dAMP:可微分抽象机混合编程系统

报告人：周鹏< zhoupeng@iscas.ac.cn >
作　者：周鹏 武延军 赵琛

中国科学院软件研究所
中国科学院大学

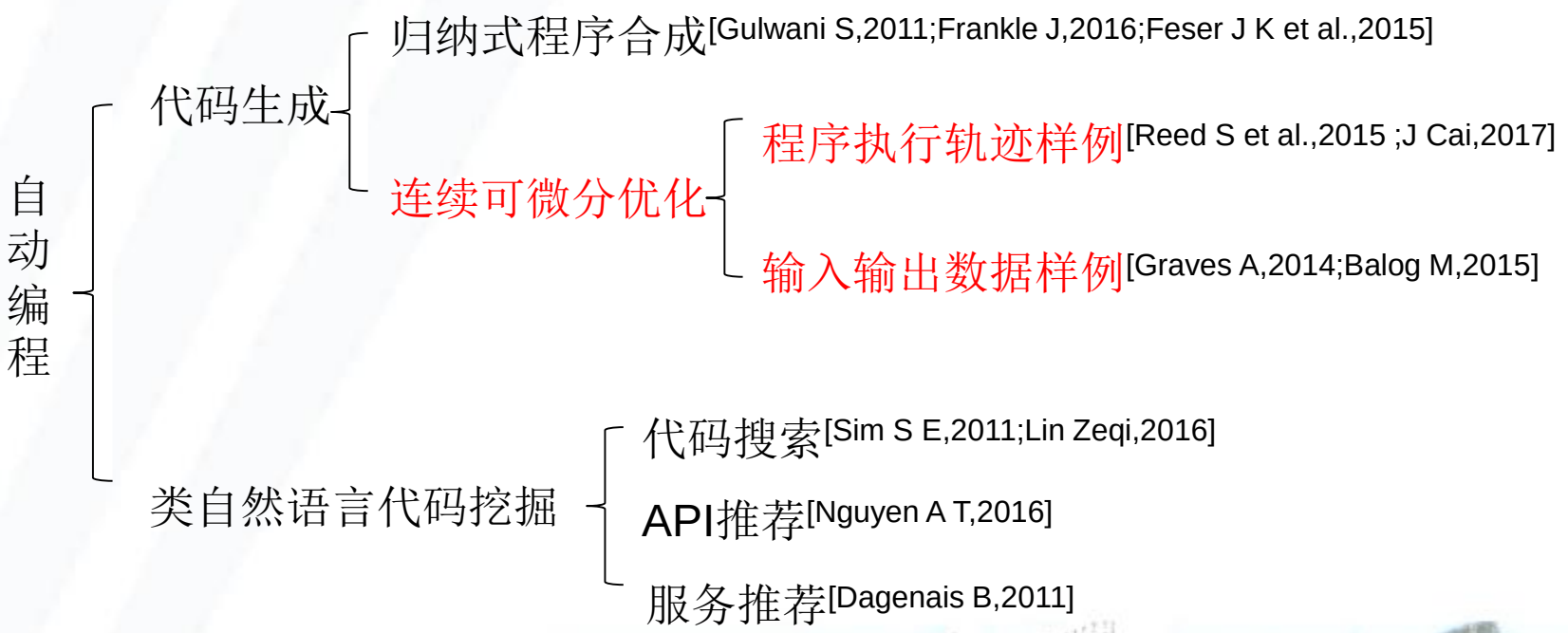
2018.11.23

提纲

- 介绍
- 整体架构
- 编程语言模型
- 关键技术
- 实验分析
- 结束语

介绍——背景

- **自动编程是智能软件的核心挑战之一,最早可追溯到Toward automatic program synthesis [Manna et al.,1971]**



介绍——问题提出——相关工作分析

- **基于执行轨迹方法**[Reed S et al.,2015;Jonathon Cai,2017;Da Xiao,2018]
  - u **本质是程序编码,没有习得新程序,轨迹标注成本高**
- **输入输出数据样例的方法**[Graves A,2014;Joulin A,2015;Balog M,2015;Deleu T,2016]
  - u **歧义性**
  - u **限定DSL**
  - u **无编程控制指令**
  - u **很难处理分支、递归等复杂控制结构**
  - u **泛化能力**
- **类自然语言源码挖掘**[Sim S E,2011;Dagenais B,2011;Lin Zeqi,2016;Nguyen A T,2016]
  - u **涉及注释、论坛、文档等,对代码自身信息的直接处理上表现不足**

介绍——问题提出——问题深层次思考总结

- 学习素材
  - 单一的以代码或输入输出数据构建的神经网络编程模型都存在局限性
- 编程语言跟自然语言
  - 程序跟自然语言差别大:结构、字面含义、统计含义
  - 源程序本身缺失大部分程序语义
- 编程者角度与机器角度
  - 编程角度习惯于以操作语义去描述程序的执行
    - 陷入编程语言复杂规范
    - 操作层面底层实现细节
    - 同时未能输入源程序的背景知识:程序员预学习的语言规范、计算机原理教程等

介绍——问题提出——我们认为

一种思路：

1.提升抽象层次

- 提升程序处理的抽象层次是本文整个研究的基本动机
- 在更抽象的层次构建模型
  - 从纯机器演算角度建模,避免过多细节或者信息不完备:公理语义
  - 用公里语义的方式将程序执行描述为抽象机的状态转换

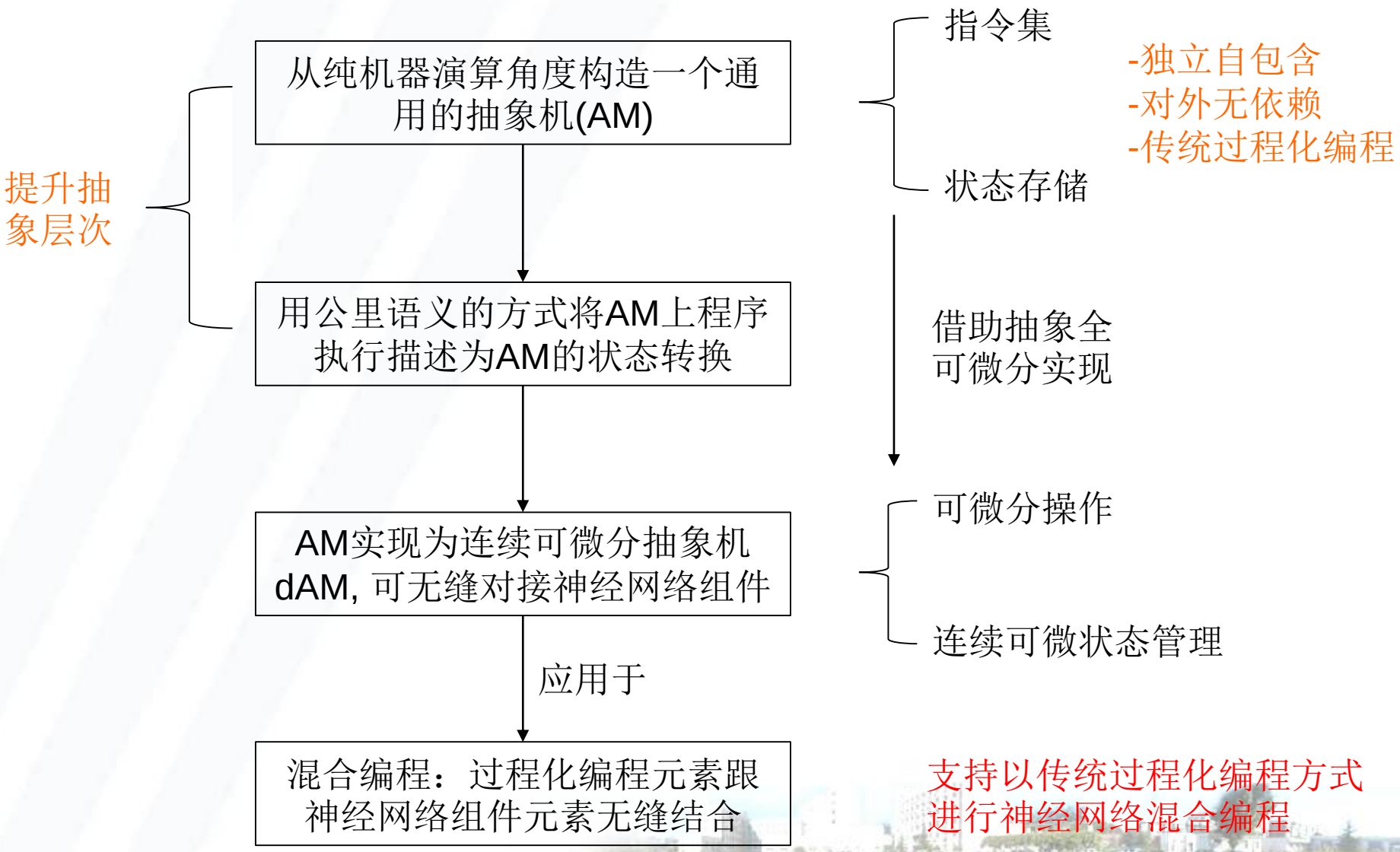
2.在抽象提升的基础上,实现

- 数据驱动神经网络学习+编程控制结合: 全可微分混合编程

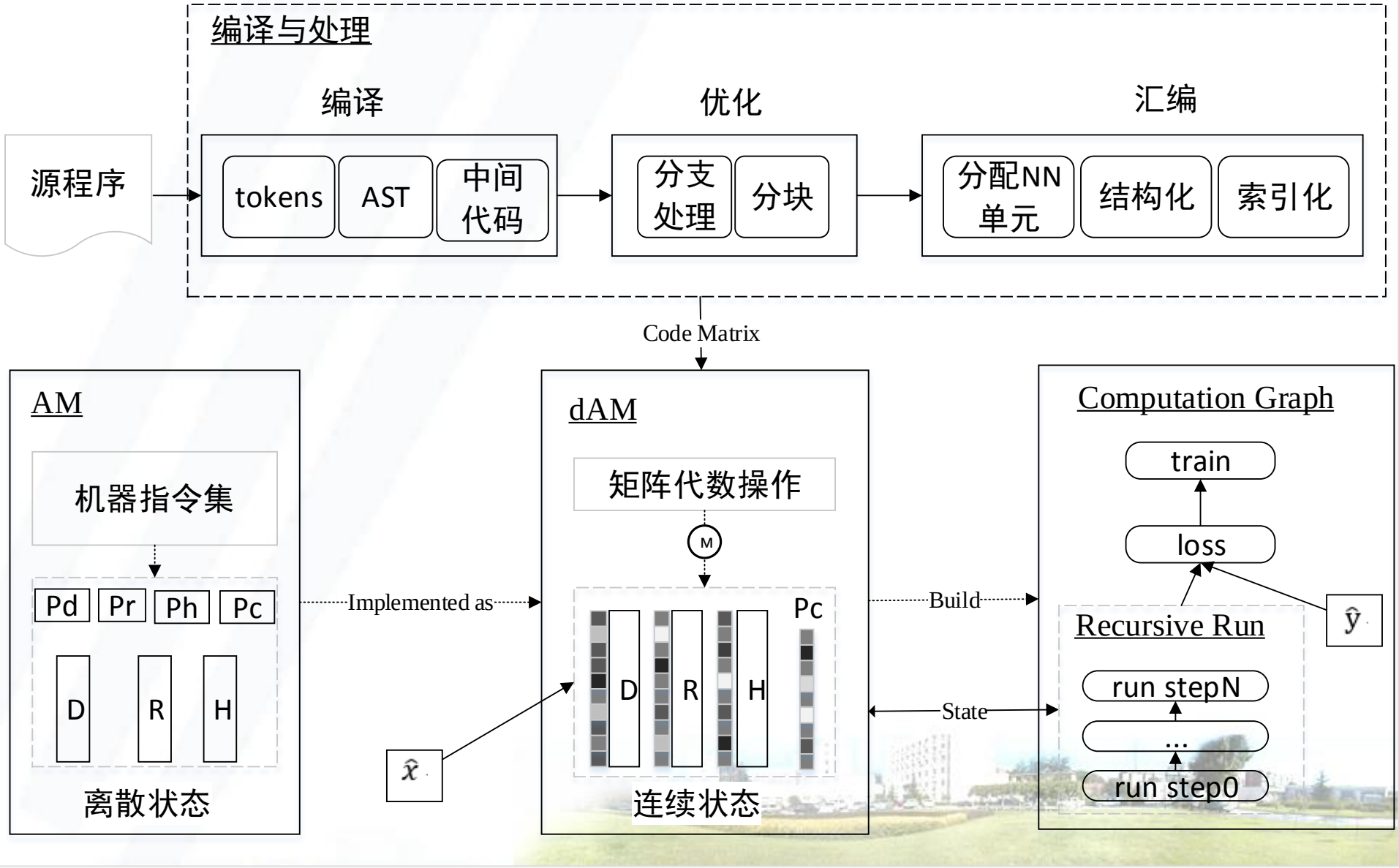
提纲

- 介绍
- 整体架构
- 编程语言模型
- 关键技术
- 实验分析
- 结束语

一般思路


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AM上程序执行描述为AM的状态转换
AM实现为连续可微分抽象机dAM, 可无缝对接神经网络组件
应用于
混合编程：过程化编程元素跟神经网络组件元素无缝结合
提升抽象层次
指令集
状态存储
-独立自包含
-对外无依赖
-传统过程化编程
借助抽象全可微分实现
可微分操作
连续可微状态管理
支持以传统过程化编程方式进行神经网络混合编程

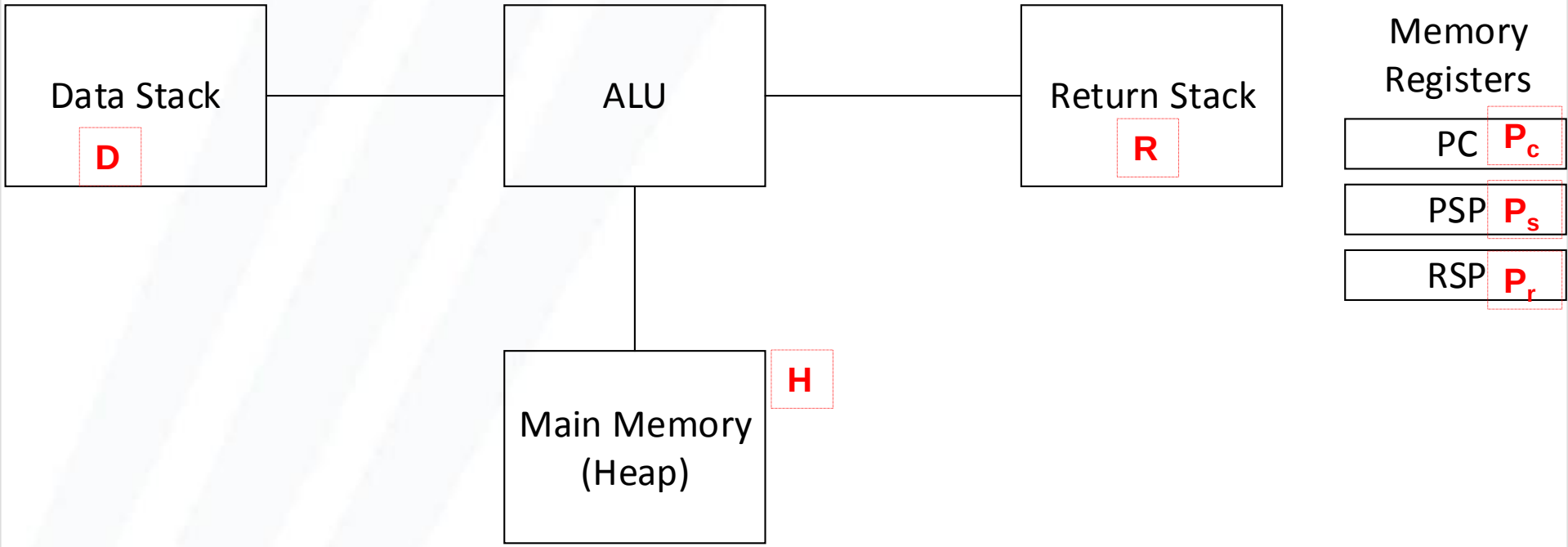
dAMP整体架构


编译与处理
编译
优化
汇编
源程序
tokens
AST
中间代码
分支处理
分块
分配NN单元
结构化
索引化
Code Matrix
AM
机器指令集
Pd
Pr
Ph
Pc
D
R
H
离散状态
Implemented as
dAM
矩阵代数操作
M
Pc
D
R
H
连续状态
Build
State
Computation Graph
train
loss
Recursive Run
run stepN
...
run step0

提纲

- 介绍
- 整体架构
- 编程语言模型
- 关键技术
- 实验分析
- 结束语

编程语言模型介绍(AM)——栈式运行语言



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

提纲

- 介绍
- 整体架构
- 编程语言模型
- 关键技术
- 实验分析
- 结束语

可微分抽象机实现——程序执行的抽象表示

- 程序执行：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抽象机状态单元的改变
  - 指令在AM上的执行表示为状态的迁移:

$$(D, R, Pc, Pd, Pr) \rightarrow (D', R', Pc', Pd', Pr')$$

- 直接的,将离散状态空间映射到dAM的连续状态空间
  - dAM状态: $s_d=(D_d, R_d, Pc_d, Pd_d, Pr_d)$
  - dAM指令执行: $f_{dam}: S \rightarrow S$
  - 抽象机指令word看做改变dAM状态的函数： $w \in f_{dam}$

可微分抽象机实现——连续可微分实现

- 状态操作连续可微分实现
  - 内存访问:
    - 类似NTM，用attention机制
      - 把栈指针看成内存单元权重, 实现READ和WRITE基本操作
      - 然后READ和WRITE线性组合实现PUSH、POP等高级操作
  - Word指令连续可微分实现:
    - 矩阵代数运算来表示word操作,比如基本指令集可分为
      - 访问抽象机状态单元,实现原理同内存访问
      - 算术逻辑运算, 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

源程序语法分析(定义BNF语法产生式)

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

源程序翻译成中间代码

算法 1. *ast2im(ast, labelAllocator)*

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
        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
```

部分中间指令

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.

中间代码优化

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) 算法2 // 第 1 遍

conRenumberedIMCode, conTable = conRenumber(labRenumberedIMCode) 算法3 // 第 2 遍

imBlocks = blockPartion(conRenumberedIMCode) 算法4 // 第 3 遍

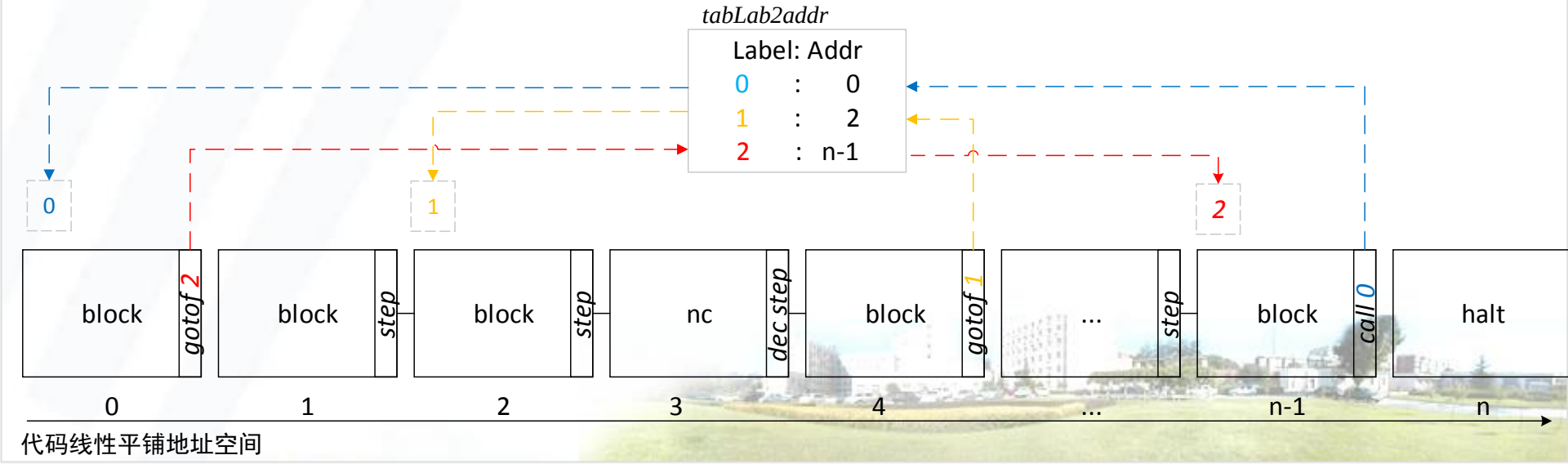
addrIM, tabLab2addr = lab2addr(imBlocks) 算法5 // 第 4 遍

optimisedIM = insertStepctl(tabLab2addr) 算法6 // 第 5 遍

RETURN (*optimisedIM, tabLab2addr, conTable, labTable*)

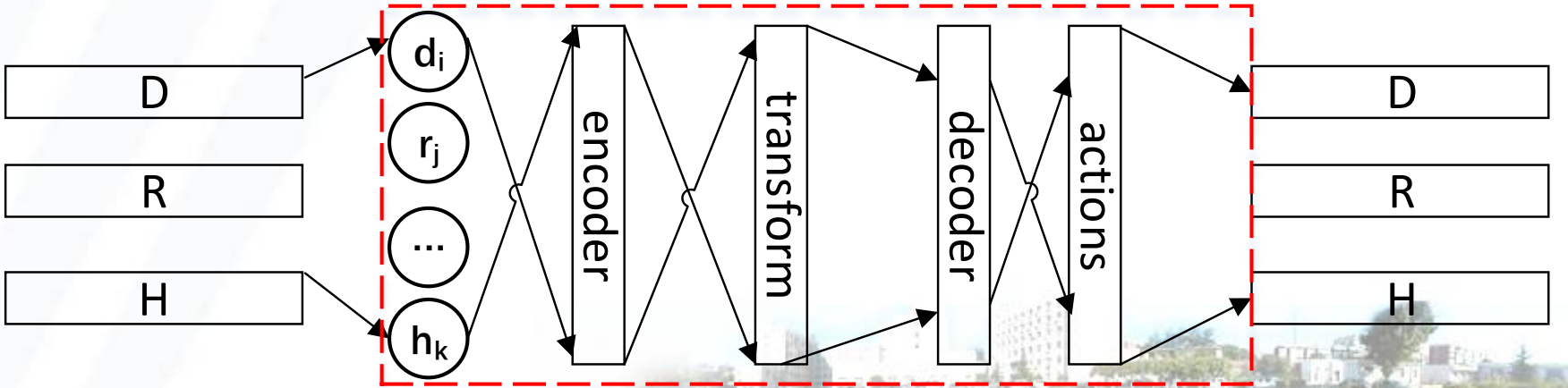
中间代码优化的目的

- 源程序到中间代码
  - 源程序以中间代码的形式在dAM上执行,中间代码非常适合转换为逐步的连续可微分执行
  - 中间代码指令功能单一、支持机械化微码求值实现
- 中间代码优化
  - 提高中间代码在dAM上执行效率
  - 提高dAM计算图的训练学习效率



程序汇编

- 汇编处理就是为每个block生成一个函数
  - 直接操作dAM状态的计算函数,在程序执行时这些函数被调用,生成计算图
- 代码块分为普通block代码块和神经网络组件(nc)
  - 普通块对应的汇编函数是矩阵的算术运算
  - NC块的汇编函数是带神经网络(待学习参数)的矩阵运算



程序可微分执行模型

- 程序的一次执行轨迹是从初始状态出发的状态列表

$exeTrace = [S_{dam\_init}, S_{dam\_1}, S_{dam\_2}, \ldots, S_{dam\_n}]$,S_{dam_init}是初始状态

- 单步执行的基本动作:

1. 根据Pc向量从代码矩阵中进行指令寻址计算

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

2. 基于C_{sel_w}选择指令执行

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

- 程序执行过程:从初始状态起,在当前状态下执行选定汇编指令, 得到的新状态就是下一步的当前状态,因此整个执行过程构成n步RNN模型

训练目标函数

- 模型使用结束状态进行训练,因此不需要逐步标注
- 模型训练目标：
  - u 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^*

$$\theta^* = ARG\ MIN_\theta \left(\sum_i ||y_i - \hat{y}_{end\ i}||_2\right)$$

  - u 用交叉熵描述：
    - ❖ 对所有的训练样本*i* 求解使得样本$\hat{y}_{end}$跟模型估值y的交叉熵最小的模型参数θ^*

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

    - ❖ 按照状态分量D、P_d、R、P_r、H 分解:

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

提纲

- 介绍
- 整体架构
- 编程语言模型
- 关键技术
- 实验分析
- 结束语

Forth实现参照

```
: 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,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
```

20行/个编程指令

dAMP混合编程实现

```
1: multiAdd (--)
   p_global @ n_global @ = if 2
     exit \return
2  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))%>
5    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 4
2  then
   ;
3  multiAdd
   carry @ . sum @ . \print the sum
```

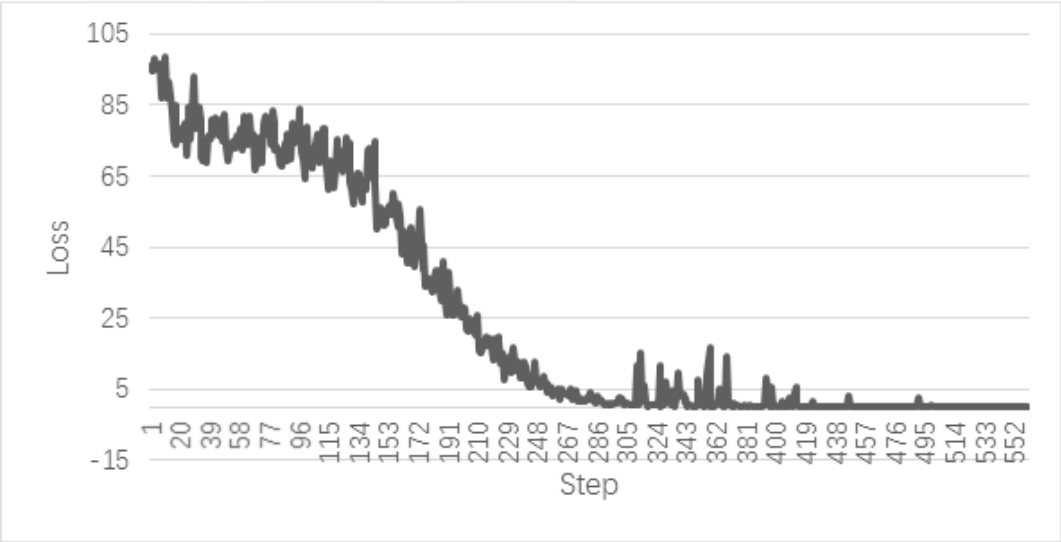
Neural Component

实验样例选取原因与目的

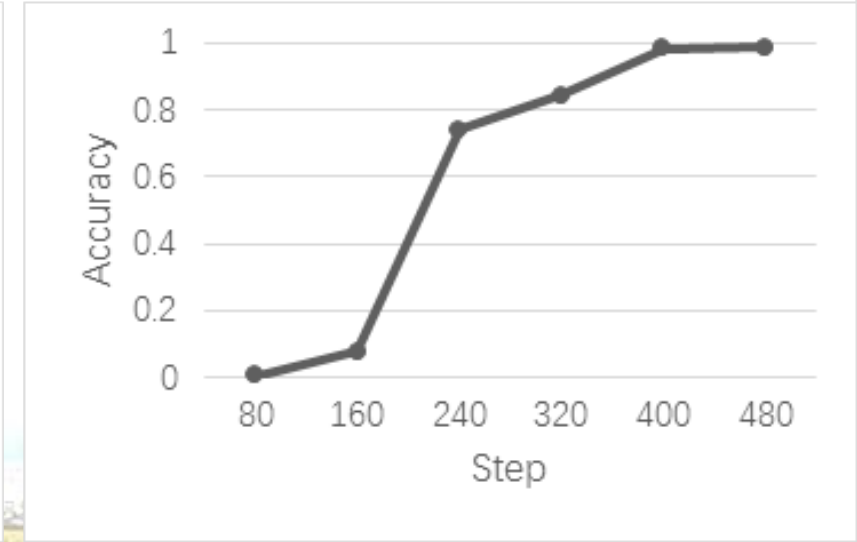
- 多位数加法的递归算法
  - 单位数加法递归实现多位数加法
  - 选取的原因与目的
    - 简化问题,但不简化程序,因此
      - 方便进行简单明了的多方面系统分析
    - 包含了顺序语句、分支控制、函数调用、递归等程序结构,这些结构是程序处理的复杂问题
      - 显然这些结构能表达其他广泛、复杂的编程问题
      - 不失对dAMP全可微分模型编程能力验证
    - 分支、函数调用、递归等控制结构处理能力也是程序生成研究关键之一[Graves A,2014;Joulin A,2015;Balog M,2015;Deleu T,2016]
      - 具有代表性和挑战性

模型训练效果分析——收敛性、准确率、泛化

- 训练样例长度为2,测试4到8位整数加法
  - u 训练400步后Loss收敛到非常接近0, 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



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



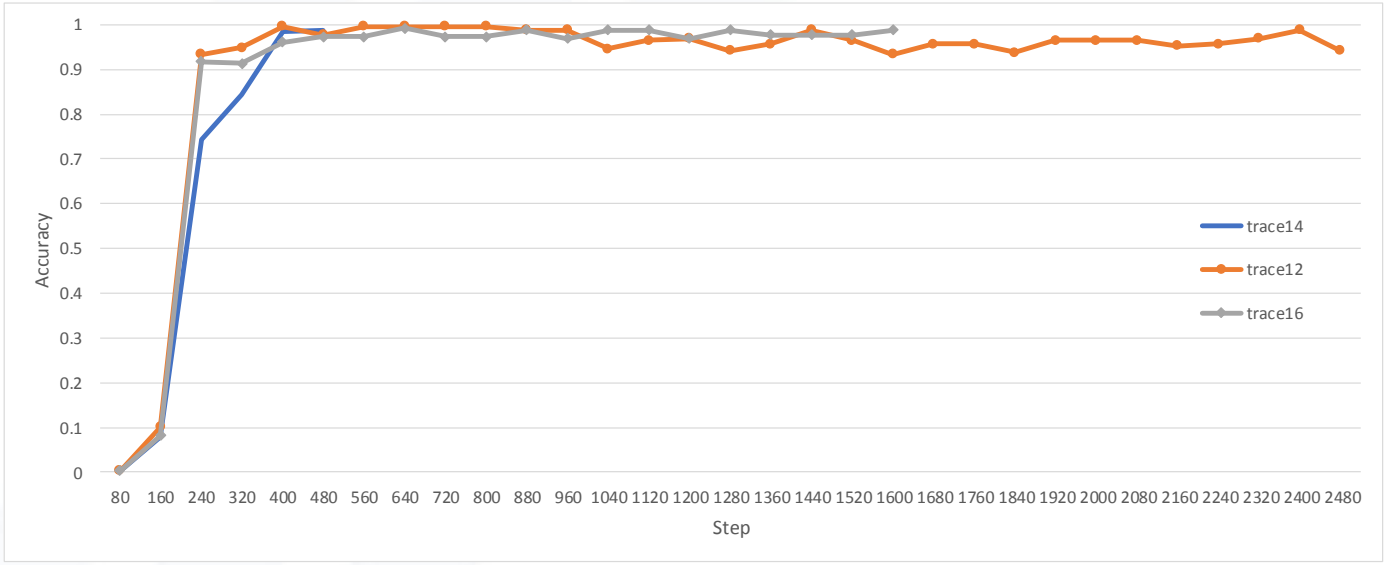
(b)Acc rate for length of training 2&test 8

模型训练效果分析——执行轨迹步数设置对训练的影响

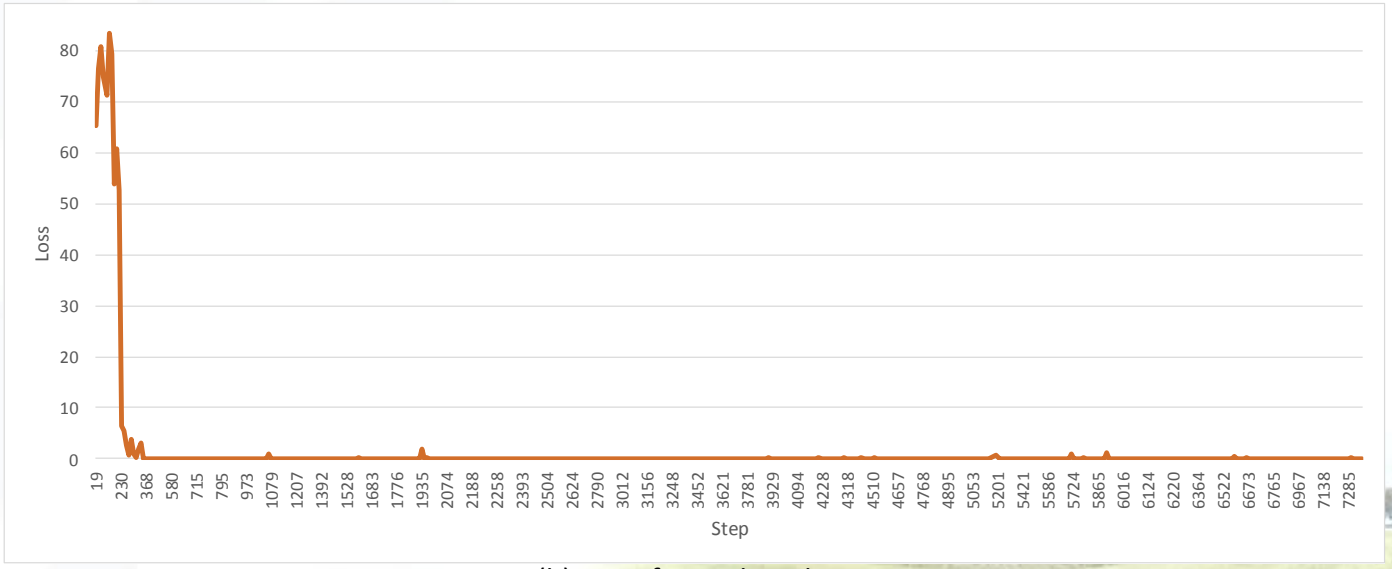
- 设置合适的状态迁移参数*exeTrace*[n]对模型训练很关键
  - n值需参照 优化后代码块级程序结构、程序输入规模

- 如图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(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

模型训练效果分析——执行轨迹步数设置对训练的影响



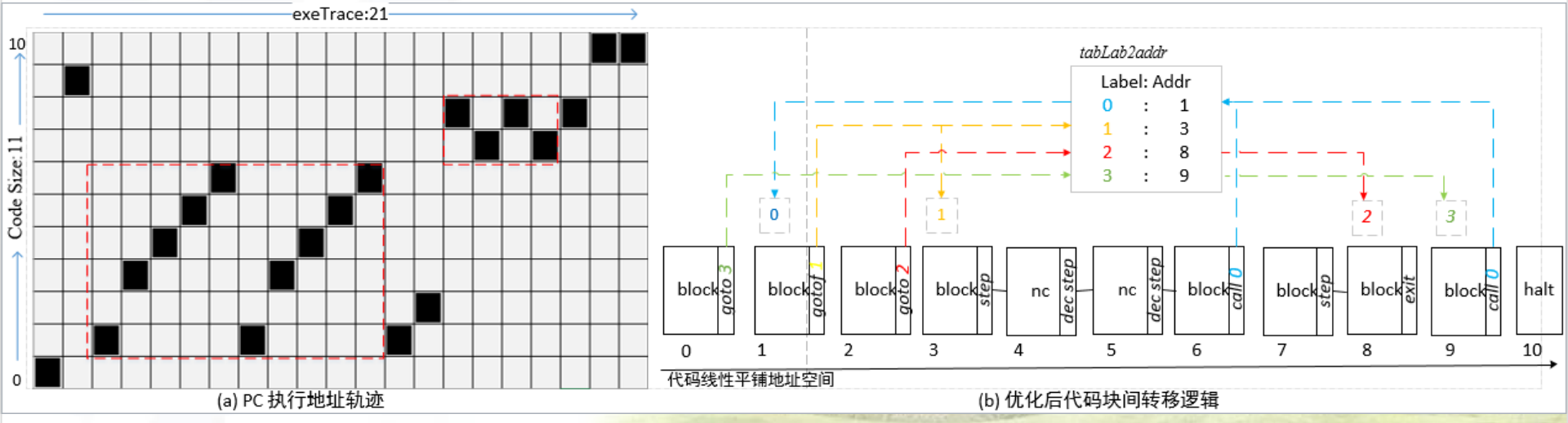
(a)Accuracy of trace length 12,14&16



(b)Loss of trace length 12

运行时执行过程正确性验证

- 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
  - 从图可以看出运行时PC移动轨迹符合预期
  - 虚框部分表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.



(a) PC 执行地址轨迹

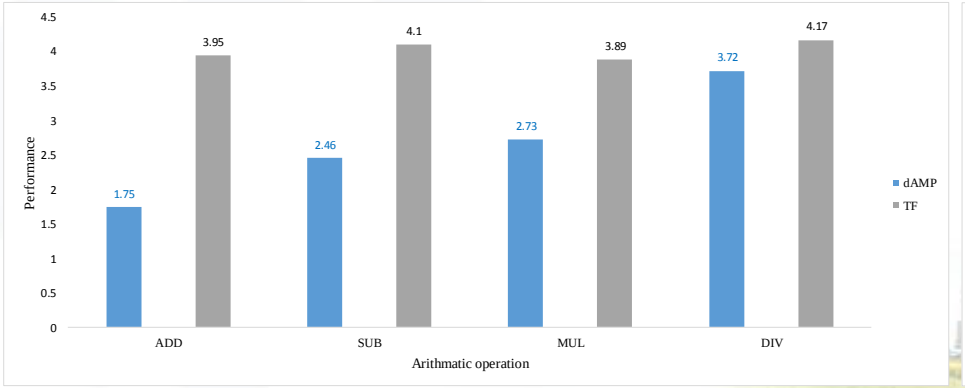
(b) 优化后代码块间转移逻辑

执行轨迹模型复杂度分析

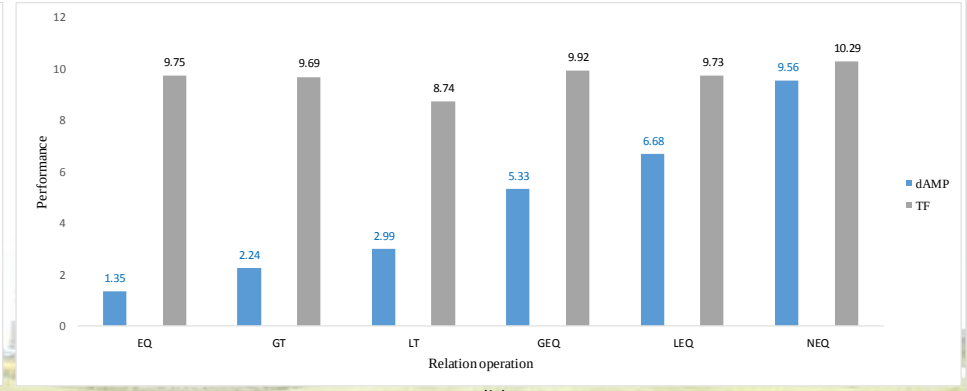
- 程序在dAMP上执行复杂度主要由n步exeTrace决定
- multiAdd为例:
  - 输入规模m,则n=2+5m+2+2m +2=7m+6=O(m)，线性复杂度.
  - 类似的,对于冒泡排序其模型复杂度是$O(n^2)$
  - 所以dAMP运行时轨迹复杂度取决于算法框架本身
- 优化跟复杂度的关系
  - 分块优化:提升执行性能,未改变程序结构因此不会改进复杂度
  - 梯度计算放在程序执行结尾状态而不是执行过程中每个状态
- 用nc处理控制结构
  - 是否能消除分支(可以)、循环、递归结构从而达到优化算法复杂度还有待未来做专门探讨

性能评估

- 开发测试环境:Python 3.5.2、NumPy 1.14.5、TensorFlow 1.10.0 CPU版、Ubuntu 16.04 X86_64, 8核心Intel Xeon E5-2407处理器、64G内存
- dAMP性能横向评估
  - 选取TensorFlow功能相近的操作,使用相同的数据做测试对照.
  - TF.equal、TF.greater、TF.less、TF.greater_equal、TF.less_equal、TF.not_equal
  - TF.add、TF.subtract、TF.multiply、TF.div



(a)



(b)

提纲

- 介绍
- 整体架构
- 编程语言模型
- 关键技术
- 实验分析
- 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
- 给出该思路的一个验证系统dAMP
  - 实现了一种可无缝结合过程化编程语言与神经网络组件的混合编程模型
- 实验分析
  - 包含顺序块、条件分支、函数调用、递归等复杂程序结构和神经网络组件程序样例,表明具备表达复杂程序的能力
  - 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
- 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

下一步探讨

如何走向广泛的实际编程应用：

1.类C语法作前端

- 通过扩展C语言子集,设计类C语法前端,便于程序员编程思想表达

2.编程语言原语设计

- 设计丰富的混合编程语言原语,以增强其编程表达能力

3.全面正向探讨或反向总结dAMP编程潜力

- 传统C程序是确定性的:编写完整的程序,求解完全明确的问题
- 本模型支持引入非确定性编程支持,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
  - 非完整编程:出于降低编程难度或提高编程效率

谢谢！
Thanks!