知行学堂

感知编程语言约束的深度学习模型

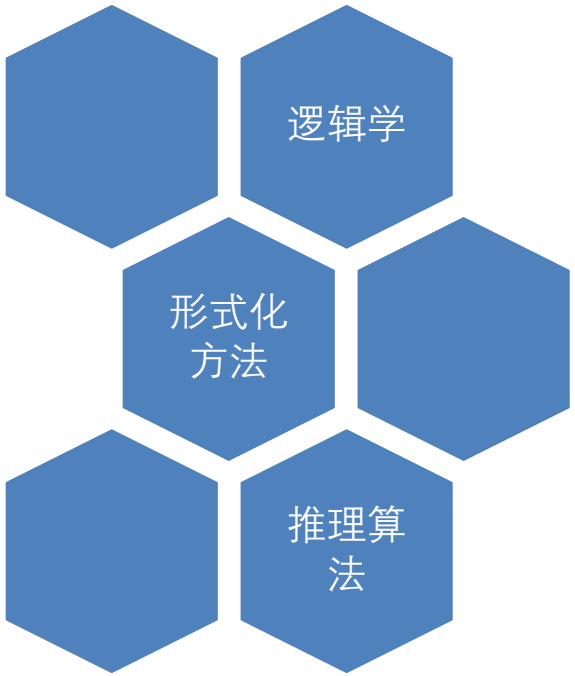
汇报人：熊英飞

北京大学计算机学院软件研究所
长聘副教授、研究员

人工智能的两条主要途径

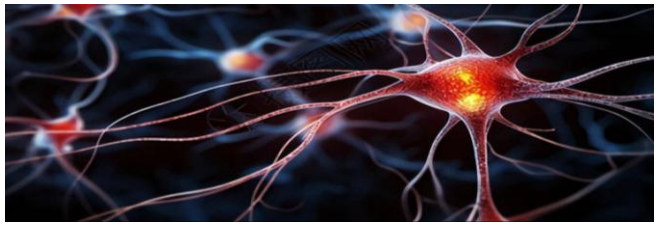
符号主义

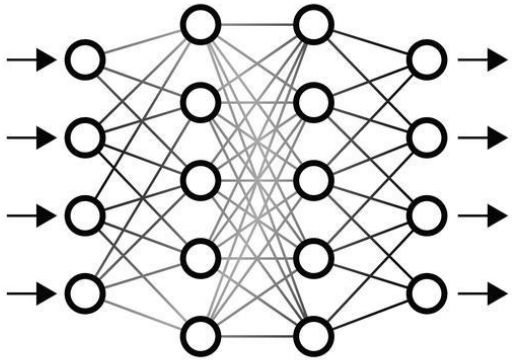
- 从人类的知识体系出发构建智能



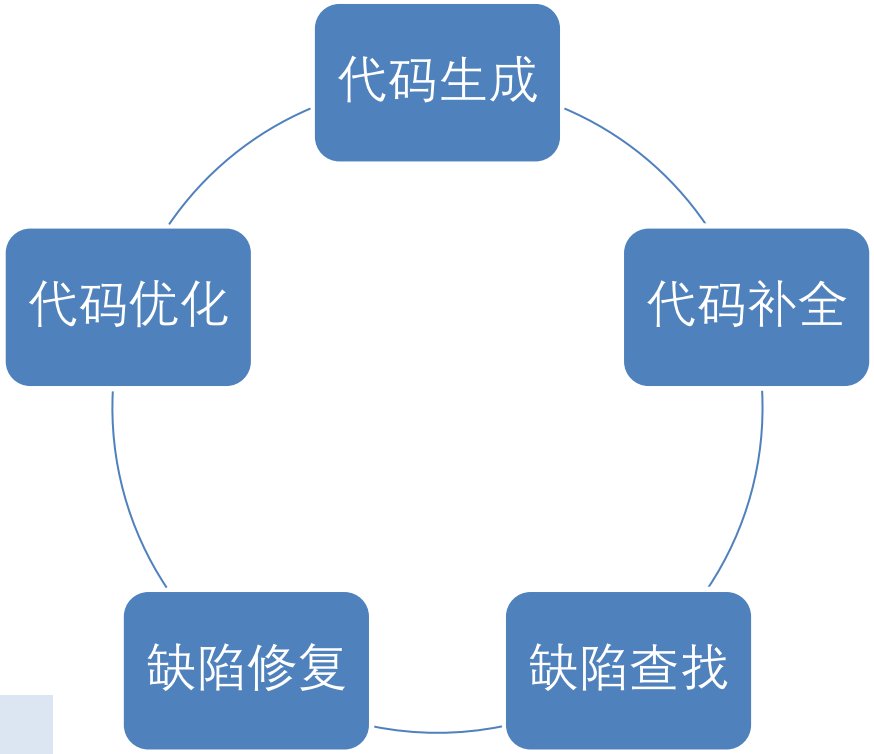
联结主义

- 从大自然进化的结果出发构造智能





辅助软件开发：当代AI最重要的应用之一



麦肯锡报告：采用生成式大模型辅助之后，软件研发成本可以节省 20%-45%。

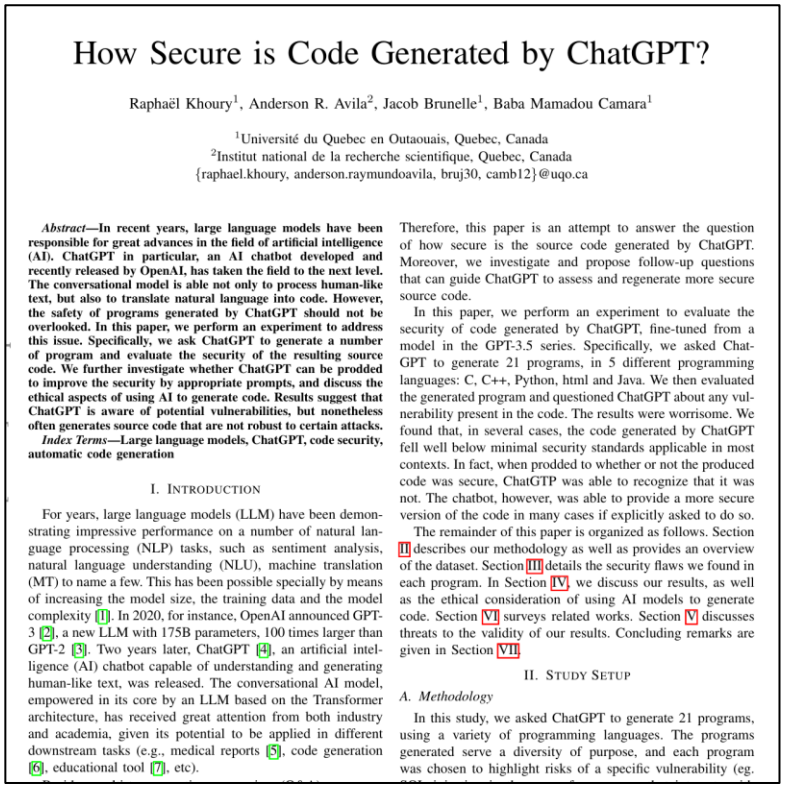
文心快码 Baidu Comate　通义灵码　deepseek　Copilot

联结主义和程序生成的矛盾

程序天然是符号系统，包含基于符号规则定义的

- 语法约束：()+5(不合法
- 类型约束：1+true不合法
- 语义约束：malloc分配的内存不调用free会导致泄漏

等，基于联结主义的神经网络难以学会这些规则，也难以基于规则推导。



How Secure is Code Generated by ChatGPT?

Raphaël Khoury[1], Anderson R. Avila[2], Jacob Brunelle[1], Baba Mamadou Camara[1]

[1]Université du Quebec en Outaouais, Quebec, Canada
[2]Institut national de la recherche scientifique, Quebec, Canada
{raphael.khoury, anderson.raymundoavila, bruj30, camb12}@uqo.ca

***Abstract*—In recent years, large language models have been responsible for great advances in the field of artificial intelligence (AI). ChatGPT in particular, an AI chatbot developed and recently released by OpenAI, has taken the field to the next level. The conversational model is able not only to process human-like text, but also to translate natural language into code. However, the safety of programs generated by ChatGPT should not be overlooked. In this paper, we perform an experiment to address this issue. Specifically, we ask ChatGPT to generate a number of program and evaluate the security of the resulting source code. We further investigate whether ChatGPT can be prodded to improve the security by appropriate prompts, and discuss the ethical aspects of using AI to generate code. Results suggest that ChatGPT is aware of potential vulnerabilities, but nonetheless often generates source code that are not robust to certain attacks.**

***Index Terms*—Large language models, ChatGPT, code security, automatic code generation**

I. Introduction

For years, large language models (LLM) have been demonstrating impressive performance on a number of natural language processing (NLP) tasks, such as sentiment analysis, natural language understanding (NLU), machine translation (MT) to name a few. This has been possible specially by means of increasing the model size, the training data and the model complexity [1]. In 2020, for instance, OpenAI announced GPT-3 [2], a new LLM with 175B parameters, 100 times larger than GPT-2 [3]. Two years later, ChatGPT [4], an artificial intelligence (AI) chatbot capable of understanding and generating human-like text, was released. The conversational AI model, empowered in its core by an LLM based on the Transformer architecture, has received great attention from both industry and academia, given its potential to be applied in different downstream tasks (e.g., medical reports [5], code generation [6], educational tool [7], etc).

Therefore, this paper is an attempt to answer the question of how secure is the source code generated by ChatGPT. Moreover, we investigate and propose follow-up questions that can guide ChatGPT to assess and regenerate more secure source code.

In this paper, we perform an experiment to evaluate the security of code generated by ChatGPT, fine-tuned from a model in the GPT-3.5 series. Specifically, we asked ChatGPT to generate 21 programs, in 5 different programming languages: C, C++, Python, html and Java. We then evaluated the generated program and questioned ChatGPT about any vulnerability present in the code. The results were worrisome. We found that, in several cases, the code generated by ChatGPT fell well below minimal security standards applicable in most contexts. In fact, when prodded to whether or not the produced code was secure, ChatGTP was able to recognize that it was not. The chatbot, however, was able to provide a more secure version of the code in many cases if explicitly asked to do so.

The remainder of this paper is organized as follows. Section II describes our methodology as well as provides an overview of the dataset. Section III details the security flaws we found in each program. In Section IV, we discuss our results, as well as the ethical consideration of using AI models to generate code. Section VI surveys related works. Section V discusses threats to the validity of our results. Concluding remarks are given in Section VII.

II. Study Setup

A. Methodology

In this study, we asked ChatGPT to generate 21 programs, using a variety of programming languages. The programs generated serve a diversity of purpose, and each program was chosen to highlight risks of a specific vulnerability (eg.

GPT生成的代码片段中，至少76%包含各种安全漏洞。

目录

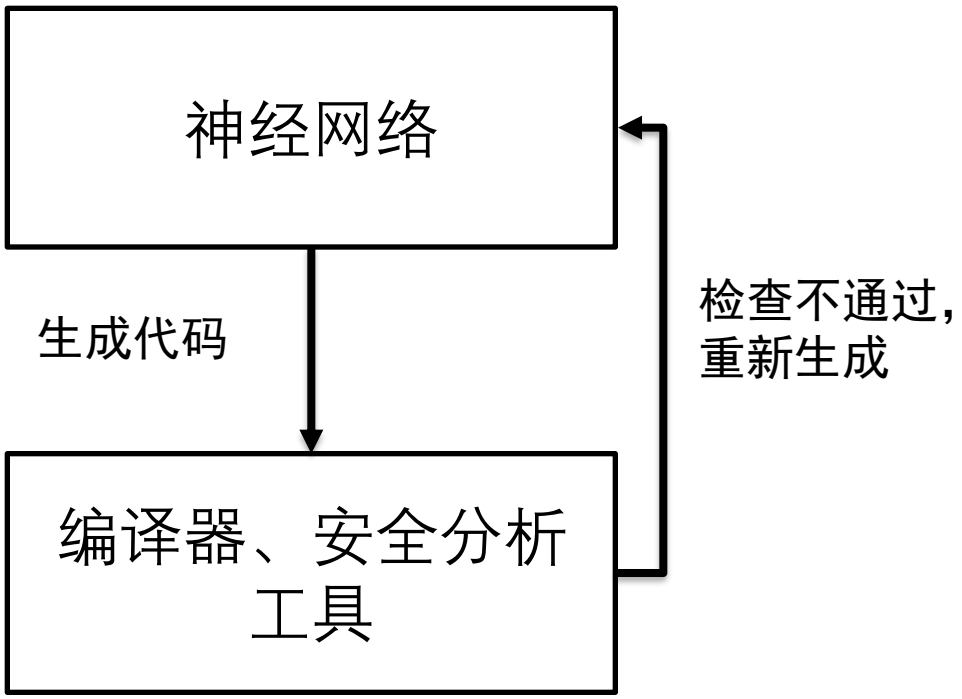
用符号方法改进神经网络

神经网络和符号方法协作

用神经网络改进符号系统

能否让神经网络仅生成安全代码？

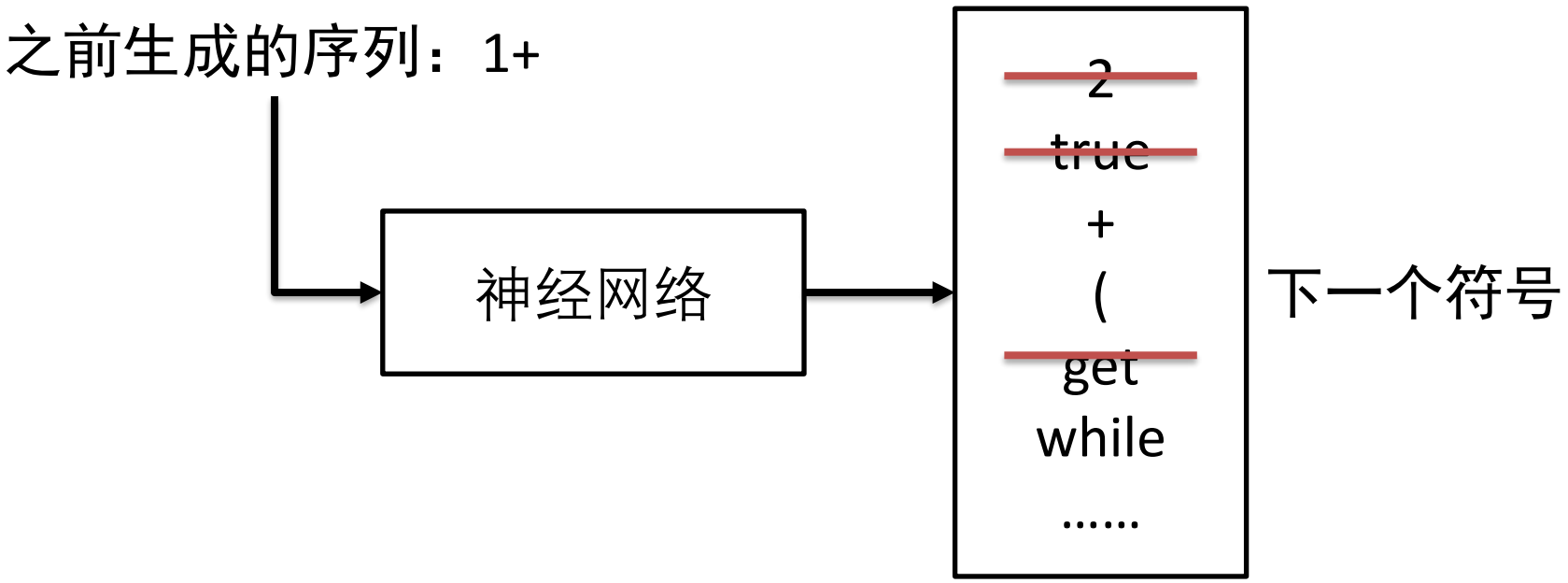
尝试1：生成之后检查



低效，反复生成包含同样错误的代码。

能否让神经网络仅生成安全代码？

尝试2：在生成的时候实时分析，过滤掉不合法的输出



部分程序的分析非常困难：神经网络输出的符号是BPE分词结果，甚至难以词法分析，更不要说语法解析代码结构。

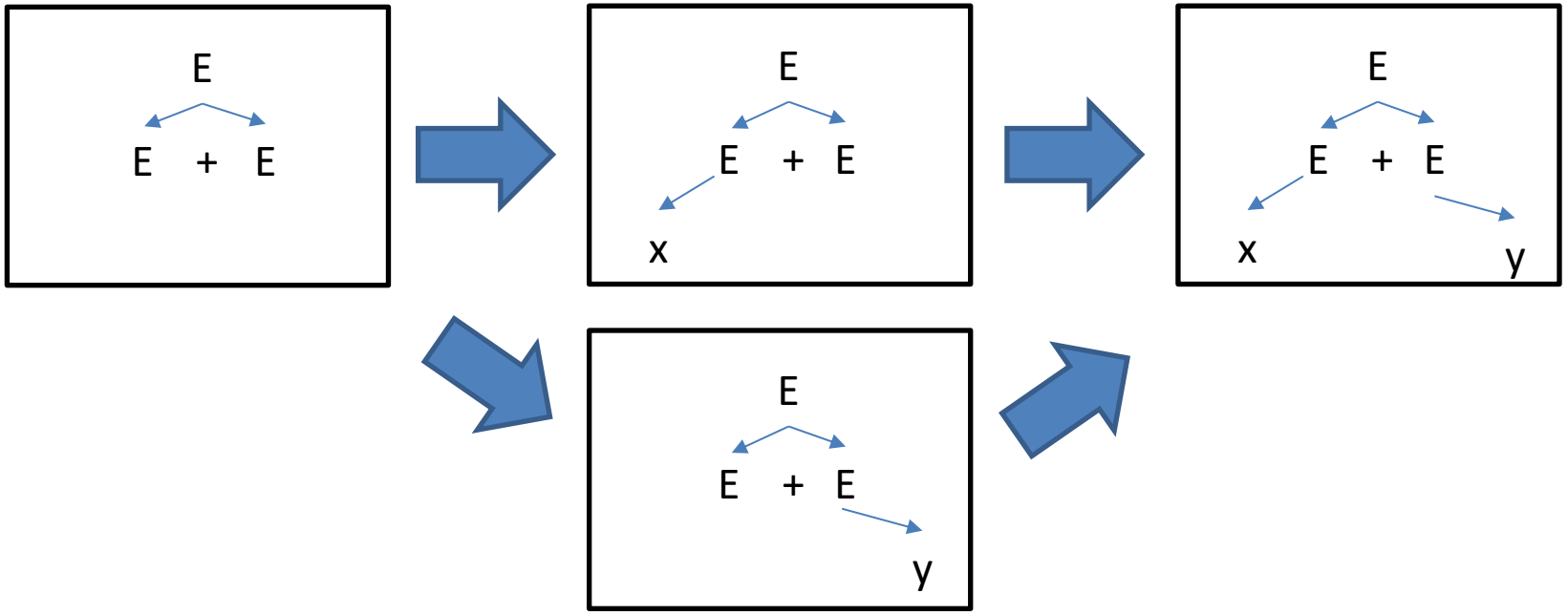
解决方案：语法规则序列表示程序

玲珑框架[TOSEM22]：通过语法规则序列表示程序。

程序	x+y
单词序列	x, +, y
规则序列	r_1, r_2, r_3

$$r_1: E \rightarrow E + E$$
$$r_2: E \rightarrow x$$
$$r_3: E \rightarrow y$$

如何计算程序的概率：问题



同一个程序可能从不同路径到达，是否影响结果？

玲珑框架——概率定理

$$P(prog \mid prompt)$$
$$= \prod_i P(\ rule_i \mid prompt, prog_i, position_i\ )$$

- 程序的概率只和规则选择概率有关，和AST结点展开顺序无关
- 可以根据需要选择合适的顺序生成

如何保证找到的程序满足编程语言的约束？

满足语法约束：直接可检查

如何满足类型和语义约束？

基于抽象解释可以对文法规则预分析

- E→E && E产生类型为Bool的表达式
- E→malloc(E)产生一块待释放的内存

基于预分析的结果快速剪枝

- 假设目前的部分程序1+E
- 可以知道E→E && E不是合法展开式

采用神经网络实现玲珑框架[AAAI20]

用Transformer实现玲珑框架中的概率模型

- 最早的采用Transformer生成代码的工作
- 将Transformer适配到文法规则上形成TreeGen

	Model	StrAcc	Acc+	BLEU
Plain	LPN (Ling et al. 2016)	6.1	–	67.1
	SEQ2TREE (Dong and Lapata 2016)	1.5	–	53.4
	YN17 (Yin and Neubig 2017)	16.2	~18.2	75.8
	ASN (Rabinovich, Stern, and Klein 2017)	18.2	–	77.6
	ReCode (Hayati et al. 2018)	19.6	–	78.4
	TreeGen-A	**25.8**	**25.8**	**79.3**
Structural	ASN+SUPATT (Rabinovich, Stern, and Klein 2017)	22.7	–	79.2
	SZM19 (Sun et al. 2019)	27.3	30.3	79.6
	TreeGen-B	**31.8**	**33.3**	80.8

应用举例：程序缺陷修复[ESEC/FSE21]

TreeGen还被后续研究应用到反编译、代码修复、代码搜索、自动化代码编辑等多个领域，均取得了显著超越SOTA的表现

Table 2: Comparison without Perfect Fault Localization

Project	jGenProg	HDRepair	Nopol	CapGen	SketchFix	FixMiner	SimFix	TBar	DLFix	PraPR	AVATAR	Recoder
Chart	0/7	0/2	1/6	4/4	6/8	5/8	4/8	**9/14**	5/12	4/14	5/12	8/14
Closure	0/0	0/7	0/0	0/0	3/5	5/5	6/8	8/12	6/10	12/62	8/12	**17/31**
Lang	0/0	2/6	3/7	5/5	3/4	2/3	**9/13**	5/14	5/12	3/19	5/11	**9/15**
Math	5/18	4/7	1/21	12/16	7/8	12/14	14/26	**18/36**	12/28	6/40	6/13	15/30
Time	0/2	0/1	0/1	0/0	0/1	1/1	1/1	1/3	1/2	0/7	1/3	**2/2**
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/2	1/1	1/6	**2/2**	**2/2**
Total	5/27	6/23	5/35	21/25	19/26	25/31	34/56	42/81	30/65	26/148	27/53	**53/94**
P(%)	18.5	26.1	14.3	**84.0**	73.1	80.6	60.7	51.9	46.2	17.6	50.9	56.4

In the cells, x/y:x denotes the number of correct patches, and y denotes the number of patches that can pass all the test cases.

将程序的修改空间通过上下文无关文法定义，然后应用TreeGen生成补丁。

神经网络修复四年多来首次超过传统修复的效果

玲珑框架局限性

在玲珑框架主要从外部加上语法、类型、语义等约束。
神经网络并没有建立起能正确推断符号规则的概率模型，
可能导致概率推断出现偏差。

```
bool m(bool a, bool b) {
    return _____
}
```

加法语句很常见，多半要用加法

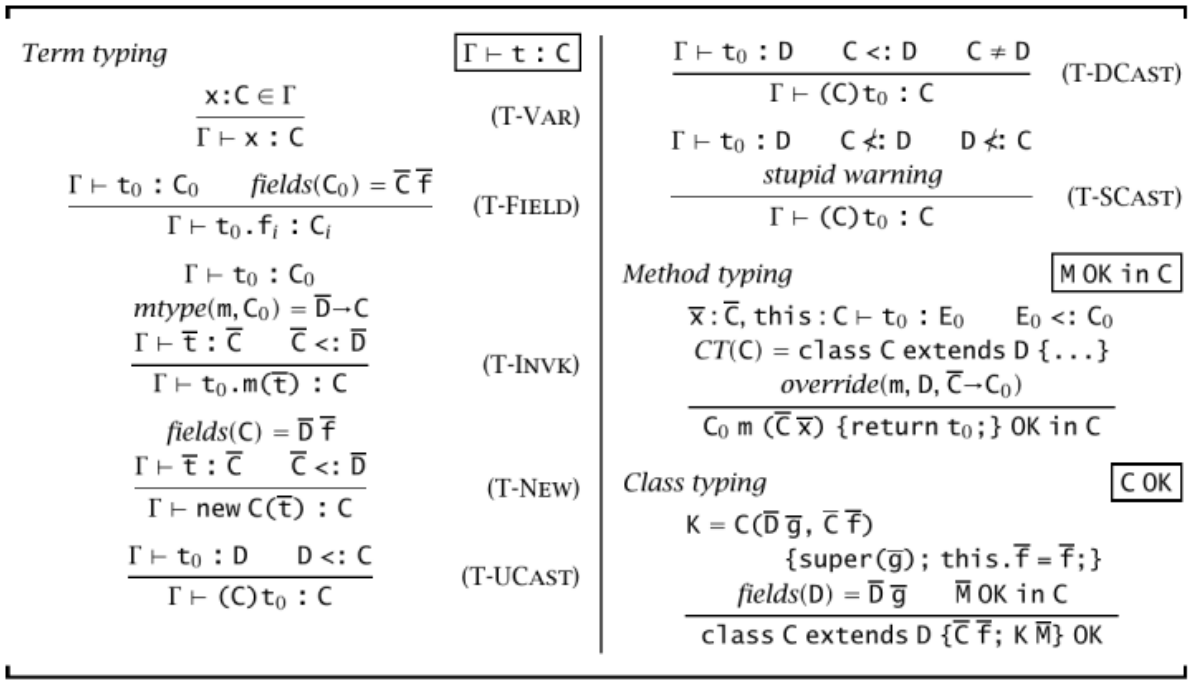
不懂类型的神经网络

参数都是bool形，与或更有可能

懂类型的神经网络

如何引导神经网络学习符号约束？

完整符号系统由多条规则组成，很复杂，难以从数据中直接学会



Term typing $\boxed{\Gamma \vdash t : C}$

$$\frac{x{:}C \in \Gamma}{\Gamma \vdash x : C} \quad \text{(T-Var)}$$

$$\frac{\Gamma \vdash t_0 : C_0 \qquad \mathit{fields}(C_0) = \overline{C}\ \overline{f}}{\Gamma \vdash t_0.f_i : C_i} \quad \text{(T-Field)}$$

$$\frac{\Gamma \vdash t_0 : C_0 \qquad \mathit{mtype}(m, C_0) = \overline{D} \to C \qquad \Gamma \vdash \overline{t} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash t_0.m(\overline{t}) : C} \quad \text{(T-Invk)}$$

$$\frac{\mathit{fields}(C) = \overline{D}\ \overline{f} \qquad \Gamma \vdash \overline{t} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash \mathsf{new}\ C(\overline{t}) : C} \quad \text{(T-New)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad D <: C}{\Gamma \vdash (C)t_0 : C} \quad \text{(T-UCast)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad C <: D \qquad C \neq D}{\Gamma \vdash (C)t_0 : C} \quad \text{(T-DCast)}$$

$$\frac{\Gamma \vdash t_0 : D \qquad C \not<: D \qquad D \not<: C \qquad \textit{stupid warning}}{\Gamma \vdash (C)t_0 : C} \quad \text{(T-SCast)}$$

Method typing $\boxed{M\ \mathsf{OK\ in}\ C}$

$$\frac{\overline{x} : \overline{C}, \mathsf{this} : C \vdash t_0 : E_0 \qquad E_0 <: C_0 \qquad \mathit{CT}(C) = \mathsf{class}\ C\ \mathsf{extends}\ D\ \{\ldots\} \qquad \mathit{override}(m, D, \overline{C} \to C_0)}{C_0\ m\ (\overline{C}\ \overline{x})\ \{\mathsf{return}\ t_0;\}\ \mathsf{OK\ in}\ C}$$

Class typing $\boxed{C\ \mathsf{OK}}$

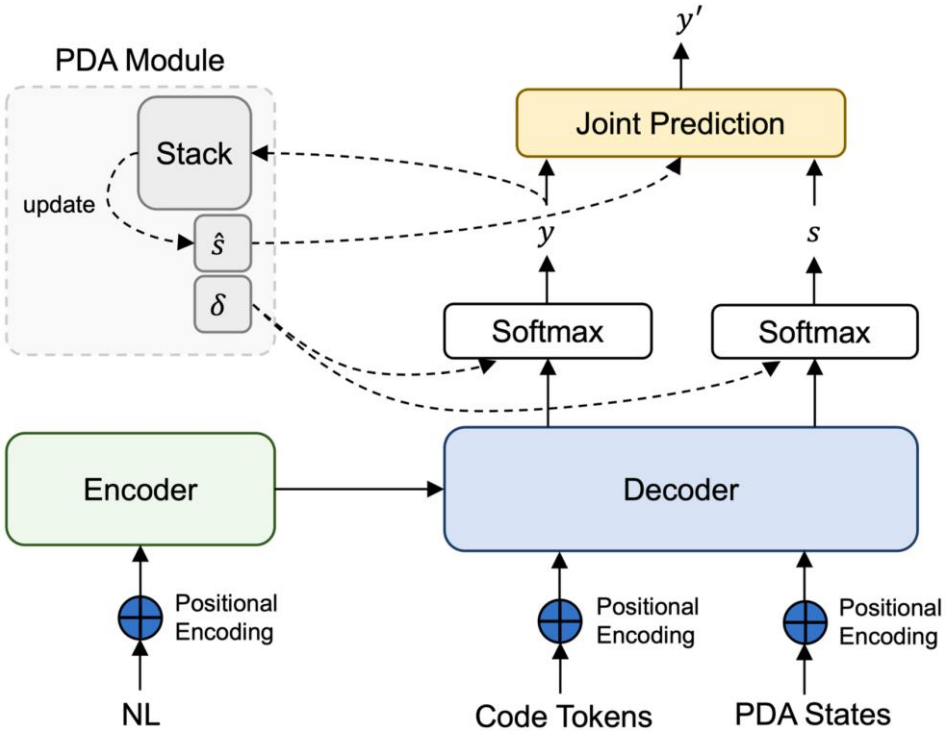
$$\frac{K = C(\overline{D}\ \overline{g}, \overline{C}\ \overline{f})\ \{\mathsf{super}(\overline{g}); \mathsf{this}.\overline{f} = \overline{f};\} \qquad \mathit{fields}(D) = \overline{D}\ \overline{g} \qquad \overline{M}\ \mathsf{OK\ in}\ C}{\mathsf{class}\ C\ \mathsf{extends}\ D\ \{\overline{C}\ \overline{f};\ K\ \overline{M}\}\ \mathsf{OK}}$$

Figure 19-4: Featherweight Java (typing)

感知语法规则[ISSTA23-北大李戈团队]

假设单条规则的检查是容易学会的

1. 实现规则检查算法，在生成的同时维护算法的内部状态
   - 下推自动机的栈
2. 在生成的时候将内部状态呈现给神经网络



感知类型规则[ICSE23]

类型规则的检查更为复杂：

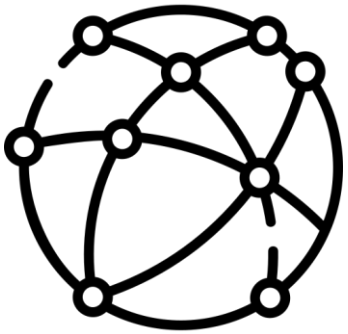
1. 状态比较庞大：设计数据结构仅表征当前检查所需的信息
2. 类型检查不能局部进行：同时输出非终结符的期望类型

类型规则

$$\frac{\Gamma \vdash v : D \quad \Gamma \vdash t : C \quad C <: D}{\Gamma \vdash v = t : \text{Void}}$$

类型关系

(1)AST和类型
(2)变量和类型
(3)类型之间的子关系



T-Graph

T-Grammar

应用举例[ICSE23]

应用于缺陷修复，形成Tare方法（参数量3510万），成功修复的数量提升26.5%

Project	Bugs	CapGen	SimFix	TBar	DLFix	Hanabi	Recoder	Recoder-F	Recoder-T	Tare
Chart	26	4/4	4/8	9/14	5/12	3/5	8/14	9/15	8/16	**11/16**
Closure	133	0/0	6/8	8/12	6/10	-/-	13/33	14/36	15/31	**15/29**
Lang	64	5/5	9/13	5/14	5/12	4/4	9/15	9/15	11/23	**13/22**
Math	106	12/16	14/26	18/36	12/28	19/22	15/30	16/31	16/40	**19/42**
Time	26	0/0	1/1	1/3	1/2	2/2	**2/2**	**2/2**	**2/4**	**2/4**
Mockito	38	0/0	0/0	1/2	1/1	-/-	**2/2**	**2/2**	**2/2**	**2/2**
Total	393	21/25	34/56	42/81	30/65	28/33	49/96	52/101	54/116	**62/115**

在最新国际修复比赛上获得Java功能性缺陷赛道冠军
超过其他团队用几亿、几十亿参数预训练模型构建的工具

大型预训练模型[ICSE24]

现代大型代码预训练模型都采用多种编程语言训练。
不同编程语言文法规则、类型规则各不相同。
以上方法能否应用于预训练模型？

<identifier> ::= <initial> | <initial> <more>
<initial> ::= <letter> | _ | $
<more> ::= <final> | <more > <final>
<final> ::= <initial> | <digit>
<letter> ::= a | b | c | … z | A | B| … | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Java文法

Python文法

```
stringliteral   ::=  [stringprefix](shortstring | longstring)
stringprefix    ::=  "r" | "u" | "R" | "U" | "f" | "F"
                     | "fr" | "Fr" | "fR" | "FR" | "rf" | "rF" | "Rf" | "RF"
shortstring     ::=  "'" shortstringitem* "'" | '"' shortstringitem* '"'
longstring      ::=  "'''" longstringitem* "'''" | '"""' longstringitem* '"""'
shortstringitem ::=  shortstringchar | stringescapeseq
longstringitem  ::=  longstringchar | stringescapeseq
shortstringchar ::=  <any source character except "\" or newline or the quote>
longstringchar  ::=  <any source character except "\">
stringescapeseq ::=  "\" <any source character>
```

大型预训练模型

可简单将文法合并为更大的文法
神经网络能学会不同文法规则之间的关联
其他预训练技术（如BPE分词）也能做到和文法兼容

```
Root -> JavaRoot
      |  PythonRoot
      |  ...
JavaRoot -> Imports Classes
...
```

学习“声明–使用”关系

已有预训练任务随机排列文件

神经网络可能先看到方法调用再看到方法定义

上下文依赖解析：

- 从文件中挖掘“声明–使用”关系
- 对文件排序，保证声明出现在使用之前

大型预训练模型

效果全面超越原同规模最好的CodeT5-base模型

同参数量约三倍的CodeT5-Large效果相当

	Natural Language-Based Code Generation							
Models	**Concode**			**Conala**		**Django**		**MBPP**
Metric	**BLEU**	**EM**	**CodeBLEU**	**BLEU**	**EM**	**BLEU**	**EM**	**pass@80**
GPT-C(110M)	30.85	19.85	33.10	30.32	4.80	72.56	68.91	10.40
CodeGPT-adapted(110M)	35.94	20.15	37.27	31.04	4.60	71.24	72.13	12.60
CoTexT(220M)	19.19	19.72	38.13	31.45	6.20	75.91	78.43	14.00
PLBART(220M)	36.69	18.75	38.52	32.44	5.10	72.81	79.12	12.00
CodeT5-small(60M)	38.13	21.55	41.39	31.23	6.00	76.91	81.77	19.20
CodeT5-base(220M)	40.73	22.30	43.2	38.91	8.40	81.40	84.04	24.00
CodeT5-large(770M)	**42.66**	22.65	45.08	39.96	7.40	82.11	83.16	**32.40**
Unixcoder(110M)	38.73	22.65	40.86	36.09	10.20	78.42	75.35	22.40
GrammarT5-small(60M)	38.68	21.25	41.62	39.18	8.00	81.20	82.77	26.00
GrammarT5-base(220M)	42.30	**24.75**	**45.38**	**41.42**	**10.40**	**82.20**	**84.27**	32.00

其他应用：DeepSeek

DeepSeek
Coder-V1-33B

- 我国首个达到全球顶尖水平的开源代码生成模型
- 其7B版本就超过了ChatGPT3.5
- 由团队博士生朱琪豪实习期间领导开发
- 应用了上述上下文依赖解析技术

未来展望

历史上研究较多的方向，应用效果获得验证，但距离大面积推广仍需进一步发展

- 在更大的模型上检验效果
- 更通用的集成符号约束的方法
- 适配代码片段等不易符号分析的场景

目录

用符号方法改进神经网络

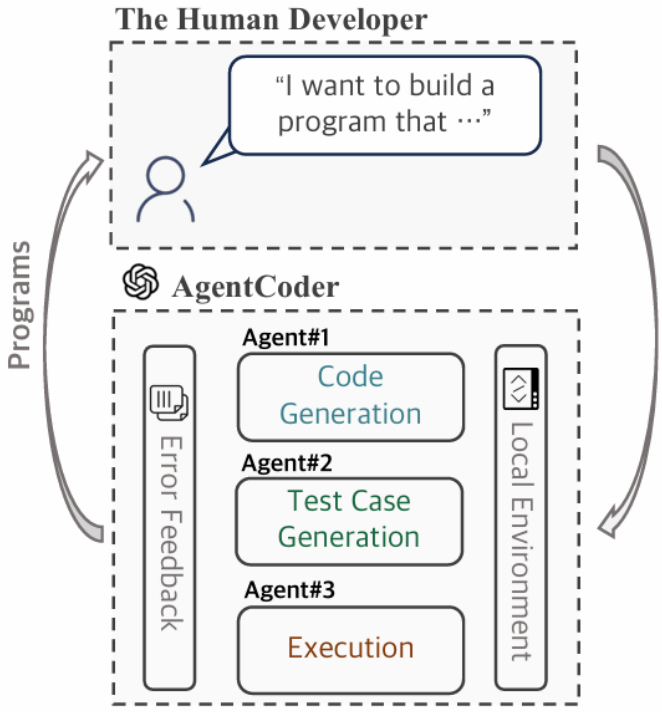
神经网络和符号方法协作

用神经网络改进符号系统

多智能体系统[AgentCoder-KCL张洁等]

经过指令微调的模型有很好的上下文学习能力。

符号系统和大模型扮演不同的智能体角色，通过文本提示进行协作。



- Agent1：编写代码
- Agent2：编写测试
- Agent3：符号系统，负责执行测试并反馈

- 目前HumanEval排名第二的方法

未来展望

目前研究最为集中的方向，进展迅速

- 但过于拥挤，重复工作很多

在大量软件工程应用中潜力还没有充分发挥

- 程序验证
- 程序分析
- 程序优化
- 缺陷定位和修复
- 程序生成

可能显著降低程序验证等传统成本过高的方法，改变软件开发实践

目录

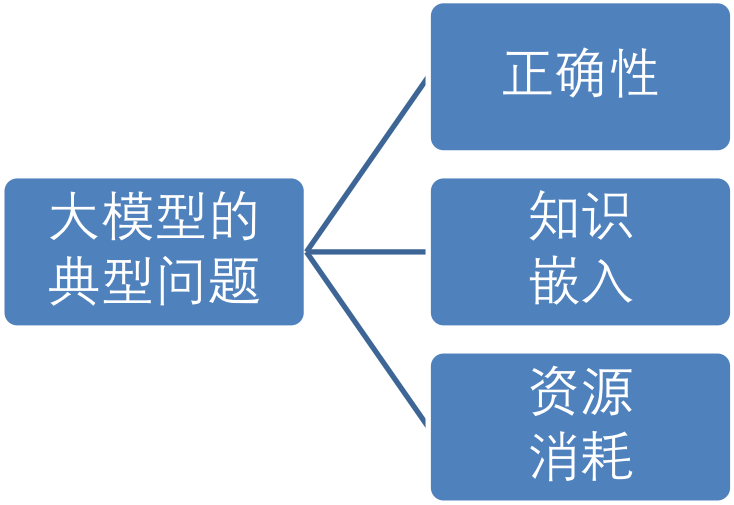
用符号方法改进神经网络

神经网络和符号方法协作

用神经网络改进符号系统

为什么还要改进符号系统？

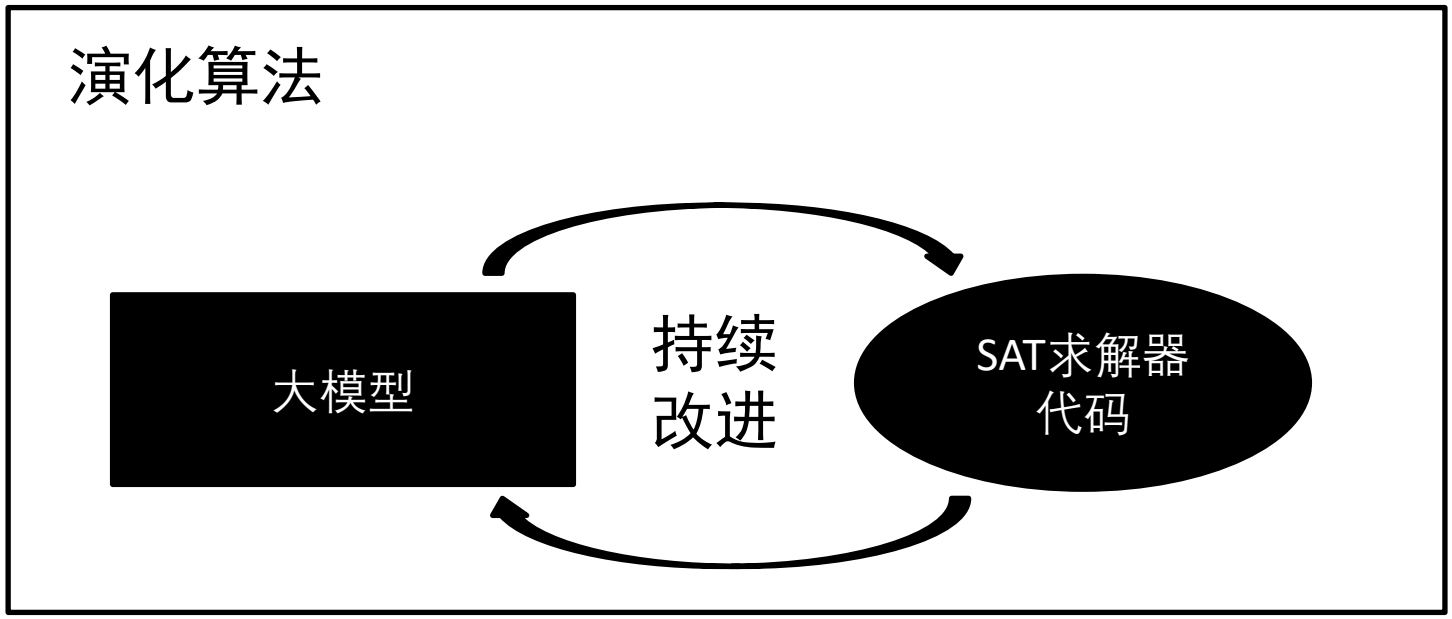
- 代码天然是符号系统产物
- 符号系统处理代码有天然优势
  - 当x=10，分析x=x+1的执行结果
  - 符号系统瞬间得到答案
  - 大模型消耗大量计算资源，还不保证正确
- 大模型的典型问题对符号系统也不是问题



用神经网络改进符号系统[AutoSAT-蔡少伟等]

神经网络拥有很强的分析和编码能力，可直接改进符号系统

样例：基于大模型的SAT求解器设计



从通用SAT实现出发，不断让大模型改进。

结合演化算法和特定领域的测试集不断演化。

多次迭代后，在特定领域达到目前最佳的效果。

未来展望——符号主义的问题

JOURNAL ARTICLE

Why Expert Systems Fail

Michael Z. Bell

The Journal of the Operational Research Society, Vol. 36, No. 7 (Jul., 1985), pp. 613-619 (7 pages)

https://doi.org/10.2307/2582480 · https://www.jstor.org/stable/2582480

- 专家没空
- 专家无法表述规则
- 专家不想表述规则
- 没有专家
- 缺乏常识

劳动力不够

- 知识表示语言不够强
- 难以测试
- 用户不信任

所有智能系统的公共问题

未来展望——用大模型改进符号系统

- 核心问题是劳动力不够
  - 并且是高阶智力劳动力（专家）
- 大模型就是这样的劳动力
  - 学会了各种人类知识
  - 可以按需工作
- 有望形成和大模型能力接近，但保留符号方法优点的系统

小结

- 代码是符号主义的产物
- 联结主义的神经网络天然不擅长学习和推断符号规则
- 结合符号方法和神经网络可有效提升代码任务的效果
- 结合方法
  - 用符号方法改进神经网络
    - 已有研究较多，效果获得验证，但仍需继续推进
  - 神经网络和符号方法协作
    - 目前研究焦点，有较广发展空间，但过于拥挤
  - 用神经网络改进符号方法
    - 目前研究较少，但有较好发展潜力

敬请各位专家批评指正!