的状态转换, AM 有独立的对外界无依赖的状态存储和指令集,支持传统的高级编程语言过程化编程,然后以某种方式将 AM 实现为连续可微分抽象机 dAM,该实现方式跟典型的神经网络实现是同质的,因此可以无缝对接神经网络组件,借助 dAM,高级过程化程序可以在 dAM 上以连续可微分的方式直接执行,dAMP 混合编程是研究如何编写程序框架提供经验信息、如何在程序中嵌入神经网络组件为 dAM 编写程序框架,以及如何使用训练数据学习生成完整的程序,抽象机的可选取原型目标很多,这里选取 Forth[5,6]编程语言执行模型作为构造抽象机的目标.本文会对 dAM 的实现方式做必要够用的介绍,但重点是给出 dAMP 混合编程的关键实现技术,并对 dAMP 程序自动生成的学习机制做详细深入的实验分析.

本文主要贡献:1)提出在抽象层研究程序自动生成问题的一种思路,并给出该思路的一个完整验证系统 dAMP;2)dAMP 给出了一种支持经验输入、编程语言跟神经网络组件元素相结合的混合编程模型,并给出了其设计实现的关键技术;3)给出了丰富的实验验证,对 dAMP 程序自动生成的学习机制做了深入的实验分析.

本文第 1 节对 dAMP 框架做一个整体介绍.第 2 节介绍编程语言模型,包括理解本文需要预备的编程语言概念、语言运行模型知识.第 3 节介绍连续可微分抽象机实现方法.第 4 节详细展开实现 dAMP 的关键技术.第 5 节实验验证并对模型的学习机制做深入的实验分析.第 6 节介绍相关性工作.第 7 节结束语.

1 整体架构

可微分抽象机混合编程系统(dAMP)总体可分为源程序、编译与处理、连续可微分抽象机和计算图四个主要组成部分,如图 1 所示.

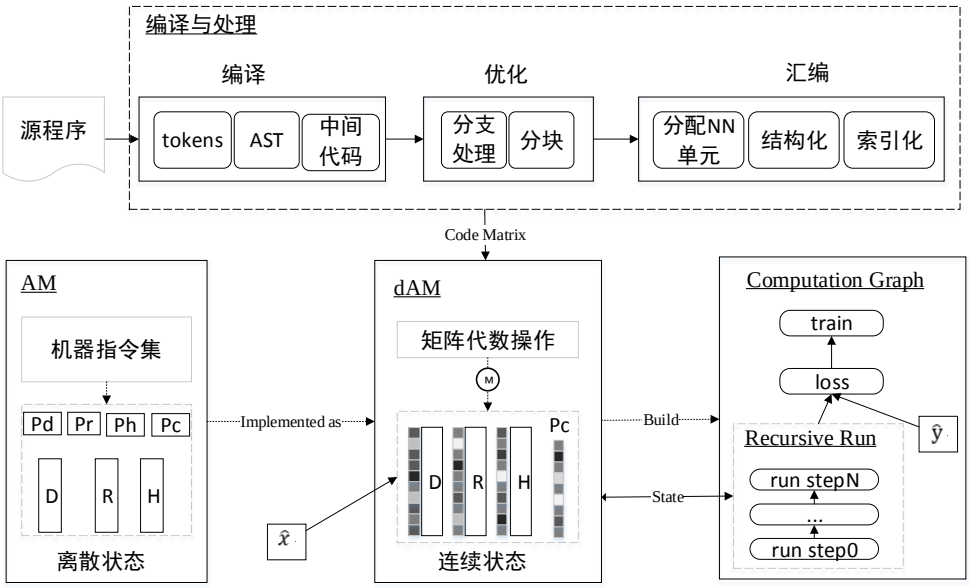


Fig.1 dAMP architecture

图 1 dAMP 整体架构

总体而言,图 1 的上半部分是对源程序处理,下半部分是程序的运行时环境和训练学习环境,源程序首先经过编译形成中间代码,然后对中间代码做优化处理和汇编处理生成结构化的适合连续可微分执行的代码矩阵表示,接着连续可微分抽象机(dAM)加载训练样本的输入部分$\hat{x}$到状态缓存的数据栈并执行代码矩阵,而执行的结果便是构建生成程序执行的计算图表示,其中计算图的程序执行部分总体是 N 步递归结构,最后以 dAM 的状态为输入执行计算图,并以训练样本的标记部分$\hat{y}$为输入计算 loss、以梯度下降可微分优化的方式进行训练学习.图中实箭头连线是数据流,虚箭头连线是控制依赖.下面对四个主要组成部分分别做介绍:

(1) 源程序.

使用 Forth 高级编程语言和神经网络组件元素(简称 nc)混合编写的程序,跟常规编程需要编写程序的完整细节不同,dAMP 编程程序员可以基于编程经验只描述程序的框架,而把一些细节用待学习的神经网络组件(nc)占位,通过框架和 nc 结构提供经验信息.

(2) 编译与处理.

把源程序转换为更适合 dAM 高效处理的形式,包括编译、优化和汇编三个阶段.编译将源程序翻译为中间代码,包括词法分析、语法树生成以及生成中间代码,在中间代码生成期间会额外做一些宏替换、控制语句处理等操作,比如将循环、条件分支、函数调用等都统一转换为标签跳转、统一分配标签等;优化主要包括对分支的处理和代码划块,分支的处理主要按照控制语句标签定义出现顺序进行序列化编码,实现标签的表格化管理,代码划块主要是将程序代码划分为若干粗粒度的顺序代码块,便于后面将程序的状态转移由行粒度压缩到块粒度,从而显著提升执行效率和模型训练效果;汇编主要工作是将优化后的中间转换为可以在 dAM 上执行的代码矩阵,包括为神经网络占位符分配神经网络单元、代码的表格结构化管理、控制语句地址标签的表格结构化管理、以及在结构化基础上对代码和地址的索引化引用,并将索引化引用以代码原地(inplace)修改或替换方式反

馈到代码中,结构化和索引化为在 dAM 上做连续可微分转化和神经网络学习模型构建做准备.

(3) 连续可微分抽象机.

连续可微分抽象机 dAM 以代码矩阵和训练样本的输入部分$\hat{x}$为输入,将程序以逐步的连续状态转换方式解释执行,其执行结果是生成对应的程序计算图表示.dAM 是使用矩阵代数对 Forth 语言执行抽象机模型(AM)的连续可微分表示,主要包括状态操作和指令集的连续可微分实现,因为跟神经网络一样都使用矩阵代数实现形式,因此在 dAM 上高级编程语言元素跟神经网络单元元素能无缝结合.

(4) 计算图.

计算图是程序生成模型的最终表示,在 dAM 执行代码矩阵期间构建生成,主要包括程序递归执行、学习目标 loss 损失函数、训练函数 train 三个部分,其中程序递归执行逻辑表达为递归访问抽象机状态空间的 RNN 网络,训练期间训练样本的标记部分$\hat{y}$为输入.

2 编程语言模型介绍

本节介绍文中用到的预备知识编程语言模型,主要包括 Forth 语言元素、语言运行模型,其中语言元素只介绍跟文中样例代码相关的内容,如果想详细了解语言细节请阅读参考文献[7].

2.1 语言执行模型

Forth 是一种栈式语言,随着时间推进,其执行虚拟机模型有多个变体,本文采用一个比较经典而简介的双栈模型,如图 2 所示,执行虚拟机模型[6]包括一个求值数据栈 Data Stack、一个子程序(函数)返回栈 Return Stack、一个算术逻辑运算单元 ALU 和一个随机内存存储 Main Memory(其中包括以栈的方式管理帧,在本文不是必须的,忽略),另外包括 5 个内存访问寄存器:PC 程序计数器、PSP 参数栈指针(数据栈栈顶指针)、RSP 返回栈指针、FP 帧指针、FEP 帧尾指针(FP、和 FEP 可忽略).注意,虽然 Return Stack 返回栈从名字上看是保存子程序调用返回地址,实际上它还被用于临时数据交换.

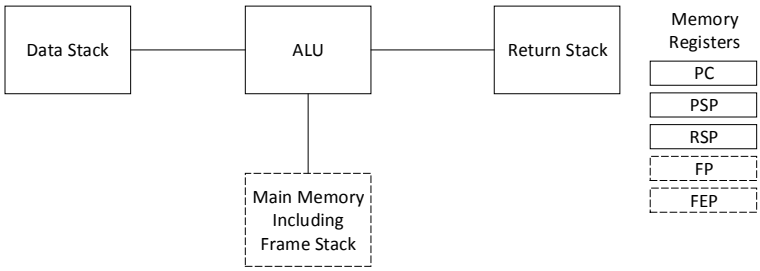


Fig.2 Forth VM

图 2 Forth 虚拟机模型

Forth 表达式采用逆波兰式(RPN)表示,因此这种双栈模型非常适合 Forth 程序的高效执行,图 3 以一个简单的例子 2+4=6 说明 Forth 语言执行模型:

2 4 + . 6 → 1. *t1=D.pop*(), *t2=D.pop*()
2. *t3=t1+t2, D.push*(*t3*)

Fig.3 Example of expression evaluation

图 3 表达式求值示例

2.2 部分语言元素

Forth 将指令或函数称为 word,它是一种可扩展语言,支持函数定义(subroutine)和函数调用,新定义的函数又称新 word,可以像内置 word 一样被使用,因此定义新函数的同时也扩展了 Forth 语言,表 1 是相关 word 功能说明,详细可参考[7].

Table 1 Part of Forth words

表 1 部分 Forth 指令

Word	Data Stack	Description
DUP	(n -- n n)	Duplicates n.
DROP	(n --)	Removes n from the data stack.
1-	(n1 -- n2)	Subtract 1, n2=n1-1.
1+	(n1 -- n2)	Add 1, n2=n1+1.
SWAP	(n1 n2 -- n2 n1)	Exchanges the top 2 stack entries.
OVER	(n1 n2 -- n1 n2 n3)	n3 is a copy of n1.
>	(n1 n2 -- flag)	Is n1 > n2 ?
=	(n1 n2 -- flag)	True if n1 is equal to n2.
>R	(n --)	Moves a number to the return stack.
R>	(-- n)	Moves a number from the return stack to the data stack.
R@	(-- n)	Copies the top of the return stack to the data stack.
IF	(flag--)	Executes following code if flag is true.
ELSE	(--)	Executes the following code if IF failed.
THEN	(--)	Terminates an IF ... ELSE ... THEN.
: name		Starts the compilation of a new word definition named as name, which denotes subroutine.
;	(--)	Finishes the compilation of a word(subroutine) definition.
EXIT	(--)	Returns from the current word.
n	(-- n)	Pushes the number n to the data stack.
+	(n1 n2 -- n3)	Adds n1+n2.
-	(n1 n2 -- n3)	n1 subtracts n2.
*	(n1 n2 -- n3)	Multiplies n1 times n2.
/	(n1 n2 -- n3)	n1 divided by n2.
(	(--)	Begins a comment.
)	(--)	Ends a comment.

3 可微分抽象机实现方法

由第 2.1 节 Forth 语言执行模型可知,其程序执行,既可看成指令执行序列,又可从公里语义的角度忽略指令的实现细节,只评估指令执行后对 Forth 状态单元的改变,这种完全以状态的迁移来描述程序执行的方式我们称之为抽象机模型 AM,Forth 程序在 AM 上的指令执行可以表示为状态的迁移:

$$(\mathbf{D},\mathbf{R},\mathbf{Pc},\mathbf{Pd},\mathbf{Pr})\rightarrow(\mathbf{D}',\mathbf{R}',\mathbf{Pc}',\mathbf{Pd}',\mathbf{Pr}'). \quad (1)$$

其中 D 数据栈、R 返回栈、Pc 程序指针、Pd 数据栈栈顶指针、Pr 返回栈栈顶指针,它们构成了 Forth 抽象机 AM 的状态单元,左边是指令执行前状态,右边是指令执行后的状态.直接的,将 Forth 抽象机的离散状态空间映射到可微分抽象机 dAM 的连续状态空间.dAM 的状态用 $s_d=(\mathbf{D_d}, \mathbf{R_d}, \mathbf{Pc_d}, \mathbf{Pd_d}, \mathbf{Pr_d})$表示,dAM 执行过程即状态空间 $\boldsymbol{S}$ 中状态到状态的映射f_{dam}: $\boldsymbol{S}\rightarrow\boldsymbol{S}$,抽象机的 word 指令 $w\in f_{\text{dam}}$可被看为改变 dAM 状态的函数.

实现连续可微分抽象机的方法原理:1)状体操作连续可微分实现:首先对 Forth 抽象机的状态访问的基本操作以矩阵代数的方式实现为连续可微分的,包括读(READ)和写(WRITE)操作,类似 attention 机制,通过把栈指针看成内存单元权重的方式实现 READ 和 WRITE 操作,比如数据栈 READ 操作实现为数据栈指针 Pd 跟数据栈 D 的矩阵乘法,其原理类似 NTM[3]内存访问实现,然后基于 READ 和 WRITE 线性组合实现 PUSH、POP 等内存高级操作;2)word 操作的连续可微分实现:借助矩阵代数运算来表示 word 操作,从而实现为连续可微分的,比如 word 指令的基本功能可分为访问抽象机状态单元或算术逻辑运算,其中状态访问的实现原理跟 1)类似,而算术逻辑运算则可借助 one-hot 矩阵对数据进行编码,然后通过矩阵算术运算计算结果.

4 可微分抽象机混合编程实现关键技术

本节详细说明实现 dAMP 的关键技术,是本文的重点章节.

4.1 源程序到中间代码编译

词法分析

这里给出本文词法分析正规式定义,即生成 token 的规则,源程序经过词法分析,转换为 token 序列,如正规式公式(2)所示.

$$\begin{cases} letter\_\rightarrow[a-z]|[A-Z]|\_ \\ id\rightarrow[letter\_|\backslash d]+[letter\_|\backslash d|\backslash+\backslash-]* \\ int\rightarrow[0-9]+ \\ space\rightarrow(\backslash s)+ \\ comment\rightarrow\backslash((.)*\backslash) \\ word\rightarrow dup|drop|1-|1+|swap|over|>|=|>r|r>|r@|exit|n|+|-|*|/|nop|abort \\ gword\rightarrow \text{word}|\text{id} \\ keyword\rightarrow begin|while|repeat|do|loop|:|;|if|then|else|\{|\}|observe|->|choose|manipulate \end{cases} \tag{2}$$

语法分析(AST 生成)

$$\begin{cases} <program>::=<p>\ token("end") \\ <p>::=(<expr>\ |\ <def\_sub>)\ |\ <p\_r> \\ <p\_r>::=(<expr>\ |\ <def\_sub>)\ <p\_r>\ |\varepsilon \\ <exprs>::=<expr><exprs\_r> \\ <exprs\_r>::=<expr>\ exprs\_r\ |\varepsilon \\ <expr>::=<number>\ |\ <ifthenelse>\ |\ <ifthen>\ |\ <gword>\ |\ <\boldsymbol{nc}> \\ <\boldsymbol{nc}>::=\boldsymbol{token}(\boldsymbol{<\%})\ <\boldsymbol{enc}>\ \boldsymbol{token}(":")\ <\boldsymbol{transforms}><\boldsymbol{dec}>\ \boldsymbol{token}(\%>) \\ <transforms>::=<transform\_\text{fun}>\ token(":")\ <transforms\_r> \\ <transforms\_r>::=<transform\_\text{fun}>\ token(":")\ |\varepsilon \\ <transform\_fun>::=<sigmoid>\ |\ <tanh>\ |\ <linear><numbers> \\ <numbers>::=<int><numbers\_r> \\ <numbers\_r>::=<int><numbers\_r>\ |\varepsilon \\ <enc>::=<observe> \\ <observe>::=\ token("observe")\ <state\_elems> \\ <state\_elems>::=<d><state\_elems\_r> \\ <state\_elems\_r>::=<d><state\_elems\_r>\ |\varepsilon \\ <d>::=\ d0\ |\ d1\ |..|\ dn\ |\ r0\ |\ r1|..|\ rn \\ <dec>::=<choose>\ |\ <manipulate> \\ <def\_sub>::=\ token(":")\ <id><body>\ token(";") \\ <body>::=<exprs> \\ <manipulate>::=\ token("manipulate")\ <state\_elems> \\ <choose>::=\ token("choose")\ <exprs> \\ <ifthen>::=\ token("if")\ <body>\ token\ <"then"> \\ <ifthenelse>::=\ token("if")\ <body>\ token("else")\ <body>\ token("then") \\ <gword>::=<word>\ |\ <id> \end{cases} \tag{3}$$

语法分析以源程序的 token 序列为输入生成抽象语法树(AST),Forth 通过栈和利用后缀表达式(RPN)的特点进行解析,解析简单同时极其灵活,不需要复杂的语法制导解析过程,因此当前 Forth 语言没有类似于 C 语言的静态 BNF 产生式来限定语法[8].本文是经过裁剪定制的 Forth 验证语言,为了描述清晰规范,这里给出实现本文验证用 Forth 语法分析的 BNF 主要产生式项,如产生式公式(3)所示,其中<*nc*>条目用于产生内嵌神经网络组件,以实现对用神经网络组件跟 Forth 元素混合编写的源程序进行语法分析.为了简单清晰,这里省略了一些简单的 term 元素和跟 ifthenelse 类似结构(如 while)的产生式.

中间代码生成

算法 1 描述从 AST 生成中间代码,命名 2 同 to,中间代码指令集的部分指令跟 Forth 指令同名且功能相同,但也有一些专门指令(控制指令为主),算法 1 借助注释描述了相关中间代码指令含义,易混淆语义的指令由表 2 给出专门解释..

算法 1. *ast2im*(*ast, labelAllocator*) //*ast2im* 取其读音代表 *astToim*

输入: *ast*-抽象语法树;*labelAllocator*-分配管理不重名的跳转标签.

输出: *imCode*-中间代码.

imCode = []

```
FOR node IN ast
    IF node.type is defsub
        labelOverSubdef = labelAllocator.newLabel()
        imCode.append(("goto", labelOverSubdef))     //跳转到函数代码块的最后一行的下一行位置.
        imCode.append(("label", node.value.id)) //函数调用被翻译为跳转,label 指向函数块起始地址.
        imCode.extend(ast2im(node.value.body, labelAllocator)
        imCode.append(("exit",))
        imCode.apend(("label", labelOverSubdef))
    ENDIF
    IF node.type is number
        imCode.append(("constant", node.value))
    ENDIF
    IF node.type is gword          //如果是普通 word,而非关键字
        IF node.type is word      //如果是系统定义的内置普通 word
            imCode.append((node.value,))
        ELSE                       //如果是自定义 word(这里是函数)标识符(id),即函数调用.
            imCode.append(("call", node.value))
        ENDIF
    ENDIF
    IF node.type is ifthenelse     //对应 Forth 语法 if true_branch else false_branch then.
        labelOverThen = labelAllocator.newLabel()
        labelOverElse = labelAllocator.newLabel()
        trueBranchIM = ast2im(node.value.trueBranch, labelAllocator)
        falseBranchIM = ast2im(node.value.falseBranch, labelAllocator)
        imCode.append(("gotof", labelOverElse)) //如果测试条件为假,跳转到 labelOverElse.
        imCode.extend(trueBranchIM)
        imCode.append(("goto", labelOverThen)) //无条件直接跳转.
        imCode.append(("label", labelOverElse))
        imCode.extend(falseBranchIM)
        imCode.append("label", labelOverThen)
    ENDIF
    IF node.type is nc
        decIM = ()
        IF dec.type is choose
            chooseItems = ast2im(node.value.dec, labelAllocator).toTuple()
            decIM = ("choose", chooseItems)
        ELSE
            raiseException("Not supported dec type: %s" % (dec.type))
        ENDIF
        encTupled = (node.value.enc.type, node.value.enc.value)
        imCode.append(("nc", encTupled, node.transforms, decIM, node.value.label))
    ENDIF
```

```
    IF node.type is doloop        //do body loop
        imCode.append(("do",)) //tos 表示 DataStack 栈顶元素,nos 表示次顶元素.
        labelBegin = labelAllocator.newLabel()
        imCode.append(("label", labelBegin))
        imCode.extend(ast2im(node.value.body, labelAllocator))
        imCode.append("loop", labelBegin) //loop 执行 tos++,如果 tos<nos,跳转到 labelBegin,否则 pc++.
        imCode.append("enddo",)
    ENDIF
ENDFOR
RETURN imCode
```

源程序以中间代码的形式在可微分抽象机上执行.算法 1 中,函数调用通过 *call* 指令实现,该指令在控制转移前执行保存现场操作,*call* 保存现场只保存返回地址,不处理函数局部变量或函数参数传递,出于简化本文没实现 Frame Stack,典型的 Forth 实现也没有 Frame Stack,但这不影响实现函数的嵌套及递归调用,函数体以 *exit* 指令结束,消耗返回地址、恢复控制现场,函数定义的代码生成在生成函数体代码前,先插入一个跳转标签,指向函数结尾,从而保证正确的执行转移,标签地址是符号地址,随后的中间代码优化处理阶段替换为实际地址,参照函数调用的实现技巧可以很容易理解条件分支、循环等控制指令实现方式.

神经网络组件元素 *nc* 包括(encoder, transforms, decoder)三个部分,其功能分别是状态编码、状态转换、状态解码产生状态操作动作,这三者构成一个完整的带有可学习参数的神经网络组件,并可编程客制.

中间代码指令功能单一、支持机械化微码求值实现,算法 1 中间代码生成过程将函数定义、函数调用、*ifthenelse* 等复杂结构转换为基本中间指令序列,因此将源程序转换为中间代码后非常适合转换为逐步的连续可微分执行,即适合在 dAM 上运行,而混合编程语言语法特点是适合高效创作复杂源程序.

Table 2 Part of IM commands

表 2 部分中间代码指令

指令	描述
GOTO L	无条件跳转到 L
GOTOF L	当测试条件为假(0)时跳转到 L,否则不跳转,PC++,测试条件即当前 Data Stack 栈顶值.
CALL L	保存当前 PC+1(返回地址)到 Return Stack,然后跳转到 L.
EXIT	与 CALL 配对,弹出 Return Stack 栈顶返回地址,然后跳转到该地址继续执行.
LABEL L	在当前位置定义并设置一个跳转目标标签 L.
LOOP L	DataStack 栈顶元素加 1,然后,若小于次顶元素值,则跳转到 L 继续执行循环,否则退出循环.
DO	跟 LOOP 和 ENDDO 配对,在 doloop 循环开始做初始化,从 datastack 拷贝参数到 returnstack.
STEP	显式地将程序控制推进一步,即 PC++.
ENDDO	跟 LOOP 和 DO 配对,表示 doloop 循环结束,无实际执行动作,在中间代码优化时可剔除.

4.2 中间代码优化

为了提高中间代码在 dAM 上执行效率和 dAM 执行输出计算图的训练学习效率,需要对中间代码进行优化处理,本小节介绍中间代码优化处理的算法.中间代码优化比较复杂也非常关键,需经多遍处理才能完成,每一遍的优化目标不同,因此,根据不同目标,我们首先分步骤把算法拆分成若干子算法分别做详细说明,然后整合起来形成中间代码优化总算法.

算法 2. *labRenumber*(*imCode*)

输入: *imCode*-跳转标签重编号前中间代码.

输出: *labRenumberedIMCode*-标签重编号后中间代码;*labTable*-标签重编码表.

```
labTable = {}
labRenumberedIMCode = []
FOR word in imCode                          //第 1 遍:label 收集并按照先后顺序编号.
```

```
    IF word.type is label
        labTable[word.value] = labTable.len()
    ENDIF
ENDFOR
DEF labReplace(imCode, labTable)                    //第 2 遍:用编号重命名标签指令的标签属性.
    replacedIM = []
    FOR word in imCode
        IF word.type is label
            replacedIM.append(("label", labTable[word.value]))
        ELSEIF word.type IN [call, goto, gotof, loop]
            replacedIM.append((word.type, labTable[word.value]))
        ELSEIF word.type is nc
            IF word.value.dec.type is choose
                items = labReplace(word.value.dec.value.toList(), labTable) //dec.value 是 exprs,如公式(3)
                decIM = ("choose", items.toTuple())
            ELSE
                raiseException("Not supported dec type: %s" % (word.value.dec.type))
            ENDIF
            replacedIM.append(("nc", word.value.enc, word.value.transforms, decIM, word.value.label))
        ELSE
            replacedIM.append(word)
        ENDIF
    ENDFOR
ENDDEF
REUTURN (labRenumberedIMCode = labReplace(imCode, labTable), labTable)
```

算法 2 对中间代码做两次遍历:第 1 遍按照先后顺序给标签定义分配从 0 开始的连续整数编号,生成标签编码表 *labTable*,第 2 遍根据 *labTable*,对所有指令类型为标签跳转或标签定义的指令,用编号重命名其标签属性.

算法 3. *conRenumber*(*imCode*)

输入: *imCode*-常量重编号前中间代码.

输出: *conRenumberedIMCode*-常量重编号后中间代码; *conTable*-常量重编码表.

算法 3 跟算法 2 类似,但因为常量不分定义和使用,只需对中间代码做一次遍历:按照遍历时首次访问顺序给常量分配从 0 开始的连续整数编号,并生成常量编码表 *conTable*(字典类型,键是常量的 *value* 属性,值是分配的整数编号),对所有指令类型为常量(*constant*)的指令,用编号重命名其常量值属性,遇到重复的常量,复用 *conTable* 已分配编号.算法 3 主体代码跟 *labReplace* 类似,故不详细展开其伪代码.

算法 4. *blockPartion*(*imCode*)

输入: *imCode* 分块前中间代码.

输出: *imBlocks* 分块后中间代码.

```
imBlocks = []
newBlock = []
DEF addNewBlock()
    IF newBlock is not empty
        imBlocks.append(("block", newBlock.toTuple()))
```

```
        newBlock.empty()
    ENDIF
ENDDEF
FOR word IN imCode
    IF word.type IN [call, goto, gotof, loop, exit]
        newBlock.append(word)
        addNewBlock()
    ELSEIF word.type IN [label, nc]
        addNewBlock()
        imBlocks.append(word) //不把 label 和 nc 指令并入前一个新 block,而是单独做 block.
    ELSE //连续的非控制指令普通 word 划分到一个块.
        newBlock.append(word)
    ENDIF
ENDFOR
addNewBlock() //添加末尾块
RETURN imBlocks
```

把中间代码划分为块,通常一个块主要由不做控制转移的普通 word 组成一个元组,块结尾才可能有控制转移 word,块的类型包括 block 块、label 块、nc 块,分块处理后的中间代码,其每个块可看作一个大 word.

算法 5. *lab2addr*(*imBlocks*)

输入: *imBlocks*-分块后的中间代码.

输出: *addrIM*-目的标签转换为目的地址后的中间代码块;*tabLab2addr*-标签地址对照表.

```
tabLab2addr = {}
addrIM = []
FOR word IN imBlocks //第一遍,清除目标标签,并创建目标标签到目标地址映射表.
    IF word.type is not label
        addrIM.append(word)
    ELSE
        tabLab2addr[word.value] = addrIM.len()
    ENDIF
ENDFOR
RETURN (addrIM, tabLab2addr)
```

算法 5 将目标标签转换为目标地址,建立标签到地址的映射表,并删除标签定义指令.

算法 6. *insertStepctl*(*addrIM*)

输入: *addrIM*-清除目标标签并建立标签到地址映射后的中间代码.

输出: *blockStepIM*-补全 *step* 控制指令后的中间代码块.

```
blockStepIM = []
FOR word IN addrIM
    IF word.type is block
        IF word.value.lastItem.type IN [call, goto, gotof, loop, exit, nc, step]
            blockStepIM.append(word)
        ELSE          //如果普通块结尾无控制指令,在块尾插入 step 指令.
            blockStepIM.append(("block", word.value + (("step",),)))
```

```
            ENDIF
        ELSEIF word.type is nc
            IF word.value.dec.type is choose
                chooseItems = word.value.dec.value.toList()
                FOR idx, item IN enumerate(chooseItems)
                    IF item.type NOT IN [call, goto, gotof, loop, exit, nc, step]
                        chooseItems[idx] = ("block", ((item.type, item.value), ("step",)))
                    ENDIF
                ENDFOR
                decSteped = ("choose", chooseItems.toTuple())
                blockStepIM.append ("nc", word.value.enc, word.value.transforms, decSteped, word.value.label)
            ENDIF
        ELSEIF word.type NOT IN [call, goto, gotof, loop, exit, nc, step]
            blockStepIM.append(("block", (word, ("step",))))
        ELSE // 不属于 block 的单独控制指令
            blockStepIM.append(word)
        ENDIF
    ENDFOR
    RETURN blockStepIM.append(("halt",)) //程序以 halt 指令结束.
```

算法 6 遍历代码块,为没有以控制转移指令结尾的代码块追加插入 *step* 指令,从而确保每个代码块执行结束时都更新 PC,即保证程序持续向前推进执行.

算法 7. *optimiseIM*(*imCode*)

输入: *imCode*-优化前中间代码.

输出: *optimisedIM*-优化后中间代码; *tabLab2addr*; *conTable*; *labTable.*

```
labRenumberedIMCode = [], labTable = {}
conRenumberedIMCode = [], conTable = {}
imBlocks = []
addrIM = [], tabLab2addr = {}, optimisedIM = []
labRenumberedIMCode, labTable = labRenumber(imCode)                    // 第 1 遍
conRenumberedIMCode, conTable = conRenumber(labRenumberedIMCode) // 第 2 遍
imBlocks = blockPartion(conRenumberedIMCode)                           // 第 3 遍
addrIM, tabLab2addr = lab2addr(imBlocks)                               // 第 4 遍
optimisedIM = insertStepctl(tabLab2addr)                               // 第 5 遍
RETURN (optimisedIM, tabLab2addr, conTable, labTable)
```

算法 7 是中间代码优化的整个流程,经过 5 次遍历,生成最终在 dAM 上执行的优化代码,代码优化是一个复杂但关键的过程,代码优化不仅能显著提高训练学习效率,也为实现结合神经网络的深度学习模型提供了基础. 图 4 是中间代码优化结果示意图,优化后的代码以不同大小的基本块组织,并以块为单位编址构成线性平铺代码地址空间,每个块都以显式的控制转移指令结尾,如果没有则在尾部插入 *step*,程序以块为单位逐步(块)执行,如果碰到标签跳转指令则按照标签目标执行跳转,标签按照第 1 遍优化时分配的连续整数编码命名,并按照第 4 遍优化时生成的标签地址对照表 *tabLab2addr* 查找在代码空间的实际地址.

代码优化对实现程序连续可微分执行、建模和训练有重要意义.第 3 节介绍了程序在 dAM 上的执行被描述为在某个连续状态空间中的连续状态转换,这显然适合以 RNN 为基础建模,RNN 有一个问题是如果步骤过多

会显著降低训练效果和效率,因此把以指令字为单位转换为以块为单位的单步执行能显著减少迁移步数,相当于通过压缩状态转换序列长度显著提升训练效果和效率;中间代码将地址空间转换为线性平铺的连续编号、常量和标签跳转编码为连续的整数编号,这些优化处理为基于神经网络的预测训练建模提供了便利.

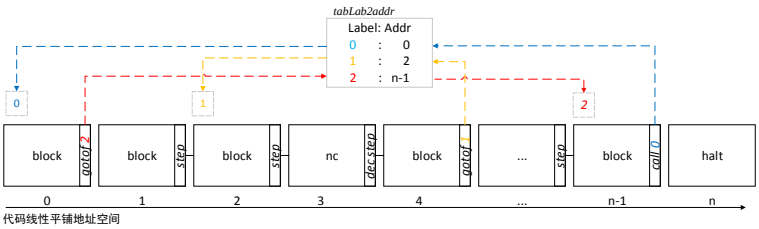


Fig.4 Intermediate code optimization

图 4 中间代码优化

4.3 程序汇编

本节主要介绍对优化后的中间代码块做进一步汇编处理,汇编处理就是为每个代码块转换为一个可直接操作可微分状态机状态的函数,这个函数就是中间代码块对应的汇编代码,因此块经过汇编处理后可以在运行阶段直接操作 dAM 运行并生成计算图.

每个 *block* 块或 *nc* 汇编代码包括两个部分用于执行时的两个阶段,第一阶段是在符号状态机上以符号执行的方式对中间代码代码块做化简,形成用符号变量、符号栈表示的块代码执行结果,因为符号执行时基于符号变量做计算,因此不依赖栈和变量的具体值,只需为每个块提供一个初始态的符号状态机做操作参数,不需要考虑块序列状态依赖、状态传递等问题,符号执行取得的符号化简结果跟状态无关,一次执行永远有效,所以只在中间代码块首次被访问时执行一次并可复用;第二阶段在连续可微分抽象机上求值,首先需要创建分配一个可微分抽象机,该抽象机除了提供基本操作、指令集等连续可微分实现,还需要分配、初始化和管理各状态组件,包括数据栈、返回栈、栈指针、Pc 指针、代码词汇表、代码矩阵、标签地址映射表、常量编码表等,其中词汇表和代码矩阵是中间代码优化并经过汇编处理后的代码字,代码矩阵是代码编号 one-hot 表示,可微分抽象机创建并初始化后,并构成了状态机的初始状态,中间代码在这个初始状态基础上逐步执行,逐步执行并不意味着一直顺序执行,比如碰到分支跳转指令会改变控制流.

神经网络组件(*nc*)的汇编处理相对要复杂一些,*nc* 块包括 *nc.type* 和 *nc.value* 两个部分,*vaule* 包括 *dec*、*enc*、*transforms* 和 *label* 四个部分,前三个需要专门处理.生成 dec 的汇编处理结构,需要把 dec 按照项目循环展开处理,算法 8 是生成 dec 的汇编代码指令字算法流程概要结构,算法 9 是为相应 enc 编码部分生成汇编代码的算法概要结构.

算法 8. *dec2decAsse* (*dec*).

输入: dec-神经网络组件(*nc*)的值的 dec 解码部分,即中间代码块条目列表.

输出: *decAsse*-汇编处理后的指令字结构.

decItems = *dec.value*

decItemsAsse = []

FOR *item* IN *decItems*:

    sym = *new symMachine*

    itemAsse = *block2Asse*(*sym, item*)

    decItemsAsse.append(*itemAsse*)

ENDFOR

RETURN *decAsse* = *chooseDecAsse*(*decItemsAsse*)

算法 9. *enc2encAsse* (*enc, transforms, outputDim*).

```
输入: enc-网络组件(nc)值的 enc 状态编码部分;transforms-状态转换函数列表; outputDim-编码输出维度.
输出: encAsse-汇编处理后的指令字结构.
transformFuns = []
odim = outputDim // 因为 enc 的输出是 dec 的输入,所以该输出维度取决于 dec 解码组件的输入维度.
FOR item IN transforms.reverse()
    IF item.type is linear
        ffnImputDim = item.value
        ffnOutputDim = odim
        DEF ffnLinearTrans(input)
            return weights(ffnOutputDim, ffnImputDim) * input+ bias
        ENDDEF
        transformFuns.append(ffnLinearTrans)
        odim = ffnImputDim
    ELSEIF item.type is tanh
        transformFuns.append(tf.tanh)
    ENDIF
ENDFOR
DEF transFormAsse(input)
    r = input
    FOR f IN transformFuns.revrse()
        r = f(r)
    ENDFOR
    RETURN r
ENDDEF
IF enc.type is observe
    elems = enc.value
    FOR ele IN elems
        inputDim += ele.valueSize
    ENDFOR
    observeAsse = weights(odim, inputDim) * stateInput + bias
ENDIF
encAsse =   observeEncAsse(observeAsse, transFormAsse)
RETURN encAsse
```

4.4 程序可微分执行模型

第一步创建连续可微分抽象机 dAM,并初始化,包括汇编后的词汇表 vocab、代码矩阵、常量编码表、标签地址映射表、PC 指针初始指向代码地址 0、新建返回栈并初始化为干净状态、数据栈和数据栈指针则由占位符变量初始化,用来接收训练样例输入,dAM 对象包含了指令集实现函数和抽象机当前完整状态,初始 dAM 即初始状态(用 S_{dam_init} 表示).程序在 dAM 上一次执行轨迹就是由从初始状态出发的状态列表 $exeTrace = [S_{dam\_init}, S_{dam\_1}, S_{dam\_2}, \ldots, S_{dam\_n}]$,初始值$[S_{dam_init}]$,每个状态 S_{dam_i} 都是在前一个状态上执行某个代码块汇编函数结果,显然不同的初始状态可能会产生不同的执行轨迹,包括导致 PC 序列可能存在差异.

程序的可微分执行模型:创建 S_{dam_init},然后在 S_{dam_init} 上继续执行 n 步,n 是针对程序特点和输入规模计算的执行步数,所以程序的执行模型是一个 RNN 模型,该 RNN 模型在时间序列上展开后由 n 步组成,而每一步执行

包括两个动作,第一个动作是根据 Pc 向量从代码矩阵中做指令寻址,其连续可微分矩阵计算可描述为公式(4):

$$C_{sel\_w} \leftarrow P_c^T Code,\ \sum_i P_c(i)=1, 0 \leqslant p_c(i) \leqslant 1,\ 0 \leqslant i \leqslant codesize-1 \quad (4)$$

其中 Code 是汇编代码块编号的 one-hot 代码矩阵,Pc 是 one-hot 表示程序指针向量,训练期间其分量是代码分布权重,codesize 是代码包含的代码块个数,或者汇编后词汇表指令字个数,C_{sel_w} 是指令寻址权重分布,在锐化处理或者训练稳定后 C_{sel_w} 精确选取单个汇编指令,第二个动作是基于 C_{sel_w} 选择指令执行,该动作有两种实现方式,方法 1 是执行所有代码块,然后按照 C_{sel_w} 计算 attention 权重和,可描述为公式(5):

$$step\_run \leftarrow C_{sel\_w}^T RUN(Code, S_{current}),\ \sum_i C_{sel\_w}^T(i)=1, 0 \leqslant C_{sel\_w}^T(i) \leqslant 1,\ 0 \leqslant i \leqslant codesize-1 \quad (5)$$

其中 $S_{current}$ 是当前状态,RUN(Code, $S_{current}$)表示在 $S_{current}$ 状态下分别执行代码 Code 的每个汇编后指令,一共 codesize 次,每个执行结果得到一个新状态,所有新状态构成结果状态向量,然后以 $C_{sel\_w}^T$ 取结果向量权重和就是本次单步执行的结果.方法 2 是对 C_{sel_w} 做锐化处理,基于锐化结果选定一条指令执行并改变可微分抽象机状态,可描述为公式(6):

$$step\_run \leftarrow RUN(sharpen(C_{sel\_w}^T)Code, S_{current}),\ \sum_i C_{sel\_w}^T(i)=1, 0 \leqslant C_{sel\_w}^T(i) \leqslant 1,\ 0 \leqslant i \leqslant codesize-1 \quad (6)$$

其中 *sharpen* 函数对 $C_{sel\_w}^T$ 做锐化处理得到一个 one-hot 向量,该 one-hot 向量跟 Code 矩阵乘法取出唯一的指令编号,然后该指令在当前状态 $S_{current}$ 下执行的结果就是本次单步执行的结果.理论上两种方法都是可行的,但锐化处理因为不需要每次执行所有指令字所以效率更高.

从初始状态 S_{dam_init} 开始每执行一步产生下一个状态,这些状态最终形成一个完整的状态迁移轨迹,从公式(4)、(5)或(6)可知,执行过程中根据 $S_{current}.P_c$ 以连续可微分方式做指令寻址,根据指令寻址选择汇编指令函数做连续可微分执行,并基于 $S_{current}$ 和执行结果更新生成下一个状态 S_{next},其中包括 $S_{next}.P_c$、$S_{next}.DataStack$、$S_{next}.P_d$ 等属性,因此逐步执行生成 n 步执行轨迹的过程就是生成程序的连续可微分计算图表示的过程.

4.5 训练目标函数

模型使用程序执行轨迹的结束状态进行训练,中间状态在训练模型时不需考虑,因此不需要像 NPI[2]那样对执行轨迹的每一步做标注,因而显著降低了训练数据的生成和标注成本.($\hat{x}_{init.i}$, $\hat{y}_{end.i}$)表示训练样本 *i*, $\hat{x}_{init}$ 是初始状态,标注目标 $\hat{y}_{end}$ 是结束状态,$y_i=R_n(\hat{x}_{init.i}, \Theta)$表示对应于初始状态 $\hat{x}_{init.i}$,在参数为 Θ 的 dAM 模型上执行 *n* 步后得到的状态,模型训练目标是对于所有的训练样本 *i*,求解使得 *y* 跟 $\hat{y}_{end}$ 差距最小的参数 Θ^*:

$$\Theta^* = ARG\ MIN_\Theta\ (\sum_i ||y_i - \hat{y}_{end\ i}||_2) \quad (7)$$

也用交叉熵来描述训练目标,即求解使得样本 $\hat{y}_{end}$ 跟模型估值 y 的交叉熵尽可能小的模型参数 Θ^*,即:

$$\Theta^* = ARG\ MIN_\Theta \mathcal{H}(\hat{y}_{end}, y) = ARG\ MAX_\Theta \sum_i \hat{y}_{end\ i} \log(y_i) \quad (8)$$

每个状态最多包括 DataStack(D)、P_d、ReturnStack(R)、P_r、Heap(H)五个状态分量,所以公式(8)可分解为五个分量的和:

$$\Theta^* = ARG\ MIN_\Theta[\mathcal{H}(\hat{D}_{end}, D) + \mathcal{H}(\hat{P}_{d\_end}, P_d) + \mathcal{H}(\hat{R}_{end}, R) + \mathcal{H}(\hat{P}_{r\_end}, P_r) + \mathcal{H}(\hat{H}_{end}, H)] \quad (9)$$

其中 $\mathcal{H}$ 是交叉熵函数,一般的,在具体算法场景,其损失函数不需要所有 5 个分量,本文的实验只用到 P_d 和 D 两个分量,给定算法场景用不到的分量的交叉熵为 0,因此可在公式中去掉相应交叉熵分量的计算.因为系统是连续可微分的,所以可以使用反向传播算法训练优化损失函数.

5 实验分析

本节对 dAMP 系统做实验分析,首先给出同一个多位数加法算法的 3 种代码实现实例,结合具体例子对比分析了 dAMP 的代码生成特性、实例选取的原因动机,然后以该算法程序为例对模型的训练效果、训练过程中运行时状态对技术设计目标正确性的反馈验证和模型复杂度进行实验分析,最后对系统的性能进行评估.

5.1 多位数初等加法算法

多位数初等加法运算是由低位往高位移动做单位数加法,每步单位数加法基本操是基于低位进位、本位被加数和本位加数计算向高位的进位和本位和,多位数加法是递归执行单位数加法,该样例好处是问题简单,但包

含了顺序语句、分支控制、函数调用、递归等复杂程序结构元素(显然这些结构足以表达其他广泛、复杂的编程问题),因此方便进行简单明了的多方面系统分析,又不失对 dAMP 全可微分模型编程能力验证,分支、函数调用、递归等控制结构处理能力也是程序生成研究关键之一[2,3,4].

图 5 给出了该多位数初等加法的类 C 语言伪代码、Forth 代码以及跟 Forth 实现相对应的 dAMP 神经网络组件混合代码的实现,图中黑体加粗代码是计算进位和计算本位和的代码的三种对照实现,其中 Forth 代码实现是确定性的,需 20 个 word 实现,每个 word 是 Forth 的一行代码(把 20 个 word 放在 5 行以节省空间),而 dAMP 实现则是不确定的,不需编写完整算法,其中最复杂的每步计算进位和本位和操作用待训练学习的神经网络组件代替,通过编写框架主要提供 2 个编程经验输入:这是个递归算法结构、每步计算进位和本位和的取值范围分别是 0-1 和 0-9.

类C伪代码实现

```
int c=0, s=0;
int sum[10]={0};
int lt[2]={4,1};
int rt[2]={5,9}, n=2, p=0;
DEF multiAdd(){
 if p == n{
  return;
 else
  s=lt[p]+rt[p]+c;
  c=s/10;
  sum[p]=s%10;
  p++;
  multiAdd();
 }
}
multiAdd();
print(carry, sum);
```

Forth代码实现

```
: multiAdd(--)
 p_global @ n_global @ = if
  exit \return
 else
  c_global @ \D-2
  rt_global p_global @ CELLS + @ \D-1
  lt_global p_global @ CELLS + @ \D0
  2DUP >R >R DUP >R \在R中生成a1,b1,c的副本,顺序为c,b1,a1
  + + \加数、进位、被加数和sum=b1+c, sum+=a1
  R> SWAP R> SWAP R> SWAP \a1,b1,c副本移动到D,每步与sum交换
  >R @R 10 / \sum移动到R做副本,再复制回D,carry=sum/10
  R> 10 MOD \从R移动sum副本到D,本位和sum%10
  s_global ! \s=sum,需要保存或转移的栈上结果
  c_global ! \c=carry,需要保存或转移的栈上结果
  drop drop drop \drop D-2 D-1 D0,不需要保存的栈上结果
  s_global @ sum_global p_global @ CELLS + ! \sum[p]=s
  p_global @ 1 + p_global ! \p++
  multiAdd
 then
;
multiAdd
carry @ . sum @ . \print the sum
```

dAMP混合编程实现

```
: multiAdd(--)
 p_global @ n_global @ = if
  exit \return
 else
  c_global @ \D-2
  rt_global p_global @ CELLS + @ \D-1
  lt_global p_global @ CELLS + @ \D0
  <%observe(D0,D-1,D-2]):transform(linear(30)):
   decoder(choose(0,1,2,3,4,5,6,7,8,9))%>
  s_global ! \s=sum,需要保存或转移的栈上结果
  <%observe(D0,D-1,D-2):transform(linear(10)):
   decoder(choose(0,1))%>
  c_global ! \c=carry,需要保存或转移的栈上结果
  drop drop drop \drop D-2 D-1 D0,不需要保存的栈上结果
  s_global @ sum_global p_global @ CELLS + ! \sum[p]=s
  p_global @ 1 + p_global ! \p++
  multiAdd
 then
;
multiAdd
carry @ . sum @ . \print the sum
```

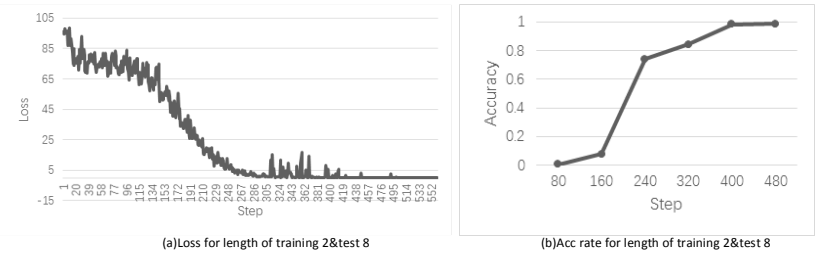
Fig.5 Elementary multidigit addition

图 5 多位数初等加法

5.2 模型训练效果分析

模型可收敛性实验验证:可通过反向传播算法训练优化是模型的关键,第 3、4 节介绍 dAMP 实现时从理论上说明 dAMP 是连续可微分的,图 6 所示实验数据对此进行了验证,该实验以一位加数、一位被加数和进位的输入输出数据为训练样例,以 2 到 4 位(加数加被加数长度共 8 位)整数加法构建测试样例,图 6(a)可知在 Loss 稳定收敛,训练约 400 步时 Loss 收敛到非常接近 0,图 6(b)可知在模型训练超过 400 步时,2 到 4 数加法的测试准确率达 100%,因此模型具有很好的收敛性.另外我们测试了用 1 位数加法训练得到的模型做 2 到 34 位整数加法,得到 100%准确率,所以模型具有很好的泛化能力.

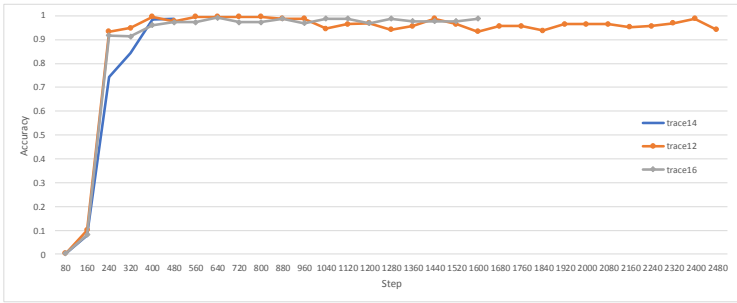
轨迹步数设置对训练的影响:4.4 节介绍了 dAMP 程序执行模型是由 n 步状态迁移构成的 *exeTrace*,因此设置合适的 n 对模型训练很关键,n 的设置需参照中间代码优化后代码块个数和程序的输入规模,比如图 7(a)所示,训练加法算法输入规模为 2 时,n=14 左右效果最佳,n>=16 明显增加训练步数,当 n<=12 时准确率波动很大,同时图 7(b)显示其 Loss 收敛很好,这种情况很可能是因 n 偏小造成过拟所致.



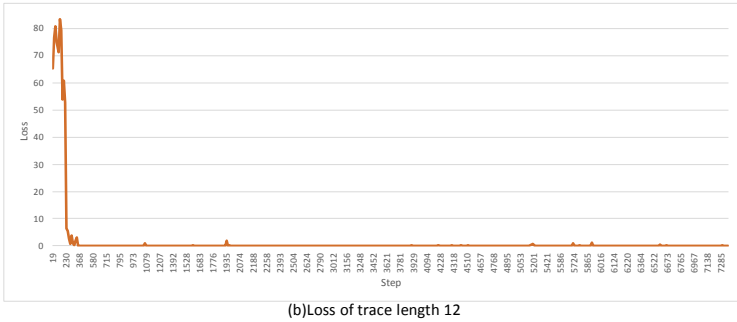
(a)Loss for length of training 2&test 8 (b)Acc rate for length of training 2&test 8

Fig.6 Convergence of model training

图 6 模型训练收敛情况



(a)Accuracy of trace length 12,14&16



(b)Loss of trace length 12

Fig.7 The effect of number of trace step on training

图 7 轨迹步数对训练的影响

5.3 运行时执行过程验证

验证方法是记录加法程序在 dAMP 上训练运行时 PC 的值,分析在一个执行轨迹周期中其值的变化是否符合预期,图 8(a)是 PC 的实际执行轨迹,该轨迹取自 2 位数加法训练过程,图 8(b)给出了多位数加法程序经过中间代码优化和汇编后的代码块,以及以块为基本执行单位的程序控制转移逻辑作为参照.从图可以看出运行时 PC 移动轨迹完全符合预期,其中虚框部分证明模型正确执行了函数调用与递归,右上角最后两步停留在 Halt 指令不再转移,说明程序执行正常结束,该实验现象也说明 exeTrace 设置为 20 步即可满足要求.

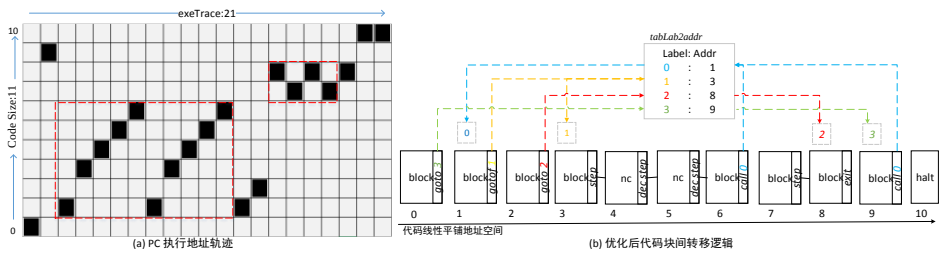


Fig.8 Runtime PC-trace correctness verification

图 8 运行时 PC 轨迹正确性验证

5.4 模型复杂度分析

4.4 节给出了程序在 dAMP 上执行模型是 n 步 exeTrace,因此程序在 dAM 上执行复杂度主要由 n 决定.以多位数加法为例,设加数(或被加数)位数为 m,则 n=2+5m+2+2m +2=7m+6=O(m),即问题输入规模的线性复杂度,该计算过程可参照图 8 运行时轨迹,类似的,对于冒泡排序其模型复杂度是 $O(n^2)$,所以 dAMP 运行时轨迹复杂度取决于算法框架本身;4.2 节中间代码优化将程序按照控制结构分块,将程序从逐指令粒度优化到块粒度执行从

而有效提升执行性能,但因为没有改变程序结构因此不会达到改变复杂度,另外将梯度计算放在程序执行结尾而不是执行过程中的每个状态迁移也是性能优化的一个方面.显然 dAMP 混合模型能够利用 nc 组件消除条件分支等控制结构,但进一步,对于给定问题如冒泡排序,是否能设计影响程序循环或递归结构从而达到优化算法复杂度还有待未来做专门探讨.

5.5 性能评估

dAMP 开发环境为 Python 3.5.2、NumPy[10] 1.14.5、TensorFlow[11] 1.10.0 CPU 版、Ubuntu 16.04 X86_64, 8 核心 Intel Xeon E5-2407 处理器、64G 内存,测试环境同开发环境.性能评估选用有代表性的算术操作和关系操作,实验方法为随机生成 50 组形状为(1,10)的操作数,统计每个操作执行 50 次运算的总时间,单位为秒(s),所以 Performance 数值越小表示性能越好.为了对 dAMP 性能有一个横向的评估,实验同时选取了 TensorFlow(TF)功能相近的操作,使用相同的数据做测试对照.图 9(a)是算术操作性能测评结果,对照 TF.add、TF.subtract、TF.multiply、TF.div,图 9(b)是关系操作测评结果,对照 TF.equal、TF.greater、TF.less、TF.greater_equal、TF.less_equal、TF.not_equal,测试数据表明 dAMP 表现出很好的性能.这里性能测试未做任何专门优化,如果使用 GPU 加速,理论上性能还会有明显提升.

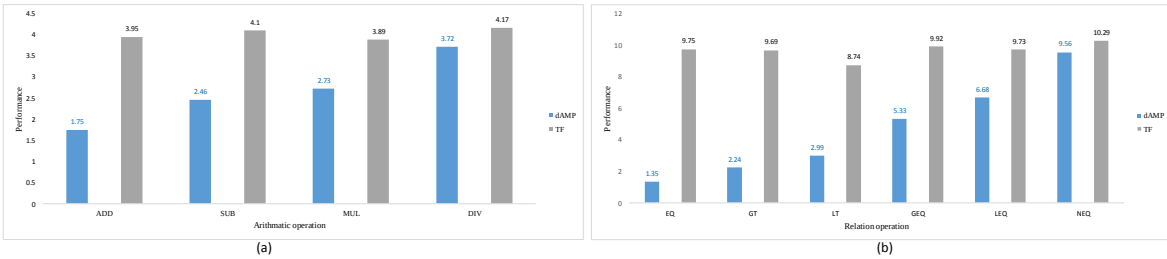


Fig.9 Performance of fundmental operations

图 9 基础操作性能

6 相关工作

机器编程相关研究始于 20 世纪七十年代[1],经过近 50 年的发展取得了长足进步,但是其通用能力、泛化能力、程序覆盖能力仍明显不足,因此限制了实际应用.主要相关性研究:

类自然语言研究方法:将自然语言处理研究的方法迁移到编程语言领域[12,13,14]相关工作集中在代码注释、论坛等自然语言语料到代码关键字、标识符等的对照关系处理上,对代码自身信息的直接处理上表现不足.

归纳式程序合成方法:该方法将程序生成定义为搜索问题. 所有程序样例的全集构成一个离散程序空间, 程序生成的过程就是通过观察输入输出样例数据特征, 在程序空间归纳搜索跟样例数据行为一致的程序, 受程序合成研究方法习惯于用规则和抽象提高问题处理效果启发[15, 16, 17], 本文提出在更具抽象层设计连续可微分抽象机的混合编程的程序处理思路, 但是不同于归纳式程序合成把程序生成定义为在离散空间搜索, 本文则是基于连续可微分优化的更通用的一种方法.

基于连接的神经网络方法:该方法为程序处理寻找一个连续空间表示,将问题看成是在连续空间待学习参数和一套可微分规则组成的系统,程序生成问题被转换为在连续空间采用可微分优化方法来逼近一个解,这是本文最相关的研究方法.典型的 NPI[2]使用神经网络从程序执行轨迹中学习预测下一步调用的子指令,非完全可微分,依赖于收集高成本程序执行轨迹,本质上是算法编码而不是生成.NTM[3]通过注意力机制将外部存储跟神经网络内部存储耦合,扩展了神经网络的记忆容量,实现了全微分方式访问外部缓存,为实现 dAMP 的连续可微分栈访问操作提供了借鉴,但 NTM 无指令集因此无法实现可微分编程控制.Forth[9]提出了一种 forth 编程语言可微分解释器的初步实现方法但不完整,有很好的借鉴意义,与之相比本文提出并验证了一种在更具抽象层解决程序生成问题的混合编程研究思路、给出了更完整、成熟、严谨的实现框架、对 PC 寻址锐化显著提升效

率、提出验证了多种训练目标函数以适应不同问题、提出并分析了如何合理设置执行轨迹步数、对关键学习机理进行了理论和实验分析.Deepcoder[4]提出了一种基于神经网络在给定的 DSL 语言下计算指令频度的方法,本质上是结合神经网络的归纳式程序合成方法.

7 结束语

本文提出在抽象层研究程序自动生成问题的一种思路,给出其验证系统 dAMP,实现了一种可无缝结合高级过程化编程语言与神经网络组件的混合编程模型,给出了包含顺序块、条件分支、函数调用、递归等常规复杂程序结构元素和神经网络组件结构的程序样例实验分析,显然这些结构具备表达其他更广泛复杂程序的能力,完整严谨的技术设计和实验分析表明通过抽象层次提升,这种基于可微分抽象、支持表达复杂问题程序结构的混合编程模型在理论和技术上是可行的,与已有的程序生成方法相比,本文所提方法具有更灵活的可微分编程控制能力、良好的程序自动生成与泛化能力、优秀的性能.

原型验证在理论和技术上可行只是研究工作的一个方面,最终希望走向广泛的应用,而应用不仅关心技术可行性,比如首先可能还关心用户接口,因此我们认为下一步有必要对以下几个工作做探讨 1)通过扩展 C 语言子集,设计支持神经网络混合编程的类 C 语法作前端,基于本文的技术作为后端支持,从而更便于程序员编程思想表达,发挥其编程创造性;2)在前一个工作基础上,设计丰富的神经网络混合编程语言原语,以 C 语言、Java 等语言为例,丰富的编程原语对编程模型走向实际应用有重要意义,dAMP 全可微分特性支持嵌入包括神经网络在内任何可微组件,因此理论上可把典型的神经网络模型(如 seq2seq、encoder-decoder、CNN、GAN 等)甚至非神经网络但可微分模型设计为该编程模型原语级支持,以增强其编程表达能力;3)全面探讨或反向总结该可微分混合编程模型的应用场景适应性或编程潜力,参照 Java 等大多数编程模型发展经验,这不是一个完全可正向评估的纯技术问题,往往依赖于 Java 用户生态的编程创造性的反向反馈,即具有开放性和创造性,在正向上,因为支持丰富复杂的程序结构以及可微分混合编程特性,因此从技术上显然有能力表达广泛复杂的程序并因新语言结构的融合而引入新编程表达能力(如传统 C 程序是确定性的,因此主要用于编写完整的程序,求解完全明确的问题之类场景,而该模型特性则支持以传统过程化编程方式在确定性框架下引入非确定性编程支持,而非确定性局部又可借助数据驱动自学习生成,因此适合但不限于“不完全明确但有数据可依赖的问题编程”或者出于降低编程难度或提高编程效率的“非完整编程”),但在编程便利性应用上还有赖于下一步诸如类 C 语法前端和原语丰富性设计等工作,从而更好的激发程序员利用模型特性的编程创造性发挥.

References:

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

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

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

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

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

[6] Pelc S. Updating the Forth virtual machine[C]//24th EuroForth Conference. 2008: 24-30.

[7] UOR. Forth Words, [EB/OL]. 2013[2018-08-31] http://astro.pas.rochester.edu/Forth/forth-words.html

[8] Charles H. Moore. Forth, [EB/OL]. 2007[2018-08-31] https://www.cs.mcgill.ca/~rwest/wikispeedia/wpcd/wp/f/Forth.htm

[9] Bošnjak M, Rocktäschel T, Naradowsky J, et al. Programming with a differentiable forth interpreter[J]. arXiv preprint arXiv:1605.06640, 2016.

[10] NumPy developers, NumPy, [EB/OL]. [2018-08-31] http://www.numpy.org/

[11] Abadi M, Barham P, Chen J, et al. Tensorflow: a system for large-scale machine learning[C]//OSDI. Berkeley, CA: USENIX Association, 2016: 16: 265-283.

[12] 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).

[13] Dagenais B, Robillard M P. Recommending adaptive changes for framework evolution[J]. ACM Transactions on Software Engineering and Methodology, 2011, 20(4): 19.

[14] Nguyen A T, Hilton M, Codoban M, et al. API code recommendation using statistical learning from fine-grained changes[C]//Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM, 2016: 511-522.

[15] Gulwani S. Automating string processing in spreadsheets using input-output examples[C]//ACM SIGPLAN Notices. New York: ACM, 2011: 46(1): 317-330.

[16] 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.

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

附中文参考文献:

[1] 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.

周鹏(1984-),男,博士生,主要研究领域为机器学习、系统安全、操作系统与虚拟化.

武延军(1979-),男,博士,研究员,博士生导师,主要研究领域为操作系统、系统安全.

赵琛(1967-),男,博士,研究员,博士生导师,主要研究领域为编译技术、操作系统与网络软件.