arXiv:2301.04589v1 [cs.CL] 10 Jan 2023

记忆增强型大语言模型具有计算通用性

Dale Schuurmans Google
Brain 与阿尔伯塔大学

摘要

我们证明，基于Transformer的大语言模型在配备外部记忆后具有计算通用性。任何以长度有界的字符串为条件的确定性语言模型都等价于有限自动机，因此其计算能力有限。然而，为这类模型增添读写存储器后，便有可能处理任意大的输入，并有望模拟任何算法。我们证明，现有的大语言模型 Flan-U-PaLM 540B 与关联式读写存储器相结合后，可以精确模拟通用图灵机的执行，$U_{15,\ 2}$。这一发现的一个关键之处在于，无需修改语言模型的权重。相反，这种构造仅依赖于设计一种存储指令计算机，之后可通过一组特定的提示词对其进行编程。

1 引言

自 GPT-2、GPT-3 和 InstructGPT 早期取得成功以来，人们对大语言模型的兴趣显著增长 [Radford 等人，2019，Brown 等人，2020，Ouyang 等人， 2022],而近来 ChatGPT 的流行更是推动了这一趋势 [Schulman 等人， 2022]。除了简单的问答（输入字符串提出问题，可能会引出包含合理答案的输出字符串）之外，一项重要发现是上下文学习的出现：在问题前添加一组相关的（问题、答案）对，可以显著提高问答准确率 [Radford 等人， 2019]。在示例对之前添加自然语言指令，似乎还能进一步增强语言模型的能力 [Brown 等人， 2020]。最近，人们发现，在需要经过多个推理步骤才能得出最终答案的场景中，例如解答数学应用题，思维链提示能够提升问答能力 [Wei 等人，2022 b]。

尽管取得了这些成果，当前基于 Transformer 的大型语言模型仍存在根本性局限，因为它们只能以长度有限的输入字符串为条件，例如 4096 个词元。这使得这类模型在形式上等价于有限自动机，因此能够表达的计算也受到限制。然而，近期研究已开始探索一种技术：通过处理模型输出，再

将其作为后续输入传回模型，从而串联对语言模型的多次调用。一个例子是由简至繁提示：对于复杂的推理问题，先提示模型生成较简单的子问题，再将每个子问题及其答案传回模型，以帮助解答后续子问题，直至得到最终答案 [Zhou 等人， 2022]。另一个例子是关于语言模型级联的研究，该研究探讨了处理模型输出并将其作为输入传递给后续语言模型调用的各种策略[Dohan 等人， 2022]。这类研究引出了一个问题：通过外部反馈循环增强语言模型，究竟只是带来实用帮助，还是从根本上拓展了其能够执行的计算范围？为探究这一问题，我们考虑用外部读写存储器增强语言模型，并探讨这是否赋予它模拟任意算法处理任意输入的能力。

本文通过证明特定大语言模型 Flan-U-PaLM 540B [Chung 等人， 2022], 在配备关联式读写存储器后具有计算普适性，给出了肯定的回答。这一结果的关键在于构建了一种简单形式的存储指令计算机 [冯·诺伊曼， 1945] ，将语言模型与关联记忆连接起来，然后遵循一个简单的指令周期：从内存中取出要传递给语言模型的下一条输入提示，解析语言模型的输出以提取变量赋值，并将其存入关联记忆中，随后取出下一条指令（即将提供给语言模型的下一条输入提示）。语言模型与内存之间的每个解析步骤都通过简单的正则表达式匹配（即有限自动机）完成。

创建存储指令计算机后，设计一个特定的“提示程序”，驱动系统模拟通用图灵机$U_{15,2}$ [Neary，2008，Neary 和 Woods， 2009]。证明模拟的保真度，归结为检查有限组提示—结果行为，并验证语言模型针对可能遇到的有限组输入提示字符串中的每一个，都能生成正确的输出。也就是说，尽管 Flan-U-PaLM 540B 模型的整体输入输出行为尚未得到充分理解，但仍可分离并控制其输入输出映射中一个足够可靠的子集，以模拟通用计算机。重要的是，这一结果不涉及对语言模型进行任何额外“训练”（即不修改其预训练权重），而仅依靠向模型提供特定提示字符串，并解析其输出，以确定要保存到内存中的值。

2 存储指令计算机

如前所述，有许多方法可以协调语言模型的输出与后续输入提示之间的反馈 [Zhou 等，2022，Dohan 等， 2022]。在构建简单的反馈循环时，我们希望尽可能减少外部处理，并尽可能多地由语言模型执行计算，同时仍支持计算通用性。为此，我们考虑一种简单的存储指令计算机形式[冯·诺伊曼， 1945], 其中语言模型充当中央处理器（CPU），而随机存取存储器（RAM）由外部关联记忆提供

。这种架构支持简单的交互循环，从而实现通用计算和便捷的可编程性。在这种架构中，外部关联记忆是一个简单的“字典” MEMORY，它将唯一的键映射到值，或者等价地，将变量名映射到值，或将地址位置映射到值。与物理计算机中的 RAM 不同，为了便于与语言模型交互，变量名将是字符串（即由有限字母表中的符号组成的有限长度序列），而值将是字符串或整数。

为确保计算普适性并非仅仅源于外部处理能力，语言模型与内存之间的所有交互都将限制为有限状态计算，例如简单的正则表达式解析器。

以下所有代码均使用 Python 3 和标准正则表达式库re编写。请注意，用于预处理和后处理的正则表达式较为基础，可以通过多种方式轻松改进；本文给出的版本仅足以说明主要观点。

2.1 语言模型输出的后处理

语言模型生成的输出字符串将由一个简单的正则表达式解析，该表达式会检测符合 variable_name = "value"形式的赋值，然后将其作为 MEMORY[variable_name] = "value" 应用于关联记忆。变量赋值函数 assignments 如下所示。

```
def assignments(string):global MEMORY
regex = '(?s)(?:((?:\w|\-)+)\s*=\s*(?:\"((?:.*\n)|(?:[^\"]*))\"))(.*)'
matches = re.findall(regex, string)suffix = ''while len(matches) > 0:
label, value, suffix = matches[0]MEMORY[label] = value
matches = re.findall(regex, suffix)return suffix
```

此外，在将值字符串赋给内存之前，允许进行拼接；也就是说，我们引入了一个正则表达式解析器，用于检测任意值字符串中出现的模式 %[variable_name]，并在赋值之前，将任何此类出现替换为内存位置variable_name处的字符串，即 MEMORY[variable_name]。替换函数 substitute 如下所示。

```
def substitute(string, char):global MEMORY, BLANK
regex = f"(?s)(.*?)(?:{char}\[(\w+)\])(.*)"
matches = re.findall(regex, string)string = suffix = ''
while len(matches) > 0:prefix, label, suffix = matches[0]
if label not in MEMORY:
MEMORY[label] = BLANK # new label has BLANK value by default
string += prefix + str(MEMORY[label])
matches = re.findall(regex, suffix)string += suffixreturn string
```

最后，为了方便起见，我们还允许存储整数值，并对其进行递增或递减。整数变量更新通过解析输出字符串，查找模式 `variable_name += increment` or `variable_name -= decrement` 的出现位置，然后在内存中对 `variable_name` 应用更新来实现。重要的是，整数加法也是一种有限状态操作（例如，参见 [Sipser，2013，问题 1.32]），而 Python 标准库提供了能够处理任意大整数的 `bignum` 类型。更新函数 `updates` 如下所示。

```
def updates(string):global MEMORYregex = '(\w+)\s*((?:\+|\-)=)\s*(\d+)'
matches = re.findall(regex, string)if matches != None:
for match in matches:label, operator, valuestring = match
sign = 1 if operator == "+=" else -1value = int(valuestring) * sign
if label in MEMORY and isinstance(MEMORY[label], int):
MEMORY[label] += valueelse:
MEMORY[label] = value # new or non-int starts from 0 by default
```

2.2 语言模型输入预处理

语言模型的每个输入提示都将从一个特殊的内存位置`op` 中读取，并在每个计算周期中作为提示传递给语言模型；也就是说，`MEMORY['op']` 将充当“指令寄存器”。随后，只需在某个计算周期中为 `MEMORY['op']` 分配不同的提示字符串，即可实现指令分支。

为了访问存储的内存值，我们还允许将从 op中读取的输入提示字符串拼接进来。具体而言，输入提示的正则表达式解析器首先检测形如 @[variable_name] 的模式，并在将提示字符串传递给语言模型之前，通过拼接从 MEMORY[variable_name] 中读取的字符串来替换这些模式。对于输入预处理而言，允许重复替换嵌套的@ 出现形式也很方便，因此我们在下文中加入重复替换函数 substitute_nested 。请注意，从技术上讲，允许任意深度的嵌套替换可以模拟上下文无关文法 [Sipser, 2013], ，这违反了有限状态计算的约束；不过，下面我们只使用有界深度嵌套（深度上限为 2），以确保前处理和后处理步骤都仍可通过有限状态计算实现。

```
def substitute_nested(string, char):
  regex = f"(?s)(.*?)(?:{char}\[(\w+)\])(.*)"
  while re.match(regex, string) != None:
    string = substitute(string, char)
  return string
```

2.3 计算周期

最后，存储指令计算机可以运行一个计算周期：从 MEMORY['op']中取出下一条提示字符串；处理该提示字符串（可能会拼接内存中的其他字符串）；将提示字符串传递给语言模型；处理输出字符串（可能会拼接内存中的其他字符串），检测所有赋值以及递增/递减更新，并将其应用于内存；然后重复这一过程。计算将持续进行，直到 MEMORY['op'] 中的下一条指令是特殊指令字符串 'halt'。具体而言，主循环按如下方式构造。

```
def main():
  global MEMORY
  while True:
    op = MEMORY['op']
    if op == 'halt':
      return None
    prompt = substitute_nested(op, '@')
    result = call_llm_server(prompt)
    result = substitute(result, '%')
    suffix = assignments(result)
    updates(suffix)
```

这个主指令循环展示了语言模型如何扮演 CPU 的角色：它根据提示字符串获取下一条指令及其操作数，并据此执行操作，生成结果字符串，再用该结果字符串更新内存。

3 通用图灵机

通用计算机这一概念——一种能够模拟任何其他计算机器在任意输入上的执行过程的计算机器——由艾伦·图灵提出，用以解决判定问题 [Turing, 1937]。根据丘奇-图灵论题，所有计算机制都可以用图灵机来表示。图灵机大致由有限状态控制器和容量无限大的“纸带”内存组成，并配有一个“读写头”，每个计算周期中都可以访问纸带上的一个位置，并向左或向右移动一个位置[Sipser，2013，Chapter 3]。

形式上，图灵机由一个元组 $\mathcal{M} = (Q, \Sigma, b, q_0, T, f)$ 构成，其中 Q 是有限状态集，Σ 是有限的磁带符号集，$b \in \Sigma$ 是空白符号，$q_0 \in Q$ 是初始状态，$T \subseteq Q \times \Sigma$ 是停机（状态，符号）对的集合，而 $f\colon Q \times \Sigma \to \Sigma \times \{-1, +1\} \times Q$ 是有限的转移规则集，用于指定机器在每个计算周期中的操作。我们假设磁带在两个方向上都无限延伸，因此存储位置可用整数 $i \in \mathbb{Z}$ 编号。令 $i_0 \in \mathbb{Z}$ 表示磁带头的初始位置。

图灵机的执行过程可定义如下。磁带存储器初始化时包含有限个非空白符号，其余位置均为空白；$\mathcal{M}$ 处于状态 q_0，磁带头位于位置 i_0。每个计算周期开始时，磁带头位于某个位置 $i \in \mathbb{Z}$，机器处于某个状态 $q \in Q$，且磁带头下方有某个符号$\sigma \in \Sigma$。这些条件共同决定更新 $f(q, \sigma) \mapsto (\sigma', m, q')$，表示在当前内存位置 i写入符号 σ'，将机器状态q更新为 q'，并将磁带头向左移动一步至位置 $i' = i - 1$（若 $m = -1$），否则向右移动一步至位置 $i' = i + 1$（若 $m = +1$）。计算周期不断重复，直到机器遇到构型 $(q, \sigma) \in T$。也可能出现非停机计算。

[Shannon 1956] 开启了一项研究，旨在确定按所用状态数和磁带符号数衡量的最小通用图灵机。已知通用图灵机在状态数和符号数方面的上界和下界之间仍有差距[Neary，2008，Neary 和 Woods，2009]，但人们已经发现了规模逐渐更小的通用图灵机。本文将考察其中一台机器：$U_{15,\ 2}$，它仅使用 15 个状态和 2 种磁带符号 [Neary 和 Woods，2009]。就已知的最小通用图灵机而言，这台图灵机是帕累托最优的 [Neary，2008]。形式上，图灵机$U_{15,\ 2}$可定义为一个元组（Q，Σ，b，q_0，T，f），其中 $Q = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}$，$\Sigma = \{0, 1\}$，$b = 0$，$q_0 = A$，$T = \{$（$J$，1）$\}$，而转移函数 f 定义见表1。初始读写头位置 i_0 取决于如何为给定的问题实例初始化内存。

4 使用提示程序模拟$U_{15,\ 2}$

本节将展示：如果语言模型能够正确执行一组有限的条件赋值和求值，那么第2节中定义的存储指令计算机就可以编程来模拟通用图灵机 $U_{15,\ 2}$。也就是说，我们首先提出一个特定的提示程序；如果该程序能够正确执行，就能精确地

	A	B	C	D	E	F	G	H
0	$0,+,B$	$1,+,C$	$0,-,G$	$0,-,F$	$1,+,A$	$1,-,D$	$0,+,H$	$1,-,I$
1	$1,+,A$	$1,+,A$	$0,-,E$	$1,-,E$	$1,-,D$	$1,-,D$	$1,-,G$	$1,-,G$

	I	J	K	L	M	N	O
0	$0,+,A$	$1,-,K$	$0,+,L$	$0,+,M$	$0,-,B$	$0,-,C$	$0,+,N$
1	$1,-,J$	halt	**1**$,+,N$	**1**$,+,L$	**1**$,+,L$	$0,+,O$	$1,+,N$

表1：通用图灵机 $U_{15,\ 2}$ 的转移表。行按读符号 σ 编号，列按状态 q 编号，每个表项 (σ', m, q') 指定写符号 σ'、磁带头移动方向 $m \in \{-1, +1\}$ 和下一状态 q'。

模拟 $U_{15,\ 2}$。下一节将验证一个特定的大语言模型 Flan-U-PaLM540B 确实能够正确执行每条程序指令。

提示程序由一组有限的预先设计的字符串组成，这些字符串存储在内存中，并在主计算周期中通过调用第2节概述的 `main()` 向语言模型提供输入提示。为了模拟 $U_{15,\ 2}$ 的行为，我们按如下方式设计提示程序。首先，设计一个“启动”提示，用来“指示”语言模型如何处理变量赋值、赋值后的变量求值，以及 if-then 条件语句。

```
启动 = """ 结果 = " 运算符="%[B]" %[i]="0" i+=1 " 如果 0==1 则 结果 = " 运算符="%[A]" %[i]="1" i+=1 " $结果 " 运算符="%[B]" %[i]="0"i+=1 "
```

```
结果 = " op="%[B]" %[i]="0" i+=1 " 如果 1==1 那么结果 = " op="%[A]" %[i]="1" i+=1 " $result " op="%[A]" %[i]="1"i+=1 " 结果 = " op="%[C]" %[i]="1" i+=1 " 如果 0==1 那么结果 = " op="%[A]" %[i]="1" i+=1 " $result " op="%[C]" %[i]="1"i+=1 " 结果 = " op="%[C]" %[i]="1" i+=1 " 如果 1==1 那么结果 = " op="%[A]" %[i]="1" i+=1 " $result " op="%[A]" %[i]="1"i+=1 " 结果 = " op="%[G]" %[i]="0" i-=1 " 如果 0==1 那么结果 = " op="%[E]" %[i]="0" i-=1 " $result " op="%[G]" %[i]="0"i-=1 "
```

```
结果 = " 操作="%[G]" %[i]="0" i-=1 " 若 1==1 则 结果 = " 操作="%[E]" %[i]="0"i-=1 "
```

```
$结果
" op="%[E]" %[i]="0" i-=1 "

result = " op="%[F]" %[i]="0" i-=1 " 如果 0==1 则
result = " op="%[E]" %[i]="1"i-=1 " $result " op="%[
F]" %[i]="0"i-=1 "

结果 = " 操作="%[F]" %[i]="0" i-=1 " 如果 1==1 则 结
果 = " 操作="%[E]" %[i]="1"i-=1 " $结果 " 操作="%[E]"
%[i]="1"i-=1 "

result = " 操作="%[K]" %[i]="1" i-=1 "
如果 0==1 那么 result = " 操作="停止"
" $result " 操作="%[K]" %[i]="1"i-=1 "

result = " op="%[K]" %[i]="1" i-=1 " if
1==1 then result = " op="halt"" $
result " op="halt" "

结果 = " 操作="%[L]" %[i]="0" i+=1 " 如果 0==1 那么
结果 = " 操作="%[N]" %[i]="1"i+=1 " $结果 " 操作="%[
L]" %[i]="0"i+=1 "

结果 = " 操作="%[L]" %[i]="0" i+=1 " 如果 1==1 那么
结果 = " 操作="%[N]" %[i]="1"i+=1 " $结果 " 操作="%[
N]" %[i]="1"i+=1 "

结果 = " 操作="%[C]" %[i]="0" i-=1 " 如果 1==1 那么
结果 = " 操作="%[O]" %[i]="0"i+=1 " $结果 " 操作="%[
O]" %[i]="0"i+=1 "

"""
```

接下来，定义一系列“指令”提示词。这些字符串旨在表达表1中指定的相应图灵机状态的逻辑，其中 $U_{15,\ 2}$ 。

```
A = """@[启动]结果 = " 运算="%[B]" %[i]="0" i+=1 " 如果
@[@[i]]==1 则 结果 = " 运算="%[A]" %[i]="1" i+=1 " $结果 "
"" B = """@[启动]结果 = " 运算="%[C]" %[i]="1" i+=1 " 如果
@[@[i]]==1 则 结果 = " 运算="%[A]" %[i]="1" i+=1 " $结果 "
""
```

C = """@[启动]结果 = " 运算="%[G]" %[i]="0" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[E]" %[i]="0" i-=1 " $结果 "" " D = """@[启动]结果 = " 运算="%[F]" %[i]="0" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[E]" %[i]="1" i-=1 " $结果 "" " E = """@[启动]结果 = " 运算="%[A]" %[i]="1" i+=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[D]" %[i]="1" i-=1 " $结果 "" F = """@[启动]结果 = " 运算="%[D]" %[i]="1" i-=1 " $结果 """ G = """@[启动]结果 = " 运算="%[H]" %[i]="0" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[G]" %[i]="1" i-=1 " $结果 "" H = """@[启动]结果 = " 运算="%[I]" %[i]="1" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[G]" %[i]="1" i-=1 " $结果 "" I = """@[启动]结果 = " 运算="%[A]" %[i]="0" i+=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[J]" %[i]="1" i-=1 " $结果 "" " J = """@[启动]结果 = " 运算="%[K]" %[i]="1" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="停止" " $结果 """ K = """@[启动]结果 = " 运算="%[L]" %[i]="0" i+=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[N]" %[i]="1" i+=1 " $结果 """ L = """@[启动]结果 = " 运算="%[M]" %[i]="0" i+=1 " 如果 @[@[i]]==1 则 结果 = " 运算 ="%[L]" %[i]="1" i+=1 " $ 结果 """ M = """@[启动]结果 = " 运算="%[B]" %[i]="0" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[L]" %[i]="1" i+=1 " $结果 """ N = """ @[启动]结果 = " 运算="%[C]" %[i]="0" i-=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[O]" %[i]="0" i+=1 " $结果 """ O = """@[启动]结果 = " 运算="%[N]" %[i]="0" i+=1 " 如果 @[@[i]]==1 则 结果 = " 运算="%[N]" %[i]="1" i+=1 " $结果 """

理解这个提示程序的预期工作方式会有所帮助。首先，请注意，内存位置 'i' 旨在跟踪图灵机读写头的当前位置，因此，任何更新 i-=1 都对应于将读写头向左移动一步，而 i+=1则对应于将读写头向右移动一步。接下来，考虑对某个结果字符串进行后处理，例如 " op=%[N]" %[i]="1" i+=1 "。在这个字符串中，表达式 %[i]="1" 旨在将目标符号 '1' 写入由 MEMORY['i']索引的内存位置。也就是说，在后处理中，如第2节所述，任何形如 %[x]的子字符串都会先被 MEMORY['x'] 中的字符串替换，然后才执行任何赋值。因此，给定 %[i]="1" ，子字符串 %[i] 会先被 MEMORY['i']中的值替换，例如 ℓ，随后该值便作为内存中的标签，并被赋值为 '1'。例如，假设 MEMORY['i'] = 42 和 MEMORY['42'] = '0'，那么后处理并赋值后，我们将得到 MEMORY['42'] = '1'。控制分支，也就是状态转移，是通过将 {A, ... O} 中的新指令字符串赋给指令寄存器 MEMORY['op']来实现的，如示例中的赋值字符串 op="%[N] 所示。

在预处理阶段，提示字符串从 MEMORY['op']中获取，随后将形如 @[x] 的子字符串替换为存储在 MEMORY['x']中的字符串。这样一来，便可以使用更紧凑的指令字符串 A、.....、 O，因为较长的 boot 字符串可以在预处理期间直接拼接进去。更重要的是，可以使用 @[@[i]]读取当前读写头位置的符号。要理解其工作原理，请注意，预处理器会按照第2节中的讨论，对 @[x] 模式应用嵌套替换。因此，如果我们继续假设 MEMORY['i'] = 42和 MEMORY['42'] = '1'，则第一次替换 @[@[i]]会得到 @['42'], 第二次替换会得到 '1'。也就是说，预处理完成后，子字符串 @[@[i]] 会被替换为 MEMORY[MEMORY['i']]中的值，即磁带头当前位置的符号；在本例中，该符号将是 '1'。

基于这一理解，很容易验证，每个指令字符串 A, ...,O 都正确模拟了表1中对应状态 A, ..., O 的逻辑，包括根据读写头位置的当前符号进行条件判断、向当前读写头位置写入正确的符号、沿正确方向移动读写头位置，以及通过将正确的下一条指令赋给 'op'来更新状态。

最后，为了模拟 $U_{15,2}$ 的行为，我们还必须考虑磁带存储器的初始内容。令变量 TAPE 赋值为一个字符串，该字符串覆盖 $U_{15,2}$ 初始存储器的非空白部分；根据定义，这部分必定是有限长度的。另令 i= i_0 包含磁带头的初始位置。要从此构型模拟图灵机，我们首先按如下方式初始化关联记忆 MEMORY，然后只需按照第2节中的说明调用 main() 。

```
MEMORY = {'boot':boot}
for s in 'ABCDEFGHIJKLMNO':
  MEMORY[s] = eval(s)
for loc in range(len(TAPE)):
  MEMORY[str(loc)] = TAPE[loc]
BLANK = '0'
MEMORY['i'] = i
MEMORY['op'] = A
main()
```

现在我们声称， `main()` 的每个计算周期都与图灵机 $U_{15,\ 2}$的每个计算周期保持精确等价。证明通过对计算周期数进行归纳得出。

在初始化时，以下内容彼此等价：图灵机内存带的内容与按位置编号的 `MEMORY` 内容（假设所有未分配位置均为空白）；初始带头位置 i_0 与 `i`；初始状态 A与初始指令 `A`。

接着，归纳地假设在一个计算周期开始时仍保持同样的等价关系。随后，图灵机将根据表1中的规范，按照 $(q, \sigma) \mapsto (\sigma', m, q')$ 进行更新。现在假设，经过预处理后，使用 `MEMORY[MEMORY['i']]` 中相同的当前符号（即模拟磁带头当前所在的位置）构成的对应提示字符串，在调用 `call_llm_server(prompt)` 后会返回正确的结果字符串。于是可以针对每一组（状态、符号）逐一验证：结果字符串指定了要写入当前带头位置的相同符号、要移动读写头的相同方向，以及对应的下一条指令，因此在该周期结束时，等价关系仍然成立。

为说明其中一种（状态、符号）验证情况，考虑表1中的第一项，它指定图灵机更新 $(A, 0) \mapsto (0, +, B)$。注意，假设语言模型运行正常（下文将验证这一点），在位置 `MEMORY[MEMORY['i']]` ，将指令`A` 与相同的输入符号 `'0'` 预处理后，会返回结果字符串 `" op=%[B]" %[i]="0" i+=1 "` 。在这种情况下，后处理阶段会将相应的符号`'0'` 写入当前内存位置，将读写头向右移动 `+=1`，并将 `B` 指定为下一条指令，从而保持等价性。

类似地，如果当前输入为 1，图灵机更新为 $(A,1) \mapsto (1, +, A)$。在这种情况下，注意假设语言模型运行正常，将指令 `A` 与相同的输入符号 `'1'` 在`MEMORY[MEMORY['i']]`预处理后，条件将为真，结果字符串将为 `" op=%[A]"%[i]="1" i+=1 "` 。后处理将再次保持等价性。

对于全部 29 种（状态、符号）情况，也能通过类似的验证。（请注意，状态 F 中的更新不依赖于输入，因此情况数比（状态、符号）对的总数少一个。）

现在只需验证：给定经过当前内存符号预处理的任意指令字符串，语言模型能否生成正确的结果字符串。

5 使用 Flan-U-PaLM 540B 验证执行正确性

我们现在考虑具体的语言模型 Flan-U-PaLM 540B [Chung 等人， 2022], 它是一个拥有 540B 参数的大型参数模型，并经过了额外的指令微调。为确保确定性计算，将该语言模型的解码温度设为零（纯贪心解码）。

为了完成论证，我们现在只需枚举每一种可能的（状态，符号）组合，并验证语言模型在给定由相应指令组成、并根据相应输入符号预处理的输入提示字符串时，是否会生成正确的结果字符串。这只是一个穷举证明：针对每种可能的输入提示调用语言模型，并验证它确实返回了正确的结果字符串。共有 29 种情况。（所以，嗯，虽然人类看得懂，但读起来并不愉快，抱歉。）

验证测试 1（状态 A，读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = A
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 1

```
@[boot]result = " op="%[B]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[A]" %[i]="1" i+=1 "
$result

" op="%[B]" %[i]="0" i+=1 "
```

验证测试 2（状态 A，读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = A
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 2

```
@[boot]result = " op="%[B]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[A]" %[i]="1" i+=1 "
$result

" op="%[A]" %[i]="1" i+=1 "
```

验证测试 3（状态 B 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = B
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 3

```
@[boot]result = " op="%[C]" %[i]="1" i+=1 "
if @[@[i]]==1 then result = " op="%[A]" %[i]="1" i+=1 "
$result

" op="%[C]" %[i]="1" i+=1 "
```

验证测试 4（状态 B 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = B
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 4

```
@[boot]result = " op="%[C]" %[i]="1" i+=1 "
if @[@[i]]==1 then result = " op="%[A]" %[i]="1" i+=1 "
$result

" op="%[A]" %[i]="1" i+=1 "
```

验证测试 5（状态 C 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = C
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 5

```
@[boot]result = " op="%[G]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[E]" %[i]="0" i-=1 "
$result

" op="%[G]" %[i]="0" i-=1 "
```

验证测试 6（状态 C 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = C
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 6

```
@[boot]result = " op="%[G]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[E]" %[i]="0" i-=1 "
$result

" op="%[E]" %[i]="0" i-=1 "
```

验证测试 7（状态 D 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = D
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 7

```
@[boot]result = " op="%[F]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[E]" %[i]="1" i-=1 "
$result

" op="%[F]" %[i]="0" i-=1 "
```

验证测试 8（状态 D 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = D
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 8

```
@[boot]result = " op="%[F]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[E]" %[i]="1" i-=1 "
$result

" op="%[E]" %[i]="1" i-=1 "
```

验证测试 9（状态 E 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = E
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 9

```
@[boot]result = " op="%[A]" %[i]="1" i+=1 "
if @[@[i]]==1 then result = " op="%[D]" %[i]="1" i-=1 "
$result

" op="%[A]" %[i]="1" i+=1 "
```

验证测试 10（状态 E 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = E
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 10

```
@[boot]result = " op="%[A]" %[i]="1" i+=1 "
if @[@[i]]==1 then result = " op="%[D]" %[i]="1" i-=1 "
$result

" op="%[D]" %[i]="1" i-=1 "
```

验证测试 11（状态 F 读取 $0 ==$ 状态 F 读取 1）

```
op = F
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 11

```
@[boot]result = " op="%[D]" %[i]="1" i-=1 "
$result

op="%[D]" %[i]="1" i-=1
```

验证测试 12（状态 G 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = G
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 12

```
@[boot]result = " op="%[H]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[G]" %[i]="1" i-=1 "
$result

" op="%[H]" %[i]="0" i-=1 "
```

验证测试 13（状态 G 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = G
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 13

```
@[boot]result = " op="%[H]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[G]" %[i]="1" i-=1 "
$result

" op="%[G]" %[i]="1" i-=1 "
```

验证测试 14（状态 H 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = H
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 14

```
@[boot]result = " op="%[I]" %[i]="1" i-=1 "
if @[@[i]]==1 then result = " op="%[G]" %[i]="1" i-=1 "
$result

" op="%[I]" %[i]="1" i-=1 "
```

验证测试 15（状态 H 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = H
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 15

```
@[boot]result = " op="%[I]" %[i]="1" i-=1 "
if @[@[i]]==1 then result = " op="%[G]" %[i]="1" i-=1 "
$result

" op="%[G]" %[i]="1" i-=1 "
```

验证测试 16（状态 I 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = I
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 16

```
@[boot]result = " op="%[A]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[J]" %[i]="1" i-=1 "
$result

" op="%[A]" %[i]="0" i+=1 "
```

验证测试 17（状态 I 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = I
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 17

```
@[boot]result = " op="%[A]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[J]" %[i]="1" i-=1 "
$result

" op="%[J]" %[i]="1" i-=1 "
```

验证测试 18（状态 J 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = J
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 18

```
@[boot]result = " op="%[K]" %[i]="1" i-=1 "
if @[@[i]]==1 then result = " op="halt" "
$result

" op="%[K]" %[i]="1" i-=1 "
```

验证测试 19（状态 J 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = J
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 19

```
@[boot]result = " op="%[K]" %[i]="1" i-=1 "
if @[@[i]]==1 then result = " op="halt" "
$result

" op="halt" "
```

验证测试 20（状态 K 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = K
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 20

```
@[boot]result = " op="%[L]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[N]" %[i]="1" i+=1 "
$result

" op="%[L]" %[i]="0" i+=1 "
```

验证测试 21（状态 K 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = K
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 21

```
@[boot]result = " op="%[L]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[N]" %[i]="1" i+=1 "
$result

" op="%[N]" %[i]="1" i+=1 "
```

验证测试 22（状态 L 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = L
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 22

```
@[boot]result = " op="%[M]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[L]" %[i]="1" i+=1 "
$result

" op="%[M]" %[i]="0" i+=1 "
```

验证测试 23（状态 L 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = L
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 23

```
@[boot]result = " op="%[M]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[L]" %[i]="1" i+=1 "
$result

" op="%[L]" %[i]="1" i+=1 "
```

验证测试 24（状态 M 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = M
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 24

```
@[boot]result = " op="%[B]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[L]" %[i]="1" i+=1 "
$result

" op="%[B]" %[i]="0" i-=1 "
```

验证测试 25（状态 M 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = M
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 25

```
@[boot]result = " op="%[B]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[L]" %[i]="1" i+=1 "
$result

" op="%[L]" %[i]="1" i+=1 "
```

验证测试 26（状态 N 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = N
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 26

```
@[boot]result = " op="%[C]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[O]" %[i]="0" i+=1 "
$result

" op="%[C]" %[i]="0" i-=1 "
```

验证测试 27（状态 N 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = N
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 27

```
@[boot]result = " op="%[C]" %[i]="0" i-=1 "
if @[@[i]]==1 then result = " op="%[O]" %[i]="0" i+=1 "
$result

" op="%[O]" %[i]="0" i+=1 "
```

验证测试 28（状态 O 读取 0）

```
head = MEMORY['i']
MEMORY[str(head)] = '0'
op = O
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 28

```
@[boot]result = " op="%[N]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[N]" %[i]="1" i+=1 "
$result

" op="%[N]" %[i]="0" i+=1 "
```

验证测试 29（状态 0 读取 1）

```
head = MEMORY['i']
MEMORY[str(head)] = '1'
op = 0
prompt = substitute_nested(op, '@')
result = call_llm_server(prompt)
print(op)
print(result)
```

输出 29

```
@[boot]result = " op="%[N]" %[i]="0" i+=1 "
if @[@[i]]==1 then result = " op="%[N]" %[i]="1" i+=1 "
$result

" op="%[N]" %[i]="1" i+=1 "
```

至此，证明完成。

6 讨论

希望读者至此已被说服。关于这项研究，有一些值得分享的感想。

尽管验证计算普适性很直接，但语言模型的行为却很脆弱。并非所有考察过的大语言模型都能成功，而且还需要花费精力来设计提示词。例如，指令字符串 `A、.....、 0` 以及替换符号 `@` 和 `%`，对于人类读者而言十分简短，也不太优雅；然而，要让语言模型生成正确结果，简洁性似乎至关重要。要让模型正确求值变量赋值有时颇具挑战，但迄今最大的挑战，是让语言模型正确理解条件语句。读者可能会注意到，这里的条件语句已简化为 if-then，而非 if-then-else 形式。这并非偶然：我无法让语言模型可靠地为 if-then-else 条件语句生成正确输出。条件语句带来的困难，也使得模拟其他规模更小的通用图灵机颇具挑战，例如 $U_{6,\ 4}$ [Neary 和 Woods， 2009], 因为这需要为每个状态设置一组三个条件语句，而我无法让其正常运行，除非引入虚设状态，并将指令简化为仅包含单个条件语句，最终得到的构造也就远不那么易于理解了。可以预料，底层语言模型的改进将会缓解此类挑战。

本工作的早期版本曾考虑模拟一维元胞自动机规则 110 [Wolfram, 2002], ，利用了已知该自动机（弱）图灵完备这一事实 [Cook, 2004]。尽管规则 110 在视觉上更具吸引力，但要模拟任意图灵机，它需要无界周期性初始化；相比之下，最终

本文提出的更直接的$U_{15,\ 2}$模拟只需要有界初始化，因此似乎更具说服力。

这与第2节和第4节中提出的“编程语言”，以及一些最早的编程语言之间存在有趣的类比：[B¨ohm, 1954]，包括最早的汇编语言 [Booth 和 Britten，1947]。后者尤其令人联想到前者，因为它依赖人类可读的标签来实现分支控制。值得思考的是，软件工程发展史上的其他哪些概念（例如高级语言、模块化、库等）或许有助于从大语言模型中引出预期的计算行为。

本文的结果有别于先前关于神经序列模型计算通用性的研究，例如循环神经网络[Siegelmann 和 Sontag, 2019, Weiss 等人，2018] 以及 Transformer [P´erez 等人，2019],[Bhattamishra 等人, 2020, Wei 等人, 2022a]。关键区别在于，我们考察的是一个权重冻结的固定语言模型，并展示了如何通过外部记忆增强引出通用计算行为。相比之下，先前这些研究展示了如何通过操纵神经网络的权重来实现计算通用行为，通常是利用无界（或足够高）的精度权重来编码数据结构，例如多个栈。这些先前研究的一个优点是，无需任何外部记忆即可展现通用计算行为。另一方面，据目前所知，这些结果并不适用于不改变权重的现有大语言模型。本文的结果表明，只要能够访问无界外部记忆，大语言模型以当前形态就已经具备计算通用性。

致谢

衷心感谢 Noah Fiedel、Ramki Gummadi、Andr´as Gy¨orgy、Chris Harris、Tengyu Ma、Jason Wei、Sherry Yang、Denny Zhou 和 Martin Zinkevich，他们的宝贵讨论促成了这项工作。也感谢 Google Brain 和我的团队成员为开展探索性研究提供了理想的环境。同时，谨此致谢 CIFAR 加拿大人工智能研究主席计划、加拿大自然科学与工程研究委员会和阿尔伯塔机器智能研究所提供的支持。

参考文献

[Bhattamishra 等人， 2020] Bhattamishra, S., Patel, A., and Goyal, N. (2020). 关于 Transformer 的计算能力及其对序列建模的影响。载于计算自然语言学习会议（CONLL）。

[B¨ohm, 1954] B¨ohm, C.（1954）。利用机器自身在程序设计中的逻辑数学公式解码器进行数字计算。博士论文，苏黎世联邦理工学院。

[Booth 和 Britten, 1947] Booth, A. 和 Britten, K.（1947）。通用电子数字计算机设计的一般考量。技术报告，高等研究院，普林斯顿。第2版。

[Brown 等人, 2020] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. (2020). 语言模型是少样本学习者。载于《神经信息处理系统进展》（NeurIPS）。

[Chung 等人, 2022] Chung, H. W., Hou, L., Longpre, S., Zoph, B., Tay, Y., Fedus, W., Li, Y., Wang, X., Dehghani, M., Brahma, S., Webson, A., Gu, S. S., Dai, Z., Suzgun, M., Chen, X., Chowdhery, A., Castro-Ros, A., Pellat, M., Robinson, K., Valter, D., Narang, S., Mishra, G., Yu, A., Zhao, V., Huang, Y., Dai, A., Yu, H., Petrov, S., Chi, E., Dean, J., Devlin, J., Roberts, A., Zhou, D., Le, Q., and Wei, J. (2022). 扩展指令微调语言模型。arXiv 2210.11416.[Cook, 2004] Cook, M. (2004). 初等元胞自动机中的通用性。《复杂系统》，15:1–40.[Dohan 等人, 2022] Dohan, D., Xu, W., Lewkowycz, A., Austin, J., Bieber, D., Lopes, R. G., Wu, Y., Michalewski, H., Saurous, R. A., Sohl-dickstein, J., Murphy, K., and Sutton, C. (2022). 语言模型级联。arXiv 2207.10342.[Neary, 2008] Neary, T. (2008). 小型通用图灵机。博士论文，爱尔兰国立大学梅努斯分校。[Neary 和 Woods, 2009] Neary, T. and Woods, D. (2009). 四种小型通用图灵机。《信息学基础》，91:105–126。[Ouyang 等人, 2022] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Gray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. (2022). 通过人类反馈训练语言模型遵循指令。载于《神经信息处理系统进展》（NeurIPS）。[P´erez 等人, 2019] P´erez, J., Marinkovi´c, J., and Parcel´o, P. (2019). 现代神经网络架构的图灵完备性。载于《国际学习表征会议》（ICLR）。[Radford 等人, 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). 语言模型是无监督多任务学习者。[Schulman 等人, 2022] Schulman, J., Zoph, B., Kim, C., Hilton, J., Menick, J., Weng, J., Uribe, J. F. C., Fedus, L., Metz, L., Pokorny, M., Lopes, R. G., Zhao, S., Vijayvergiya, A.,

Sigler, E., Perelman, A., Voss, C., Heaton, M., Parish, J., Cummings, D., Nayak, R., Bal- com, V., Schnurr, D., Kaftan, T., Hallacy, C., Turley, N., Deutsch, N., and Goel, V. (2022). ChatGPT：优化用于对话的语言模型。https://openai.com/blog/chatgpt.[Shannon， 1956] Shannon, C. (1956). 具有两个内部状态的通用图灵机。《自动机研究》，《数学研究年刊》，34:157–165。[Siegelmann 和 Sontag， 2019] Siegelmann, H. and Sontag, E. (2019). 神经网络的计算能力。载于《学习理论会议》（COLT）。[Sipser， 2013] Sipser, M. (2013).《计算理论导论》。Cengage Learning，第3版。[图灵， 1937] Turing, A. (1937). 论可计算数及其在判定问题中的应用。《伦敦数学会会刊》，2, 42:230–265。[冯·诺伊曼， 1945] von Neumann, J. (1945). EDVAC报告初稿。[Wei 等人，2022a] Wei, C., Chen, Y., and Ma, T. (2022a). 统计意义上的近似：以使用 Transformer 近似图灵机为例。载于《神经信息处理系统进展》（NeurIPS）。[Wei 等人，2022b] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2022b). 思维链提示可激发大语言模型的推理能力。载于《神经信息处理系统进展》（NeurIPS）。[Weiss 等人， 2018] Weiss, G., Goldberg, Y., and Yahav, E. (2018). 有限精度循环神经网络在语言识别方面的实际计算能力。载于《计算语言学协会会议》（ACL）。[Wolfram， 2002] Wolfram, S. (2002).《一种新科学》。Wolfram Media。[Zhou 等人， 2022] Zhou, D., Sch¨arli, N., How, L., Wei, J., Scales, N., Wang, X., Schuur- mans, D., Cui, C., Bousquet, O., Le, Q., and Chi, E. (2022). 由简到繁提示有助于大语言模型进行复杂推理。arXiv 2205.10625。