

Memory Augmented Large Language Models are Computationally Universal

Dale Schuurmans
Google Brain & University of Alberta

Abstract

We show that transformer-based large language models are computationally universal when augmented with an external memory. Any deterministic language model that conditions on strings of bounded length is equivalent to a finite automaton, hence computationally limited. However, augmenting such models with a read-write memory creates the possibility of processing arbitrarily large inputs and, potentially, simulating any algorithm. We establish that an existing large language model, Flan-U-PaLM 540B, can be combined with an associative read-write memory to exactly simulate the execution of a universal Turing machine, $U_{15,2}$. A key aspect of the finding is that it does not require any modification of the language model weights. Instead, the construction relies solely on designing a form of stored instruction computer that can subsequently be programmed with a specific set of prompts.

1 Introduction

Interest in large language models has grown dramatically since the early successes of GPT-2, GPT-3 and InstructGPT [Radford et al., 2019, Brown et al., 2020, Ouyang et al., 2022], and more recently with the popularity of ChatGPT [Schulman et al., 2022]. Beyond simple question answering, where an input string posing a question might elicit an output string containing a reasonable answer, an important discovery has been the emergence of *in-context learning*, where prepending a question with a set of related (question, answer) pairs significantly improves question answering accuracy [Radford et al., 2019]. Even adding a natural language instruction before example pairs appears to further enhance language model capabilities [Brown et al., 2020]. More recently, *chain of thought prompting* has been found to improve question answering ability in scenarios where multiple reasoning steps are required to arrive at a final answer, such as answering math word problems [Wei et al., 2022b].

Despite these results, current transformer-based large language models remain fundamentally limited as they can only condition on an input string of bounded length, such as 4096 tokens. This makes such models formally equivalent to finite automata, hence restricted in the computations they can express. However, recent works have begun to investigate techniques for chaining multiple calls to a language model by processing model outputs then

passing these back as subsequent inputs to the model. An example is *least to most prompting*, where a complex reasoning question is answered first by prompting the model to produce simpler sub-questions, then passing each sub-question and resulting answer back into the model to help answer subsequent sub-questions, until a final answer is reached [Zhou et al., 2022]. Another example is work on *language model cascades* that investigates various strategies for processing model outputs and feeding these as inputs to subsequent language model calls [Dohan et al., 2022]. Such works raise the question of whether augmenting a language model with an external feedback loop is merely useful, or fundamentally expands the range of computations that can be performed. To investigate this question, we consider augmenting a language model with an external read-write memory and ask whether this confers the ability to simulate any algorithm on any input.

This paper gives an affirmative answer by establishing computational universality for a specific large language model, Flan-U-PaLM 540B [Chung et al., 2022], augmented with an associative read-write memory. A key aspect of the result is that it is achieved by developing a simple form of *stored instruction computer* [von Neumann, 1945] that connects the language model to an associative memory, then follows a simple instruction cycle where the next input prompt to be passed to the language model is retrieved from memory, the output of the language model is parsed to recover any variable assignments that are then stored in the associative memory, and the next instruction is retrieved (i.e., the next input prompt to be provided to the language model). Each parsing step between the language model and memory is performed by a simple regular expression match (i.e., a finite automaton).

Once a stored instruction computer has been created, a specific “prompt program” is designed to drive the system to simulate a universal Turing machine $U_{15,2}$ [Neary, 2008, Neary and Woods, 2009]. Proving the fidelity of the simulation reduces to checking a finite set of prompt-result behaviours and verifying that the language model produces the correct output for each of the finite set of possible input prompt strings it might encounter. That is, although the overall input-output behaviour of the Flan-U-PaLM 540B model is not fully understood, a sufficiently reliable subset of its input-output map can be isolated and controlled to simulate a universal computer. Importantly, this result does not involve any additional “training” of the language model (i.e., no modification of its pre-trained weights), but instead relies solely on providing specific prompt strings to the model and parsing its outputs to determine values to be saved in memory.

2 Stored instruction computer

As noted, there are many ways to orchestrate feedback between the outputs of a language model and subsequent input prompts [Zhou et al., 2022, Dohan et al., 2022]. In developing a simple feedback loop we would like to minimize external processing and perform as much of the computation with the language model as possible, while still supporting computational universality. To achieve this, we consider a simple form of *stored instruction computer* [von Neumann, 1945], where the language model plays the role of a central processing unit (CPU), and the random access memory (RAM) is supplied by an external associative mem-

ory. Such an architecture allows for a simple interaction loop that can support general computation and convenient programmability. In this architecture, the external associative memory is a simple “dictionary”, `MEMORY`, that maps unique keys to values, or equivalently, maps variable names to values, or address locations to values. Unlike the RAM in a physical computer, variable names will be strings (i.e., finite length sequences of symbols from a finite alphabet) to support convenient interaction with a language model, while values will be strings or integers.

To ensure that computational universality does not follow simply from external processing capability, all interaction between the language model and the memory will be restricted to *finite state computation*, such as simple regular expression parsers.

All code below will be given in Python 3 using the standard regular expression library `re`. Note that the regular expressions used for pre and post processing are rudimentary and can easily be improved in several ways; the versions given are merely sufficient to establish the main points in this paper.

2.1 Post processing language model outputs

The output string from the language model will be parsed by a simple regular expression that detects assignments in the form `variable_name = "value"`, which are then applied to the associative memory as `MEMORY[variable_name] = "value"`. The variable assignment function `assignments` is shown below.

```
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
```

Additionally, splicing is allowed in value strings before being assigned to memory; that is, we include a regular expression parser that detects occurrences of the pattern `%[variable_name]` in any value string, replacing any such occurrence with the string at memory location `variable_name`, i.e., `MEMORY[variable_name]`, before assignment. The substitution function `substitute` is shown below.

```

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 += suffix
    return string

```

As a final convenience, we also allow integer values to be stored and incremented or decremented. Such integer variable updating is achieved by parsing the output string for occurrences of the pattern `variable_name += increment` or `variable_name -= decrement` then applying the updates to `variable_name` in memory. Importantly, integer addition is also a finite state operation (see, for example, [Sipser, 2013, Problem 1.32]), while the Python standard implements the `bignum` type that handles arbitrarily large integers. The update function updates is shown below.

```

def updates(string):
    global MEMORY
    regex = '(\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 -1
            value = int(valuestring) * sign
            if label in MEMORY and isinstance(MEMORY[label], int):
                MEMORY[label] += value
            else:
                MEMORY[label] = value # new or non-int starts from 0 by default

```

2.2 Pre processing language model inputs

Each input prompt to the language model will be retrieved from a special memory location `op` and passed as a prompt to the language model in each computational cycle; that is, `MEMORY['op']` will serve as an “instruction register”. Instruction branching can then be achieved simply by assigning a different prompt string to `MEMORY['op']` during a computational cycle.

To access stored memory values, we also allow splicing in the input prompt string retrieved from `op`. In particular, the regular expression parser for the input prompt first detects patterns of the form `@[variable_name]` and replaces these by splicing in the string retrieved from `MEMORY[variable_name]` before passing the prompt string to the language model. For input pre processing, it will also be convenient to allow repeated substitutions of nested `@` occurrences, so we add the repeated substitution function `substitute_nested` below. Note that, technically, allowing arbitrarily nested substitutions can simulate a context free grammar [Sipser, 2013], which violates the constraint of finite state computation; however, we will only use bounded depth nesting (depth bound 2) below to ensure the pre and post processing steps all remain achievable by finite state computation.

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

2.3 Compute cycle

Finally, a stored instruction computer can run a single compute cycle: retrieve the next prompt string from `MEMORY['op']`; process the prompt string by possibly splicing in other strings from memory; pass the prompt string to the language model; process the output string, possibly splicing in other strings from memory, detecting all assignments and increment/decrement updates, applying these to memory; and repeat. Computation proceeds until the next instruction in `MEMORY['op']` is the special instruction string `'halt'`. In particular, the main loop is constructed as follows.

```
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)
```

This main instruction loop demonstrates how the language model plays the role of the CPU, effectively taking the next instruction and its operands expressed by the prompt string and acting on these to produce the result string, which is then used to update the memory.

3 Universal Turing machine

The concept of a universal computer—a computing machine that can simulate the execution of any other computing machine on any input—was developed by Alan Turing to solve the *Entscheidungsproblem* [Turing, 1937]. By the Church-Turing thesis, all computational mechanisms are considered to be expressible by a *Turing machine*, which informally consists of a finite state controller and an unboundedly large “tape” memory with a “head” that can access a single tape location and move one location left or right in each compute cycle [Sipser, 2013, Chapter 3].

Formally, a Turing machine consists of a tuple $\mathcal{M} = (Q, \Sigma, b, q_0, T, f)$, where Q is a finite set of states, Σ is a finite set of tape symbols, $b \in \Sigma$ is the blank symbol, $q_0 \in Q$ is the start state, $T \subseteq Q \times \Sigma$ is the set of halting (state, symbol) pairs, and $f : Q \times \Sigma \rightarrow \Sigma \times \{-1, +1\} \times Q$ is a finite set of transition rules that specify the operation of the machine in each compute cycle. We assume the tape is bi-directionally unbounded, so memory locations can be indexed by an integer $i \in \mathbb{Z}$. Let $i_0 \in \mathbb{Z}$ denote the initial location of the tape head.

The execution of a Turing machine can then be defined as follows. The tape memory is initialized with a finite number of non-blank symbols with all other locations blank, \mathcal{M} starts in state q_0 , and the tape head starts at location i_0 . At the start of each compute cycle, the tape head is at some location $i \in \mathbb{Z}$, the machine is in some state $q \in Q$, and some symbol $\sigma \in \Sigma$ is under the tape head. This combination determines the update $f(q, \sigma) \mapsto (\sigma', m, q')$, specifying that the symbol σ' is written at the current memory location i , the machine state q is updated to q' , and the tape head is moved one step left, to location $i' = i - 1$ if $m = -1$, otherwise one step right, to location $i' = i + 1$ if $m = +1$. The compute cycle repeats until the machine encounters a configuration $(q, \sigma) \in T$. Non-halting computations are possible.

[Shannon, 1956] began an effort to identify the smallest universal Turing machines in terms of the number of states and tape symbols used. A gap remains between the known upper and lower bounds on the state and symbol counts for a universal Turing machine [Neary, 2008, Neary and Woods, 2009], but progressively smaller universal Turing machines have been identified. We will consider one such machine in this paper, $U_{15,2}$, which uses only 15 states and 2 tape symbols [Neary and Woods, 2009]. This Turing machine is Pareto optimal in terms of the smallest known universal Turing machines [Neary, 2008]. Formally, the Turing machine $U_{15,2}$ can be defined by a tuple $(Q, \Sigma, b, q_0, T, f)$, where $Q = \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O\}$, $\Sigma = \{0, 1\}$, $b = 0$, $q_0 = A$, $T = \{(J, 1)\}$, and the transition function f is defined in Table 1. The initial head position i_0 depends on how the memory is initialized for a given problem instance.

4 Simulating $U_{15,2}$ with a prompt program

In this section, we show that the stored instruction computer defined in Section 2 can be programmed to simulate the universal Turing machine $U_{15,2}$, provided that a finite set of conditional assignments and evaluations can be correctly performed by the language model. That is, we first propose a specific prompt program that, if executed correctly, exactly

	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	

Table 1: *Transition table for the universal Turing machine $U_{15,2}$. Rows are indexed by the read symbol σ , columns are indexed by the state q , and each table entry (σ', m, q') specifies the write symbol σ' , the tape head move $m \in \{-1, +1\}$, and the next state q' .*

simulates $U_{15,2}$. The next section will then verify that a specific large language model, Flan-U-PaLM 540B, is indeed able to execute each of the program instructions correctly.

A prompt program consists of a finite set of pre designed strings stored in memory that provide input prompts to the language model as part of the main compute cycle, via a call to `main()` outlined in Section 2. To mimic the behaviour of $U_{15,2}$, we design the prompt program as follows. First, a “boot” prompt is designed that “instructs” the language model about the behaviour of variable assignments, variable evaluations after assignment, and if-then conditionals.

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

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

result = " op=%[C] " %[i]="1" i+=1 "
if 0==1 then result = " op=%[A] " %[i]="1" i+=1 "
$result
" op=%[C] " %[i]="1" i+=1 "

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

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

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

```

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

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

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

result = " op="%[K]" %[i]="1" i--=1 "
if 0==1 then result = " op="halt" "
$result
" op="%[K]" %[i]="1" i--=1 "

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

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

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

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

""

```

Next, a series of “instruction” prompts are defined. Each of these strings is intended to express the logic of a corresponding Turing machine state in $U_{15,2}$ specified in Table 1.

```

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

```



```

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

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

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

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

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

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

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

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

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

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

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

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

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

```

It helps to understand how this prompt program is intended to work. First, note that the memory location 'i' is intended to keep track of the current location of the Turing machine head, so that any update $i-=1$ will correspond to moving the head one step left, and $i+=1$ will correspond to moving the head one step right. Next, consider the post processing of one of the result strings, for example " op=%[N]" %[i]="1" i+=1 ". In this string, the expression %[i]="1" is intended to write the target symbol '1' to the memory location indexed by MEMORY['i']. That is, during post processing, any substring of the form %[x] will first be replaced by the string in MEMORY['x'] before performing any assignments, as explained in Section 2. Thus, given %[i]="1" the substring %[i] will first be replaced by the value in MEMORY['i'], say ℓ , which then serves as the label in memory to be assigned the value '1'. For example, if we assume MEMORY['i'] = 42 and MEMORY['42'] = '0', then after post processing and assignment we will have MEMORY['42'] = '1'. Control branching, i.e., a state transition, is achieved by assigning a new instruction string from {A, ..., 0} to the instruction register MEMORY['op'], as specified by the assignment string op=" %[N]" in the example.

In the pre processing phase, the prompt string is obtained from MEMORY['op'], then substrings of the form @[x] are replaced by the string stored in MEMORY['x']. This allows for more compact instruction strings A, ..., 0, since the lengthy boot string can just be spliced in during pre processing. More importantly, the symbol at the current head position can be read with @[@[i]]. To see why this works, note that the preprocessor will apply *nested* substitutions of the @[x] patterns, as discussed in Section 2. Therefore, if we continue to assume that MEMORY['i'] = 42 and MEMORY['42'] = '1', the first substitution of @[@[i]] will result in @['42'], and the second substitution will result in '1'. That is, after pre processing, the substring @[@[i]] is replaced with the value found in MEMORY[MEMORY['i']], i.e., the symbol at the current position of the tape head, which in this case will be '1'.

Given this understanding, it is easy to verify that each of the instruction strings A, ..., 0 correctly mimics the logic of the corresponding states A, \dots, O in Table 1, including conditioning on the current symbol in the head position, writing the correct symbol to the current head position, moving the head position in the correct direction, and updating the state by assigning the correct next instruction to 'op'.

Finally, to simulate the behaviour of $U_{15,2}$, we also have to consider the initial contents of the tape memory. Let the variable TAPE be assigned to a string that covers the non-blank portion of the initial memory for $U_{15,2}$, which must be finitely long by definition. Also let $i = i_0$ contain the initial position of the tape head. To simulate the Turing machine from this configuration, we first initialize the associative memory, MEMORY, as follows, then simply call main() as specified in Section 2.

```

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

```

We now claim that each compute cycle of `main()` maintains an exact equivalence with each compute cycle of the Turing machine $U_{15,2}$. The proof is by induction on the number of compute cycles.

At initialization, there is an equivalence between: the contents of the Turing machine memory tape and the contents of `MEMORY` labelled by location numbers (with all unassigned locations assumed to be blank); the initial tape head location i_0 and `i`; the initial state A and initial instruction `A`.

Then, inductively, assume the same equivalence holds at the onset of a compute cycle. The Turing machine will then be updated according to $(q, \sigma) \mapsto (\sigma', m, q')$ following the specification in Table 1. Now assume the corresponding prompt string after pre processing with the same current symbol in `MEMORY[MEMORY['i']]` (i.e., the current location of the simulated tape head) returns the correct result string after the call to `call_llm_server(prompt)`. Then one can verify, on a case by case basis for each (state, symbol) pair, that the result string specifies the same symbol to be written to the current head location, specifies the same direction to move the head, and specifies the corresponding next instruction, thus the equivalence is maintained at the end of the cycle.

To illustrate one of the (state, symbol) verifications, consider the first entry in Table 1, which specifies the Turing machine update $(A, 0) \mapsto (0, +, B)$. Observe that the instruction `A` pre processed with the same input symbol `'0'` at `MEMORY[MEMORY['i']]` will return the result string `" op=%[B]" %[i]="0" i+=1 "` assuming the language model operates correctly (verified below). In this case, the post processing phase will write the corresponding symbol `'0'` to the current memory location, move the head right `+=1`, and assign `B` to be the next instruction, thus maintaining the equivalence.

Similarly, if the current input is `1`, the Turing machine update is $(A, 1) \mapsto (1, +, A)$. In this case, observe that if the instruction `A` pre processed with the same input symbol `'1'` at `MEMORY[MEMORY['i']]`, the condition will be true and the result string will be `" op=%[A]" %[i]="1" i+=1 "` assuming the language model operates correctly. Post processing will once again maintain the equivalence.

A similar verification succeeds for all 29 (state, symbol) cases. (Note that the update in state F does not depend on the input, so there is one fewer case than the total number of (state, symbol) pairs.)

It remains only to verify that a language model can produce the correct result string given any of the instruction strings after pre processing with the current memory symbol.

5 Verifying correct execution using Flan-U-PaLM 540B

We now consider the specific language model Flan-U-PaLM 540B [Chung et al., 2022], which is a large 540B parameter model that has been refined with additional instruction fine-tuning. To ensure deterministic computation, the decoding temperature for the language model is set to zero (pure greedy decoding).

To complete the argument, we now simply enumerate each of the possible (state, symbol) combinations and verify that the language model produces the correct result string given an input prompt string composed from the corresponding instruction and pre processed with the corresponding input symbol. This is simply a brute force proof, calling the language model with each possible input prompt and verifying that the correct result string is indeed returned. There are 29 cases. (So, yeah, human readable but not human enjoyable, apologies.)

Verification test 1 (state A read 0)

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

Output 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 "
```

Verification test 2 (state A read 1)

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

Output 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 "
```

Verification test 3 (state B read 0)

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

Output 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 "
```

Verification test 4 (state B read 1)

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

Output 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 "
```

Verification test 5 (state C read 0)

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

Output 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 "
```

Verification test 6 (state C read 1)

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

Output 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 "
```

Verification test 7 (state D read 0)

```

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

```

Output 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 "
```

Verification test 8 (state D read 1)

```

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

```

Output 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 "
```

Verification test 9 (state E read 0)

```

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

```

Output 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 "
```

Verification test 10 (state E read 1)

```

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

```

Output 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 "
```

Verification test 11 (state F read 0 == state F read 1)

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

Output 11

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

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

Verification test 12 (state G read 0)

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

Output 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 "
```

Verification test 13 (state G read 1)

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

Output 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 "
```

Verification test 14 (state H read 0)

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

Output 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 "
```

Verification test 15 (state H read 1)

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

Output 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 "
```

Verification test 16 (state I read 0)

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

Output 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 "
```

Verification test 17 (state I read 1)

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

Output 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 "
```

Verification test 18 (state J read 0)


```

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

```

Output 18

```

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

```

```

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

```

Verification test 19 (state J read 1)

```

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

```

Output 19

```

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

```

```

" op="halt" "

```

Verification test 20 (state K read 0)

```

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

```

Output 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 "

```

Verification test 21 (state K read 1)

```

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

```

Output 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 "

```

Verification test 22 (state L read 0)

```

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

```

Output 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 "

```

Verification test 23 (state L read 1)

```

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

```

Output 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 "

```

Verification test 24 (state M read 0)

```

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

```

Output 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 "

```

Verification test 25 (state M read 1)

```

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

```

Output 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 "

```

Verification test 26 (state N read 0)

```

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

```

Output 26

```

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

```

```

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

```

Verification test 27 (state N read 1)

```

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

```

Output 27

```

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

```

```

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

```

Verification test 28 (state O read 0)

```

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

```

Output 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 "

```

Verification test 29 (state O read 1)

```

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

```

Output 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 "

```

This completes the proof.

6 Discussion

Hopefully the reader has been convinced by this point. There are some reflections on this study that are useful to share.

Although the verification of computational universality is straightforward, the language model's behaviour was brittle. Success was not achieved with every large language model considered, and effort was required to engineer the prompts. For example, the instruction strings A, ..., 0 with the substitution symbols @ and % are terse and not particularly elegant for humans to read, yet compactness seemed essential to get the language model to produce correct results. Eliciting correct evaluations of variable assignments was sometimes a challenge, but by far the biggest challenge was getting the language model to interpret conditionals properly. The reader might notice that the conditionals have been reduced to if-then rather than if-then-else forms. This was not an accident: I was not able to get the language model to reliably produce correct outputs for if-then-else conditionals. The difficulty with conditionals also makes it challenging to simulate other, smaller universal Turing machines, such as $U_{6,4}$ [Neary and Woods, 2009], since this requires a series of 3 conditionals for each state, which I could not get to work without introducing phantom states and reducing instructions to only a single conditional, ultimately ending up with a far less digestible construction. Presumably improvements in the underlying language models will mitigate such challenges.

Earlier versions of this work considered simulating Rule 110 for a one dimensional cellular automaton [Wolfram, 2002], leveraging the fact that this is known to be a (weakly) Turing complete [Cook, 2004]. Although far more visually appealing, Rule 110 requires an unbounded periodic initialization to an simulate arbitrary Turing machine, and ultimately

the more direct simulation of $U_{15,2}$ presented in this paper, which requires only a bounded initialization, appears to be more convincing.

There is an interesting analogy to the “programming language” developed in Sections 2 and 4 and some of the earliest programming languages [Böhm, 1954], including the first assembly languages [Booth and Britten, 1947]. The latter is particularly reminiscent given the reliance on human readable labels for branch control. It is interesting to speculate about what other concepts in the history of software engineering (e.g., high level languages, modularity, libraries, etc.) might be useful for eliciting desired computational behaviour from a large language model.

The result in this paper is distinct from previous studies that investigate the computational universality of neural sequence models, such as recurrent neural networks [Siegelmann and Sontag, 2019, Weiss et al., 2018] and Transformers [Pérez et al., 2019], [Bhattachamishra et al., 2020, Wei et al., 2022a]. The key distinction is that we consider a *fixed* language model with frozen weights, and show how external memory augmentation can elicit universal computational behaviour. By contrast these past studies have shown how computationally universal behaviour can be recovered by *manipulating* the weights of the neural network, typically using unbounded (or sufficiently high) precision weights to encode data structures, like multiple stacks. An advantage of these past works is that they do not require any external memory to demonstrate universal computational behaviour. On the other hand, these results do not apply to existing large language models without altering their weights (as far as currently known). The results in this paper show that large language models are already computationally universal—as they exist currently—provided only that they have access to an unbounded external memory.

Acknowledgments

Sincere thanks to Noah Fiedel, Ramki Gummadi, András György, Chris Harris, Tengyu Ma, Jason Wei, Sherry Yang, Denny Zhou and Martin Zinkevich for essential discussions leading to this work. Thanks also to Google Brain and my team members for providing an ideal environment for conducting exploratory research. Support from the CIFAR Canada AI Research Chairs program, NSERC, and Amii is also gratefully acknowledged.

References

- [Bhattachamishra et al., 2020] Bhattachamishra, S., Patel, A., and Goyal, N. (2020). On the computational power of Transformers and its implications in sequence modeling. In *Conference on Computational Natural Language Learning (CONLL)*.
- [Böhm, 1954] Böhm, C. (1954). *Calculatrices digitales du déchiffrement de formules logico-mathématiques par la machine même dans la conception du programme*. PhD thesis, ETH Zürich.

- [Booth and Britten, 1947] Booth, A. and Britten, K. (1947). General considerations in the design of an all purpose electronic digital computer. Technical report, Institute for Advanced Study, Princeton. 2nd ed.
- [Brown et al., 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). Language models are few-shot learners. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [Chung et al., 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). Scaling instruction-finetuned language models. arXiv 2210.11416.
- [Cook, 2004] Cook, M. (2004). Universality in elementary cellular automata. *Complex Systems*, 15:1–40.
- [Dohan et al., 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). Language model cascades. arXiv 2207.10342.
- [Neary, 2008] Neary, T. (2008). *Small universal Turing machines*. PhD thesis, National University of Ireland, Maynooth.
- [Neary and Woods, 2009] Neary, T. and Woods, D. (2009). Four small universal Turing machines. *Fundamenta Informaticae*, 91:105–126.
- [Ouyang et al., 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). Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [Pérez et al., 2019] Pérez, J., Marinković, J., and Parceló, P. (2019). On the Turing completeness of modern neural network architectures. In *International Conference on Learning Representations (ICLR)*.
- [Radford et al., 2019] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners.
- [Schulman et al., 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., Balcom, V., Schnurr, D., Kaftan, T., Hallacy, C., Turley, N., Deutsch, N., and Goel, V. (2022). ChatGPT: Optimizing language models for dialogue. <https://openai.com/blog/chatgpt>.
- [Shannon, 1956] Shannon, C. (1956). A universal Turing machine with two internal states. *Automata Studies, Annals of Mathematics Studies*, 34:157–165.
- [Siegelmann and Sontag, 2019] Siegelmann, H. and Sontag, E. (2019). On the computational power of neural nets. In *Conference on Learning Theory (COLT)*.
- [Sipser, 2013] Sipser, M. (2013). *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition.
- [Turing, 1937] Turing, A. (1937). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2, 42:230–265.
- [von Neumann, 1945] von Neumann, J. (1945). First draft of a report on the EDVAC.
- [Wei et al., 2022a] Wei, C., Chen, Y., and Ma, T. (2022a). Statistically meaningful approximation: a case study on approximating Turing machines with Transformers. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [Wei et al., 2022b] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2022b). Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems (NeurIPS)*.
- [Weiss et al., 2018] Weiss, G., Goldberg, Y., and Yahav, E. (2018). On the practical computational power of finite precision RNNs for language recognition. In *Conference of the Association for Computational Linguistics (ACL)*.
- [Wolfram, 2002] Wolfram, S. (2002). *A New Kind of Science*. Wolfram Media.
- [Zhou et al., 2022] Zhou, D., Schärli, N., How, L., Wei, J., Scales, N., Wang, X., Schuurmans, D., Cui, C., Bousquet, O., Le, Q., and Chi, E. (2022). Least-to-most prompting enables complex reasoning in large language models. arXiv 2205.10625.