《人工神经网络》大作业报告——自主选题

李欣隆*
2018010324
计92
xl-li18@mails.tsinghua.edu.cn

陈炫中
2019011236
计92
xz-chen19@mails.tsinghua.edu.cn

甘乔尹
2019011240
计92
ganqy19@mails.tsinghua.edu.cn

1 简介

表达式求值是一类推理问题，其中含有许多不同的具体任务，我们设计了一个新的数据集，其中包含了一些不同的表达式求值任务的例子，模型需要通过每个任务的例子学会对应任务。

这个问题的研究背景为神经网络是否能在不引入过强先验的情况下通过例子学习到表达式求值类任务。我们认为表达式求值任务是一种重要的逻辑推理任务，并且认为其能够做为衡量模型在递归深度上的泛化能力的重要依据。研究价值在于这个数据集能够衡量一个模型是否能从数据中学到一个非平凡的带有递归的程序。

这个问题的难点在于数据集中包含许多不同任务，模型很难在所有数据集中表现都较好，并且不同的任务可能要求不同形式的递归。同时我们的数据集关注于模型在推理深度上的泛化，可能包含长度很长的表达式，实验表明transformer 模型在长度较长的表达式上正确率比短的表达式显著更低。关于现有方法的缺陷我们会在相关工作中详细论述。

本工作的创新点为三个部分：

(1)我们提出了新的任务，新的数据集。

(2)我们测试了transformer 模型在任务上的表现，并探究了其在数据规模变化时的泛化能力。

(3)在部分表达式求值任务中，我们在数据末尾增加了中间结果的标注，并测试相比于无中间结果标注的数据，模型的正确率能否获得较大提升。

2 相关工作

目前与我们的任务相关性较大的研究有四类：

1. 使用语言模型求解应用题，如OpenAI 最近研究了使用微调后的GPT-3 模型解决小学数学题[1]。他们的任务和我们的共同点是都使用了带中间结果标注的数据来辅助模型学习，区别是我们的任务是符号推理任务，不需要自然语言；我们的任务更关注递归深度上的泛化能力，而他们的任务中数字都较小。

2. 使用seq2seq 模型求解简单微积分[2] 等表达式求值类问题，他们的任务和我们的共同点是都属于符号推理任务，不需要自然语言。他们的任务和我们的区别是我们的任务更关注递归深度上的泛化能力；而他们的任务中系数都非常小，原文中描述为$-10, 10$ 之间的整数。

*请将组长放到第一个位置

3. 使用seq2seq 模型实现自然语言到代码的映射，如使用带有抽象语法树先验的transformer模型来实现自然语言到代码的翻译[3]。他们的任务更偏向于机器翻译任务，需要有一个较为清晰的自然语言描述，才能将该描述直接翻译成代码，而我们的模型侧重于通过给出例子的中间结果来学习。

4. 使用带有启发式的与或图搜索框架实现基于例子的程序生成，如[4]。他们的数据集中每个任务有一些例子，例子中指名一些允许使用的基本操作函数，以及一些形如$f(x_1, x_2, ...) = y$ 的限制，模型需要搜索到一份只允许使用给定的基本操作函数的代码，使得能通过限制。

Example 5 (Programming by Examples (PBE) with Strings)**.** Consider the following example:

```
(set-logic PBE_SLIA)
(synth-fun f ((fname String) (lname String)) String
  ((y_str String) (y_int Int))
  ((y_str String (" " fname lname
                   (str.++ y_str y_str)
                   (str.replace y_str y_str y_str)
                   (str.at y_str y_int)
                   (str.from_int y_int)
                   (str.substr y_str y_int y_int)))
     (y_int Int (0 1 2
                 (+ y_int y_int)
                 (- y_int y_int)
                 (str.len y_str)
                 (str.to_int y_str)
                 (str.indexof y_str y_str y_int)))))
(constraint (= (f "Nancy" "FreeHafer") "Nancy FreeHafer"))
(constraint (= (f "Andrew" "Cencici") "Andrew Cencici"))
(constraint (= (f "Jan" "Kotas") "Jan Kotas"))
(constraint (= (f "Mariya" "Sergienko") "Mariya Sergienko"))
(check-synth)
```

Figure 1: 字符串拼接的例子

他们的任务和我们的共同点是都属于符号推理任务，不需要自然语言。他们的任务和我们的区别是这一类搜索框架中一部分框架是不可学习的，另一部分只能学习搜索一个分支的概率，学习能力较差；他们的搜索框架不需要搜索出递归的代码（如循环和递归的函数调用），只需要利用给定的基本操作，这些基本操作中有一部分是支持传入参数后递归的；同时为了实现与或图搜索，他们需要人工设计的针对给定的基本操作的反函数。

5. 神经符号方法，这里描述一个专门针对方便并行的任务设计的，类似神经图灵机的方法[5]。这个方法可以在与我们数据集类似的数据集上训练，在二进制加法和乘法任务上取得了很好的表现，在2000 个长度不超过20 的数据上训练后测试长度1000 的数据，并且取得了1 的正确率。他们的模型是专门用来处理方便并行的任务的，引入的结构先验对加法，乘法以及简单字符串操作相当有效。但他们只测试了二进制加法，乘法，字符串翻转，拼接任务，在最重要的表达式求值任务上没有经过测试，并且因为表达式求值不能简单地并行处理，我们认为这个模型不会有好的表现。

3 数据集

我们主要的工作为设计了新的任务和数据集。数据集为我们自己设计的基于例子的表达式求值数据集，数据集中包含许多不同的具体任务。对每种任务有对应该任务的一些数据，数据是以例子的形式给出的。可以对每个具体任务训练神经网络并测试，也可以将所有数据一起训练。

数据集中包含一些不同的任务：

布尔表达式求值bool_expr：输入为一个包含0,1 以及not，xor，and，or 四种运算符的布尔表达式，输出为0 或1。

二进制比较bin_cmp：输入为两个低位到高位的二进制串，输出为<,>,=。

二进制加法bin_add：输入两个低位到高位的二进制串，输出一个低位到高位的二进制串表示和。

二进制异或bin_xor：输入两个低位到高位的二进制串，输出一个低位到高位的二进制串表示异或和。

二进制乘法bin_mul：输入两个低位到高位的二进制串，输出一个低位到高位的二进制串表示乘积。

二进制表达式求值bin_add_mul：输入一个含有括号，加号，乘号的表达式，输出一个二进制串表示表达式求值的答案。

二进制字符串（翻转）连接concat,rev_concat：输入两个二进制串，输出为两个字符串连接后的结果或将第一个字符串翻转后和第二个字符串连接的结果。

每个任务的测试数据有多个测试集，其中一个测试集和训练集同分布且平均长度基本相等，其他测试集（Extratest）可能平均长度更长或有更复杂的结构（仅限布尔，二进制表达式求值）。数据的长度服从指数分布。

这里给出对其中部分任务的形式化描述:

3.1 二进制字符串（翻转）连接

这个任务考虑字符串拼接（concat）、字符串翻转（rev）、将第一个字符串翻转后和第二个字符串拼接（rev-concat）三个操作。

使用上下文无关文法描述这个任务需要考虑的表达式:
expr ::= "concat" expr expr | "rev-concat" expr expr | "rev" expr | data
data ::= "data" str
str ::= "0" str | "1" str | <空串>

表达式expr 的值是递归定义的:
eval("data" str)=str
eval("concat" expr1 expr2) 定义为eval(expr1) 和eval(expr2) 的字符串拼接
eval("rev-concat" expr1 expr2) 定义为eval(expr1) 翻转后和eval(expr2) 的字符串拼接
eval("rev" expr1)=eval("rev-concat" expr1 ("data"))

任务的输入是表达式expr，输出是字符串表示eval(expr)。

("rev-concat" ("0") ("1" "0" "0" "1" "0" "1" "1" "1" "1")) ?= ("0" "1" "0" "0" "1" "0" "1" "1" "1" "1")
("rev-concat" ("0" "0" "1" "0" "1" "0" "0" "0" "0" "1") ("0" "0" "1")) ?= ("1" "0" "0" "0" "0" "1" "0" "1" "0" "0" "0" "0" "1")
("rev-concat" ("1" "1" "1" "0" "1" "1" "0" "1" "1") ("0" "1" "0")) ?= ("1" "1" "0" "1" "1" "0" "1" "1" "1" "0" "1" "0")
("rev-concat" ("1" "0" "0" "0" "0" "0" "1" "1" "1") ("1" "1" "0")) ?= ("1" "1" "1" "0" "0" "0" "0" "0" "1" "1" "1" "0")
("rev-concat" ("0" "0" "0") ("1" "1" "0" "0" "0" "0" "1" "0" "0" "1" "0")) ?= ("0" "0" "0" "1" "1" "0" "0" "0" "0" "1" "0" "0" "1" "0")
("rev-concat" ("0" "0" "1" "1" "1" "0" "1" "1" "0" "0") ("0" "0" "1")) ?= ("0" "0" "1" "1" "0" "1" "1" "1" "0" "0" "0" "1")
("rev-concat" ("0" "0" "0" "1" "1" "0" "1" "0" "0" "1") ("1" "0" "1" "0" "1" "0" "0" "1")) ?= ("1" "0" "0" "1" "0" "1" "1" "0" "0" "0" "1" "0" "1" "0" "1" "0" "0" "1")
("rev-concat" ("0" "0" "1" "0" "0" "1" "0" "0" "1" "0" "0" "0") ("0" "1" "1")) ?= ("0" "0" "0" "1" "0" "0" "1" "0" "0" "1" "0" "0" "0" "1" "1")
("rev-concat" () ("0" "0" "1" "0" "0" "0" "1" "0" "0" "0" "1" "0" "1" "0" "1")) ?= ("0" "0" "1" "0" "0" "0" "1" "0" "0" "0" "1" "0" "1" "0" "1")

Figure 2: 二进制字符串（翻转）连接

3.2 布尔表达式求值

使用上下文无关文法描述这个任务需要考虑的表达式:
expr ::= "and" expr expr | "or" expr expr | "xor" expr expr | "not" expr | data
data ::= "data" "0" | "data" "1"

表达式expr的值eval(expr)是递归地定义的:
eval("data" "0") = 0
eval("data" "1") = 1
eval("and" expr1 expr2) = min(eval(expr1),eval(expr2))
eval("or" expr1 expr2) = max(eval(expr1),eval(expr2))
eval("xor" expr1 expr2) = (eval(expr1)+eval(expr2))eval("not" expr) = 1-eval(expr)

任务的输入是表达式expr，输出是0或1表示eval(expr)的值。

```
("not" ("not" ("xor" ("data" "1") ("or" ("data" "1") ("or" ("xor" ("data" "1") ("data" "0")) ("data" "1"))))) ?= ("data" "0")
("xor" ("xor" ("not" ("data" "1")) ("not" ("and" ("and" ("data" "1") ("data" "1")) ("xor" ("data" "0") ("or" ("data" "1") ("data" "0")))))) ("data" "1")) ?= ("data" "1")
("or" ("not" ("or" ("not" ("and" ("not" ("data" "1")) ("data" "0"))) ("data" "0"))) ("data" "1")) ?= ("data" "1")
("and" ("data" "0") ("or" ("and" ("data" "1") ("not" ("or" ("data" "0") ("or" ("data" "1") ("data" "0"))))) ("data" "0"))) ?= ("data" "0")
("and" ("or" ("not" ("xor" ("not" ("data" "1")) ("or" ("not" ("not" ("data" "0"))) ("data" "0")))) ("or" ("data" "1") ("data" "0"))) ("and" ("data" "1") ("data" "0"))) ?= ("data" "0")
("and" ("xor" ("not" ("data" "1")) ("data" "0")) ("xor" ("data" "1") ("data" "0"))) ?= ("data" "0")
("or" ("and" ("and" ("data" "0") ("and" ("or" ("data" "1") ("data" "1")) ("data" "1"))) ("data" "1")) ("data" "0")) ?= ("data" "0")
("xor" ("and" ("not" ("data" "1")) ("and" ("xor" ("data" "0") ("xor" ("data" "0") ("data" "1"))) ("data" "1"))) ("data" "0")) ?= ("data" "0")
```

Figure 3: 布尔表达式求值

3.3 二进制加法和乘法

这个任务考虑二进制表示的整数的加法和乘法:
expr ::= "+" expr expr | "*" expr expr | number
number ::= "data" "0" | "data" positive
positive ::= "0" positive | "1" positive | "1"

表达式expr的值eval(expr)是递归地定义的:
eval("data" "0") = 0
eval("data" positive) = eval(positive)
eval("0" positive) = 2eval(positive)
eval("1" positive) = 2eval(positive)+1
eval("1") = 1
eval("+" expr1 expr2) = eval(expr1)+eval(expr2)
eval("*" expr1 expr2) = eval(expr1)+eval(expr2)
eval("xor" expr1 expr2) = (eval(expr1)+eval(expr2))%2 eval("not" expr) = 1-eval(expr)

```
("bin-add" ("1" "0" "1" "0" "1") ("1" "1" "1" "0" "1" "1" "1" "1")) ?= ("0" "0" "1" "1" "0" "0" "0" "0" "1")
("bin-add" () ("1" "1" "0" "1" "1" "1" "1" "0" "1" "1" "0" "1" "0" "0" "0" "0" "1" "0" "1" "1") ("1" "1" "0" "1" "1" "1" "1" "1" "0" "1" "1" "0" "1" "0" "0" "0" "1" "0" "1" "1" "0" "1") ?= ("1" "1" "0" "1" "1" "1" "1" "0" "1" "1" "0" "1" "0" "0" "0" "1" "0" "1" "1" "0" "0" "1")
("bin-add" ("0" "1" "0" "0" "1" "0" "1" "1" "1") ("1" "1" "1" "1" "1" "1" "1")) ?= ("1" "0" "0" "0" "1" "0" "0" "0" "1")
("bin-add" ("0" "0" "0" "1" "0" "0" "0" "0" "1" "0" "0" "0" "1" "1" "0" "0" "0" "1") ("0" "0" "0" "0" "0" "1" "0" "0" "0" "0" "0" "0" "0" "1")) ?= ("0" "0" "0" "1" "0" "1" "0" "0" "1" "0" "0" "0" "1" "0" "1" "0" "0" "1")
("bin-add" ("1" "1" "0" "1" "1") ("0" "0" "0" "0" "0" "1")) ?= ("1" "1" "0" "1" "1" "1")
("bin-add" ("1" "0" "0" "1" "1" "1" "1" "1" "1" "1") ("0" "1" "0" "1" "0" "1" "1" "1" "1" "1" "0" "1" "0" "0" "0" "0" "1" "1" "0" "0" "0" "1")) ?= ("1" "1" "0" "0" "0" "1" "1" "1" "1" "0" "0" "1" "0" "0" "0" "1" "1" "0" "0" "1")
("bin-add" ("1" "1" "0" "1" "0" "1" "0" "0" "1" "1" "1" "1" "0" "1" "1" "0" "1" "1") ("1" "0" "1")) ?= ("0" "0" "0" "0" "1" "1" "0" "0" "1" "1" "1" "1" "1" "0" "1" "1" "1")
("bin-add" ("1" "1" "0" "1" "0" "0" "0" "1") ("0" "0" "0" "0" "0" "0" "0" "1")) ?= ("1" "1" "0" "1" "1" "0" "0" "0" "1")
("bin-add" ("0" "0" "0" "0" "1" "0" "1") ("0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "0" "1" "0" "0" "1")) ?= ("0" "0" "0" "0" "1" "0" "1" "0" "0" "0" "0" "0" "0" "1" "0" "0" "1")
("bin-add" ("0" "0" "0" "1") ("0" "0" "0" "0" "1" "0" "0" "1" "0" "1")) ?= ("0" "0" "0" "1" "1" "0" "0" "1" "0" "1")
```

Figure 4: 二进制加法

```
("bin-mult" ("0" "1") ("1" "0" "1" "1" "1")) ?= ("0" "1" "0" "1" "1" "1")
("bin-mult" ("1" "1" "1" "1" "0" "1" "1" "1") ("1")) ?= ("1" "1" "1" "1" "0" "1" "1" "1")
("bin-mult" ("1" "1") ("0" "1" "1" "0" "0" "1")) ?= ("0" "1" "0" "0" "1" "1" "1")
("bin-mult" () ("1" "0" "0" "1" "0" "0" "0" "0" "0" "0" "0" "0" "1")) ?= ()
("bin-mult" () ("1" "1" "1" "1")) ?= ()
("bin-mult" ("1" "1" "1" "1" "1" "1" "1") ("1")) ?= ("1" "1" "1" "1" "1" "1" "1")
("bin-mult" () ()) ?= ()
("bin-mult" ("0" "1" "1" "0" "0" "0" "0" "1" "0" "1") ("0" "0" "0" "1")) ?= ("0" "0" "0" "0" "1" "1" "0" "0" "0" "0" "1" "0" "1")
("bin-mult" ("1" "0" "1") ("0" "1" "1" "0" "1" "1")) ?= ("0" "1" "1" "1" "0" "0" "0" "1" "1")
("bin-mult" ("0" "1") ("1" "0" "0" "1" "1" "0" "1" "0" "0" "1" "1" "1" "1" "1" "1" "1")) ?= ("0" "1" "0" "0" "1" "1" "0" "1" "0" "0" "1" "1" "1" "1" "1" "1")
("bin-mult" () ("0" "0" "1" "0" "0" "0" "1")) ?= ()
```

Figure 5: 二进制乘法

3.4 带中间结果的布尔表达式求值

对于and,or,xor,not，有形如
tip("and" expr1 expr2) = tip(expr1) ++ tip(expr2) ++ (("and" expr1 expr2) "=" eval("and" expr1 expr2) ";")
的提示信息，其中++为序列拼接
另外
tip("data" "0") = 空序列
tip("data" "1") = 空序列
任务的输入和布尔表达式求值相同，是表达式expr，输出是((tip(expr)) "[ans]" eval(expr))

```
("data" "0") ?= (() "[ans]" ("data" "0"))
("data" "1") ?= (() "[ans]" ("data" "1"))
("or" ("data" "0") ("data" "1")) ?= ((((("or" ("data" "0") ("data" "1")) "=" ("data" "1") ";")) "[ans]" ("data" "1"))
("not" ("data" "0")) ?= ((((("not" ("data" "0")) "=" ("data" "1") ";")) "[ans]" ("data" "1"))
("xor" ("or" ("data" "0") ("data" "1")) ("data" "0")) ?= ((((("or" ("data" "0") ("data" "1")) "=" ("data" "1") ";") ((("xor" ("or" ("data" "0") ("data" "1")) ("data" "0")) "=" ("data" "1") ";")) "[ans]" ("data" "1"))
```

Figure 6: 带中间结果的布尔表达式求值

4 实验

4.1 实验设置

数据集采用我们设计的表达式求值数据集，原始数据集需要经过简单处理后才能使用。

我们采用了最基本的transformer 模型（来自开源项目，详情见readme）来得到基线，主要测试了transformer 模型在训练集和测试集长度不同时的泛化能力，以及中间结果提示对模型表现的提升。

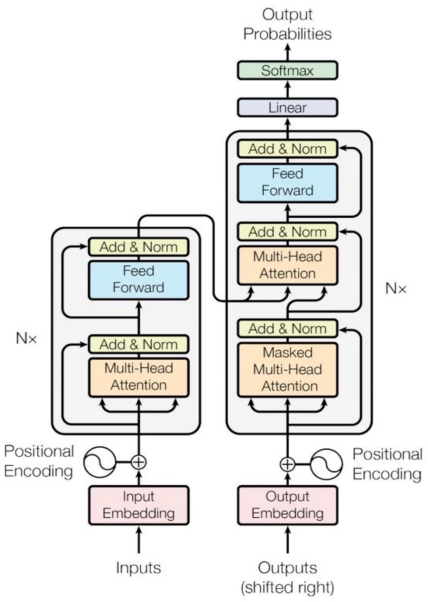

Figure 7: Transformer 模型

数据集	num_layers	epoch	batch_size
bool_expr	4/5	200	400
bin_cmp	4	200	400
bin_add	4	200	400
bin_mul	4	400	400
bin_add_mul	4	400	400
bitwise_xor	2	200	400
concat	4	100	400
rev_concat	4	200	400
bool_tip	4	200	400

Table 1: 模型的超参数

对于表达式求值的一组数据，输入X 和输出Y 都是序列，使用transformer 模型，有几种处理方式如下：

4.2 无额外空间的处理

encoder 输入X，decoder 输出Y [end]，对应基线方法。

此时模型必须将所有计算过程在|X|+|Y|步中完成，我们期望模型无法泛化到更大规模的数据，以及输出长度短而计算过程长的情况。

4.3 有中间结果提示的处理

encoder输入X，decoder输出S [sep] Y [end]，S 由人或程序生成标注。

我们期望模型能利用S，按类似于标注的方式输出表达式求值的中间结果，且通过显式地标注的中间结果，希望提高模型的泛化能力，同时模型可以在自回归时将中间结果当作额外内存使用。在布尔表达式求值任务中，由于答案只有一个字符，没有中间结果提示的模型只能进行常数步推导，比有中间结果提示的模型推理深度差很多。

另外，通过不同方式给出S 的标注也可能可以影响模型的表现。

4.4 实验结果

	Baseline	Extra	Baseline长度	Extra长度
bool_expr	0.7677	-	17.44	-
bin_cmp	0.9930	0.8221	23.03	35.18
bin_add	0.9960	0.6637	20.19	29.46
bin_mul	0.9430	0.5336	20.19	29.46
bin_add_mul	0.6634	0.2745	91.88	84.55
bitwise_xor	0.9963	0.7085	23.75	35.36
concat	0.9978	0.6545	18.11	27.25
rev_concat	0.9997	0.6650	18.11	27.15

Table 2: Baseline以及在Extratest 上的实验结果

可以看出模型在除了布尔表达式求值，二进制表达式求值任务之外的所有任务的基本数据上表现不错，但在平均长度更长的Extratest 上准确率大幅度下降。

4.5 加法和乘法任务

对二进制加法任务，我们设计了5 个平均长度不同的数据集，以其中一个为训练集进行训练，另一个为测试集进行测试，平均长度为31，48，85，156，225。

	1	2	3	4	5
1	0.9960	0.6217	0.2938	0.0917	0.0417
2	0.9978	0.9650	0.7567	0.3740	0.1267
3	0.9833	0.9680	0.9528	0.9067	0.7975
4	0.9771	0.9690	0.9575	0.9420	0.9225
5	0.9758	0.9627	0.9487	0.9434	0.9271

Table 3: 二进制加法任务

对二进制乘法任务，我们设计了6 个平均长度不同的测试集来测试模型对序列长度的泛化。

	正确率	长度
1	0.9430	36.78
2	0.4825	58.03
3	0.2367	93.60
4	0.1120	170.23
5	0.0102	310.63
6	0.0001	450.03

Table 4: 二进制乘法任务

4.6 中间结果提示的对比

任务	正确率	长度
无中间结果	0.7677	17.44
有中间结果	0.9791	17.44
有中间结果Extra	0.9376	26.56

Table 5: 中间结果提示的对比

通过实验结果可以发现中间结果可以显著增加模型在同长度数据下的表现，并且也能增加模型在训练数据和测试数据长度不同时的泛化能力。

5 总结

我们的工作为：

1. 提出了新的任务以及对应的数据集和Baseline。

2. 测试了transformer 模型在数据平均长度变长时的泛化能力，发现正确率显著下降。我们推测是因为transformer 模型通过位置编码来引入序列位置的先验，但这里对位置编码的操作需要学习，所以在不同长度的数据间难以泛化。

3. 在布尔表达式求值任务上引入中间结果提示，发现模型的正确率显著提高，我们推测是因为在没有中间结果提示的情况下，因为输出量只有一个字符，模型自回归的推理深度为常数，并且只有常数个字长的额外内存。在有中间结果提示的数据上模型能通过自回归获得足够的推理深度，并且中间结果可以引导模型计算。

我们项目的其他部分已经在https://github.com/chenxz1111/Expression-Transformer 上开源。

参考文献

References

[1] Cobbe, K., V. Kosaraju, M. Bavarian, et al. Training verifiers to solve math word problems. vol. abs/2110.14168. 2021.

[2] Lample, G., F. Charton. Deep learning for symbolic mathematics. 2020.

[3] Sun, Z., Q. Zhu, Y. Xiong, et al. Treegen: A tree-based transformer architecture for code generation. pages 8984–8991, 2020.

[4] Ji, R., Y. Sun, Y. Xiong, et al. Guiding dynamic programing via structural probability for accelerating programming by example. *Proc. ACM Program. Lang.*, 4(OOPSLA):224:1–224:29, 2020.

[5] Łukasz Kaiser, I. Sutskever. Neural gpus learn algorithms. 2016.